Язык программирования Тривиль

Версия языка: 0.9.4

Алексей Недоря

30.06.2024

Содержание

1. Назначение

Язык программирования Тривиль разработан в рамках работы на семейством языков программирования Языки выходного дня (ЯВД) проекта Интенсивное программирования.

Тривиль является предварительным языком семейства ЯВД, предназначенным для реализации компиляторов и экосистемы следующих языков семейства. В рамках классификации языков, принятом в проекте Интенсивное программирования, 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 поведение, 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 файлов.

Пример программы:

```
1  модуль x
2
3  импорт "стд/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), ключевых 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 опционально завершающим знаком препинания:

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

```
Идентификатор: Слово ((' ' | '-') Слово)* Знак-препинания?
Слово: Буква (Буква | Цифра)*
Буква: Unicode-letter | '_' | '№'
Цифра: '0' .. '9'
Знак-препинания: '?' | '!'
```

Примеры идентификаторов:

```
буква
буква-или-цифра
№-символа
Цифра?
Пора паниковать!
```

3.4. Ключевые слова

Следующие ключевые 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, заключенные в двойные кавычки. Строковый литерал может содержать символы, закодированные с помощью escape-последовательности, которая начинается с символа `'\'`. Строковый литерал должен целиком находиться на одной строке текста.

```
Строковый-литерал
    : '"'
    (~('"' | '\\' | '\n' | '\r' | '\t') | Escape)*
    '"'
Escape
    : '\\'
    ( 'u' Цифра16 Цифра16 Цифра16 Цифра16
    | 'n' | 'r' | 't'
    | '"'
    | "'"
    )
```

Тип строкового литерала: `Строка` (§5.1).

3.6.4. Многострочные литералы

Многострочный литерал - это последовательность символов, заключенные в обратные кавычки: `` `Привет` ``. Последовательность может содержать любой символ, кроме обратной кавычки.

```
Многострочный-литерал
    : '`' (~('`' )* '`'
```

В многострочном литерале escape-последовательности не обрабатываются. В частности, обратная косая черта не имеет выделенного значения и строка может содержать символ конца строка (`\n`). Последовательность `\r\n` заменяется на `\n`, а одиночный символ `\r` игнорируется (не попадают в литерал).

```
1  `это длинный
2  многострочный литерал,
3  содержащий символы конца строки`
```

Тип многострочного литерала: `Строка` (§5.1).

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 escape-последовательности, которая начинается с символа `'\'`.

```
Символьный-литерал
    : "'"
    ~("'" | '\\' | '\n' | '\r' | '\t') | escape_value)
    "'"
```

Тип символьного литерала: `Символ` (§5.1).

3.7. Модификаторы

Модификаторы используются в исходном тексте, чтобы внести изменение в 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 теле функции (§4.5) или теле входа (§9.3), является часть *блока* (§7.1), в котором описан идентификатор, от точки завершения описания и до завершения этого блока.

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 (§6.13).

```
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 константы:

```
1  конст (
2      / / операции
3      ПЛЮС = 1    // тип Цел64, значение = 1
4      МИНУС       // тип Цел64, значение = 2
5      ОСТАТОК     // тип Цел64, значение = 3
6      // ключевые 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: (':=' | '=') ('позже' | Выражение)
```

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

Каждая переменная должна быть явно проинициализирована. 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*.

Явное указание типа:

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

Неявное указание типа:

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

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

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

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

Вместо выражения может быть указано ключевое слово **позже**, это *поздняя инициализация*. Такая форма инициализации разрешена только для переменных уровня модуля, при этом тип таких переменных, должен быть явно указан.

Значение переменной с поздней инициализацией должно быть задано в инициализации модуля (§9.3). Текущая реализация компилятора запрещает позднюю инициализацию для переменных с единственным присваиванием.

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).

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

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.

```
фн извлечь число(стр: Строка, число:= Цел64): Лог {...}

пусть рез := 0
если ~ извлечь число(с, рез) {
    вывод.ф("не удалось извлечь число")
}
```

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`.

```
фн сумма(числа: ...Цел64): Цел64 {...}

// В вызове можно указать любое число аргументов
ответ := сумма() // без аргументов
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).

```
фн печать по формату(формат: Строка, аргументы: ...*) {
...
}
```

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.

```
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.

```
тип К = класс {}

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 переопределеного 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):

```
тип К1 = класс (К) {}

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`
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 (§6.10, §6.12).

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:

```
пусть с8 = "Привет"(:Строка8)
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).

Для явного указании типа используется имя типа, перед которым может стоять ключевое слово **мб** (*может быть*).

```
Указ-типа: 'мб'? Имя-типа
Имя-типа: Идентификатор | Импорт-идент
Импорт-идент: Идентификатор '.' Идентификатор
```

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

Множество значений объекта с типом **мб** T состоит из значения, обозначенного предопределенным идентификатором **пусто** и значений типа T. Тип такого объекта называется *может быть T* (§5.6).

5.3. Описание типов

Описание типа связывает идентификатор с новым или существующим типом.

```
Описание-типа:
    'тип' Идент-оп '='
    ( Тип-вектора
    | Тип-класса
    | Тип-протокола
    | Тип-функции
    | Указ-типа
    )
```

Если в описание типа указан тип вектора (§5.4) или тип класса (§5.5), 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).

```
тип Цел = Цел64
тип Текст= []Строка
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
```

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

Длина вектора может изменятся 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:

```
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
```

```
тип Человек = класс {
    имя: Строка := ""
    возраст: Цел64 := 0
}
```

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 значение 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 ключевое 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 называется *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.

```
тип Работник = класс (Человек) {
    зарплата := 0.0
}
```

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 использовать 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 ключевое 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 применятся.

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* (§6.12.3)
- Операция подтверждения типа `'^'` (§6.9), позволяющая перейти от значения типа *мб T* к значению типа *T*

Пример указания типа и использования объекта:

```
пусть кличка: мб Строка := пусто
...
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 *К*, 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 *К реализует* протокол *П*, как в этом примере:

```
тип П = протокол { фн мин(а: Цел64, б: Цел64): Цел64 }
тип К = класс {}

фн (к: К) мин(а: Цел64, б: Цел64): Цел64 {
    надо а <= б иначе вернуть б
    вернуть а
}

вход{
    пусть п: П = К{}
}
```

Так как в описании класса, реализующего протокол, нет упоминания типа протокола, обычно говорят, что при проверки 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 (§6.11) или преобразование типа (§6.5).

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

```
тип П = протокол { фн мин(а: Цел64, б: Цел64): Цел64 }
тип К = класс {}
тип К1 = класс (К) {}

фн (к: К1) мин(а: Цел64, б: Цел64): Цел64 {
    надо а <= б иначе вернуть б
    вернуть а
}

вход{
    пусть к: К := К1{}
    пусть п: П := к
}
```

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:

```
тип К = класс {}
фн (к: К) сообщить(ошибка: Строка) {}

тип П = протокол { фн сообщить(ошибка: Цел64) }

вход{
    пусть п: П := К{} // несовместимость типов
}
```

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).

```
тип Ввод-Вывод = протокол {
    фн прочитать(): Символ
    фн записать(с: Символ)
}

фн прочитать строку(вв: Ввод-Вывод): Строка {
    пусть сб = строки.Сборщик{}
    пока истина {
        пусть с = вв.прочитать()
        надо с # '\n' иначе вернуть сб.строка()
        сб.добавить символ(с)
    }
    авария("")
}
```

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 сигнатурой.

Пример корректного использования:

```
тип Сообщить = фн (с: Строка)

фн записать(с: Строка) {}

тип Консоль = класс {}
фн (к: Консоль) напечатать(с: Строка) {}

вход{
    пусть сообщить: Сообщить := записать
    сообщить("Привет")
    сообщить := Консоль{}.напечатать
    сообщить("Еще привет")
}
```

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
    | Выражение Бинарная-операция Выражение

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

Проверка-типа: Выражение 'типа' Указ-типа
```

6.1. Первичные выражения

Первичные выражения - это выражения, которые являются операндами унарных (§6.10) и бинарных (§6.12) операций .

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

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

6.2. Операнды

Операнды - это элементарные значения в выражении. Операнд может быть литералом, именем объекта, обозначающим константу, переменную или функцию, или выражением в скобках.

```
Операнд: Литерал | Имя-объекта | '(' Выражение ')'
Имя-объекта: Идентификатор | Импорт-идент
Импорт-идент: Идентификатор '.' Идентификатор
```

Если в качестве имени объекта используется *Импорт-идент*, то первый идентификатор в нем должен обозначать имя импортированного модуля (§9.2), а второй идентификатор должен быть идентификатором экспортированного из этого модуля объекта.

Операнд	**Тип выражения**
Имя объекта	тип объекта
Целочисленный-литерал	Цел64
Вещественный-литерал	Вещ64
Строковый-литерал	Строка
Символьный-литерал	Символ
Выражение в скобках	тип выражения

6.3. Доступ к полям и методам

Доступ применяется к первичному выражению.

```
Доступ: '.' Идентификатор
```

В выражении `что-то.имя`:

- `что-то` должно быть выражением типа *T*, где *T* - это тип класса.
- а `имя` должно быть идентификатором поля или метода типа *T* или базового класса *T*

Типом выражения доступа является тип 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 ключевое 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:

```
тип Байты = []Байт
пусть байты = "Привет"(:Байты)

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	*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
произвольный тип *Т*	полиморфный тип (*)	полиморфное значение содержит значение типа *Т*

TBD: разделить статическую и динамическую семантику.

Преобразование полиморфного типа к базовому:

```
фн хочу целое(п: *) {
    пусть к = п(:Цел64)
}
```

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 исключение 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.

```
фн Факториал(ц: Цел64): Цел64 { /*тело*/ }

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 выполняется 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:

```
фн Соединить(список: ...Строка) {/*тело*/}

тип Строки = []Строка

фн Сохранить(строки: Строки) {
    пусть текст = Соединить(строки...)
    /* сохранение текста */
}
```

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.

```
тип Числа = []Цел64

пусть ч = Число[] // вектор длины 0
пусть ч1 = Число[1, 2, 3] // вектор длины 3
```

Пример использования конструктора вектора в вызове:

```
тип Строки = []Строка
фн Сохранить(строки: Строки) { /*тело*/ }

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 для конструируемого вектора
*	задает значение по умолчанию для всех элементов, для которых оно не задано явно
Индекс	задает значение элемента с индексом *Индекс*

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 (§6.13)
- 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:

```
тип Числа = []Цел64

пусть ч1 = Число[длина: 3, *: 0]
// вектор [0, 0, 0]

пусть ч2 = Число[длина: 5, *:0, 1: 1, 3: 3]
// вектор [0, 1, 0, 3, 0]

пусть ч3 = Число[*: 0, 1: 1, 3: 3]
// [0, 1, 0, 3], длина - макс индекс + 1
```

Пример ошибки - значение по умолчанию не задано:

```
пусть ч3 = Число[1: 1, 3: 3]
```

Вектор является динамическим массивом, длина которого может изменяться во время выполнения. Конструктор вектора выделяет память под max(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.

```
пусть ч1 = Число[выделить: 100]
// 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. Унарные операции

```
Унарная-операция: '-' | '~' | ':~'
```

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

Унарные операторы имеют приоритет выше, чем бинарные, то есть выражение `-x + 1` выполняется как `(-x) + 1`. Тип результата унарных операций равен типу операнда.

6.11. 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, 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
```

6.12. Бинарные операции

```
Бинарная-операция
    : Арифметические
    | Сравнения
    | Логические
    | Битовые
    | Сдвиги
Арифметические: '+' | '-' | '*' | '/' | '%'
Сравнения: '=' | '#' | '<' | '<=' | '>' | '>='
Логические: '|' | '&'
Битовые: ':|' | ':&' | ':\'
Сдвиги:   '<<' | '>>'
```

6.12.1. Приоритеты операции

Язык определяется пять уровней приоритета для бинарных операций. Операции умножения имеют самый высокий приоритет, а операция логическое ИЛИ (`|`) имеет самый низкий приоритет.

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

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

6.12.2. 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

6.12.3. 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 *T*	операнд типа *T*
мб *T*	'пусто' или операнд типа **мб** *T*

6.12.4. Логические операции

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

`а | б` означает если а, то истина, иначе б
`а & б` означает если а, то б, иначе ложь

6.12.5. Битовые операции

Операции *битовое ИЛИ* (`:|`), *битовое И* (`:&`) и *XOR - битовое исключающее ИЛИ* (`:\`) применимы к операндам целых типов (Байт, Цел64, Слово64), оба операнда должны быть одного типа. Тип результата равен типу операндов.

6.12.6. Операции сдвига

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

6.13. 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. Блоки

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

```
Блок: '{' Список-операторов? '}'
Список-операторов: Оператор (Разделитель Оператор)*
```

7.2. Локальные описания

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

```
Локальное-описание: Описание-переменной
```

Для локальных переменных не может быть задана *поздняя инициализация* (§4.4).
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(а)
```

7.4. Оператор присваивания

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

```
Оператор-присваивания: Выражение ':=' Выражение
```

Выражение в левой части должно быть *изменяемым*:

Левое выражение		**Условие**
Имя объекта	`пер` или `мод.пер`	*пер* - это изменяемая переменная (§4.4)
Доступ к полю	`что-то.поле`	*что-то* - это изменяемое выражение, *поле* - это изменяемое поле (§5.5.1)
Индексация	`что-то[индекс]`	*что-то* - это изменяемое выражение
Преобразование	`что-то(:Тип)`	*что-то* - 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 веткви, в противном случае, выполняется ветвь **иначе**, если она задана.

```
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!() }
```

7.7. Оператор 'надо'

Оператор **надо** используется для завершения исполнения операторов некоторого контекста, если условие, заданое 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
```

7.8. 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 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 варинты 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 ключевого слова **выбор** отсутствует, то это оператор *выбора по предикатам*.

```
Выбор-по-выражению:
    '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 рано истине, выполняется список операторов этого варианта. Остальные варинты 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-выражению:
    '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 использовать тип `К2`, так как это тип объекта `к`:

```
1  тип К1 = класс {}
2  тип К2 = класс (К1) {}
3  вход{
4      пусть к: К = К2{}
5      выбор тип к { ... }
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 варинты игнорируются. Если ни один из вариантов не выполнен, и есть ветка **другое**, выполняется список операторов этой ветки.

Сравнение типа выражение оператора выполняется на строгое равентсво, наследование не учитывается. Таким образом, в примере сработает ветка `когда К2`, независимо от порядка веток в операторе.

```
тип К1 = класс {}
тип К2 = класс (К1) {}
вход{
    пусть к: К = К2{}
    выбор тип к {
    когда К1:
    когда К2: // сработает эта ветка
    }
}
```

Если в операторе указана *переменная варианта*, то в каждой ветки исполнения (но не в ветке **другое**) будет доступна переменная с именем, заданным идентификатором, и типом, указанном в варианте. При этом в варианте должен быть указан только один тип.

```
тип Человек = класс { имя := "" }
тип Работник = класс (Человек) {
    зарплата := 0.0
}
фн зарплата(чел: Человек): Вещ64 {
    выбор пусть х: тип чел {
    когда Работник: вернуть х.зарплата
    другое вернуть 0.0
    }
}
```

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 `истина`. Выражение вычисляется перед каждой итерацией.

```
Оператор-пока: 'пока' Выражение Блок
```

```
пока а < б {
    а := а * 2
}
```

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?

```
пусть числа = Числа[9, 8, 7]
цикл [№]число среди числа {
    вывод.ф("%v:%v, ", №, число)
}
// 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'
```

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

7.12. Оператор 'вернуть'

Оператор **вернуть** завершает выполнение тела функции, метода или входа, и, возможно, возращает 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).

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

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

```
фн цифра?(сим: Символ): Лог {
    вернуть  // ошибка компиляции
        '0' <= сим & сим <= '9'
}
```

А так верно:

```
фн цифра?(сим: Символ): Лог {
    вернуть '0' <= сим
      & сим <= '9'
}
```

7.13. Оператор 'авария' и аварийная ситуация

Оператор **авария** запускает аварийную ситуация, 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.

```
пусть д1 = длина("Привет") // равно 6

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	тег(Цел64)	Тег указанного типа
описанный тип	тег(Байты)	Тег указанного типа
полиморфный параметр	тег(п)	Тег параметра
элемент вариативного полиморфного параметра	тег(п[номер])	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).

```
фн Что это?(п: *) {
    если тег(п) = тег(Цел64) {
        вывод.ф("это число: %v", нечто(п)(:Цел64))
    }
    иначе {
        вывод.ф("это не число")
    }
}
```

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, зависищей от реализации. Результат функции используется для динамической проверки типа.

8.2. Встроенные методы для векторов

Для всех типов векторов определены стандартные методы:

- `добавить`

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:

```
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.

```
тип Числа = []Цел64

пусть фиб = Числа[1, 1]
фиб.добавить(2, 3, 5)
```

9. Модули

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

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 идентификтор, 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 "стд::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 папки файловой системы и пути, относительно этой папки:

```
импорт "стд::юникод/utf8"

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 аболютным 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:

```
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*.

```
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 недопределенным.

Рассмотрим, в качестве примера, обобщенный модуль Стек (полный текст см. "стд/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 "стд/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:

```
модуль пример

импорт "стд/вывод"
импорт "модули/стек-цел"
импорт "модули/стек-строк"

вход {
    пусть с1 = стек-цел.Стек{}
    пусть с2 = стек-строк.Стек{}

    с1.положить(777)
    с2.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-строк`.

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 операция или при переопределения 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:

```
тип Числа = Цел64
пусть ч: Число = 1
пусть ц: Цел64 = 2
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 = выражение`, выражение должно быть совместимо с типом `T`
- оператор присваивания: `x := выражение`
- оператор `вернуть выражение`, выражение должно быть совместимо с типом результата функции
- передача фактических параметров при вызове: `Ф(аргумент1, ..., аргументN)`, аргумент должен быть совместим с типом параметра

Значение типа *ТВ* совместимо по присваиванию с целевым типом *ТП* (типом переменной, константы или параметра), если выполняется одно из условий:

- *ТП* и *ТВ* эквивалентны
- *ТП* это целый тип (Байт, Цел64, Слово64), а значение - это литерал целого типа, причем значение литерала входит в допустимый диапазон значений *ТП*
- *ТП* и *ТВ* это типы векторов и типы их элементов эквивалентны
- *ТП* это ***мб*** *T1*, а значение есть 'пусто' или *ТП* это ***мб*** *T2* и типы *T1* и *T2* эквивалентны
- *ТП* и *ТВ* это типы классов и *ТВ* является расширением *ТП*
- 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 ключевым словом осторожно, кроме того исходный файл, в котором используется *небезопасное преобразование* тоже должен быть помечен ключевым 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).

A. Версии языка

A.1. Версия 0.9.0 от 07.09.2023

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

A.2. 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)

A.3. Версия 0.9.2 от 08.10.2023

- Добавлены многострочные литералы (§3.6.4)

A.4. Версия 0.9.3 от 29.04.2024

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

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

A.5. Версия 0.9.4 от 15.06.2024

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

A.6. Версия 0.9.5 (в работе)

- Будут добавлены может-быть протоколы