Язык программирования Тривиль

Алексей Недоря

16.02.2023

Содержание

1. Назначение

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

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

Основными требованиями к языку при разработке были поставлены

- Язык должен быть минимально достаточным для удобной разработки компиляторов. Требование это во многом субъективно, так как компиляторы можно писать существенно по разному.
- Язык должен быть русскоязычным и с синтаксисом минимизирующим переключение на латиницу 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.

Несмотря на узкую направленность на разработку компиляторов, Тривиль является языком программирования общего назначения, пригодным для решения широкого круга задач.

Язык (и экосистема) обладает существенными предпосылками для использование его в качестве учебного языка для обучения студентов разработке компиляторов, библиотек, средств разработки, 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, автоматическим управлением памятью (сборка мусора), с поддержкой ООП.

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

Пример программы:

```
1  модуль х
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, §3.7, §3.8, §3.9) и модификаторов (§3.10). 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 таких 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:

```
1  буква
2  буква-или-цифра
3  №-символа
4  Цифра?
5  Пора паниковать!
```

3.4. Ключевые слова

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

авария	есть	когда	надо	прервать
вернуть	иначе	конст	осторожно	пусть
вход	импорт	мб	пока	тип
если	класс	модуль	позже	фн

3.5. Знаки операций и знаки препинания

Следующие последовательности символов обозначают знаки операций и знаки препинания:

```
+    -    *    /    %
&    |    ~
=    #    <    <=   >    >=
:=   ++   --
```

```
(   )   [   ]   {   }
(:  .   ^   ,   :   ;
```

3.6. Целочисленные литералы

```
Целочисленный-литерал: Цифра+  | '0x' Цифра16+
Цифра16: '0'..'9' | 'a'..'f' | 'A'..'F'
```

3.7. Вещественные литералы

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

```
Вещественный-литерал: Цифра+ '.' Цифра*
```

3.8. 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
    : '"'
    (~('"' | '\\' | '\n' | '\r' | '\t') | Escape)*
    '"'
Escape
    : '\\'
    ( 'u' Цифра16 Цифра16 Цифра16 Цифра16
    | 'n' | 'r' | 't'
    | '"'
    | "'"
    )
```

3.9. Символьные литералы

Символьный литерал задает значение для Unicode code point, - это последовательность символов, заключенные в двойные кавычки. 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)
    "'"
```

3.10. Модификаторы

Модификаторы используются в исходном тексте, чтобы внести изменение в в семантику и/или синтаксис конструкции языка, см., например, §4.10.2.

```
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 (§4.1). Идентификатор может быть описан как тип, константа, переменная, функция или как поле класса.

```
Описание
        : Описание-типов
        | Описание-констант
        | Описание-переменных
        | Описание-функций
```

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 видимости:

- Областью действия предопределенного идентификатора является Универсум
- Областью действия идентификатора, описанного на верхнем уровне (вне какой-либо функции), является весь модуль (§8).
- Областью действия имени импортируемого модуля является файл (часть модуля), содержащего импорт (§8.1).
- Областью действия идентификатора, обозначающего параметр функции, является тело функции (§4.10).
- Областью действия идентификатора, описанного в теле функции (§4.10) или теле входа (§8.2), является часть *блока* (§6.1), в котором описан идентификатор, от точки завершения описания и до завершения этого блока.

В описании сразу за идентификатором может следовать признак экспорта '*', указывающий, что идентификатор *экспортирован* и может использоваться в другом модуле, *импортирующем* данный (§8.1).

```
Идент-оп:  Идентификатор '*'?
```

4.1. Предопределенные идентификаторы

Следующие идентификаторы неявно описаны в области действия *Универсум*.

Типы (§4.2):

```
Байт Цел64 Слово64 Вещ64 Лог Символ Строка
```

Константы типа Лог (§4.2):

```
ложь истина
```

Литерал nullable (§4.9):

```
пусто
```

Стандартные функции (§7):

```
длина тег нечто
```

Кроме того, для векторных типов определен набор встроенных методов (§7.3).

4.2. Предопределенные типы

Следующие типы обозначаются предопределенными идентификаторами, значениями данных типов являются:

Тип	Множество значений
Байт	множество целых числа 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 литералов

Операции над значениями этих типов определены в (§5.7).

4.3. 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 (§4.4), переменных (§4.4) и полей класса (§4.8).

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

```
Указ-типа: 'мб'? Квалидент
Квалидент: Идентификатор ('.' Идентификатор)?
```

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* (§4.9).

4.4. 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 (§5). Выражение для константы должно быть константным выражением (§5.9).

```
конст к1: Цел64 = 1 // тип Цел64
конст к2: Байт = 2 // тип Байт
конст к3 = 3 // тип Цел64
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:

```
конст *(
    Счетчик = 1
    Имя = "Вася"
)
```

Пример неявного задания значения для констант:

```
конст (
    / / операции
    ПЛЮС = 1    // тип Цел64, значение = 1
    МИНУС        // тип Цел64, значение = 2
    ОСТАТОК    // тип Цел64, значение = 3
    // ключевые слова
    ЕСЛИ = 21    // тип Цел64, значение = 21
    ИНАЧЕ         // тип Цел64, значение = 22
    ПОКА           // тип Цел64, значение = 23
)
```

4.5. 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 (§5).

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 (§6.2, §6.3).

Явное указание типа:

```
пусть ц1: Цел64 = 1
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. 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 (§8.2). Текущая реализация компилятора запрещает позднюю инициализацию для переменных с единственным присваиванием.

4.6. Описание типов

Описание типа связывает идентификатор с новым типом, который определяет структуру данных этого типа, и, как следствие, набор операций над данными этого типа.

```
Описание-типа: 'тип' Идент '=' (Тип-вектора | Тип-класса)
```

В текущей версии языка поддерживают два структурных типа: тип вектора (§4.7) и тип класса (§4.8).

4.7. 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 (§5.4). 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 изменятся во время выполнения, другими словами, вектор - это *динамический массив*. Для получения текущей длины вектора используется стандартная функция длина (§7.1). Начальное значение для объекта типа вектор задается с помощью конструктора вектора (§5.5). Далее, длину вектора можно поменять с помощью стандартных методов (§7.3).

Пример инициализации вектора из трех элементов:

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

4.8. Тип класса

Тип класса - это структура, состоящая из полей. С типом класса могут быть связаны функции, называемые методами (§4.11).

Описание класса состоит из опционального указания базового класса и списка полей. Если базовый класс указан, то класс *наследует* поля и методы базового класса (§4.8.2).

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 (§5.2).

```
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 (§5).

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 (§6.2, §6.3).

Если в инициализации задано *Выражение*, то начальным значение поля является значение выражения. 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 (§5.6).

4.8.1. 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.

4.8.2. Наследование

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

Базовый класс *Б*, указанный в описании класса *К*, называется *прямым базовым классом*. Так как *Б* 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.

4.9. 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*.
- 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
- Сравнение на равно/не равно с 'пусто' или с другим объектом типа *мб T*
- Операция `'^'` перехода от значения типа *мб T* к значению типа *T*

Подробнее в §5.8, §9.2.

Пример указания типа и использования объекта:

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

4.10. Описание функций

Описание функции состоит из идентификатора, сигнатуры и тела функции. Сигнатура определяет формальные параметры и тип результата (если таковой имеется). Особым видом функции является метод, см. §4.11.

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

Вместо тела функции может стоять модификтор (§3.10) `@внеш`, см. §4.10.2.

Если у функции задан тип результата, тело функции должно завершаться операторами вернуть (§6.4) или авария (§6.5).

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

4.10.1. Параметры

Формальные параметры - это идентификаторы, которые обозначают *аргументы* (фактические параметры), указанные при вызове функции (§5.3).

```
Параметр: Идентификатор ':' '...'? ('*' | Указ-типа)
```

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 вместо типа параметра указан символ `'*'`, то параметр называется полиморфным. Аргумент, соотвествующий 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 (§7.2).

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

4.10.2. 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.

4.11. Описание методов

Метод - это функция, связанная с типом классом. Для вызова метода (§5.3) 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 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 переопределеного метода должна совпадать с сигнатуром переопределяемого метода, а именно:

- Число параметров должно совпадать
- Типы параметров должны быть эквивалентны §9.1
- Признаки вариативности и полиморфности должны совпадать

Пример ошибки при переопределении (разное число параметров):

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

5. Выражения

5.1. Операнды

5.2. Доступ к полям и методам

5.3. Вызов функции или метода

5.4. Индексация

5.5. Конструктор вектора

5.6. Конструктор экземпляра класса

5.7. Операции

5.8. Операции для объектов может быть типа

5.9. Константные выражения

6. Операторы

6.1. Блоки

6.2. Оператор присваивания

6.3. Инкремент и декремент

6.4. Оператор вернуть

6.5. Оператор авария

7. Стандартные функции

7.1. Длина

7.2. Функции для полиморфных параметров

7.3. Встроенные методы для векторов

8. Модули

Заголовок модуля не является описанием, имя модуля не принадлежит никакой области действия. Его цель - идентифицировать файлы, принадлежащие одному и тому же модулю.

8.1. Импорт

Ошибка компиляции, если знак экспорта указан для идентификатора, описанного не на уровне модуля.

8.2. Вход или инициализация модуля

9. Правила совместимости

9.1. Эквивалентность типов

9.2. Совместимость по присваиванию