Разработка языка Тривиль. Часть 2

А. Е. Недоря, г. Санкт-Петербург

Статья является второй из серии статей, в которых описывается разработка языка программирования Тривиль. В первом статье описывался первый этап разработки языка: определение целей, требований и критериев выбора. Эта статья содержит обоснование основных языковых решений. Как и вся серия, статья нацелена, большей частью, не на программиста, который использует язык, а на разработчика языков программирования.

Введение

В первой статье серии [1] были определены **цели языка**: разработка и прототипирование компиляторов и библиотек, и **основные требования**, суть которых в том, чтобы получить нужный язык и компилятор в краткие сроки с минимальными затратами ресурсов. Кроме того, была рассмотрена принципиально важный критерий **энергосбережения разработчика**, следование которому помогает и далее работать над семейством языков быстро и экономично.

Данная статья описывает конструкции языка Тривиль, уделяя внимание тем конструкциям, которые еще не устоялись в языках программирования[1] и которые можно сделать по-разному, как с точки зрения синтаксиса, так и семантики.

Типы

Система типов — это ядро языка, а выбор набора типов существенно определяет выразительность языка и его применимость для решения тех или иных задач. Соответственно, при разработке языка общего назначения, стоит уделить особое внимание полноте системы типов, см. [2]. А при разработке специализированного языка, которым является Тривиль, надо добавить те и только те типы, которые нужны для предметной области.

Так как предметная область — это разработка компиляторов, то мы приходим к вопросу: какие типы нужны для разработки компилятора? А для того, чтобы на него ответить, надо иметь хотя бы общее понимание: как будет устроен компилятор. Замечу, что одновременная разработка языка и компилятора — это особенность конкретной разработки, вытекающая из требования минимизации объема работы.

Архитектура компилятора: первый взгляд

Учебники по разработке компилятора, как правило, начинаются с вопроса о числе проходов компилятора (см. [3]), и принципиальным является выбор между однопроходным и многопроходным компилятором.

Однопроходные компиляторы выполняет трансляцию за один проход по исходному тексту, делая семантические проверки и генерацию кода по ходу выполнения синтаксического анализа. В многопроходном компиляторе, как правило, сначала выполняется синтаксический анализ, который строит некоторое представление программы. Следующие проходы обрабатывают не исходный текст, а представление, которое может трансформироваться от прохода к проходу или заменяться на другое представление. Например, компилятор языка Rust последовательно создает и использует такие представления [4]: AST[2] => HIR => MIR => LLVM IR.

Как правило, однопроходный компилятор короче (меньше строк кода), но вовсе не обязательно проще. Нарушение принципа *разделения проблем* (separation of concerns) приводит к более сложному и трудно модифицированному коду.

Так как одна из задач языка — прототипирование, а одно из требований — простой и легко модифицируемый компилятор, однопроходный компилятор не годится. При этом, проходов должно быть небольшое количество, а промежуточное представление желательно использовать одно, так как каждое промежуточное представление существенно увеличивает объем работы.

Итак, компилятор Тривиля должен быть построен из нескольких проходов на базе одного промежуточного представление. Тогда выбор представления очевиден — это классическое

1 AST - Abstract Syntax Tree

2 см. функцию “ф”, реализованную в библиотеках стд::строки и стд::вывод.

абстрактное синтаксическое дерево (АСД или AST).

АСД программы (точнее единицы компиляции) — это дерево, в котором каждой конструкции языка соответствует узел, например, узел для описания оператора `если` или узел для операции сложения. Следовательно, язык должен включать типы, достаточные для описания узлов, которые традиционно делятся на

- базовые или предопределенные типы
- и конструируемые (user-defined) типы

Предопределенные типы

Список предопределенных типов Тривиля:

Байт	байт, беззнаковое целое (8 битов)
Цел64	целое число (64 бита)
Слово64	беззнаковое целое число (64 бита)
Вещ64	плавающее число (64 бита)
Лог	логическое (boolean)
Символ	кодировка знака в Юникоде (unicode code point, 32 бита)
Строка	последовательность знаков в кодировке UTF-8 (см. след. раздел)

Несколько пояснений:

1) Почему такой неполный набор числовых типов? Где целые размера 8, 16, 32 бита, где беззнаковые размера 16 и 32, вещественные 32, 80, 128?

Вспоминаем требование: *ничего, кроме того, что необходимо для целевой области*. Мой опыт показывает, другие типы не обязательны для компилятора. Добавление каждого числового типа в язык приведет к увеличению компилятора, и если для каждого такого типа надо добавить всего несколько десятков строк, то в совокупности добавление, скажем 7 типов, даст уже заметную добавку в строках и усложнение семантического анализа и генерации.

2) Зачем в именах числовых типах указывается размер (Цел64, а не Цел)? Для упрощения понимания и переносимости программ. Многие современные языки системного уровня идут по этому пути, например: Go: типы int32, float64, Rust: i32, f64. Эти языки добавляют еще и целые типы (например, int в Go), размер которых зависит от платформы. Для низкоуровневого программирования в таких типах есть некоторый смысл, но в общем, я считаю их скорее вредными, чем полезными.

Символы и Строки

Строки и символы являются обязательными типами данных, и подход к этим типам существенно менялся со временем.

В наше время правильный ответ на вопрос: *какой набор символов должен поддерживать языка программирования*, очевидно, Юникод.

А вот ответ на следующий вопрос, вопрос выбора кодировки, уже не так однозначен. Впрочем, современные языки, выбирая между UTF-8, UTF-16 и UTF-32, дружно выбирают UTF-8. Основные преимущества: компактность, совместимость с ASCII, отсутствие преобразований при работе с файлами.

Итак, берем в Тривиль Юникод и UTF-8, и у нас есть проблема. На строку в кодировке UTF-8 можно смотреть двумя способами:

- как на последовательность символов (code point) Юникода
- как на последовательность байтов (в которой каждый символ задан переменным числом байтов)

Для последовательности символов прямая индексация не определена, а для последовательности байтов определена. Эта двойственность должна быть как-то отражена в языке, и, желательно, так

чтобы разработчик (энергосбережение!) сразу понимал, с чем он имеет дело. Приведу пример на Go, где это пожелание не выполнено:

	Код	Вывод
1	`var s = "ю"`	
2	`fmt.Println(s[0])`	209
3	`for _, c := range s {`	
4	`  fmt.Println(c)`	1102
5	`}`	

Первая печать (строка 2) выведет числовое значение первого байта кодировки символа “ю” (русские буквы в кодировке UTF-8 занимают два байта), а вторая (строка 4) кодировку букву “ю” (code point).

Странность здесь в том, что для любого другого массива (не строки) длины 1 индексация и обход (цикл foreach) выведут одно и тоже. Это еще один пример скрытой магии: понять нельзя, надо запомнить.

В Тривиле для решения проблемы двойственности, добавлен вспомогательный тип `Строка8`:

- `Строка`: неизменяемая (immutable) последовательность символов Юникод, индексация которой **запрещена**
- `Строка8`: неизменяемая последовательность байтов, индексация которой **разрешена**

Преобразование между этими типами происходит без накладных расходов:

	Код	Вывод
1	**пусть** ст = "ю"	
2	вывод.ф("$;\n", длина(ст))	1
3	**пусть** ст8 = ст(:Строка8)	
4	вывод.ф("$;\n", длина(ст8))	2

В строке 3 записано преобразование типа. Как видно, длина строки равна числу символов, а длина строки, преобразованной к типу `Строка8`, равна числу байтов. Так как строку нельзя индексировать, то путаницы нет.

Конструируемые типы

Какие типы нужны для построения АСД? Очевидно, что надо описывать

- Узлы — нужен **класс**
- Последовательности (описаний, операторов и т.д.) — нужен контейнер, содержаний обеспечивающий упорядоченное множество элементов: массив или **вектор**
- И, так как Тривиль — язык с безопасными ссылками (null safe), а в узле могут быть необязательные под-узлы, то нужен **опциональный** тип.

Другие типы не обязательны. В действительности, нужен еще словарь (hash map), и об этом мы еще поговорим отдельно, но, как и в большинстве языков, я предпочитаю сделать его библиотечным типом, а не встроенным в язык[3].

Тип **класс**, как обычно, описывает структуру объекта, для которого может быть задан набор методов. Для класса определено понятие наследования.

Тип **вектор** определяет динамический массив элементов одного типа. Динамический здесь означает, что размер вектора может меняется во время выполнения программы.

Опциональный тип (или тип *может-быть*) создает новый тип на основе ссылочного типа

3 Я не уверен, что языке Дракон стоит писать, но это один из языков, который стоит знать.

(класс, вектор, Строка) добавляя специальное значение `пусто` (отсутствие значения).

Для каждого конструируемого типа определен обычный набор операций, доступ к полю, индексация, проверка на отсутствие значения и т.д.

Обычно в языках существенно больше конструируемых типов. Таблица показывает, каких типов нет в языке, в сравнении с другими языками (см. обзор в [2]):

1	структура (struct)	Go, Rust, Swift
2	массив фиксированной длины	Go, Rust
3	функциональный тип	все современные языки
4	тип-произведение: кортеж (tuple)	Rust, Swift, Go (частично)
5	тип-сумма: объединение (union) или перечисление (enum)	Kotlin, Rust, Swift
6	интерфейс (статический, через наследование)	Kotlin, Rust, Swift
7	интерфейс (утиная типизация)	Go

Любопытно, что типов, который нет в Тривиле больше, чем типов, которые есть. Это следствие специализации языка. Еще раз замечу, что в менее специализированном языке надо думать о полноте системы типов, но у нас не тот случай.

Так как целью статьи не является полное описание языка (см. [5]), то для примера приведу фрагмент описания АСД из компилятора (см. трик/асд/типы.tri в [6]), комментарии убраны для сокращения текста:

```
1  тип ТипКласс* = класс (Тип) {
2     Т-базовый*: мб Тип := пусто
3     поля* = Поля[]
4     методы*  = Методы[]
5     атрибуты* = Словарь{}
6  }
7
8  тип Поле* = класс (Описание) {
9    значение*: мб Выражение := пусто
10    задать-позже* := ложь
11    одно-присваивание* := ложь
12 }
13
14 тип Поля* = []Поле
15 тип Методы* = []Функция
```

В строках 1-6 описывается узел для типа класса, в котором есть поля:

- базовый тип класса, опциональный, так как может отсутствовать
- вектор полей класса
- вектор методов класса
- словарь (hash-map) атрибутов, то есть и полей и методов для быстрого поиска

ТипКласс является наследником типа Тип (строка 1). Запись поля* = Поля[] означает, что поле с именем поля инициализируется пустым вектором типа Поля. Тип поля выводится из типа инициализирующего значения.

Тип Поля определен в строке 14, как вектор элементов типа Поле.

Символ ‘*’ после имени означает, что имя *экспортировано*, то есть может быть использовано в

модулях, которые импортируют модуль с этими описаниями.

В примере видна еще одна особенность идентификаторов Тривиля, возможность использование символа тире ‘-’. На мой взгляд, задать-позже читается существенно проще чем задать_позже, не говоря уже о верблюжьем стиле.

В описании поля можно использовать идентификатор с пробелами, я использую здесь идентификатор с тире, как часть стиля:

- имена функция: с пробелами
- имена переменных и полей: с тире
- имена типов с большой буквы

Развитие типовой системы

Те, кто прочитает свежее описание языка Тривиль [6] обнаружит еще два конструируемых типа в дополнение к тем, что описаны в предыдущем разделе, а именно:

- тип протокола
- и тип функции

Эти типы добавлены для следующего языка семейства, языка Арвиль[4], который является надмножеством языка Тривиль, но доступны и в Тривиле. В компиляторе Тривиля они не используются.

Описания

Описания в языке программирования определяют новые сущности и дают им имена. Основной набор сущностей устоялся, это

- типы
- константы и переменные
- функции

Многие языки добавляют другие описания, но для Тривиля этот набор достаточен.

Описание типов

О типах мы достаточно подробно поговорили в предыдущем разделе. Замечу только, что нотация в стиле языков Oberon и Go с использованием одного ключевого слова **тип** и общего синтаксиса — это тоже часть энергосбережения разработчика. Тип есть тип, и он описывается, как тип.

Языки, которые используют разные нотации для описания разных типов (например: Swift, Kotlin) вносят некоторую путаницу в головы разработчиков. Например, становится возможным вопрос: А класс — это тип?

Описание констант и переменных

Тривиль, как и большинство языков, позволяет описывать

- константы, значения которых вычисляется до исполнения программы (во время компиляции)
- изменяемые переменные
- неизменяемые переменные, то есть переменные, которые инициализируются один раз и далее значение их не может изменится. В описании языка они называются *переменными с единственным присваиванием*.

Пример описаний:

```
конст PI = 3.14159  // константа
пусть число-элементов := -1 // изменяемая переменная
пусть макс-индекс = длина(аргументы) - 1 // неизменяемая переменная
```

Для всех констант (что обычно) и переменных (что не обычно) в описании обязательно задано инициализирующее значение. Описание изменяемых и неизменяемых переменных отличается знаком инициализации: ‘`:=`’ для изменяемых и ‘`=`’ для неизменяемых. Я полагал, что сам придумал эту нотацию, но потом мне подсказали, что также было сделано в языке Algol-68 (ничто не ново под луной).

Обязательность инициализации — это энергосбережение, так как заставляет думать при

4 Во многих языках используется термин lvalue (левостороннее выражение).

описании переменных и уменьшает число ошибок. Тип переменной, в большинстве случаев, выводится из инициализирующего выражения, но может быть задано явно, а в некоторых случаях обязательно задается явно, например, для переменной опционального типа:

```
пусть тек-элемент: мб Элемент := пусто
```

Для констант есть синтаксис, позволяющий задать группу констант с последовательными значениями. Подсмотрено в Go и очень удобно для компиляторов. Например, виды лексем задаются так:

```
// Лексемы:
конст (
   НЕОП = 0
   ИДЕНТ
   ЦЕЛ
   ВЕЩ
   …
)
```

Описаний функции и методов

Описание функций и методов, с одной стороны, является сложной частью многих языков, но в Тривиле сделано обычным для современных языков образом. Приведу для примера:

```
фн факториал(n: Цел64): Цел64 {
   надо n > 1 иначе вернуть 1
   вернуть n * Факториал(n - 1)
}
```

В теле функции используется оператор `надо`, о котором мы поговорим в следующем разделе. Относительно нетривиальной частью описания функции, являются вариативные и полиморфные параметры, использование которых, например, позволяет написать библиотечную функцию[5] вывода на консоль значения любого типа, а это еще одна важная часть энергосбережения разработчика. Но эта тема достаточно специфичная, и выходит за рамки статьи.

А вот об описании методов и классов надо поговорить подробнее. В большинстве современных языков, описание методов класса текстуально входит в описание класса, исключение — это языки Rust и Go, в которых методы текстуально описаны за границами описания того типа, с которым связан метод.

Rust	Go

`struct Rectangle {`
`    width: u32,`
`    height: u32,`
`}`
 	
`impl Rectangle {`
`  fn area(self: &self) -> u32 {`
`    self.width * self.height`
`  }`
`}`
	`type Rectangle struct  {`
	`    width uint32`
	`    height uint32`
	`}`
	
	`func (rec *Rectangle) area() uint32 {`
	`  return rec.width * rec.height`
	`}`

5 Например, оператор `пока` (`while`) сделан одинаково в современных языках (с точностью до деталей синтаксиса)

Я недостаточно хорошо знаю Rust, чтобы уверенно сказать, почему выбрана такая нотация, в случае же Go, есть, по крайней мере, две важных причины:

- метод может быть определен для разных типов, не только для класса (структуры)
- синтаксис метода взят из языка Oberon-2[7], который оказал существенное влияние на Go.

Вынесение методов из описания класса (типа) дает важные преимущества с точки зрения понимания текста. Рассмотрим для иллюстрации пример на языке Kotlin:

```
class Rectangle {

        // много строк кода

   fun area(): Int { /* тело */  }

        // много строк кода
     }

fun foo() { /* тело */ }
```

1. Можем ли мы, читая строку с описанием функций area и foo понять, что именно описывается — функция или метод, не привлекая контекст (который может быть весьма длинным)? Это пример неявной и не очевидной семантики. В Go функции и методы синтаксически различаются.
2. Rust и Go позволяют разработчику описывать рядом методы и функции, которые методы используют. И это тоже важно, потому что, правильное упорядочивание исходного текста существенно влияет на чтение и понимание текста.
3. В Go, так как исходный текст единицы компиляции может быть разложен на несколько файлов, семантически близкие методы и другие описания могут находится в одном файле, а другие методы в других файлах. Такая структуризация очень полезна. Например, методы парсера в компиляторе могут быть сгруппированы: разбор выражений отдельно, разбор описаний отдельно и т.д.

Мне кажется, что группировка методов в области описания класса — это скорее дурная привычка, чем что-то еще. Я понимаю, почему так сделано в языке Java, так как в нем нет ничего, кроме классов. Java представляет собой крайнее выражение языков, реализующие CLOP (class-oriented programming). Я выделяю такой класс языков или подходов, потому что OOP вообще не требует наличия классов, языки Lua, Javascript и Go это подтверждают. Я не вижу преимуществ такого подхода, в языках, которые отходят от чистого CLOP, как Kotlin или Swift.

Еще одно отличие языков Go и Тривиль — это явный идентификатор для получателя (receiver) или другими словами, для объекта, к которому применен метод. То есть вместо использования this (Java, Kotlin) или self (Rust, Swift) разработчик может использовать значимое имя.

Тот же пример на Тривиле:

```
тип Прямоугольник = класс {
  ширина: Цел64 := 0
  высота: Цел64 := 0
}

фн (фигура: Прямоугольник) площадь(): Цел64 {
   вернуть фигура.ширина * фигура.высота
}
```

На первый взгляд, использование явного имени может показаться мало значащим, но это оказывает существенное влияние на упрощение семантики языка. Рассмотрим пример-загадку на языке Kotlin:

```
class C {
   var a: Int = this.foo()
   var b = this.a + 1
   fun foo(): Int { return this.b }
     }

fun main() {
    val c = C()
    println(c.a)
    println(c.b)
}
```

Вопрос: что напечатает эта программа? Собственно, что именно она напечатает, нам не важно, важно, то, что это не очевидно. Причина неочевидности в магическом символе `this`, который можно использовать не только в методах, где его использование понятно, но и в любом выражении внутри класса. Мне могут возразить, что все использования `this` из этого текста на Котлине можно убрать. Да, это так. Но неявное использование `this` только усиливает туманность этого кода. А для разработчиков языка и компилятора становится головной болью, так как существенно осложняет семантические правила и их проверку в компилятора.

Если же мы посмотрим в примеры на Тривиле и Go, то в них отсутствуют такие семантические сложности, так как нет никакого магического символа, а есть обычный идентификатор (например, `фигура`) которые используется **регулярным**(!) образом и только в методах. И это очень важно: не преумножай число сущностей сверх необходимого, иначе в них можно утонуть.

Операторы и выражения

Договоримся о терминологии:

- *оператор* (statement) управляет вычислениями
- *выражение* (expression) определяет вычисление значения путем применения операций и функций к операндам

Перевод statement как оператор является устоявшимся, но приводит к неоднозначности на границе русского и английского языка. Суть её в том, что operator и оператор имеют разные значения:

- оператор — это statement, например, while statement — это оператор цикла ‘пока’
- operator — это знак операции, например, ‘+’ — это знак операции сложения

Все современные императивные языки разделяют операторы и выражения, но одни делают упор на *выражения* (Kotlin, Swift: почти все есть выражение), другие — на *операторы* (Go). Например, в языке Kotlin `if` — это выражение, которое также может быть использовано (как и любое выражение) в позиции оператора.

```
// if в позиции выражения:
val x = if (cond) { 1 } else { 2}
// if в позиции оператора:
val y: Int
if (cond) { y = 1 } else { y = 2 }
```

В языке Go `if` — это оператор, который не вычисляет выражение, и не может быть использован в позиции выражения.

Впрочем, все современные языки сходятся на том, что есть операторы, которые не могут быть выражениями, например, оператор присваивания, оператор цикла или оператор `break`.

Выбор между двумя подходами во многом субъективен, но для Тривиля, если вспомнить требование простоты языка и компилятора, очевиден выбор в сторону операторов. Поясню на немного измененном примере:

```
if (cond) { 1 } else { 2.1 }
if (cond) { 1 } else { 2.1 } + 1
```

Вопрос: корректны ли эти выражения? И какого они типа?

Мне, в данном случае, не важно, какой ответ дает Kotlin. Пример нужен для того, чтобы показать: добавление в язык условного выражения добавляет целый пласт вопросов, на которые должна отвечать спецификация языка. В итоге, язык и компилятор становятся сложнее.

Поэтому Тривиль — это язык операторов (как Go). Операторы в нем, как и выражения, большей частью, самые обычные для современных языков программирования:

- оператор присваивания (`:=`)
- операторы увеличить и уменьшить (`++` и `--`)
- условный оператор `если` (if)
- оператор цикла `пока` (while)
- оператор `выбор` (switch)
- операторы `вернуть` (return) и `прервать` (break)

но есть и не совсем обычные:

- оператор `надо`
- оператор `авария`

О том, что необычно мы поговорим в следующих разделах.

Оператор надо

Оператор `надо` является условным оператором, как и оператор `если`. Отличие его в том, что оператор `надо` проверяет логическое условие и завершает работу программы, функции или цикла, если условие не выполнено. Аналогичный оператор в языке Swift называется guard statement. В сети можно легко найти статьи, демонстрирующие использование и преимущества этого оператора.

Пусть надо написать функцию, которая сравнивает на равенство две опциональные строки. Для сравнения значение каждого параметра надо сначала проверять на отсутствие значения (`пусто`), а затем сравнивать строки. Напишем, сначала, в классическом стиле с использованием оператора `если`:

```
1  фн равны?(стр1: мб Строка, стр2: мб Строка): Лог {
2    если стр1 # пусто {
3      если стр2 # пусто {
4        вернуть стр1^ = стр2^
5      } иначе {
6        вернуть ложь
7      }
8    } иначе {
9      вернуть стр2 = пусто
10   }
11 }
```

Знак операции ‘^’ в строке 4, означают операцию подтверждения типа, которая выполняет (во время исполнения) проверку, что значение типа `мб Т` есть `Т`, а не `пусто`. В Kotlin аналогичная операция обозначается знаком ‘!!’, а в Swift знаком ‘!’.

Полагаю, что текст выше выглядит вполне обычно, по крайней мере, так могут написать. Перепишем пример с использованием оператора `надо`:

```
1  фн равны?(стр1: мб Строка, стр2: мб Строка): Лог {
2    надо стр1 # пусто иначе вернуть стр2 = пусто
3    надо стр2 # пусто иначе вернуть ложь
4    вернуть стр1^ = стр2^
5  }
```

В строке 2 проверяется условие, и его выполнение гарантируется для следующих операторов текущего блока. То же самое в 3-й строке. В 4-й строке оба параметра — это строки (не пусто), и их

можно сравнивать. Текст функции сократился в 2 раза, и он стал гораздо проще. Энергосбережение очевидно.

На первый взгляд, может показаться, что эту функцию можно написать также понятно и с помощью `если`:

```
1  фн равны?(стр1: мб Строка, стр2: мб Строка): Лог {
2    если стр1 = пусто { вернуть стр2 = пусто }
3    если стр2 = пусто { вернуть ложь }
4    вернуть стр1^ = стр2^
5  }
```

Но так, как правило, никто не пишет, а скорее пишут так (по крайней мере, аналогичный текст на Go будет отформатирован именно так):

```
1  фн равны?(стр1: мб Строка, стр2: мб Строка): Лог {
2    если стр1 = пусто {
3      вернуть стр2 = пусто
4    }
5    если стр2 = пусто {
6      вернуть ложь
7    }
8    вернуть стр1^ = стр2^
9  }
```

Впрочем, основная разница не в числе строк, а в том, что

- оператор `надо` дает гарантию завершения блока в случае невыполнения условия
- и для `если` приходится инвертировать условие, а это тоже может сказаться на сложности понимания.

Вообще, мне кажется, программисты редко отслеживают и обычно недооценивают сложности понимания вложенных условных операторов, а это постоянный источник ошибок.

Дополнение: в визуальном языке Дракон [8] вводится понятие главного пути вычисления, то есть того вычисления, ради которого и сделан алгоритм[6]. Оператор `надо` убирает помехи с главного пути, и позволяет сделать его более явным.

Оператор выбора: сопоставление с образцом на минималках

Сопоставление с образцом (pattern matching), в той или иной степени, присутствует во всех современных языках (см., например, Swift, Kotlin, Rust). При этом, в каждом из этих языков есть свои существенные особенности в этом механизме, что означает, что механизм еще не устоялся. Во всех этих языках, механизм достаточно сложный, как с точки зрения конструкций языка, так и с точки зрения реализации. С другой стороны, он полезен при написании компиляторов, то есть мы имеем классическую ситуацию: хочется и колется.

Для Тривиля это означает, что надо найти баланс между сложностью и полезностью, а именно: взять минимальный набор образцов, самых важных для предметной области и простых в реализации, и отбросить остальные.

Какие образцы нужны для компилятора, основой которого является АСД? АСД — это дерево объектов разных типов (классов), объекты надо различать и тогда принципиально нужным образцом является *сопоставление с типом*, а без остальных можно обойтись.

Если это принять, то уже есть язык, который именно так подходит к сопоставлению с образцом, впрочем, вообще не используя этот термин. В Go есть три варианта оператора выбора (switch), один из которых обеспечивает сопоставление с типом, второй — это классический выбор по выражению, а третий, который я называю предикатным, позволяет делать, пусть и не самым удобным способом, сопоставление с образцом в общем виде. Тривиль использует такой же подход, с тремя вариантами оператора выбора.

6 Я использую термин единица компиляции, как обобщающий для модулей, пакетов, crates и т.п.

Для иллюстрации приведу фрагмент из текста компилятора, проверка семантики операторов (см. трик/семантика/контроль/кон-операторы.tri в [6]):

```
1  фн (кон: Контроль) оператор(оп: асд.Оператор) {
2     выбор пусть тек: тип оп {
3     когда асд.ОператорПрисвоить:
4        кон.выражение(тек.Л)
5        кон.выражение(тек.П)
6        ...
7     когда асд.ОператорЕсли:
8        ...
9     когда асд.ОператорПока:
10       ...
11    другое
12       авария(строки.ф("необработанный оператор: $тип;", оп))
13    }
14 }
```

В строке 2 заголовок оператора выбора по типу определяет переменную тек. Значение этой переменной равно оп, а тип в каждом варианте выбора будет равен типу, указанному в варианте. Например, в строке 4 тип переменной тек — это асд.ОператорПрисвоить, что позволяет работать напрямую с полями объекта этого типа. Тем самым, кроме простого сопоставления с типовым образцом, мы получаем простой вариант того, что в Kotlin называется “умным” преобразованием (smart cast).

Если переменная в вариантах выбора не нужна, то её можно опустить: **выбор тип** оп {}.

Оператор выбор по выражению и выбор по предикатом аналогичны операторам языка Go.

Преобразование типа

Преобразование типа (type casting, type conversion) – это еще один механизм, для которого не найдено общее решение в языках программирования. Следующая таблица подтверждает это, показывая синтаксис преобразования целого числа в вещественное в нескольких языках:

C	`(double) 5`
Go	`double(5)`
Kotlin	`5.toDouble()`
Swift	`Double(5)`

И это только верхняя часть айсберга, так как во многих языках есть дополнительные нотации для преобразования типов (в таблице и далее: expr — выражение, T — целевой тип преобразования):

Go	type assertion: `expr.(T)`
Kotlin	nullable cast: `expr as? T` unsafe cast: `expr as T`
Swift	conditional cast: `expr as? T` forced cast: `expr as! T`

Для подтверждения слов об айсберге, замечу, что в C++ есть, как минимум, пять вариантов преобразования: static_cast, dynamic_cast, const_cast, reinterpet_cast и преобразование в стиле C. Привожу этот факт, как подтверждение того, что проблема есть, она известная и не простая.

Увы, для Тривиля не подходит ни одна из известных нотация, потому, вместо заимствования, пришлось определить требования и искать свое решение.

Требования:

1. один синтаксис для всех преобразований
2. возможность высокоуровневых преобразования и низкоуровневых (сохраняющих битовое представление) преобразований и их очевидное различие (несмотря на 1.)
3. возможность использования преобразования в левой части присваивания без дополнительных скобок

Последний пункт надо пояснить. В языке Go конструкция type assertion позволяет написать: `expr.(Type).field`, то есть написать без лишних скобок то, что в Go называют первичным выражением[7]. А в Котлине придется писать `(expr as Class).field`, что на мой взгляд, более энергоемко из-за необходимости ставить скобки вокруг бинарной операции `as` и помнить об этом. Впрочем, синтаксис Go мне также представляется неудобным и неочевидным: нельзя понять, надо запомнить.

В Тривиле используется для всех преобразований используется запись вида: `выражение(:Тип)`. Например, преобразование целого в вещественное: `5(:Вещ64)`.

Обратите внимание, что

1. текст читается слева направо естественным образом, сначала выражение, которое первично, потом целевой тип преобразования. Как видно в таблице выше, только в языке Kotlin текст читается слева направо, и только за счет того, что преобразование делается методами..
2. Знак ‘`:`’ устойчиво ассоциируется с типом в описании переменных и параметрах, использование его в преобразовании достаточно естественно.
3. Как и в Go, не нужны лишние скобки, так как операция преобразование типа — это не бинарное выражение, а унарное постфиксное: `тек(:ИмяКласса).поле`.

Для низкоуровневых преобразование к записи добавляется ключевое слово: ‘осторожно’, например, преобразование переменной класса к слову: `тек(:осторожно Слово64)`. В подсветке синтаксиса это слово выделено красным, как в примере.

И последнее, замечу, что есть еще проблема, связанная с тем, может ли произойти аварийная ситуация при преобразовании типа, и она тоже не простая, но она выходит за рамки статьи.

Модульность

Модульность — это огромная и сложная тема, которую можно рассматривать с разных сторон. Я коснусь только одной стороны: как устроено взаимодействие между единицами компиляции.

Взаимодействие может быть жестко задано (например, как в языках Modula-2, Oberon, Go), а именно, в каждой единицы компиляции[8] явно определено:

- какие из описанных сущностей могут быть использованы в других единицах компиляции (экспорт)
- сущности из каких единиц компиляции могут быть использованы в ней (импорт)

Во многих языках одно или оба из этих правил не выполняются. Например, в C/C++ нет ни импорта, ни экспорта (хотя можно запретить внешнее использование), и нет прямой связи между используемой сущностью (взятой из файла заголовка) и тем, где это сущность определена. А в Java есть экспорт, но нет импорта (несмотря на наличие конструкции import), так как в коде можно использовать полное имя (fully qualified name) любой сущности, независимо от того, есть ли импорт единицы компиляции, в которой эта сущность определена.

Не погружаясь в детали, скажу лишь, что наличие импорта и экспорта делает более явной структуру кода и способствует пониманию, а значит и энергосбережению. Поэтому, именно этот подход принят в Тривиле.

Недостатки такого подхода тоже есть, но о них я буду писать, когда доберусь до следующих, после Тривиля, языков программирования.

В Тривиле единица компиляции – это модуль:

7 Статьи о следующих языках семейства: Арс и Арвиль будут следующими за текущей серией.

8 В языке Go тип map — это встроенный тип.

```
1  модуль привет
2
3  импорт "стд::вывод"
4
5  вход {
6     вывод.строка("Привет, мир!")
7  }
```

Программа выводит: “Привет, мир!”, используя импортированную библиотеку вывода.

Несколько пояснений:

1. Доступ к экспортированным сущностям импортированного модуля всегда идет через локальное имя модуля (квалифицированный импорт, qualified import), как вывод.строка в примере. В качестве локального имени модуля, по умолчанию, используется последнее имя в строке импорта. В большинстве языков есть понятие неквалифицированного импорта, то есть возможность использовать имя сущности, например, строка, без префикса. Есть случаи, когда такой импорт может быть полезен, например, для реализации DSL (Domain-specific languages), но это плохо влияет на понимание текста, и, следовательно, на энергосбережение.

2. Код модуля в Тривиле (как, например, в Go) может быть размещен в нескольких исходных файлах. Это существенно увеличивает возможность структуризации текста, так как позволяет сгруппировать семантически связанные описания в одном файле, и, как следствие, улучшить понимание и энергосбережение. Дополнительные усилия в компиляторе (при правильной архитектуре) пренебрежимо малы.

Промежуточное заключение

Задачей этой статьи было дать читателю почувствовать подход к разработке языка программирования, основанный на предварительно определенных целях и требованиях. Другими словами, это попытка продвинуться в сторону методики разработки языков программирования.

Естественно, что в статье показана работа далеко не над всеми частями языка. Существенной частью незатронутой работы был подбор терминологии, ключевых слов и стандартных имен. Эта часть работы занимала много времени, так как русская терминология и русские ключевые слова не являются устоявшимися, особенно для современных конструкций, таких, так *может-быть* типы и оператор надо. Цель автора была подобрать слова, которые

- были бы интуитивно понятны
- соответствовали семантике
- и вписывались в общую стилистику языка

В ходе этой работы, многие часы были затрачены на подбор синонимов и эксперименты с синтаксисом. Безусловно, работа по подбору терминологии, синтаксиса и ключевых слов будет продолжена во время работы над следующими языками семейства.

Описание разработки Тривиля на этом не заканчивается, в продолжении серии будут разобраны сложные случаи, в которых полное или традиционное решение противоречило требованию минимизации объема работы и приходилось искать баланс между полнотой (выразительностью) конструкции и простотой ее реализации. Первый пример такой работы уже был дан в разделе про оператор выбора.

Кроме того, будет обоснована архитектура компилятора, описана последовательность и график разработки, уделяя особое внимание тому, что позволило разработать язык, компиляторы и библиотеки в короткое время с минимальными усилиями.

Литература

[1] Недоря А. Е. Разработка языка Тривиль. Первые шаги к семейству языков. Часть 1, 2024. http://digital-economy.ru/stati/razrabotka-yazyka-trivil-pervye-shagi-k-semeystvu-yazykov-chast-1

[2] Недоря А.Е. Разработка типовой системы языка программирования приложений, Открытая конференция ИСП РАН, 2019.
http://алексейнедоря.рф/wp-content/uploads/2022/10/Разработка-типовой-системы-для-ворчалок.pdf
[3] Niklaus Wirth, Compiler construction, Addison Wesley, 1996. ISBN 0-201-40353-6
[4] Diagram to Rust and language compiler design.
https://www.mo4tech.com/diagram-to-rust-and-language-compiler-design-part-1-rust-compilation-process-and-macro-expansion.html
[5] Язык программирования Тривиль.
https://gitflic.ru/project/alekseinedoria/trivil-0/blob?file=doc%2Fтривиль%2Fописание%2Freport.pdf
[6] Тривиль: публичный репозиторий. https://gitflic.ru/project/alekseinedoria/trivil-0
[7] H.Moessenboeck, N.Wirth The Programming Language Oberon-2, Institut fur Computersysteme, ETH Zurich, July 1996.
[8] Пароджанов В. Д. Умеет ли человечество писать алгоритмы? Безошибочные алгоритмы и язык ДРАКОН, 2021. https://habr.com/ru/articles/537294/#Визуальное%20структурное%20программирование

Спецификации языков программирования

[Go] The Go Programming Language Specification. https://go.dev/ref/spec
[Kotlin] Kotlin language specification. https://kotlinlang.org/spec/kotlin-spec.html
[Oberon] The Programming Language Oberon. https://people.inf.ethz.ch/wirth/Oberon/Oberon.Report.pdf
[Rust] The Rust Reference. https://doc.rust-lang.org/reference/
[Swift] The Swift Programming Language.
https://docs.swift.org/swift-book/documentation/the-swift-programming-language/

References in Cyrillics

[1] Nedorya A. E. Razrabotka yazy`ka Trivil`. Pervy`e shagi k semejstvu yazy`kov. Chast` 1, 2024.
http://digital-economy.ru/stati/разработка-языка-тривиль-первые-шаги-к-семейству-языков-часть-1
[2] Nedorya A. E. Razrabotka tipovoj sistemy` yazy`ka programmirovaniya prilozhenij, Otkry`taya konferenciya ISP RAN, 2019.
[5] Yazy`k programmirovaniya Trivil`.
https://gitflic.ru/project/alekseinedoria/trivil-0/blob?file=doc%2Fтривиль%2Fописание%2Freport.pdf
[6] Trivil: publichny`j repozitorij. https://gitflic.ru/project/alekseinedoria/trivil-0
[8] Parondzhanov V. D. Umeet li chelovechestvo pisat` algoritmy`? Bezoshibochny`e algoritmy` i yazy`k DRAKON, 2021.
https://habr.com/ru/articles/537294/#Визуальное%20структурное%20программирование

Недоря Алексей Евгеньевич, к.ф.-м.н.
ORCID 0000-0001-8998-7072
aleksei.nedoria@yandex.ru
Телеграмм канал: t.me/vorchalki_o_prog

Ключевые слова

язык программирования, семейство языков программирования, разработка языков программирования, типовая система, компилятор, архитектура компиляторов, энергосбережение разработчика

Aleksei Nedoria, Development of programming language Trivil. Part 2.

Keywords

programming language, family of programming languages, programming language development, type system, compiler, compiler architecture, developer energy saving

Abstract

The article is the second in a series of articles that describe the development of the Trivil programming

language. The first article described the first stage of language development: defining goals, requirements and selection criteria. This article provides a rationale for the main language solutions. Like the entire series, the article is aimed, for the most part, not at the programmer who uses the language, but at the developer of programming languages.