УДК: 004.43, 004.4'4

1.4. Разработка языка Тривиль. Часть 4. Реализация

А. Е. Недоря, г. Санкт-Петербург

Статья является заключительной в серии статей, описывающих разработку языка программирования Тривиль и его реализацию. В предыдущих статьях речь шла о разработке собственно языка. Эта статья посвящена реализации языка. В ней рассматривается архитектура компиляторов, ход разработки и влияние решений на трудоемкость и скорость разработки. В статье также описывается область применимости языка.

Введение

В первых трех статьях серии были описан, с некоторой степенью детализации, язык Тривиль. Основное внимание уделялось тому, как из целей и требований к языку, определенных в [1] выводились конструкции языка и ограничения.

Основная идея, которая привела к разработке Тривиля — это переход к интенсивному программированию [4], путь к достижению которого я вижу через разработку семейства языков программирования и технологии архитектурного программирования.

Впрочем, интенсивное программирование тоже не самоцель, а средство для создания торгово-промышленно-финансового интернета и в конечном счете, для экспансии человечества в космос. Создание Тривиля — это первый шаг в этом направлении.

Следующий шаг уже начат, но еще не завершен. В следующих статьях я начну описывать то, что уже понято и сделано. Впрочем, Тривиль оказался полезным сам по себе и принес неожиданные результаты, об этом поговорим в завершающей части статьи.

Разработка языка программирования не имеет смысла без его реализации, а именно, без создания инструментария, позволяющего выполнить программу, написанную на этом языке.

В минимальный инструментарий входит компилятор, библиотека и среда исполнения. Без такого минимума язык является, в лучшем случае, набором идей или математической абстракцией.

Для широко используемого языка инструментарий должен быть существенно шире минимального, упомяну интеграцию со средами разработки (IDE), системы сборки, возможность подключения кода на других языках (interoperability), тесты на соответствие компилятора языку (compliance test suite) и так далее. Кроме того, нужна документация, включая спецификацию языка, описание библиотек, руководства разного уровня, а также презентационные материалы, в состав которых должен входить онлайн-компилятор (playground) и многое другое.

Требования к инструментарию и документации приводят к тому, что для разработки промышленного языка необходимо затратить сотни человеко-лет разработчиков разных специализаций (специалисты по языкам, компиляторам, библиотекам, тестированию, интеграции, документации, внедрению и т.п.). Для примера укажу время разработки нескольких современных языков, начиная от объявленного разработчиками языка времени начала разработки до выхода первой (1.0) версии языка и инструментария:

- Go: 2007 - 2012 (около 5 лет),
- Swift: 2010 - 2014 (около 4 лет),
- Kotlin: 2010 - 2016 (около 6 лет).

При взгляде на эти цифры может показаться, что разработка языка малыми усилиями и в короткое время невозможна. Это действительно так, если речь идет о промышленном языке, в разработке которого всегда приходится учитывать множество конфликтующих требований (в том числе совместимости со старым кодом и другими языками), приходящих от разных заинтересованных сторон.

Ситуация существенно меняется, если речь идет об исследовательской работе, в рамках которой можно сконцентрироваться на главном, отбросить второстепенное и приложить максимальные усилия к упрощению языка и инструментов.

Разработка языка Тривиль началась 20 ноября 2022, а 9 сентября 2023, через 8.5 месяцев, был завершен принципиальный этап: язык, компилятор (точнее два компилятора) и среда исполнения были готовы к использованию на Windows и Linux. Собственно, точкой было завершение спецификации языка, остальное было сделано немного раньше.

В сентябре же 2023-го студенты Университета Иннополиса под руководством проф. Е.А. Зуева начали использовать язык на курсе разработки компиляторов, подробнее об этом позже.

Надеюсь, что у читателя возник вопрос: как и за счет чего можно ускорить реализацию и вместо сотен человеко-лет затратить несколько человеко-месяцев (хотя, скорее, надо считать в человеко-днях или часах).

Первая часть ответа на вопрос, почему Тривиль (язык и реализация) был сделан быстро и малыми усилиями, очевидна: у меня есть опыт подобных разработок, и я точно понимал, что и как надо сделать. Последовательность шагов разработки была для меня очевидна с самого начала:

1. набросок грамматики языка,
2. первый компилятор Тривиля, одновременно:

- разработка среды исполнения и первых библиотек,
- доработка синтаксиса и семантики языка;

3. черновик описания языка,
4. компилятор Тривиля на Тривиле (bootstrap), одновременно:
   - доработка среды исполнения и библиотек,
   - доработка языка, включая учет потребности библиотек,
   - доработка компилятора на Go;
5. описание языка.

Вторая часть ответа тоже проста: я осознавал свою ограниченность, или, говоря словами Эдсгера Дейкстры: осознавал, что я "смиренный" программист. Моей постоянной мыслью была мысль: как сделать проще, как пройти коротким путем, но при этом достичь цели.

На мой взгляд, практически вся информатика (computer science) строится на принципиально неверном философском подходе, который хорошо выразил Гради Буч [5]: "*...сложность, о которой мы говорим, является, по-видимому, необходимым свойством всех больших программных систем. Под необходимостью мы имеем в виду следующее: можно создать эту сложность, но нельзя придумать так, чтобы обойтись без нее.*"

Другими словами: "*что тут думать, трясти надо!*". И это проявление лености ума: давайте не будем решать задачу, потому что мы не знаем, как ее решить. Об этом стоило бы поговорить подробнее, но это, скорее, тема для следующих статей. Пока же напомню о другой школе: *make it as simple, as possible, but not simpler*[1]. И если мы прямо сейчас не знаем и не умеем делать проще, то надо сделать, как можем и думать дальше. И еще: *Мы сами знаем, что она [задача] не имеет решения [...] Мы хотим знать, как её решать*[2]. Подробнее см. [6].

Собственно, три первые статьи описывают то, как сделать простой, но достаточный язык. Тот же самый подход был применен к реализации компиляторов, библиотек и среды исполнения.

Два компилятора

В плане разработке (см. выше) указано, что надо разработать два компилятора. Первый, очевидно, надо писать не на Тривиле, так как Тривиль–компилятора еще нет.

Естественный вопрос: зачем писать второй компилятор? Я не буду отвечать на этот вопрос в общем, раскрутка компилятора (bootstrapping) давно применяется и полезность ее хорошо описана. Что же касается Тривиля, то разработка второго компилятора является необходимой, так как язык позиционируется как язык для разработки компилятора. Компилятор Тривиля на Тривиле является обязательной проверкой пригодности языка для целевой области.

Остается выбрать язык для первого компилятора. Этот выбор для меня был простым. Я уже писал компиляторы на Go, к тому же устройство программы на Тривиле похоже на устройство Go-программы. Это позволило использовать одинаковую архитектуру для обоих компиляторов.

Кроме того, несмотря на существенную идеологическую разницу языков, например, в подходе к безопасности ссылок и поддержки ООП, я понимал, как писать текст на подмножестве Go, так чтобы он был близок к тексту на Тривиле. Близость исходных текстов позволила на этапе разработки второго компилятора написать простую программу преобразования текста Go ⇒ Тривиль и существенно сократить трудоемкость.

Уточнение задачи компилятора

В общем виде задачу компилятора можно сформулировать так: по набору исходных текстов построить образ программы, готовый к исполнению.

Например, запуская clang hello.c на корректном исходном файле, мы получим исполняемую программу. Но если hello.c использует функцию, определенную в другом исходном файле, например, aux.c, то запуск уже должен быть clang hello.c aux.c, иначе используемая функция не будет найдена. Дело тут в том, что в языках семейства C/C++ устройство программы не определено на уровне языка и задается внешним образом, например, через перечисление исходных файлов в командной строке или в makefile.

В тех же языках, в которых устройство программы задается на уровне языка, например, в Go, для компиляции программы, состоящей из одного файла (пакета, модуля) или множества пакетов, достаточно указать головной модуль программы, например, команда go build tric.go компилирует целиком компилятор Тривиля, написанный на Go.

Тривиль, как и Go — это модульный язык с заданным (через импорт) устройством программы, что позволяет компилировать программу целиком.

Теперь мы можем уточнить задачу компилятора при компиляции программы, разбив ее на подзадачи:

- собрать список всех модулей, составляющих программу,
- скомпилировать каждый из этих модулей (в правильном порядке),
- собрать исполняемую программу.

Разбиение на подзадачи почти позволяет перейти к архитектуре компилятора, но надо уточнить еще один вопрос, существенно влияющий на сложность компилятора.

---

1 Цитата А.Эйнштейна, использованная Никлаусом Виртом как лого языка Oberon.

2 Цитата из "Понедельник начинается в субботу", А. и Б. Стругацкие

Раздельная компиляция

Под **раздельной компиляцией** понимается возможность использовать при компиляции программы результат предыдущих компиляций частей программы (модулей или исходных файлов). Раздельная компиляция является обязательной для работы с большими программными системами, так как компиляция всех исходных текстов может занимать существенное время. Раздельная компиляция позволяет при повторной компиляции обрабатывать только те модули, которые изменились (прямо или косвенно), и существенно ускоряет процесс компиляции.

Для языков без импорта, таких как C/C++, раздельная компиляции не требует никакой поддержки компилятора, проверка совместимости сущностей программы происходит при сборке (линковке). Такой вариант раздельной компиляции обычно называется **независимой компиляцией**.

Для модульных языков поддержка раздельной компиляции является непростой задачей и требует проверки совместимости модулей. Например, если модуль А импортирует Б и модуль Б был изменен, то в общем случае нельзя использовать результат предыдущей компиляции как модуля Б, так и А, их надо компилировать заново.

Алгоритм определения минимального набора модулей, которые надо компилировать, является непростым, при этом его сложность существенно зависит от требований. Например, пусть в модуль Б была добавлена функция, которую модуль А не использует. Можно ли при этом использовать результат предыдущей компиляции А? Это далеко не простой и не единственный возникающий вопрос.

Но даже если использовать самый простой способ определения минимального набора модулей, компилятор, поддерживающий раздельную компиляцию, должен уметь записывать информацию об экспортированных сущностях модуля и читать её быстро. Иначе никакой выгоды от раздельной компиляции не будет. Задача сериализация/десериализация сложной структуры данных весьма нетривиальна.

Итак, поддержка раздельной компиляции в компиляторе требует существенных усилий, но является ли она необходимой для Тривиля?

На мой взгляд, нет. Компилятор Тривиля не является большой программной системой, и полная компиляция всех его модулей должна происходить быстро (и это подтверждено практикой).

Отказ от раздельной компиляции – это еще один пример упрощения, только теперь не языка (см. [3]), а компилятора.

Замечу, что для следующих языков семейства раздельная компиляция является необходимой и обязательно будет сделана, но сначала до этих не-Тривиль языков надо добраться.

Архитектура компилятора

После уточнения задачи компилятора можно перейти к рассмотрению архитектуры компилятора. На верхнем уровне компилятор состоит из двух логических частей, первая обрабатывает программу целиком, вторая работает с модулем.

Компилятор всегда вызывается на головном модуле программы и работает схематически так:

1. выполняет синтаксический анализ головного модуля и рекурсивно всех модулей, которые импортирует каждый обработанный модуль;
2. строит упорядоченный список модулей, начиная с тех модулей, которые ничего не импортируют и заканчивая головным модулем;
3. для каждого модуля из этого списка выполняет семантический анализ и генерацию;
4. собирает исполняемую программу.

Как видно из этого описания, сначала выполняется синтаксический анализ для всех модулей программы, при этом лексический анализ является составной частью синтаксического. После завершения синтаксического анализа программа представлена в виде линейного списка модулей, где каждый модуль представлен своим абстрактным синтаксическим деревом (АСД). При этом, если исходно модуль состоит из нескольких файлов, то на этом уровне для каждого модуля уже выполнено объединение АСД-файлов в АСД-модуля. Во время объединения также выполняется настройка обобщенных модулей, и далее на стадии семантического анализа и генерации обобщенных модулей не существует, что существенно упрощает компилятор.

Для АСД каждого модуля выполняется последовательно несколько проходов семантического анализа и проход генерации выходного языка.

Семантический анализ

Основная задача семантического анализа — проверка корректности конструкций языка, которая разбивается на две подзадачи, каждая из которых решается на отдельном проходе:

- разрешение имен и областей видимости,
- контроль типов.

Семантический анализ получает на вход АСД после синтаксического анализа и дорабатывает его.

Результатом первого прохода для корректного модуля является АСД, в котором для каждого использования идентификатора указан объект, который идентификатор обозначает в точке использования, а это может быть:

- предопределенный объект, например, тип Строка или константа истина,
- объект (тип, константа или функция), описанный в данном модуле в одной из объемлющих точку использования областей видимости,

- или импортированный объект.

На втором проходе проверяется совместимости типов для всех операций языка, кроме того, в случае некоторых неявных преобразований АСД достраивается, чтобы преобразование стало явным и было представлено узлом в АСД.

Разбиение семантического анализа на два последовательных прохода позволяет существенно упростить компилятор. Во-первых, за счет того, что решение одной задачи проще, чем решение двух (single responsibility principle), во-вторых, контроль типов всегда получает на вход АСД, в котором нет ошибок, связанных с именами.

Архитектурно семантический анализ сделан так, чтобы в него легко можно было добавить дополнительные проходы, например, оптимизирующие проходы.

Генерация

Задача генерации — отобразить АСД на выходной язык, который, в общем, может быть кодом процессора, байт-кодом виртуальной машины, промежуточным языком или языком программирования.

С точки зрения минимизации усилий генерация кода конкретного процессора неприемлема, так как ограничивает переносимость компилятора и программ. Генерация байт-кода также не лучший вариант, так как добавляет существенную зависимость от виртуальной машины.

По сути, два варианта, которые имеет смысл рассматривать — это генерация LLVM IR или текста на языке C. Оба этих варианта обеспечивают:

- переносимость,
- качество кода, за счет оптимизаций на уровне LLVM или C,
- относительную простоту генерации.

При этом генерация кода на C проще, так как уровень языка выше и не нужно учитывать специфику LLVM IR, а учитывать её в некоторых местах довольно сложно, как показывает мой опыт генерации LLVM IR.

Замечу, что я не рассматриваю генерацию на другие языки программирования, язык C — это единственный из распространенных языков, который является переносимым ассемблером и занимает промежуточное место между языками высокого и низкого уровня. Писать программы на C не полезно, но при использовании в качестве промежуточного языка многие его недостатки как языка программирования становятся достоинствами. Например, отсутствие проверки индекса массива является недостатком для языка программирования, но достоинством для промежуточного языка, так как позволяет корректно отобразить в коде семантику исходного языка.

По совокупности генерация кода на C существенно выигрывает в простоте реализации и удобству отладки по сравнению со всеми другими вариантами. Для генерации используется стандарт C99, а для получения исполняемой программы, обычно, компилятор clang.

Устройство компилятора

Теперь можно перечислить модули, из которых состоит компилятор[3]:

- трик: головной модуль, обработка командной строки, запуск компиляции или других действий;
- асд: описание классов абстрактного синтаксического дерева;
- компилятор: компиляция программы,
- парсер: синтаксический анализ,
  - лексер: лексический анализ;
- семантика,
  - имена: разрешение имен,
  - контроль: контроль типов;
- генерация,
- основа: обеспечение работы, включая чтение/запись файлов и обработку ошибок.

Устройство простое и, на мой взгляд, идеальное, полученное исходя из опыта работы над предыдущими компиляторами. Основное здесь — это последовательное применение принципа единственной ответственности (single responsibility principle). При этом модули достаточно крупные, дробление их на более мелкие привело бы к увеличению количества связей и увеличению сложности программирования и сопровождения.

Компилятор и ООП

Если спросить меня, в какой парадигме написан компилятор, то я скажу — в ООП. Но если приглядеться, то это весьма странный ООП, который кто-то вообще не признает за ООП.

Рассмотрим АСД. Оно состоит из объектов классов, и все классы являются наследниками класса Узел. Вот сокращенная диаграмма классов, корень которого класс Узел:

Узел

- Тип
  - ТипВектор
  - ТипКласс
  - *другие типы*

3 Как уже упоминалось, оба компилятора, компилятор на Go и компилятор на Тривиле, архитектурно одинаковы.

- Описание
  - ОписаниеКонстанты
  - ОписаниеФункции
  - *другие описания*
- Оператор
  - ОператорПрисвоить
  - ОператорЕсли
  - *другие операторы*
- Выражение
  - *разные выражения*

ООП как есть. Но вот в чем вопрос: а что с **инкапсуляцией**? На каком уровне нужна инкапсуляция? Ответ: *каждый класс должен быть защищен от других* - не подходит. Любой из этих классов отдельно не имеет смысла. АСД — это одна структура данных, состоящая из объектов многих классов, ставить защиту между ними так же бессмысленно, как между полями одного класса.

Тогда может быть нам нужна *инкапсуляция на уровне модуля*? Опять мимо, АСД — это структура, которая обрабатывается всеми проходами компилятора. Естественно сравнить АСД с заготовкой, которую двигают по конвейеру и последовательно обрабатывают разными станками. Если станок не должен изменять какие-то части заготовки, то это настройка станка, а не заготовки. К тому же, на конвейер можно поставить дополнительный станок (проход компилятора). Неужели из-за этого надо менять заготовку? Это задача станка, и это снова принцип единственной ответственности.

Может быть, третий ответ: если у класса есть метод, для его работы нужны дополнительные данные, то доступ к ним имеет смысл контролировать. Вроде бы разумно, но давайте посмотрим, какие это могут быть методы.

Мы знаем, что узел АСД последовательно обрабатывается проходами, например, семантическими. Кажется правильным добавить, например, метод проверить тип для каждого узла выражения. Это же здорово, проход контроля типов просто вызывает этот метод на каждом узле и готово.

Но что мы тогда получим:

- мы переместили весь сущностный код проверки типов в описание классов и **потеряли инкапсуляцию** на уровне прохода: теперь все проходы имеют доступ к проверке типа;
- каждый класс имеет собственный метод, хотя для многих операций, например, для бинарных операций “+” и “-” проверка одинаковая, и это приводит нас к дублированию кода или к необходимости хитрить, чтобы не дублировать;
- если для проверки типа нам надо выйти за пределы одного узла, то нам придется снова хитрить.

Но это были мелкие неудобства. Если мы добавили метод (или методы) для проверки типа, то естественно добавить методы для других семантических проверок, а также для генерации кода и для других дополнительных действий. Например, Тривиль позволяет записать АСД в виде списочного выражения (S-expression), чтобы его можно было удобно визуализировать. А еще, выдать экспортированные сущности модуля в виде, удобном для чтения. Оба этих действия тоже работают с АСД. Если мы будем последовательны, нам придется добавить десятки методов в каждый класс. Это приведет к тому, что практически весь код компилятора окажется в модуле асд, и мы получим то, что называется *спагетти-код*.

Как видно, неверный подход и ложно понятая инкапсуляция приводит к программному кошмару и к полной потере инкапсуляции. А какая инкапсуляция нам нужна?

Если задуматься, ответ очевиден: нам нужна инкапсуляция на уровне прохода (обрабатывающего станка). Каждый проход решает свою задачу, использует свои алгоритмы для работы, и ему нужны свои локальные данные. Никакие **локальные данные прохода не должны попасть в АСД**, даже если это кажется удобным и увеличивающим скорость работы. Джентльмен в контроле типов не отвечает за то, что делает джентльмен в генерации. И не только не отвечает, но и ничего не знает об этом.

Такая изоляция позволяет менять код прохода независимо от других проходов, добавлять дополнительные проходы, менять одну генерацию на другую, то есть превращает компилятор в конструктор (или конвейер) из компонент.

Какие же тогда методы есть смысл делать у классов, описывающих узлы АСД? Наверно те, которые не связаны с обработкой, а отображают некоторые свойства объектов. Сколько таких объектов есть в АСД Тривиля? Ответ: **ни одного**.

В модуле асд описаны 64 класса, и ни у одного из них нет методов. Замечу, что если бы мне понадобился какой-то полезный метод, я бы его добавил, но ни разу не понадобилось.

Соответственно, вторая священная корова ООП — **полиморфизм**, в данном месте «сдохла». Нет методов, нет полиморфизма. А вот **наследование** используется как инструмент структурирования и дополнительного контроля типов, позволяющий разделить разные сущности, такие как описания, операторы и выражения.

Еще раз, методы нужны, но там, где они уместны. Например, каждый проход компилятора определяет объект, содержащий локальные данные прохода и методы, обрабатывающие АСД и отдельные узлы. Здесь методы есть, инкапсуляция есть (на уровне модуля), а наследования нет, и полиморфизма тоже нет.

На мой взгляд, обычная реализация полиморфизма, а именно: возможность переопределить метод и использовать метод базового типа (супертипа) — это в 99% случаев архитектурная ошибка[4]. Правильное использование полиморфизма — это реализация абстрактного метода. Допустимое использование — это новая реализация метода в подтипе, не зависящая от реализации в супертипе.

Тривиль разрешает переопределение метода, но не позволяет вызывать метод супертипа. На мой взгляд, это правильный ООП, который надо дополнить средствами утиной типизации, но это другая тема.

Ход и итоги разработки

В этой и в предыдущих статьях много раз говорилось об упрощении языка и компилятора. Остается показать, как эти упрощения сказались на ходе разработки. Длительность этапов в часах рассчитывается по средней нагрузке 8 часов за неделю (приблизительный расчет).

Дата	Событие	Этап и длительность
20.11.2022	Начало разработки	● Цели и требования ● Черновая грамматика 2 недели/16 часов
02.12.2022	Выложен первый исходный текст компилятора на Go (первый коммит)	● Разработка компилятора на Go ● Разработка среды исполнения ● Разработка первых библиотек 9 недель/72 часа
03.02.2023	Компилятор Тривиля на Go компилирует все основные конструкций, код тестов выполняется.	
19.02.2023	Начало написания спецификации языка (черновик)	● Разработка черновой спецификации ● Доработка компилятора 3 недели/24 часа
16.03.2023	Выложен первый исходный текст компилятора на Тривиле (первый коммит)	● Разработка компилятора на Тривиле ● Доработка компилятора на Go ● Разработка библиотек, необходимых компилятору 14 недель/112 часов
25.06.2023	Компилятор на Тривиле компилирует сам себя. Достигнута неподвижная точка (см. ниже)	
07.09.2023	Спецификация языка версии 0.9 завершена	● Доработка спецификации, компиляторов, среды исполнения и библиотек ● Перенос на другие платформы[5] 10 недель/80 часов

Поясню, что понимается в таблице под **неподвижной точкой**. Применяем компилятор на Go к исходным текстам компилятора на Тривиле, получаем набор исходных текстов на языке Си. Из этих исходных текстов собираем исполняемый код компилятора (используем clang). Новым компилятором компилируем те же исходные тексты, получаем второй набор исходных текстов на языке Си. Сравниваем эти исходные тексты. Если они полностью совпадают, то это и есть неподвижная точка. Сколько бы раз мы ни компилировали вновь собранным компилятором его же исходные тексты, результат не изменится.

Достижение неподвижной точки является существенным признаком работоспособности и качества компилятора, так как компилятор для самого себя является достаточно большим и сложным тестом.

4 а в оставшихся случаях, это грубая архитектурная ошибка.

5 Перенос на Linux, FreeBSD, MacOS выполнен Дмитрием Соломенниковым и Иваном Тюляндиным

Замечу, что после достижения неподвижной точки, разработка компилятора на Go была практически заморожена (кроме исправления ошибок), а доработка и улучшения компилятора на Тривиле продолжилась. Так что сейчас неподвижной точки уже нет.

Приведу размеры исходных текстов на начало сентября 2023 года:

Программная часть	**Язык**	**Размер в строках**
Компилятор на Go	Go	11,200
Компилятор на Тривиле	Тривиль	10,500
Среда исполнения	C99	1,800
Библиотеки	Тривиль	2,300

Итого, меньше чем за 9 календарных месяцев, минимальными усилиями (8 часов в неделю), написаны два компилятора, среда исполнения и библиотеки, суммарно 25 тысяч строк на трех языках программирования [6].

На мой взгляд, это существенное подтверждение мысли, что упрощение работает и делает невозможные вещи не просто возможными, а обозримыми и выполнимыми.

Понятно, что для широкого или коммерческого использования надо еще много сделать, но то, что уже сделано — это не MVP (minimum value product), это не прототип на выброс, а рабочий продукт, который активно используется для разработки следующих языков.

Области применения

Основная область применения Тривиля очевидна и совпадает с заявленной целью языка: разработка компиляторов. Язык и компилятор используются для разработки следующих языков семейства, языков Арс и Арвиль. О них я буду писать в следующих статьях.

Вторая область стала видна уже в ходе разработки, это **полигон для обучения** студентов разработке языков, компиляторов, инструментов (например, статических анализаторов), библиотек и т.д. Простота языка и компилятора позволяет в рамках студенческого курса сделать практическую работу по множеству направлений, подробнее см. [8].

На практике использование Тривиля было проверено в 2023-2024 учебном году в Университете Иннополиса. В рамках курса по разработке компиляторов под руководством проф. Е.А. Зуева группы студентов делали генерацию кода для Тривиля в LLVM IR, JVM и .NET, и еще одна группа сделала набор тестов на соответствие компилятора спецификации языка. Все группы успешно выполнили учебную задачу, и хотя ни одна из групп не добилась включения результатов работы в основной репозиторий Тривиля, но такая задача перед ними не ставилась.

Заключение

Разработка Тривиля была (и есть) большим экспериментом, попыткой “впихнуть невпихуемое” и прорваться к звездам. Эксперимент продолжается, и о его развитии будут следующие статьи.

Моя огромная благодарность всем, кто участвовал, критиковал, предлагал решения и поддерживал меня на этом пути. Отдельная благодарность: Е. Зуев, А. Канатов, Д. Соломенников, И. Туляндин, А. Чесноков, Н. Шилов. А также всем участникам семинара STEP[6] [9], Университету Иннополиса и журналу “Цифровая экономика”.

Тривиль — это личинка хорошего языка программирования, и из него, при вложении сил, может вылупиться бабочка. Но мы пойдем другим путем.

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 языка Тривиль. Часть 2, 2024. http://digital-economy.ru/stati/razrabotka-yazyka-trivil-chast-2
3. Недоря А. Е. Разработка языка Тривиль. Часть 3. Баланс, 2024. http://digital-economy.ru/stati/razrabotka-yazyka-trivil-chast-3-balans
4. Недоря А. Е. Интенсивное программирование, 2022. http://digital-economy.ru/stati/intensivnoe-programmirovanie

6 Семинар по фундаментальным вопросам программной инженерии, теории и экспериментальному программированию (Software Engineering, Theory and Experimental Programming)..

5. Буч Г. Объектно-ориентированное проектирование с примерами применения, М.: Конкорд, 1992.
6. Непейвода, Н. Н. Методы Кристобаля Хунты, https://ru.wikibooks.org/wiki/Методы_Кристобаля_Хунты
7. Тривиль: публичный репозиторий. https://gitflic.ru/project/alekseinedoria/trivil-0
8. Недоря А. Е. Язык программирования для обучения технологиям компиляции и трансформации. Труды Института системного программирования РАН. 2023;35(6):95-102. https://doi.org/10.15514/ISPRAS-2023-35(6)-5
9. Семинар STEP, https://persons.iis.nsk.su/en/STEP-2023, https://persons.iis.nsk.su/en/STEP-2024

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 yazy`ka Trivil`. Chast` 2, 2024. http://digital-economy.ru/stati/разработка-языка-тривиль-часть-2
3. Nedorya A. E. Razrabotka yazyka Trivil'. Chast' 3. Balans, 2024. http://digital-economy.ru/stati/разработка-языка-тривиль-часть-3-баланс
4. Nedorya A. E. Intensivnoe programmirovanie, 2022. http://digital-economy.ru/stati/интенсивное-программирование
5. Buch G. Ob"ektno-orientirovannoe proektirovanie s primerami primeneniya, M.: Konkord, 1992.
6. Trivil: publichny`j repozitorij. https://gitflic.ru/project/alekseinedoria/trivil-0
7. Nepejvoda, N. N. Metody Kristobalya Hunty, https://ru.wikibooks.org/wiki/Методы_Кристобаля_Хунты
8. Nedorya A. E. Yazyk programmirovaniya dlya obucheniya tekhnologiyam kompilyacii i transformacii. Trudy Instituta sistemnogo programmirovaniya RAN. 2023;35(6):95-102.

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.

Aleksei Nedoria, Development of programming language Trivil. Part 4. Implementation.

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

DOI: 10.34706/DE-2025-01-04

JEL classification:

Abstract
The article is the final in a series of articles describing the design of the Trivil programming language and its implementation. In previous articles, we talked about the development of the language itself. This article is about the implementation of the language. It examines the architecture of compilers, the course of development, and the impact of solutions on the complexity and speed of development. The article also describes the scope of the language.