УДК: 004.932, 517.5

1.5. Разработка языка Тривиль. Первые шаги к семейству языков. Часть 1

А. Е. Недоря, г. Санкт-Петербург, Россия

Это первая статья из серии статей, в которых описывается разработка языка программирования Тривиль: от рассуждения о необходимости разработки нового языка, определения целей и требований и до выбора и обоснования конкретных языковых решений. Статьи, в основном, нацелены не на программиста, который использует язык, а на разработчика языков программирования. В статьях автор использует опыт лекций о разработке языков в МФТИ, ИТМО и Университете Иннополиса.

Введение

В статье «Интенсивное программирования» [1] я рассматривал переход от экстенсивного к интенсивному программированию и в качестве одного из шагов перехода упоминал необходимость разработки семейства языков программирования. Уточню еще раз: не языка, а семейства языков, так как один язык не может покрыть все целевые области, удовлетворить требования разработчиков и стилевые предпочтения.

Эта статья описывает первые шаги разработки и реализации семейства языков. Основное внимание в ней я буду уделять не описанию того, что было сделано, а той цепочке рассуждений, которая привела к выбору решений и их реализации.

Почему я считаю это принципиально важным? Больше 40 лет назад, когда я писал свой первый компилятор, основными моими учебниками были книга Гриса «Конструирование компиляторов для цифровых вычислительных машин»[1] и Хартманна «A Concurrent Pascal Compiler for Minicomputers»[2]. Книга Гриса, насколько я помню, была хорошим учебником, типа Dragon Book того времени, а книга Хартманна была совсем другой — в ней автор шаг за шагом описывал разработку конкретного компилятора, указывая, почему то или иное решение было выбрано. Эта книга учила думать! С тех пор я считаю, что «почему» и «зачем» — вот основные вопросы, которые должен задавать себе исследователь и разработчик. Из них следует «что» и «как» делать.

В статье я описываю путь, который был пройден с ноября 2022 года по сентябрь 2023. За это время был разработан сам язык, два компилятора и написана спецификация языка [2].

- **Зачем семейство языков?**

Подробный ответ на этот вопрос выходит за рамки данной статьи, поэтому тезисно:

- современная большая программная система, как правило, содержит части, написанные на разных языках программирования.
- это вызвано тем, что требования для разработки частей системы (таких, как GUI, бизнес логика, системные библиотеки) существенно разные и не могут быть покрыты одним языком без снижения скорости разработки и качества системы.
- использование частей на разных языках приводит к проблемам взаимодействия языков, таким как:
    - разный ABI (application binary interface);
    - разные типовые системы;
    - разное размещение данных в памяти;
    - разный подход к управлению памяти (сборка мусора, счетчики ссылок, ручное управление и т.д.);
    - кроме технологических проблем, есть и бизнес-проблемы, упомяну только необходимость иметь в штате разработчиков, с существенно разным опытом (например, с опытом работы на Rust, C++, Java, Javascript) и квалификацией.

Возможный подход к решению таких проблем: надо разрабатывать не отдельные языки, а семейство совместимых языков, код на которых хорошо взаимодействует. Естественно, что взаимодействие может быть ограничено, например, взаимодействие между языками системного уровня и языком декларативного описания интерфейса, скорее всего, должно быть односторонним.

Не углубляясь в детали, примем за основу следующую постановку задачи: необходимо спроектировать и разработать минимально представительное семейство языков, которое даст подтверждение правильности подхода.

Думаю, что читатель здесь ожидает перехода к определению состава семейства и проектированию языков. Это был бы добротный, академический подход.

1 Д. Грис, Конструирование компиляторов для цифровых вычислительных машин, МИР, Москва, 1975

2 Alfred C. Hartmann, A Concurrent Pascal Compiler for Minicomputers, Lecture Notes in Computer Science (LNCS, volume 50), 1977.

Но я пошел другим путем, путем практического устранения препятствий на пути к намеченной цели. Почему? Просто потому, что в ноябре 2022 года я не мог определить состав семейства языков. Этого слона надо было есть по частям. Теперь, в 2024, пройдя этот путь, я могу перейти к составу семейства, но это тема следующих статей.

Итак, что должно быть сделано, чтобы можно было перейти к проектированию и реализации семейства? Тут мне придется сделать небольшое отступление и поговорить о том, как разрабатываются языки программирования.

Несмотря на то, что на данное время, по разным подсчетам насчитывается от 2.5 до 9 тысяч языков программирования, эта деятельность является, в большей степени, искусством, нежели наукой. Языки создаются методом проб и ошибок. Разработчики изучают то, что было сделано в других языках, предлагают новые конструкции и делают прототипы. Проверяют, получают отзывы, вносят изменения и снова делают прототипы и проверяют.

Получаем несколько парадоксальную ситуацию: чтобы разработать язык, нужно делать прототипы и проверять их, а для этого нужен компилятор. А чтобы делать компилятор, нужно, чтобы язык уже был (на каком-то уровне) разработан.

Тем более, если мы хотим разработать семейство (экспериментальных) языков, нужно уметь быстро делать и переделывать языки и компиляторы. Нужна инструментальная поддержка, но я не знаю ни одной готовой системы, которая могла бы в этом помочь[3].

Выход я вижу такой: написать первый компилятор таким образом, чтобы его части легко переделывались, так чтобы он был основой для следующих компиляторов.

И это приводит нас к следующему вопросу.

Какой язык выбрать для разработки первого компилятора?

Принципиально разных варианта два:

1. взять один из существующих языков;
2. сделать новый специализированный язык

Плюсы использования существующего языка:

- не надо тратить время на разработку промежуточного языка и компилятора.

Плюсы разработки нового языка:

- Такой язык полезен в составе семейства, так как следующие компиляторы тоже надо на чем-то писать.
- Разработка специализированного языка позволяет сделать плавный вход в семейство, так как требования к такому языку мне понятны и разработать его существенно проще, чем следующие языки семейства.
- Минимизация зависимостей и влияния других языков.

О последнем пункте стоит поговорить подробнее. Любой язык, на котором мы программируем, влияет на мышление разработчика. Мы привыкаем к синтаксису, к приемам программирования на этом языке, и, далее, часто неосознанно, вообще не рассматриваем другие подходы и решения. При разработке чего-то принципиально нового стоит поставить себе психологический барьер, который заставляет обдумывать каждое решение и взвешивать за и против.

Если мы рассматриваем язык для разработки компиляторов, как прототип следующих языков семейства, то это убирает и единственный плюс использования существующего языка, так как это не дополнительная работа, а работа непосредственно над семейством языка.

Итак, первый язык семейства — это специализированный язык для разработки компиляторов. Начинаем проектирование с главного: с определения и фиксации целей и требований к языку.

Цели и требования

Сформулируем явно цели разработки языка:

- разработка компиляторов;
- разработка библиотек, нужных для компилятора;
- прототипирование следующих языков семейства, нотации и базовых конструкций.

Перед тем, как перейти к требованиям, надо учесть еще одно важное условие: так как семейство языков я делаю в качестве личного проекта, то ресурсы для разработки ограничены, как правило, я могу вкладывать в эту работу около 8 часов в неделю.

3 А.С. Пушкин

Требования, вытекающие из целей и условий:

	Требование	**Важность**	**Уточнение требований**
1	Продуктивность разработчика	Высокая	●легкость чтения, понимания, написания ●выразительность (есть все необходимые конструкции)
2	Скорость разработки	Высокая	●быстро работающий компилятор и другие инструменты ●минимизация поиска/исправления ошибок времени исполнения и времени на отладку. Отсюда требование **безопасности языка**
3	Минимизация объема работы	Высокая	●простой язык (минимальный набор конструкции) ●в языке нет ничего, кроме того, что необходимо для целевой области ●простой компилятор (нет сложно реализуемых конструкций)
4	Безопасность языка	Высокая	●безопасность ссылок ●управление памятью: сборка мусора ●модульность
5	Производительность компилятора программ	Низкая	

Как видно, основные требования связаны с тем, чтобы разработать язык и компилятор в приемлемые сроки с минимальными ресурсами. При этом производительность компилятора и создаваемых им программ не является приоритетным требованием. Производительность имеет смысл рассматривать, только если её не хватает. Что же касается скорости работы компилятора, то исходя из требования *нет сложно реализуемых конструкций*, медленно работающий компилятор написать трудно.

В требования заложено некоторое противоречие:

- есть все необходимые конструкции;
- нет ничего, кроме того, что необходимо для целевой области.

Это сделано намеренно, чтобы для каждой конструкции вставал вопрос о ее необходимости. Подробнее разговор об этом будет дальше.

Обратим внимание на требование безопасности языка:

- На мой взгляд, требование *безопасности указателей* (null safety) является обязательным для современного языка. Очевидно, со мной согласны разработчики Rust [3], Kotlin [4] и Swift [5].
- Требование использования *сборки мусора* для управления памятью истекает из того, что сборка мусора является лучшим способом с точки зрения производительности разработчика. Безусловно, реализация сборки мусора является сложной программной частью, но подключение существующих реализаций, таких как Boehm GC [6] или Memory Pool System [7], является достаточно простым.
- Отнесение *модульности* в раздел безопасности может показаться странным, но модульность ведет к инкапсуляции и локализации кода и тем самым к локализации ошибок.

Что в имени тебе моем?[4]

У языка программирования должно быть имя, и оно, обычно, как-то соответствует целям языка, идеям или увлечениям авторов. Вспомним:

- FORTRAN: FORmula TRAnslation;
- PL/I (Programming Language One): *one language to rule them all* ;
- Modula: модульное программирование;
- Oberon: спутник Урана, который сфотографировал “Вояджер-2” в 1986 г.;
- Kotlin: Java – это остров, тогда и у нас тоже остров.

Для меня было важно связать имя языка с его характеристиками и условиями, а именно:

- простой язык;
- просто понимаемый язык;

4 В.И. Гололобов, Б.Г. Чеблаков, Г.Д. Чинин, Описание языка ЯРМО. Машинно-независимое ядро. Новосибирск, 1980.

- нет времени, чтобы делать революцию, отсюда - желательно использование проверенных решений.

Имя **Тривиль (**от *тривиальный*) мне показалось подходящим.

Как показала дальнейшая работа над языком, имя стало важным психологическим ограничителем. Вопрос: *достаточно ли тривиальная (проста, понятна, привычна) эта конструкция* – помогал отбрасывать сложные конструкции, что привело к упрощению и ускорению работы.

- **Что надо проектировать?**

Рассмотрим, какие части языка надо проектировать и какие вопросы возникают при этом. Таблица содержит очевидный и неполный список вопросов, на которые приходится искать ответы разработчику языка:

Часть языка	**Вопросы**
Лексика	● кодировка исходного текста ● ключевые слова: на английском или нет ● использование ‘;’: завершитель (Rust) или разделитель (Go) ● числовые литералы: виды литералов, системы счисления (десятичная, шестнадцатеричная, еще) ● строковые литералы: кодировка, кавычки, escape последовательности ● другие литералы (unicode code point, …) ● знаки операций и пунктуации
Синтаксис	● стиль: C/C++/C#/Java vs. Go, Kotlin, Swift
Типовая система	● набор предопределенных типов ● набор конструируемых типов (user-defined types)
Описания	● список описываемых сущностей ● константы: времени компиляции или времени исполнения или оба ● переменные: изменяемые, неизменяемые
Функции/ Методы	● виды параметров: именованные или нет, значение по умолчанию ● способы передачи параметров ● тип результата
Операторы и выражения	● язык с упором на выражения - почти все есть выражение (Kotlin) или язык с упором на операторы (Go) ● набор операторов ● виды выражения
Модульность	● что такое единица компиляции ● раздельная компиляции
Остальное	● обработка исключительных ситуаций ● отношение язык - стандартная библиотека

Увы, но список не полон не только по числу пунктов, но и *в глубину*: при определении каждой конструкции языка мы должны определить ее семантику (или смысл). Не углубляясь в детали, скажу лишь, что для каждой конструкции надо определить две части семантики:

- **статическую семантику**: корректна ли записанная конструкция с точки зрения правил языка (часто используется термин validity rules);
- **динамическую семантику**: что должно произойти в результате выполнения конструкции.

Для тех, кто нечасто сталкивается с семантикой языков программирования, покажу разницу на примере кода, написанного на двух языках:

Kotlin	if (1 + 1) { /* something */ }
Typescript	if (1 + 1) { console.log(“Yes”) }

В Котлине конструкция является некорректной с точки зрения одного из правил статической семантики, так как выражение 1 + 1 в операторе if должно быть типа Boolean. Для Typescript текст корректен, и результатом выполнения будет вывод строки “Yes” на консоль, то есть с точки зрения динамической семантики языка значение выражения, равное 2, считается истиной, и ветка оператора **if** будет выполнена.

Список в таблице (плюс семантика) является достаточно внушительным и должен вызвать вопрос: Как делать выбор, если есть несколько вариантов решения?

Ответ:

- Во-первых, применяя требования к каждому решению и к каждой конструкции языка. Например, из требования безопасности языка, надо (в примере выше) выбрать подход Котлина, уменьшающий количество потенциальных ошибок.
- Во-вторых, использовать оценку, которую я называю **энергосбережением разработчика**.
- **Энергосбережение разработчика**

Сначала небольшое отступление: мы, разработчики, привыкли думать о производительности (эффективности) программ, о размере занимаемой программой памяти, и о сохранении энергии аккумулятора в мобильном телефоне. Но мы редко думаем об эффективной работе разработчика и об экономии его энергии.

Повторю еще раз мысль, уже высказанную выше в статье: производительность программы нам должна быть интересна только в том случае, если ее не хватает[5]. Вот если ее не хватает, надо вкладывать усилия (прикладывая их конкретно к тому месту, которое недостаточно быстро работает). Например, переписать на другом языке или, что еще лучше - изменить алгоритм на более быстрый. Или изменить архитектуру программы, чтобы этого узкого места вообще не было.

Вот всех остальных случаях достаточно не делать глупостей, то есть не принимать плохих решений. Увы, как правило, все происходит не так. Сначала делается язык с упором на производительность разработчика, у которого плохая производительность by design, потом он используется там, где нужна производительность и приходится вкладывать огромные усилия в оптимизацию. Или сначала делается язык с огромными дырами в безопасности, потом язык используется для программирования критических приложений, а чтобы программы на этом языке как-то работали, язык обкладывается статическими анализаторами и санитайзерами. Как мы с коллегами грустно шутим - масса усилий вложена, чтобы мы получали большие зарплаты.

Безусловно, есть те области, в которых производительность должна быть почти на первом месте, где-то сразу после корректности программы. Но если рассмотреть область исследования и прототипирования, то на первый план должна выходить эффективность разработчика, а для этого его энергосбережение.

На мой взгляд, энергосбережение разработчика включает:

- использование родного языка и естественно читаемые идентификаторы;
- чтение исходного кода слева направо и знаки препинания;
- привычную математическую нотацию;
- очевидность семантики;
- полноту информации при выводе ошибок времени исполнения.

Разберем подробно эти пункты.

Использование родного языка и естественно читаемые идентификаторы

В 1980 году вышел препринт “Описание языка ЯРМО”[6], приведу фрагмент программы из него (стр. 35) с минимальными изменениями, чтобы подчеркнуть суть:

```
общая процедура ПОИСК В ТИ (ДАННОЕ ИМЯ) =
  начало переменные ТЕК ЭЛЕМ;
    вход
   если ОГЛАВЛЕНИЕ[ФУНКЦИЯ РАССТАНОВКИ(ДАННОЕ ИМЯ)] -> ТЕК ЭЛЕМ ≠ 0
      то повторять
```

Я читаю этот текст без усилий, благодаря русской нотации и идентификаторам с пробелами, например: ПОИСК В ТИ (поиск в таблице имен), ДАННОЕ ИМЯ. Меня никак не затормаживают сокращения, как ТЕК ЭЛЕМ - текущий элемент. Использование только строчных букв в идентификаторах выглядит непривычно, возможно, это было вызвано ограничениями устройств ввода-вывода на БЭСМ-6. Обратите внимание на знак неравенства.

А это фрагмент из компилятора Тривиля на Тривиле:

5 Из книги: Идеи, определившие облик информатики/ под редакцией Гарри Р. Льюиса, - М.: ДМК Пресс, 2023.

6 См., например, escape analysis в языке Go.

```
фн найти описание(область: асд.Область, имя: Строка): асд.Описание {
  пусть оп = найти в области(область, имя)
  если оп # пусто {
     вернуть оп^
  }
  основа.добавить ошибку("СЕМ-НЕ-НАЙДЕНО", имя)
}
```

Этот текст ближе к привычной нотации языков программирования, чем текст на ЯРМО. Главные отличия: русская нотация и идентификаторы с пробелами: найти описание, добавить ошибку.

Почему русская нотация? Потому что мы быстрее и с меньшими затратами читаем текст на родном языке, мы привыкли к этому с детства. На мой взгляд, когда мы пишем на английском, мы тратим немного больше энергии, чем англоязычный программист, и это наш недостаток и их преимущество.

Для использования русской нотации надо настроить редактор. Я использую макросы, чтобы вообще не переключаться на английскую раскладку. Это может быть неудобно сначала, но, на мой взгляд, оправдано. Тем более, что читаем программу мы гораздо больше, чем пишем.

Что же касается идентификаторов, попробуйте непредвзято посмотреть на один и тот же идентификатор, записанный разными способами, и оценить трудоемкость чтения:

найти описание	найтиОписание
найти_описание	НайтиОписание

Чтение исходного кода слева направо и знаки препинания

Наша письменная культура приучила нас читать слева направо, и, такое чтение для нас проще и энергоэффективней.

Рассмотрим примеры описания функционального типа на нескольких языках:

C	typedef int (*Operation)(int, int);
Kotlin	typealias Operation = (Int, Int) -> Int
Go	type Operation func(x, y int) int
Rust	type Operation = fn(x: int, y: int) -> int;
Тривиль	тип Операция = фн(x: Цел64, y: Цел64): Цел64

Описание на языке C читается примерно так: *определяется тип Operation, как указатель на функцию с двумя параметрами типа int и результатом типа int*. Обратите внимание на прыжки по тексту. Описания на остальных языках читаются слева направо, без прыжков между частями текста.

Теперь сравним текст на Go и Rust. На мой взгляд, отсутствие в Go знака препинания между именем параметра и типом, а также между списком параметров и типом результата заставляет сделать лишнее усилие, провести дополнительную интерпретацию текста. Текст на Rust я читаю без этого лишнего усилия. Кажется, в Go перебрали с краткостью нотации.

В тексте на Rust есть другой недостаток: знак ‘;’ в конце описания лишний. Большинство современных языков (в том числе Go и Kotlin в примере выше) используют точку с запятой, как необязательный разделитель, который нужен, только если конструкция не заканчивается в конце строки текста.

Пример показывает, что как отсутствие знаков препинания, так и их наличие может приводит к дополнительным усилиям при чтении текста.

Привычная математическая нотация

Рассмотрим еще один простой пример:

C	if (x == 1 && y) {}
Go	if x == 1 && y {}
Тривиль	если x = 1 & y {}

Первое, что мы видим, лишние скобки в C, очевидно мешающие восприятию. Но не это главное, обратим внимание на операцию И (&& - conditional AND) и операцию сравнения. Начиная с языка C, операция И записывается двумя амперсандами. История появления такого значка понятна. Сначала была введена битовая операция &, которая соответствует команде процессора, потом добавлена &&, реализуемая через условный переход.

В логике C все правильно, C – это переносимый ассемблер. Но если задуматься, то операцию && мы используем гораздо чаще, кроме, может быть, специальных областей применения, как вычисление хэш-функций, криптографии, и может быть, низкоуровневой графики.

В итоге, есть традиция (и привычка), которая приводит к тому, что разработчики тратят дополнительное время на запись и чтение кода. Впрочем, есть и другая традиция:

C традиция	**Oberon, Тривиль**
&&, ||	&, |
==	=
!=	#
!	~
=	:=

Кроме И и ИЛИ, есть операция NOT и операции сравнения и присваивания. Заметим, что в привычной школьной математической нотации, например: *x + 5 = 0*, знак = это не присваивание, математическая нотация не императивна. Это знак эквивалентности. Мы в школе учим одно, потом переучиваемся (или, скорее, привыкаем к другому).

Вопрос на засыпку: есть ли кто-то из читателей, кто не делал таких ошибок: if (x = 1 && y) {}, использовав знак =, вместо == и после этого разбираясь с неверным поведением программы?

Не буду долго останавливаться на этом, задам читателям только один вопрос: есть ли какая (ассоциативная или логическая) связь между восклицанием ‘!’ и операцией NOT?

Очевидность семантики и неявная вредная магия

Мысль о том, что очевидная семантика, то есть семантика, понятная с первого взгляда, упрощает чтение и понимание, сама по себе не нуждается в примерах и объяснении.

…мы должны делать (как мудрые программисты, осознающие свои ограничения) все возможное, чтобы сократить концептуальную пропасть между статической программой и динамическим процессом…

Э. Дейкстра, О вреде оператора go to.[7] (1968)

В языках программирования есть множество общепринятых конструкций, которые приводят к тому, что понимание фрагмента текста не очевидно и требует изучения контекста, возможно, что очень обширного. Избежать этого нельзя, например, описание используемого типа может быть в другой единице компиляции, но можно попытаться упростить понимание. Во многом, это задача разработчика, а именно, работа над именами сущностей, правильный выбор типов и т.д.

Здесь же мы поговорим о том, что можно сделать в языке программирования, а точнее, о том, что не надо делать.

Приведу проявление скрытой семантики, такие проявления я называю: *неявная вредная магия* (implicit evil magic).

В примере на Go задается описание

- интерфейса (строки 5-7);
- структуры (строка 9)
- и метода (строки 11-13), добавление которого приводит к тому, что “класс” *SM (указатель на структуру) реализует интерфейс в смысле утиной типизации.

7 Генераторы парсеров (как ANTLR) и генераторы кода (LLVM) покрывают малую и самую несущественную часть работы.

```
 1   package main
 2
 3   import "fmt"
 4
 5   type IM interface {
 6     Method()
 7   }
 8
 9   type SM struct {i int}
10
11   func (x *SM) Method() {
12     x.i = 1
13   }
14   func check(im IM) {
15     if im != nil {
16       im.Method()
17     }
18     fmt.Println("done")
19   }
20
21   func main() {
22     check(nil) // 1
23
24     var S_nil *SM = nil
25     check(S_nil) // 2
26   }
```

Далее

- в строках 14-20 описана функция, которая проверяет, определен ли интерфейс (указатель не равен nil[8]). Если определен, то вызывает метод. В конце функции, выводит строку “done”.
- в теле функции main функция проверки вызывается два раза, один раз с литералом nil (строка 22), второй раз с переменной класса, реализующего интерфейс, которая проинициализирована значением nil (строка 25).

Ожидаемый вывод: две строки “done”. Вместо этого получаем (из Go playground [8]) такое сообщение об ошибке, включая трассировку стека вызовов:

```
done
panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x0 pc=0x48ebe0]

goroutine 1 [running]:
main.(*SM).Method(0x4d3468?)
 /tmp/sandbox3933137847/prog.go:12
main.check({0x4d3448?, 0x0?})
 /tmp/sandbox3933137847/prog.go:16 +0x26
main.main()
 /tmp/sandbox3933137847/prog.go:25 +0x25
```

Результат первого вызова ожидаемый - печатает done, второй приводит к nil pointer dereference в теле метода в строке 12, в которой изменяется поле структуры через нулевой указатель. При этом, метод вызывается из строки 16 после проверки интерфейса на nil. То есть в строке 16 интерфейс не равен nil, и метод из объекта вызвать можно, а в строке 12 объект равен nil, и обратиться к нему нельзя,

Как мы видим, пример простой. Полагаю, что начинающий программист на Go с достаточно высокой вероятностью напишет что-то подобное. По крайней мере, я сам когда-то написал в компиляторе на Go примерно такой же текст и был сильно удивлен.

Описанная проблема известна, причины и средства борьбы с ней легко найти в сети. Я не буду об этом писать, скажу только, что источником этой *неявной злой магии*, как и других аналогичных примеров часто является принципиальная сложность в разработке языков программирования, известная как **взаимодействие конструкций языка** (feature interaction). Речь идет о том, что отдельные конструкции могут быть хорошо продуманы, но их взаимодействие приводит к неожиданным эффектам, в том числе и к таким злым чудесам. Это важная тема, но не входящая в рамки данной статьи.

Добавлю еще, что скрытая семантика может влиять не только на усложнение понимания, но и на производительность, например, если происходит неявное выделение памяти в куче[9].

8 То же самое рассуждение можно применить к размеру кода программы.

9 Большинство современных языков используют null, а Go nil.

Вывод ошибок времени исполнения

Посмотрите на вывод ошибки в playground Go в предыдущей части. Такой вывод очень существенно экономит силы разработчика.

На мой взгляд, если среда исполнения программы на некотором языке не выдает трассировку стека (stack trace) при ошибках, на таком языке не надо писать, слишком много лишних усилий тратится на поиск ошибки.

В заключение об энергосбережении

Привычка свыше нам дана,
Замена счастию она
А.С. Пушкин, Евгений Онегин

Многие меня не поймут, когда я пишу про русскоязычную нотацию, просто потому что у них нет опыта программирования на русском. Я написал свой первый компилятор 40 лет назад на русскоязычном Автокоде Эльбрус (Эль-76) [9], а потом написал еще несколько сотен тысяч строк в среде программирования Вир[10]. Так что для меня русская нотация привычна.

Более того, я спокойно отношусь к любой нотации. Например, в коммерческом языке, который я разрабатываю, используются знаки операций из С традиции, включая круглые скобки вокруг выражений в операторах if и while.

Здесь я говорю именно об энергосбережении, об экономичности работы разработчика (в первую очередь, своей работы). Она складывается из мелочей, из чтения слева направо, простоты чтения идентификаторов, знакомых с детства математических символов. Я не пытался измерить “экономию”, но даже если я вместо часа делаю ту же работу за 55 минут и меньше устаю или могу работать с полной отдачей 4 часа, вместо 3, то игра стоит свеч. Субъективно я могу сказать, что программирование на Тривиле для меня легче и радостней, чем программирование на других языках.

И, второе, о том, что “привычка - замена счастью”. Собственно, в привычках нет ничего плохого, привычки позволяют сосредоточиться на новом и главном. Но, под капот каждой привычки (например, привычной нотации языка программирования) стоит заглядывать и осознанно решать: годится ли она (в современных условиях) или ее стоит заменить на новую.

Любопытно было бы посчитать каким-либо объективным образом затраты энергии разработчика и сравнить языки программирования или хотя бы энергозатраты на чтение кода на разных языках. Насколько я понимаю, для этого нужна совместная работа психологов, физиологов и программистов.

Переходим к проектированию

В этой статье мы проделали предварительную, но очень важную, работу: определили цели языка, требования к языку и компилятору и способы выбора решений. Осталось только пройти по списку того, что надо проектировать, и принимать решения по каждому пункту, а потом рассматривать то, что получилось целиком. И каждый раз делать шаг назад, если нашли противоречие и неудобство.

Основная мысль, которую я хочу высказать здесь, а потом иллюстрировать на конкретных примерах в следующей статье: тщательное продумывание целей и требований влияет принципиальным образом на качество языка и на трудоемкость разработки.

Могут ли измениться требования по ходу? Конечно, могут, и в моей практике разработки языков это было. В таком случае надо заново проверить все решения и безжалостно переделать все, что не соответствует новым требованиям.

Литература

1. Недоря А. Е. Интенсивное программирование, 2022. http://digital-economy.ru/stati/интенсивное-программирование
2. Тривиль: публичный репозиторий. https://gitflic.ru/project/alekseinedoria/trivil-0
3. The Rust Reference. https://doc.rust-lang.org/reference/
4. Kotlin language specification. https://kotlinlang.org/spec/kotlin-spec.html
5. The Swift Programming Language. https://docs.swift.org/swift-book/documentation/the-swift-programming-language/
6. Boehm-Demers-Weiser Garbage Collector. https://hboehm.info/gc/
7. Memory Pool System. https://memory-pool-system.readthedocs.io/en/latest/guide/overview.html
8. The Go Playground. https://play.golang.com/
9. [9] Пентковский В. М. Автокод эльбрус, М.: Наука, 1982
10. [10] Недоря А. Е. Технология разработки мультиплатформенных программ на основе явных схем программ, 2018. http://digital-economy.ru/stati/tekhnologiya-razrabotki-multiplatformennykh-programm-na-osnove-yavnykh-skhem-programm

References in Cyrillics

1. Nedorya A. E. Intensivnoe programmirovanie, 2022. http://digital-economy.ru/stati/интенсивное-программирование
2. Trivil: publichny`j repozitorij. https://gitflic.ru/project/alekseinedoria/trivil-0
3. [9] Pentkovskij V. M. Avtokod e`l`brus, M.: Nauka, 1982.

4. [10] Nedorya A. E. Texnologiya razrabotki mul`tiplatformenny`x programm na osnove yavny`x sxem programm, 2018. http://digital-economy.ru/stati/tekhnologiya-razrabotki-multiplatformennykh-programm-na-osnove-yavnykh-skhem-programm

Ключевые слова

язык программирования, семейство языков программирования, разработка языков программирования, компилятор, прототипирование компиляторов, энергосбережение разработчика

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

Aleksei Nedoria, Development of programming language Trivil. The first steps to the language family. Part 1.

Keywords

programming language, family of programming languages, programming language development, compiler, compiler prototyping, developer energy saving

DOI: 10.34706/DE-2024-04-05
JELclassification – C65, E42

Abstract

This is the first article in a series that describe the development of the Trivil programming language: from reasoning about the need to develop a new language, to defining goals and requirements, and to choosing and justifying specific language solutions. The articles are mainly aimed not at the programmer who uses the language, but at the developer of programming languages. In the articles, the author uses the experience of lectures on language development at MIPT, ITMO and Innopolis University.