1. НАУЧНЫЕ СТАТЬИ

УДК: 04.43, 004.45, 004.4'2

1.1. Проектирование языка системного уровня. Постановка задачи

А. Е. Недоря, г. Санкт-Петербург

Суверенитет страны не является полным, если страна не делает или не контролирует операционные системы, базы данных и управляющие системы, а также инструменты для разработки этих систем. Ключевыми инструментами разработки являются языки программирования и компиляторы. Статья перечисляет основные требования к современному языку системного программирования.

Введение

Суверенитет страны является неполным, если страна не имеет свои или контролируемые ей средства производства. Например, изготовление снарядов (или выращивание овощей) находится под угрозой, если страна импортирует станки для изготовления снарядов или закупает семена. Если заглянуть глубже, то нужны станки для производства станков, на которых делаются станки для изготовления снарядов и так далее. При этом задача полного контроля всех средств производства является экономически бессмысленной и недостижимой. Своими должны быть ключевые инструменты и технологии, плюс те инструменты и технологии, которыми можно обмениваться в рамках кооперации с партнерами.

В программной индустрии ключевыми инструментами являются (в том числе) языки программирования, компиляторы и инструменты разработки, а ключевыми технологиями — технологии разработки этих инструментов.

Предположим, что у некой организации в нашей стране есть намерение, решимость, деньги и люди для разработки языка программирования. Вопрос: с какой области применения стоит начать?

Уточню, язык программирования — это инструмент, и для разных областей применения нужны разные инструмент, для одних лучше микроскоп, а для других — молоток.

На мой взгляд, самой актуальной сейчас областью применения является системное программирование. Нужны инструменты для создания операционных систем, баз данных и управляющих систем, включая военные, космические и другие инфраструктурные системы, обеспечивающие функционирование и безопасность.

Две очевидные причины, выделяющие системное программирования:

- Взлом операционных и управляющих систем и ошибки в них являются предельно опасными;
- В мире отсутствуют адекватные языки программирования и инструменты для системного программирования.

Второй пункт надо пояснить. Существующие операционные системы написаны и пишутся на C, C++ или Rust. Возможно, что в каких-то случаях используются другие языки, например, D, Zig или Nim, но мне не известны такие системы. Что же касается используемых языков, то их недостатки слишком велики и принципиальны.

Недостатки языков C, C++:

- Ручное управление памятью;
- Небезопасная система типов;
- Небезопасные указатели;
- Допустимость неопределенного поведения;
- Нет поддержки модульности.

Недостатки языка Rust: несмотря на то, что все перечисленные выше недостатки C, C++ в Rust тем или иным способом устранены, в нем есть принципиальный недостаток:

- Ограничения, наложенные на структуры данных и алгоритмы настолько велики, что они вынуждают разработчиков писать часть кода в небезопасном (unsafe) режиме. Причем процент небезопасного кода достаточно велик.

Это, как ни парадоксально, делает Rust таким же небезопасным языком, как и C++, при этом еще и существенно менее “дружелюбным” для разработчика.

Я не буду рассматривать языки D, Nim, Zig и другие, в качестве потенциальных кандидатов для системного языка программирования. Просто потому, что с точки зрения суверенитета разработка своего системного языка необходима.

Возможно, что получится использовать какие-то идеи из этих языков, или из экспериментальных языков, таких как Carbon, VLang, Pony, Verona, но ни один из этих языков нельзя взять в качестве образца или за основу. Думаю, что это утверждение станет очевидным после того, как мы выработаем

требования к проектируемому языку системного программирования. Для определенности будем называть его Яс, сокращение от “Язык системный”.

Общефилософские требования

Перед тем, как перейти к требованиям для конкретного языка, перечислю то, что я скорее бы назвал философскими требованиями. Выполнение их трудно проверить и тем более измерить, оценки их выполнения, как правило, субъективны. Несмотря на это, указанные требования очень важны, они позволяют посмотреть со стороны на принимаемые в языке решения, и, что еще важнее, объединяют команду разработки тем, что я бы назвал *этическими нормами* разработки.

Перечислю требования, которые я считаю принципиально важными:

- Простота языка
- Удобство и скорость программирования
- Регулярность
- Баланс

Распишу подробнее то, что для меня стоит за каждым из этих требований в произвольном порядке:

Простота языка:

- В языке есть только то, что нужно
- В языке нет того, что не нужно
- Ортогональность конструкций
- Регулярность конструкций
- Непротиворечивость

Удобство и скорость программирования (включая не только написание кода, но и жизненный цикл программных систем в целом):

- Легкость чтения, легкость понимания
- Наглядность языка
- Лаконичность записи
- В языке (или библиотеках) есть все, что нужно для предметной области (но не больше)
- Наглядность (семантика видна сквозь синтаксис)
- Поддержка хорошего кода (написать ошибочный код сложнее, чем правильный)
- Раннее обнаружение ошибок
- Скорость и удобство использования компилятора и других инструментов
- Качество сообщений об ошибках компилятора и других инструментов
- Минимизация ошибок, обнаруживаемых во время исполнения
- Минимизация потребности в отладке
- Удобство модификации и сопровождения
- Простота использования инструментов (сами инструменты могут быть сложными, насколько это необходимо)
- Мультиплатформенность, как возможность писать код, работающий на всех актуальных платформах, включая, кросс-разработку.

Регулярность:

- Регулярный синтаксис (похожий синтаксис для близких конструкций, разный для разных)
- Регулярная семантика (минимум исключений из правил)
- Нет дублирующих конструкций

Баланс:

- Конструкции языка примерно одного уровня
- Конструкции не доводятся до абсурда (например, как в unified type system, об этом надо говорить отдельно)

Полностью удовлетворить эти требования невозможно, часть из них или противоречит друг другу, или не могут быть достигнуты одновременно.

Не знаю, как читателю, но мне сразу хочется расставить приоритеты, но это стоит делать для конкретного языка или предметной области, для разных предметных областей приоритеты могут быть разными.

От предметной области к техническим требованиям

Перейдем теперь собственно к Яс и будем говорить о производительности, безопасности, управлении памятью и прочих привычных для программиста вещах.

Сначала посмотрим на предметную область. Яс нацелен на создание **больших**, **критических** (и, значит, сложных) и **долгоживущих** систем. Возьмем, для примера операционные системы, они меняются, развиваются и живут долго: выход Линукса 1991 год, выход Windows NT — 1993 год и эти ОС вовсе не умирают. Так что время жизни ОС точно не меньше 50 лет.

Что это значит для Яс:

1. Любой код, кроме одноразового, читается чаще, чем пишется. Код системного уровня читают в десятки или сотни раз чаще, чем пишут, и это значит, что для Яс **легкость чтения и понимания** является одним из самых приоритетных требований.
2. Так как системы сложные, то на уровне языка должно быть уделено особое внимание **безопасности** кода, надежности и корректности системы.
3. Критически важным является **поддержка разработки правильного кода**, включая
   - **раннее обнаружение** ошибок
   - **гарантии** отсутствия ошибок, по крайней мере, ошибок самых опасных классов
   - минимизация ошибок времени исполнения
   - минимизация необходимости тестирования и отладки, включая возможность тестирования и отладки только новых или измененных частей с гарантией **изоляции**, отсюда **модульность** и **компонентность**.
4. Возможность (простота) **формальной верификации**, и отсюда тоже следует требования изоляции (надо “есть слона по частям”).
5. Нужно думать о защите кода от взлома, для языка — это отсутствие уязвимостей. Весь инструментарий должен сразу делаться с учетом требований **информационной безопасности**.
6. **Эффективность**, во всех ее составляющих (подробнее ниже), тоже обязательное требование к Яс.
7. **Расширяемость**: при жизни 50+ лет, мы не можем рассчитывать, что учли все требования. Надо думать про точки расширения языка, и, в первую очередь, на возможность расширения на уровне библиотек.

Теперь можно переходить к более подробному расписыванию требований. Я буду расписывать требования в некотором удобном для себя порядке, полагая, что систематизацию и структуризацию лучше делать, когда первичный материал уже собран.

Безопасность

Раскладывается на

- Безопасность типов: ортогональная типовая система с доказанной (формальными методами) корректностью (type system soundness),
- Безопасность ссылок: нет явных указателей, нет обращения по ошибочной или неопределенной ссылке (null safety),
- Обязательная инициализация всех переменных до использования,
- Безопасность памяти: нет ошибок, связанных с неверным управлением памятью, независимо от способа управления памятью,
- Нет действий для которых поведением не определенно (undefined behavior),
- Нет языковых исключений (runtime exceptions). При выполнении возможны только те исключительные ситуации, от которых на уровне языка нельзя защититься, например, ошибки аппаратуры или невозможность выделить память,
- Модульность: исходный код разделен на достаточно сильно изолированные части,
- Компонентность: есть возможность замены компоненты или добавления изолированных компонент исключая влияние на остальные части системы.

Если внимательно рассмотреть эти пункты, то станет понятно, что, в первую очередь, в них речь идет о проверках и гарантиях во время компиляции. Таким образом, в требованиях неявно заложена задача сделать компилятор, включающий продвинутые анализаторы, то есть то, что для дырявых языков, типа C++, пытаются делать отдельными статическими анализаторами и санитайзерами.

Сразу возникает вопрос: можно ли написать всю ОС на безопасном коде? Пока такая возможность не доказана, хотя бы конструктивно, я буду считать, что нельзя. Как же тогда быть с требованием безопасности? Ответ, с формальной точки зрения, вполне очевиден: небезопасный код надо писать на другом языке.

Полностью ли это другой язык или это под(над) множество нам сейчас не важно. Важно то, что код на Яс (безопасный язык) и на Яс-НБ (Яс НеБезопасный) должны быть разделены, по меньшей мере, размещены в разных модулях или файлах с разными расширениями, явно выделенными заголовками и т.д. Если в модуле/файле есть хоть одна небезопасная строка, то весь файл небезопасен. Использование (импорт) небезопасного кода тоже должно быть явно выделено. Такое разделение позволит нам ввести дополнительную защиту, возможно на уровне пред и пост условий и, в дальнейшем, перейти к доказательству корректности небезопасного кода. Для этого желательно сделать Яс-НБ как можно меньше, с точки зрения функциональности. Если писать на Яс-НБ будет трудно, это существенно улучшит общий уровень безопасности разрабатываемой системы.

Я пишу здесь про доказательство корректности только небезопасного кода, потому что **гарантировать корректность** основного кода должен компилятор.

Итак, мы получили дополнительный пункт в разделе Безопасность:

- Разделение (отсутствие смешивания) безопасного и небезопасного кода.

Замечу, как минимум на переходный период, который может быть очень длительным, надо обеспечить возможность вызова C кода. Понятно, что любой внешний код является небезопасным, и надо думать, какими проверками его обкладывать.

В качестве одного из вариантов Яс-НБ можно рассмотреть использование (одного из) внутренних представлений компилятора. Например, компилятор Rust использует MIR [1], а компилятор Swift SIL [2] в качестве внутреннего представления более высокого уровня, чем LLVM IR.

Мы можем использовать аналогичное внутреннее представление в качестве Яс-НБ если оно:

1. Является переносимым (машинно-независимым),
2. Имеет текстовую форму,
3. Позволяет использовать ограниченный набор небезопасных операций,
4. Включает дополнительные проверки для таких операций.

Модульность и компонентность

Я уже упоминал выше оба этих термина. На мой взгляд, они описывают два разных подхода к структуризации:

- Модульность: определяет статическую структуризацию через инкапсуляцию (частичную изоляцию) и импорт/экспорт;
- Компонентность: определяет, в общем случае, динамическую структуризацию, которая может сводиться к статической посредством оптимизации.

Компонентность имеет серьезные преимущества для больших и развиваемых программных систем. Подробнее см. [3]. Замечу, что хотя термин **компонентность** в указанной статье не используется, но речь идет именно об этом.

Для Яс я считаю необходимой поддержку и модульности и компонентности.

Эффективность

Я предпочитаю использовать слово эффективность, а не производительность, так как эффективность — это более широкое понятие, которое включает в себя:

- производительность (скорость работы),
- количество используемой памяти,
- отсутствие задержек на служебные действия (например, на сборку мусора),
- скорость инициализации (startup time),
- экономию батареи.

Для разных задач на первый план выходят разные стороны эффективности.

Говоря об эффективности, я не имею в виду только уровень оптимизации кода, мы должны учитывать, что эффективность может достигаться на нескольких уровнях:

- Архитектурный уровень,
- Алгоритмический уровень,
- Уровень кода (оптимизация кода).

Очевидно, что переход, например, от сортировки пузырьком к быстрой сортировке дает выигрыш, который невозможно достичь оптимизацией кода. На уровне архитектуры выигрыш может быть еще на порядки больше, например, за счет удаления ненужных действий, уменьшения количества уровней архитектуры или уменьшения потока данных между устройствами.

Мы должны думать на всех уровнях, и тогда становится понятно, что эффективность существенно связана с понятиями модульности и компонентности.

Впрочем, это тема требует отдельного разговора, а пока, на уровне языка можно сформулировать только самые общие требования:

- Принцип “не навреди”: конструкции, которые принципиально ухудшают эффективность, должны быть исключены из языка,
- Принцип “вторичности”: эффективность нельзя достигать за счет нарушения безопасности.

Типовая система

Требования к типовой системе для меня очевидны:

- Безопасность,
- Полнота, с точки зрения набора поддерживаемых парадигм,
- Ортогональность,
- Поддержка разных видов полиморфизма, включая обобщенное программирование.

В последнем пункте я намеренно не использую термин обобщенные типы (generic types), так как за этим термином стоит набор привычных решений, которые добавляют в язык неоправданную сложность. Надо найти достаточно простое и достаточно мощное решение. Наметки такого решения в Тривиле у меня продуманы, но не сделаны.

Управление памятью

Управление памятью — это ключевой пункт, который во многом определит успешность языка.

Как уже сказано в разделе Безопасность, в Яс нужны гарантии безопасной памяти, а именно

- Нет висячих ссылок,
- Нет использования блока памяти после освобождения,

- Нет доступа к нераспределенной памяти, включая доступ по пустому указателю (null pointer dereference),
- Нет утечек памяти.

Любопытно, что Rust считает утечки памяти допустимыми [4]. Некая логика в этом есть, так как утечки не приводят к авариям, пока памяти достаточно. На мой взгляд, для Яс утечки памяти тоже являются критическими, хотя может быть предусмотрено корректное освобождение недоступной памяти по ее исчерпанию.

Как мы знаем, безопасное управление памятью может быть реализовано посредством

- Сборки мусора,
- Счетчиков ссылок и слабых указателей (weak pointers),
- Статического распределения памяти,
- Контролем ссылок во время компиляции с автоматическим удалением,
- Комбинацией подходов, включая разделение на регионы с разным управлением.

Каждый из этих подходов имеет свои недостатки. На мой взгляд, мы не можем и не должны ограничиваться в Ясе одним подходом. В любой системе могут быть разные части, для одних критичным будет отсутствие задержек, для других удобство программирования, и так далее.

Поэтому добавим требование:

- Возможность выбора управление памятью на некоторых или всех перечисленных уровнях:
  - программной системы,
  - компоненты,
  - контейнера (структуры данных).
- Возможность использования объектов из областей памяти с разными управлением.

Для обеспечения этих возможностей, надо думать об ограничениях на ссылки (reference capabilities). См. например, ограничения в языках Pony [5] и Verona [6].

Самым интересным для Яс подходом является управление памятью во время компиляции, основной проблемой здесь является накладываемые ограничения, которые могут существенно снизить гибкость, мощность языка и удобство программирования, а также сделать проблематичной раздельную компиляцию. Один из вариантов, которые сейчас обсуждается на рабочей группе по управлению памятью, это двухуровневый подход с меньшими ограничениями и большей гибкостью на уровне компоненты (нижний уровень), и со строгим ограничениями на взаимодействие компонент (верхний уровень).

Параллелизм

Еще один ключевой пункт, наряду с управлением памятью. Очевидно, что в Яс надо думать больше о конкурентности (concurrency), чем о параллельной обработке данных (parallelism). К сожалению, подходящего русского термина для concurrency я не знаю.

Самым безопасным подходом выглядит акторная модель или модель с активными объектами. Например, в языке Pony, построенной на этой модели, гарантировано выполнение следующие свойств:

- Отсутствие гонок памяти (data races)
- Отсутствие тупиков (deadlocks)

Но в Яс надо думать и о распределении процессов или потоков на ядра процессора, а это существенно более низкий и опасный уровень. Как только мы говорим об одновременном доступе к данных, мы выходим на уровень мониторов, семафоров и мьютексов и теряем гарантии.

Достаточно ли нам в языке поддержки активных объектов, не требующих блокировки, а более низкоуровневые и опасные конструкции должны быть в языке Яс-НБ? Я не знаю.

У меня нет готовых ответов, предполагаю, что пусть к параллелизму лежит через понятия

- Неизменяемость (immutability),
- Владения (ownership) в смыслах Rust [7] и Argentum [8],
- Не блокирующих контейнерах (lock-free),
- Изоляции в смыслах Pony [5] и Dart [9],

с использованием методов формального доказательства.

Семейство языков

Говоря об языке системного программирования нельзя забывать о том, что в этом языке должен быть существенный сдвиг в сторону безопасности и эффективности, из чего безусловно следуют более высокие требования к разработчикам, и уменьшение продуктивности тех разработчиков, которые не дотягивают до нужного уровня.

Из этого, в свою очередь следует то, что для разработки приложений, должны быть другие языки. Очень желательно, я бы сказал, необходимо, чтобы эти языки были совместимы с Яс, минимизируя проблемы взаимодействия.

Например, естественным и выгодным решением является разработка на Яс системы поддержки выполнения (runtime) для языка приложений. Отсутствие лишних барьеров, отсутствие необходимости конвертации при передаче данных от одного языка к другому, удобство разработки гибридных приложений — это очень существенное движение в сторону увеличения продуктивности разработчиков приложений.

Из всего этого следует, что разрабатывая Яс мы должны думать о разработке семейства языков, состоящем из нескольких языков для разных предметных областей. Оценивая предложения, в первую очередь, по типовой системе, управлению памятью и параллелизму в Яс мы должны думать и о том, как это отразится на других языках семейства.

Наблюдаемость

Под **наблюдаемостью** я понимаю возможность видеть (визуализировать) решения и процессы. Например, если компилятор делает нечто аналогичное созданию интерфейса по объекту в Go или перебрасывает структуру данных со стека в кучу (Go escape analysis [10]), то у разработчика должна быть возможность получить список всех мест, в которых такое преобразование применено.

А если в системе изменилась или добавилась новая компонента, то должна быть простая возможность во время исполнения включить запись (протоколирование) всех взаимодействий с этой компонентой. Замечу, такая возможность должна быть реализована в инструментах и среде выполнения и не требовать от разработчика вставки вызовов для протоколирования.

Разрабатывая язык и инструменты нам надо перечислить места неявных преобразований или взаимодействий и думать о возможность сделать их наблюдаемыми.

Информационная безопасность

С точки зрения языка все очевидно: в языке не должно быть уязвимостей. Единственный способ добиться этого, это включить специалистов нужного профиля в состав группы разработчиков языка и использовать формальные методы для доказательства всего, что можно доказать.

Безопасность инструментов — это большая тема, о которой надо говорить отдельно.

Запреты

Кроме требований вида “надо” и “желательно”, должны быть требования, запрещающие добавлять в язык определенные конструкции. Это сокращает пространство решений и упрощает разработку.

Вот мой личный список “так не должно быть”:

- Макросы: как минимум в стиле C, а может и любые,
- Явные указатели,
- Унифицированная система типов,
- Множественное наследование,
- Перегрузка операторов, разве что кроме строго локализованного и контролируемого использования,
- Неявный импорт,
- Неквалифицированный импорт,
- Стирание типов (type erasure),
- Неявная подмена типов (smart cast).

Для каждого пункта у меня есть обоснование, которое я здесь опущу. Естественно, что это черновые требования, которые стоит обсуждать.

Заключение

Эта статья определяет черновой набор требования к Яс. Следующий этап работы: критика, обсуждение, структуризация и приоритизация набора требований.

Я сознательно не затронул часть требований, которая касается контроля языка и инструментов. Для меня очевидно, что

- Мы должны сочетать открытую разработку с жестким контролем изменений;
- Писать исходные тексты, документацию, статьи на русском языке, публиковать только в русских изданиях;
- Хранить информацию на серверах внутри страны.

Остальное должно быть выработано в сотрудничестве с профессионалами в этой области. И мы не должны забывать про полезный урок, который дал нам Линус Торвальдс, выкинув русских разработчиков из ядра Линукса.

Еще одна часть требований, которая оставлена за пределами статьи, это бизнес требования. О них нет смысла говорить до тех пор, пока идет инициативная разработка.

Литература

[1] The MIR (Mid-level IR), https://rustc-dev-guide.rust-lang.org/mir/index.html
[2] Swift Intermediate Language (SIL), https://deepwiki.com/swiftlang/swift/4-swift-intermediate-language-(sil)
[3] Недоря А. Е. Третья структурная эволюция. Введение в архитектурное программирование, программирование // Цифровая экономика. – 2025. - № 2(32). – с.52-59.
[4] Rust. Reference Cycles Can Leak Memory, https://doc.rust-lang.org/book/ch15-06-reference-cycles.html
[5] Pony. Reference Capabilities, https://tutorial.ponylang.io/reference-capabilities
[6] Reference Capabilities for Flexible Memory Management, https://dl.acm.org/doi/epdf/10.1145/3622846
[7] Rust. Understanding Ownership, https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html

[8] Argentum programming language, https://aglang.org/
[9] Concurrency in Dart, https://dart-dev.web.app/language/concurrency
[10] Stack Allocations and Escape Analysis, https://goperf.dev/01-common-patterns/stack-alloc/

References in Cyrillics

[3] Nedorya A. E. Tret'ya strukturnaya ehvolyuciya. Vvedenie v arkhitekturnoe programmirovanie, http://digital-economy.ru/stati/третья-структурная-эволюция-введение-в-архитектурное-программирование

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
системное программирование, язык системного программирования, семейство языков программирования, разработка языков программирования, безопасность языков программирования, типовая система, управление память, параллелизм, информационная безопасность

Aleksei Nedoria, Designing a system-level language. Problem statement

Keywords
system programming, system programming language, family of programming languages, programming language development, programming language security, type system, memory management, parallelism, information security

DOI: 10.34706/DE-2026-01-01
JEL classification: C65, E42

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.

A country's sovereignty is not complete if a country does not make or own operating systems, databases, and control systems, as well as the tools for developing these systems. Programming languages and compilers are key development tools. The article lists the basic requirements for a modern system programming language.

.