**Проект SLang**

**Введение**

В университете Иннополис в последние годы проводятся проектные и конструкторские работы, ставящие целью создание оригинального языка программирования (рабочее название – **SLang**), а также его базового и прикладного окружения: системных и прикладных библиотек, поддержки управления программными конфигурациями и т.д.

Целевая ориентация проекта SLang предусматривает использование нового языка в широком спектре приложений, важнейшими из которых служат следующие:

* Разработка больших и сложных программных и программно-аппаратных систем с повышенными требованиями к надежности (mission-critical applications).
* Встроенные системы промышленного назначения: робототехнические комплексы, беспилотные летательные аппараты, автомобили и другие мобильные устройства.
* ПО для обеспечения безопасной и эффективной передачи данных: поддержка технологии блокчейн.
* Программные системы с ограниченными возможностями по энергопотреблению.
* Системы с элементами искусственного интеллекта: нейронные сети, обработка и анализ больших массивов информации («big data»), реализация эффективных алгоритмов распознавания образов.
* Системы управления процессами разработки ПО; программные комплексы верификации и сертификации ПО по различным критериям.

**Предпосылки: зачем новый ЯП?**

Язык программирования (ЯП), со своей инфраструктурой – системами поддержки времени выполнения, стандартными и прикладными библиотеками, инструментами и средами разработки (компиляторы, редакторы связей, IDE) представляет собой ключевой инструмент создания программного обеспечения (ПО) и, тем самым, служит определяющим фактором обеспечения эффективного, безопасного и надежного функционирования многочисленных и разнообразных электронных устройств в современном мире.

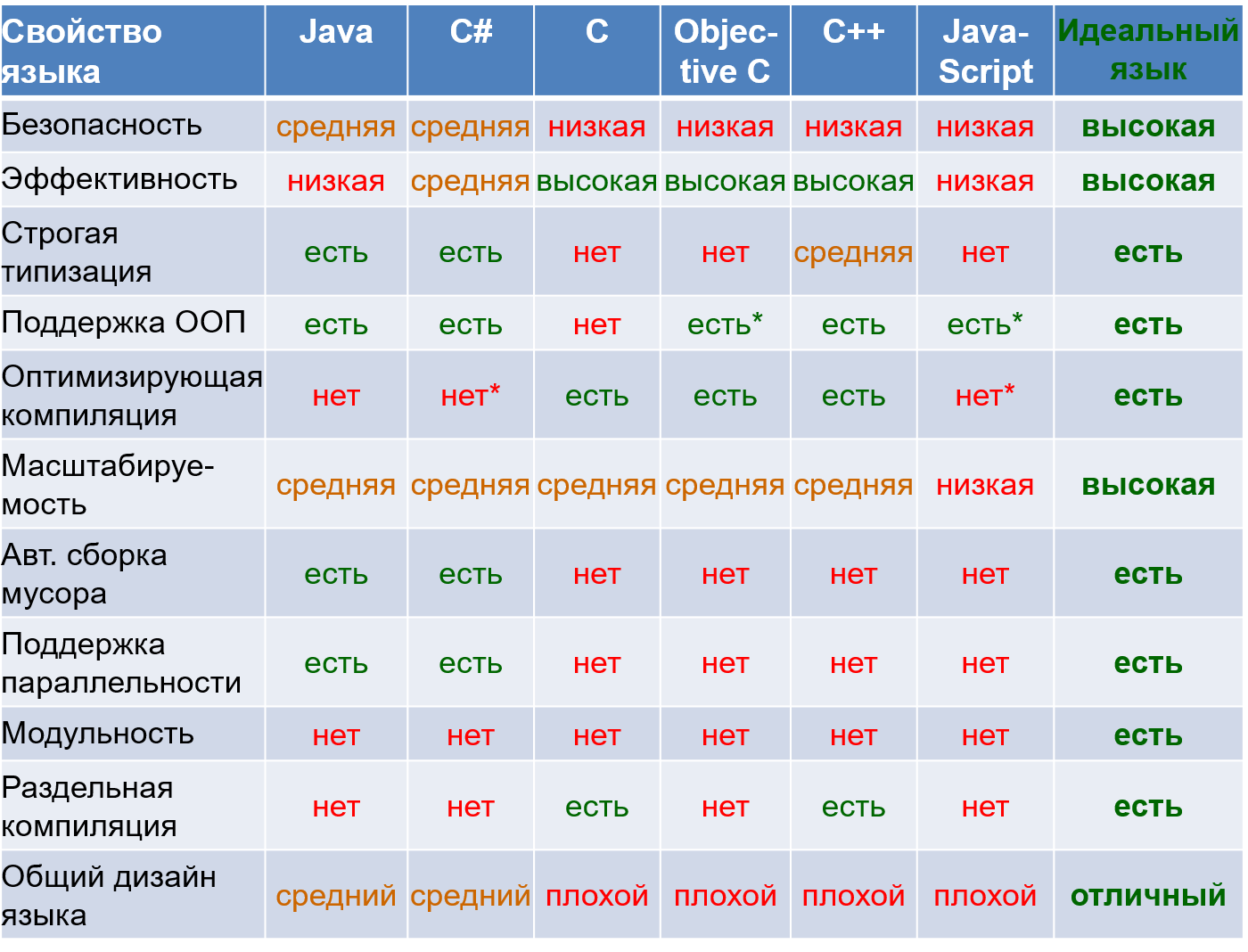
Существующее положение в сфере языков программирования и, шире, в области инструментария современного программирования весьма далеко от идеала. Большинство языков программирования, наиболее широко используемых в настоящее время, создано более двадцати лет назад и в настоящее время совершенно не адекватно практически ни одному из современных требований, предъявляемых к инструментам разработки современного ПО. Эти языки архаичны, неуклюжи, громоздки, неудобны и сложны в практическом использовании, не способствуют надежности и эффективности создаваемого ПО, зачастую несут явный отпечаток вкусовых пристрастий и причудливых взглядов их создателей.

Новые языки программирования, в изобилии появляющиеся в последние пять-семь лет, пытаются преодолеть указанные недостатки, однако в значительной степени повторяют устаревшие и порочные подходы проектирования, берущие своё начало в восьмидесятых годах прошлого столетия.

Представляется, что ключевые концептуальные требования, предъявляемые к современным инструментам разработки программ, и одновременно фундаментальные свойства современного языка программирования могут быть сведены к следующим:

* Один язык для различных аппаратных платформ; один язык для разных классов задач.
* Поддержка современных парадигм программирования –процедурной, объектно-ориентированной, обобщенной, функциональной.
* Модульность и раздельная компиляция.
* Использование гибких моделей выполнения – непосредственный машинный код, интерпретируемый код, смешанный режим, оптимизации времени выполнения – в зависимости от особенностей целевой аппаратуры и от требований к разрабатываемому ПО.

В 2015 году с участием автора данного представления был проведён качественный анализ нескольких распространенных языков программирования на предмет их соответствия современным требованиям. Результат анализа, проведённого методом экспертных оценок, представлен в виде следующей таблицы:



Как видно из таблицы, ни один из современных ЯП в полной мере не удовлетворяет базовым требованиям и не обладает нужным ассортиментом свойств.

Правая колонка приведенной выше таблицы представляет сочетание необходимых свойств, характеризующих «идеальный» язык программирования и одновременно может считаться целями реализации проекта SLang.

**Предпосылки: зачем нужны отечественные компиляторы?**

В настоящее время проблема импортозамещения в сфере программного обеспечения (ПО) находится в центре внимания профессиональной общественности. Только на пути создания отечественных программных решений можно достигнуть необходимой степени безопасности и надежности функционирования сложных современных комплексов различного назначения.

Одним из наиболее существенных аспектов создания отечественного ПО, которое, к сожалению, находится на периферии усилий в этом направлении, представляется разработка инструментов и сред программирования. В первую очередь это относится к проблеме создания отечественных компиляторов для современных языков программирования (ЯП).

Традиционно считается, что ключевым аспектом безопасности являются надежное базовое и прикладное ПО (то есть, операционные системы и прикладные пакеты), и именно эти категории программ нуждаются в импортозамещении в первоочередном порядке. С другой стороны, инструментальное ПО (компиляторы и сопутствующие средства разработки) не является критическим в аспекте безопасности, так как реализует сугубо технический механизм эквивалентного преобразования исходного текста программ в машинные коды. Поэтому для создания прикладного ПО достаточно взять за основу какой-либо общедоступный инструмент, распространяемый по свободной лицензии, адаптировав его для целевой аппаратной платформы.

Представляется, что данная точка зрения на инструментальные средства разработки ПО является поверхностной и потенциально несущей в себе изъяны и угрозы в плане безопасности создаваемого ПО.

Обычная технология адаптации наиболее распространённых инструментов разработки (семейство компиляторов gcc или программный комплекс llvm/clang) предполагает использование машинно-независимой фронтальной части такого инструмента («front-end compiler») и разработку собственного машинно-зависимого генератора кода («back-end compiler») для целевого оборудования.

Будучи на первый взгляд вполне естественной, такая технология, тем не менее, несёт в себе ряд потенциальных уязвимостей. Дело в том, что интерфейсом между «стандартной» фронтальной частью компилятора и генератором кода служит так называемое промежуточное представление (ПП) исходной программы. Это ПП порождается фронтальной частью компилятора и используется в качестве исходной информации для любого генератора кода, создаваемого в рамках описанной технологии. Формат и алгоритмы порождения ПП фронтальной частью полностью определяются авторами инструмента и считаются их «внутренней кухней». Этот формат, как правило, плохо документирован (особенно в случае комплекта gcc) и может быть изменен авторами в любой момент, о чём в документации имеются специальные предупреждения.

Однако, наиболее проблематичным служит то обстоятельство, что отсутствуют какие бы то ни было гарантии адекватности ПП, созданного фронтальным компилятором, тексту исходной программы. Чтобы удостовериться в такой адекватности, необходим детальный анализ алгоритмов работы фронтальной части, который из-за большого объёма исходного текста компилятора и сложности алгоритмов практически не представляется возможным.

Ситуация осложняется нестандартной постановкой задачи: целью такого анализа является не поиск ошибок (который в ряде случаев можно хотя бы частично автоматизировать), а установление *семантического соответствия* между фрагментами текста исходной программы и соответствующими фрагментами промежуточного представления.

Несоответствие структуры и содержания ПП исходному тексту прикладной программы может возникнуть как в результате ошибки проектирования или программирования компилятора, так и оказаться намеренной «закладкой» в его коде. Оба этих варианта в равной степени делают созданный по описанной технологии компилятор неприемлемым для целей разработки надежного прикладного ПО, удовлетворяющего специальным требованиям.

Преодоление описанного органического недостатка технологии использования общедоступных инструментальных средств возможно только на пути разработки собственных компиляторов ЯП. Таким образом, реализация отечественных инструментальных средств представляется одной из первоочередных задач в контексте общего направления на импортозамещение в сфере ПО.

**Проект SLang**

Основное содержание проекта заключается в том, чтобы спроектировать и реализовать оригинальный язык программирования вместе с необходимой экосистемой: компилятором, подсистемой поддержкой времени выполнения, стандартными и прикладными библиотеками, редактором связей (комплексатором), системой конфигурирования и средствами отладки программ.

Важнейший смысл проекта заключается в том, чтобы предложить разработчикам программного обеспечения XXI века инструмент, который бы позволил решать разнообразные задачи наиболее простым и надежным способом в соответствии с предъявляемым требованиям и применительно к разнообразным уровням квалификации разработчиков.

Язык SLang воплощает адекватное понимание существа процесса проектирования и разработки современного ПО. Он включает свойства, обеспечивающие надежность, безопасность, эффективность и масштабируемость программ, создаваемых с его использованием. В то же время он достаточно прост для обучения, освоения и использования, что обеспечит «гладкий» процесс разработки и сопровождения, а также предоставит возможность включить в сферу разработки ПО более широкие, нежели в настоящее время, сообщества разработчиков, в том числе, специалистов прикладных областей, не являющихся профессионалами-программистами.

Язык SLang включает широкий набор свойств, в совокупности обеспечивающих весь спектр современных подходов, методик и парадигм разработки программ (объектно-ориентированное, функциональное, обобщенное и модульное программирование). При этом язык спроектирован таким образом, чтобы избежать или свести к минимуму те недостатки современных ЯП, которые не позволяют считать их подходящими для создания эффективных и высоконадежных программ, отвечающих современным требованиям.

**Ожидаемые преимущества**

Использование языка SLang для разработки ПО, решающего широкий класс задач, может принести ряд несомненных выгод, осторожные оценки которых приводятся в следующей таблице.

|  |  |  |
| --- | --- | --- |
| **Ожидаемое преимущество** | **Консервативная оцен-ка (по сравнению с использованием С++)** | **Свойства языка, за счёт которых преимущество будет достигнуто** |
| Надежность разрабатываемого кода | В 3-5 раз выше | * Предикаты: предусловия, постусловия, инварианты. * Надежность системы типов: полная поддержка конформности и явных преобразований * Отсутствие нулевых указателей и неинициализированных данных |
| Стоимость разработки и поддержки ПО | В 3-5раз ниже | * Множественное наследование с эффективным разрешением конфликтов. * Безопасное взаимодействие с существующими программными компонентами. * Поддержка процедурного, объектно-ориентированного, функционального и параллельного программирования. |
| Производитель-ность создаваемого ПО | Не хуже C++ | * Языково-зависимые высокоуровневые оптимизации. * Оптимизации на стадии генерации кода (с использованием возможностей платформ класса LLVM). * Оптимизации времени выполнения. |
| Сложность параллельного программирования | В 3-5раз меньше | * Явный параллелизм на уровне входного языка: несколько простых принципов и одно дополнительное служебное слово. * Автоматический параллелизм для тел подпрограмм. |

**Язык программирования SLang: общее описание**

Ниже приводится список основных свойств языка SLang, проектируемого в рамках одноименного проекта:

* Строгая типизация с продвинутой поддержкой механизма выведения типов на этапе компиляции.
* Непротиворечивая семантика и ясный синтаксис языка.
* Автоматическое управление памятью.
* Многоуровневый, безопасный и простой в использовании параллелизм.
* Возможности для оптимизации программ.
* Защитное программирования с предикатами и поддержка автоматической верификации.
* Типовая безопасность; контроль неинициализированных объектов.
* Поддержка параметризации разных видов.
* Оригинальный подход к модульности – контейнеры.

Более подробное описание основных свойств и особенностей языка SLang приводится в Приложении.

**Экосистема языка SLang: дорожная карта**

|  |  |  |  |
| --- | --- | --- | --- |
| **Квар- тал** | **Содержание этапа** | **Трудо- затраты** | **Результат этапа** |
| **Фаза 1. Анализ и эскизное проектирование** | | | |
| **1** | - Проектирование языка.  - Проектирование стандартной библиотеки.  - Проектирование общей конфи-гурации системы программиро-вания.  - Анализ потенциальных целе-вых аппаратных платформ.  - Анализ существующих интегри-рованных сред на предмет их использования в системе программирования.  - Разработка детального плана реализации проекта. | 5 человек | - Нормативное описание языка программирования (версия 1).  - Структура и состав стандарт-ной библиотеки языка (эскиз-ный проект).  - Описание системы програм-мирования (эскизный проект).  - Детальный план реализации проекта. |
| **2** | - Проектирование компилятора языка.  - Проектирование стратегии тестирования компилятора и спецификаций тестов.  - Проектирование стратегии тестирования стандартной библиотеки языка.  - Разработка инструментария тестирования компилятора и стандартной библиотеки. | 5 человек | - Эскизный проект компилято-ра языка.  - Описание принципов и стра-тегии тестирования компиля-тора и стандартной библиоте-ки.  - Разработка монитора тести-рования. |
| **3** | - Разработка прототипа компи-лятора языка для платформы .NET.  - Разработка ключевых компо-нентов стандартной библиотеки.  - Разработка пакета тестов для тестирования компилятора и стандартной библиотеки.  - Оценка качества и параметров прототипа компилятора и стан-дартной библиотеки.  - Анализ текущего состояния проекта и коррекция общего плана работ. | 9 человек | - Готовность прототипа ком-пилятора.  - Готовность прототипа стан-дартной библиотеки.  - Готовность системы тести-рования (ограниченная версия).  - Протокол (акт) оценки каче-ства и готовности прототипа компилятора и стандартной библиотеки.  - Протокол (акт) оценки теку-щего состояния проекта с предложениями по коррекции общего плана работа. |
| **Фаза 2. Реализация прототипа компилятора и библиотек** | | | |
| **4-5** | - Реализация полной версии ком-пилятора для .NET.  - Реализация системы тести-рования компилятора и стан-дартной библиотеки  - Тестирование компилятора и стандартной библиотеки (параллельно с реализацией). | 9 человек | - Полная реализация фрон-тальной части компилятора.  - Полная реализация генера-тора кода для платформы .NET.  - Полная готовность системы тестирования.  - Протокол тестовых испыта-ний компилятора и стандарт-ной библиотеки. |
| **Фаза 3. Проектирование и реализация компонентов системы программирования** | | | |
| **6** | - Проектирование комплексатора (редактора связей)  - Проектирование подсистемы конфигурирования программ  - Проектирование интегриро-ванной среды разработки программ. (***Вариант***: проекти-рование механизма встраивания системы программирования в существующую интегрированную среду.)  - Проектирование протокола взаимодействия компонентов системы программирования и интегрированной среды разработки | 9 человек | - Технический проект компо-нентов системы программиро-вания: комплексатора и под-системы конфигурирования.  - Технический проект механиз-ма взаимодействия системы программирования и интегри-рованной среды. |
| **7-8** | - Реализация компонентов системы программирования и интегрированной среды.  - Включение компилятора и компонентов системы програм-мирования в интегрированную среду  - Разработка стратегии интегра-льного тестирования системы в целом.  - Интегральное тестирование всех разработанных компонен-тов. | 9 человек | - Полная готовность компо-нентов системы программи-рования.  - Готовность интегрированной среды с включенным компиля-тором и компонентами систе-мы программирования.  - Протокол (акт) интегрального тестирования. |
| **Фаза 4. Завершение проекта** | | | |
| **9** | - Разработка типовых сценариев и стресс-тестов для системы программирования.  - Комплексное тестирование системы и анализ эффективности компонентов.  - Разработка инструкций по разработке программ, инструкций и рекомендаций по использованию системы.  - Разработка учебных пособий и курсов по разработке программ.  - Разработка рекомендаций по дальнейшему развитию и усовершенствованию системы.  - Разработка плана и проведение приемо-сдаточных испытаний системы. | 9 человек | - Протокол тестовых испытаний системы программирования.  - Предложения по повышению эксплуатационных характеристик и общему развитию системы.  - Описание языка и учебник по программированию.  - Инструкции: руководство пользователя, руководство программиста.  - Рекомендации по дальнейшему развитию и усовершенствованию системы.  - Акт (протокол) приемо-сдаточных испытаний системы программирования. |

**Заключение**

Проект развивается под руководством профессора университета Иннополис **Евгения Зуева**, который имеет существенный опыт проектирования и реализации языков программирования. К числу наиболее значимых проектов, выполненных под его руководством и при его непосредственном участии, относится создание первого в России ISO-совместимого компилятора языка С++ (компания Интерстрон, 2000), используемого при создании ПО для ряда отечественных микропроцессоров.

Новый язык проектируется совместно с ведущими специалистами других организаций. В реализации отдельных компонентов проекта SLang заняты студенты старших курсов университета Иннополис.

Результаты работы докладывались на ведущих отечественных профильных конференциях, в частности CEE-SECR в 2015 и 2017 гг., а также на конференции Programming Languages and Compilers (PLC-2017, Ростов-на-Дону). По темам, связанным с проектом SLang, был опубликован ряд статей.

В настоящее время реализуется прототип («нулевая версия») компилятора языка SLang для нескольких распространенных аппаратных и программных платформ, а также создаются экспериментальные версии важнейших компонентов инфраструктуры языка: стандартная библиотека, конфигуратор программ и т.д. Кроме того, готовится ряд документов (книг) как с нормативным, так и неформальным описанием языка.

**Представляется, что степень проработанности проекта SLang дает достаточные основания, чтобы предложить его в качестве перспективного основного языка программирования для реализации ПО со специальными требованиями и характеристиками.**

**Приложение 1  
Краткая характеристика языка SLang**

**Модульность**

Программа на SLang формируется по модульному принципу: строительными блоками любой программы служат *модули* – независимо определяемые, независимо хранимые и независимо компилируемые единицы со строго определенными интерфейсами, согласно которым они могут вступать в различные отношения друг с другом: использование, агрегация, наследование, активация и т.д.

В языке определены два основных вида модулей: контейнеры и подпрограммы. Контейнеры представляют агрегации логически связанных ресурсов (данных и подпрограмм-членов), подпрограммы реализуют некоторую функциональность и, в свою очередь, могут представлять собой процедуры или функции.

Продолжая эту логику, естественно считать, что компонент любого из указанных видов может служить *единицей компиляции*, то есть, допускать раздельную компиляцию. Отсутствие ограничений способствует созданию композиции программы, адекватной требованиям и особенностям ее использования. Программа может представлять собой, по сути, произвольную композицию единиц, от простого набора взаимодействующих подпрограмм до сложной комбинации контейнеров различных видов.

В предельном случае единица компиляции или вся программа может представлять собой единственную единицу – простую последовательность операторов. Если необходимо написать несколько строк кода, которые будут служить реакцией на нажатие клавиши мыши в некотором средстве просмотра Интернет-страниц, то нет необходимости писать подпрограмму – достаточно просто задать последовательность операторов, которая выполнит нужное действие. Если же решение задачи предполагает более сложную логику, то результат можно получить, комбинируя отдельные подпрограммы, контейнеры и блоки. Таким образом, единая языковая нотация может быть использована для решения максимально широкого класса задач.

**Строгая типизация**

Понятие типа является одним из базовых понятий любого языка. Под ***типом*** некоторого объекта, существующего в программе, понимается тройственная сущность, определяемая множеством ***значений***, которые может принимать данный объект, связанным с ним множеством ***операций***, допустимых над значениями данного типа, а также множеством ***отношений*** между данным типом и другими типами.

Язык SLang представляет собой язык со ***статической типизацией***. Это означает, что тип является статически неизменным свойством объекта; это свойство присуще объекту с момента его возникновения в программе и не может измениться во время жизни этого объекта. В терминах программирования это означает, что тип объекта назначается объекту (явно или неявно) при его объявлении, и не существует возможностей (языковых конструкций), позволяющих изменить тип в процессе выполнения программы.

Статическая типизация является в настоящее время признанным средством обеспечения надежности и высокой производительности программ.

Тип может быть (явно) приписан объекту программистом при объявлении объекта, либо (неявно) выведен компилятором из контекста объявления этого объекта. Примером контекста в данном случае может служить тип инициализирующего выражения из объявления объекта.

Оба варианта назначения типа – явный и неявный – являются статическими по своей природе и отличаются друг от друга только агентом, производящим назначение типа объекту: в первом случае таким агентом является программист, во втором – компилятор.

Текущее значение объекта может быть преобразовано в значение другого типа. Правила преобразования типов определяются свойствами этих типов, представленными в форме соответствующих операций; предопределённые правила преобразования в языке отсутствуют.

**Поддержка различных парадигм программирования**

SLang – мультипарадигменный язык, в том смысле, что в нём воплощены важнейшие современные концепции программирования, включая объектно-ориентированное, обобщённое (generic) и функциональное программирование.

В языке имеется понятие типа (класса), которое реализуется посредством языковой конструкции «контейнер» (см. ниже), со всеми традиционными свойствами – инкапсуляцией имён, наследованием, полиморфизмом. Имеется возможность задавать абстрактные классы, а также реализовывать между классами отношения множественного наследования.

Любой контейнер или подпрограмма может быть параметризована типом (типами), либо константными значениями. Настройка обобщенных программных единиц предполагает задание конкретных типов и/или значений. Тем самым, в языке обеспечивается полная поддержка парадигмы обобщенного программирования, что позволяет проектировать компоненты программы в виде, максимально независимом от контекстов их использования.

Средства обобщённого программирование, реализованные в языке, представляют определённый компромисс между полным, но весьма громоздким и трудным для использования механизмом шаблонов языка С++ и «облегчёнными», но явно недостаточными для практического программирования средствами типовой статической параметризации Java, C# и Eiffel.

В языке поддерживается необходимый набор средств функционального подхода к программированию, включая лямбда-выражения и замыкания. Эти свойства основаны на трактовке подпрограмм как объектов, а также предполагают свободное использование понятия неизменяемых (immutable) объектов.

Реализация элементов функционального программирования в языке SLang носит в большой степени «инженерный» характер, схожий по ассортименту средств с аналогичными средствами С++ и C#. Поддерживается только минимально необходимый «функциональный» набор. Реализация функциональной парадигмы в полном объёме, свойственная таким языкам, как Haskell или F#, по мнению авторов, не обусловлено потребностями разрабатываемого ПО, потребует радикальной перестройки мышления разработчиков, что может усложнить изучение языка и отпугнёт программистов излишней сложностью.

**Контейнер: модуль, класс и тип в одном флаконе**

Важнейшими концепциями, используемыми при разработке программного обеспечения (ПО), служат понятия атрибутов (данных) и подпрограмм (действий). Атрибуты могут изменяться подпрограммами в процессе работы программы; они образуют ее вычислительный контекст, в то время как подпрограммы задают алгоритм решения задачи. Между атрибутами и подпрограммами есть логические связи, и объединяя атрибуты и подпрограммы в единый именованный контейнер, мы просто фиксируем эту связь. Таким образом, понятие *контейнера* можно считать простым средством агрегации логически связанных данных и действий в единое целое.

Более строго, ***контейнер*** (***unit***) можно определить как поименованную совокупность атрибутов и подпрограмм, которая может быть параметризована типами и/или константами, и может быть использована для задания типов, конструирования новых контейнеров при помощи наследования или для прямого использования атрибутов и подпрограмм данного контейнера в других контейнерах и подпрограммах.

Контейнер можно рассматривать как определение множества данных и операций над ними – то есть, как задание некоторого **типа**. Тем самым, можно определить объект, тип которого будет контейнером. Во-вторых, можно предоставить открытое (общедоступное) содержимое контейнера для **использования** в некотором программном коде, то есть, включить его ресурсы в некоторый контекст. Наконец, атрибуты и подпрограммы контейнера могут (пере)использоваться при создании нового контейнера. Такой механизм носит название **наследования**.

Таким образом, различные варианты использования контейнера приводят к понятиям *типа*, *модуля* и *класса*.

В большинстве современных языков программирования все перечисленные варианты композиции реализуются посредством понятия класса. Так, в С++ класс, все члены которого являются статическими, по существу, представляет собой простую агрегацию атрибутов и подпрограмм (вариант модуля). Аналогичное решение («статические классы») принято в языке C#. Заметим, что для представления модуля на основе класса приходится привлекать не вполне адекватное (обусловленное историческими причинами) понятие «статических» членов, а гибкое использование такого «модуля» (прежде всего, механизм его включения в определенный контекст) отсутствует.

В языке Scala предпринята попытка отделить «модульную» часть класса в специальную конструкцию «объект-спутник» (companion object) с тем же именем, что и класс; однако на использование такого объекта накладываются существенные ограничения (в частности, он должен компилироваться в том же контексте, что и класс, спутником которого он является).

Отдаленными аналогами модуля можно считать механизм пространств имен (namespaces) в языках C++ и C#, а также пакеты (packages) языка Java, однако это крайне слабое средство модуляризации, введенное в языки прежде всего для разрешения конфликтов имен.

Таким образом, преимущества использования единственного понятия класса для задания различных видов программных контейнеров представляются сомнительными как с инженерной, так и с концептуальной точек зрения. Некоторые авторы приводят убедительные обоснования необходимости явного разделения этих понятий. В то же время сосуществование в рамках единой языковой нотации понятий модуля и класса (например, в языках Ada и Oberon) выглядит несколько искусственным – при многих других несомненных достоинствах этих языков.

В языке SLang сохраняются преимущества единой нотации задания контейнеров, с возможностью явного задания *различных способов использования* контейнеров.

**Однородная система типов**

Система типов языка SLang является *однородной*. Это означает, что в языке отсутствует деление на различные категории типов (например, «встроенные в язык» и «определяемые пользователем»). Любой тип, используемый в программе, определяется единообразно, посредством универсальной конструкции «контейнер» (unit). Единственное различие в системе типов заключается в том, что некоторые наиболее часто используемые на практике типы – целый, вещественный, булевский, а также такие структуры, как массивы, списки, словари и т.д. – определены в качестве библиотечных. Для операций над объектами таких типов компилятор может порождать более эффективный объектный код. Типы, определяемые пользователем, используются наравне с библиотечными типами без каких-либо ограничений.

Понятие контейнера предоставляет удобное и универсальное средство создания разработчиком новых типов на основе существующих. При этом допускаются все традиционные методы и практики определения типов, в том числе агрегирование и наследование.

Возможность создавать и использовать в программе некоторый новый тип – необходимое условие гибкости языка и, как следствие, залог его возможностей по адекватной реализации множества реальных структур данных и отношений между ними. Использование существующих типов в качестве составных частей других типов в значительной степени обеспечивает мощность языка и выразительность программ, написанных на этом языке.

Однородность системы типов существенно упрощает понятийный базис языка, делая его стройным, логичным, простым для понимания, тем самым обеспечивая максимально возможную простоту, ясность, недвусмысленность и надёжность программ. В то же время, однородность не ограничивает выразительные возможности языка, предоставляя в распоряжение разработчика полный спектр инструментов для создания структур данных любой сложности.

**Объектный подход и множественное наследование**

Как уже говорилось, язык в полном объёме поддерживает весь сложившийся к настоящему времени комплекс средств объектно-ориентированного программирования: понятие класса (выраженное в конструкции «контейнер»), наследование с возможностями управления характером наследования, полиморфизм с абстрактными классами и методами.

В отличие от многих современных языков, наследование в языке SLang реализовано полностью: контейнер может наследовать свойства нескольких других контейнеров.

Сведение понятия наследования к единичному, характерное для некоторых современных языков (Java, C#) продиктовано, скорее, техническими причинами – желанием упростить реализацию механизма наследования, избавиться от возможных неоднозначностей и сделать программы более понятными для читателя. В то же время, это ограничение существенно сужает возможности разработчиков по адекватной реализации требований к архитектуре программных комплексов.

Семантика наследования в SLang определена таким образом, чтобы избавить программиста от необходимости обеспечивать безусловную корректность полного графа наследования. Контроль корректности привязан к точкам использования сущностей из базовых классов.

**Контрактное программирование**

Язык SLang поддерживает полный спектр механизмов *контрактного программирования*, включая пред- и постусловия для подпрограмм и инварианты контейнеров и циклов. Система поддержки времени выполнения обеспечивает эффективную проверку условий и инвариантов – на этапе компиляции, если это возможно, и/или параллельно с выполнением основной программы.

Подход к проектированию программ на основе понятия *контракта* (Design by contract ©), изначально разработанный и обоснованный Б. Майером и реализованный в языке Эйфель, в настоящее время является общепринятым средством повышения надежности и верифицируемости ПО. Этот подход в том или ином объеме реализован во многих современных ЯП.

**Параллельное программирование**

В отличие от большинства современных языков, где поддержка параллельности реализована на уровне библиотек и носит ограниченный и слабо верифицируемый характер, SLang включает удобный и достаточно надежный механизм распараллеливания программ на уровне самого языка. В языке имеется простой и компактный набор конструкций и спецификаторов для задания многопоточности и синхронизации по доступу к данным. Кроме того, семантика языка допускает автоматическое распараллеливание исполнения.

**Безопасность**

Проблема, связанная с неконтролируемым использованием нулевых указателей («пустых» или «повисших» ссылок), является одной из наиболее распространенных в практике программирования, а также одной из самых опасных по своим последствиям с точки зрения обеспечения надежности программ. В то же время, контроль доступа по таким указателям не имеет удовлетворительного решения в традиционных языках.

В языке SLang проблема пустых указателей трактуется не как самостоятельная проблема, а как часть более общей проблемы некорректной работы с *неинициализированными атрибутами*. Пустая ссылка считается разновидностью неинициализированного атрибута, и в языке имеются механизмы, которые строго ограничивают случаи, когда нам действительно нужны неинициализированные атрибуты, от ситуаций, когда всякая сущность должна иметь определенное значение. В дополнение к этому имеется надежный механизм перехода от потенциально неинициализированных атрибутов к инициализированным – своего рода мостик от «опасного» мира в «безопасный».

**Замечание о синтаксисе**

Обычно, говоря о каком-либо языке программирования, в первую очередь имеют в виду правила составления его конструкций – то есть, синтаксис. Под выражением «выучить язык», как правило, понимают освоение именно этих правил. Более того: многие руководства по новым ЯП в первых же строках как бы успокаивают читателя: мол, если вы знаете язык X, то вам не составит труда выучить и наш язык, так как мы сделали его *очень похожим* на ваш любимый язык X! И в подавляющем большинстве случаев под «похожестью» имеется в виду именно синтаксис.

Понятно, что подобного рода сентенции имеют отчетливый маркетинговый характер, имея своей целью привлечь на свою сторону программистов и их руководителей, интересующихся новинками, но опасающихся чрезмерных усилий и времени на их освоение.

Но это еще полбеды; гораздо важнее (и опаснее!), что такие успокоительные заверения маскируют существенные смысловые, *семантические различия* между языками. В самом деле: если новый язык не несёт в себе каких-то сущностных нововведений, новых понятий, позволяющих решать новые сложные задачи – зачем было придумывать его?

При проектировании языка SLang авторы не ставили задачу следовать повсеместно принятым в настоящее время синтаксическим и лексическим традициям и тем самым «понравиться» программистам. Если некоторое понятие, существенное для языка или предметной области, требует введения новой и непривычной нотации, такая нотация вводится в язык. То же относится к «стилю синтаксиса» – распространённым в настоящее время традициям обозначений блоков (областей действия), служебным словам, разделителям и т.д. Основным мотивом выбора лексики или синтаксиса для той или иной конструкции языка служит её наглядность, ясность для понимания и удобство чтения. Авторы следуют парадигме, впервые высказанной создателями языка Ада, согласно которой программы в той же степени являются средством общения между людьми, как и средством общения людей с компьютерами.

В то же время в следовании таким установкам авторы не собираются становиться фанатиками и не хотят доходить до абсурда. Конструкции языка не должны быть излишне громоздкими и многословными, удивлять разработчиков и читателей программ своей экзотичностью и заставлять их прикладывать чрезмерные усилия для их освоения и понимания. По этой же причине в языке отсутствуют средства его произвольного изменения, такие, как макросы, синтаксические «расширители» и средства задания собственных операций.

**Приложение 2  
Язык SLang и технология блокчейн**

В связи с нарастающей популярностью технологии блокчейн в проект SLang планируется включение ряда инструментов разработки приложений как под существующие реализации этой технологии, так и под приватные сети, создаваемые с помощью будущих инструментов языка SLang.

В качестве первой платформы для интеграции выбран проект Ethereum (https://www.ethereum.org), прежде всего ввиду его распространенности и доступности детального описания Ethereum Virtual Machine (EVM). Перенос кода SLang-программ в байт-код EVM не составит больших трудностей и будет содержанием первого этапа реализации.

В то же время в своём нынешнем виде экосистема Ethereum недостаточно стабильна и нуждается в инструментах для упрощения разработки приложений с использованием смарт-контрактов.

В данный момент проекты с использованием сети Ethereum представляют собой трехслойную модель:

* Фронтальная часть приложения создается на языке JavaScript и отвечает за взаимодействие с пользователем и связь со смарт-контрактами Ethereum.
* Создание самих смарт-контрактов происходит на одном из специальных языков (на момент написания этого текста самым популярным является Solidity).
* Если в приложении требуется более сложная бизнес-логика, она реализуется (в виде «бэкенда») на некотором третьем языке.

Таким образом, создание приложения предполагает использование трёх различных языков, каждый из которых обладает собственным окружением и экосистемой. Кроме того, язык Solidity, который является наиболее развитым языком под EVM, подвергается обоснованной критике (https://news.ycombinator.com/item?id=14691212).

В качестве решения этой проблемы для всех этапов разработки так называемых DApp (Decentralized Applications) предлагается использовать язык SLang. С помощью библиотек, позволяющих разрабатывать смарт-контракты внутри проекта на языке SLang, можно существенно уменьшить время разработки подобных приложений за счет исключения «прослойки» в виде нескольких инструментов разработки и развертывания смарт-контрактов в сеть Ethereum – всё это будет скрыто от разработчика в системных библиотеках.

Однако, недостаточно просто добавить поддержку написания смарт-контрактов. Радикальное улучшение архитектуры будет достигнуто за счёт создания на базе проекта SLang целого комплекса библиотек и фреймворков для разработки децентрализованных приложений, разработки инфраструктуры приватных блокчейнов, как, например, Hyperledger и Hyperledger Composer (https://www.hyperledger.org/), а также создание удобного тестового окружения без неудобств для разработчика.

**Приложение 3  
Об авторе представления**

**Евгений Зуев**, сотрудник университета Иннополис, к.ф.-м.н., руководитель лаборатории языков программирования, компиляторов и операционных систем. Специалист в сфере проектирования и реализации компиляторов ЯП.

Среди проектов, выполненных с его участием и под его руководством, в частности, следующие:

1998 Компилятор переднего плана языка С++ (стандарт ISO 1998) для компании ACE, Нидерланды. Работа выполнялась в МГУ им. М.В. Ломоносова.

2000 Полный ISO-совместимый компилятор С++ для ряда отечественных микропроцессоров (компания Интерстрон, Москва).

2005 Компилятор языка Zonnon для платформы Microsoft .NET (Федеральный технологический институт, Цюрих, Швейцария).

2009 Портирование компилятора языка Scala с платформы Java Virtual Machine на платформу Microsoft .NET (Федеральный технологический институт, Лозанна, Швейцария). (Проект не завершен.)

2011 Проектирование и реализация языка Zeppelin для разработки финансовых приложений (компания Meyer & Nicca, Швейцария).

2013 Компилятор языка JavaScript с продвинутой функциональностью статического анализа (проект Hermes, Samsung R&D Institute, Москва)

2015 Экспериментальная реализация языка Apple Swift для ОС Tizen (Samsung R&D Institute, Москва).