Компонентный ассемблер. Часть 2. Дух языка

> *Общеизвестно, что 2018 год стал переломным годом для русской философии. В этом году были сформулированы окончательные (ultimate) ответы на «вечные» русские вопросы «Кто виноват?» и «Что делать?» (ответы приписываются С. Лаврову и капитану «Беззаветного», см. соответствующие разделы учебника). Начиная с 2019 г. основным вопросом русской философии стал вопрос: «Как сделать?»*
>
> «Краткая история русской философии», Галактопедия, 2119 г.

Введение

Эта статья является второй (и не последней) частью длинного рассуждения, итогом которого должно быть описание экспериментального языка программирования. Так как автор в гораздо большей степени является практиком, чем теоретиком, одновременно со статьями идет разработка компилятора и инструментов разработки, на базе среды разработки Вир [1].

Статьи могут опережать создание компилятора или описывать уже разработанные и проверенные части, но, в целом, написание статей и изготовление инструментов идет параллельно и эти работы должны одновременно прийти к финишу.

В первой части [2] было приведено обоснование нужности экспериментального компонентного ассемблера и общие требования к языку.

Перейдем от требований верхнего уровня к самым основам языка. Я не хочу использовать здесь слова «синтаксис» и «семантика», так как скорее речь пойдет о чем-то вроде «духа» языка, который, естественно, будет потом выражен (и спрятан) в формальных правилах синтаксиса и менее формальных семантических правилах, но это будет потом. А сначала надо его уловить и описать.

Для меня, главное в языке: простота (as simple as possible, but not simpler), однозначность понимания и возможность естественного выражения мысли. Если есть главное, то остальное, включая надежность и производительность, достигается гораздо легче.

Поговорим о главном…

Простота и однозначность понимания

Когда я читаю программный текст, я хочу его понимать. О, как много в этой фразе для меня отозвалось….

- Во-первых, мы читаем программный код гораздо больше раз, чем пишем. Особенно, если это не разовый код, а код, который мы пишем надолго, желательно навсегда.
- Во-вторых, читает код больше людей, чем пишет.
- В-третьих, читать мало, нужно понимать.

Понимание – штука сложная, потому я ограничусь локальной областью: я хочу однозначно понимать смысл фрагмента исходного текста, который могу охватить одним взглядом. Я не рассматриваю понимание алгоритма, который может занимать много экранов (страниц) текста, тем более понимание работы компоненты или программной системы целиком. Об этом надо говорить отдельно.

Что очевидно мешает однозначному пониманию фрагмента?

Как минимум:

- Перегрузка операций (operator overloading)
- Наследование реализации (implementation inheritance)
- Неявный импорт (не квалифицированный импорт)

Все три пункта приводят к тому, что, глядя на имя (или знак операции), невозможно однозначно понять, какая сущность соответствует этому имени: «не верь глазам своим».

В итоге, язык, в котором есть все три механизма (например, C++), становится write only языком – писать программы на нем можно, читать нельзя.

Замечание: я вовсе не хочу сказать, что, проектируя язык, надо учитывать только readability. Свойство writability тоже важно, но оно, безусловно, должно иметь приоритет ниже, чем readability.

Вернемся к компонентному ассемблеру. Так как читабельность и простота понимания – это первоочередное требование, то в языке не может быть:

- перегрузки операторов
- наследования реализации (см. раздел ООР)
- неявного (неквалифицированного импорта). Если из модуля А (пространства имен, пакета) импортируется объект с именем Б, то он доступен только с указанием имени А (или псевдонима А).

Русский язык

Я хочу писать программный код на русском языке. Просто потому, что думаю я на русском. Необходимость выражения мысли на английском (для обычных языков программирования) приводит к потере эффективности.

Английский бывает удобнее, как ни странно тем, что он не родной. Используешь какое-то слово, как Шалтай-Болтай:

> *— Когда я беру слово, оно означает то, что я хочу, не больше и не меньше, — сказал Шалтай презрительно.*
> *— Вопрос в том, подчинится ли оно вам, — сказала Алиса*.

И в этом заключается подвох, потому что для других это слово может означать не совсем то же самое или совсем не то же самое.

Замечу, говоря про русский язык, я имею в виду русские имена (идентификаторы). Именно они должны быть обязательно на русском. Что же касается ключевых слов – то это «иероглифы», которые могут быть записаны на любом языке, лишь бы они сразу были понятны.

Например, я не вижу смысла переводить nil (или null) или div на русский. Впрочем, один смысл может быть – минимизация переключения между русской и английской клавиатурой. Но это решается макросами в программном редакторе.

Для ключевых слов я использую правило, если русское ключевое слово (или служебный идентификатор) выглядит естественно, используем его (примеры: если, пока, завершить, константа), если нет, заменяем на привычное английское слово или на подходящий символ.

Естественно, что если мы пишем на русском, то язык должен использовать Unicode, а не ASCII. И это сразу дает дополнительные возможности, например, использование символов, которых нет в ASCII, например: →.

Понятные идентификаторы

Это ключевой вопрос для увеличения читабельности и упрощения понимания текста. Все извращения в способе записи идентификаторов, вроде: CompileModule или compile_module – это всего лишь костыли, читать которые трудно.

Я хочу писать имена привычным образом, с пробелами и использованием любых знаков:

Примеры понятных идентификаторов:

Заменить все вхождения значка в строке
Компилировать модуль
Существует файл?

В Вир/а1 используется простое решение – так как в Юникоде много парных кавычек, одна пара выделяется для идентификаторов: «Заменить все вхождения значка в строке», и т.п.

Без кавычек в Вир/а1 можно писать простые идентификаторы (без пробела). В простых идентификаторах кроме букв и цифр можно использовать “-“ и “№”, например: №-символа (если знак минус используется как знак арифметической операции, он должен быть отделен от идентификатора пробелом. И это обычный способ увеличить читаемость текста).

Можно разрешить использовать в простых идентификаторах другие символы, которые не является буквой, цифрой или служебным символом, но этих, на мой взгляд достаточно.

Замечу, что естественная запись идентификаторов на русском языке использовалась в языке ЯРМО [3] еще в 70-х годах прошлого века.

Простой и лаконичный синтаксис

Ничего нового не придумываю, синтаксис Go [4] считаю, в целом, подходящим, достаточно читабельным, лаконичным и достаточно общепринятым. Поэтому использую его с некоторыми изменениями в тех местах, которые считаю необходимыми (см. оператор присваивания).

Программисты, знающие Go или Си (и русский язык), смогут читать текст на Вир/а1 без напряжения.

```
если «Существует файл?»(имя-файла) {
        «Компилировать модуль» (имя-файла)
}
```

Оператор присваивания

Выбор способа записи оператора присваивания основывается в изрядной степени на ощущении правильности (можно сказать, на личном предпочтении). Соответственно, он является наименее обязательным.

На мой взгляд, привычное всем левое присваивание (переменная слева) плохо отражает естественное движение мысли. В большинстве случаев, я сначала думаю, что сделать, а потом куда положить результат.

Поэтому, запись с правым присваиванием:

Выражение → Переменная

является более естественной.

Пример:

x + y → z

«Существует файл?»(имя-файла) → признак: лог // присваивание с объявлением переменной

В Вир/a0 (на котором я программирую много лет) тоже используется правое присваивание, так что я давно убедился в удобстве такой записи. Замечу, мысль о правом присваивании появилась при чтении описания языка ЯРМО [3]. Первый раз я познакомился с ЯРМО еще в студенчестве, и с тех пор помнил об этом.

Вторая причина, по которой я думаю о синтаксисе оператора присваивание, это выверт мозгов, который происходит при записи присваивания в языках с Си-синтаксисом (полагаю, что происходит в мозгах любого человека с математическим образованием). В программном коде

z = x + y

знак “=” вовсе не обозначает равенство (в математическом смысле), а обозначает совсем другую операцию. Использование “=” есть ложь.

А использование для сравнения == (а в некоторых языках и в каких-то случаях еще и ===), ломает голову еще сильнее: два равна, три равна, «понять нельзя, надо запомнить».

Можно сделать скидку на то, что авторы языков мучились в рамках ASCII, и искали компактную запись (кроме Н. Вирта – «:=»), но зачем нам в XXI веке возводить мучения в правило и создавать помехи собственному рассуждению?

Стиль: отбрось лишнее, потом работай

В Вир/a0 кроме оператора **если** есть оператор **проверить** – условный выход из функции:

проверить Условие **завершить** [Выражение]

Используется, как правило, в начале функции для проверки условий применимости функции, например:

```
{
    проверить Условие1 завершить
    проверить Условие2 завершить
    проверить Условие3 завершить

    // основное тело функции
}
```

Такой линейный текст существенно проще для понимания, чем обычно используемые варианты:

```
{
    если Условие1 { завершить }
    иначе если Условие2 { завершить }
    иначе если Условие3 { завершить }
    иначе {
          // основное тело функции
    }
}
```

Или

```
{
    если ! Условие1 && ! Условие2 && ! Условие3 {
          // основное тело функции
    }
    иначе // обработка ошибки
}
```

Линейность текста упрощает чтение и понимание «с первого взгляда». Естественно, что можно использовать последовательность операторов **если,** но оператор **проверить** дает большую наглядность.

В каком-то смысле, оператор **проверить** является инструментом реализации «главного маршрута алгоритма» в Дракон диаграммах [5], с помощью этого оператора главный маршрут очищается от проверок.

Решение о включении оператора **проверить** в Вир/a1 пока не принято, перед этим я собираюсь тщательно изучить инструменты обработки ошибок в Go, в том числе новые предложения для Go 2.

Типизация

… расцветали яблони и груши ….
… пусть расцветают сто цветов …

Недавно я наткнулся на полемику (holy war) на Хабре, предметом спора в которой было – что лучше: статическая или динамическая типизация. Был сильно удивлен, так как такая полемика сродни спору между сторонниками дождливой и солнечной погоды (можно еще вспомнить яйца Джонатана Свифта).

Оба способа типизации необходимы и полезны и, кроме пары статическая/динамическая типизация есть еще пара именная/структурная.

Все эти разновидности (или способы) типизации необходимы:

- Для межкомпонентного взаимодействия нужна динамическая типизация (точнее динамическое знакомство[1]) и структурная типизация (так как именная накладывает слишком сильные ограничения). Подробнее см. [2]
- Статическая и именная – дают максимальную защиту и производительность
- Динамическая дает гибкость
- И так далее

Все разновидности нужны и будут в том или ином виде реализованы в Вир/a1.

В Вир/a1, скорее всего, не будет аналога созданию нового типа в Go (с именной совместимостью), таких как:

```
type (
      Груши int
```

[1] Динамическим знакомством я называю динамическую проверку, которая выполняется один раз, после чего работает статика. Пример динамического знакомства (помимо динамической типизации) – получение адреса функции из DLL.

```
        Яблоки int
    )
```

То есть компилятор не будет запрещать складывать груши с яблоками, все же Вир/a1 – это ассемблер. И, соответственно, не будет возможности добавлять методы к таким отдельным типам (методы только для Груш).

Из того, что является обязательной частью типовой системы в Вир/a1 – это метаданные и рефлексия. Об этом надо говорить отдельно.

ООР

Разработчика на C++ смело уподоблю пожарнику, как тот, так и другой никогда не знают, где загорится и по какой причине.

iКозьма

Об ООР я писал уже многократно [6], краткая выжимка:

- ООР – это хорошо, соответствует человеческому видению и управлению миром
- CLOP (Class Oriented Programming) – это плохо
- Go – хорошо, C++ - плохо

Соответственно, в Вир/a1:

- Интерфейсы (Go like)
- Возможно, отдельная конструкция для безопасного адреса функции (safe pointer to function)

Требования к IDE

Потом хозяйством занялась,
Привыкла и довольна стала.
Привычка свыше нам дана:
Замена счастию она.

А.С. Пушкин

Обычно разработчики языков программирования не выдвигают требований к IDE, хотя могут определять требования к системе сборки и исполнения.

Но, так как редкий язык программирования живет в вакууме, есть смысл выдвинуть требования и к IDE тоже. Тем более, что, на мой взгляд, разработчики современных IDE списывают плохие идеи друг у друга, и не думают о том, чтобы обеспечить разработчикам минимальные удобства.

Рассмотрим самые простые, я бы сказал, бытовые требования к редактору кода для Вир/a1:

- Редактор кода основан на Юникоде, то есть позволяет использовать любые символы Юникода.
- Редактор кода использует обычные текстовые шрифты – `не моноширинные`, потому что читать текст, набранный обычными шрифтами, удобнее.
- Редактор кода позволяет делать обычные текстовые выделения – bold, italic, цвет и размер шрифта, фон и т.д. Отсутствие обычных средств редактирования текста в программных редакторах вообще ничем разумным объяснить нельзя.

  Почему разработчик, когда пишет документацию, может использовать все средства форматирования, а когда пишет программный код, существенно ограничен в форме выражение мысли? Костыль в виде подсветки синтаксиса почти ничего не дает, так как выделять нужно фрагменты разные, с точки зрения семантики кода, например,

отладочные фрагменты, служебные и т.д. Выделение синтаксиса редко помогает опытному разработчику, но всегда мешает.

Поднимемся на уровень выше. Почему мы представляем программный код в виде линейного текста? Ничем, кроме привычки и косности мышления, я это объяснить не могу.

Замечу, что разработчики IDE понимают, что работать с длинным линейным текстом неудобно и добавляют инструменты сворачивания части текста (folding).

По сути, сворачивание текста делает из линейного текста структурный. Структурные тексты люди придумали сотни лет назад, и до сих пор книги состоят из разделов, глав и абзацев. Открою еще одну тайну разработчикам IDE, которые, видимо, книги не читают: электронные книги снабжаются удобным оглавлением с возможностью перехода по ссылкам.

Вот и сейчас, когда я пишу эту статью, слева я вижу оглавление статьи в виде дерева, и это удобно.

Почему, когда я пишу программный модуль, я не могу его сразу разбивать на разделы, главы, функции, в конце концов, и видеть структуру одним взглядом? За что я так наказан?

Тут я, конечно, нагнетаю страсти, так как, работая в Вире, я вижу структуру и именно мне работать удобно, но могу же я позаботиться об остальных разработчиках?

Замечу еще, что в отличие от книг, программный код мы редко читаем подряд, поэтому структура и оглавление для программного кода становится еще более важной.

Линейное выкладывание кода в наше время вообще не имеет смысла, так же, как и нумерация строк, это пережиток прошлого века, медленных компьютеров и консольных приложений.

Понятно, что в случае программного сбоя нужно, чтобы было определено место, в котором локализована ошибка. IDE может использовать для этого номера строк, но это внутреннее дело IDE, мне же нужно, чтобы я мог увидеть локализованный фрагмент текста.

Чтобы не затягивать разговор об IDE, завершу этот раздел краткой выжимкой:

- Текст должен быть структурным (иметь оглавление)
- Линейный текст вообще не имеет смысла, так как подряд программный текст приходится читать очень редко.
- Редактор программного текста должен давать удобные возможности оформления текста, в том числе, и подсветку синтаксиса, если это кому-то удобно.
- Я не говорю о других, в разной степени полезных, инструментах IDE. Если коротко, то я за все хорошее, против всего плохого. Возможно, что мои представления о хорошем и плохом не совсем общеприняты, но это тема для отдельного разговора.

Заключение

Надеюсь, что несмотря на то, что язык Вир/a1 практически остался за скобками, понимание о «духе» языка и направлении движения появилось.

Если кто-то разочарован, что были не упомянуты модные или любимые конструкции, типа лямбда-исчисления, generic-типов или обработки исключений, напомню: make it simple, as possible, but not simpler.

Включение любой такой конструкции должно быть обосновано, то есть она должна попасть в категорию: but not simpler, чем…

Об этом мы еще успеем поговорить. Продолжение следует.

Список литературы

[1] Недоря А.Е. “Технология разработки мультиплатформенных программ на основе явных схем программ” // http://digital-economy.ru/stati/tekhnologiya-razrabotki-multiplatformennykh-programm-na-osnove-yavnykh-skhem-programm, 2018

[2] Недоря А.Е. “Компонентный ассемблер для цифрового пространства” // http://digital-economy.ru/stati/komponentnyj-assembler-dlya-tsifrovogo-prostranstva, 2018

[3] Гололобов В.И., Чеблаков Б.Г., Чинин Г.Д. “Описание языка ЯРМО. Машинно-независимое ядро”, Новосибирк, 1980 (Препринт 247 ВЦ СОАН СССР)

[4] The Go Programming Language Specification // https://golang.org/ref/spec

[5] Пароджанов В.Д. “Дружелюбные алгоритмы, понятные каждому“ – М.: ДМК Пресс, 2016.

[6] Недоря А.Е. “CLIP/CLOP vs pure OOP” // http://алексейнедоря.рф/?p=152