Дневник разработки Тривиля

Дневник состоит из записей в блоге http://алексейнедоря.рф, которые писались по ходу разработки Тривиля и собраны здесь **без редактирования**. Часть материалов была использована в серии статей: http://digital-economy.ru/temy/тривиль

Замечание: я не переношу в этот текст комментарии читателей блога. Их было много, и многие из них были очень полезны для работы. Увы, я просто не потяну объем работы. Впрочем, пишите, и я вставлю ваш комментарий, который вы считаете принципиально важным.

В текст добавлены сноски (январь 2025), обычно для того, чтобы пояснить, что было сделано иначе.

20.11.2022. ЯВД: Куда и как идти?

Направление движения для меня определено — Интенсивное программирование и так же определена промежуточная точка — семейство языков «Языки выходного дня».

Разработка такого уровня экспериментальности может быть только итеративной — сделали шаг, проверили, вернулись, переделали. Поэтому неверно пытаться сначала полностью разработать ЯВД, а потом идти дальше. Надо как можно быстрее дойти до шага «проверить».

Что для этого нужно? Хотя бы один язык, пусть очень черновой и минимальная экосистема: компилятор, система выполнения, библиотеки.

Тогда вопрос «как идти» можно упростить до «какую минимальную экосистему построить в первую очередь». Требование минимальности исходит из малого количества ресурсов. Я рассчитываю на одну голову и одну пару рук, как это было при разработке системы «Вир». Если еще кто-то подключится, хорошо, если нет: «делай что должен, и будь что будет».

Итак экосистема: язык + компилятор + среда исполнения + минимальный набор библиотек.

Начну с языка. Как я уже писал, семейство предположительно состоит из

- языка уровня 1: динамически типизированный язык (желательно с возможностями gradual typing);
- языка уровня 2: статически типизированный язык с динамическими возможностями и сборкой мусора;
- язык уровня 3: статически типизированный язык с ARC;
- язык уровня 4: статически типизированный язык с ручным управлением памятью.

Эта классификация уже показала свою применимость. Кратко мы говорим, например, L2 (язык уровня 2), и это сразу дает понимание.

Другой взгляд на те же уровни через область применения:

- L1: приложения
- L2: приложения и высокоуровневые библиотеки;
- L3: приложения, критические по производительности или безопасности, и библиотеки;

- L4: ОС, драйверы, низкоуровневые библиотеки.

Понятно, что классификация условная, особенно для языков-монстров типа C++. Кроме того, 4-х уровней явно недостаточно, поэтому надо использовать опыт нумерации строк в Бейсике :) и добавить пару нолей, то есть добавить подуровни.

Если попытаться разбросать существующие языки (не сильно задумавшись и очень примерно):
- Typescript — L100
- Go — L220
- Swift — L300
- Rust — L400
- C++ — L420
- C — L440
- LLVM IR — L470
-

С какого языка начать? Мой опыт говорит о том, что для проверки языка принципиально важно написать на языке компилятор самого языка (bootstrapping).

Значит, язык должен быть
1. маленький и простой с точки зрения компиляции;
2. достаточный для того, чтобы на нем написать компилятор (в том числе в последствии и для других языков семейства);
3. достаточный для экспериментов с компонентами — впрочем, это не сразу.

На мой взгляд, надо начинать с языка L3[1]. Отсутствие GC упрощает экосистему (хотя потребует weak pointers или чего-то аналогичного).

И тогда последовательность шагов примерно такая.

Этап 1:
- минимальная грамматика языка
- парсер
- минимальный семантический анализ
- кодо-генератор. Думаю, что строить надо код на C, что достаточно просто и еще упрощает взаимодействие с C библиотеками.
- система выполнения на C/C++
- минимальный набор библиотек (в начале обертки для C libraries)

Этап 2:
- bootstrap
- система тестирования
- улучшения

Дальше развитие языка, добавление других языков семейства, эксперименты и тому подобное.

И последний вопрос — на каком языке писать первый компилятор?

Я предпочитаю Go, основные соображение — простой и знакомый язык, кросс-платформенность, бесплатные средства разработки. Впрочем, это почти не важно, так как он нужен только на первом

[1] Позже я решил, что все же L2.

этапе.

20.11.2022. Язык Тривиль

Итак, L3 язык. Нужно рабочее название. Первое, что пришло мне в голову — Триэль. Но это слишком напоминает известный анекдот — «Шуба с трихуелью есть?».

Поэтому: Тривиль. То есть тривиальный язык третьего уровня. Перестанет быть тривиальным, можно будет сменить название.

Что нужно в тривиальном языке:

- модули или пакеты, скорее модули в стиле Оберона, чем пакеты в стиле Го, потому что проще[2].
- импорт/экспорт
- переменные, константы, функции
- примитивные типы — знаковые целые, вещественный, логический, символ и строка
- конструкторы типов — динамические массивы и классы
- операторы: присваивание, условный, выбор, цикл
- выражения: обычный набор арифметики и логики, композиты массивов и классов, проверка типа, конвертация

Вроде все что нужно. Как ни странно, думать надо скорее над лексикой, чем над синтаксисом. Просто потому, что все хочу на русском. И хочу идентификаторы с пробелами.

Следующий шаг — грамматика.

20.11.2022. Тривиль: грамматика — первый заход

Первый набросок: грамматика[3] в формате ANTLR4 (написал с листа и никак не проверял).

Нет лексики, не расписаны выражения, и вообще очень предварительно. На мой взгляд, в целом неплохо, вот только ключевое слово ‘пер’ или ‘перем’ вместо ‘var’ мне не нравится. Может быть стоит сразу найти хорошее слово для ‘mut’ и ‘imm’ — mutable, immutable, так как точно понадобится.

Язык пока выглядит как урезанный Го с добавлением Оберона. Впрочем, дальше расхождений будет больше. Выглядит примерно так, вполне тривиально:

```
модуль Привет

импорт «консоль»

вход {
  консоль.печать(«Привет, мир!»)
```

[2] Многофайловые модули полезней, и их удалось сделать очень дешево, поэтому - с стиле Go.
[3] См. https://gitflic.ru/project/alekseinedoria/trivil-0/doc/тривиль/история/2022-11-20-grammar.txt

```
}
```

Я сделал приватный проект в gitflic — думаю туда выкладывать какое-то время, а потом запрошу сделать его публичным.

26.11.2022. Тривиль: описание переменных

Вариант 1: традиционный

Ключевые слова: перем или пер для var, знач или пусть для let.

Чем не нравится: длинно, не очевидно, не привычно.

Длинно: можно справится с помощью макросов в редакторе.

Вариант 2: привычные англоязычные иероглифы: var и let.

Чем не нравится: переключение регистра, так как имена (особенно функций) я хочу называть по русски и с пробелами.

Переключение регистров: тоже макросы

Вариант 3: расширить Go short variable declarations

Go: x := 1 заменяет var x = 1

Делаем 3 значка:

- присваивание существующей переменной, например :=
- описание переменной, например ::= (с возможностью явно задать тип) x: цел := 1
- описание значения, например :-

Чем не нравится: чем больше я писал на Го, тем меньше использовал короткую форму, так как понимаемость текста var x = 1 заметно выше, чем x := 1

Вариант 4: перейти к правому присваиванию (следом за ЯРМО), снова в стиле короткого присваивания

Например (значки условные):

- 1 -> x
- 1 => x (с явным типом 1 => x: цел)
- 1 =:> x

Чем не нравится: тем же, что и вариант 3 — не очевидность понимания. Все же мы читаем слева направо.

Вариант 5: использовать какие-то «иероглифы», например, греческую букву в качестве ключевого слова

Заглянем в греческий словарь:

переменный => μεταβλητή — μ
постоянный => σταθερή — σ
Вот только эти буквы не бросаются в глаза, но можно не привязываться к переводу слов и использовать, например:

- Ψ — var
- Ω — let

То есть писать: `Ψ счетчик: цел = 1`

Ну или другие буквы или знаки Unicode, благо есть из чего выбрать.

Конечно, придется делать макросы для вставки, но все остальное, на мой взгляд, здорово — коротко и наглядно.

Для кого-то может быть проблемой переход от ASCII к Unicode, но это только программисты застряли в прошлом веке…

Вывод: я склоняюсь к варианту 5 или 1. В обоих случаях надо еще покрутить, какие символы/слова использовать.

26.11.2022. Тривиль и null safety

У меня нет сомнений, что любой язык, чуть более развитый, чем Тривиль должен быть null safe. Что же касается Тривиля, то здесь не так однозначно. Стоит ли вносить null safety? Будет ли компилятор сложнее? Или проще?

И второй вопрос — а если делать null safe в Тривиле, то как?

Есть два основных подхода (или больше?):

- через разделение nullable и non-nullable типов — Kotlin, Swift, C# (T?)
- через вариантные типы (unions, enums) — Typescript (T | null), Rust (Option based on enums)

Исходя из соображения простоты реализации Тривиля — лучше делать nullable/non-nullable типы. Причем, nullable можно делать только ссылочные типы, расширяя потом в НеТривиле.

Но есть и другое соображение — если объединенная типовая система семейства языков будет включает вариантные типы, то null safety, вроде бы, естественно делать через них. Чтобы потом не переделывать. Или нет?

Например, в Swift есть и enums и nullable типы, то есть, вроде бы, некое излишество.

Итого, главный вопрос — разделять понятия или выражать одно (nullable) через другое (variant types)?

27.11.2022. Тривиль: грамматика второй заход

Второй заход: весь синтаксис[4] (вроде бы), кроме определения «разделителя» — нужен для избавления от лишних ‘;’ и нет лексики, где определение идентификатора — это самое интересное.

Я нашел пригодное решение для переменных, слово ‘пусть’, потом для assign once переменных можно использовать ‘дано’:

```
фн Факториал(ч: цел64): цел64 {
  пусть рез: цел64 = 1
  пока ч > 1 {
    рез := рез * ч
    ч--
  }
  вернуть рез
}
```

И решение, которое объединяет конверсию (conversion) и контроль типа (type assertion)

```
пусть pi = 3.14
пусть упрощенное_pi = pi ┌цел64┐
```

Эти странные скобочки мне нравятся[5], но критерием истины является практика, так что посмотрим дальше, приживется или нет.

Еще один заход — теперь в лексику и можно писать компилятор. Думаю, что Тривиль-0 компилятор сделать однопроходным.

30.11.2022. Тривиль — вопросы и уточнения

1) Оператор return — как он должен называться в русскоязычном языке?

- вернуть? — не очень хорошо, если результата нет (процедура, а не функция)
- возврат?
- другое?

2) Преобразование типа и динамическая проверка типа

Я не хочу делать две разных конструкции, как это сделано в Go, где есть

- конверсия (conversion): int64(x),
- и динамическая проверка типа (type assertion): x.(T).

Если один синтаксис, то надо сделать возможным использовать эту обобщенную конверсию на уровне PrimaryExpr или, как это называлось в Модуле/Обероне — designator. Я перебрал множество вариантов и остановился на вот таких скобках: ‹ › (U+2039, U+203A), то есть

[4] См. https://gitflic.ru/project/alekseinedoria/trivil-0/doc/тривиль/история/2022-11-27-grammar.txt

[5] Все мои попытки использовать Юникод-символы разбились о простую помеху, notepad++ не умеет делать макросы с такими символами, увы, только ascii.

- x‹цел64› — конверсия
- x‹Узел› — динамическая проверка типа, а по сути конверсия к типу Узел

Тогда эти скобки останутся для обобщенных типов: « ».

3) Null safety

Я решил, что надо делать с самого начала, так как если добавлять потом, придется исправлять уже написанный текст .

Долго маялся способом записи и тем, как лучше читается:

- сначала тип, потом признак nullable — как в Kotlin/Swift: T? (постфиксная запись)
- или наоборот, сначала признак, потом тип (префиксная запись).

Это существенно для составных типов, например, для массивов, если использовать постфиксную запись: `[]цел64?` — к чему относится признак — к массиву или к элементу массива?

Решил, что префиксная лучше читается и придумал такую запись:

`мб Тип` — где мб сокращение от «может быть».

Тогда для массивов:

- `мб []Узел` — массив может быть не задан (равен null в тех терминах), а элемент не может
- `[]мб Узел` — массив всегда задан, а элемент может быть не задан.
- `мб []мб Узел` — и массив и элемент могут быть не заданы.

Вроде неплохо, а дальше практика покажет.

А для выражений надо добавить операторы ‘?’ и ‘!’, такие же как в Kotlin/Swift .

4) Идентификатор

Расписал идентификатор с пробелами, ‘-‘, ‘№’ и вопросительным, восклицательным знаком в конце. Примеры:

- `Паниковать!`
- `Делится на три?`
- `проверка-массива`
- `пока № > 0 { №– }`

5) Добавил в лексику «модификатор», то что обычно называется annotation или decorator. Нужно, для того, чтобы отметить заголовок «внешней» функции и «пред-описанной» функции — нужно, так как первый компилятор хочу сделать однопроходным.

Собственно, практически все. Можно писать компилятор.

Моя оценка разработки компилятора — две рабочие недели, 80 часов[6]. Понятно, что из-за того, что работа будет с перерывами, то займет больше и растянется на несколько месяцев.

---

[6] Я балдею, как это парень (ну то есть, я, конечно, но 2 года назад), точно оценил трудоемкость компилятора! Всегда бы так!

11.12.2022. Тривиль — первая программа скомпилирована и исполнена

Программа:

```
модуль x
@внешняя фн tri_welcome()
вход {
  tri_welcome()
}
```

Скомпилирована в C:

```
#include «trirun.h»
#include «x.h»
int main() {
   tri_welcome();
   return 0;
}
```

и выполнена:

```
D:\0-Projects\trivil-0>trivil x.tri
Тривиль-0 компилятор v0.0
Без ошибок

D:\0-Projects\trivil-0>cd _genc

D:\0-Projects\trivil-0\_genc>clang x.c trirun.c

D:\0-Projects\trivil-0\_genc>a.exe
Trivil!
```

Функция `tri_welcome` написана на C и печатает: Trivil!

Трудозатраты на компилятор до этого момента — примерно 25-30 часов, не пытался точно засекать. Я вполне доволен скоростью разработки.

Компилятор делаю в самом удобном для меня стиле — сначала делаю «скелет»[7], потом наращиваю «жир».

Скелет готов, в него входит[8]:

- Сканер — практически полный, нет экспоненты у вещественных.
- Парсер — часть грамматики: модуль, функция, оператор только один, выражение — только вызов. Строит AST
- Семантика — один обход AST — построение таблицы имен и поиск, типы не проверяются

[7] Тут надо пояснить - это скорее не скелет, а сквозной проход - “стрела”. Компилятор сразу сделан полностью, но с минимальной функциональностью.

[8] Исходя из этого списка, я уже отказался от мысли делать однопроходный компилятор, о чем писал 30.11.2022

- Генерация — только того, что есть в AST
- Обработка ошибок вполне рабочая

Далее наращивание возможностей:
- более-менее полный набор операторов и выражений. Проверка: факториал
- импорт. Проверка: библиотека консольной печати
- массивы и классы

16.12.2022. Тривиль — факториал

Компилируется и работает:

```
1   модуль x
2
3   @внешняя фн print_int(a: Цел)
4
5   фн Факториал(n: Цел): Цел {
6     если n <= 1 { вернуть 1 }
7     вернуть n * Факториал(n-1)
8   }
9
10  вход {
11    print_int(Факториал(5))
12  }
13
```

На этом создание скелета компилятора закончено. Теперь пойду более последовательно — парсер, семантический анализ, генерация.

17.12.2022. Тривиль — первый кусочек не-совсем-тривиля

Маленький кусочек не совсем тривиального: в идентификаторе можно использовать пробелы и завершать его знаками ‘?’ и ‘!’.

Например:
- `фн Операция завершена?(): Лог`
- `если Операция завершена?() & ответ = Беда! { Тревога!() }`

Кроме того, в буквы добавлен, кроме привычного подчеркивания ‘_’, знак ‘№’:
- `пока № > 0 {  №– }`

И наряду с пробелом можно использовать минус ‘-‘:
- `номер-метода`
- `номер-поля`

На мой взгляд, читается намного лучше, чем `номер_метода` или `номерМетода`.

Вот правила:

```
ident
    : word ((' ' | '-') word)* punctuation?
    ;

word
    : letter (letter | digit)*
    ;

letter
    : unicode_letter
    | '_'
    | '№'
    ;

punctuation
    : '?'
    | '!'
    ;
```

В Вире я уже использовал идентификаторы с пробелами, ‘-‘ и ‘№’, и пришел к выводу, что это существенно увеличивает читаемость текста.

Впрочем, в Вире правила были другие, и можно было, например, использовать запятую в имени функции: «Если нажата кнопка, отправить сообщение». В Тривиле это приводит к неоднозначности в списке аргументов. Впрочем, еще будет время об этом подумать.

18.12.2022. Тривиль — планомерное движение

После перехода от изготовления сквозного «скелета» компилятора к планомерному продвижению, сделано:

- парсер, практически полный, кроме «может быть» типов и null safety операторов — это оставлю на потом, надо еще обдумать.
- первая часть семантического анализа
- для отладки добавлены
    - визуализация AST (см. ниже)
    - тесты сканера, парсера и семантики (через go test)

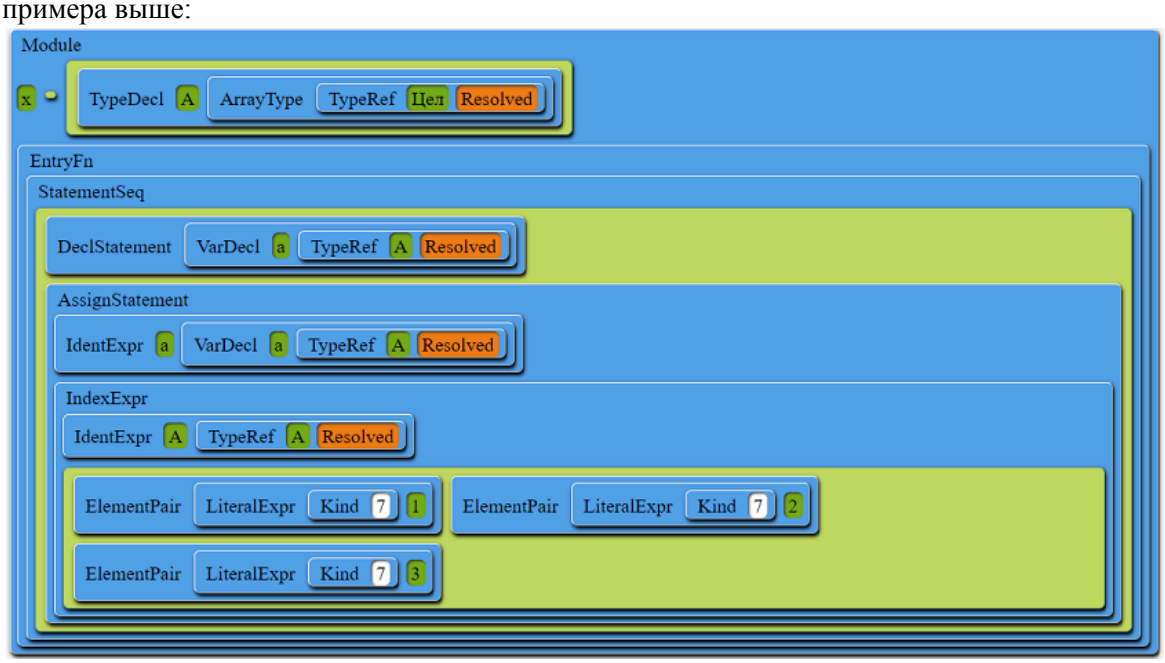
Теперь можно уже писать многое, но существенная часть этого «многого» еще не будет переводится в Си.

Пример текста с конструктором массива (в терминологии Го: composite literal):

```
1  модуль x
2
3  тип A = []Цел
4
5  вход {
6      пусть a: A
7      a := A[1, 2, 3]
8  }
9
```

Компилятор умеет выдать AST в виде S-выражения, который можно вполне удобно рассматривать на сайте s-exploration. Идея визуализации в таком виде принадлежит Дмитрию Соломенникову.

Текст S-выражения строится с помощью рефлексии. Вот такая картинка AST получается для примера выше:



В модуле есть описание типа (TypeDecl) и Вход (EntryFn) — и далее видны под-узлы и атрибуты.

20.12.2022. Тривиль — вопрос про стандартные функции

Стандартные функции, увы, нужны. Например, для дин массивов (slices) в Го есть функции: len, cap (capacity), make, append и copy, работающие с любыми слайсами (независимо от типа элемента). Необходимый минимум — это len, make, append.

Вопрос в том, как их добавить в язык? Обычный способ (Оберон, Го) — встроить обработку в компилятор (compiler magic) и реализовать в run-time.

В этой подходе, есть одна проблема — Лень. Та, что двигатель прогресса. Во-первых, встраивать — это надо делать. Во-вторых, если не встраивать, то будет проще менять их имена, добавлять/удалять. А это очень удобно для прототипирования языка.

Вариант с библиотечной реализацией тоже есть, даже несколько. Очевидный — это параметризованные функции (generic functions).

Описываем: фн длина<A>(a: A) {}, возможно задаем ограничение (constraint), что это должен быть массив и реализуем. Все замечательно, но это совсем не Тривиль, сложность компилятора в разы возрастает. И начинаются всякие фокусы, смотри C#, Java, Kotlin… и прочие.

Еще один способ — через overloading, но для этого набор типов, к которым применяется функция, должен быть ограничен.

Итого, известные варианты:

1. встроить в компилятор (Go)

2. обобщенные типы, возможно, с добавкой перегрузки (много где)

Можно ли придумать что-нибудь достаточно простое для понимания и с тривиальной реализацией?

Перехожу в режим бреда (или мозгового штурма).

Вариант 3. Над-типы
Добавляем в язык тип «Любой дин массив» (Любой вектор?) и разрешаем использовать его в параметрах функции и, может быть, где-то еще (пока не важно). Пишем:

фн длина(а: Любой вектор) @внешняя{имя: «vectorlen»}

И так как любой дин массив, это дескриптор, то реализация на Си тривиальна.

Правда если подумать о том, что длина бывает еще у строк и у статических массивов, становится понятно, что решение не достаточно. Можно, конечно, задать разные имена для длин, применимых к разным типам, но как-то не хочется.

Вариант 4. Над-типы + параметризация + ограниченная перегрузка
Перегрузка (overloading), в общем случае, зло. А если ограничить? Введем понятие «мульти-функции», то есть параметризованной функции с несколькими настройками. Это типа обобщенная функция, но с явно заданным «вручную» набором реализаций:

```
мульти фн длина<А>(а: А): Цел

настройка фн длина<А=Строка> @внешняя{имя: «stringlen»}

настройка фн длина<А=Любой вектор> @внешняя{имя: «vectorlen»}
```

Разрешаем делать настройку только в том же модуле, в котором определена мульти-функция (чтобы было попроще), делаем модуль и импортируем его (явно или неявно). Вуаля, есть длина.

Правда, остается куча вопросов:

- Как, например, написать сигнатуру функции append? (делать что-то вроде Swift associated types?)
- Что делать со статическим массивом, для которого мы ожидаем, что длина есть константа?
- Что делать, если захочется поддержать многомерные статические массивы? Как записывать длину второго измерения?

Кажется, что Лень уже на стороне встроенных функций. Но, может быть, есть еще варианты?

Вариант 5. ???

31.12.2022. Тривиль 2022

Я закончил проход по лексическому, синтаксическому и семантическому анализу. Сделано все, кроме:

- может-быть ссылок и операций с ними
- импорта
- оператора выбора

Попутно, дочистил грамматику, сделал несколько полезных изменений. Например, сделал обязательной инициализацию переменных, и добавил простейший type inference:
пусть Праздник = истина

В компиляторе сейчас 5611 строк в 41 файле и 175 тестов, большая часть на семантику.

Дальше
- доработка генерации — строки, массивы, классы, конверсии.
- импорт, компиляция всей программы
- оператор выбора
- минимальная рефлексия — универсальная печать

Надеюсь доделать компилятор за праздники. Зависит, главным образом, от возможности выделить время свободное от празднования :)

Всех с наступающим! Всем нам великих свершений в следующем году.

08.01.2023. Тривиль

На конец праздничных дней (между выпить/закусить, выездами на природу и прочими развлечениями):
- доделана генерация в Си
- сделана компиляция программы из нескольких модулей (есть пара недоделок)

Вот пример (модули в отдельных файлах, соединены для показа):

```
модуль х

импорт "модули/вывод"

вход {
    вывод.строка("хорошо и ")
    вывод.привет()
}
-------------
модуль вывод

фн строка*(с: Строка) @внеш("имя":"print_string")
фн println*() @внеш

фн привет*() {
    строка("привет!")
    println()
}
```

Печатает: хорошо и привет!

Из изменений сделанных и намеченных:

1. Изменен синтаксис приведения (конверсии) типа. Изменен вроде бы вынуждено, так как я не смог пробиться через Notepad++, но то, что получилось мне нравится больше. Например, приведение целого к вещественному: `ц(:Вещ64)`. Символ `(:` можно читать “как”).

- `ц(:Вещ64)` — ц как Вещ64
- `человек(:Работник)` — человек как Работник

При этом, за “:” как и в описаниях следует тип, а за счет скобок сохраняется синтаксическая однозначность.

Работают преобразования встроенных типов, проверки дин. типа объектов (downcasting), преобразования строки в вектор байтов или символов и обратно. Для последних в runtime добавлены функции кодирования/декодирования UTF-8.

2. Я пришел к мысли, что надо подключать GC вместо ARC. Основное соображение — ускорение реализации. Подключать буду Boehm GC.

Вообще компилятор близок к завершению. Кроме небольших доработок, осталось сделать мб ссылки и оператор выбора (в том числе по типам). В runtime есть что делать, и надо подумать над хранением/доступом к модулям (раскладка кодовой базы).

Еще я думаю, написать систему тестирования на Тривиле. Этого не хватает, так как я не хочу использовать Go механику для исполняемых файлов. Думаю сразу сделать так, чтобы потом использовать с любым компилятором (в том числе с Тривилем на Тривиле). К тому же это будет хорошим тестом самого Тривиля и заставит написать нужные библиотеки.

По сути, у меня остался один важный нерешенный вопрос: как сделать map. Компилятор без map’а на Тривиле я писать не хочу. А как сделать дешево и сердито, пока не придумал.

Статистика:

- Компилятор: 6583 строк на Го (7 пакетов, 44 файла) — думаю, что в итоге будет около 7500 строк.
- Тесты пакетов компилятора: 193 штуки (используется go test)
- Runtime: 509 строки на Си

PS. Если у кого-то есть желание поиграть с компилятором, напишите, подумаем как, хотя еще малость рановато.

23.01.2023. Тривиль — уже вполне весело

Я хорошо продвинулся за пару выходных, сделано (кроме всякого разного):

- многофайловые модули
- экспорт/импорт
- полиморфизм в параметрах

Начал писать библиотеки, вот например (вставляю картинки, чтобы была подсветка текста):

```
модуль строки

тип Байты = []Байт

тип Сборщик* = класс {
    байты = Байты[]
    число-символов := 0
}

фн (сб: Сборщик) добавить строку*(ст: Строка) {
    пусть б = ст(:Байты)
    сб.число-символов := сб.число-символов + длина(ст)
    сб.байты.добавить(б...)
}

фн (сб: Сборщик) добавить символ*(си: Символ) {
    авария("не сделано")
}

фн (сб: Сборщик) символов*(): Цел64 {
    вернуть сб.число-символов
}

фн (сб: Сборщик) байтов*(): Цел64 {
    вернуть длина(сб.байты)
}

фн (сб: Сборщик) строка*(): Строка {
    вернуть сб.байты(:Строка)
}
```

Использование:

```
модуль х

импорт "стд/вывод"
импорт "стд/строки"

вход {
    пусть сб = строки.Сборщик{}
    сб.добавить строку("привет")
    сб.добавить строку(" мир!")
    вывод.строка(сб.строка())
}
```

Запуск в командной строке:

```
D:\0-Projects\trivil-0>tric .
Тривиль-0 компилятор v0.0
Execute: ./_genc/x  Rebuild C code: ./_genc/build.bat
Без ошибок

D:\0-Projects\trivil-0>_genc/x
привет мир!
D:\0-Projects\trivil-0>
```

Оставшиеся шаги до достаточно полного языка:

- форматный вывод (не язык, а библиотека, но без него неудобно)
- оператор выбора
- null safety
- и описание языка.

Может быть еще (до описания языка) сделаю обобщенные классы/функции (самые простые).

После того, как вчитался в реализацию Swift, думаю, что можно сделать просто (без мономорфизации), но не уверен, что вижу все грабли.

03.02.2023. Тривиль — весь язык

Все конструкции языка компилируются и выполняются. Естественно, есть недоделки, но их немного и чистить буду по ходу. Собственно дальше или описание языка или проработка того, как сделать hash map, так как для само-компилятора hash map мне нужен. Или и то и другое одновременно. И далее уже тянем себя за шнурки.

Что можно посмотреть (все тексты в utf-8), так что смотреть в Notepad++ или в любом, кто правильно показывает):

- Грамматика[9]
- Подсветка синтаксиса для Notepad++ (надо изменить расширение на .xml, иначе мне не дают вставить в эту запись)
- Библиотека строки: сборщик формат
- Библиотека вывода: вывод

Пример: привет

```
модуль х

импорт "стд/вывод"

фн привет(имя: мб Строка) {
    если имя = пусто {
        вывод.ф("привет аноним!\n")
    } иначе {
        вывод.ф("привет %v!\n", имя^)
    }
}

вход {
    привет("Вася")
    привет(пусто)
}
```

Исходники лучше смотреть с подсветкой синтаксиса.

Статистика:

- Компилятор: 9001 строк на Go в 8 пакетах и 51 файле.
- Runtime: 767 на C.
- Библиотеки на Тривиле (строки, вывод, юникод): 272 строки.

Написано, как я сейчас посмотрел, ровно за 2 месяца, первый коммит 02.12.2022.

Хочу обсудить несколько конструкций языка — в следующей записи.

[9] См. https://gitflic.ru/project/alekseinedoria/trivil-0/doc/тривиль/история/2023-02-03-grammar.txt

03.02.2023. Тривиль — как лучше?

Меня малость удивляет, что язык получился красивый, несмотря на задачу сделать «тривиально». Видимо, опыт имеет значение — все таки, это 5-й проектируемый мной язык.

Впрочем, хочется еще лучше. Отсюда вопросы. В тексте я ссылаюсь на примеры кода на Тривиле, см. предыдущую запись.

1) Оператор «надо»

Это прямой аналог Swift guard.

Пример: `надо x # 0 иначе авария(«деление на ноль»)`

Насколько понятно слово ‘надо’ и есть ли лучше? ‘требую’? Что-то другое?

2) Оператор выбора

Пример:

```
когда x {
  есть 1: хорошо()
  есть 2: неплохо()
  иначе плохо()
}
```

Собственно, сначала я сделал очевидный синтаксис (в стиле Оберона):

```
выбор x {  |1: хорошо() |2: неплохо() иначе плохо() }
```

Но это приводит к неоднозначности в контексте:

```
выбор x {  |1: x := a | б …
```

Операцию условное-или я хочу писать коротко — ‘|’, поэтому этот синтаксис я отмел, и позаимствовал слово ‘когда’ из Котлина, впрочем только его. Котлин в операторе when совсем не тривиален.

Собственно, то, что получилось, мне нравится. Есть некая однородность операторов: если, когда, пока, надо.

Но может быть вместо есть использовать это?

```
когда x {  это 1: хорошо() это 2: неплохо() иначе плохо() }
```

Замечу, что выбор по типу (Go: `switch x.(type) {}`) я собираюсь добавить, когда понадобится, поэтому учитывать надо еще и: `когда … { это Тип1: … это Тип2: … }`

3) Посмотрите решение для unsafe

См. формат (речь о файле формат.tri из библиотеки “строки”) — ключевое слово «осторожно», в подсветке выделено красным, так что сразу видно.

Обоснование, почему сделано именно так, у меня есть, но много букв я сегодня писать не хочу.

4) Строка формата

Я пытался найти синтаксис для строки формата, смотрел Rust/Python. Не смог найти интуитивно понятный и простой синтаксис без переключения на латиницу, поэтому сделал пока, как в Go. Может подскажет кто или где-то уже есть?

19.02.2023. Тривиль — описание языка (черновик)

Выкладываю черновое описание языка[10]. Написано почти все, кроме двух маленьких глав. Текст не вычитан, так что орфографических ошибок должно быть много, заранее извиняюсь. Чукча был, в основном, писатель…

Тем не менее, хочу выложить, чтобы закончить этот этап работы. Описание заняло изрядное время, примерно треть от времени разработки компилятора.

Буду благодарен за замечания и предложения.

11.03.2023. Тривиль — обобщенные типы — претендую на рекорд Гиннесса

Сделал реализацию обобщенных (generic) модулей (именно модулей, а не типов или функций). Добавил в компилятор меньше 150 строк, думаю, что это рекорд реализации.

Выглядит более коряво, чем обычные generic types, но для Тривиля в самый раз.

Работает так: можно написать обобщенный модуль, например:

```
модуль стек

тип Элементы = []Элемент

тип Стек* = класс {
  элементы = Элементы[]
  верх: Цел64 := 0 // … [0..верх[
}

/* далее методы */
```

Тип элемента стека — «Элемент» в этом модуле не определен, он будет определен при конкретизации.

И далее, пишем «конкретизацию»:

[10] См. https://gitflic.ru/project/alekseinedoria/trivil-0/doc/тривиль/история/2023-02-19-report.pdf

```
модуль стек-цел
конкретизация «стд/контейнеры/стек» («Элемент»: «Цел64»)
```

Эти две строки составляют весь модуль «стек-цел». Его можно импортировать обычным образом:

импорт «модули/стек-цел»

И использовать: `пусть с = стек-цел.Стек; с.положить(1)`

Компилятор возьмет модуль «стд/контейнеры/стек», и сделает из него стек-цел, добавляя: тип Элемент = Цел64 и компилируя обычным образом. Точно так же можно определить стек чего-угодно.

В итоге получаем эффективность на уровне C++ за счет мономорфизации, при этом трудоемкость добавления в компилятор близка к нулю, и все делается во время компиляции.

Стек использую для отладки, потому что проще некуда. Теперь можно написать HashMap — и начать раскрутку компилятора.

Пока писал эту заметку, понял, что можно еще упростить и сделать более универсальным…

14.03.2023. Тривиль — еще обобщенные и разное

1. Немного еще упростил обобщенные модули и добавил главу о них в описание языка, надеюсь, что будет понятнее.
2. Договорился в середине апреля рассказать про Тривиль на онлайн семинаре STEP-2023, напишу, когда день/время будет определено.
3. Думал про синтаксис для foreach/for, просмотрел обзоры и разные языки, например, zig. Кажется, нашел интересное решение. Если у кого-то есть мысли об этом — самое время написать. Основное — нужен обход массивов, ну и позже словарей. Понятно, что можно всегда обойтись циклом «пока», но без foreach немного грустно.

16.03.2023. Тривиль — публичный репозиторий

Репозиторий открыт с сегодняшнего дня: https://gitflic.ru/project/alekseinedoria/trivil-0

Кому интересно, пробуйте. На винде работает, на Линуксе, наверно тоже, но если нет, то скоро будет.

Кто хочет внести вклад, порядок обычный — fork, pull request.

Сразу же предупреждаю — я диктатор. Диктатура абсолютная, хотя и просвещенная — то есть обсуждать я готов, а принимать буду только то, что мне подходит. В то же время, если кто-то резвится в копии, мне все равно — пусть растет 100 цветов.

PS: Конечно же, в README мало что написано, но и писать заранее не охота. Так что будет запрос, будет и ответ.

09.04.2023. Тривиль — апрельская текучка

- добавлен документ, как сделать вклад
- написан текст в главе «совместимость» описания языка
- добавлена операция «x типа T» (is)
- сделана возможность работы на Линкусе (dmitrys99mailru)
- добавлен скрипт установки компилятора для винды

Напоминаю, что 14.04 я буду рассказывать про Тривиль на семинаре STEP-2023.

22.04.2023. Продолжается подготовка к раскрутке компилятора:

- доработка runtime: сделана обработка исключительных ситуаций (windows, linux), трассировка стека (linux)
- доработка библиотек: Словарь (hash map), доработка строк
- разработка подсистемы модульного тестирования
- чистка языка и компилятора

Не все влито в мастер, разработка идет в ветках.

01.05.2023 Тривиль — модульное тестирование

Сделана «тривиальная» система модульного тестирования, встроенная в компилятор. Сделано так:

1. В папке модуля надо сделать подпапку «_тест_», исходные файлы в которой:
    - не компилируются при нормальной компиляции
    - а в режиме «модульного тестирования» компилируются вместе с модулем, как его часть. Тем самым, тесты получают доступ ко всем сущностям модуля (не только к экспортированным)
2. В тестовой части должна быть функция: `фн Тестировать(т: тест-основа.Основа)`, которая и будет вызвана при тестировании. Тест-основа — это класс, определенный в библиотеке.

Вот выдержка из исходников для тестирования библиотеки «строки» (текст можно посмотреть здесь):

```
фн Тестировать*(т: тест-основа.Основа) {
   т-соединить(т) // тест функции соединить.
}
```

Для понимания, функция «соединить» является аналогом Go strings.Join, она собирает из списка строк одну строку, вставляя разделитель:

```
фн соединить*(разделитель: Строка, строки: …Строка): Строка
```

Для тестирования функции задаются данные:

```
пусть тесты-соединить = Все-соединить[
…
Один-соединить{раз: «+», арг: ТСтроки[«ы», «ы»], отв: «ы+ы»},
Один-соединить{раз: «++», арг: ТСтроки[«ыы», «цц»], отв: «ыы++цц»},
…
]
```

И далее, в самой функции «т-соединить» эти данные перебираются, проверяя ответ:

```
пусть в = тесты-соединить[№]
пусть ответ = соединить(в.раз, в.арг…)
если ответ = в.отв {
  т.успех(ф(«соединить %v», №))
} иначе {
  т.ошибка(ф(«соединить %v», №), «‘%v’ # ‘%v'», ответ, в.отв)
}
```

Запуск «`tric +тест стд::строки`» строит исполняемый файл, который выполняет все тесты и выдает итог.

Отметить могу два момента:

- тестирование сделано без функциональных типов. Это не от вредности, а от того, что я ленюсь делать их в Тривиле. Естественно, в след. языках будет нормальная поддержка функциональных типов и замыканий.
- очевидный недостаток — это то, что аварии не перехватываются. То есть нельзя написать тесты на обязательность аварии.

Кроме модульного тестирования, идет работа над минимальным набором библиотек для раскрутки компилятора. Сделаны простейшие файлы «`стд::файлы`», надеюсь скоро добраться до лексера.

14.05.2023. Тривиль — продвижение

Сделано чуть более за месяц с прошлого замера (ветка bootstep6):

2023.04.08

- compiler: 9836 lines in 54 files (Go)
- runtime: 930 lines (C)
- libraries: 5/561 lines (Тривиль)

2023.05.14

- compiler:10369 lines in 56 files (Go)
- runtime: 1312 lines (C),
- libraries: 10/1230 lines (Тривиль)
- трик: 948 lines (Тривиль)

Итого на Тривиле написано более 2000 «боевых» строк, из них почти 1000 в компиляторе (трик).

Думаю, что это были самые трудные строки, надо было дописывать библиотеки, дорабатывать базовый компилятор, править ошибки. Скоро заработает лексер и дальше будет проще. Все же

библиотеки писать труднее, чем компилятор.

Язык при этом почти не менялся, изменился только синтаксис оператора выбора, см. описание языка.

04.06.2023. Тривиль — начало лето

В связи с существенным продвижением в Тривиль компиляторе на Тривиле, стала явной необходимость (или хотя бы полезность) добавления нескольких конструкций языка и они были добавлены:

- выбор по типу
- цикл по вектору (foreach), потенциально расширяемый до арифметического и цикла по итератору
- выходные (in out) параметры

Не описываю здесь, см. в описании языка. Смотреть надо в ветке bootstep8, она еще не слита в мастер.

В библиотеках существенно доработаны строки, полностью переделан форматный вывод, в том числе сделан другой синтаксис форматной строки. Я долго изучал Go и Rust форматы, и сделал, как мне кажется, хороший и логичный синтаксис (и без необходимости переключаться на латиницу). Реализовано еще не все, но мне стало гораздо удобнее. Краткое описание в исходнике. Как всегда, критика и предложения приветствуются.

Компилятор на Тривиле очень существенно продвинулся (написано почти 4.5 тысячи строк):

- полностью выложен AST (АСД)
- почти полностью написан парсер — осталось доделать только разбор литералов, а для этого дописать еще несколько функций в библиотеки. Хотел доделать сегодня, но бензин закончился.

Сравнение с прошлым замером:

2023.05.14

- compiler: 10369 lines in 56 files (Go)
- runtime: 1312 lines (C),
- библиотеки: 10/1230 lines (Тривиль)
- компилятор: 948 lines (Тривиль)

2023.06.04

- compiler: 11128 lines in 57 files (Go) + 800 (новые конструкции и правка ошибок)
- runtime: 1530 lines (C), +200
- библиотеки: 10/1818 lines (Тривиль) +600
- компилятор: 4352 lines (Тривиль) +3500

25.06.2023. Тривиль компилятор на Тривиле компилирует сам себя

Главное написано в заголовке и, наверно, на этом можно было бы остановится…

Тривиль компилятор на Тривиле — это 10.5К строк. Вместе с библиотеками на Тривиле написано 12.5К, и это уже достаточно много для того, чтобы убедится в качестве дизайна языка. Тривиль показал себя удобным и вполне достаточным языкам. В паре мест немного жмет, но это я буду улучшать уже в следующих языках.

Скорость компиляции компилятора и первым компилятором (на Го) и вторым (на Тривиле) мгновенная, после этого примерно 5 секунд работает clang, что дает приемлемую общую скорость компиляции.

Оба компилятора выдают полностью одинаковый Си код, так что проверка нового компилятора тривиальна — сравнение сгенерированных файлов.

Любопытно, что размер исполняемого файла компилятора на Го (3123К) примерно в три раза больше, чем компилятора на Тривиле (1122К). Но к Тривилю еще не подключена сборка мусора, которая добавит существенно к размеру файла, но явно меньше, чем 2М.

Статистика, сравнение с прошлым замером:

2023.06.04

- compiler: 11128 lines in 57 files (Go)
- runtime: 1530 lines (C)
- библиотеки: 10/1818 lines (Тривиль)
- компилятор: 4352 lines (Тривиль)

2023.06.25

- compiler: 11172 lines in 57 files (Go)
- runtime: 1776 lines (C) (+250 строк)
- библиотеки: 11/2321 lines (Тривиль) (+500 строк)
- компилятор: 10455 lines (Тривиль) (+6100 строк)

Да, чтобы не было удивления от скорости разработки (прибавилось 6К строк на три недели) — перевод с Го в Тривиль делался полуавтоматически, с помощью инструмента, который делал черновой перевод, которые надо было дочистить. И я старался сохранить соответствие между исходниками, а если хотелось что-то улучшить, то улучшал в обоих компиляторах.

Что дальше?

- Перенести/сделать модульные тесты — разница в 700 строк между размерами компиляторов как раз и объясняется тем, что на Го есть модульные тесты примерно такого размера.
- А дальше есть техническая работа — добавить несколько файловых операций, подключить сборку мусора, доработать runtime на Линуксе, еще что-то…
- Но главное, можно начинать думать о следующем языке (или следующих языках).

[конец дневника]