Как вырастить дерево до небес

Игорь Жирков

23 марта 2019 г.

Вступление, или зачем читать эту книгу

Эта книга посвящена одному из наиболее примечательных языков программирования — Forth. Несмотря на свою экзотичность, это интересный, выразительный и мощный язык. Для начала несколько интересных особенностей:

- Это язык лишь немного выше уровнем, чем язык ассемблера. Вы можете изменять память напрямую по адресам, а также вызывать код напрямую.
- Практически отсутствует типизация данных. Forth-машина оперирует с т.н. ячейками, которые могут хранить как данные, так и код.
- Синтаксис Forth описывается одним предложением: программа на Forth состоит из слов, разделённых пробелами. Никаких сложных древовидных синтаксических структур нет, таких, как, например, классическая конструкция для условного перехода:

```
if ( condition )
{
  <branch-yes>
}
else
{
  <branch-no>
}
```

- Интерпретатор и компилятор Forth можно легко написать на ассемблере.
- Основной подход к программированию на Forth – создать на основе него язык, который максимально удобен для описания решения задачи (*предметно-ориентированный язык*, *domain-specific language, DSL*).

Метапрограммирование это описание того, как сгенерировать программу. Вместо того, чтобы писать программу напрямую, можно написать другую программу, которая сгенерирует необходимую. Такие алгоритмы, конечно, можно переиспользовать.

Как это выглядит? Например, вы работаете на языке, в котором есть возможность конструировать циклы `while`, а циклов `for` в нём нет. Однако, для того, чтобы перебирать элементы массива по порядку, удобнее использовать `for`. Тогда вы используете возможности метапрограммирования для того, чтобы описать, как сгенерировать кусочек программы, который бы эмулировал поведение `for`, собрав его из доступных вам

конструкций – в частности, `while` там тоже будет использован. Единожды правильно описав эту новую конструкцию, вы больше не сделаете ошибку в ней по невнимательности. Это похоже на написание

Текстовые макросы, которые используются в C или ассемблере – это распространённый инструмент метапрограммирования. Однако они работают с текстом программы напрямую в его плоском представлении, то есть, они не понимают границ разных синтаксических конструкций языка. Важно лишь, чтобы в результате макроподстановок получилось что-то, что парсер языка программирования мог бы разобрать, сконструировав необходимое компилятору или интерпретатору синтаксическое дерево программы (*Abstract Syntax Tree*).

Forth же позволяет писать расширения своего компилятора на самом Forth. Так в язык добавляются новые, сколь угодно сложные конструкции, которые затем можно использовать как если бы они были неотъемлемой частью языка. Ядро языка остаётся при этом очень маленьким – порядка 60 слов, включая арифметические и логические операции.

В большинстве распространённых языков вам необходимо ждать, пока создатели добавят ту или иную возможность; языки очень слабо поддерживают возможности метапрограммирования. Например, в C# был добавлен специальный небольшой внутренний язык LINQ чтобы удобно описывать некоторые операции с данными. Исключением являются, пожалуй, только LISP и его диалекты, Scala, Haskell и Ocaml (с помощью `camlp5`).

Обратная сторона возможностей метапрограммирования тоже есть. Каждый программист будет на основе языка общего назначения делать свои языки и писать программы на них. Тогда если другой человек захочет внести изменения в такую программу, ему придётся сначала выучить этот самописный язык.

Интерпретаторы различных диалектов Forth вы можете встретить:

- В загрузчике FreeBSD;
- В программах управления роботами (ведь очень удобно написать интерпретатор форта на голом железе, а затем описывать логику работы на высокоуровневом языке!);
- Во встраиваемом ПО принтеров и другой бытовой техники;
- В ПО космических аппаратов;
- и во многих других местах.

В этой книге мы обзорно изучим один из диалектов Forth и устройство его абстрактной машины. Затем мы напишем интерпретатор и компилятор Forth на ассемблере, после чего подменим интерпретатор на ассемблере интерпретатором на Forth. Количество кода на ассемблере в интерпретаторе сведётся к минимуму, необходимому для обеспечения последовательной выборки команд Forth, а вся логика будет написана на самом Forth. Наконец, мы научимся метапрограммированию на Forth, что позволит очень легко выражать сложные концепции – например, добавить реализацию управляемой кучи всего в несколько сот строчек кода.

Всё это интересует нас прежде всего через призму дизайна языков программирования, с разговора о котором мы и начнём.

Глава 1

Немного о дизайне языков

В этой главе мы неформально поговорим о том, из чего состоит язык программирования, какие задачи стоят перед его создателем, и как вы в повседневной жизни программиста сталкиваетесь с похожими задачами даже если не проектируете свой язык.

1.1 Что такое язык программирования

Программирование это описание логики работы программы, а для передачи компьютеру информации о том, что он должен делать, нужно иметь способ её описывать. **Язык программирования** предоставляет нам базовые конструкции, наделённые смыслом, а также способ их комбинировать, чтобы выражать более сложные смыслы.

Таким образом, можно сказать, что при создании языка программирования у нас есть несколько фундаментальных творческих и инженерных задач:

- На уровне синтаксиса/формы (конструирования «правильно построенных» программ):
  - Выбор базовых конструкций
  - Выбор способов их комбинирования.
- На уровне семантики/содержания (что именно вы хотите, чтобы происходило в абстрактном вычислителе, когда программист написал ту или иную конструкцию языка):
  - Каким смыслом наделить эти синтаксические конструкции.

Придумав язык программирования, мы можем его внимательно изучить и оценить, насколько он получился хорошим. Как это сделать, по какой шкале оценить качество языка? Это, в том числе, зависит от того, где мы хотим применять этот язык.

1.2 Какие бывают языки программирования?

Для начала заметим: есть языки универсальные (или стремящиеся таковыми быть), а есть специализированные, заточенные под конкретную задачу.

Универсальные языки, такие, как C, Java, Python создаются так, чтобы их можно было использовать в принципе для чего угодно. Есть, однако, два важных взаимосвязанных соображения:

1. Каким бы ни был универсальным язык, некоторые задачи на нём будет решать *неудобно*. Выбор базовых конструкций естественным образом делает выражение решений к одним задачам более простым (потому что, грубо говоря, уже сочетание нескольких базовых конструкций приведёт нас к желаемому результату).

2. Невозможно сделать удобный язык, на котором одинаково легко будет выразить решение любой практически значимой задачи. Серебряной пули нет. Поэтому для каждого языка, который выдаёт себя за универсальный, нужно понимать, для решения каких задач он подходит хорошо, а для каких – хуже.

Специализированные, или предметно-ориентированные языки не стремятся быть универсальными. При их создании явно указывается, для каких задач они будут использоваться. Всё проектирование таких языков строится вокруг максимально удобного решения узкого круга задач, что часто влечёт за собой дополнительные ограничения. Зачастую другие задачи решать на них и вовсе невозможно.

Специализированных языков очень много, например:

- LaTeX для верстки статей и книг.
- SQL для работы с базами данных.
- Регулярные выражения для задач текстовых поиска и замены.
- BNF (формы Бэкуса-Наура) для описания синтаксиса языков программирования.
- XML, HTML для описания древовидных структур данных.
- Verilog и VHDL для описания аппаратного обеспеччения.
- Mathematica, Maple для описания символьных математических вычислений.

Заметьте, на многих из этих языков программы писать невозможно или практически очень трудно! Однако они имеют свои ниши применения и в них они лучше, чем языки общего назначения.

Нельзя оценивать универсальные и специализированные языки одинаково – для первых важно, насколько легко вы можете их использовать для решения широкого спектра задач, для вторых – как удобно на них решать только некоторые, заранее определённые задачи. Гибкости от них не требуется.

1.3 Что делает язык хорошим?

Программы сами по себе сложны, они объёмны и нетривиально структурированы. Язык, который используется для их выражения, должен помогать программисту бороться

с этой сложностью. Для этого он должен быть выразительным. Выразительность помогает приблизить программу к системе понятий, которой оперирует программист, и которая меняется от программы к программе.

Если программист пишет Web-сервер, он хочет думать о своей задаче в терминах “клиент”, “сервер”, “запрос”; он также хотел бы максимально кратко и декларативно описывать процессы обмена запросами и ответами. Он не хотел бы задумываться:

- Пересылки байтов между двумя компьютерами;
- Деталей сетевого протокола, по которому большие запросы или ответы делятся на более мелкие фрагменты для удобной (компьютеру) пересылке по сети;
- Записей и чтений из переменных в памяти.

Это не только и не столько вопрос уровня абстракции. Это, более общо, вопрос несоответствия системы понятий языка программирования и системы понятий, в которой программист придумывает решение задачи.

Сложность языка, которая не ведёт к большей выразительной мощи, является его недостатком.

Мы постулируем следующие желательные свойства для языка программирования:

- **Согласованность** (цельность, непротиворечивость, логическая стройность).

  Несогласованность нарушает логическую красоту и интуитивную понятность языка. Почему в одном случае конструкция допустима, а в другом, аналогичном, нет?

- Можно выделить **минимальное ядро с ортогональной функциональностью** (нет частей, которые предоставляют дублирующие фукнции).

  В противном случае появляется ненужная сложность. Заметим, что не обязательно это должно быть удобное для использования ядро! При должной выразительности, однако, из него легко можно сделать базовый вариант языка, уже гораздо более удобный – см. например [1].

- **Выразительность**, когда легко комбинировать конструкции языка для выражения всё более и более высокоабстрактных концепций.

- Минимизация возможностей сделать **неочевидные ошибки** – например, по невнимательности.

  Эмпирически мы знаем, что программы тяжелее отлаживать, чем писать. Чем меньше возможностей «выстрелить себе в ногу», тем лучше. В современном мире как правило предпочтительнее писать менее производительное, но более безопасное программное обеспечение.

  В этом помогает, например, хорошо продуманная система типов, которая еще на этапе компиляции лишит нас возможности запустить явно некорректную программу.

- В современном мире промышленной разработки необходимо, чтобы программы на языке можно было удобно автоматически анализировать и изменять (рефакторинг).

Несколько мифов о языках программирования:

- Язык может быть или безопасным, или производительным.

  Нет, Rust и Ocaml яркие примеры безопасных языков на которых можно писать высокопроизводительный код.

- Необходима возможность напрямую манипулировать с памятью по адресам, чтобы достичь высокой производительности.

  Нет, адресная арифметика это огромный барьер на пути оптимизаций. При её наличии нужен сложный статический анализ чтобы сделать те оптимизации, которые в более высокоуровневых языках делаются тривиально.

  Например, можно ли заменить этот код на языке C на бесконечный цикл?

```
static int[] flags = {0, 1};

void f(void) {

  while (flags[1]) {
    g();
    }
  ...
  }
```

  Чтобы это сделать, необходимо проанализировать тело цикла, функцию `g(void)` и все функции, которые она вызывает на всех возможных трассах выполнения, затем эти функции рекурсивно и т.д. Если есть хоть одна возможность, что где-то `flags[1]` был изменён напрямую *или по указателю*, эту оптимизацию произвести невозможно. А если есть переменная, которая *может хранить указатель* на `flags`?

- Сложные языки сложны потому, что богаты возможностями, а если бы они были проще, они были бы бесполезны.

  Сложность языков очень часто является следствием плохого дизайна. Иногда новые языки намеренно копируют старые неудачные решения исключительно ради того, чтобы быть похожими на уже существующие инструменты и, как следствие, быть более лёгкими для освоения программистами, уже с ними знакомыми. Например, это можно сказать про язык Go.

  Огромное количество сложности C++ является следствием попытки поддержать большое количество несовместимых стилей программирования, а также конфликтующих и противоречащих друг другу возможностей (см. например [2]).

1.4 Хороший и плохой дизайн

Теперь приведём некоторые примеры решений в существующих языках программирования, которые могут иметь негативный эффект на деятельность программистов, чтобы

проиллюстрировать наши, пока достаточно теоретические рассуждения.

1.4.1 Statement vs Expression

Синтаксически, некоторые конструкции в языках программирования можно сгруппировать в классы. Такие классы называются **синтаксические категории**. Часто в императивных языках встречаются синтаксические категории *statement* (предложение языка, соответствующее выполнению действия) и *expression* (предложение языка, соответствующее вычислению данных). Разделение синтаксических категорий языков на statement и expression невыразительно. Например, в C или Java следующий код некорректен:

```
int x = {
  int z = f( 42 );
  z + 99 + g(2)
  }
```

Это происходит потому, что блок является statement'ом, а statement не возвращает значение. Мы не можем присвоить значение блока в переменную.

В другом языке, например, Scala, похожий код означал бы запуск последовательного вычисления выражений, разделённых точкой с запятой. Результатом этой цепочки вычислений был бы результат последнего вычисленного выражения, а именно `z + 99 + g(2)`. Это число и было бы записано в переменную `x`. Иначе говоря, такой код был бы эквивалентен следующей программе на C:

```
int anon_function() {
  int z = f( 42 );
  return z + 99 + g(2)
}

...

int x = anon_function();
```

Дополнительной выгодой было бы то, что к переменной `z` мы бы не имели доступ как только блок с ней завершился. Такие ограничения помогают делать код более локальным для понимания: чем меньше сущностей из разных частей программы даже теоретически могли бы повлиять на то, как функционирует программа в данном месте, тем лучше.

У разделения statement vs expression исторические корни. Раньше представление о том, как и для чего пишется типичная программа, имело больший акцент на действиях, которые она совершает, а не на данных, которые она вычисляет (и которые управляют тем, какие именно действия будут совершены). Оно до сих пор встречается в языках.

1.4.2 Реализация if

С общеизвестной реализацией `if` в C есть как минимум две проблемы:

- Dangling else (висячий `else`). Рассмотрим следующий код:

```
if (x)
if (y)
{ puts("hello"); }
else { puts("no"); }
```

К какому из `if`'ов относится этот `else`: к первому или второму? Здесь нет определённости. Логика поведения программы, однако, меняется: на наборе данных `x = 1, y = 0` в одном случае будет выведена строка `no`, а в другом не будет выведено вообще ничего.

Решением проблемы могли бы быть:

- Обязательные фигурные скобки после `if`;
- Ключевые слова `then` и `endif`

- Наличие тернарного оператора. Он становится ненужен, как только `if` может возвращать значение, например, в Rust:

```
let x = if 10 > 0 { 11 } else { 12 };
```

Сейчас же он ограниченно дублирует функциональность `if`.

1.4.3 The lexer hack

Процесс парсинга (построения дерева кода на основании текста) бывает сильно усложнён неоднозначностью языковых конструкций. Первая стадия компиляции это, как правило, лексический анализ: разбиение программы на лексемы. Каждая лексема это базовая единица языка: строковые и числовые литералы, ключевые слова, идентификаторы. Часть компилятора, которая занимается этим, называется **лексер**. Полное разделение лексера и парсера, который на основе лексем строит дерево кода, позволяет распараллеливать разбор программы на несколько потоков.

Рассмотрим следующий код.

```
(A)*B
```

В зависимости от контекста, этот код может интерпретироваться как две совершенно разные конструкции:

```
int A, B;
A * B ; /* умножение */


typedef long A;
int* B;
(A) *B ; /* приведение (*B) к long */
```

The lexer hack – это устоявшееся название для решения проблемы с неоднозначным разбором на лексемы, когда чтобы определить, соответствует ли идентификатор типу или переменной, между лексером и парсером устанавливается обратная связь; так лексер может пользоваться результатами семантического анализа.

1.4.4 Синтаксис для generic-функций в Java

Рассмотрим следующий пример. [1]

```
public class HelloWorld{

   public static <A,B> int f() { return 0; }

   public static int g( int x ){ return 0; }

   public static void main( String[] args ){

     g( f<Integer,Integer>( 42 ) );

    }
}
```

Мы не сможем скомпилировать эту программу:

```
$javac HelloWorld.java
HelloWorld.java:7: error: cannot find symbol
        g( f<Integer,Integer>( 42 ) );
            ^
  symbol:   variable f
  location: class HelloWorld
HelloWorld.java:7: error: cannot find symbol
        g( f<Integer,Integer>( 42 ) );
             ^
  symbol:   variable Integer
  location: class HelloWorld
HelloWorld.java:7: error: cannot find symbol
        g( f<Integer,Integer>( 42 ) );
                    ^
  symbol:   variable Integer
  location: class HelloWorld
3 errors
```

Изначально в Java не было generic'ов. Когда они появились (в Java 5), возникла проблема: строчки, подобные написанным выше, невозможно однозначно разобрать.

- Или это вызов функции `f` с двумя типовыми параметрами `Integer` и `Integer`.
- Или это вызов функции `g` с двумя параметрами: `f<Integer` и `Integer>(42)`.

Чтобы разрешить эту синтаксическую неоднозначность, пришлось уйти от единообразного написания типовых параметров всегда справа от символа и в некоторых случаях писать их слева:

```
new ArrayList<String>();

/* НО*/
```

[1] Пример позаимствован из лекции Андрея Бреслава в Computer Science Club осенью 2017

```
Collections.<String>emptyList()
```

Как эта проблема решена в:

- C#: в языке допускается синтаксическая неоднозначность, но она разрешается автоматически в пользу варианта с двумя типовыми параметрами, а не с двумя сравнениями. Если есть необходимость интерпретировать эту строчку другим способом, то можно заключить аргументы функции `g` в скобки.

- Scala в данном случае отказалась от использования угловых скобок вообще, так как они уже используются для сравнений на больше/меньше. Типовые параметры пишутся всегда в квадратных скобках; доступ по индексам в массивах осуществляется с помощью круглых скобок, поэтому неоднозначности не возникает.

  ```
  new List[Str]()
  emptyList[Str]()

  myArray(42)
  ```

  Обратим внимание на то, что такой способ обращения к массивам на самом деле более логичен, чем кажется, потому что массивы ничто иное, как способ кодирования частичных функций из натуральных чисел. В этом смысле массивы являются частным случаем чистых функций, а потому использование похожего синтаксиса уместно.

1.4.5 Отсутствие модульной системы в C

Во многих языках программирования есть поддержка системы модулей. Каждый модуль содержит код, экспортирует какие-то объекты другим модулям и требует для загрузки наличия тех или иных объектов от уже загруженных модулей.

В C и C++ модульная система отсутствует. Вместо этого для похожих целей используется механизм заголовочных файлов. Каждый заголовочный файл может содержать объявления функций, процедур и классов. Он включается «как есть» в файлы с кодом.

- Каждый раз когда файл с кодом включает в себя заголовочный файл `header.h`, мы должны произвести его парсинг, а также парсинг всех заголовочных файлов, которые включает `header.h`. Это огромное количество дублирующейся работы.

- В C++ шаблоны устроены таким образом, что функции с шаблонными параметрами должны находиться в заголовочных файлах. Это дополнительно увеличивает объём работы.

- Механизм включений очень хрупок; он нередко служит источником коллизий между макросами.

1.4.6 Отсутствие анонимных функций и рефакторинг в C

Язык C не очень дружествен к рефакторингам. Например, любые рефакторинги, которые связаны с изменением сигнатуры функции, например, добавление аргумента, крайне сложны, а в общем случае невозможны.

Поясним этот тезис следующими примерами:

- Добавим фиктивный аргумент в функцию `f`, существует указатель на эту функцию.

До

```
int f(void) { }

int (*v)(void) = f;

int main(void) {
    v = &main;
    return 0;
}
```

После

```
int f( int p ) { }

int (*v)(void) = f; // ошибка

int main(void) {
    v = &main;
    return 0;
}
```

Если мы добавим аргумент в функцию `f`, потребуется также изменить тип всех указателей на неё. Но в эти указатели могут быть присвоены также адреса других функций! Значит, типы этих функций тоже нужно изменить. Но тип функции `main`, например, изменить нельзя...

Решением этой проблемы могли бы быть анонимные функции. Например, Intellij IDEA предлагает решение с их использованием для Scala. Попробуем добавить аргумент типа `Float` в функцию `f`:

До

```
def f(x: Int): Int = 42

val p: Int => Int = f
```

После

```
def f(x: Int, y: Float): Int = 42

val p: Int => Int =
    (x: Int) => f(x, 99.0f)
```

Изначально мы имели функцию, принимающую `Int` и возвращающую `Int`, а также ссылку на неё под названием `p`. Добавив в неё аргумент `y:Float` мы не были вынуждены изменить тип `p`! Вместо этого мы поменяли значение, присваиваемое в `p`: теперь это анонимная функция-адаптер, которая всегда передаёт вторым параметром в `f` фиксированное значение. В данном случае это значение `99.0f`.

1.4.7 const в C++

Система константных типов в C++ не всегда работает так, как хотелось бы. Например, представим, что у нас есть переменная типа `std::vector<const char*>*`. Это

означает «указатель на неизменяемый вектор, содержащий неизменяемые строки». Попробуем присвоить в него переменную типа `std::vector<char*>*`, что значит «указатель на изменяемый вектор, содержащий изменяемые строки». Сделать это должно быть возможно, потому что константный тип гарантирует полную защиту от перезаписи. Однако компилятор не позволит произвести данное присваивание, потому что типы этих двух переменных с его точки зрения несовместимы.

Таких проблем не возникает во многих языках, где `const` можно проставить только к конкретной переменной, но не к типу. Как мы видим, `const`-типы плохо взаимодействуют с возможностью параметризовывать типы другими типами.

Многие, очень многие проблемы возникают в языках от того, что их ядро добавляются новые возможности, которые плохо взаимодействуют со старыми. Те же `const`-типы достались C++ от C. В проектировании ПО такие ситуации тоже возникают.

1.5 Зачем изучать дизайн языков?

- Проектирование языка отчасти похоже на проектирование программы.

  При проектировании программы есть два классических подхода:

  1. Сверху вниз, когда сложная задача разбивается на подзадачи, они на еще более мелкие подзадачи и так пока мы не дойдём до уже реализованных концепций (например, до базовых конструкций языка).
  2. Снизу вверх, когда мы комбинируем базовые конструкции в более высокоуровневые, выращивая удобную среду для решения сложной задачи. Это похоже на создание собственного специализированного языка, на котором решение задачи будет выражаться почти тривиально.

  На практике обычно процесс проекирования идёт в двух направлениях сразу.

- Проектирование фреймворков и библиотек похоже на проектирование языков. Здесь вам тоже необходимо найти набор базовых сущностей, которые удобно использовать вместе и с помощью которых легко решать некие типовые задачи.

- Создавать предметно-ориентированные языки для удобного описания логики решений.

  Некоторые фреймворки включают в себя предметно-ориентированные языки, например Spring для Scala. Они помогают удобно выражать бизнес логику, абстрагируясь от процессов, происходящих внутри фреймворка.

  Некоторые библиотеки также предоставляют свои DSL, например, многие библиотеки для написания парсер-комбинаторов, такие, как Parsec в Haskell или стандартная библиотека Scala `scala.util.parsing.combinator`.

- Не копировать плохие решения.

  Мы часто мыслим аналогиями и в ситуации, когда требуется решить проблему, инстинктивно думаем: «а не видели ли мы решение похожей проблемы раньше?».

Критическое мышление по отношению к тем решениям, что мы встречаем на профессиональном пути, позволяет нам более осознанно использовать опыт предшественников и *не копировать неудачные решения*. Решение может «делать то, что нужно», но порой цена этого очень велика – например, нестабильность, или дорогостоящая поддержка решения. По крайней мере, мы должны осознавать, к каким проблемам то или иное решение может привести в будущем, чтобы сделать осознанный выбор между нескольких зол.

Чем же нам интересен Forth? В отличие от большинства распространёных языков, он:

- Имеет минимальное ядро, которое очень легко реализовать. Всё ядро Forth занимает около 900 строчек на ассемблере.
- Предоставляет возможность легко его расширять сколь угодно сложными дополнительными конструкциями.

При этом Forth может быть сложно отлаживать из-за того, что:

- Низкоуровневый язык с прямым доступом к памяти;
- Отсутствия строгой статической типизации;
- Слабой локализации эффектов (но как правило они локализованы в верхних ячейках стека данных).

Глава 2

Основы Forth

В этой главе мы познакомимся с устройством Forth-машины и научимся писать простейшие программы на Forth.

Существует множество диалектов Forth, похожих друг на друга. Поэтому на многих ресурсах по Forth вы будете видеть немного разную информацию. Мы будем использовать Forthress, диалект, специально нами созданный для учебных целей. Его главная цель – минималистичность.

Клонируйте репозиторий `http://github.com/sayon/forthress` и выполните `make` для сборки проекта. Вам потребуется только `make` и `nasm`.

Чтобы запустить Forthress, используйте скрипт `./start`. Он запустит минимальное ядро Forthress и подгрузит дополнительную библиотеку с основными конструкциями языка, например, `if-then-else`.

Forthress работает только на Linux и WSL под Windows 10 на архитектуре Intel 64.

Диагностика. Для того, чтобы опробовать интерпретатор Forthress, вам могут пригодиться следующие слова:

- Слово `.S` используется чтобы посмотреть всё состояние стека не разрушая его;
- слово `?` предоставляет дополнительную информацию о вершине стека, уничтожая её.
- Чтобы посмотреть документацию на слово `dup` введите `' dup ??` сохраняя пробелы вокруг `dup`.

2.1 Основные принципы

Абстрактный вычислитель для языка Forth состоит из процессора, стека данных, стека адресов возвратов и линейно адресуемой памяти (Рис. 2.1). Большинство слов работают со стеком данных, поэтому по-умолчанию мы говорим именно о нём, если не указано обратное. Один из главных параметров Forth-машины это **размер ячейки** – обычно его берут соответствующим размеру машинного слова физического процессора, на котором реализуют Forth. Оба стека состоят из ячеек такого размера.

Программы состоят из слов, разделённых пробелами. Слова выполняются последовательно.

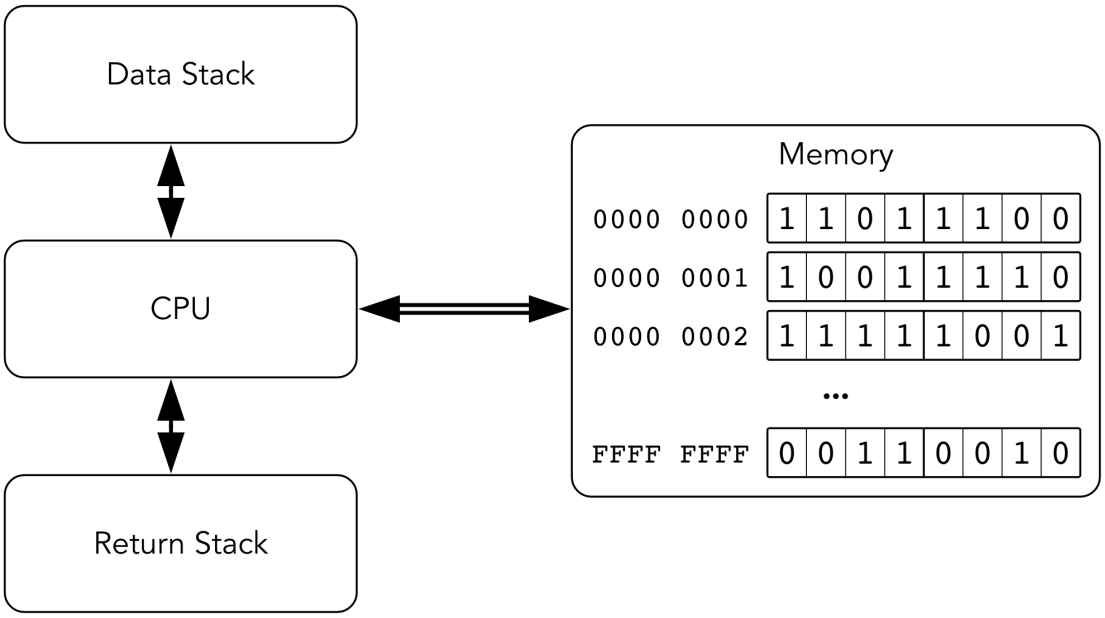


Рис. 2.1: Архитектура Forth-машины.

Пример 2.1.1. Наша первая программа на Forth посчитает сумму 40 и 2 и выведет её на экран.

```
40 2 + .
```

Она состоит из четырёх слов: 40, 2, + и точки. Числа просто кладутся в стек данных. Арифметические операции достают из стека два верхних элемента, совершают над ними действие и кладут результат в стек; аргументы при этом уничтожаются. Точка выводит вершину стека на экран и уничтожает её.

Стековые диаграммы. Для описания слов в Forth приняты **стековые диаграммы**. Например, для слова `swap`, которое меняет местами два верхних элемента в стеке, она выглядит так:

```
swap (a b -- b a)
```

Слева в скобках указано состояние стека до выполнения слова, справа – после. В качестве элементов стека могут выступать буквы, обозначающие любые значения; они помогают проследить за элементами. Вершина стека в обоих случаях справа. Конечно, диаграмма не обязательно полностью описывает то, что делает слово, но по ней уже понятно, как меняется стек после его выполнения.

Еще один пример: слово `dup` дублирует вершину стека:

```
dup (a -- a a)
```

Как правило, слова определены так, что они потребляют аргументы в стеке, заменяя их своим результатом. Так все арифметические операции (`+`, `-`, `*` и `/`) вынимают из сте-

ка два операнда, совершают соответствующее арифметическое действие и записывают результат в стек.

Операнды со стека мы начинаем снимать с конца. Следующее выражение поместит на вершину стека -1, а не 1:

```
1 2 -
```

Определение новых слов. Кроме этих двух встроенных видов слов мы можем определять свои слова как последовательности уже определённых слов. Для этого используются слова “двоеточие” и “точка с запятой”, ограничивающие определение.

Следующая конструкция создаст слово `sq`:

```
: sq dup * ;
```

Теперь каждый раз когда мы напишем в программе слово `sq`, будет выполняться последовательность действий `dup` (продублировать вершину стека) и `*` (перемножить числа на вершине стека). Затем мы вернёмся к тому месту, где слово `sq` было вызвано; в общем, так же, как и в обычных языках при вызове процедур.

Имена слов могут содержать любые непробельные символы, состоять полностью из цифр, пунктуации или псевдографики. Например, вы можете определить слово `42`. В любом случае слова нужно отделять пробелами, даже слова `:` и `;`.

Слова, определённые через другие слова, называются colon-слова (англ. *colon* – двоеточие).

Пример 2.1.2. Следующая программа подсчитает дискриминант уравнения $1x^2 + 2x + 3 = 0$.

```
: sq dup * ;
: discr rot 4 * * swap sq swap - ;
1 2 3 discr
```

Слово `rot` переставит на вершину третье число из стека:

```
rot   (a b c -- b c a)
```

Выполним программу `discr a b c` по шагам для каких-то чисел a b и c. После каждого слова напишем состояние стека после его выполнения.

```
a     ( a )
b     ( a b )
c     ( a b c )
```

Далее выполняется слово `discr`, развернём его:

```
rot   ( b c a )
4     ( b c a 4 )
*     ( b c (a*4) )
*     ( b (c*a*4) )
swap  ( (c*a*4) b )
sq    ( (c*a*4) (b*b) )
```

```
swap ( (b*b)  (c*a*4) )
-    ( (b*b - c*a*4) )
```

Теперь для конкретных значений 1 2 3:

```
1    ( 1 )
2    ( 1 2 )
3    ( 1 2 3 )
rot  ( 2 3 1 )
4    ( 2 3 1 4 )
*    ( 2 3 4 )
*    ( 2 12 )
swap ( 12 2 )
sq   ( 12 4 )
swap ( 4 12 )
-    ( -8 )
```

2.2 Интерпретация и компиляция

Forth работает в двух режимах: интерпретации и компиляции. Когда Forth считывает слова с потока ввода и они выполняются – мы в режиме интерпретации. По умолчанию мы находимся именно в этом режиме.

Во время интерпретации Forth просто сканирует поток ввода и читает все символы подряд, выделяя слова и пропуская пробелы. Каждое выделенное слово он попытается выполнить.

Введя двоеточие и имя слова мы создаём заголовок под новое слово в словаре и переходим в режим компиляции. Каждое следующее введённое слово будет не выполняться, а дописываться к текущему определению.

Исключение составляют immediate-слова. Когда мы встречаем их в режиме компиляции, они не дописываются к определению слова сразу; вместо этого мы приостанавливаем компиляцию и выполняем его!

Мощь метапрограммирования в Forth – сочетание immediate-слов и возможностей узнать состояние Forth-системы: например, текущую позицию в определяемом слове. Это позволяет immediate-словам работать как расширения компилятора.

Мы уже знаем первое immediate-слово: это точка с запятой. Оно не дописывается к незавершённому определению слова, а запускается и выходит из режима компиляции. Другие слова не завершают режим компиляции, возвращаясь к интерпретации, а управляют им. Они могут, например:

- Использовать стек;
- Дописать множество других слов к текущему определению;
- Сохранить в стеке текущую позицию, которую считает какое-то другое, последующее immediate-слово.
- Вызывать другие слова.
- Считать с ввода какую-то текстовую информацию.

Именно это позволяет расширять синтаксис Forth неограниченно.

2.2.1 Реализация комментариев

Рассмотрим, как в Forth реализованы комментарии. Это слово «открывающая скобка». Оно считывает символы с потока ввода пока не встретится закрывающая скобка. Это immediate-слово, что позволяет писать комментарии внутри определений слов.

Для определения этого слова нам понадобится знать следующие слова:

- Конструкция `repeat ... until` позволяет повторить блок команд. В конце каждой итерации мы проверяем: какое число на вершине стека? Если это 0, то убираем его и запускаем цикл снова; если не ноль, то убираем его и выходим из цикла.
- `readc` считывает с потока ввода один символ. Его код кладётся на вершину стека.
- `=` сравнивает два числа на вершине стека, уничтожая их. На вершину кладётся 0, если числа не равны, иначе 1.

Итак, вот определение слова для написания комментариев. Что здесь происходит?

```
: ( repeat readc 41 = until ; IMMEDIATE
```

- Определяется слово `(`.
- Повторяем последовательность `readc 41 = not` пока в конце итерации на стеке не окажется ненулевое число.
  - `readc` считывает с потока ввода один символ. Его код кладётся на вершину стека.
  - Затем мы кладём на вершину стека число 41 (ASCII-код закрывающей скобки).
  - Сейчас в стеке два числа числа: код введённого символа и 41. Выполняем `=`, после чего на стеке остаётся или 0 (коды символов не равны), или 1.
  - Если на стеке 1, мы только что считали закрывающую скобку. Комментарий кончился, цикл остановится.
- Точка с запятой завершает определение слова. `IMMEDIATE` помечает последнее из определённых слов как immediate-слово.

Другие примеры простых базовых слов можно найти в начале файла `stdlib.frt`.

2.3 Стек адресов возврата

Как мы видим, первый стек используется для вычислений. А зачем нужен второй стек?

- Для адресов возвратов из colon-слов.

  Каждый раз когда colon-слово запускается, адрес, с которого произошёл его вызов, нужно сохранить. В противном случае мы не будем знать, куда возвращаться после его завершения. Этот адрес возврата хранится в стеке адресов возвратов.

В ассемблере для Intel 64 для этого используется обычный стек, см. описание команд `call` и `ret`. Выделение хранилища адресов возврата в отдельный стек позволяет организовать более органичный обмен данными между словами через один стек данных.

- Для облегчения написания логики программы, когда использовать стек данных неудобно. Например, когда необходимо сохранить три элемента с вершины стека, совершить какие-то действия с четвёртым, а затем вернуть три элемента на место.

- Для хранения информации о состоянии, которую нежелательно смешивать с данными. Например, в циклах текущий номер итерации и максимальное их количество лежат в стеке адресов возвратов.

Для работы со стеком возвратов используются три слова:

- `>r` Кладёт элемент с вершины обычного стека в стек возвратов.

- `r>` Вытаскивает элемент и кладёт его на вершину обычного стека данных;

- `r@` Как `r>`, но не удаляет элемент из стека адресов возвратов.

Повреждение стека возвратов. Стек адресов возврата критически важен для правильного функционирования Forth-машины. Если в момент выхода из слова на стеке лежит неправильный адрес возврата, то программа аварийно завершится.

Например, выполните в режиме интерпретации слова `40 >r`. Интерпретатор Forth написан на самом Forth, и когда во время его работы на стеке адресов возврата окажется лишнее число 40, это нарушит его работу.

Еще один способ аварийно завершить работу Forth:

```
: myword 42 >r ;
myword
```

В момент выхода из `myword` на вершине стека адресов возвратов будет не адрес возврата из `myword`, а число 42. Попытка перейти по нему чревата проблемами.

А вот такая программа не нарушит работу Forth:

```
: myword 42 >r r> ;
myword
```

Число 42 будет положено в стек возвратов, а затем перемещено обратно в стек данных. В момент выхода из слова `myword` на вершине стека возвратов будет лежать адрес возврата оттуда.

Compile-only слова. Существуют слова, которые пользуются вторым стеком в служебных целях. Прежде всего это слова, которые реализуют циклы. Например, в цикле `do` ... `loop` на каждой итерации цикла в стеке возвратов лежат максимальное и текущее значения счётчика цикла. Если вы их измените или выбросите из стека, последствия могут быть разрушительными.

2.4 Вывод

Вот несколько слов чтобы выводить информацию в поток вывода.

- Чтобы вывести вершину стека, используйте точку.
- Чтобы вывести строку “string” напишите `."string"`. Обратите внимание на пробел после первой кавычки, он обязателен! Точка и кавычка подряд образуют forth-слово.
- Чтобы вывести символ по его коду, используйте `emit`.
- Чтобы перевести строку, используйте `cr`.

Например:

```
: printi ." the number is " . cr ;
42 printi
```

Вывод:

```
 the number is 42
```

2.5 Конструкции ветвления

В ядре Forthress только две конструкции для организации переходов: `branch` и `0branch`. Это низкоуровневые слова, которые мы изучим позднее; как правило, не стоит их использовать напрямую. С помощью метапрограммирования и immediate-слов на их базе реализованы более высокоуровневые примитивы:

- Ветвление.

  Слово `if` выполняет тело если вершина стека ненулевая; иначе выполняется необязательная ветка `else`.

  ```
  ( outputs either non-zero or zero )
  : test if ." non-zero" else ." zero" then ;

  0 test    ( outputs zero )

  ( only outputs non-zero )
  : test2 if ." non-zero" then ;

  0 test    ( outputs nothing )
  ```

- Конструкция `repeat ... until` позволяет повторить блок команд. В конце каждой итерации мы проверяем: какое число на вершине стека? Если это 0, то убираем его и запускаем цикл снова; если не ноль, то убираем его и выходим из цикла.

  В следующем примере слово `test` делит число на два нацело пока мы не дойдём до единицы.

```
: test repeat dup . 2 cr / dup 1 = until drop ; 10 test
```

- `for ... endfor` удобен чтобы повторить блок кода фиксированное количество раз. Типичное использование может выглядеть так:

  ```
  : test 10 0 for r@ . ." hello "  cr endfor ;
  ```

  В этом примере 10 это лимит, а 0 – индекс. Блок кода между `for` и `endfor` повторяется пока индекс не пробежит от начального значения до лимита. При этом на каждой итерации текущее значение индекса лежит на вершине стека возврата (не данных). В примере мы копируем его из стека возвратов с помощью `r@` и выводим на экран с помощью точки. Запуск слова `test` выдаст следующие строчки:

  ```
  0 hello
  1 hello
  2 hello
  3 hello
  4 hello
  5 hello
  6 hello
  7 hello
  8 hello
  9 hello
  ```

  Цикл может не повториться ни разу если лимит меньше или равен индексу.

Обратите внимание, что все эти конструкции compile-only, то есть использовать их можно только внутри определений слов, но не в режиме интерпретации.

2.6 Примеры

Пример 2.6.1. `rot` и обратный ему `-rot`:

```
( a b c )
: rot >r swap r> swap ;
: -rot swap >r swap r> ;
```

После выполнения последовательности команд `rot -rot` содержимое стека будет неизменным (если там было хотя бы три элемента).

Пример 2.6.2. Проверка принадлежности числа интервалу.

`in-range` принимает число x, левую и правую границы интервала $[x,y]$. Результат выполнения – 0 или 1.

```
: over >r dup r> swap ;

( a x y - 0/1 )
```

```
: in-range rot swap over >= -rot <= land ;
```

Пример 2.6.3. Подсчёт n-ого числа Фибоначчи.

```
: over >r dup r> swap ;

( n -- )
: fib-n
  dup 0 < if ." Negative argument " else
    dup 2 < if 1 else
      >r
      1 1
      r> 1              ( data: 1 1 n 1 , ret:  )
      do
          swap over +
      loop
      swap drop
    then
then ;
```

2.7 Задание

Используем готовую реализацию Forthress чтобы немного привыкнуть к новому языку программирования.

1. Изучите файл `README.md` в репозитории Forthress. Он хранит в себе все слова, которые определены в ядре Forthress.

2. Изучите программы из этой главы.

3. Напишите программу, проверяющую число на четность.

   Вы можете подавать программу на вход скрипту `start` и она будет выполняться при запуске. Слив его содержимое с потоком ввода с помощью `cat` вы можете получить эффект аналогичный предзагрузке этого файла.

4. Напишите программу, проверяющую число на простоту.

5. Напишите программу так, чтобы она выделяла с помощью `allot` ячейку в памяти, записывала туда результат и возвращала её адрес. Вам могут потребоваться слова `@`, `!`, `@c`, `!c`. Слово `allot` принимает количество байт для выделения в глобальной области данных.

   Обратите внимание, вы не обязаны делать все эти действия в одном и том же слове. Делайте максимально короткие слова и комбинируйте их.

6. Создать строку в куче в Forthress можно так: `m"string"`. Слово `prints` печатает строку по указателю. Память в куче должна освобождаться с помощью `heap-free`.

С помощью `heap-show` можно получить диагностическую информацию о куче.

С помощью `?` можно получить диагностическую информацию о любом числе или адресе. Например, попробуйте ввести:

```
42 ?
' dup ?
m"  hello, world" ?
```

`' dup` это слово-амперсанд. Оно читает Forth-слово с потока ввода и помещает в стек адрес его реализации.

7. Прочитайте про слова `c@`, `c!`. Напишите слово, которое принимает указатели на две строки и возвращает их конкатенацию. Память под строку-результат вы можете выделить с помощью слова `heap-alloc`, которое принимает количество байт для выделения и

   Протестируйте полученное слово. Нет ли утечки памяти?

Вторая часть задания организована по вариантам. Ваш вариант подсчитывается с помощью следующей функции, применённой к вашей фамилии:

```
( str - num )
: string-hash
  0 >r ( init accumulator )
  repeat
    dup c@ ( stacks: str char, acc )
    dup if ( not end of the line )
        r> 13 * + 65537 % ( iteration of hash computations )
        >r 1 +  0
    else ( end of line )
         drop drop r> 1
    then
  until
;
```

Это слово уже определено в файле `hash.frt`. Номер варианта подсчитывается так: от вашей фамилии берётся хэш, затем вы берёте значение хэша по модулю 3.

Варианты:

0 Напишите слово, которое для положительного числа построит последовательность Коллатца.

  Последовательность Коллатца строится так:

  - Если текущий член последовательности делится на два, то делим его пополам.
  - Иначе умножаем его на 3 и прибавляем 1.

  Последовательность Коллатца всегда приходит к единице, но никто не знает, почему.

1 Напишите слово, которое проведёт проверку числа на примарность.

Примарным называется число, которое представляется в виде произведения простых без повторов. Например, число 4 не примарное, т.к. $4 = 2^2$; 20 – не примарное число, т.к. $20 = 2^2 \cdot 5^1$; 15 – примарное число, т.к. $15 = 3^1 \cdot 5^1$.

2 Напишите слово, которое найдёт произведение всех простых делителей числа (его радикал).

Например, для числа 20 это 10, т.к. $20 = 2^2 \cdot 5^1$, его простые делители 2 и 5.

Глава 3

Принципы работы Forthress

В этой главе мы познакомимся с тем, как устроена Forth-машина в деталях. Мы узнаем, что такое шитый код (*threaded code*), поймём, как он выполняется, а также узнаем, как хранятся слова в Forthress.

Ядро Forth составляют следующие компоненты:

- Словарь, хранящий информацию об уже определённых словах.
- Внутренний интерпретатор (*inner interpreter*) – небольшая ассемблерная программа, которая будет осуществлять выборку слов из памяти и их запуск.

Поверх этого ядра на самом Forthress реализован интерпретатор, который считывает слова из потока ввода, находит их в словаре и выполняет (или дописывает к определению текущего слова во время его определения). Этот интерпретатор называется внешним (*outer interpreter*).

Мы описываем устройство одной из возможных Forth-машин; есть много архитектурных решений, которые можно было бы принять по-другому. Иногда мы будем обращать внимание на такие возможности.

3.1 Шитый код

Шитый код (*threaded code*) – способ организации кода, при котором программа состоит только из вызовов, прямых или косвенных. Поскольку Forth-программа состоит из последовательного вызова слов, каждое из которых изменяет глобальное состояние машины, шитый код уместно использовать в реализации Forth-машины. Можно сказать, что он является байткодом для языковой виртуальной машины Forth.

3.1.1 Три вида шитого кода

Мы продемонстрируем разновидности шитого кода с помощью следующей Forth-программы:

```
dup * .
```

Эту программу можно скомпилировать в ассемблер множеством разных способов; мы изучим три из них.

Subroutine Threaded Code Программа состоит из многочисленных ассемблерных инструкций `call`, каждая из которых вызывает реализацию слова на ассемблере (процедуру, *subroutine*). Выполнение программы происходит так же, как и выполнение любой программы на языке ассемблера, так как программа сама является правильной последовательностью ассемблерных инструкций.

```
_dup: push rsp
      ret
_dot: call print_int
      ret
_mul: ...


program:
      call _dup
      call _mul
      call _dot
```

Иными словами, это подход с прямым вызовом обработчиков инструкций.

Direct Threaded Code Программа состоит из адресов обработчиков инструкций. В этом подходе вызов непосредственно ассемблерного кода происходит с одним уровнем косвенности.

Этот код уже нельзя выполнить напрямую: выполнение происходит с помощью небольшого цикла `next`, который считывает следующую команду – адрес обработчика инструкции – и вызывает код по этому адресу. Выполнение всех обработчиков заканчивается прыжком на внутренний интерпретатор.

Необходимо зарезервировать виртуальный регистр `pc` – счётчик команд, указывающий на следующую инструкцию для выборки и выполнения.

В примере ниже необходимо начать выполнение программы с метки `init`.

```
%define pc r15

_dup: push rsp
      ret
_dot: call print_int
      ret
_mul: ...


program:
      dq _dup, _mul, _dot

init:
      mov pc, program
next:
      call [pc]
      add pc, 8
      jmp next
```

Indirect Threaded Code Программа состоит из адресов адресов обработчиков инструкций, то есть в этом подходе вызов непосредственно ассемблерного кода происходит с двумя уровнями косвенности.

Forthress использует Indirect Threaded Code. Он резервирует два специальных виртуальных регистра:

- `w` (*word*) указывает на адрес обработчика текущего слова. По адресу `w` хранится адрес ассемблерного кода обработчика текущей инструкции.
- `pc` (*program counter*) это счётчик команд. По адресу `pc` хранится адрес адреса ассемблерного кода обработчика следующей инструкции.

Эти виртуальные регистры можно сопоставить настоящим регистрам или ячейкам памяти. В Forthress используются регистры `r15` для `pc` и `r14` для `w`.

Этот код также нельзя выполнить напрямую: выполнение происходит с помощью небольшого цикла `next`; это и есть внутренний интерпретатор, сердце Forth-машины. Выполнение всех обработчиков заканчивается прыжком на внутренний интерпретатор.

```
%define pc r15
%define w  r14

i_dup: push rsp
       jmp next

i_dot: call print_int
       jmp next
i_mul: ...

xt_dup: dq i_dup
xt_dot: dq i_dot
xt_mul: dq i_mul

program:
      dq xt_dup, xt_mul, xt_dot

init:
      mov pc, program
next: mov w, [pc]
      add pc, 8
      jmp [w]
```

Ячейки, которые хранят непосредственно адрес обработчиков инструкций, называются *execution token*'ы. В этом примере ими являются адреса ячеек `xt_dup`, `xt_dot`, `xt_mul`. В дальнейшем мы будем сокращать термин execution token до XT.

3.1.2 Indirect Threaded Code и colon-слова

Мы показали пример Indirect Threaded Code, в котором были только нативные слова. Однако этого нам недостаточно: мы хотим составлять новые слова из уже существующих. Из нашего примера ITC-кода неочевидно, как их реализовать.

В Forthress colon-слова устроены так:

- Все colon-слова имеют одинаковую реализацию, называемую `docol`.
- Непосредственно после XT colon-слова хранится последовательность XT слов, которые составляют его определение.
- Выполнение любого colon-слова завершается с помощью нативного слова `exit`.

Роль реализации `docol` заключается в том, чтобы сохранить старое значение `pc` в стеке возвратов и настроить его на новое значение: ячейку, непосредственно следующую за XT слова. Там, как мы сказали, последовательно хранятся токены слов, составляющих определение colon-слова.

Сейчас мы вручную сформируем программу со словами `*`, `dup`, `sq` и посчитаем квадрат числа 3. Мы закодируем следующую программу на Forthress:

```
: sq dup mul ;
sq . bye
```

Листинг 3.1: `sq_itc.asm`

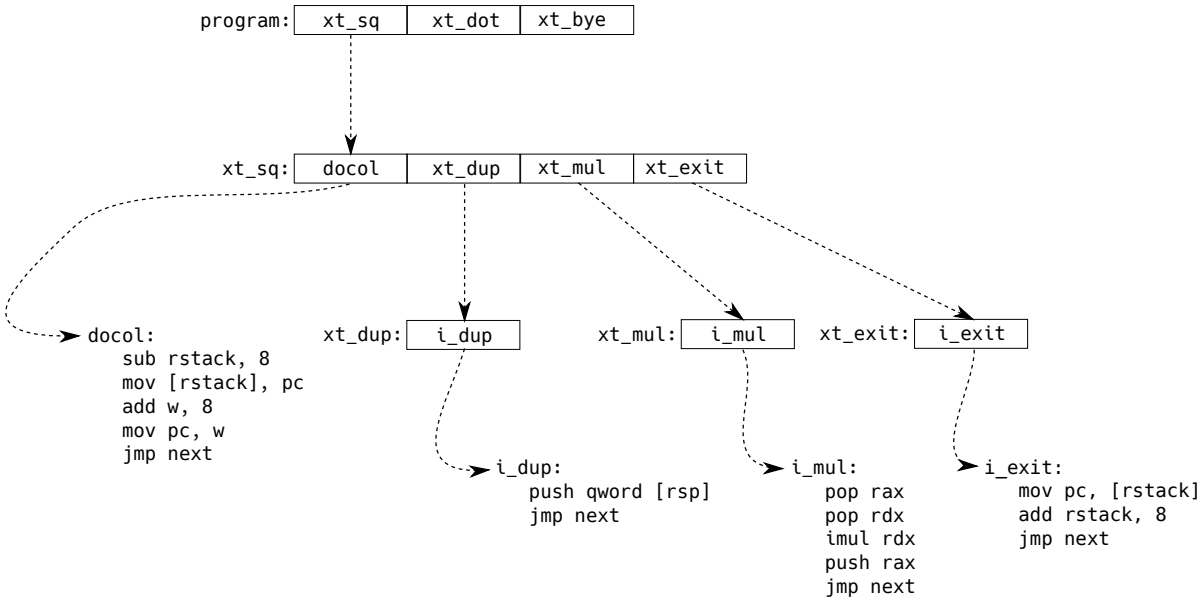
```
%include "lib.inc"
global _start
%define pc     r15
%define w      r14
%define rstack r13

section .bss
resq 1023
rstack_start: resq 1

section .data
xt_mul:  dq i_mul
xt_exit: dq i_exit
xt_bye:  dq i_bye
xt_dup:  dq i_dup
xt_dot:  dq i_dot

xt_sq:   dq docol
         dq xt_dup, xt_mul, xt_exit

section .text

_start:  mov rstack, rstack_start
         mov pc, program
         push 3
         jmp next

i_bye:   mov rax, 60
         xor rdi, rdi
         syscall

i_exit:  mov pc, [rstack]
         add rstack, 8
         jmp next
```

```
i_mul:   pop rax
         pop rdx
         imul rdx
         push rax
         jmp next

i_dup:   push qword [rsp]
         jmp next

i_dot:   pop rdi
         call print_int
         jmp next

docol:   sub rstack, 8
         mov [rstack], pc
         add w, 8
         mov pc, w
         jmp next

next:    mov w, [pc]
         add pc, 8
         jmp [w]

program: dq xt_sq, xt_dot, xt_bye
```

Перед запуском программы в стек кладётся число 3. Внешняя процедура `print_int` используется для вывода целого числа в поток вывода.

Старт виртуальной машины происходит по метке `_start`. Мы начинаем с установки корректного состояния машины: необходимо инициализировать стек возвратов, поместить в регистр `pc` адрес первой инструкции (`program`). В стек данных мы помещаем число 3. Затем происходит прыжок на внутренний интерпретатор `next`.

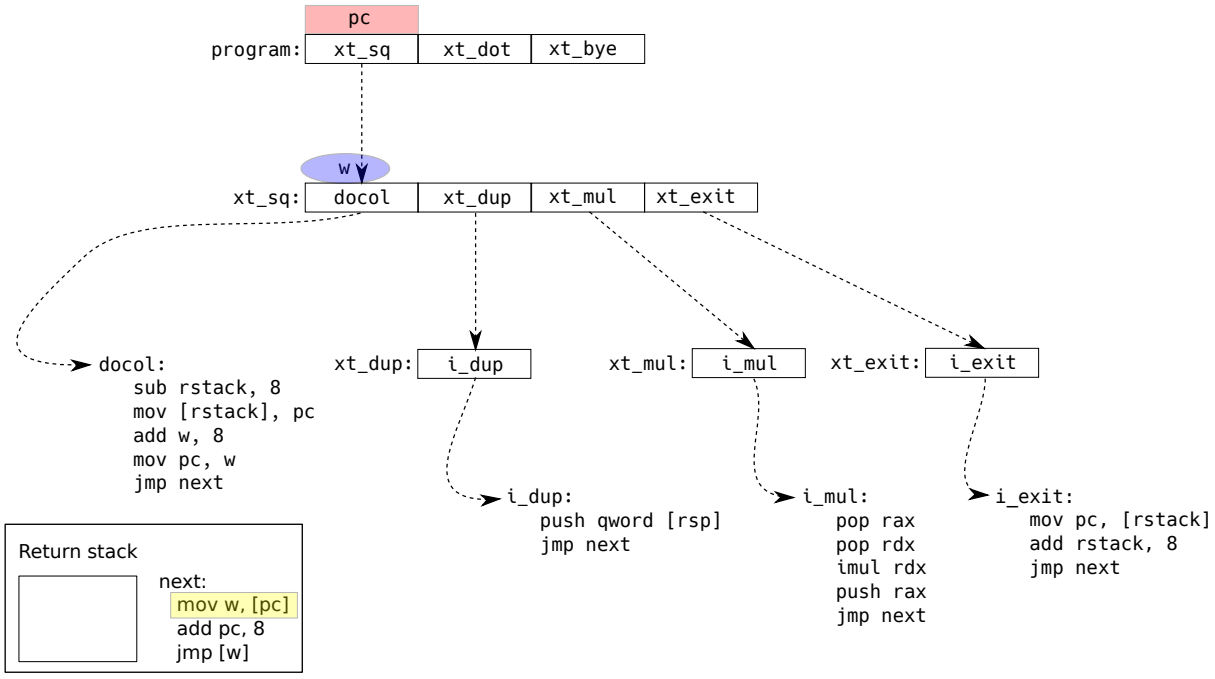
Чтобы прочувствовать то, как работает интерпретатор, изобразим фрагмент программы выше в виде графа и проведём трассировку выполнения программы:


program:
xt_sq
xt_dot
xt_bye
xt_sq:
docol
xt_dup
xt_mul
xt_exit
docol:
sub rstack, 8
mov [rstack], pc
add w, 8
mov pc, w
jmp next
xt_dup:
i_dup
xt_mul:
i_mul
xt_exit:
i_exit
i_dup:
push qword [rsp]
jmp next
i_mul:
pop rax
pop rdx
imul rdx
push rax
jmp next
i_exit:
mov pc, [rstack]
add rstack, 8
jmp next

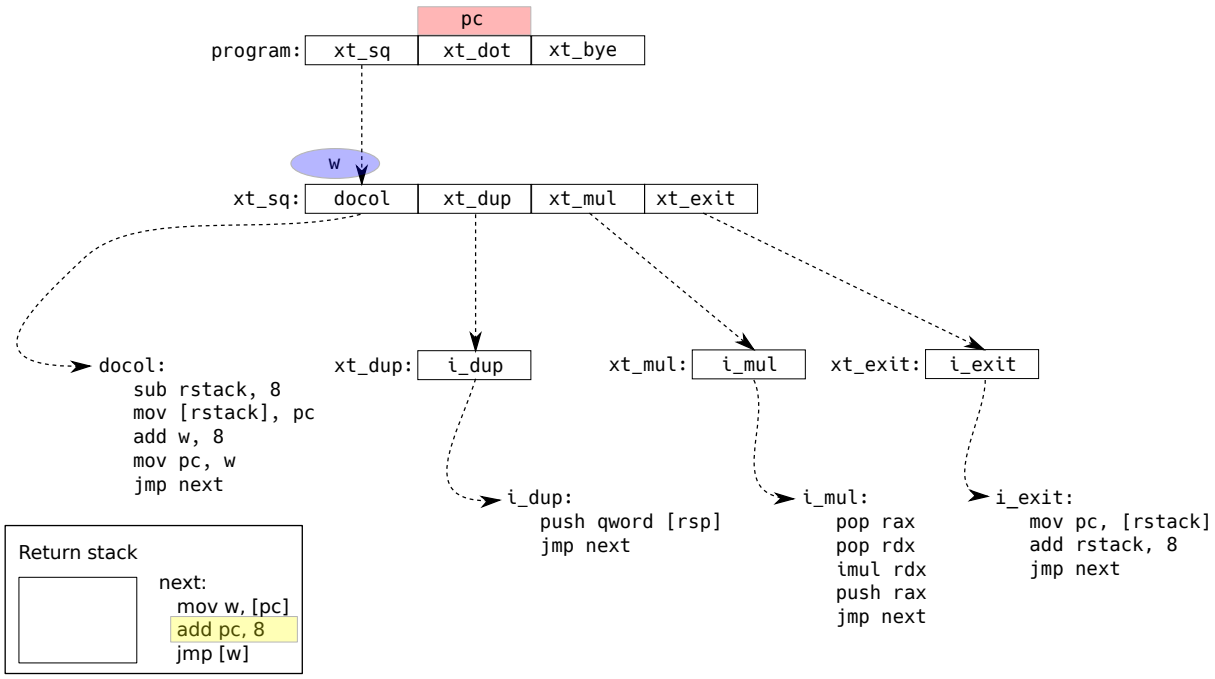
Шаг 1. В начальном состоянии, установленном в `_start`, регистр `pc` хранит `program` – адрес следующей инструкции. Мы прыгаем на внутренний интерпретатор `next`.

`pc` указывает на XT слова `sq`.

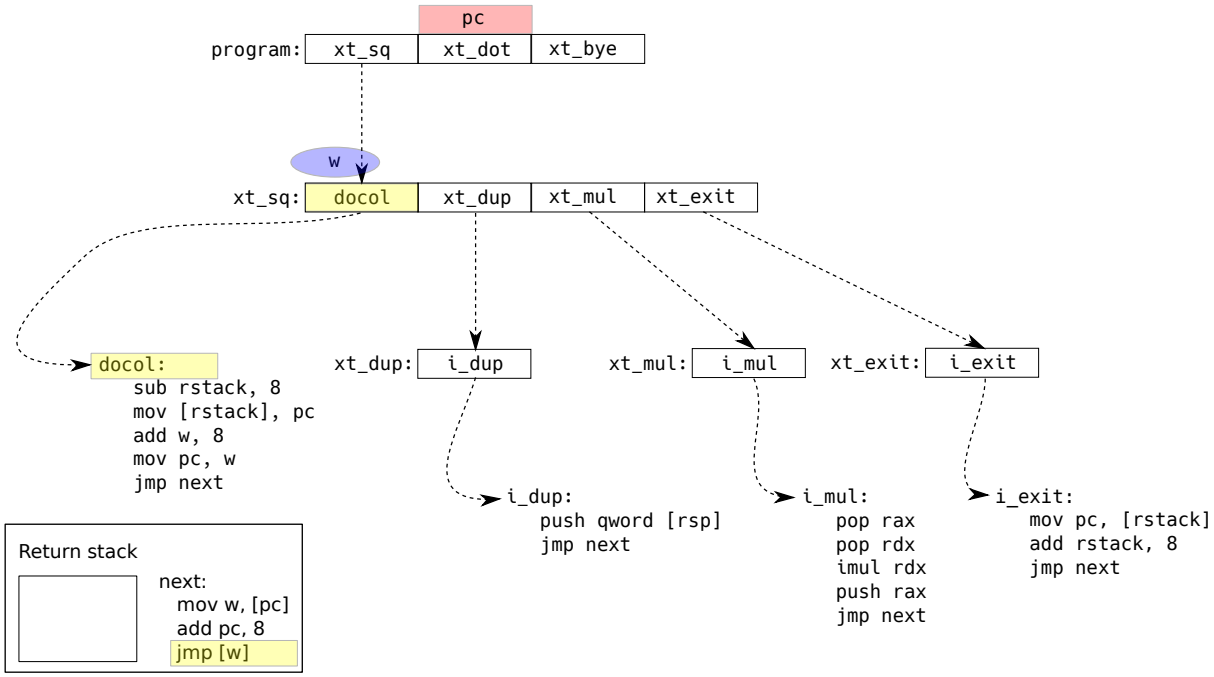
Здесь начинает выполняться colon-слово `sq`. Интерпретатор `next` начинает с установки `w`=<XT слова `sq`>. Это важно именно для colon-слов.



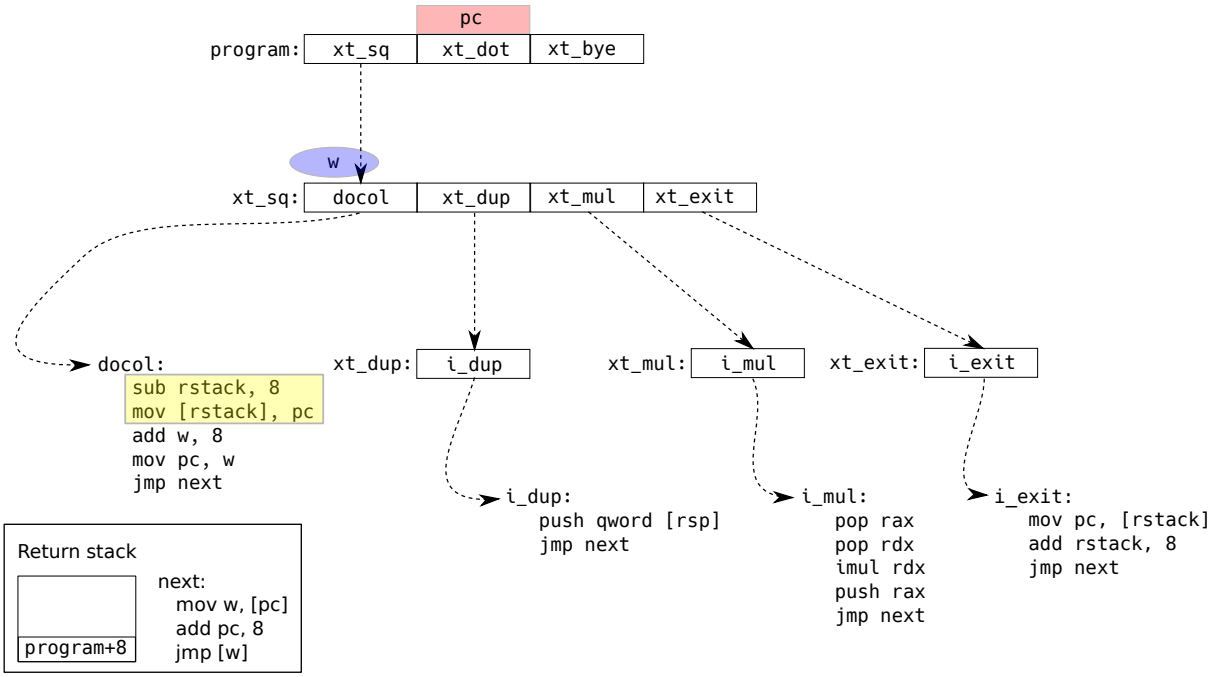
Шаг 2. Затем `pc` увеличивается чтобы указывать на следующую инструкцию:



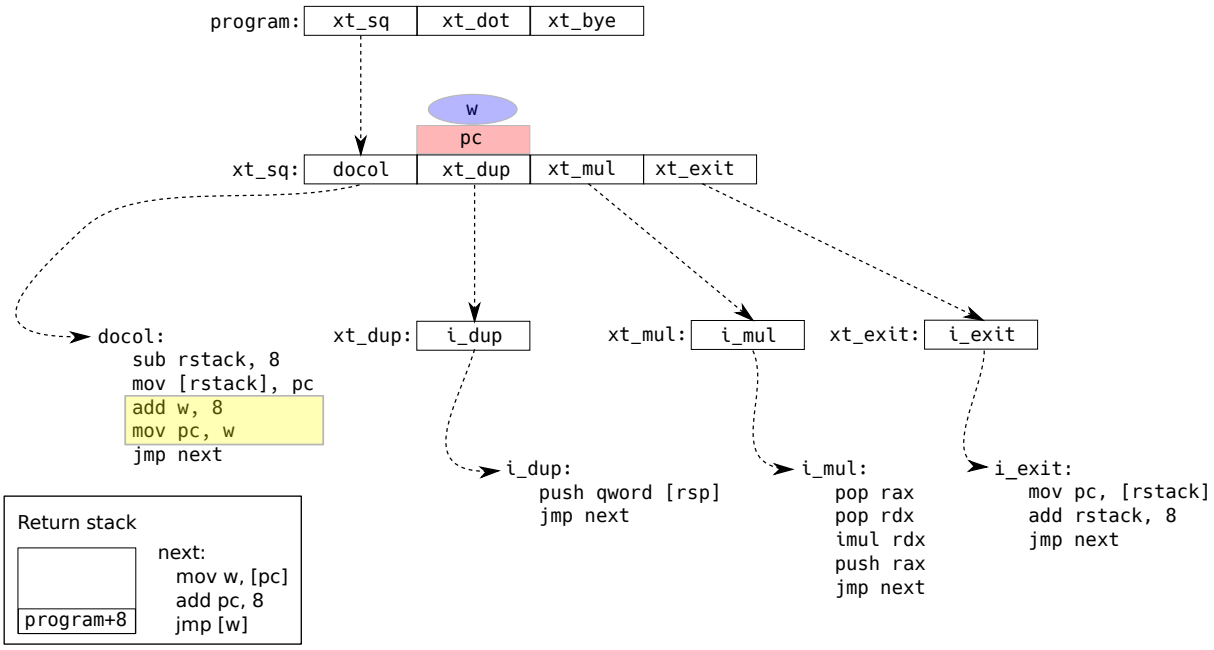
Шаг 3. Благодаря тому, что мы использовали `w`, мы можем перейти на `[w]` и запустить `docol` – реализацию всех colon-слов.



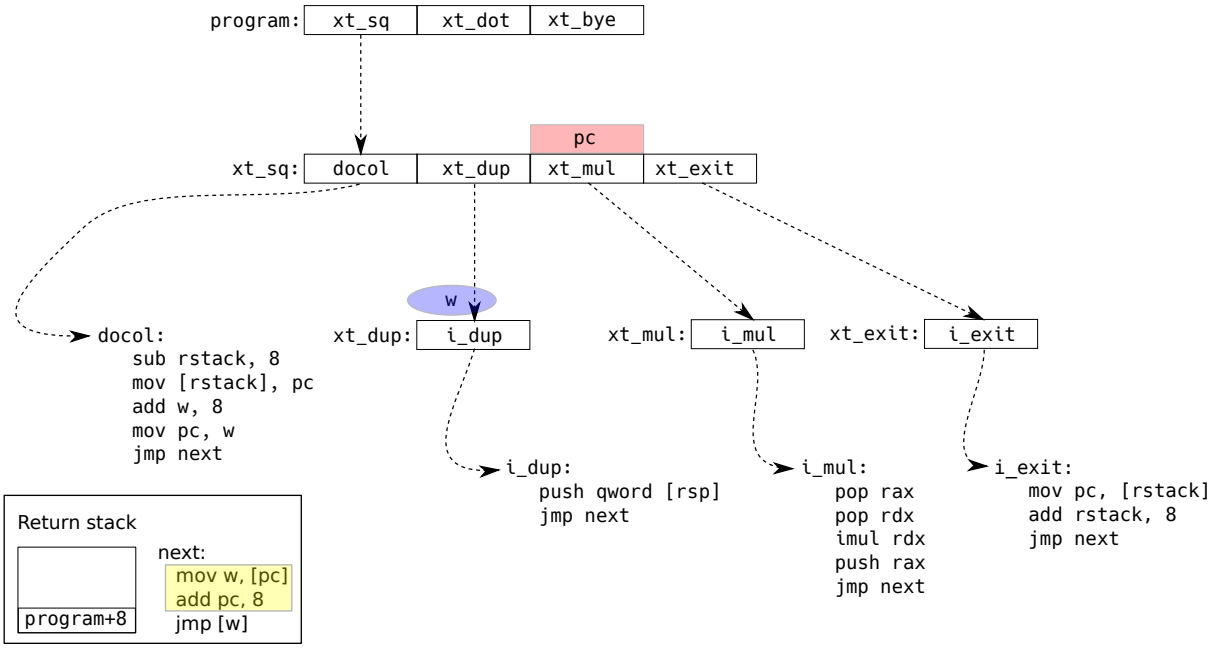
Шаг 4. Процедура `docol` делает то, что делает инструкция `call` на ассемблере: сохраняет счётчик команд в стеке (в случае Forth это стек возвратов).



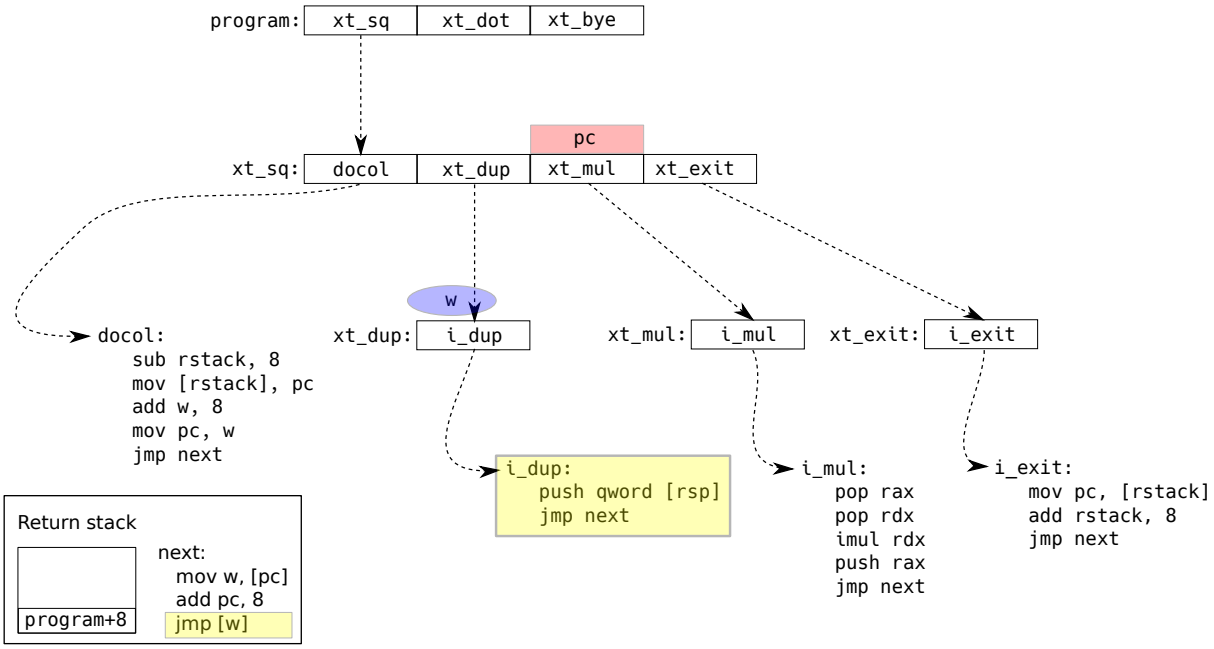
Шаг 5. Сохранив старое значение `pc`, мы устанавливаем его на первую инструкцию текущего слова `sq`. Мы только что показали, как начинает выполняться colon-слово `sq`.



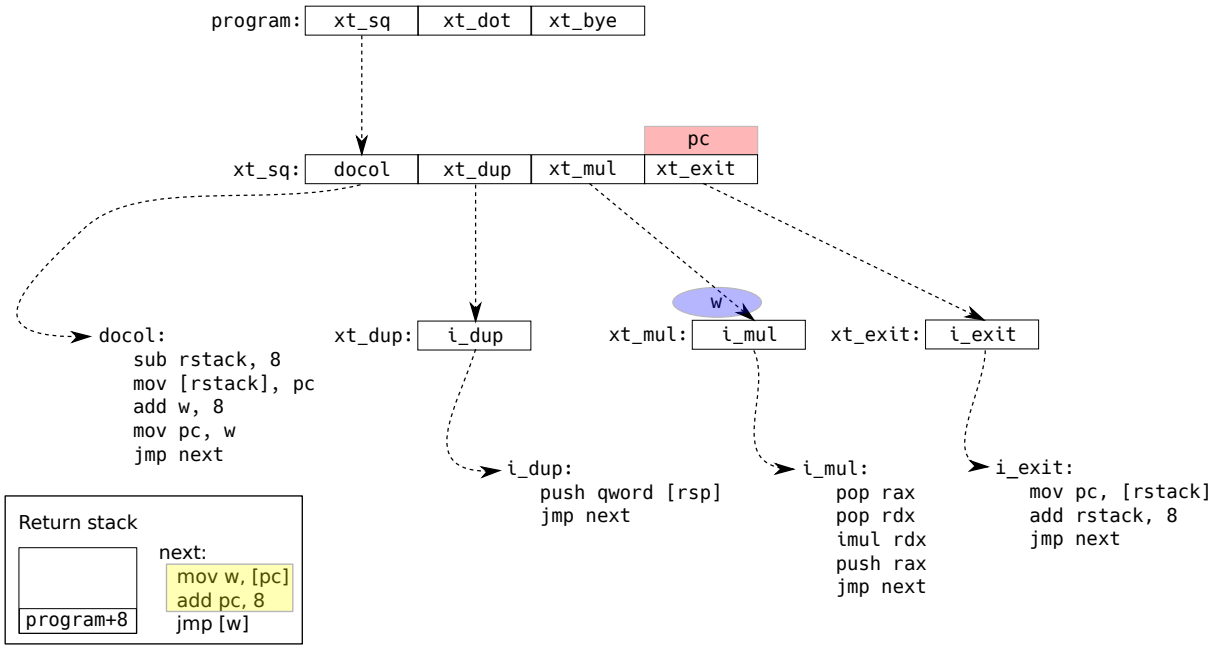
Шаг 6. На этом шаге мы видим, как начинает выполняться нативное слово dup. Выполнить нативное слово проще, достаточно выполнить его ассемблерную реализацию. Поэтому на этом шаге мы всего лишь используем w чтобы пройти один уровень косвенности и продвинуть счётчик команд pc вперёд.

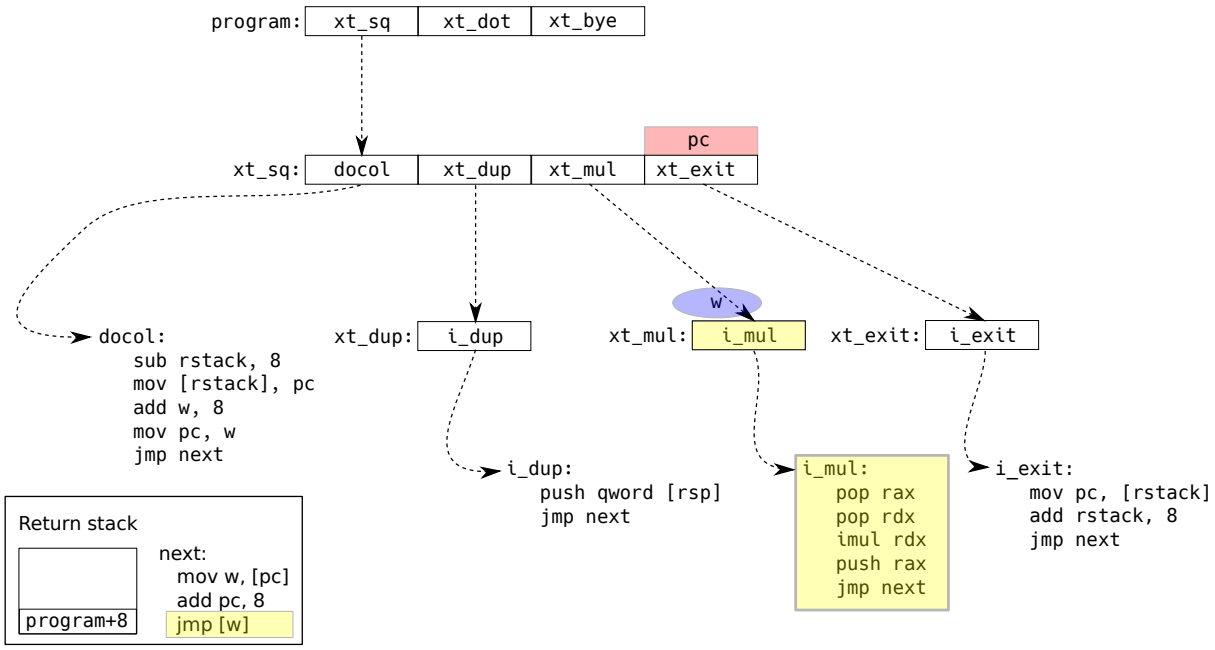


Шаг 7. Здесь мы просто запускаем ассемблерную реализацию слова dup.

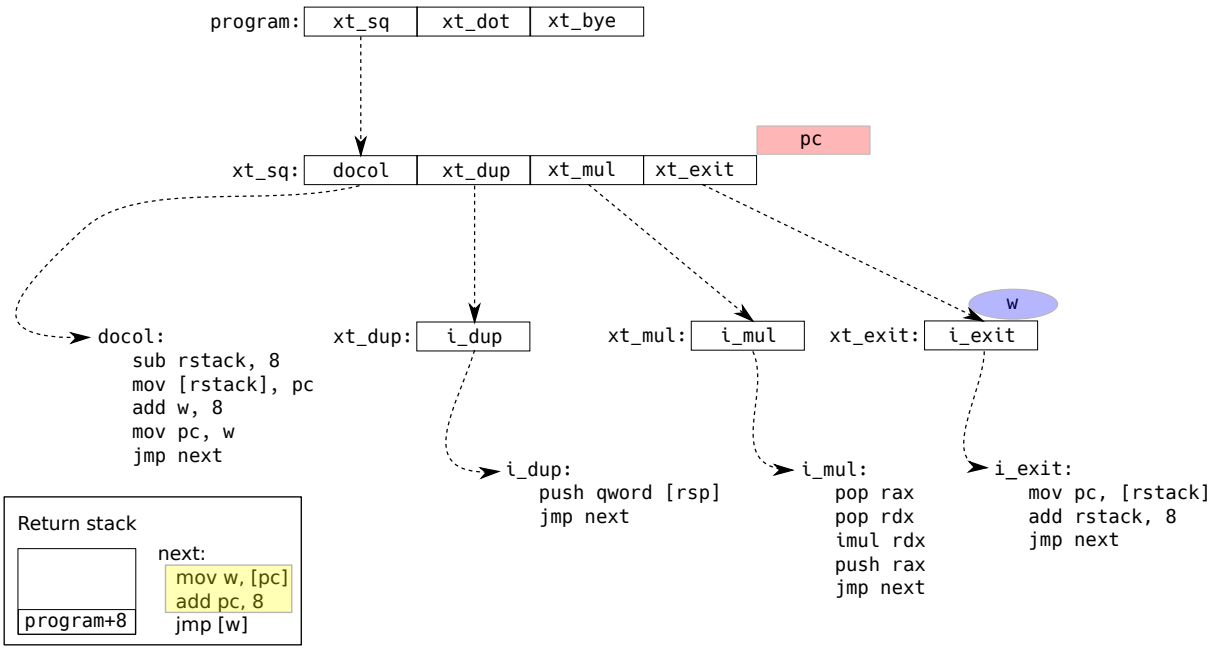


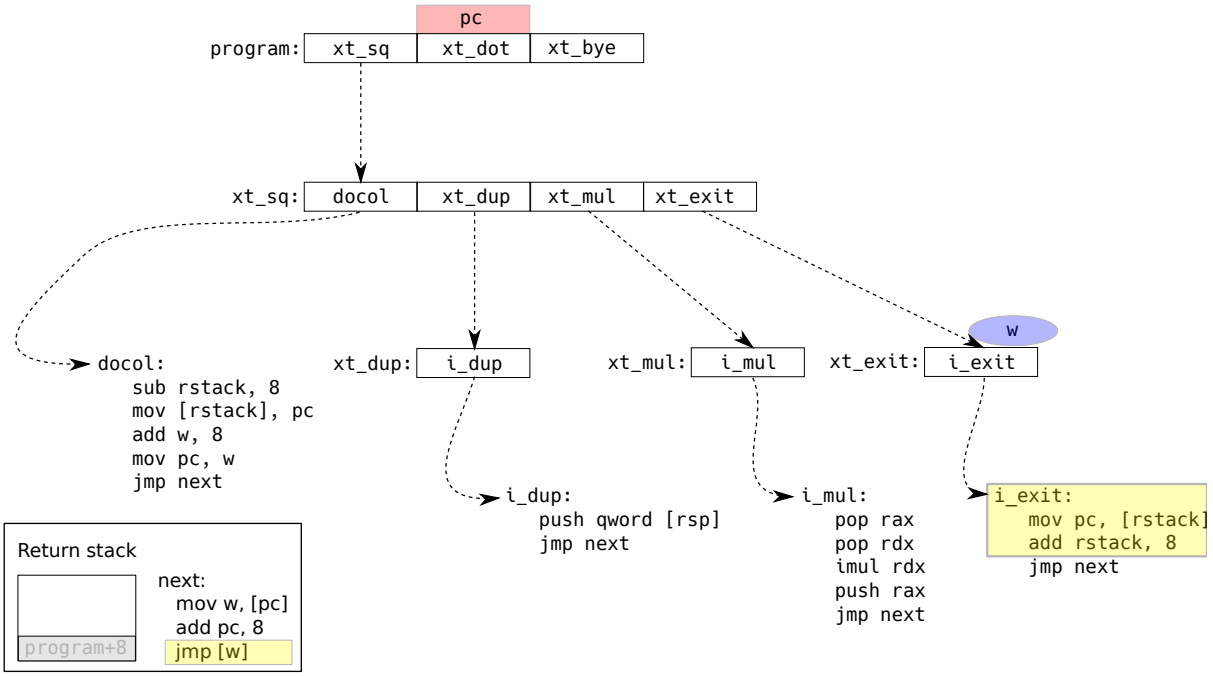
Шаг 8 и Шаг 9. Эти шаги аналогичны шагам 6 и 7 с той разницей, что теперь выполняется нативное слово `*`, а не `dup`.





Шаги 10 и 11. Теперь мы запускаем особенное слово `exit`, которым заканчивается любое colon-слово. Его смысл в том, чтобы вернуть `pc` в состояние до вызова текущего слова. В этом плане оно похоже на ассемблерную инструкцию `ret`.





Теперь мы продолжаем выполнение программы со следующей инструкции после вызова слова `sq`.

3.1.3 Слова `lit` и `branch`/`0branch`

В словах, которые должны класть на стек непосредственно заданные значения, нельзя просто вписать число в поток инструкций – оно будет воспринято как XT и Forthress попытается по нему перейти. Чтобы этого избежать, используется специальное слово `lit`, которое кладёт операнд по адресу `[pc]` в стек и увеличивает `pc` на размер ячейки.

Аналогично, слова branch/`0branch`, кладущие новое значение в `pc`, берут его из потока инструкций Forth-машины.

3.2 Словарь

Чтобы слова можно было искать динамически (например, во время определения новых слов), они все объединяются в словарь. Это позволит нам в дальнейшем сделать интерпретатор, который будет искать XT слова по его имени.

Словарь организован как связный список слов, в начало которого добавляются новые слова. Поэтому заголовок начинается со ссылки на следующее слово; последнее слово в словаре хранит в этом поле нули.

Каждое слово снабжается метаинформацией: именем и флагами.

Для начала рассмотрим строение нативного слова `dup` (Рис. 3.1).

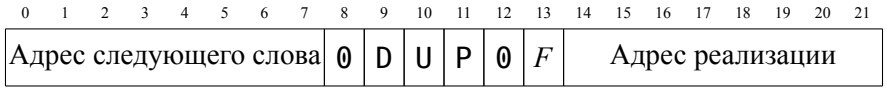


Рис. 3.1: Нативное слово `dup` в словаре.

Затем идёт нулевой байт[1] и название слова в виде нуль-терминированной строки. После этого один байт *F* зарезервирован для флага `immediate`. Наконец, далее идёт адрес реализации слова на ассемблере. Напомним, что адрес поля “Адрес реализации” – это *execution token* слова.

Пример 3.2.1. Закодируем словарь, в котором есть только слова `dup` и `*`.

```
section .data

last_word: dq wh_mul        ; dictionary start


wh_mul:  dq wh_dup          ; next word
         db 0               ; separator
         db "*", 0          ; name
         db 0               ; flags
xt_mul:  dq i_mul           ; link to implementation

wh_dup:  dq 0               ; next word
         db 0               ; separator
         db "dup", 0        ; name
```

[1] Позволяет определить имя слова по его XT.

```
          db 0                ; flags
xt_dup:   dq i_dup            ; link to implementation


section .text

i_mul:    ...                 ; implementation of 'mul'

i_dup:    ...                 ; implementation of 'dup'
```

Теперь поговорим о colon-словах. Как мы знаем, у всех colon-слов одинаковая реализация `docol`, а непосредственно за адресом `docol` идёт последовательность XT слов из определения. В словаре заголовки colon-слово выглядят так же, как и нативных.

Мы неоднократно использовали для примера слово `sq`.

```
: sq dup * ;
```

Его запись в словаре вместе с заголовком изображена на Рис. 3.2.

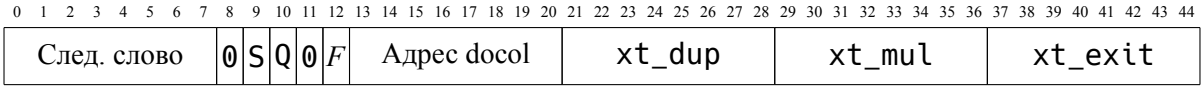


Рис. 3.2: colon-слово `sq` в словаре.

Напоминаем, что каждое colon-слово завершается словом `exit`, которое аналогично инструкции ассемблера `ret`. Оно достаёт из стека возвратов адрес возврата и переходит по нему.

Файл `src/words.inc` определяет начальное значение словаря в момент запуска Forthress.

3.3 DSL для описания словаря

Если мы будем кодировать словарь так, как делали это в прошлом разделе, мы просто сойдём с ума от количества повторяющегося кода. Кроме того нужно будет следить за правильностью составления словаря: все слова должны выстроиться в последовательность, никакое слово не должно из неё выпасть.

Логичным решением является использование метапрограммирования для того, чтобы ассемблер генерировал макросами код за нас, а мы максимально кратко описывали бы Forth-слова, добавляя к ним декларативную информацию (например, о флагах).

NASM предоставляет нам достаточно мощную систему текстовых макросов, которые могут существенно облегчить кодирование словаря. Нам понадобятся: `%macro` / `%endmacro`, `%define`, `%+`.

Благодаря тому, что `%define` можно использовать многократно для одного и того же препроцессорного символа, мы можем с его помощью поддерживать такое макроопределение, которое содержит последнее определённое статически слово. Определение каждого следующего слова должно обновлять эту ссылку, записывая туда свой адрес.

Для начала мы создадим макрос `native`, который принимает 3 параметра: имя слова, часть идентификатора, из которой мы составим имя меток (например, `wh_dup`, `i_dup` и `xt_dup`), а также флаги. Он создаёт в секции данных кусок словаря, а в секции кода – метку, по которой находится реализация слова на ассемблере.

```
; last word
%define _lw 0

%macro native 3
    section .data
    wh_ %+ %2 : dq _lw
    db 0
    db %1, 0
    db %3

; update the reference to the last word
% define _lw wh_%+ %2

xt_ %+ %2 :  dq i_ %+ %2

; implementation starts here
    section .text
    i_ %+ %2:
%endmacro
```

Сравните удобство описания слов на ассемблере без макросов и с их помощью.

Без макросов:

```
section .data

w_plus:  dq w_mul    ; previous
         db '+',0
         db 0
xt_plus: dq plus_impl

section .text

i_plus:  pop rax
         add [rsp], rax
         jmp next
```

С макросами:

```
native '+', plus, 0
     pop rax
     add [rsp], rax
     jmp next
```

Так как большинство слов не снабжены никакими флагами, то можно создать перегрузку макроса `native` с двумя параметрами, которая содержит в себе вызов полной версии с нулём в качестве флагов:

```
%macro native 2
native %1, %2, 0
%endmacro
```

Самостоятельно создайте макрос colon, полностью аналогичный предыдущему, но рассчитанный на то, чтобы создавать colon-слова как в примере ниже:

```
colon '>', greater
   dq xt_swap, xt_less, xt_exit
```

Не забудьте про адрес `docol` в первой ячейке после заголовка записи в словаре!

3.4 Задание 2, часть 1: ядро Forthress

Мы предлагаем вам реализовать Forth через технику, называемую bootstrap. Её суть в том, чтобы реализовать на ассемблере минимальный набор слов, необходимый для написания с их помощью цикла интерпретации/компиляции, и среду для их выполнения (внутренний интерпретатор). Затем можно просто запустить на ассемблере Forth-слово, считывающее текст с потока ввода, разбирающее его на слова и выполняющее их.

Чтобы задание было легче, мы разбили его на две части. В первой части вам необходимо реализовать ядро Forthress: закодировать корректное начальное состояние, реализовать все необходимые для написания интерпретатора нативные слова на ассемблере. Требуется, чтобы машину можно было настроить на выполнение непосредственно закодированной на ассемблере последовательности Forth-слов, как в примере на странице 32.

Минимальный набор слов Чтобы закодировать интерпретатор на самом Forthress, вам потребуются как минимум следующие базовые слова, которые можно выразить только на ассемблере.

- Стековые комбинаторы:

  ```
  drop swap dup
  ```

- Арифметические операции:

  ```
  + * / % - = <
  ```

- Логические операции:

  ```
  not and or land lor
  ```

- Работа со стеком возвратов:

  ```
  >r r> r@
  ```

- Работа с памятью

  ```
  @ ! c! c@
  execute
  forth-dp
  ```

- Управление выполнением:

```
docol
branch 0branch
exit
```

- Служебные:

```
lit
forth-sp
forth-stack-base
```

- Интерфейс системных вызовов:

```
syscall  ( call_num a1 a2 a3 a4 a5 a6 -- new_rax new_rdx )
```

Добавьте к этому списку еще слово `.s` для просмотра состояния стека. Его можно реализовать на самом Forthress, т.к. доступ к стековой памяти напрямую можно получить с помощью слов `forth-sp` и `forth-stack-base`, но сделать это сложнее. С помощью `.S` протестируйте эти слова. Для этого вы можете писать небольшие программы как в примере на странице 32.

Например, так можно протестировать сложение (`xt_show_stack` это XT слова `.S`):

```
dq xt_lit, 40, xt_lit, 2, xt_plus, xt_show_stack
```

Первую часть задания не нужно присылать; задание разбито на две части для вашего удобства.

Литература

[1] Simon L Peyton Jones and David R Lester. Implementing Functional Languages: a tutorial. Technical report, Technical Report, University of Glasgow, 2000.

[2] Kressin Yosef. Defective C++. `http://yosefk.com/c++fqa/defective.html`.