Fift: Краткое введение

Николай Дуров 6

февраля 2020

Аннотация

Целью данного текста является краткое описание Fift, нового языка программирования, специально разработанного для создания и управления смарт-контрактами TON Blockchain, и его функций, используемых для взаимодействия с виртуальной машиной TON [4] и блокчейном TON [5].

Введение

В этом документе приводится краткое описание Fift, основанного на стеке языка программирования общего назначения, оптимизированного для создания, отладки и управления смарт-контрактами TON Blockchain.

Fift был специально разработан для взаимодействия с виртуальной машиной TON (TON VM или TVM) [4] и блокчейном TON [5]. В частности, он предлагает встроенную поддержку 257-битной целочисленной арифметики и манипуляций с ячейками TVM, совместно используемых с TVM, а также интерфейс к криптографии на основе Ed25519, используемой TON Blockchain. Макроассемблер для TVM-кода, полезный для написания новых смарт-контрактов, также входит в состав
Fift дистрибуция.

Будучи языком, основанным на стеке, Fift мало чем отличается от Forth. Из-за краткости этого текста некоторые знания Форта могут быть полезны для понимания Fift. [1] Тем не менее, существуют значительные различия между двумя языками. Например, Fift принудительно проверяет типы среды выполнения и сохраняет значения различных типов (не только целые числа) в своем стеке.

[1] Хорошие введения в Forth существуют; мы можем рекомендовать [1].

Список слов (встроенных функций или примитивов), определенных в Fift, вместе с их краткими описаниями представлен в Приложении А.

Обратите внимание, что текущая версия этого документа описывает предварительную тестовую версию Fift; некоторые незначительные детали, вероятно, изменятся в будущем.

Содержание

1 Обзор

Fift — это простой стековый язык программирования, предназначенный для тестирования и отладки виртуальной машины TON [4] и блокчейна TON [5], но потенциально полезный и для других целей. При вызове Fift (обычно путем выполнения двоичного файла с именем fift) он либо считывает, анализирует и интерпретирует один или несколько исходных файлов, указанных в командной строке, либо переходит в интерактивный режим и интерпретирует Fift команды, прочитанные и проанализированные из стандартного ввода. Существует также «режим скрипта», активируемый параметром командной строки -s, в котором все аргументы командной строки, кроме первого, передаются в Fift программу с помощью переменных $*n* и $#. Таким образом, Fift можно использовать как для интерактивных экспериментов и отладки, так и для написания простых скриптов.

Все данные, которыми манипулирует Fift, хранятся в стеке (LIFO). Каждая запись стека дополняется *type тегом*, который однозначно определяет тип значения, хранящегося в соответствующей записи стека. Типы значений, поддерживаемые Fift включают *Integer* (представляющий знаковые 257-битные целые числа), *Cell* (представляющий ячейку TVM, которая состоит из 1023 бит данных и до четырех ссылок на другие ячейки, как описано в [4]), *Slice* (частичное представление *ячейки,* используемое для синтаксического анализа ячеек) и *Builder* (используется для строительства новых ячеек). Эти типы данных (и их реализации) совместно используются с TVM [4] и могут быть безопасно переданы из стека Fift в стек TVM и обратно при необходимости (например, когда TVM вызывается из Fift с помощью Fift примитива, такого как runvmcode).

В дополнение к типам данных, совместно используемым с TVM, Fift вводит некоторые уникальные типы данных, такие как *Bytes* (произвольные последовательности байтов), *String* (строки UTF-8), *WordList* и *WordDef* (используемые Fift для создания новых «слов» и манипулирования их определениями). Фактически, Fift могут быть расширены для манипулирования произвольными «объектами»

(представленными универсальным типом *Object*), при условии, что они являются производными от класса C++ td::CntObject в текущей реализации.

Fift исходные файлы и библиотеки обычно хранятся в текстовых файлах с суффиксом .fif. Путь поиска библиотек и включенных файлов передается исполняемому файлу Fift либо в аргументе командной строки -I, либо в переменной среды FIFTPATH. Если ни один из них не задан, используется путь поиска библиотеки по умолчанию /usr/lib/fift.

При запуске стандартная библиотека Fift считывается из файла Fift.fif перед интерпретацией любых других источников. Он должен присутствовать в пути поиска библиотеки, иначе Fift выполнение завершится ошибкой.

Фундаментальной Fift структуры данных является ее *глобальный словарь*, содержащий *слова*, или, точнее, *определения слов,* которые соответствуют как встроенным примитивам и функциям, так и определяемым пользователем функциям.[2] Слово может быть выполнено в Fift, просто набрав его имя (строка UTF-8 без пробелов) в интерактивном режиме. Когда Fift запускается, некоторые слова (*примитивы*) уже определены (некоторым кодом C++ в текущей реализации); другие слова определены в стандартной библиотеке Fift.fif. После этого пользователь может расширить словарь, определив новые слова или переопределив старые.

Предполагается, что словарь должен быть разделен на несколько *словарей* или *пространств имен*; однако пространства имен еще не реализованы, поэтому все слова в настоящее время определяются в одном глобальном пространстве имен.

Синтаксический анализатор Fift для входных исходных файлов и для стандартного ввода (в интерактивном режиме) довольно прост: ввод читается построчно, затем пропускаются пустые символы, а самый длинный префикс оставшейся строки, которая является (названием)

[2]Fift слова, как правило, короче, чем функции или подпрограммы других языков программирования. Хорошее обсуждение и некоторые рекомендации (для четвертых слов) можно найти в [2].

словарного слова, обнаруживается и удаляется из строки ввода.[3] После этого слово, найденное таким образом, выполняется, и процесс повторяется до конца строки. Когда входная строка исчерпана, последующая строка считывается из текущего входного файла или стандартного входа.

Чтобы быть обнаруженными, большинство слов требуют пустого символа или конца строки сразу после них; это отражается добавлением пробела к их именам в словаре. Другие слова, называемые *префиксными словами*, не требуют пустого символа сразу после них.

Если слово не найдено, строка, состоящая из первых оставшихся символов входной строки до следующего пустого символа или символа конца строки, интерпретируется как *целое число* и помещается в стек. Например, если мы вызываем Fift, введите 2 3 + . (и нажмите клавишу ВВОД), Fift сначала помещает *в стек целочисленную* константу, равную 2, а затем еще одну целочисленную константу, равную 3. После этого встроенный примитив «+» разбирается и находится в словаре; при вызове он берет два самых верхних элемента из стека и заменяет их их суммой (5 в нашем примере). Наконец, «.» — это примитив, который печатает десятичное представление целого числа в верхней части стека, за которым следует пробел. В результате мы наблюдаем «5 ok», напечатанное Fift интерпретатором в стандартный вывод. Строка "ok" печатается интерпретатором всякий раз, когда он заканчивает перевод строка, считываемая со стандартного ввода в интерактивном режиме.

2 Основы Fift

В этой главе представлено введение в основные возможности Fift языка программирования. Обсуждение сначала неформальное и неполное, но постепенно становится более формальным и более точным. В некоторых случаях в последующих главах и Добавлении A содержится более подробная информация о словах, впервые упомянутых в этой главе; аналогичным образом, некоторые трюки, которые будут покорно

[3]Обратите внимание, что в отличие от Форта, названия Fift слов чувствительны к регистру: dup и DUP — это разные слова.

объяснены в последующих главах, уже используются здесь, где это уместно.

2.1 Список типов значений стека Fift

В настоящее время значения следующих типов данных можно хранить в стеке Fift:

- *Integer* — знаковое 257-битное целое число. Обычно обозначается x, y или z в нотации стека (когда описывается эффект стека Fift слова).
- *Cell* — ячейка TVM, состоящая из бита данных до 1023 и до 4 ссылок на другие ячейки (ср. [4]). Обычно обозначается c или его вариантами, такими как *c0* или *c2*.
- *Slice* — частичное представление ячейки TVM, используемое для синтаксического анализа данных из *ячеек. Обычно обозначается* s.
- *Builder* — частично построенная ячейка, содержащая до 1023 бит данных и до четырех ссылок; может использоваться для создания новых *ячеек. Обычно обозначается буквой* b.
- *Null* — тип с ровно одним значением null. Используется для инициализации новых *полей. Обычно обозначается* $\perp$.
- *Tuple* — упорядоченная коллекция значений любого из этих типов (не обязательно одинаковых); может использоваться для представления значений произвольных алгебраических типов данных и списков в стиле Lisp.
- *String* — строка (обычно печатаемая) в кодировке UTF-8. Обычно обозначается S.
- *Bytes* — произвольная последовательность 8-битных байтов, обычно используемая для представления двоичных данных. Обычно обозначается буквой B.

- *WordList* — (частично созданный) список ссылок на слова, используемый для создания новых Fift определений слов. Обычно обозначается l.
- *WordDef* — маркер выполнения, обычно представляющий определение существующего Fift слова. Обычно обозначается e.
- *Box* — место в памяти, которое можно использовать для хранения одного значения стека. Обычно обозначается .
- *Atom* — простая сущность, однозначно идентифицируемая по своему имени, строке. Может использоваться для представления идентификаторов, меток, имен операций, тегов и маркеров стека. Обычно обозначается буквой a.
- *Object* — произвольный объект C++ любого класса, производного от базового класса td::CntObject; может использоваться расширениями Fift для управления другими типами данных и взаимодействия с другими библиотеками C++.

Первые шесть типов, перечисленных выше, являются общими для TVM; остальные Fift специфичны. Обратите внимание, что не все типы стека TVM присутствуют в Fift. Например, тип TVM *Continuation* явно не распознается Fift; если значение этого типа оказывается в стеке Fift, им манипулируют как универсальным *Объектом*.

2.2 Комментарии

Fift распознает два вида комментариев: "// " (за которым должен следовать пробел) открывает однострочный комментарий до конца строки, а /* определяет многострочный комментарий до */. Оба слова // и /* определены в стандартной библиотеке Fift (Fift.fif).

2.3 Прекращение Fift

Слово bye завершает интерпретатор Fift нулевым кодом выхода. Если требуется ненулевой код выхода (например, в Fift скриптах), можно использовать слово halt, которое завершает Fift заданным кодом выхода (передается как *Integer* в верхней части стека). Напротив, quit не выходит в операционную систему, а скорее выходит на верхний уровень интерпретатора Fift.

2.4 Простая целочисленная арифметика

Когда Fift сталкивается со словом, отсутствующим в словаре, но которое можно интерпретировать как целочисленную константу (или «литерал»), его значение выталкивается в стек (как объяснено в 2.8 более подробно). Кроме того, определено несколько целочисленных арифметических примитивов:

- + ($x\ y - x + y$), заменяет два *целых числа* x и y, переданных в верхней части стека, их суммой $x+y$. Все более глубокие элементы стека остаются нетронутыми. Если x или y не является целым числом или если сумма не помещается в знаковое 257-разрядное *Integer*, создается исключение.
- - ($x\ y - x - y$), вычисляет разность $x - y$ двух *целых чисел* x и y. Обратите внимание, что первый аргумент x является второй записью из верхней части стека, в то время как второй аргумент y берется из верхней части стека.
- negate ($x - -x$), изменяет знак целого *числа*.
- * ($x\ y - xy$), вычисляет произведение xy двух *целых чисел* x и y.
- / ($x\ y - q := \lfloor x/y \rfloor$), вычисляет коэффициент $\lfloor x/y \rfloor$ с округлением до меньшего целого из двух *Целых чисел.*
- mod ($x\ y - r := x \bmod y$), вычисляет остаток $x \bmod y = x - y \cdot \lfloor x/y \rfloor$ деления x на y.

- /mod ($x\ y - q\ r$), вычисляет как частное, так и остаток.
- /c, /r ($x\ y - q$), слова деления, похожие на /, но использующие округление потолка ($q := \lceil x/y \rceil$) и ближайшее целочисленное округление ($q := \lfloor 1/2 + x/y \rfloor$) соответственно.
- /cmod, /rmod ($x\ y - q\ r := x - qy$), слова деления, похожие на /mod, но использующие потолочное или ближайшее целое число округления.
- << ($x\ y - x \cdot 2^y$), вычисляет арифметический сдвиг влево двоичного числа x на $y \geq 0$ позиций, что дает $x \cdot 2^y$.
- >> ($x\ y - q := \lfloor x \cdot 2^{-y} \rfloor$), вычисляет арифметический сдвиг вправо на $y \geq 0$ позиций.
- >>c, >>r ($x\ y - q$), аналогично >>, но с использованием потолочного или ближайшего целочисленного округления.
- and, or, xor ($x\ y - x \oplus y$), вычисляют побитовое значение AND, OR или XOR двух *целых чисел.*
- not ($x - -1 - x$), побитовое дополнение целого *числа*.
- */ ($x\ y\ z - \lfloor xy/z \rfloor$), "умножьте-затем-разделите": умножает два целых числа x и y , получая 513-битный промежуточный результат, затем делит произведение на z.
- */mod ($x\ y\ z - q\ r$), аналогично */, но вычисляет как частное, так и остаток.
- */c, */r ($x\ y\ z - q$), */cmod, */rmod ($x\ y\ z - q\ r$), аналогично */ или */mod, но с использованием потолочного или ближайшего целочисленного округления.
- *>>, *>>c, *>>r ($x\ y\ z - q$), аналогично */ и его вариантам, но с разделением, замененным сдвигом вправо. Вычисление $q = xy/2^z$ с округлением указанным образом (пол, потолок или ближайшее целое число).

- <</, <</c, <</r ($x\ y\ z - q$), аналогично */, но с умножением заменено сдвигом влево. Вычисление $q = 2^z x / y$, округленное указанным образом (обратите внимание на различный порядок аргументов y и z по сравнению с */).

Кроме того, слово «.» может использоваться для печати десятичного представления целого *числа,* переданного в верхней части стека (за которым следует один пробел), а «x.» печатает шестнадцатеричное представление целого числа верхней части стека. После этого целое число удаляется из стека.

Вышеуказанные примитивы могут быть использованы для использования интерпретатора Fift в интерактивном режиме в качестве простого калькулятора для арифметических выражений, представленных в обратной польской нотации (с символами операций после операндов). Например,

```
7 4 - .
```

вычисляет $7 - 4 = 3$ и печатает "3 ок", и

```
2  3 4 * + .
2 3 + 4 * .
```

вычисляет $2 + 3 \cdot 4 = 14$ и $(2 + 3) \cdot 4 = 20$ и печатает "14 20 ok".

2.5 Слова для работы со стеком

Слова для работы со стеком изменяют порядок одного или нескольких значений в верхней части стека, независимо от их типов, и оставляют все более глубокие значения стека нетронутыми. Некоторые из наиболее часто используемых слов для работы со стеком перечислены ниже:

- dup ($x\ -\ x\ x$), дублирует верхнюю часть стека. Если стек пуст, создается исключение.[4]

- drop (x –), удаляет запись в верхней части стека.

[4]Обратите внимание, что имена Fift слов чувствительны к регистру, поэтому нельзя вводить DUP вместо dup.

- swap ($x\ y - y\ x$), заменяет две самые верхние записи стека.
- rot ($x\ y\ z - y\ z\ x$), вращает три самые верхние записи стека.
- -rot ($x\ y\ z - z\ x\ y$), поворачивает три самые верхние записи стека в противоположную сторону. Эквивалент rot rot.
- over ($x\ y - x\ y\ x$), создает копию второй записи стека сверху над записью верхней части стека.
- tuck ($x\ y - y\ x\ y$), эквивалентный swap over.
- nip ($x\ y - y$), удаляет вторую запись стека сверху. Эквивалентно swap drop.
- 2dup ($x\ y - x\ y\ x\ y$), эквивалентно over over.
- 2drop ($x\ y -$), эквивалентно drop drop.
- 2swap ($a\ b\ c\ d - c\ d\ a\ b$), заменяет две самые верхние пары записей стека.
- pick (x_n ... $x_0\ n - x_n$ ... $x_0\ x_n$), создает копию n-й записи из верхней части стека, где $n \geq 0$ также передается в стеке. В частности, 0 pick эквивалентен dup, а 1 pick эквивалентно over.
- roll (x_n ... $x_0\ n - x_{n-1}$ ... $x_0\ x_n$), поворачивает верхние n записей стека, где $n \geq 0$ также передается в стеке. В частности, 1 roll эквивалентен swap, а 2 roll эквивалентно rot.
- -roll (x_n ... $x_0\ n - x_0\ x_n$ ... x_1), поворачивает верхние n записей стека в обратном направлении, где $n \geq 0$ также передается в стеке. В частности, 1 -roll эквивалентен swap, а 2 -roll эквивалентно -rot.
- exch (x_n ... $x_0\ n - x_0$ ... x_n), заменяет верхнюю часть стека n-й записью стека сверху, где $n \geq 0$ также берется из стека. В частности, 1 exch эквивалентно swap, а 2 exch для swap rot.

- exch2 (... n m – ...), заменяет n-ю запись стека сверху с m-й записью стека сверху, где $n \geq 0$, $m \geq 0$ берутся из стэка.
- ?dup (x – x x или 0), дублирует *Integer* x, но только если оно не равно нулю. В противном случае оставляет его нетронутым.

Например, "5 dup * ." вычисляет $5 \cdot 5 = 25$ и печатает "25 ok".

Можно использовать слово «.s», которое печатает содержимое всего стека, начиная с самых глубоких элементов, не удаляя элементы, напечатанные из стека, чтобы в любое время проверить содержимое стека и проверить эффект любых слов манипуляции стеком. Например

```
1 2 3 4 .s
rot .s
```

Отпечатки

```
1 2 3 4
ok
1 3 4 2
ok
```

Когда Fift не знает, как напечатать значение стека неизвестного типа, вместо этого он печатает ???.

2.6 Определение новых слов

В своей простейшей форме определение новых Fift слов очень легко и может быть сделано с помощью трех специальных слов: «{», «}» и «:». Один просто открывает определение с помощью { (обязательно с последующим пробелом), затем перечисляет все слова, которые составляют новое определение, затем закрывают определение на } (за которым также следует пробел) и, наконец, присваивает результирующее определение (представленное значением *WordDef* в стеке) новому слову, написав: ⟨new-word-name⟩.

Например

```
{ dup * } : square
```

определяет новое слово square, который при вызове выполняет dup и *. Таким образом, ввод 5 square становится эквивалентным набору 5 dup *, и дает тот же результат (25):

```
5 square .
```

печатает "25 ок". Можно также использовать новое слово как часть новых определений:

```
{ dup square square * } : **5
3 **5 .
```

печатает «243 ok», что действительно 3^5.

Если слово, указанное после ":", уже определено, оно молчаливо переопределяется. Тем не менее, все существующие определения других слов будут по-прежнему использовать старое определение переопределенного слова. Например, если мы переопределим square после того, как мы уже определили **5, как указано выше, **5 будет продолжать использовать исходное определение square.

2.7 Именованные константы

Можно определить *(именованные) константы*, то есть слова, которые при вызове передают предопределенное значение, используя определяющую константу слова вместо определяющего слова «:» (двоеточие). Например

```
10000000000 константа Gram
```

определяет константу Gram, равную целому числу 10^9. Другими словами, 1000000000 будет вставлен в стек всякий раз, когда вызывается Gram:

```
Gram 2* .
```

напечатает "2000000000 ок".

Конечно, можно использовать результат вычисления для инициализации значения константы:

```
Gram 1000 / константа mGram
mGram.
```

печатает "1000000 ок".

Значение константы не обязательно должно быть целым числом. Например, можно определить строковую константу таким же образом:

```
«Hello, world!» константа hello
hello type cr
```

печатает «Hello, world!» на отдельной строке.

Если константа будет переопределена, все существующие определения других слов будут продолжать использовать старое значение константы. В этом отношении константа не ведет себя как глобальная переменная.

Можно также сохранить два значения в одну «двойную» константу, используя определяющее слово 2constant. Например

```
355 113 2constant pifrac
```

определяет новое слово pifrac, которое при вызове будет отправлять 355 и 113 (в указанном порядке). Две составляющие двойной константы могут быть разных типов.

Если кто-то хочет создать константу с фиксированным именем в определении блока или двоеточия, следует использовать =: и 2=: вместо constant и 2constant:

```
{ dup =: x dup * =: y } : setxy
3 setxy x . y . x y + .
7 setxy x . y . x y + .
```

производит

```
3 9 12 ок
7 49 56 ок
```

Если кто-то хочет восстановить значение времени выполнения такой «константы», он может префиксировать имя константы словом @':

```
{ ." ( " @' x . ." , " @' y . .") " } : showxy
```

```
3 setxy showxy
```

производит

```
( 3 , 9 ) ок
```

Недостатком этого подхода является то, что @' должен искать текущее определение констант x и y в словаре каждый раз при выполнении showxy. Переменные (см. 2.14) обеспечивают более эффективный способ достижения аналогичных результатов.

2.8 Целочисленные и дробные константы, или литералы

Fift распознает безымянные целочисленные константы (называемые *литералами*, чтобы отличить их от именованных констант) в десятичном, двоичном и шестнадцатеричном форматах. Двоичные литералы имеют префикс 0b, шестнадцатеричные литералы префикс 0x, а десятичные литералы не требуют префикса. Например, 0b1011, 11 и 0xb представляют одно и то же целое число (11). Целочисленный литерал может иметь префикс знаком минус "-" для изменения его знака; знак минус принимается как до, так и после префиксов 0x и 0b.

Когда Fift встречает строку, отсутствующую в словаре, но являющуюся допустимым целочисленным литералом (вписывающимся в 257-битное знаковое целое число типа *Integer*), его значение передается в стек.

Кроме того, Fift предлагает некоторую поддержку десятичных и общих дробей. Если строка состоит из двух допустимых целочисленных литералов, разделенных косой чертой /, то Fift интерпретирует ее как дробный литерал и представляет ее двумя целыми *числами* p и q в стеке, числителем p и знаменателем q. Например, -17/12 толкает -17 и 12 в стек Fift (будучи эквивалентным -17 12), а -0x11/0b1100 делает то же самое. Десятичные, двоичные и шестнадцатеричные дроби, такие как 2.39 или -0x11.ef, также представлены двумя целыми числами p и q, где q — подходящая степень основания (10, 2 или 16 соответственно). Например, 2.39 эквивалентен 239 100, а -0x11.ef эквивалентен -0x11ef 0x100.

Такое представление дробей особенно удобно для их использования с примитивом масштабирования */ и его вариантами, тем самым

преобразуя общие и десятичные дроби в подходящее представление с фиксированной точкой. Например, если мы хотим представить дробные количества граммов целыми количествами нанограммов, мы можем определить некоторые вспомогательные слова.

```
10000000000 constant Gram
{ Gram * } : Gram *
{ Gram swap */r } : Gram */
```

и затем запишите 2.39 Gram*/ или 17/12 Gram*/ вместо целочисленных литералов 2390000000 или 1416666667.

Если нужно часто использовать такие литералы Gram, можно ввести новое активное префиксное слово GR$ следующим образом:

```
{ bl word (number) ?dup 0= abort"not a valid Gram amount"
   1- { Gram swap */r } { Gram * } cond
   1 'nop
} ::_ GR$
```

делает GR$3, GR$2.39 и GR$17/12 эквивалентным целочисленным литералам 3000000000, 2390000000 и 1416666667 соответственно. Такие значения могут быть напечатаны в аналогичной форме с помощью следующих слов:

```
{ dup abs <# ' # 9 times char . hold #s rot sign #>
  nip -trailing0 } : (. GR)
{ (. GR) . "GR$" type space } : . GR
-17239000000 . GR
```

производит GR$-17.239 ок. В приведенных выше определениях используются трюки, описанные в последующих частях этого документа (особенно в главе 4).

Мы также можем манипулировать дробями сами по себе, определяя подходящие «рациональные арифметические слова»:

```
// a b c d -- (a*d-b*c) b*d
{ -rot over * 2swap tuck * rot - -rot * } : R-
// a b c d -- a*c b*d
```

```
{ rot * -rot * swap } : R*
// a b - -
{ swap ._ ." /" . } : R.
1.7 2/3 R- R.
```

выведет "31/30 ok", указывая, что $1.7 - 2/3 = 31/30$. Здесь "._" является вариантом ".", который не печатает пробел после десятичного представления целого *числа*.

2.9 Строковые литералы

Строковые литералы вводятся с помощью префиксного слова ", которое сканирует оставшуюся часть строки до следующего символа ", и выталкивает полученную таким образом строку в стек как значение типа *String*. Например, "Hello, world!" вставляет соответствующую *строку* в стек:

```
"Hello, world!" .s
```

2.10 Простая манипуляция со строками

Для управления строками можно использовать следующие слова:

- "⟨*string*⟩" (– *S*), проталкивает *литерал String* в стек.
- ." ⟨*string*⟩" (–), выводит константную строку в стандартный вывод.
- type (*S* –), выводит *String* *S* , взятую из верхней части стека, в стандартный вывод.
- cr (–), выводит возврат каретки (или символ новой строки) в стандартный вывод.
- emit (*x* –), выводит символ в кодировке UTF-8 с кодовой точкой Юникода, заданной целым *числом* *x* , в стандартный вывод.
- char ⟨*string*⟩ (– *x*), отправляет целое число с кодовой точкой Юникода первого символа ⟨*string*⟩.
- bl (– *x*), толкает кодовую точку Юникода пробела, т.е. 32.

- space (–), печатает один пробел, эквивалентный bl emit.
- $+ ($S\ S' – S.S'$), объединяет две строки.
- $len ($S – x$), вычисляет длину байта (не длину символа UTF-8!) строки.
- +"⟨*string*⟩" ($S\ –\ S'$), объединяет *String* S со строковым литералом. Эквивалент "⟨*string*⟩" $+.
- word ($x\ –\ S$), анализирует слово, разделенное символом, с кодовой точкой Юникода x из оставшейся части текущей входной строки и отправляет результат в виде *String*. Например, bl word abracadabra type будет печатать строку «abracadabra». Если $x = 0$, пропускает начальные пробелы, а затем сканирует до конца текущую входную строку. Если $x = 32$, пропускает начальные пробелы перед синтаксическим анализом следующего слова.
- (.) ($x\ –\ S$), возвращает *значение String* с десятичным представлением целого *числа* x.
- (number) ($S\ –\ 0$ или $x\ 1$ или $x\ y\ 2$), пытается разобрать строку S как целое число или дробный литерал, как описано в 2.8.

Например, ."*", тип "*", 42 emit и char * emit четыре различных способа вывода одной звездочки.

2.11 Логические выражения или флаги

Fift не имеет отдельного типа значений для представления логических значений. Вместо этого любое ненулевое *целое число* может быть использовано для представления истины (где -1 является стандартным представлением), в то время как *нулевое целое число* представляет ложь. Примитивы сравнения обычно возвращают -1, чтобы указать на успех, и 0 в противном случае.

Константы true и false могут быть использованы для отправки этих специальных целых чисел в стек:

- true (– -1), выталкивает -1 в стек.

- false (– 0), вталкивает 0 в стек.

Если логические значения являются стандартными (либо 0, либо −1), ими можно манипулировать с помощью побитовых логических операций и, или, xor, нет (перечислено в 2.4). В противном случае их необходимо сначала свести к стандартной форме с использованием 0<>:

- 0<> (*x* – *x* ≠ 0), толкает −1 , если *целое число x* не равно нулю, в противном случае 0.

2.12 Целочисленные операции сравнения

Для получения логических значений можно использовать несколько целочисленных операций сравнения:

- < (*x y* – ?), проверяет, *< ли x y* (т.е. толкает −1, если *x* < *y*, 0 в противном случае).
- >, =, <>, <=, >= (*x y* – ?), сравните *x* и *y* и нажмите −1 или 0 в зависимости от результата сравнения.
- 0< (*x* – ?), проверяет, *< ли x* 0 (т.е. толкает −1, если *x* отрицательный, 0 в противном случае). Эквивалентно 0 <.
- 0>, 0=, 0<>, 0<=, 0>= (*x* – ?), сравните *x* с нулем.
- cmp (*x y* – *z*), толкает 1, если *x* > *y*, −1, если *x* < *y*, и 0, если *x* = *y*.
- sgn (*x* – *y*), толкает 1, если *x* > 0, −1, если *x* < 0, и 0, если *x* = 0. Эквивалентно 0 cmp.

Пример:

2 3 < .

печатает «-1 ок», потому что 2 меньше 3.

Более запутанный пример:

```
{ "true " "false " rot 0= 1+ pick type 2drop } : ?.
2 3 < ?. 2 3 = ?. 2 3 > ?.
```

печатает "true false false ok".

2.13 Операции сравнения строк

Строки можно лексикографически сравнить с помощью следующих слов:

- $= ($S\ S'$ – ?), возвращает -1, если строки S и S' равны, 0 в противном случае.
- $cmp ($S\ S' - x$), возвращает 0, если строки S и S' равны, -1, если S лексикографически меньше S', и 1, если S лексикографически больше S'.

2.14 Именованные и безымянные переменные

В дополнение к константам, представленным в 2.7, Fift поддерживает *переменные*, которые являются более эффективным способом представления изменяемых значений. Например, последние два фрагмента кода 2.7 могли быть написаны с помощью переменных вместо констант следующим образом:

```
переменная x переменная y
{ dup x ! dup * y ! } : setxy
3 setxy  x @ . y @ . x @ y @ + .
7 setxy  x @ . y @ . x @ y @ + .
{ ." ( " x @ . ", " y @ . ") " } : showxy
3    setxy showxy
```

производит тот же выход, что и раньше:

```
3 9 12 ok
```

```
7 49 56 ок
( 3 , 9 ) ок
```

Фразы variable x создает новый *Box*, то есть место в памяти, которое можно использовать для хранения ровно одного значения любого типа, поддерживаемого Fift, и определяет x как константу, равную этому *Box*:

- variable (–), сканирует пустое имя слова S из оставшейся части входных данных, выделяет пустое *Box* и определяет новое обычное слово S как константу, которая будет отправлять новый *Box* при вызове. Эквивалентен hole constant.
- hole (– p), создает новый *Box* p , который не содержит никакого значения. Эквивалентно null box.
- box (x – p), создает новый *Box* , содержащий указанное значение x. Эквивалентно hole tuck !.

Значение, хранящееся в настоящее время в *Box* , может быть извлечено с помощью слова @ (произносится «fetch») и изменено с помощью слова ! (произносится «store»):

- @ (p – x), извлекает значение, хранящееся в настоящее время в *Box* p.
- ! (x p –), сохраняет новое значение x в *Box* p.

Существует несколько вспомогательных слов, которые могут изменять текущее значение более сложным образом:

- +! (x p –), увеличивает целое значение, хранящееся в *Box* p , на *целое число* x. Эквивалентно tuck @ + swap !.
- 1+! (p –), увеличивает целое значение, хранящееся в *Box* p , на единицу. Эквивалентно 1 swap +!.
- 0! (p –), сохраняет *целое число* 0 в *Box* p. Эквивалентно 0 swap !.

Таким образом, мы можем реализовать простой счетчик:

```
variable counter
{ counter 0! } : reset-counter
{ counter @ 1+ dup counter ! } : next-counter
reset-counter next-counter . next-counter . next-counter .
reset-counter next-counter .
```

производит

```
1 2 3 ок
1 ок
```

После того, как эти определения будут введены в действие, мы можем даже забыть определение счетчика с помощью фразы forget counter. Тогда единственный способ получить доступ к значению этой переменной - с помощью reset-counter и next-counter.

Переменные обычно создаются variable без значения, или, скорее, со значением *Null*. Если кто-то хочет создать инициализированные переменные, он может использовать фразу box constant:

```
17 box constant x
x 1+! x @ .
```

печатает "18 ок". Можно даже определить специальное определяющее слово для инициализированных переменных, если они нужны часто:

```
{ box constant } : init-variable
17 init-variable x
"test" init-variable y
x 1+! x @ . y @ type
```

печатает "18 тест ок".

Переменные пока имеют только один недостаток по сравнению с константами: доступ к их текущим значениям приходится с помощью вспомогательного слова @. Конечно, можно смягчить это, определив слово «getter» и «setter» для переменной и используя эти слова для написания более привлекательного кода:

```
variable x-box
{ x-box @ } : x
{ x-box ! } : x!
{ x x * 5 x * + 6 + } : f(x)
{ ." ( " x . ", " f(x) . ") " } : .xy
3 x! .xy 5 x! .xy
```

печатает "(3 , 30) (5 , 56) ok", которые являются точками $(x,f(x))$ на графике $f(x) = x^2 + 5x + 6$ с $x = 3$ и $x = 5$.

Опять же, если мы хотим определить «getters» для всех наших переменных, мы можем сначала определить определяющее слово, как описано в 4.8, и использовать это слово для определения как getter, так и setter одновременно:

```
{ hole dup 1 ' @ does create 1 ' ! does create } : variable-set
variable-set x x!
variable-set y y!
{ ." x=" x . ." y=" y . ." x*y=" x y * . cr } : show
{ y 1+ y! } : up
{ x 1+ x! } : right
{ x y x! y! } : reflect
2 x! 5 y! show up show right show up show reflect show
```

производит

```
x=2 y=5 x*y=10
x=2 y=6 x*y=12
x=3 y=6 x*y=18
x=3 y=7 x*y=21
x=7 y=3 x*y=21
```

2.15 Кортежи и массивы

Fift также поддерживает *Tuples,* т.е. неизменяемые упорядоченные коллекции произвольных значений типов значений стека (см. 2.1). Когда *Tuple* t состоит из значений x_1, ..., x_n (в указанном порядке), мы записываем $t = (x_1,...,x_n)$. Число n называется *длиной Tuple* t; оно также обозначается |t|. Кортежи длины два также называют *парами*, кортежи длиной три являются *тройными*.

- tuple (x_1 ... x_n n – t), создает новый *кортеж* $t := (x_1,...,x_n)$ из $n \geq 0$ самых верхних значений стека.
- pair (x y – t), создает новую пару $t = (x,y)$. Эквивалентно 2 tuple.
- triple (x y z – t), создает новый тройной $t = (x,y,z)$. Эквивалентно 3 tuple.
- | (– t), создает пустой *Tuple* $t = ()$. Эквивалентно 0 tuple.
- , (t x – t'), добавляет x к концу *Tuple* t и возвращает полученный *Tuple* t'.
- .dump (x –), сбрасывает самую верхнюю запись стека так же, как .s сбрасывает все элементы стека.

Например, и то, и другое

| 2, 3, 9 , .dump

и

2 3 9 triple .dump

построить и распечатать тройной $(2,3,9)$:

[2 3 9] ок

Обратите внимание, что компоненты Tuple не обязательно имеют один и тот же тип и что компонент Tuple также может быть кортежем:

```
1 2 3 triple 4 5 6 triple 7 8 9 triple triple constant Matrix
Matrix .dump cr
| 1 “one” pair, 2 “two” pair, 3 “three” pair, .dump
```

производит

```
[ [ 1 2 3 ] [ 4 5 6 ] [ 7 8 9 ] ]
[ [ 1 "one" ] [ 2 "two" ] [ 3 "three" ] ] ок
```

После того, как кортеж был построен, мы можем извлечь любой из его компонентов или полностью распаковать *кортеж* в стек:

- untuple ($t\ n\ -\ x_1\ \ldots\ x_n$), возвращает все компоненты кортежа $t = (x_1,\ldots,x_n)$,но только в том случае, если его длина равна n. В противном случае возникает исключение.
- unpair ($t - x\ y$), распаковывает пару $t = (x,y)$. Эквивалентно 2 untuple.
- untriple ($t - x\ y\ z$), распаковывает тройной $t = (x,y,z)$. Эквивалентно 3 untuple.
- explode ($t\ -\ x_1\ \ldots\ x_n\ n$), распаковывает *кортеж* $t = (x_1,\ldots,x_n)$ неизвестной длины n и возвращает эту длину.
- count ($t - n$), возвращает длину n = |t| Кортеж t.
- tuple? ($t\ -\ ?$), проверяет, является ли t кортежем, и возвращает -1 или 0 соответственно.
- [] ($t\ i - x$), возвращает $(i + 1)$-ый компонент t_{i+1} *кортежа* t, где $0 \leq i <$ |t| .
- first ($t - x$), возвращает первый компонент кортежа. Эквивалентно 0 [].

- second ($t - x$), возвращает второй компонент кортежа. Эквивалент 1 [].
- third ($t - x$), возвращает третий компонент кортежа. Эквивалент 2 [].

Например, мы можем получить доступ к отдельным элементам и строкам матрицы:

```
1 2 3 triple 4 5 6 triple  7 8 9 triple triple constant Matrix
Matrix .dump cr
Matrix 1 [] 2 [] . cr
Matrix triple .dump cr
```

производит

```
[ [ 1 2 3 ] [ 4 5 6 ] [ 7 8 9 ] ]
6
[ 7 8 9 ]
```

Обратите внимание, что *кортежи* чем-то похожи на массивы других языков программирования, но неизменны: мы не можем изменить один отдельный компонент *кортежа*. Если мы все еще хотим создать что-то вроде массива, нам нужен *кортеж коробок* (*Tuple of Boxes*) (ср. 2.14):

- allot ($n - t$), создает кортеж, который состоит из n новых пустых *Boxes.* Эквивалент | { hole , } rot times*.*

Например,

```
10 allot constant A
| 3 box , 1 box , 4 box , 1 box , 5 box , 9 box , constant B
{ over @ over @ swap rot ! swap ! } : swap-values-of
{ B swap [] } : B[]
{ B[] swap B[] swap-values-of } : swap-B
{ B[] @ . } : . B[]
0 1 swap-B  1 3 swap-B  0 2 swap-B
0 . B[] 1 . B[] 2 . B[] 3 . B[]
```

создает неинициализированный массив A длиной 10, инициализированный массив B длины 6, а затем заменяет некоторые элементы B и печатает первые четыре элемента результирующего B:

4 1 1 3 ок

2.16 Списки

Списки в стиле Lisp также могут быть представлены в Fift. Прежде всего, для манипулирования значениями типа *Null*, используемых для представления пустого списка (не путать с пустым *Tuple*) введены два специальных слова:

- null (– ⊥), отправляет единственное значение ⊥ типа *Null*, которое также используется для представления пустого списка.
- null? (x – ?), проверяет, *ли* x является значением *Null*. Также может использоваться для проверки пустости списка.

После этого cons и uncons определяются как псевдонимы для pair и unpair:

- cons (h t – l), строит список из его головы (первого элемента) h и его хвоста (списка, состоящего из всех оставшихся элементов) t. Эквивалентно pair.
- uncons (l – h t), разлагает непустой список на голову и хвост. Эквивалент unpair.
- car (l – h), возвращает заголовок списка. Эквивалентно first.
- cdr (l – t), возвращает хвост списка. Эквивалентно second.
- cadr (l – h'), возвращает второй элемент списка. Эквивалент cdr car.
- list (x_1 ... x_n n – l), строит список l длины n с элементами x_1, ..., x_n, в указанном порядке. Эквивалентно null ' cons rot times.

- .l (l –), печатает список в стиле Lisp l.

Например

```
2 3 9  3 tuple .dump cr
2 3 9  3 list dup .dump space dup .l cr
"test" swap cons .l cr
```

производит

```
[ 2 3 9 ]
[ 2 [ 3 [ 9 (null) ] ] ] (2 3 9)
("test" 2 3 9)
```

Обратите внимание, что трехэлементный список $(2\ \ 3\ \ 9)$ отличается от тройного $(2,3,9)$.

2.17 Атомы

Atom — это простая сущность, однозначно идентифицируемая по своему имени. Атомы можно использовать для представления идентификаторов, меток, имен операций, тегов и маркеров стека. Fift предлагает следующие слова для манипулирования атомами:

- (atom) ($S\ x$ – a -1 или 0), возвращает единственный *Atom* a с именем, заданным строкой S. Если такого атома еще нет, либо создает его (если *Integer* x не равно нулю), либо возвращает один ноль, чтобы указать на сбой (если x равен нулю).
- atom (S – a), возвращает единственный *Atom* a с именем S, создавая такой атом при необходимости. Эквивалентно true (atom) drop.
- '⟨word⟩ (– a), вводит литерал *Атома* , равный единственному *Атому* с именем, равным ⟨word⟩. Эквивалент "⟨word⟩ " atom.
- anon (– a), создает новый уникальный анонимный *Atom*.
- atom? (u – ?), проверяет, *ли* u является *атомом*.

- eq? (*u v* – ?), проверяет, *ли u* и *v* равны *Integers, Atoms или Nulls.* Если они не равны или относятся к разным типам или не относятся к одному из перечисленных типов, возвращает ноль.

Например

'+ 2 '* 3 4 3 list 3 list .l

создает и печатает список

(+ 2 (* 3 4))

который является Lisp-style представлением арифметического выражения $2+3\cdot4$. Интерпретатор для таких выражений может использовать eq? для проверки знака операции (см. 3.5 для объяснения рекурсивных функций в Fift):

```
variable  'eval
{ 'eval @ execute } : eval
{ dup tuple? {
      uncons uncons uncons
      null? not abort"three-element list expected"
      swap eval swap eval rot
      dup '+ eq? { drop + } {
         dup '- eq? { drop - } {
            '* eq? not abort"unknown operation" *
         } cond
      } cond
   } if
} 'eval !
'+ 2 '* 3 4 3 list 3 list dup .l cr eval . cr
```

напечатается как

```
(+ 2 (* 3 4))
14
```

Если мы загрузим Файл Lisp.fif для включения синтаксиса списка в стиле Лисп, мы сможем ввести

"Lisp.fif" include

```
( '+ 2 ( '* 3 4 ) dup .l cr eval . cr
```

с тем же результатом, что и раньше. Слово (, определенное в Lisp.fif, использует анонимный *Atom,* созданный anon, чтобы отметить текущую позицию стека, а затем) строит список из нескольких верхних записей стека, сканируя стек до тех пор, пока не будет найден анонимный *маркер Atom*:

```
variable ')
{ ") without  (" abort } ') !
{ ') @ execute } : )
{ null { -rot 2dup eq? not } { swap rot cons } while 2drop
} : list-until-marker
{ anon dup ') @ 2 { ') ! list-until-marker } делает ') ! } : (
```

2.18 Аргументы командной строки в режиме сценария

Интерпретатор Fift можно вызвать в *режиме script mode*, передав -s в качестве параметра командной строки. В этом режиме все дополнительные аргументы командной строки не проверяются на наличие параметров командной строки Fift запуска. Вместо этого следующий аргумент после -s используется в качестве имени файла исходного файла Fift, а все остальные аргументы командной строки передаются в Fift программу с помощью специальных слов $*n* и $#:

- $# (– x), передает общее количество аргументов командной строки, переданных Fift программе.
- $*n* (– S), передает n-й аргумент командной строки в виде *строки* S. Например, $0 передает имя выполняемого сценария, $1 — первый аргумент командной строки и так далее.
- $() ($x$ – S), передает аргумент командной строки x-й аналогично $*n*, но с *целым числом x,* взятым из стека.

Кроме того, если самая первая строка исходного файла Fift начинается с двух символов «#!», эта строка игнорируется. Таким образом, простые Fift скрипты могут быть написаны в системе *ix. Например, если

```
#!/usr/bin/fift -s
```

```
{ ."usage: " $0 type ." <num1> <num2>" cr
    ." Computes the product of two integers." cr 1 halt } : usage
{ ' usage if } : ?usage
$# 2 <> ?usage
$1 (number) 1- ?usage
$2 (number) 1- ?usage
* . cr
```

сохраняется в файл cmdline.fif в текущем каталоге, и устанавливается его бит выполнения (например, chmod 755 cmdline.fif), затем его можно вызвать из оболочки или любой другой программы, при условии, что интерпретатор Fift установлен как /usr/bin/fift, а его стандартная библиотека Fift.fif установлена как /usr/lib/fift/Fift.fif:

```
$ ./cmdline.fif 12  -5
```

напечатает

```
-60
```

при вызове из оболочки *ix, такой как оболочка Bourne-again (Bash).

3 Блоки, циклы и условные обозначения

Подобно арифметическим операциям, поток выполнения в Fift управляется примитивами на основе стека. Это приводит к инверсии, типичной для обратной польской нотации и арифметики на основе стека: сначала в стек помещается блок, представляющий условную ветвь или тело цикла, а затем вызывает условный или итерированный примитив выполнения. В этом отношении Fift больше похож на PostScript, чем на Forth.

3.1 Определение и выполнение блоков

Блок обычно определяется с помощью специальных слов "{" и "}". Грубо говоря, все слова, перечисленные между { и } составляют тело нового блока, который выталкивается в стек как значение типа *WordDef*. Блок может быть сохранен как определение нового Fift слова с помощью определяющего слова ":", как описано в 2.6, или выполнен посредством слова execute:

17 { 2 * } execute .

печатает "34 ок", будучи по существу эквивалентным "17 2 * .". Немного более запутанный пример:

{ 2 * } 17 over execute swap execute .

применяет «анонимную функцию» $x \mapsto 2x$ дважды к 17 и выводит результат $2 \cdot (2 \cdot 17) = 68$. Таким образом, блок является маркером выполнения, который может быть продублирован, сохранен в константе, использован для определения нового слова или выполнен.

Слово ' восстанавливает текущее определение слова. А именно, конструкция ' ⟨*word-name*⟩ отправляет маркер выполнения, эквивалентный текущему определению слова ⟨*word-name*⟩. Например

' dup execute

эквивалентно dup, и

' dup : duplicate

определяет duplicate как синоним (текущее определение) dup.

Кроме того, мы можем продублировать блок, чтобы определить два новых слова с тем же определением:

{ dup * }
dup : square : **2
определяет как square, так и **2 как эквивалентные dup *.

3.2 Условное исполнение блоков

Условное исполнение блоков достигается с помощью слов if, ifnot и cond:

- if (*x e* –), выполняет *e* (который должен быть маркером выполнения, т.е. *WordDef*), [5] но только если *целое число x* не равно нулю.
- ifnot (*x e* –), выполняет маркер выполнения *e*, но только если *целое число x* равно нулю.

[5] *WordDef* является более общим, чем *WordList*. Например, определение примитива + является *WordDef*, но не *WordList*, потому что + не определяется в терминах других Fift слов.

- cond (x e e' –), если *целое число* x не равно нулю, выполняет e, в противном случае выполняет e'.

Например, последний пример в 2.12 можно более удобно переписать как

```
{ { ." true " } { ." false " } cond } : ?.
2 3 < ?. 2 3 = ?. 2 3 > ?.
```

по-прежнему приводит к "true false false ok".

Обратите внимание, что блоки могут быть произвольно вложенными, как уже показано в предыдущем примере. Можно написать, например,

```
{ ?dup
   { 0<
      { ." negative " }
      { ." positive " }
      cond
   }
   { ." zero " }
   cond
} : chksign

-17 chksign
```

для получения "negative ok", потому что -17 является отрицательным.

3.3 Простые петли

Простейшие циклы реализуются через times:

- times (e n –), выполняет e ровно n раз, если $n \geq 0$. Если n отрицательно, создается исключение.

Например

```
1 { 10 * } 70 times .
```

вычисляет и печатает 10^{70}.

Мы можем использовать этот тип цикла для реализации простой факториальной функции:

```
{ 0 1 rot { swap 1+ tuck * } swap times nip } : fact
5 fact .
```

печатает «120 ок», потому что $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$.

Этот цикл можно модифицировать для вычисления чисел Фибоначчи:

```
{ 0 1 rot { tuck + } swap times nip } : fibo
6 fibo .
```

вычисляет шестое число Фибоначчи $F_6 = 13$.

3.4 Циклы с условием выхода

Более сложные петли могут быть созданы с помощью до тех пор, пока:

- until (e –), не выполнит e, затем удалит целое число верхней части стека и проверит, равно ли оно нулю. Если это так, то начинается новая итерация цикла путем выполнения e. В противном случае выходит из цикла.

- while (e e' –), выполняет e, затем удаляет и проверяет целое число верхней части стека. Если он равен нулю, выходит из цикла. В противном случае выполняется e', а затем начинается новая итерация цикла, выполняя e и проверяя условие выхода.

Например, мы можем вычислить первые два числа Фибоначчи больше 1000:

```
{ 1 0 rot { -rot over + swap rot 2dup >= } until drop
} : fib-gtr
1000 fib-gtr . .
```

напечатает "1597 2584 ok".

Мы можем использовать это слово для вычисления первых 70 десятичных цифр золотого сечения $\varphi = (1 + \sqrt{5})/2 \approx 1.61803$:

```
1 { 10 * } 70 times dup fib-gtr */ .
```

печатает "$161803\ldots\ 2604$ ok».

3.5 Рекурсия

Обратите внимание, что всякий раз, когда слово упоминается в блоке { ...}, текущее определение (время компиляции) включается в создаваемый *WordList*. Таким образом, мы можем ссылаться на предыдущее определение слова, определяя новую его версию:

```
{ + . } : print-sum
{ ." number " . } : .
{ 1+ . } : print-next
2 . 3 . 2 3 print-sum 7 print-next
```

производит "number 2 number 3 5 number 8 ok". Обратите внимание, что print-sum продолжает использовать исходное определение ".", но print-next уже использует измененное ".".

Эта функция может быть удобной в некоторых случаях, но она мешает нам вводить рекурсивные определения самым простым способом. Например, классическое рекурсивное определение факториала

```
{ ?dup { dup 1- fact * } { 1 } cond } : fact
```

не удастся скомпилировать, потому что факт оказывается неопределенным словом при компиляции определения.

Простой способ обойти это препятствие — использовать слово @' (ср. 4.6), которое ищет текущее определение следующего слова во время выполнения, а затем выполняет его, аналогично тому, что мы уже делали в 2.7:

```
{ ?dup { dup 1- @' fact * } { 1 } cond } : fact
5 fact .
```

производит "120 ok", как и ожидалось.

Однако это решение довольно неэффективно, поскольку оно использует поиск по словарю каждый раз, когда fact выполняется рекурсивно. Мы можем избежать этого поиска по словарю, используя переменные (см. 2.14 и 2.7):

```
variable  'fact
{ 'fact @ execute } : fact
{ ?dup { dup 1- fact * } { 1 } cond }  ‘fact !
5 fact .
```

Это несколько более длинное определение факториала позволяет избежать поиска в словаре во время выполнения, вводя специальную переменную 'fact для удержания окончательного определения факториала.[6] Затем fact определяется для выполнения любого *WordDef*, который в настоящее время хранится в 'fact, и как только тело рекурсивного определения факториала построено, оно сохраняется в этой переменной с помощью фразы 'fact!, которая заменяет более привычную фразу : fact.

Мы могли бы переписать приведенное выше определение, используя специальные слова «getter» и «setter» для векторной переменной 'fact, как мы сделали для переменных в 2.14:

```
variable 'fact
{ 'fact @ execute } : fact
{ 'fact ! } : :fact
forget 'fact
{ ?dup { dup 1- fact * } { 1 } cond } :fact
5 fact .
```

Если нам нужно ввести много рекурсивных и взаимно рекурсивных определений, мы могли бы сначала ввести пользовательское определяющее слово (ср. 4.8) для одновременного определения слов «getter» и «setter» для анонимных векторных переменных, аналогично тому, что мы сделали в 2.14:

```
{ hole dup 1 { @ execute } does create 1 ' ! does create
```

[6] Переменные, содержащие *WordDef* для последующего выполнения, называются *векторными переменными*. Процесс замены факта на 'fact @ execute, где 'fact является векторной переменной, называется *векторизацией*.

```
} : vector-set
vector-set fact :fact
{ ?dup { dup 1- fact * } { 1 } cond } :fact
5 fact .
```

Первые три строки этого фрагмента определяют fact и :fact по существу так же, как они были определены в первых четырех строках предыдущего фрагмента.

Если мы хотим сделать факт неизменным в будущем, мы можем добавить строку forget:fact после того, как определение факториала будет завершено:

```
{ hole dup 1 { @ execute } does create 1 ' ! does create
} : vector-set
vector-set fact :fact
{ ?dup { dup 1- fact * } { 1 } cond } :fact
forget :fact
5 fact .
```

В качестве альтернативы мы можем изменить определение vector-set таким образом, чтобы :fact забыл себя после его вызова:

```
{ hole dup 1 { @ execute } does create
  bl word tuck 2 { (forget) ! } does swap 0 (create)
} : vector-set -once
vector-set -once fact :fact
{ ?dup { dup 1- fact * } { 1 } cond } :fact
5 fact .
```

Однако некоторые векторные переменные должны быть изменены более одного раза, например, чтобы изменить поведение слова сравнения меньше в алгоритме сортировки слияния:

```
{ hole dup 1 { @ execute } does create 1 ' ! does create
} : vector-set
```

```
vector-set sort :sort
vector-set merge :merge
vector-set less :less
{ null null rot
   { dup null? not }
   { uncons swap rot cons -rot } while drop
} : split
{ dup null? { drop } {
       over null? { nip } {
          over car over car less ' swap if
          uncons rot merge cons
       } cond
   } cond
} :merge
{ dup null? {
       dup cdr null? {
          split sort swap sort merge
       } ifnot
   } ifnot
} :sort
forget :merge
forget :sort
// set 'less' to compare numbers, sort a list of numbers
' < :less
3 1 4 1 5 9 2 6 5  9 list
dup .l cr sort .l cr
// set 'less' to compare strings, sort a list of strings
{ $cmp 0< } :less
"once" "upon" "a" "time" "there" "lived" "a" "kitten"  8 list
dup .l cr sort .l cr
```

получение следующих выходных данных:

(3 1 4 1 5 9 2 6 5)
(1 1 2 3 4 5 5 6 9)
(«once» «upon» «a» «time» «there» «lived» «a» «kitten»)
(«a» «a» «kitten» «lived» «once» «there» «time» «upon»)

3.6 Создание исключений

Два встроенных слова используются для создания исключений:

- abort (S –), создает исключение с сообщением об ошибке, взятым из *строки* S.
- abort"⟨$message$⟩" (x –), выдает исключение с сообщением об ошибке ⟨$message$⟩, если x является ненулевым целым числом.

Исключение, создаваемое этими словами, представлено исключением C++ fift::IntError со значением, равным указанной строке. Обычно он обрабатывается в самом интерпретаторе Fift путем прерывания всего выполнения до верхнего уровня и печати сообщения с именем интерпретируемого исходного файла, номером строки, интерпретируемым в данный момент словом и указанным сообщением об ошибке. Например:

```
{ dup 0= abort"Division by zero" / } : safe/
5 0 safe/ .
```

печатает "safe/: Division by zero", без обычного "ok". Стек очищается в процессе.

Кстати, когда интерпретатор Fift сталкивается с неизвестным словом, которое не может быть проанализировано как целочисленный литерал, возникает исключение с сообщением «-?», с эффектом, указанным выше, включая очистку стека.

4 Словарь, интерпретатор и компилятор

В этой главе мы представим несколько конкретных Fift слов для работы со словарем и управления компилятором. «Компилятор» является частью интерпретатора Fift, который строит списки ссылок на слова (представленные значениями стека *WordList*) из имен слов; он активируется примитивом «{», используемым для определения блоков, как описано в 2.6 и 3.1.

Большая часть информации, включенной в эту главу, является довольно сложной и может быть пропущена во время первого чтения. Тем не менее, методы, описанные здесь, в значительной степени используются ассемблером Fift, используемым для компиляции кода TVM. Поэтому этот раздел незаменим, если кто-то хочет понять текущую реализацию Fift ассемблера.

4.1 Состояние переводчика Fift

Состояние интерпретатора Fift управляется внутренней целочисленной переменной state, которая в настоящее время недоступна из самого Fift. Когда состояние равно нулю, все слова, проанализированные из входных данных (т.е. исходный файл Fift или стандартный ввод в интерактивном режиме), просматриваются в словаре и сразу же выполняются впоследствии. Когда state положительное, слова, найденные в словаре, не выполняются. Вместо этого они (а точнее ссылки на их текущие определения) *компилируются*, т.е. добавляются в конец создаваемого *WordList.*

Как правило, state равно количеству открытых в данный момент блоков. Например, после интерпретации "{ 0= { ." zero"" переменная state будет равна двум, так как есть два вложенных блока. Создаваемый *Wordlist* хранится в верхней части стека.

Примитив "{" просто помещает новый пустой *WordList* в стек и увеличивает state на единицу. Примитив "}" выдает исключение, если state уже равно нулю; в противном случае он уменьшает состояние на единицу и оставляет результирующее *WordList* в стеке, представляющий только

что построенный блок. [7] После этого, если результирующее значение состояния не равно нулю, новый блок компилируется как литерал (безымянная константа) в охватывающий блок.

4.2 Активные и обычные слова

Все словарные слова имеют специальный флаг, указывающий, являются ли они *активными* словами или *обычными словами* . По умолчанию все слова являются обычными. В частности, все слова, определяемые с помощью «:» и constant, являются обычными.

Когда переводчик Fift находит определение слова в словаре, он проверяет, является ли оно обычным словом. Если это так, то текущее определение слова либо выполняется (если state равно нулю), либо «компилируется» (если state больше нуля), как описано в 4.1.

С другой стороны, если слово активно, то оно всегда исполняется, даже если состояние положительное. Ожидается, что активное слово оставит некоторые значения x_1 ... x_n n e в стеке, где $n \geq 0$ — целое число, x_1 … x_n — это n значений произвольных типов, а e — маркер выполнения (значение типа *WordDef*). После этого интерпретатор выполняет различные действия в зависимости от состояния: если состояние равно нулю, то n отбрасывается и e выполняется, как если бы была найдена фраза nip execute. Если состояние не равно нулю, то эта коллекция «компилируется» в текущем *WordList* (расположенном непосредственно под x_1 в стеке) так же, как если бы был вызван примитив (compile). Эта компиляция сводится к добавлению некоторого кода в конец текущего *WordList,* который будет отправлять x_1, ..., x_n в стек при вызове, а затем добавляет ссылку на e

[7] Слово } также преобразует этот *WordList* в *WordDef*, который имеет другой тег типа и, следовательно, является другим Fift значением, даже если в реализации C++ используется тот же базовый объект C++.

(представляющую отложенное выполнение *e*). Если *e* равно специальному значению 'nop, представляющему маркер выполнения, который ничего не делает при выполнении, то этот последний шаг опускается.

4.3 Компиляция литералов

Когда интерпретатор Fift сталкивается со словом, отсутствующим в словаре, он вызывает примитив (number), чтобы попытаться проанализировать его как целое число или дробный литерал. Если эта попытка увенчалась успехом, то выдвигается специальное значение 'nop, и интерпретация протекает так же, как если бы активное слово были встречены. Другими словами, если состояние равно нулю, то литерал просто остается в стеке; в противном случае (compile) вызывается для изменения текущего *Списка WordList* таким образом, чтобы при выполнении он отправлял литерал.

4.4 Определение новых активных слов

Новые активные слова определяются аналогично новым обычным словам, но с использованием «::» вместо «:». Например

```
{ bl word 1 ' type } :: say
```

определяет активное слово say, которое сканирует следующее пустое слово после себя и компилирует его как литерал вместе со ссылкой на текущее определение type в текущий *WordList* (если state не равно нулю, т. е. если интерпретатор Fift компилирует блок). При вызове это дополнение к блоку будет помещать сохраненную строку в стек и выполнять type, таким образом, печатая следующее слово после say. С другой стороны, если state равно нулю, то эти два действия выполняются интерпретатором Fift немедленно. Таким образом,

```
1 2 say hello + .
```

распечатает "hello3 ok", в то время как

```
{ 2 say hello + . } : test
1 test 4 test
```

распечатает "hello3 hello6 ok".

Конечно, блок может быть использован для представления требуемого действия вместо ' type. Например, если нам нужна версия say, которая печатает пробел после сохраненного слова, мы можем написать

```
{ bl word 1 { type space } } :: say
{ 2 say hello + . } : test
1 test 4 test
```

для получения "hello 3 hello 6 ok".

Кстати, слова " (вводя строку литерала) и ." (печать строкового литерала) может быть определена следующим образом:

```
{ char " word 1 'nop } ::_ "
{ char " word 1 ' type } ::_ ."
```

Новое определяющее слово "::_" определяет активное *префиксное* слово, т.е. активное слово, которое впоследствии не требует пробела.

4.5 Определение слов и манипуляции со словарем

Определяющие слова — это слова, которые определяют новые слова в словаре Fift. Например, ":", "::_" и constant являются определяющими словами. Все эти определяющие слова могли быть определены с помощью примитива (create); фактически, пользователь может ввести пользовательские определяющие слова, если это необходимо. Перечислим некоторые определяющие слова и слова для работы со словарем:

- create ⟨*word-name*⟩ (*e* –), определяет новое обычное слово с именем, равным следующему слову, отсканированному из ввода, используя *WordDef e* в качестве его определения. Если слово уже существует, оно молчаливо переопределяется.
- (create) (*e S x* –), создает новое слово с именем, равным *String*

S и определение равно *WordDef* e, используя флаги, передаваемые в *Integer* $0 \le x \le 3$. Если бит $+1$ установлен в x, создает активное слово; если бит $+2$ установлен в x, создается префиксное слово.

- : ⟨*word-name*⟩ (e –), определяет новое обычное слово ⟨*word-name*⟩ в словаре, используя *WordDef* e в качестве его определения. Если указанное слово уже присутствует в словаре, оно молчаливо переопределяется.

- forget ⟨*word-name*⟩ (–), забывает (удаляет из словаря) определение указанного слова.

- (forget) (S –), забывает слово с именем, указанным в *строке* S. Если слово не найдено, создается исключение.

- :_ ⟨*word-name*⟩ (e –), определяет новый обычный *префикс* слова ⟨*word-name*⟩, означающий, что пустой символ или символ конца строки не требуется Fift синтаксическому анализатору после имени слова. Во всем остальном он похож на ":".

- :: ⟨*word-name*⟩ (e –), определяет новое *активное* слово ⟨*word-name*⟩ в словаре, используя *WordDef* e в качестве его определения. Если указанное слово уже присутствует в словаре, оно молчаливо переопределяется.

- ::_ ⟨*word-name*⟩ (e –), определяет новый активный *префикс* слова ⟨*word-name*⟩, означающий, что пустой символ или символ конца строки не требуется Fift синтаксическому анализатору после имени слова. Во всем остальном он похож на "::".

- constant ⟨*word-name*⟩ (x –), определяет новое обычное слово ⟨*word-name*⟩, которое при вызове будет выталкивать заданное значение x.

- 2constant ⟨*word-name*⟩ (*x y* –), определяет новое обычное слово с именем ⟨*word-name*⟩, которое будет выталкивать заданные значения *x* и *y* (в указанном порядке) при вызове.

- =: ⟨*word-name*⟩ (*x* –), определяет новое обычное слово ⟨*word-name*⟩, которое будет выталкивать заданное значение *x* при вызове, аналогично константе, но работает внутри блоков и двоеточий определений.

- 2=: ⟨*word-name*⟩ (*x y* –), определяет новое обычное слово ⟨*word-name*⟩, которое будет перемещать заданные значения *x* и y (в указанном порядке) при вызове, аналогично 2constant, но работает внутри блоков и двоеточий определений.

Обратите внимание, что большинство из приведенных выше слов могли быть определены в терминах (create):

```
{ bl word 1 2 ' (create) } "::" 1 (create)
{ bl word 0 2 ' (create) } :: :
{ bl word 2 2 ' (create) } :: :_
{ bl word 3 2 ' (create) } :: ::_
{ bl word 0 (create) } : create
{ bl word (forget) } : forget
```

4.6 Поиск по словарю

Для поиска слов в словаре можно использовать следующие слова:

- ' ⟨*word-name*⟩ (– *e*), выдвигает определение слова hword-namei, восстановленное во время компиляции. Если указанное слово не найдено, возникает исключение. Обратите внимание, что ' ⟨*word-name*⟩ execute всегда эквивалентно ⟨*word-name*⟩ для обычных слов, но не для активных слов.

- nop (–), ничего не делает.

- 'nop (– *e*), передает определение nop по умолчанию — маркер выполнения, который ничего не делает при выполнении.
- find (*S* – *e* −1 или *e* 1 или 0), ищет *строку S* в словаре и возвращает ее определение как *WordDef e* , если найдено, за которым следует −1 для обычных слов или 1 для активных слов. В противном случае нажимает 0.
- (') ⟨*word-name*⟩ (– *e*), аналогично ', но возвращает определение указанного слова во время выполнения, выполняя поиск по словарю при каждом его вызове. Может использоваться для восстановления текущих значений констант внутри определений слов и других блоков с помощью фразы (') ⟨*word-name*⟩ execute.
- @' ⟨*word-name*⟩ (– *e*), аналогично ('), но восстанавливает определение указанного слова во время выполнения, выполняя поиск по словарю при каждом его вызове, а затем выполняет это определение. Может использоваться для восстановления текущих значений констант внутри определений слов и других блоков с помощью фразы @' ⟨*word-name*⟩, эквивалентной (') ⟨*word-name*⟩ execute, ср. 2.7.
- [compile] ⟨*word-name*⟩ (–), компилирует ⟨*word-name*⟩, как если бы это было обычное слово, даже если оно активно. По существу эквивалентно ' ⟨*word-name*⟩ execute.
- words (–), печатает названия всех слов, определенных в настоящее время в словаре.

4.7 Создание списков слов и управление ими

В стеке Fift списки ссылок на определения слов и литералы, которые будут использоваться в качестве блоков или определений слов, представлены значениями типа *WordList*. Некоторые слова для манипулирования *Wordlists* включают в себя:

- { (– l), активное слово, которое увеличивает внутреннюю переменную state на единицу и отправляет новый пустой *WordList* в стек.

- } (l – e), активное слово, которое преобразует *WordList* l в *WordDef* (маркер выполнения) e, тем самым делая невозможными все дальнейшие модификации l , и уменьшает внутреннюю переменную state на единицу и толкает целое число 1, за которым следует 'nop. Чистый эффект заключается в преобразовании построенного *WordList* в маркер выполнения и отправке этого маркера выполнения в стек либо немедленно, либо во время выполнения внешнего блока.

- ({) (– l), помещает пустой *WordList* в стек.

- (}) (l – e), преобразует *WordList* в маркер выполнения, делая невозможными все дальнейшие модификации.

- (compile) (l x_1 ... x_n n e – l'), расширяет *WordList* l так, чтобы он отправлял $0 \le n \le 255$ значений x_1, ..., x_n в стек и выполнял маркер выполнения e при вызове, где $0 \le n \le 255$ — *целое число*. Если e равно специальному значению 'nop, последний шаг опускается.

- does (x_1 ... x_n n e – e'), создает новый маркер выполнения e', который будет отправлять n значений x_1, ..., x_n в стек, а затем выполнять e. Это примерно эквивалентно комбинации ({), (compile) и (}).

4.8 Пользовательские определяющие слова

Слово does на самом деле определяется в терминах более простых слов:

```
{ swap ({) over 2+ -roll swap (compile) (}) } : does
```

Это особенно полезно для определения пользовательских определяющих слов. Например, constant и 2constant могут быть определены с помощью does и create:

```
{ 1 'nop does create } : constant
```

{ 2 'nop does create } : 2constant

Конечно, нетривиальные действия могут выполняться словами, определенными с помощью таких пользовательских определяющих слов. Например

{ 1 { type space } does create } : says
"hello" says hello
"unknown error" says error
{ hello error } : test
test

будет печатать «hello unknown error ok», потому что hello определяется с помощью пользовательского определяющего слова, говорит печатать «hello» при каждом вызове, и аналогично error печатает «unknown error» при вызове. Приведенные выше определения по существу эквивалентны

{ ." hello" } : hello
{ ."unknown error" } : error

Однако пользовательские определяющие слова могут выполнять более сложные действия при вызове и предварительно обрабатывать их аргументы во время компиляции. Например, сообщение может быть вычислено нетривиальным образом:

"Hello, " "world!" $+ says hw

определяет слово hw, которое печатает «Hello, world!» при вызове. Строка с этим сообщением вычисляется один раз во время компиляции (при вызове says), а не во время выполнения (при вызове hw).

5 Манипуляции с клетками

Мы обсудили основные Fift примитивах, не связанных с TVM или TON Blockchain до сих пор. Теперь мы обратимся к словам, специфичным для TON, используемым для манипулирования *Cells.*

5.1 Фрагменты литералов

Напомним, что ячейка (TVM) состоит максимум из 1023 бит данных и не более четырех ссылок на другие *Cells, Slice* — это представление части ячейки только для чтения, а *Builder* используется для создания новых *ячеек. Fift имеет специальные положения для определения литералов Slice* (т. е. безымянных констант), которые также могут быть преобразованы в *Cells* при необходимости.

Slice литералы вводятся с помощью активных префиксных слов x{ и b{:

- b{⟨*binary-data*⟩} (– *s*), создает *Slice* s , который не содержит ссылок и до 1023 бит данных, указанных в ⟨*binary-data*⟩, который должен быть строкой, состоящей только из символов '0' и '1'.

- x{⟨*hex-data*⟩} (– *s*), создает *Slice* s , который не содержит ссылок и до 1023 бит данных, указанных в ⟨*hex-data*⟩. Точнее, каждая шестнадцатеричная цифра из ⟨*hex-data*⟩ преобразуется в четыре двоичные цифры обычным способом. После этого, если последний символ ⟨*hex-data*⟩ является подчеркиванием _, то все конечные двоичные нули и двоичный непосредственно предшествующий им удаляются из результирующей двоичной строки (подробнее см. [4, 1.0]).

Таким образом, b{00011101} и x{1d} отправляют один и тот же *Slice,* состоящий из восьми битов данных и без ссылок. Аналогично, b{111010} и x{EA_} отправляют один и тот же *Slice,* состоящий из шести битов данных. Пустой *Slice* может быть представлен как b{} или x{}.

Если кто-то хочет определить *константу Slices с некоторыми* ссылками на ячейки, можно использовать следующие слова:

- |_ ($s\ s'\ -\ s''$), при наличии двух *Slices* s и s', создает новый *Slice* s'', который получается из s путем добавления новой ссылки на *ячейку,* содержащую s'.

- |+ ($s\ s' - s''$), объединяет два *фрагмента* s и s'. Это означает, что биты данных нового *Slice* s'' получаются путем объединения битов данных s и s', а список ссылок на ячейки s'' строится аналогичным образом путем объединения соответствующих списков для s и s'.

5.2 Строители примитивов

Следующие слова могут быть использованы для манипулирования *Builders,* которые впоследствии могут быть использованы для создания новых ячеек:

- <b (– b), создает новый пустой *Builder*.

- b> ($b - c$), преобразует *Builder* b в новую ячейку c , содержащую те же данные, что и b.

- i, ($b\ x\ y\ -\ b'$), добавляет двоичное представление большого порядкового числа знакового y-битного целого *числа* x к *Builder* b, где $0 \le y \le 257$. Если в b недостаточно места (т.е. если b уже содержит более $1023\ -\ y$ битов данных), или если целое *число* x не укладывается в *биты* y, возникает исключение.

- u, ($b\ x\ y\ -\ b'$), добавляет двоичное представление большого порядкового числа *в* знака y-битного целого *числа* x x без *Builder* b, где $0 \le y \le 256$. Если операция невозможна, создается исключение.

- ref, ($b\ c\ -\ b'$), добавляет к *Builder* b ссылку на Ячейку c. Если b уже содержит четыре ссылки, создается исключение.

- s, ($b\ s - b'$), добавляет биты данных и ссылки, взятые из *Slice* s в *Builder* b.

- sr, (b s – b'), создает новую *ячейку,* содержащую все данные и ссылки из *Slice* s, и добавляет ссылку на эту ячейку *в Builder* b. Эквивалентно <b swap s, b> ref,.

- $, ($b$ S – b'), добавляет *строку* S в *Builder* b. Строка интерпретируется как двоичная строка длиной $8n$, где n — количество байтов в представлении S в кодировке UTF-8.

- B, (b B – b'), добавляет *Bytes* B в *Builder* b.

- b+ (b b' – b''), объединяет двух *Builder* b и b'.

- bbits (b – x), возвращает количество битов данных, уже сохраненных в *Builder* b. Результатом x является *целое число* в диапазоне $0\ldots 1023$.

- brefs (b – x), возвращает количество ссылок, уже сохраненных в *Builder* b. Результатом x является *целое число* в диапазоне $0\ldots 4$.

- bbitrefs (b – x y), возвращает как количество битов данных x, так и количество ссылок y, уже сохраненных в *Builder* b.

- brembits (b – x), возвращает максимальное количество дополнительных битов данных, которые могут храниться в *Builder* b. Эквивалент bbits 1023 swap -.

- bremrefs (b – x), возвращает максимальное количество дополнительных ссылок на ячейки, которые могут храниться в *builder* b.

- brembitrefs (b – x y), возвращает как максимальное количество дополнительных битов данных $0 \le x \le 1023$, так и максимальное количество дополнительных ссылок на ячейки $0 \le y \le 4$, которые могут храниться в *Builder* b.

Полученный *Builder* может быть проверен с помощью неразрушающего примитива дампа стека .s или с помощью фразы b> <s csr.. Например:

{ <b x{4A} s, rot 16 u, swap 32 i, .s b> } : mkTest

17239 -1000000001 mkTest
<s csr.

Выходы

```
BC{000e4a4357c46535ff}
  ok
x{4A4357C46535FF}
  ok
```

Можно заметить, что .s сбрасывает внутреннее представление *Builder*, с двумя байтами тегов в начале (обычно равным количеству ссылок на ячейки, уже хранящихся в *Builder*, и в два раза большему количеству полных байтов, хранящихся в *Builder*, увеличенному на единицу, если присутствует неполный байт). На с другой стороны, csr. сбрасывает *Slice* (построенный из *ячейки* <s, ср. 5.3) в форме, аналогичной той, которая используется x{ для определения литералов Slice (ср. 5.1).

Кстати, слово mkTest, показанное выше (без .s в его определении), соответствует конструктору TL-B

```
test#4a first:uint16 second:int32 = Test;
```

и может использоваться для сериализации значений этого типа TL-B.

5.3 Срезовые примитивы

Следующие слова можно использовать для управления значениями типа *Slice*, который представляет доступное только для чтения представление части ячейки. Таким образом, данные, ранее сохраненные в ячейке , могут быть десериализованы путем предварительного преобразования ячейки в *фрагмент*, а затем поэтапного извлечения необходимых данных из этого *фрагмента*.

- <s (c – s) преобразует ячейку c в *фрагмент* s , содержащий те же данные. Обычно это знаменует собой начало десериализации клетки.
- s> (s –), создает исключение, если *фрагмент* s не пуст. Обычно он отмечает конец десериализации ячейки, проверяя, остались ли какие-либо необработанные биты данных или ссылки.

- i@ ($s\ x - y$), извлекает знаковое большое порядковое x-битное целое число из первых x бит Slice s. Если s содержит меньше x бит данных, создается исключение.
- i@+ ($s\ x - y\ s'$), извлекает знаковое большое порядковое x-битное целое число из первых x бит *Slice* s аналогично i@, но возвращает и оставшуюся часть s.
- i@? ($s\ x - y\ -1$ или 0), извлекает знаковое целое число из *Slice* аналогично i@, но впоследствии отправляет целое число -1 при успешном выполнении. Если в x *бит* осталось меньше s, нажимает целое число 0, чтобы указать на сбой.
- i@?+ ($s\ x - y\ s'\ -1$ или $s\ 0$), извлекает знаковое целое число из *Slice* s и вычисляет оставшуюся часть этого *Slice* аналогично i@+, но после этого нажимает -1 , чтобы указать на успех. При сбое нажимает на неизмененные *фрагменты* s и 0 , чтобы указать на сбой.
- u@, u@+, u@?, u@?+, аналоги i@, i@+, i@?, i@?+ для десериализации целых чисел без знака.
- B@ ($s\ x - B$), извлекает первые x байт (т.е. $8x$ бит) из *Slice* s и возвращает их в виде *Bytes* значения B. Если в s недостаточно битов данных, возникает исключение.
- B@+ ($s\ x - B\ s'$), аналогично B@, но возвращает и оставшуюся часть *Slice* s .
- B@? ($s\ x - B\ -1$ или 0), аналогично B@, но использует флаг для указания на сбой вместо создания исключения.
- B@?+ ($s\ x - B\ s'\ -1$ или $s\ 0$), аналогично B@+, но использует флаг для указания на сбой вместо создания исключения.
- $@, $@+, $@?, $@?+, аналоги B@, B@+, B@?, B@?+, возвращая результат в виде строки (UTF-8) вместо значения *Bytes*. Эти

примитивы не проверяют, является ли чтение байтовой последовательности допустимой строкой UTF-8.

- ref@ ($s - c$), извлекает первую ссылку из *Slice* s и возвращает *ячейку* c , на которую ссылается. Если ссылок не осталось, создается исключение.
- ref@+ ($s - s'\ c$), аналогично ref@, но возвращает и оставшуюся часть s.
- ref@? ($s - c\ -1$ или 0), аналогично ref@, но использует флаг для указания на сбой вместо того, чтобы выдавать исключение.
- ref@?+ ($s - s'\ c\ -1$ или $s\ 0$), аналогично ref@+, но использует флаг для указания на сбой вместо того, чтобы выдавать исключение.
- empty? ($s - ?$), проверяет, пуст ли *фрагмент* (т. е. не имеет битов данных и ссылок), и возвращает ли он соответственно -1 или 0.
- remaining ($s - x\ y$), возвращает как количество битов данных x, *так* и количество ссылок на ячейки y, оставшихся в *Slice* s.
- sbits ($s - x$), возвращает количество битов данных x , оставшихся в *Slice* s.
- srefs ($s - x$), возвращает количество ссылок на ячейки x, оставшихся в *Slice* s.
- sbitrefs ($s - x\ y$), возвращает как количество битов данных x, *так* и количество ссылок на ячейки y, оставшихся в *Slice* s. Эквивалентно remaining.
- $>s ($S - s$), преобразует *строку* S в *фрагмент*. Эквивалент <b swap $, b> <s.
- s>c ($s - c$), создает ячейку c непосредственно из *фрагмента* s. Эквивалент <b swap s, b>.

- csr. (*s* –), рекурсивно печатает *фрагмент s*. В первой строке биты данных *s* отображаются в шестнадцатеричной форме, встроенной в конструкцию x{...}, аналогичную той, которая используется для *литералов Slice* (ср. 5.1). На следующих строках ячейки, обозначаемые *s,* печатаются с большим отступом.

Например, значения теста типа TL-B, обсуждаемые в 5.2

test#4a first:uint16 second:int32 = Test;

могут быть десериализованы следующим образом:

```
{ <s 8 u@+ swap 0x4a <> abort"constructor tag mismatch"
      16 u@+ 32 i@+ s> } : unpackTest
x{4A4357C46535FF} s>c unpackTest swap . .
```

печатает "17239 -1000000001 ok", как и ожидалось.

Конечно, если нужно часто проверять теги конструктора, для этой цели можно определить вспомогательное слово:

```
{ dup remaining abort"references in constructor tag"
  tuck u@ -rot u@+ -rot <> abort"constructor tag mismatch"
} : tag?
{ <s x{4a} tag? 16 u@+ 32 i@+ s> } : unpackTest
x{4A4357C46535FF} s>c unpackTest swap . .
```

Мы можем сделать еще лучше с помощью активных префиксных слов (ср. 4.2 и 4.4):

```
{ dup remaining abort"references in constructor tag"
  dup 256 > abort"constructor tag too long"
  tuck u@ 2 { -rot u@+ -rot <> abort"constructor tag mismatch" }
} : (tagchk)
{ [compile] x{ 2drop (tagchk) } ::_ ?x{
{ [compile] b{ 2drop (tagchk) } ::_ ?b{
```

```
{ <s ?x{4a} 16 u@+ 32 i@+ s> } : unpackTest
x{4A4357C46535FF} s>c unpackTest swap . .
```

Более коротким, но менее эффективным решением было бы повторное использование ранее определенного тега?:

```
{ [compile] x{ drop ' tag? } ::_ ?x{
{ [compile] b{ drop ' tag? } ::_ ?b{
x{11EF55AA} ?x{11E} dup csr.
?b{110} csr.
```

сначала выводит "x{F55AA}", а затем выдает исключение с сообщением "несоответствие тега конструктора".

5.4 Хэш-операции ячейки

Есть несколько слов, которые действуют на клетки напрямую. Наиболее важный из них вычисляет (на основе sha256) хэш представления данной ячейки (ср. [4, 3.1]), который можно примерно описать как хэш sha256 битов данных ячейки, сцепленных с рекурсивно вычисленными хэшами ячеек, на которые ссылается эта ячейка:

- hashB ($c - B$), вычисляет хэш представления на основе sha256 *Ячейка c* (ср. [4, 3.1]), которая однозначно определяет c и всех его потомков (при условии отсутствия коллизий для sha256). Результат возвращается в виде значения *Bytes*, состоящего ровно из 32 байт.
- hashu ($c - x$) вычисляет хэш представления с на основе sha256 , как указано выше, но возвращает результат в виде 256-битного целого числа без знака с большим порядком байтов.
- shash ($s - B$) вычисляет хэш представления Slice на основе sha256 , сначала преобразуя его в ячейку. Эквивалент s>c hashB.

5.5 Операции с ячейками

Мешок *ячеек* представляет собой совокупность одной или нескольких клеток вместе со всеми их потомками. Обычно его можно сериализовать в последовательность байтов (представленную значением *Bytes* в Fift), а затем сохранить в файл или передать по сети. После этого он может быть десериализован для восстановления исходных клеток. TON Blockchain систематически представляет различные структуры данных (включая блоки TON Blockchain) в виде дерева ячеек в соответствии с определенной схемой TL-B (ср. [5], где эта схема подробно объясняется), а затем эти деревья клеток обычно импортируются в мешки ячеек и сериализуются в двоичные файлы.

Fift слов для манипулирования мешками клеток включают в себя:

- B>boc ($B - c$), десериализует «стандартный» мешок клеток (т. е. мешок клеток с ровно одной корневой клеткой), представленный байтами B, и возвращает корневую *ячейку* c.
- boc+>B ($c\ x - B$), создает и сериализует «стандартный» мешок клеток, содержащий один корень *Cell* c вместе со всеми его потомками. Параметр *Integer* $0 \le x \le 31$ используется для передачи флагов, указывающих на дополнительные параметры сериализации пакета ячеек, причем отдельные биты имеют следующий эффект:
  - $+1$ позволяет создавать индекс мешков клеток (полезно для ленивой десериализации больших мешков клеток).
  - $+2$ включает CRC32-C всех данных в сериализацию (полезно для проверки целостности данных).
  - $+4$ явно сохраняет хэш корневой ячейки в сериализации (так что его можно быстро восстановить впоследствии без полной десериализации).
  - $+8$ хранит хеши некоторых промежуточных (нелистовых) клеток (полезно для ленивой десериализации больших мешков клеток).

– +16 хранит биты кэша ячеек для управления кэшированием десериализованных ячеек.

Типичными значениями x являются $x = 0$ или $x = 2$ для очень маленьких мешков ячеек (например, TON Blockchain external messages) и $x = 31$ для больших пакетов ячеек (например, блоков TON Blockchain).

- boc>B ($c - B$), сериализует небольшой «стандартный» мешок клеток с корнем *Cell c* и всеми его потомками. Эквивалентно 0 boc+>B.

Например, ячейка, созданная в 5.2 со значением типа TL-B Test, может быть сериализована следующим образом:

```
{ <b x{4A} s, rot 16 u, swap 32 i, b> } : mkTest
17239 -1000000001 mkTest boc>B Bx.
```

выходы "B5EE9C7201040101010000000000900000E4A4357C46535FF ok". Здесь Bx. — это слово, которое выводит шестнадцатеричное представление *значения Bytes*.

5.6 Операции ввода-вывода двоичных файлов и байтов

Следующие слова можно использовать для манипулирования значениями типа *Bytes* (произвольные байтовые последовательности) и их считывания или записи в двоичные файлы:

- B{⟨*hex-digits*⟩} (– B), отправляет литерал *Bytes* , содержащий данные, представленные четным числом шестнадцатеричных цифр.

- Bx. (B –), выводит шестнадцатеричное представление значения *Bytes*. Каждый байт представлен ровно двумя шестнадцатеричными цифрами в верхнем регистре.

- file>B (S – B), считывает (двоичный) файл с именем, указанным в *строке* S, и возвращает его содержимое в виде *значения Bytes*. Если файл не существует, создается исключение.

- B>file (B S –), создает новый (двоичный) файл с именем, указанным в *строке* S , и записывает данные из *байтов* B в новый файл. Если указанный файл уже существует, он перезаписывается.

- file-exists? (C – ?), проверяет, существует ли файл с именем, указанным в *строке* S.

Например, мешок ячеек, созданный в примере в 5.5, может быть сохранен на диск как sample.boc следующим образом:

```
{ <b x{4A} s, rot 16 u, swap 32 i, b> } : mkTest
17239 -1000000001 mkTest boc>B "sample.boc" B>file
```

Он может быть загружен и десериализован впоследствии (даже в другом Fift сеансе) с помощью file>B и B>boc:

```
{ <s 8 u@+ swap 0x4a <> abort"constructor tag mismatch"
      16 u@+ 32 i@+ s> } : unpackTest
"sample.boc" file>B B>boc unpackTest swap . .
```

печатает "17239 -1000000001 ok".

Кроме того, есть несколько слов для непосредственной упаковки (сериализации) данных в *значения Bytes* и их последующей распаковки (десериализации). Их можно комбинировать с B>file и file>B для сохранения данных непосредственно в двоичные файлы и загрузки их впоследствии.

- Blen (B – x), возвращает длину *Bytes* значения B в байтах.
- BhashB (B – B'), вычисляет хэш sha256 значения *Bytes*. Хэш возвращается в виде 32-байтового значения *Bytes*.
- Bhashu (B – x) вычисляет хэш sha256 значения *Bytes* и возвращает хэш в виде беззнакового 256-битного целого числа с большим порядком байтов.
- B= (B B' – ?), проверяет, равны ли две последовательности байтов.
- Bcmp (B B' – x), лексикографически сравнивает две последовательности Bytes и возвращает -1, 0 или 1, в зависимости от результата сравнения.
- B>i@ (B x – y), десериализует первые $x/8$ байт *Bytes* значения B как знаковое большое порядковое x-битное *целое число* y.

- B>i@+ ($B\ x\ -\ B'\ y$), десериализует первые $x/8$ байт B как знаковое бигендианское .x-bit *Integer* y аналогично B>i@, но также возвращает оставшиеся байты B

- B>u@, B>u@+, варианты B>i@ и B>i@+, десериализующие целые числа без знака.

- B>Li@, B>Li@+, B>Lu@, B>Lu@+, варианты с младшим порядковым числом байтов B>i@, B>i@+, B>u@, B>u@+.

- B| ($B\ x\ -\ B'\ B''$), вырезает первые x байт из *Bytes* значения B и возвращает как первые x байт (B'), так и остальные (B'') в качестве новых значений *Bytes.*

- i>B ($x\ y\ -\ B$), хранит подписанное большое байтовое y-битное *целое число* x в *значение Bytes* B, состоящее из ровно $y/8$ байт. Целое число y должно быть кратно восьми в диапазоне $0 \ldots 256$.

- u>B ($x\ y\ -\ B$), хранит беззнаковое y-битное *целое число* x в *значении Bytes* B, состоящем из точно $y/8$ байт, аналогично i>B.

- Li>B, Lu>B, варианты i>B и you>B с младшим порядком байтов.

- B+ ($B'\ B''\ -\ B$), объединяет две последовательности байтов.

6 Операции, специфичные для TON

В этой главе описываются специфичные для TON Fift слова, за исключением слов, используемых для манипулирования клетками, которые уже обсуждались в предыдущей главе.

6.1 Криптография Ed25519

Fift предлагает интерфейс к той же криптографии эллиптической кривой Ed25519, используемой TVM, описанной в Приложении A [5]:

- now ($-\ x$), возвращает текущее Unixtime в виде целого *числа*.

- newkeypair ($-\ B\ B'$), генерирует новую пару закрытых/открытых ключей Ed25519 и возвращает как закрытый ключ B, так и открытый

ключ B' в виде значений 32байт *байт байт* . Качество ключей достаточно хорошее для целей тестирования. Реальные приложения должны подавать достаточную энтропию в OpenSSL PRNG перед генерацией пар ключей Ed25519.

- priv>pub (B – B'), вычисляет открытый ключ, соответствующий закрытому ключу Ed25519. Как открытый ключ B' , так и закрытый ключ B представлены 32-байтовыми значениями *Bytes* .

- ed25519_sign (B B' – B''), подписывает данные B закрытым ключом Ed25519 B' (значение 32 байта *Bytes*) и возвращает подпись в виде 64-байтового *Bytes* значения B''.

- ed25519_sign_uint (x B' – B''), преобразует 256-битное целое число x с большим порядком байтов в 32-байтовую последовательность и подписывает его с помощью закрытого ключа Ed25519 B' аналогично ed25519_sign. Эквивалентно swap 256 u>B swap ed25519_sign. Целое число x , подлежащее подписанию, обычно вычисляется как хэш некоторых данных.

- ed25519_chksign (B B' B'' – ?), проверяет, является ли B' действительной подписью Ed25519 данных B с открытым ключом B''.

6.2 Анализатор адресов смарт-контрактов

Два специальных слова могут быть использованы для анализа адресов смарт-контрактов TON в удобочитаемых (base64 или base64url) формах:

- smca>$ (x y z – S), упаковывает стандартный адрес смарт-контракта TON с рабочей цепочкой x (подписанное 32-битное *целое число*) и адресом в рабочей *цепочке* y (беззнаковое 256-битное *целое число*) в 48-символьную строку S (человекочитаемое представление адреса) в соответствии с флагами z. Возможные отдельные флаги в z : $+1$ для неотправляемых адресов, $+2$ для адресов только тестовой сети и $+4$ для вывода base64url вместо base64.

- $>smca ($S$ – x y z -1 или 0), распаковывает стандартный адрес смарт-контракта TON из его удобочитаемого строкового представления S.

При успешном выполнении возвращает подписанную 32-разрядную рабочую цепочку x, неподписанный 256-разрядный адрес рабочей $цепочки\ y$, флаги z (где $+1$ означает, что адрес не подлежит отказу, $+2$, что адрес является адресом только testnet) и -1. При сбое толкает 0.

Пример удобочитаемого адреса смарт-контракта может быть десериализован и отображен следующим образом:

"Ef9Tj6fMJP-OqhAdhKXxq36DL-HYSzCc3-9O6UNzqsgPfYFX"
$>smca 0= abort"bad address"
rot . swap x. . cr

выходы "-1 538fa7... 0f7d 0", что означает, что указанный адрес находится в рабочей $цепочке\ -1$ (мастерчейн блокчейна TON), и что 256-битный адрес внутри рабочей цепи -1 является 0x538... f7d.

6.3 Манипуляции со словарем

Fift имеет несколько слов для работы *с hashmap* или *(TVM) словарем*, соответствующих значениям типа TL-B HashmapE $n\ X$, как описано в [4, 3.3]. Эти (TVM) словари не следует путать с Fift словарем, что совершенно другое дело. Словарь типа TL-B HashmapE $n\ X$ по сути представляет собой коллекцию ключ-значение с различными n-битными ключами (где $0 \le n \le 1023$) и значениями произвольного типа TL-B типа X. Словари представлены деревьями ячеек (полный макет можно найти в [4, 3.3]) и хранятся в виде значений типа *Cell* или *Slice* в стеке Fift. Иногда пустые словари представлены значением *Null*.

- dictnew ($-\ D$), отправляет значение *Null*, представляющее новый пустой словарь.
- idict! ($v\ x\ D\ n\ -\ D'\ -1$ или $D\ 0$), добавляет новое значение v (представленное *Slice*) с ключом, заданным знаковым большим порядком n-битного целого числа x в словарь D (представленный ячейкой или *null*) с n-битными ключами, и возвращает новый словарь D' и -1 об успехе. В противном случае возвращается неизменный словарь D и 0.

- idict!+ ($v\ x\ D\ n - D'\ -1$ или $D\ 0$), добавляет новую пару ключ-значение (x,v) в словарь D аналогично idict!, но терпит неудачу, если ключ уже существует, возвращая неизменный словарь D и 0.

- b>idict!, b>idict!+, варианты idict! и idict!+, принимающие новое значение *v* в *Builder* вместо *Slice*.

- udict!, udict!+, b>udict!, b>udict!+, варианты idict!, idict!+, b>idict!, b>idict!+, но с беззнаковым n-битным целым числом в *x, используемым* качестве ключа.

- sdict!, sdict!+, b>sdict!, b>sdict!+, варианты idict!, idict!+, b>idict!, b>idict!+, но с первыми n битами данных *Slice* x , используемыми в качестве ключа.

- idict@ ($x\ D\ n - v\ -1$ или 0), ищет ключ, представленный знаковым большим порядковым n-битным *целым числом* x в словаре, представленном *ячейкой* D. Если ключ найден, возвращает соответствующее значение в виде *Slice v* и -1. В противном случае возвращает значение 0.

- idict@- ($x\ D\ n - D'\ v\ -1$ или $D\ 0$), ищет ключ, представленный знаковым n-битным целым числом x с большим порядком *байтов* в словаре, представленном *ячейкой* D. Если ключ найден, удаляет его из словаря и возвращает измененный словарь D', соответствующее значение в виде *Slice* v и -1. В противном случае возвращает неизмененный словарь D и 0.

- idict- ($x\ D\ n - D'\ -1$ или $D\ 0$), удаляет целочисленный ключ x из словаря D аналогично idict@-, но не возвращает значение, соответствующее x в старом словаре D.

- udict@, udict@-, udict-, варианты idict@, idict@-, idict-, но с *беззнаковым большим порядком байт* n-битного *целого числа* x, используемого в качестве ключа.

- sdict@, sdict@-, sdict-, варианты idict@, idict@-, idict-, но с ключом, предоставленным в первых n битах *Slice* k.

- dictmap ($D\ n\ e - s'$), применяет маркер выполнения e (т.е. анонимную функцию) к каждой из пар ключ-значение, хранящихся в словаре D с n-битными ключами. Маркер выполнения выполняется один раз для каждой пары ключ-значение, при этом *построитель* b и *Slice* v (содержащий значение) передаются в стек перед выполнением e. После выполнения e должен оставить в стеке либо модифицированный *Builder* b' (содержащий все данные из b вместе с новым значением v') и -1, либо 0, указывающий на сбой. В последнем случае соответствующий ключ опускается из нового словаря.

- dictmerge ($D\ D'\ n\ e - D''$), объединяет два словаря D и D' с n-битными ключами в один словарь D'' с одинаковыми клавишами. Если ключ присутствует только в одном из словарей D и D', этот ключ и соответствующее значение дословно копируются в новый словарь $D00$. В противном случае маркер выполнения (анонимная функция) e вызывается для объединения двух значений v и v', соответствующих одному и тому же *ключу* k в D и D' соответственно. Перед вызовом e отправляется *построитель* b и два *фрагмента* v и v', представляющие два значения, подлежащие объединению. После выполнения e оставляет либо модифицированный *Builder* b' (содержащий исходные данные из b вместе с комбинированным значением) и -1, либо 0 при сбое. В последнем случае соответствующий ключ опускается из нового словаря.

Fift также предлагает некоторую поддержку префиксных словарей:

- pfxdict! ($v\ k\ s\ n - s'\ -1$ или $s\ 0$), добавляет пару ключ-значение (k,v),обе представленные *Slices, в префиксный словарь* s с ключами длиной не более n. При успешном выполнении возвращает измененный словарь s' и -1. При сбое возвращает исходный словарь s и 0.

- pfxdict!+ ($v\ k\ s\ n - s'\ -1$ или $s\ 0$), добавляет пару ключ-значение (k,v) в префикс словарь s аналогично pfxdict!, но не удается, если ключ уже существует.

- pfxdict@ ($k\ s\ n - v\ -1$ или 0), ищет ключ k (представленный *Slice*) в префиксном словаре s с длиной ключей, ограниченной n битами. При успешном выполнении возвращает найденное значение v и -1. При сбое возвращает $\text{значение}\ 0$.

6.4 Вызов TVM из Fift

TVM может быть связан с Fift переводчиком. В этом случае становится доступным несколько Fift примитивов, которые можно использовать для вызова TVM с аргументами, предоставленными из Fift. Аргументы могут быть подготовлены в стеке Fift, который полностью передается новому экземпляру TVM. Полученный стек и код выхода передаются обратно Fift и могут быть проверены впоследствии.

- runvmcode (... s – ... x), вызывает новый экземпляр TVM с текущим продолжением cc, инициализированным из *Slice* s, тем самым выполняя код s в TVM. Исходный стек Fift (без s) передается полностью как начальный стек TVM. Когда TVM завершает работу, его результирующий стек используется в качестве нового стека Fift, а код выхода x помещается в его верхнюю часть. Если x не равно нулю, указывая на то, что TVM был завершен необработанным исключением, следующая запись стека сверху содержит параметр этого исключения, а x — код исключения. В этом случае все остальные записи удаляются из стека.

- runvmdict (... s – ... x), вызывает новый экземпляр TVM с текущим продолжением cc, инициализированным из *Slice* s аналогично runvmcode, но также инициализирует специальный регистр c3 с тем же значением и отправляет ноль в начальный стек TVM перед началом выполнения TVM. В типичном приложении *Slice* s состоит из

кода выбора подпрограммы, который использует integer верхней части стека для выбора подпрограммы для выполнения, что позволяет определять и выполнять несколько взаимно рекурсивных подпрограмм (см. [4, 4.6] и 7.8). Селектор, равный нулю, соответствует подпрограмме main() в большой TVM-программе.

- runvm (... s c – ... x c'), вызывает новый экземпляр TVM как с текущим продолжением cc, так и со специальным регистром c3, инициализированным из *Slice* s, аналогично runvmdict (без нажатия лишнего нуля на исходный стек TVM; при необходимости его можно явно отправить под s), а также инициализирует специальный регистр c4 («корень постоянных данных», ср. [4, 1.4]) с *ячейкой* c. Конечное значение c4 возвращается в верхней части конечного стека Fift как другая *ячейка* c'. Таким образом, можно эмулировать выполнение смарт-контрактов, которые проверяют или изменяют их постоянное хранилище.

- runvmctx (... s c t – ... x c'), вариант runvm, который также инициализирует c7 («контекст») *Tuple* t. Таким образом, выполнение смарт-контракта TVM внутри TON Blockchain может быть полностью эмулировано, если правильный контекст загружен в c7 (ср. [5, 4.4.10]).

- gasrunvmcode (... s z – ... x z'), газозависимая версия runvmcode, которая принимает дополнительный *Integer* аргумент z (исходный предел газа) в верхней части стека и возвращает газ, потребляемый этим TVM, как новое *целое* значение z' *в верхней части стека*.

- gasrunvmdict (... s z – ... x z'), газозависимая версия runvmdict.

- gasrunvm (... s c z – ... x c' z'), газосодержащая версия runvm.

- gasrunvmctx (... s c t z – ... x c' z'), газозависимая версия runvmctx.

Например, можно создать экземпляр TVM, выполняющий простой код следующим образом:

```
2  3  9 x{1221} runvmcode .s
```

Стек TVM инициализируется тремя целыми числами 2, 3 и 9 (в указанном порядке; 9 — самая верхняя запись), а затем *Slice* x{1221} содержащий 16 бит данных и без ссылок, преобразуется в продолжение TVM и выполняется. Обратившись к Приложению A из [4], мы видим, что x{12} — это код инструкции TVM XCHG s1, s2, а x{21} — это код инструкции TVM OVER (не путать с Fift примитив over, который, кстати, оказывает такое же влияние на стек). Результатом вышеуказанного выполнения является:

```
execute XCHG s1,s2
execute OVER
execute implicit RET
   3  2 9 2 0

      ok
```

Здесь 0 — код выхода (указывающий на успешное завершение TVM), а 3 2 9 2 — окончательное состояние стека TVM.

Если во время выполнения TVM создается необработанное исключение, код этого исключения возвращается в виде кода выхода:

```
2 3 9 x{122} runvmcode .s
```

производит

```
execute XCHG s1,s2
handling exception code 6: invalid or too short opcode
default exception handler, terminating vm with exit code 6
0 6
 ok
```

Обратите внимание, что TVM выполняется с включенным внутренним ведением журнала, и его журнал отображается в стандартном выводе.

Простые программы TVM могут быть представлены литералами Slice с помощью конструкции x{...} аналогично приведенным выше примерам. Более сложные программы обычно создаются с помощью ассемблера Fift, как описано в следующей главе.

7 Использование ассемблера Fift

Ассемблер Fift представляет собой короткую программу (в настоящее время менее 30 КБ), полностью написанную на Fift, которая преобразует читаемую человеком мнемонику инструкций TVM в их двоичное представление. Например, можно было бы написать <{ s1 s2 XCHG OVER }>s вместо x{1221} в примере, обсуждаемом в 6.4, при условии, что ассемблер Fift был загружен заранее (обычно фразой «Asm.fif»).

7.1 Загрузка сборщика Fift

Ассемблер Fift обычно находится в файле Asm.fif в каталоге библиотеки Fift (который обычно содержит стандартные файлы библиотеки Fift, такие как Fift.fif). Обычно он загружается путем размещения фразы «Asm.fif» include в самом начале программы, которая должна использовать Fift ассемблер:

- include (S –), загружает и интерпретирует исходный файл Fift по пути, заданному строкой S. Если имя файла S не начинается с косой черты, Fift включать путь поиска, обычно взятый из переменной среды FIFTPATH или аргумента командной строки -I интерпретатора Fift (и равного /usr/lib/fift, если оба отсутствуют), используется для поиска S.

Текущая реализация ассемблера Fift интенсивно использует пользовательские определяющие слова (ср. 4.8); его источник можно изучить как хороший пример того, как определяющие слова могут быть использованы для написания очень компактных Fift программ (см. также оригинальное издание [1], где обсуждается простой ассемблер 8080 Forth).

В будущем почти все слова, определенные ассемблером Fift, будут перемещены в отдельный словарь (пространство имен). В настоящее время они определены в глобальном пространстве имен, поскольку Fift еще не поддерживает пространства имен.

7.2 Основы Fift ассемблера

Ассемблер Fift наследует от Fift свою постфиксную операцию, т.е. аргументы или параметры записываются перед соответствующими инструкциями. Например, инструкция ассемблера TVM, представленная как XCHG s1,s2 в [4], представлена в Fift ассемблере как s1 s2 XCHG.

Fift ассемблерный код обычно открывается специальным открывающим словом, таким как <{, и заканчивается заключительным словом, таким как }> или }>s. Например

```
"Asm.fif" include
<{ s1 s2 XCHG OVER }>s
csr.
```

компилирует две инструкции TVM XCHG s1,s2 и OVER и возвращает результат в виде *Slice* (поскольку используется }>s). Результирующий *фрагмент* отображается csr., получая

```
x{1221}
```

Можно использовать Приложение A из [4] и убедиться, что x{12} действительно является (нулевой кодовой страницей) кодом инструкции TVM XCHG s1,s2, и что x{21} является кодом инструкции TVM OVER (не путать с Fift примитивным over).

В дальнейшем мы будем считать, что Fift assember уже загружен и опускаем фразу «Asm.fif» включить из наших примеров.

Ассемблер Fift использует стек Fift простым способом, используя несколько верхних записей стека для хранения *Builder* с собираемым кодом и аргументами для инструкций TVM. Например:

- <{ (– b), начинает часть Fift ассемблерного кода, помещая пустой *Builder* в стек Fift (и потенциально переключая пространство имен на то, которое содержит все Fift слова, специфичные для ассемблера). Приблизительно эквивалентно <b.

- }> (b – b'), завершает часть Fift ассемблерного кода и возвращает собранную часть как *Builder* (и потенциально восстанавливает исходное пространство имен). Примерно эквивалентно nop в большинстве ситуаций.

- }>c (b – c), завершает часть ассемблерного кода Fift и возвращает собранную часть в виде *ячейки* (и потенциально восстанавливает исходное пространство имен). Приблизительно эквивалентно b>.

- }>s (b – s), завершает часть ассемблерного кода Fift аналогично }>, но возвращает собранную часть как *Slice*. Эквивалент }>c <s.

- OVER (b – b'), собирает код инструкции TVM OVER, добавляя его в *Builder* в верхней части стека. Приблизительно эквивалентно x{21} s,.

- s1 (– s), толкает специальный *срез,* используемый ассемблером Fift для представления «регистра стека» s1 TVM.

- s0... s15 (– s), слова, похожие на s1, но толкающие *Slice* , представляющий другие «регистры стека» TVM. Обратите внимание, что s16... s255 должен быть доступен с использованием слова s().

- s() (x – s), принимает *аргумент Integer* $0 \le x \le 255$ и возвращает специальный *Slice,* используемый ассемблером Fift для представления «регистра стека» s(x).

- XCHG (b s s' – b'), берет два специальных *среза,* представляющих два «регистра стека» $s(i)$ и s(j) из стека, и добавляет в *Builder* b код для TVM-инструкции XCHG s(i),s(j).

В частности, обратите внимание, что слово OVER, определяемое ассемблером Fift, имеет совершенно иной эффект, чем Fift примитив over.

Фактическое действие OVER и других Fift ассемблерных слов несколько сложнее, чем у x{21} s,. Если новый код инструкции не вписывается в *Builder* b (т.е. если b будет содержать более 1023 бит данных после добавления нового кода инструкции), то эта и все последующие инструкции собираются в новый *Builder* ~b, а старый *Builder* b дополняется

ссылкой на ячейку , полученной из ˜b после генерации ˜b закончено. Таким образом, длинные участки кода TVM автоматически разделяются на цепочки допустимых *ячеек,* содержащих не более 1023 бит каждая. Поскольку TVM интерпретирует одинокую ссылку на ячейку в конце продолжения как неявный JMPREF, это разбиение кода TVM на ячейки почти не влияет на выполнение.

7.3 Выталкивание целочисленных констант

Инструкция TVM PUSHINT x, выталкивающая *целочисленную* константу x при вызове, может быть собрана с помощью Fift ассемблерных слов INT или PUSHINT:

- PUSHINT ($b\ x - b'$), собирает TVM-инструкцию PUSHINT x в *Builder*.
- INT ($b\ x - b'$), эквивалентный PUSHINT.

Обратите внимание, что аргумент PUSHINT является *целочисленным* значением, взятым из стека Fift, и не обязательно является литералом. Например, <{ 239 17 * INT }>s является допустимым способом сборки инструкции PUSHINT 4063, поскольку $239 \cdot 17 = 4063$. Обратите внимание, что умножение выполняется Fift во время сборки, а не во время выполнения TVM. Последнее вычисление может быть выполнено с помощью <{ 239 INT 17 INT MUL }>s:

```
<{ 239 17 * INT }>s dup csr. runvmcode .s 2drop
<{ 239 INT 17 INT MUL }>s dup csr. runvmcode .s 2drop
```

производит

```
x{810FDF}
execute PUSHINT 4063
execute implicit RET
4063 0
 ok
```

```
x{8100EF8011A8}
execute PUSHINT 239
execute PUSHINT 17
execute MUL
execute implicit RET
4063 0
  ok
```

Обратите внимание, что ассемблер Fift выбирает самую короткую кодировку инструкции PUSHINT x в зависимости от ее аргумента x.

7.4 Немедленные аргументы

Некоторые инструкции TVM (например, PUSHINT) принимают немедленные аргументы. Эти аргументы обычно передаются Fift слову, собирающему соответствующую инструкцию в стеке Fift. Целочисленные немедленные аргументы обычно представлены целыми числами, ячейки — ячейками, продолжениями — конструкторами и ячейками, а фрагменты ячеек — фрагментами. Например, 17 ADDCONST собирает TVM-инструкцию ADDCONST 17, а x{ABCD_} PUSHSLICE собирает PUSHSLICE xABCD_:

```
239 <{ 17 ADDCONST x{ABCD_} PUSHSLICE }>s dup csr.
runvmcode . swap . csr.
```

производит

```
x{A6118B2ABCD0}
execute ADDINT 17
execute PUSHSLICE xABCD_
execute implicit RET
0 256 x{ABCD_}
```

В некоторых случаях Fift ассемблер делает вид, что может принять немедленные аргументы, которые находятся вне диапазона для соответствующей инструкции TVM. Например, ADDCONST x определяется только для $-128 \le x < 128$, но ассемблер Fift принимает 239 ADDCONST:

```
17 <{ 239 ADDCONST }>s dup csr. runvmcode .s
```

производит

```
x{8100EFA0}
execute PUSHINT 239
execute ADD
execute implicit RET
256 0
```

Мы видим, что "ADDCONST 239" был молчаливо заменен PUSHINT 239 и ADD. Эта функция удобна, когда немедленный аргумент в ADDCONST сам по себе является результатом Fift вычислений, и трудно оценить, всегда ли он будет вписываться в требуемый диапазон.

В некоторых случаях существует несколько версий одних и тех же инструкций TVM, одна из которых принимает немедленный аргумент, а другая - без каких-либо аргументов. Например, существуют как инструкции LSHIFT *n*, так и LSHIFT. В Fift ассемблере таким вариантам присваивается отчетливая мнемоника. В частности, LSHIFT n представлен n LSHIFT#, а LSHIFT представлен самим собой.

7.5 Немедленное продолжение

Когда немедленный аргумент является продолжением, удобно создать соответствующий *Builder* в стеке Fift с помощью вложенного <{ ...} > конструкция. Например, инструкции по сборке TVM

```
PUSHINT 1
SWAP
PUSHCONT {
    MULCONST 10
} REPEAT
```

может быть собран и выполнен

```
7
<{ 1 INT SWAP <{ 10 MULCONST }> PUSHCONT REPEAT }>s dup csr.
```

runvmcode drop .

производит

```
x{710192A70AE4}
execute PUSHINT 1
execute SWAP
execute PUSHCONT xA70A
execute REPEAT
repeat 7 more times
execute MULINT 10
execute implicit RET
repeat 6 more times
...
repeat 1 more times
execute MULINT 10
execute implicit RET
repeat 0 more times
execute implicit RET
10000000
```

Существуют более удобные способы использования литеральных продолжений, созданных с помощью ассемблера Fift. Например, приведенный выше пример также может быть собран с помощью

```
<{ 1 INT SWAP CONT:<{ 10 MULCONST }> REPEAT }>s csr.
```

или даже

```
<{ 1 INT SWAP REPEAT:<{ 10 MULCONST }> }>s csr.
```

оба производят "x{710192A70AE4} ok".

Кстати, лучшим способом реализации приведенного выше цикла является REPEATEND:

```
7  <{ 1 INT SWAP REPEATEND 10 MULCONST }>s dup csr.
runvmcode drop .
```

или

```
7     <{ 1 INT SWAP REPEAT: 10 MULCONST }>s dup csr.
runvmcode drop .
```

оба производят "x{7101E7A70A}" и выводят "10000000" после семи итераций цикла.

Обратите внимание, что несколько инструкций TVM, которые хранят продолжение в отдельной ссылке ячейки (например, JMPREF), принимают их аргумент в ячейке, а не в *конструкторе*. В таких ситуациях <{ ... } >c конструкция может быть использована для получения этого немедленного аргумента.

7.6 Поток управления: петли и условные обозначения

Почти все инструкции потока управления TVM, такие как IF, IFNOT, IFRET, IFNOTRET, IFELSE, WHILE, WHILEEND, REPEAT, REPEATEND, UNTIL и UNTILEND — могут быть собраны аналогично REPEAT и REPEATEND в примерах 7.5 применительно к литеральным продолжениям. Например, ассемблерный код TVM

```
DUP
PUSHINT 1
AND
PUSHCONT {
      MULCONST 3
      INC
}
PUSHCONT {
      RSHIFT 1
}
IFELSE
```

который вычисляет $3n + 1$ или $n/2$ в зависимости от того, является ли его аргумент n нечетным или четным, может быть собран и применен к $n = 7$

```
<{ DUP 1 INT AND
    IF:<{ 3 MULCONST INC }>ELSE<{ 1 RSHIFT# }>
}>s dup csr.
7 swap runvmcode drop .
```

производит

```
x{2071B093A703A492AB00E2}
  ok
execute DUP
execute PUSHINT 1
execute AND
execute PUSHCONT xA703A4
execute PUSHCONT xAB00
execute IFELSE
execute MULINT 3
execute INC
execute implicit RET
execute implicit RET
22  ok
```

Конечно, более компактным и эффективным способом реализации этого условного выражения было бы

```
<{ DUP 1 INT AND
    IF:<{ 3 MULCONST INC }>ELSE: 1 RSHIFT#
}>s dup csr.
```

или

```
<{ DUP 1 INT AND
    CONT:<{ 3 MULCONST INC }> IFJMP
    1 RSHIFT#
}>s dup csr.
```

оба производят один и тот же код "x{2071B093A703A4DCAB00}".

Fift ассемблерные слова, которые могут быть использованы для создания таких «высокоуровневых» условий и циклов, включают IF:<{, IFNOT:<{, IFJMP:<{, }>ELSE<{, }>ELSE:, }>IF, REPEAT:<{, UNTIL:<{, WHILE:<{, }>DO<{, }>DO:, AGAIN:<{, }>AGAIN, }>REPEAT и }>UNTIL. Их полный список можно найти в исходном файле Asm.fif. Например, цикл UNTIL может быть создан с помощью UNTIL:<{ ... } > или <{ ... } >UNTIL, и цикл WHILE от WHILE:<{ ... } >DO<{ ... } >.

Если мы решим сохранить условную ветвь в отдельной ячейке, мы можем использовать <{ ... } >c конструкт вместе с такими инструкциями, как IFJMPREF:

```
<{ DUP 1 INT AND
    <{ 3 MULCONST INC }>c IFJMPREF
    1 RSHIFT# }>s
dup csr.
3 swap runvmcode .s
```

имеет тот же эффект, что и код из предыдущего примера при выполнении, но содержится в двух отдельных ячейках:

```
x{2071B0E302AB00}
  x{A703A4}
execute DUP
execute PUSHINT 1
execute AND
execute IFJMPREF (2946....A1DD)
execute MULINT 3
execute INC
execute implicit RET
10 0
```

7.7 Определения макросов

Поскольку инструкции TVM реализуются в ассемблере Fift с использованием Fift слов, которые оказывают предсказуемое влияние на

стек Fift, ассемблер Fift автоматически является ассемблером макросов, поддерживающим определения макросов. Например, предположим, что мы хотим определить определение макроса RANGE $x\ y$, которое проверяет, находится ли значение TVM в верхней части стека между целочисленными литералами x и y (включительно). Это определение макроса может быть реализовано следующим образом:

```
{ 2dup > ' swap if
  rot DUP rot GEQINT SWAP swap LEQINT AND
} : RANGE
<{ DUP 17 239 RANGE IFNOT: DROP ZERO }>s dup csr.
66 swap runvmcode drop .
```

который производит

```
x{2020C210018100F0B9B0DC3070}
execute DUP
execute DUP
execute GTINT 16
execute SWAP
execute PUSHINT 240
execute LESS
execute AND
execute IFRET
66
```

Обратите внимание, что GEQINT и LEQINT сами по себе являются макроопределениями, определенными в Asm.fif, поскольку они не соответствуют непосредственно инструкциям TVM. Например, x GEQINT соответствует инструкции TVM GTINT $x - 1$.

Кстати, приведенный выше код можно сократить на два байта, заменив IFNOT: DROP ZERO на AND.

7.8 Более крупные программы и подпрограммы

Более крупные программы TVM, такие как смарт-контракты TON Blockchain, обычно состоят из нескольких взаимно рекурсивных подпрограмм, причем один или несколько из них выбираются в качестве подпрограмм верхнего уровня (называемых main() или recv_internal() для смарт-контрактов). Выполнение начинается с одного из подпрограмм верхнего уровня, который может свободно вызывать любой из других определенных подпрограмм, который, в свою очередь, может вызывать любые другие подпрограммы, которые им нужны.

Такие TVM-программы реализуются с помощью селекторной функции, которая принимает дополнительный целочисленный аргумент в стеке TVM; это целое число выбирает саму вызываемую подпрограмму (ср. [4, 4.6]). Перед выполнением код этой функции селектора загружается как в специальный регистр c3, так и в текущее продолжение cc. Селектор основной функции (обычно ноль) помещается в начальный стек, и запускается выполнение TVM. Впоследствии подпрограмма может быть вызвана с помощью подходящей инструкции TVM, такой как CALLDICT n, где n — (целочисленный) селектор вызываемой подпрограммы.

Ассемблер Fift предлагает несколько слов, облегчающих реализацию таких больших телевизионных программ. В частности, подпрограммы могут быть определены отдельно и им могут быть присвоены символические имена (вместо числовых селекторов), которые могут быть использованы для их последующего вызова. Ассемблер Fift автоматически создает функцию селектора из этих отдельных подпрограмм и возвращает ее в качестве результата сборки верхнего уровня.

Вот простой пример такой программы, состоящей из нескольких подпрограмм. Эта программа вычисляет комплексное число $(5 + i)^4 \cdot (239 - i)$:

```
"Asm.fif" include

PROGRAM{
```

```
NEWPROC add
NEWPROC sub
NEWPROC mul

sub <{ s3 s3 XCHG2 SUB s2 XCHG0 SUB }>s PROC

// compute (5+i)^4 * (239-i)
main PROC:<{
   5 INT 1 INT // 5+i
   2DUP
   mul CALL
   2DUP
   mul CALL
   239 INT -1 INT
    mul JMP
}>

add PROC:<{
   s1 s2 XCHG
   ADD -ROT ADD SWAP
}>

a b c d -- ac-bd ad+bc : complex number multiplication
mul PROC:<{
   s3 s1 PUSH2 // a b c d a c
   MUL           //  a b c d ac
   s3 s1 PUSH2 // a b c d ac b d
   MUL           // a b c d ac bd
   SUB           // a b c d ac-bd
   s4 s4 XCHG2 // ac-bd b c a d
   MUL           // ac-bd b c ad
   - ROT MUL ADD
}>

}END>s
```

```
dup csr.
runvmdict .s
```

Эта программа производит:

```
x{FF00F4A40EF4A0F20B}
  x{D9_}
    x{2_}
      x{1D5C573C00D73C00E0403BDFFC5000E_}
      x{04A81668006_}
    x{2_}
      x{140CE840A86_}
      x{14CC6A14CC6A2854112A166A282_}
implicit PUSH 0 at start
execute SETCP 0
execute DICTPUSHCONST 14 (xC_,1)
execute DICTIGETJMP
execute PUSHINT 5
execute PUSHINT 1
execute 2DUP
execute CALLDICT 3
execute SETCP 0
execute DICTPUSHCONST 14 (xC_,1)
execute DICTIGETJMP
execute PUSH2 s3,s1
execute MUL
...
execute ROTREV
execute MUL
execute ADD
execute implicit RET
114244 114244 0
```

Ниже приводятся некоторые замечания и комментарии, основанные на предыдущем примере:

- Программа TVM открывается программой PROGRAM{ и закрывается либо }END>c (который возвращает собранную программу в виде ячейки), либо } END>s (который возвращает *фрагмент*).

- Новая подпрограмма объявляется с помощью фразы NEWPROC ⟨$name$⟩. Это объявление назначает следующее положительное целое число в качестве селектора для вновь объявленной подпрограммы и сохраняет это целое число в константе ⟨$name$⟩. Например, приведенные выше объявления определяют add, sub и mul как целочисленные константы, равные 1, 2 и 3 соответственно.

- Некоторые подпрограммы предварительно объявлены и не нуждаются в повторном объявлении NEWPROC. Например, main — это идентификатор подпрограммы, привязанный к целочисленной константе (селектору) 0.

- Другие предопределенные селекторы подпрограмм, такие как recv_internal (равный 0) или recv_external (равный −1), полезный для реализации смарт-контрактов TON Blockchain (ср. [5, 4.4]), могут быть объявлены с помощью константы (например, -1 constant recv_external).

- Подпрограмма может быть определена либо с помощью слова PROC, принимающего целочисленный селектор подпрограммы и *Slice* , содержащий код для этой подпрограммы, либо с помощью конструкции $\mathrm{hselectori}$ PROC:<{ ... } >, удобный для определения более крупных подпрограмм.

- Инструкции CALLDICT и JMPDICT могут быть собраны с помощью слов CALL и JMP, которые принимают целочисленный селектор подпрограммы, вызываемой в качестве немедленного аргумента, передаваемого в стеке Fift.

- Текущая реализация ассемблера Fift собирает все подпрограммы в словарь с 14-битными знаковыми целочисленными ключами.

Поэтому все селекторы подпрограмм должны находиться в диапазоне -2^{13}… $2^{13} - 1$.

- Если подпрограмма с неизвестным селектором вызывается во время выполнения, код автоматически вставляет Fift ассемблер, создает исключение с кодом 11. Этот код также автоматически выбирает ноль кодовой страницы для кодирования инструкций с помощью инструкции SETCP0.
- Ассемблер Fift проверяет, что все подпрограммы, объявленные NEWPROC, фактически определены PROC или PROC:<{ до окончания программы. Он также проверяет, что подпрограмма не переопределена.

Следует иметь в виду, что очень простые программы (включая простейшие смарт-контракты) можно сделать более компактными, исключив этот общий механизм выбора подпрограмм в пользу пользовательского кода выбора подпрограмм и удалив неиспользуемые подпрограммы. Например, приведенный выше пример может быть преобразован в

```
<{ 11 THROWIF
    CONT:<{ s3 s1 PUSH2 MUL s3 s1 PUSH2 MUL SUB
            s4 s4 XCHG2 MUL -ROT MUL ADD }>
    5 INT 1 INT 2DUP s4 PUSH CALLX
    2DUP s4 PUSH CALLX
    ROT 239 INT -1 INT ROT JMPX
}>s
dup csr.
runvmdict .s
```

который производит

```
x{F24B9D5331A85331A8A15044A859A8A075715C24D85C24D8588100EF7F58D9}
implicit PUSH 0 at start
```

```
execute THROWIF 11
execute PUSHCONT x5331A85331A8A15044A859A8A0
execute PUSHINT 5
execute PUSHINT 1
execute 2DUP
execute PUSH s4
execute EXECUTE
execute PUSH2 s3,s1
execute MUL
...
execute XCHG2 s4,s4
execute MUL
execute ROTREV
execute MUL
execute ADD
execute implicit RET
114244 114244 0
```

References

[1] L. Brodie, Starting Forth: Introduction to the FORTH Language and Operating System for Beginners and Professionals, 2nd edition, Prentice Hall, 1987. Available at https://www.forth.com/starting-forth/.

[2] L. Brodie, Thinking Forth: A language and philosophy for solving problems, Prentice Hall, 1984. Available at http://thinking-forth. sourceforge.net/.

[3] N. Durov, Telegram Open Network, 2017.

[4] N. Durov, Telegram Open Network Virtual Machine, 2018.

[5] N. Durov, Telegram Open Network Blockchain, 2018.

A Список Fift слов

В этом приложении приведен алфавитный список почти всех Fift слов, включая примитивы и определения из стандартной библиотеки Fift.fif, но исключая Fift ассемблерные слова, определенные в Asm.fif (поскольку ассемблер Fift является просто приложением с точки зрения Fift). Некоторые экспериментальные слова были исключены из этого списка. Другие слова, возможно, были добавлены или удалены из Fift после написания этого текста. Список всех слов, доступных в вашем Fift переводчика, может быть проверен путем выполнения **слов**.

Каждое слово описывается его именем, за которым следует его *стековая нотация* в скобках, указывающая на несколько значений в верхней части стека Fift до и после выполнения слова; все более глубокие записи стека обычно считаются нетронутыми. После этого предоставляется текстовое описание эффекта слова. Если это слово обсуждалось в предыдущем разделе настоящего документа, ссылка на этот раздел включена.

Активные слова и активные префиксные слова, которые анализируют часть входного потока сразу после их появления, перечислены здесь измененным образом. Во-первых, эти слова перечислены рядом с той частью входных данных, которую они анализируют; сегмент каждой записи, который на самом деле является Fift словом, подчеркивается для ударения. Во-вторых, их стековый эффект обычно описывается с точки зрения пользователя и отражает действия, выполняемые на этапе выполнения охватывающих блоков и определений слов.

Например, активное префиксное слово B{, используемое для определения литералов Bytes (ср. 5.6), указано как B{⟨*hex-digits*⟩}, а его эффект стека показан как (– B) вместо (– B 1 e), хотя реальный эффект выполнения активного слова B{ на этапе компиляции охватывающего блока или определения слова является последним (см. 4.2).

- ! (x p –), сохраняет новое значение x в *Box* p, ср. 2.14.
- "⟨*string*⟩" (– S), проталкивает *строковый* литерал в стек, см. 2.9 и 2.10.

- # ($x\ S - x'\ S'$), выполняет один шаг преобразования целого *числа* x в его десятичное представление путем добавления к *строке* S одной десятичной цифры, представляющей $x \bmod 10$. Также возвращается частное x' := $\lfloor x/10 \rfloor$.

- #> ($S - S'$), завершает преобразование целого *числа* в его удобочитаемое представление (десятичное или иное), начатое с <#, путем обратного изменения *строки* S. Эквивалентно $reverse.

- #s ($x\ S - x'\ S'$), выполняет # один или несколько раз, пока частное x' не станет неположительным. Эквивалентно { # over 0<= } until.

- $# ($-\ x$), передает общее количество аргументов командной строки, переданных Fift программе, ср. 2.18. Определяется только при вызове интерпретатора Fift в режиме сценария (с аргументом командной строки -s).

- $(⟨*string*⟩) (– ...), ищет слово $⟨*string*⟩ во время выполнения и выполняет его текущее определение. Обычно используется для доступа к текущим значениям аргументов командной строки, например, $(2) по существу эквивалентен @' $2.

- $() ($x - S$), передает аргумент командной строки x-й аналогично $*n*, но с *целым числом* $x \geq 0$, взятым из стека, ср. 2.18. Определяется только при вызове интерпретатора Fift в режиме сценария (с аргументом командной строки -s).

- $+ ($S\ S' - S.S'$), объединяет две строки, см. 2.10.

- $, ($b\ S - b'$), добавляет *строку* S к *Builder* b, см. 5.2. Строка интерпретируется как двоичная строка длиной $8n$, где n — количество байтов в представлении S в кодировке UTF-8.

- $n ($-\ S$), передает n-й аргумент командной строки в виде *строки* S, см. 2.18. Например, $0 передает имя выполняемого сценария, $1 — первый аргумент командной строки и так далее. Определяется только при

вызове интерпретатора Fift в режиме сценария (с аргументом командной строки -s).

- $= ($S\ S' - ?$), возвращает -1, если строки S и S' равны, 0 в противном случае, см. 2,13. Эквивалентно $cmp 0=.
- $>s ($S - s$), преобразует строку S в *срез*, см. 5.3. Эквивалент <b swap $, b> <s.
- $>smca ($S - x\ y\ z\ -1$ или 0), распаковывает стандартный адрес смарт-контракта TON из его удобочитаемого строкового представления S, см. 6.2. При успешном выполнении возвращает подписанный 32-разрядный рабочий цепь x, неподписанный 256-разрядный адрес в рабочей цепочке y, флаги z (где $+1$ означает, что адрес не является отказчивым, $+2$, что адрес является адресом только testnet) и -1. При сбое толкает 0.
- $@ ($s\ x - S$), извлекает первые x байт (т.е. $8x$ бит) из *Slice* s и возвращает их в виде строки S в кодировке UTF-8 , ср. 5.3. Если в s недостаточно битов данных, возникает исключение.
- $@+ ($s\ x - S\ s'$), аналогично $@, но возвращает и оставшуюся часть *Slice* s , ср. 5.3.
- $@? ($s\ x - S\ -1$ или 0), аналогично $@, но использует флаг для указания на сбой вместо создания исключения, ср. 5.3.
- $@?+ ($s\ x - S\ s'\ -1$ или $s\ 0$), аналогично $@+, но использует флаг для указания на сбой вместо создания исключения, ср. 5.3.
- $cmp ($S\ S' - x$), возвращает 0, если строки S и S' равны, -1, если S лексикографически меньше S', и 1, если S лексикографически больше S', ср. 2.13.
- $len ($S - x$), вычисляет длину байта (а не длину символа UTF-8!) строки, ср. 2.10.

- $pos ($S\ S' - x$ или -1), возвращает положение (смещение байта) x первого появления подстроки S' в строке S или -1.
- $reverse ($S - S'$), изменяет порядок символов UTF-8 в *строке* S. Если S не является допустимой строкой UTF-8, возвращаемое значение не определено и также может быть недопустимым.
- %1<< ($x\ y - z$), вычисляет $z := x \bmod 2^y = x\&(2^y - 1)$ для двух *целых чисел* x и $0 \le y \le 256$.
- ' ⟨*word-name*⟩ ($-$ *e*), возвращает маркер выполнения, равный текущему (время компиляции) определению ⟨*word-name*⟩, ср. 3.1. Если указанное слово не найдено, создается исключение.
- 'nop ($-$ *e*), передает определение nop по умолчанию — маркер выполнения, который ничего не делает при выполнении, см. 4.6.
- (') ⟨*word-name*⟩ ($-$ *e*), аналогично ', но возвращает определение указанного слова во время выполнения, выполняя поиск по словарю при каждом его вызове, ср. 4.6. Может использоваться для восстановления текущих значений констант внутри определений слов и других блоков с помощью фразы (') ⟨*word-name*⟩ execute.
- (-trailing) ($S\ x - S'$), удаляет из строки S все конечные символы с кодовой точкой X в кодовой кодировке UTF-8.
- (.) ($x - S$), возвращает *значение String* с десятичным представлением целого *числа* x. Эквивалент dup abs <# #s rot sign #> nip.
- (atom) ($S\ x - a\ -1$ или 0), возвращает единственный *Atom* a с именем, заданным строкой S, ср. 2.17. Если такого атома еще нет, либо создает его (если *целое число* x не равно нулю), либо возвращает один ноль, чтобы указать на сбой (если x равен нулю).
- (b.) ($x - S$), возвращает *значение String* с двоичным представлением целого *числа* x.

- (compile) ($l\ x_1\ ...\ x_n\ n\ e\ -\ l'$), расширяет *WordList* l так, чтобы он выталкивал $0 \le n \le 255$ значений x_1, ..., x_n в стек и выполнял маркер выполнения e при вызове, где $0 \le n \le 255$ — *целое число*, ср. 4.7. Если e равно специальному значению 'nop, последний шаг опускается.

- (create) ($e\ S\ x\ -$), создает новое слово с именем, равным *строке* S , и определением, равным *WordDef* e, используя флаги, переданные в *Целое число* $0 \le x \le 3$, ср. 4.5. Если бит $+1$ установлен в x, создает активное слово; если бит $+2$ установлен в x, создается префиксное слово.

- (def?) (S – ?), проверяет, определено ли слово S.

- (dump) ($x\ -\ S$), возвращает *значение типа String* с дампом самого верхнего значения стека x в том же формате, что и в .dump.

- (execute) ($x_1\ ...\ x_n\ n\ e\ -$ ...), выполняет маркер выполнения e, но сначала проверяет, что в стеке есть не менее $0 \le n \le 255$ значений, кроме самих n и e. Это аналог (компиляция), который может быть использован для немедленного «выполнения» (выполнения предполагаемого действия во время выполнения) активного слова после его немедленного выполнения, как описано в 4.2.

- (forget) (S –), забывает слово с именем, указанным в *String* S, ср. 4.5. Если слово не найдено, создается исключение.

- (number) ($S - 0$ или $x\ 1$ или $x\ y\ 2$), пытается разобрать строку S как целое число или дробный литерал, ср. 2.10 и 2.8. При сбое возвращает один 0. При успешном выполнении возвращает $x\ 1$, если S является допустимым целочисленным литералом со значением x, или $x\ y\ 2$, если S является допустимым дробным литералом со значением x/y.

- (x.) ($x\ -\ S$), возвращает *значение String* с шестнадцатеричным представлением целого *числа* x.

- ({) (– *l*), помещает пустой *WordList* в стек, ср. 4.7
- (}) (*l* – *e*), преобразует *WordList* в маркер выполнения (*WordDef*), делая невозможными все дальнейшие модификации, см. 4.7.
- * (*x y* – *xy*), вычисляет произведение *xy* двух *целых чисел x* и *y*, ср. 2,4.
- */ (*x y z* – ⌊xy/z⌋), "умножай-потом-дели": умножает два целых числа *x* и *y* , давая 513-битный промежуточный результат, затем делит произведение на *z*, ср. 2.4.
- */c (*x y z* – ⌈xy/z⌉), «умножай-потом-дели» с округлением потолка: умножает два целых числа *x* и *y* , получая 513-битный промежуточный результат, затем делит произведение на *z*, ср. 2.4.
- */cmod (*x y z* – *q r*), аналогично */c, но вычисляет как частное *q* := ⌈xy/z⌋, так и остаток *r* := *xy* – *qz*, ср. 2.4.
- */mod (*x y z* – *q r*), аналогично */, но вычисляет как частное *q* := b xy/z c , так и остаток *r* := *xy* – *qz*, ср. 2.4.
- */r (*x y z* – *q* := ⌊xy/z + 1/2⌋), «умножьте-затем-разделите» с ближайшим целым округлением: умножает два целых числа *x* и *y* с 513-битным промежуточным результатом, затем делит произведение на *z*, ср. 2,4.
- */rmod (*x y z* – *q r*), аналогично */r, но вычисляет как частное *q* := ⌊xy/z + 1/2⌋ , так и остаток *r* := *xy* – *qz*, ср. 2.4.
- *>> (*x y z* – *q*), аналогично */, но с разделением, замененным сдвигом вправо, ср. 2.4. Вычисляет *q* := $\lfloor xy/2^z \rfloor$ для $0 \leq z \leq 256$. Эквивалентно 1<< */.
- *>>c (*x y z* – *q*), аналогично */c, но с разделением, замененным сдвигом вправо, ср. 2.4. Вычисляет *q* := $\lceil xy/2^z \rceil$ для $0 \leq z \leq 256$. Эквивалентно 1<< */c.

- *>>r ($x\ y\ z - q$), аналогично */r, но с разделением, замененным сдвигом вправо, ср. 2.4. Вычисляет $q := \lfloor xy/2^z + 1/2 \rfloor$ для $0 \le z \le 256$. Эквивалентно 1<< */r.
- *mod ($x\ y\ z - r$), аналогично */mod, но вычисляет только остаток $r := xy - qz$, где $q := \lfloor xy/z \rfloor$. Эквивалент */mod nip.
- + ($x\ y - x+y$), вычисляет сумму *x+y* двух *целых чисел* x и y, ср. 2,4.
- +! ($x\ p -$), увеличивает целое значение, хранящееся в *Box* p , на *целое число* x, ср. 2.14. Эквивалентно tuck @ + swap !.
- +"⟨*string*⟩" ($S - S'$), объединяет строку S со строковым литералом, ср. 2.10. Эквивалент "⟨*string*⟩" $+.
- , ($t\ x - t'$), добавляет x к концу *кортежа* t и возвращает результирующий *Tuple t'*, ср. 2.15.
- - ($x\ y - x - y$), вычисляет разность $x - y$ двух *целых чисел* x и y, ср. 2.4.
- -! ($x\ p -$), уменьшает целое значение, хранящееся в *Box* p , на *целое число* x. Эквивалентно swap negate swap +!.
- -1 (– -1), толкает *Целое число* -1.
- -1<< ($x - -2^x$), вычисляет -2^x для $0 \le x \le 256$. Приблизительно эквивалентно 1<< свести на нет или -1 swap <<, но работает и для $x - 256$.
- -roll (x_n ... $x_0\ n - x_0\ x_n$ ... x_1), поворачивает верхние n записей стека в противоположном направлении, где $n \ge 0$ также передается в стеке, ср. 2.5. В частности, 1 -roll эквивалентен swap, а 2 –roll эквивалентно -rot.
- -rot ($x\ y\ z - z\ x\ y$), поворачивает три самые верхние записи стека в противоположную сторону, см. 2.5. Эквивалент rot rot.

- -trailing (S – S'), удаляет из строки S все конечные пробелы. Эквивалент bl (-trailing).
- -trailing0 (S – S'), удаляет из строки S все конечные символы '0'. Эквивалентно char 0 (-trailing).
- . (x –), печатает десятичное представление целого *числа* x, за которым следует один пробел,см. 2.4. Эквивалент ._ space.
- ." ⟨*string*⟩" (–), выводит константную строку в стандартный вывод, ср. 2.10.
- ._ (x –), выводит десятичное представление целого *числа* x без пробелов. Эквивалентен (.) type.
- .dump (x –), сбрасывает самую верхнюю запись стека так же, как .s сбрасывает все элементы стека, см. 2.15. Эквивалентно (dump) type space.
- .l (l –), печатает список в стиле Lisp l, см. 2.16.
- .s (–), сбрасывает все записи стека, начиная с самого глубокого, оставляя их нетронутыми, см. 2.5. Удобочитаемые представления записей стека выводятся разделенными пробелами, за которыми следует символ конца строки.
- .sl (–), сбрасывает все записи стека, оставляя их нетронутыми аналогично .s, но показывая каждую запись как список в стиле списка l , как это делает .l.
- .tc (–), выводит общее количество выделенных ячеек в стандартный поток ошибок.
- / ($x\ y$ – q := $\lfloor x/y \rfloor$), вычисляет коэффициент $\lfloor x/y \rfloor$ двух целых *чисел,* см. 2.4.

- /* ⟨*multiline-comment*⟩ */ (–), пропускает многострочный комментарий, разделенный словом "*/" (за которым следует пустой символ или символ конца строки), см. 2.2.
- // ⟨*comment-to-eol*⟩ (–), пропускает однострочный комментарий до конца текущей строки, см. 2.2.
- /c ($x\ y$ – q := $\lceil x/y \rceil$), вычисляет потолочное округленное частное $\lceil x/y \rceil$ двух *целых чисел*, см. 2.4.
- /cmod ($x\ y$ – $q\ r$), вычисляет как потолочное округленное частное q := $\lceil x/y \rceil$, так и остаток r := $x - qy$, см. 2.4.
- /mod ($x\ y$ – $q\ r$), вычисляет как округленное частное q := $\lfloor x/y \rfloor$, так и остаток r := $x - qy$, см. 2.4.
- /r ($x\ y$ – q), вычисляет ближайшее целочисленное коэффициент $\lfloor x/y + 1/2 \rfloor$ двух *целых чисел,* см. 2,4.
- /rmod ($x\ y$ – $q\ r$), вычисляет как ближайшее целочисленное коэффициент q := $\lfloor x/y + 1/2 \rfloor$, так и остаток r := $x - qy$, см. 2.4.
- 0 (– 0), нажимает *Целое число* 0.
- 0! (p –), хранит *целое число* 0 в *Box* p, ср. 2.14. Эквивалентно 0 swap !.
- 0< (x – ?), проверяет, *< ли* x 0 (т.е. толкает -1, если x отрицательный, 0 в противном случае), см. 2.12. Эквивалентно 0 <.
- 0<= (x – ?), проверяет, ≤ ли x 0 (т.е. толкает -1, если x неположителен, 0 в противном случае), см. 2.12. Эквивалентно 0 <=.
- 0<> (x – ?), проверяет, является ли $x \neq 0$ (т.е. толкает -1, если x не равен нулю, 0 в противном случае), см. 2.12. Эквивалентно 0 <>.

- 0= (x – ?), проверяет, = ли $x = 0$ (т.е. толкает -1, если x равно нулю, 0 в противном случае), см. 2.12. Эквивалентно 0 =.
- 0> (x – ?), проверяет, ли $x > 0$ (т.е. толкает -1, если x положительный, 0 в противном случае), см. 2.12. Эквивалентно 0 >.
- 0>= (x – ?), проверяет, ли $x \geq 0$ (т.е. толкает -1, если x неотрицательный, 0 в противном случае), см. 2.12. Эквивалентно 0 >=.
- 1 (– 1), нажимает *Целое число* 1.
- 1+ (x – x + 1), вычисляет $x + 1$. Эквивалентно 1+.
- 1+! (p –), увеличивает целое значение, хранящееся в *Box* p, на единицу, см. 2,14. Эквивалентно 1 swap +!.
- 1- (x – $x - 1$), вычисляет $x - 1$. Эквивалент 1 -.
- 1-! (p –), уменьшает целое значение, хранящееся в *Box* p, на единицу. Эквивалентно -1 swap +!.
- 1<< (x – 2^x), вычисляет 2^x для $0 \leq x \leq 255$. Эквивалентно 1 swap <<.
- 1<<1- (x – 2^x-1), вычисляет 2^x-1 для $0 \leq x \leq 256$. Почти эквивалентно 1<< 1-, но работает для $x - 256$.
- 2 (– 2), толкает *Целое число* 2.
- 2* (x – $2x$), вычисляет $2x$. Эквивалентно 2*.
- 2+ (x – x + 2), вычисляет $x + 2$. Эквивалентно 2+.
- 2- (x – $x - 2$), вычисляет $x - 2$. Эквивалентно 2 -.
- 2/ (x – $\lfloor x/2 \rfloor$), вычисляет $\lfloor x/2 \rfloor$. Эквивалентно 2 / или 1 >>.

- 2=: ⟨*word-name*⟩ (*x y* –), активный вариант 2constant: определяет новое обычное слово ⟨*word-name*⟩, которое будет выталкивать заданные значения *x* и *y* при вызове, см. 2.7.
- 2constant (*x y* –), сканирует пустое имя слова *S* из оставшейся части входных данных и определяет новое обычное слово *S* как двойную константу, которая будет выталкивать заданные значения *x* и *y* (произвольных типов) при вызове, см. 4.5.
- 2drop (*x y* –), удаляет две самые верхние записи стека, см. 2.5. Эквивалентно drop.
- 2dup (*x y – x y x y*), дублирует самую верхнюю пару записей стека, см. 2.5. Эквивалентно over over.
- 2over (*x y z w – x y z w x y*), дублирует вторую самую верхнюю пару записей стека.
- 2swap (*a b c d – c d a b*), заменяет две самые верхние пары записей стека, см. 2.5.
- : ⟨*word-name*⟩ (*e* –), определяет новое обычное слово ⟨*word-name*⟩ в словаре, используя *WordDef e в* качестве его определения, см. 4.5. Если указанное слово уже присутствует в словаре, оно молчаливо переопределяется.
- :: ⟨*word-name*⟩ (*e* –), определяет новое активное слово ⟨*word-name*⟩ в словаре, используя *WordDef e в* качестве его определения, см. 4.5. Если указанное слово уже присутствует в словаре, оно молчаливо переопределяется.
- :: ⟨*word-name*⟩ (*e* –), определяет новый активный префикс слово hword-namei в словаре, используя *WordDef e* в качестве его определения, см. 4.5. Если указанное слово уже присутствует в словаре, оно молчаливо переопределяется.

- : ⟨*word-name*⟩ (*e* –), определяет новый обычный префикс слово ⟨*word-name*⟩ в словаре, используя *WordDef e* в качестве его определения, см. 4.5. Если указанное слово уже присутствует в словаре, оно молчаливо переопределяется.

- < (*x y* – ?), проверяет, *ли* $x < y$ (т.е. толкает -1, если *целое число x* меньше *целого числа y*, 0 в противном случае), см. 2.12.

- <# (– *S*), отправляет пустую строку *String*. Обычно используется для начала преобразования *целого числа* в его удобочитаемое представление, десятичное или в другое основание. Эквивалент " ".

- << (*x y* – $x \cdot 2^y$), вычисляет арифметический сдвиг влево двоичного числа *x* на $y \geq 0$ позиций, что дает $x \cdot 2^y$, см. 2,4.

- <</ (*x y z* – *q*), вычисляет $q := \lfloor 2^z x/y \rfloor$ для $0 \leq z \leq 256$, производя 513-битный промежуточный результат, аналогично */, см. 2.4. Эквивалентно 1<< swap */.

- <</c (*x y z* – *q*), вычисляет $q := \lceil 2^z x/y \rceil$ для $0 \leq z \leq 256$, производя 513-битный промежуточный результат, аналогично */c, см. 2.4. Эквивалентно 1<< swap */c.

- <</r (*x y z* – *q*), вычисляет $q := \lfloor 2^z x/y+1/2 \rfloor$ для $0 \leq z \leq 256$, получая 513-битный промежуточный результат, аналогично */r, см. 2.4. Эквивалентно 1<< swap */r.

- <= (*x y* – ?), проверяет, *ли* $x \leq y$ (т.е. толкает -1, если *целое число x* меньше или равно целому *числу y*, 0 в противном случае), см. 2.12.

- <> (*x y* – ?), проверяет, является ли $x \neq y$ (т.е. толкает -1, если *целые числа x* и *y* не равны, 0 в противном случае), см. 2.12.

- <b (– *b*), создает новый пустой *Builder*, см. 5.2.

- <s (c – s), преобразует ячейку c в *фрагмент* s , содержащий те же данные, см. 5.3. Обычно это знаменует собой начало десериализации клетки.

- = (x y – ?), проверяет, *ли* $x = y$ (т.е. толкает -1, если *целые числа* x и y равны, 0 в противном случае), см. 2.12.

- =: ⟨*word-name*⟩ (x –), активный вариант константы: определяет новое обычное слово ⟨*word-name*⟩, которое при вызове будет выталкивать заданное значение x , см. 2.7.

- > (x y – ?), проверяет, *>ли* x y (т.е. толкает -1, если *целое число* x больше *целого числа* y, 0 в противном случае), см. 2.12.

- >= (x y – ?), проверяет, ли $x \geq y$ (т.е. толкает -1, если *целое число* x больше или равно целому числу y, 0 в противном случае), см. 2.12.

- >> (x y – q := $\lfloor x \cdot 2^{-y} \rfloor$), вычисляет арифметический сдвиг вправо двоичного числа x на $0 \leq y \leq 256$ позиций, см. 2.4. Эквивалент 1<< /.

- >>c (x y – q := $\lceil x \cdot 2^{-y} \rceil$), вычисляет потолочно округленное частное q x на 2^y для $0 \leq y \leq 256$, см. 2.4. Эквивалентно 1<< /c.

- >>r (x y – q := $\lfloor x \cdot 2^{-y} + 1/2 \rfloor$), вычисляет ближайшее целочисленное округленное частное q из x на 2^y для $0 \leq y \leq 256$, см. 2.4. Эквивалентно 1<< /r.

- ?dup (x – x x или 0), дублирует *целое число* x, но только если оно не равно нулю, см. 2.5. В противном случае оставляет его нетронутым.

- @ (p – x), извлекает значение, хранящееся в настоящее время в *Box* p, см. 2,14.

- @' ⟨*word-name*⟩ (– e), восстанавливает определение указанного слова во время выполнения, выполняя поиск по словарю при каждом его вызове, а затем выполняет это определение, см. 2.7 и 4.6. Может использоваться для восстановления текущих значений

констант внутри определений слов и других блоков с помощью фразы @' ⟨*word-name*⟩, эквивалентной (') ⟨*word-name*⟩ execute.

- B+ (B' B'' – B), объединяет два значения *Bytes*, см. 5,6.
- B, (b B – b'), добавляет байты B к *Builder* b, см. 5.2. Если для *B* нет места в b , создается исключение.
- B= (B B' – ?), проверяет, равны ли две последовательности байтов, и возвращает -1 или 0 в зависимости от результата сравнения (см. 5.6).
- B>Li@ (B x – y), десериализует первые $x/8$ байт значения Bytes B как знаковое байтовое x-битное *целое число* y, см. 5.6.
- B>Li@+ (B x – B' y), десериализует первые $x/8$ байт B *как знаковое* с младшим порядком байтов x *x-bit Integer* y аналогично B>Li@, но также возвращает оставшиеся байты B, см. 5.6.
- B>Lu@ (B x – y), десериализует первые $x/8$ байт значения Bytes B как беззнакового наименьшего байта x-bit *Integer* y, см. 5.6.
- B>Lu@+ (B x – B' y), десериализует первые $x/8$ байт B как беззнаковое наименьшее байтовое x-битное *целое число* y аналогично B>Lu@, но также возвращает оставшиеся байты B, см. 5.6.
- B>boc (B – c), десериализует «стандартный» мешок ячеек (т. е. мешок ячеек с ровно одной корневой ячейкой), представленный *байтами* B, и возвращает корневую ячейку *c*, см. 5.5.
- B>file (B S –), создает новый (двоичный) файл с именем, указанным в *строке* S , и записывает данные из *Bytes* B в новый файл, см. 5.6. Если указанный файл уже существует, он перезаписывается.

- B>i@ ($B\ x - y$), десериализует первые $x/8$ байт значения Bytes B как знаковое большое порядковое x-битное *целое число* y, см. 5.6.
- B>i@+ ($B\ x - B'\ y$), десериализует первые $x/8$ байт B как знаковое бигендианское x-bit *Целое число* y аналогично B>i@, но также возвращает оставшиеся байты B, см. 5.6.
- B>u@ ($B\ x - y$), десериализует первые $x/8$ байт значения Bytes B как беззнакового большого порядкового *числа* x-bit *Integer* y, см. 5.6.
- B>u@+ ($B\ x - B'\ y$), десериализует первые $x/8$ байт B как беззнаковое большое порядковое x-битное *целое число* y аналогично B>u@, но также возвращает оставшиеся байты B, см. 5.6.
- B@ ($s\ x - B$), извлекает первые x байт (т.е. $8x$ бит) из *Slice* s и возвращает их в виде *Bytes* значения B, см. 5.3. Если в s недостаточно битов данных, возникает исключение.
- B@+ ($s\ x - B\ s'$), аналогично B@, но возвращает и оставшуюся часть *Slice* s, см. 5.3.
- B@? ($s\ x - B\ -1$ или 0), аналогично B@, но использует флаг для указания на сбой вместо создания исключения, см. 5.3.
- B@?+ ($s\ x - B\ s'\ -1$ или $s\ 0$), аналогично B@+, но использует флаг для указания на сбой вместо того, чтобы выдавать исключение, см. 5.3.
- Bcmp ($B\ B'\ -\ x$), лексикографически сравнивает две последовательности байтов и возвращает -1, 0 или 1, в зависимости от результата сравнения, см. 5.6.
- Bhash ($B - x$), устаревшая версия Bhashu. Вместо этого используйте Bhashu или BhashB.
- BhashB ($B - B'$), вычисляет хэш sha256 значения *Bytes*, см. 5,6. Хэш возвращается в виде 32-байтового значения *Bytes*.

- Bhashu ($B - x$), вычисляет хэш sha256 значения *Bytes*, см. 5,6. Хэш возвращается в виде 256-разрядного целого числа с большим порядковым номером без знака .

- Blen ($B - x$), возвращает длину *Bytes* значения B в байтах,см. 5,6.

- Bx. (B –), печатает шестнадцатеричное представление значения *Bytes*, см. 5,6. Каждый байт представлен ровно двумя шестнадцатеричными цифрами в верхнем регистре.

- B{⟨*hex-digits*⟩} (– B), передает литерал Bytes, содержащий данные, представленные четным числом шестнадцатеричных цифр, см. 5.6.

- B| ($B\ x - B'\ B''$), вырезает первые x байты из *Bytes* значения B и возвращает как первые x байт (B'), так и остальные (B'') в виде новых *значений Bytes* (см. 5.6).

- Li>B ($x\ y - B$), хранит подписанное y-битное *целое число байтов в* x с младшим порядком *Bytes* значение B, состоящее из ровно $y/8$ байт. Целое число y должно быть кратно восьми в диапазоне $0 \ldots 256$, см. 5.6.

- Lu>B ($x\ y - B$), хранит беззнаковое y-битное *целое число байта в* x с младшим порядком *Bytes* значение B, состоящее из точно $y/8$ байт. Целое число y должно быть кратно восьми в диапазоне $0 \ldots 256$, см. 5.6.

- [(–), открывает внутренний сеанс интерпретатора, даже если state больше нуля, т.е. все последующие слова выполняются немедленно, а не компилируются.

- [] ($t\ i - x$), возвращает $(i + 1)$-ый компонент t_{i+1} *Tuple t*, где $0 \le i <$ |t| , см. 2.15.

- [compile] ⟨*word-name*⟩ (–), компилирует ⟨*word-name*⟩, как если бы это было обычное слово, даже если оно активное, см. 4.6. По существу эквивалентно ' ⟨*word-name*⟩ execute.

-] (x_1 ... x_n n –), закрывает внутренний сеанс интерпретатора, открытый [, и вызывает (компилирует) или (выполняет) впоследствии в зависимости от того, превышает ли состояние ноль. Например, { [2 3 + 1] * } эквивалентно { 5 * }.

- '⟨*word*⟩ (– a), вводит литерал *Atom*, равный единственному *Atom* с именем, равным ⟨*word*⟩, см. 2.17. Эквивалент "⟨*word*⟩" atom.

- abort (S –), выдает исключение с сообщением об ошибке, взятым из *строки* S, см. 3.6.

- abort"⟨*message*⟩" (x –), выдает исключение с сообщением об ошибке ⟨*message*⟩, если *целое число* x не равно нулю, см. 3.6.

- abs (x – |x|), вычисляет абсолютное значение $|x| = \max(x,-x)$ целого *числа* x. Эквивалентно dup negate max.

- allot (n – t), создает новый массив, т.е. *Tuple,* состоящий из n новых пустых *Box, см.* 2.15. Эквивалент | { hole , } rot times.

- and ($x\ y$ – $x\&y$), вычисляет побитовое AND двух *целых чисел,* см. 2.4.

- anon (– a), создает новый уникальный анонимный *Atom*, см. 2.17.

- atom (S – a), возвращает единственный *Atom* a с именем S, создавая такой атом при необходимости, см. 2.17. Эквивалентно true (atom) drop.

- atom? (u – ?), проверяет, *ли* u является *Atom*, см. 2.17.

- b+ ($b\ b'$ – b''), объединяет двух *строителей* b и b', см. 5.2.

- b. (x –), печатает двоичное представление целого *числа* x, за которым следует один пробел. Эквивалентно b._ space.

- b._ (x –), выводит двоичное представление целого *числа* x без пробелов. Эквивалентен (b) type.

- b> (b – c), преобразует *Builder* b в новую *ячейку* c, содержащую те же данные, что и b, см. 5.2.

- b>idict! (v x D n – D' -1 или D 0), добавляет новое значение v (представленное *Builder*) с ключом, заданным подписанным большим порядком n-битным целым числом x, в словарь D с n-битными ключами и возвращает новый словарь D' и -1 об успехе, ср. 6.3. В противном случае возвращается неизменный словарь D и 0.

- b>idict!+ (v x D n – D' -1 или D 0) добавляет новую пару ключ-значение (x,v), в словарь D аналогично b>idict!, но терпит неудачу, если ключ уже существует, возвращая неизменный словарь D и 0, см. 6.3.

- b>sdict! (v k D n – D' -1 или D 0), добавляет новое значение v (представленное *Builder*) с ключом, заданным *первыми* n битами *Slice* k, в словарь D с n-битными ключами и возвращает новый словарь D' и -1 об успехе, см. 6.3. В противном случае возвращается неизменный словарь D и 0.

- b>sdict!+ (v k D n – D' -1 или D 0), добавляет новую пару ключ-значение (k,v) в словарь D аналогично b>sdict!, но не удается, если ключ уже существует, возвращая неизменный словарь D и 0, см. 6.3.

- b>udict! (v x D n – D' -1 или D 0), добавляет новое значение v *(представленное Builder*) с ключом, заданным беззнаковым n-битным целым числом x в словарь D с n-битными ключами, и возвращает новый словарь D' и -1 об успехе, см. 6.3. В противном случае возвращается неизменный словарь D и 0.

- b>udict!+ ($v\ x\ D\ n\ -\ D'\ -1$ или $D\ 0$), добавляет новую пару ключ-значение (x,v) в словарь D аналогично b>udict!, но не удается, если ключ уже существует, возвращая неизменный словарь D и 0, см. 6.3.

- bbitrefs ($b\ -\ x\ y$), возвращает как количество битов данных x, так и количество ссылок y, уже хранящихся в *Builder* b, см. 5.2.

- bbits (b – x), возвращает количество битов данных, уже сохраненных в *Builder* b. Результатом x является *целое число* в диапазоне 0*…* 1023, см. 5.2.

- bl (– x), толкает кодовую точку Юникода пробела, т.е. 32, см. 2.10.

- boc+>B ($c\ x\ -\ B$), создает и сериализует «стандартный» мешок клеток, содержащий один корень *Cell* c вместе со всеми его потомками, см. 5.5. Параметр *Integer* $0 \le x \le 31$ используется для передачи флагов, указывающих на дополнительные параметры сериализации пакета ячеек, причем отдельные биты имеют следующий эффект:

  – $+1$ позволяет создавать индекс мешков клеток (полезно для ленивой десериализации больших мешков клеток).

  – $+2$ включает CRC32-C всех данных в сериализацию (полезно для проверки целостности данных).

  – $+4$ явно сохраняет хэш корневой ячейки в сериализации (так что его можно быстро восстановить впоследствии без полной десериализации).

  – $+8$ хранит хеши некоторых промежуточных (нелистовых) клеток (полезно для ленивой десериализации больших мешков клеток).

  – $+16$ хранит биты кэша ячеек для управления кэшированием десериализованных ячеек.

Типичными значениями x являются $x = 0$ или $x = 2$ для очень маленьких мешков ячеек (например, TON Blockchain external messages) и $x = 31$ для больших пакетов ячеек (например, блоков TON Blockchain).

- boc>B (c – B), сериализует небольшой «стандартный» мешок клеток с корневой клеткой c и всеми его потомками, см. 5.5. Эквивалентно 0 boc+>B.
- box (x – p), создает новый *Box* , содержащий указанное значение x, см. 2.14. Эквивалентно hole tuck !.
- brefs (b – x), возвращает количество ссылок, уже сохраненных в *Builder* b, см. 5.2. Результатом x является *целое число* в диапазоне $0…4$.
- brembitrefs (b – x y), возвращает как максимальное количество дополнительных битов данных $0 \le x \le 1023$, так и максимальное количество дополнительных ссылок на ячейки $0 \le y \le 4$, которые могут храниться в *Builder* b, см. 5.2.
- brembits (b – x), возвращает максимальное количество дополнительных битов данных, которые могут храниться в *Builder* b, см. 5.2. Эквивалент bbits 1023 swap -.
- bremrefs (b – x), возвращает максимальное количество дополнительных ссылок на ячейки, которые могут храниться в *Builder* b, см. 5.2.
- bye (–), завершает Fift интерпретатор операционной системы с нулевым кодом выхода, см. 2.3. Эквивалентно 0 halt.
- b{⟨$binary\text{-}data$⟩} (– s), создает *фрагмент* s , не содержащий ссылок и до 1023 бит данных, указанных в ⟨$binary\text{-}data$⟩, который должен быть строкой, состоящей только из символов '0' и '1', см. 5.1.
- caddr (l – h''), возвращает третий элемент списка. Эквивалент cddr car.
- cadr (l – h'), возвращает второй элемент списка, см. 2.16. Эквивалент cdr car.
- car (l – h), возвращает заголовок списка, см. 2.16. Эквивалентно first.

- cddr (*l* – *t'*), возвращает хвост хвоста списка. Эквивалент cdr cdr.
- cdr (*l* – *t*), возвращает хвост списка, см. 2.16. Эквивалентно second.
- char ⟨*string*⟩ (– *x*), отправляет целое *число* с кодовой точкой Юникода первого символа ⟨*string*⟩, см. 2.10. Например, char * эквивалентен 42.
- chr (*x* – *S*), возвращает новую строку *S,* состоящую из одного символа в кодировке UTF-8 с кодовой точкой Юникода *x*.
- cmp (*x y* – *z*), сравнивает два *целых числа x* и *y* и толкает 1, если $x > y$, −1, если $x < y$, и 0, если $x = y$, см. 2,12. Приблизительно эквивалентно sgn.
- cond (*x e e'* –), если *целое число x* не равно нулю, выполняет *e*, в противном случае выполняет *e'*, см. 3.2.
- cons (*h t* – *l*), строит список из его головы (первого элемента) *h* и его хвоста (списка, состоящего из всех остальных элементов) *t*, см. 2.16. Эквивалентно pair.
- constant (*x* –), сканирует пустое имя слова *S* из оставшейся части входных данных и определяет новое обычное слово *S* как константу, которая будет выталкивать заданное значение *x* (произвольного типа) при вызове, см. 4.5 и 2.7.
- count (*t* – *n*), возвращает длину *n* – |t| *Tuple t*, см. 2.15.
- cr (–), выводит возврат каретки (или символ новой строки) в стандартный вывод, см. 2.10.
- create (*e* –), определяет новое обычное слово с именем, равным следующему слову, отсканированному из входных данных, используя *WordDef e* в качестве его определения, см. 4.5. Если слово уже существует, оно молчаливо переопределяется.

- csr. (s –), рекурсивно печатает *Slice* s, ср. 5.3. В первой строке биты данных s отображаются в шестнадцатеричной форме, встроенной в конструкцию x{...}, аналогичную той, которая используется для *литералов Slice* (ср. 5.1). На следующих строках ячейки, обозначаемые s, печатаются с большим отступом.

- def? ⟨*word-name*⟩ (– ?), проверяет, определено ли слово ⟨*word-name*⟩ во время выполнения, и возвращает соответственно -1 или 0.

- depth (– n), возвращает текущую глубину (общее количество записей) стека Fift как *целое число* $n \geq 0$.

- dictmap ($D\ n\ e$ – D'), применяет маркер выполнения e (т.е. анонимную функцию) к каждой из пар ключ-значение, хранящихся в словаре D с n-битными ключами, см. 6.3. Маркер выполнения выполняется один раз для каждой пары значений ключа, при этом *Builder* b и *Slice* v (содержащие значение) передаются в стек перед выполнением e. После выполнения e должен оставить в стеке либо модифицированный *Builder* b' (содержащий все данные из b вместе с новым значением v') и -1, либо 0, указывающий на сбой. В последнем случае соответствующий ключ опускается из нового словаря.

- dictmerge ($D\ D'\ n\ e$ – D''), объединяет два словаря D и D' с n-битными ключами в один словарь D'' с одинаковыми ключами, см. 6.3. Если ключ присутствует только в одном из словарей D и D', этот ключ и соответствующее значение дословно копируются в новый словарь D''. В противном случае маркер выполнения (анонимная функция) e вызывается для объединения двух значений v и v', соответствующих одному и тому же *ключу* k в D и D' соответственно. Перед вызовом e отправляется *построитель* b и два *фрагмента* v и v', представляющие два значения, подлежащие объединению. После выполнения e оставляет либо модифицированный *Builder* b' (содержащий исходные данные из b вместе с комбинированным

значением) и -1, либо 0 при сбое. В последнем случае соответствующий ключ опускается из нового словаря.

- dictnew (– D), нажимает значение *Null*, представляющее новый пустой словарь, см. 6.3. Эквивалентно null.

- does (x_1 ... x_n n e – e'), создает новый маркер выполнения e', который будет отправлять n значений x_1, ..., x_n в стек, а затем выполнять e при вызове, см. 4.7. Это примерно эквивалентно комбинации ({), (compile) и (}).

- drop (x –), удаляет запись в верхней части стека, см. 2.5.

- dup (x – x x), дублирует верхнюю часть стека, см. 2.5. Если стек пуст, создается исключение.

- ed25519_chksign (B B' B'' – ?), проверяет, является ли B' допустимой подписью Ed25519 данных B с открытым ключом B'', см. 6.1.

- ed25519_sign (B B' – B''), подписывает данные B закрытым ключом Ed25519 B' (значение 32 байта *Bytes*) и возвращает подпись в виде 64-байтового *Bytes* значения B'', см. 6.1.

- ed25519_sign_uint (x B' – B''), преобразует 256-битное целое число x с большим порядком байтов в 32-байтовую последовательность и подписывает его с помощью закрытого ключа Ed25519 B' аналогично ed25519_sign, см. 6.1. Эквивалентно swap 256 u>B swap ed25519_sign. Целое число x , подлежащее подписанию, обычно вычисляется как хэш некоторых данных.

- emit (x –), печатает символ в кодировке UTF-8 с кодовой точкой Юникода, заданной целым *числом* x , в стандартный вывод, см. 2.10. Например, 42 испускают звездочку «*», а 916 — греческую дельту «Δ». Эквивалентен chr type.

- empty? (s – ?), проверяет , является ли фрагмент пустым (т. е. не имеет битов данных и ссылок), и возвращает $ли$ $соответственно$ -1 или 0 (см. 5.3).

- eq? (u v – ?), проверяет равны $ли$ u и v *целым числам, атомам или значениям nulls,* см. 2.17. Если они не равны или относятся к разным типам или не относятся к одному из перечисленных типов, возвращает ноль.

- exch (x_n ... x_0 n – x_0 ... x_n), заменяет верхнюю часть стека n-й записью стека сверху, где $n \geq 0$ также берется из стека, см. 2,5. В частности, 1 exch эквивалентно swap, а 2 exch для swap rot.

- exch2 (... n m – ...), заменяет *n*-ю запись стека сверху с *m*-й записью стека сверху, где $n \geq 0$, $m \geq 0$ взяты из штабеля, см. 2,5.

- execute (e – ...), выполняет маркер выполнения (*WordDef*) e, см. 3.1.

- explode (t – x_1 ... x_n n), распаковывает *Tuple* t = (x_1,…,x_n) неизвестной длины n и возвращает эту длину ,см. 2,15.

- false (– 0), выталкивает 0 в стек, см. 2.11. Эквивалентно 0.

- file-exists? (S – ?), проверяет, существует ли файл с именем, указанным в *строке* S (см. 5.6).

- file>B (S – B), считывает (двоичный) файл с именем, указанным в *строке* S, и возвращает его содержимое в виде *значения Bytes* (см. 5.6). Если файл не существует, создается исключение.

- find (S – e -1 или e 1 или 0), найти *строку* S в словаре и возвращает ее определение как *WordDef* e , если найдено, за которым следует -1 для обычных слов или 1 для активных слов, см. 4.6. В противном случае нажимает 0.

- first (t – x), возвращает первый компонент кортежа, см. 2,15. Эквивалентно 0 [].

- fits (*x y* – ?), проверяет, является ли *целое число x* знаковым y-битным целым числом (т. е. является ли $-2^{y-1} \le x < 2^{y-1}$ для $0 \le y \le 1023$), и возвращает -1 или 0 соответственно.

- forget (–), забывает (удаляет из словаря) определение следующего отсканированного слова из ввода, см. 4.5.

- gasrunvm (... *s c z* – ... *x c' z'*), газозависимая версия runvm, ср. 6.4: вызывает новый экземпляр TVM как с текущим продолжением cc, так и со специальным регистром c3, инициализированным из *Slice s*, и инициализирует специальный регистр c4 («корень постоянных данных», см. [4, 1.4]) с *ячейкой c*. Затем запускается новый экземпляр TVM с ограничением газа, установленным в *z*. Фактически потребленный газ *z'* возвращается в верхней части конечного стека Fift, а конечное значение c4 возвращается непосредственно под верхней частью конечного стека Fift как *другая ячейка c'*.

- gasrunvmcode (... *s z* – ... *x z'*), газозависимая версия runvmcode, см. 6.4: вызывает новый экземпляр TVM с текущим продолжением cc, инициализированным из *Slice s*, и с ограничением газа, установленным на *z*, тем самым выполняя код *s* в TVM. Исходный стек Fift (без *s*) передается полностью как начальный стек нового экземпляра TVM. Когда TVM завершает работу, его результирующий стек используется в качестве нового стека Fift, с кодом выхода *x* и фактически потребленным газом *z'*, толкаемым в его верхней части. Если *x* не равно нулю, указывая на то, что TVM был завершен необработанным исключением, следующая запись стека сверху содержит параметр этого исключения, а *x* — код исключения. В этом случае все остальные записи удаляются из стека.

- gasrunvmctx (... *s c t z* – ... *x c' z'*), газозависимая версия runvmctx, см. 6.4. Отличается от gasrunmv тем, что инициализирует c7 *Tuple t*.

- gasrunvmdict (... *s z* – ... *x z'*), газозависимая версия runvmdict, см. 6.4: вызывает новый экземпляр TVM с текущим продолжением cc,

инициализированным из *Slice* s, и устанавливает ограничение газа $в$ z аналогично gasrunvmcode, но также инициализирует специальный регистр c3 с тем же значением и отправляет ноль в начальный стек TVM до начала выполнения TVM. Фактически потребленный газ возвращается в виде целого *числа* z'. В типичном приложении *Slice* s состоит из кода выбора подпрограммы, который использует integer верхней части стека для выбора подпрограммы для выполнения, что позволяет определять и выполнять несколько взаимно рекурсивных подпрограмм (см. [4, 4.6] и 7.8). Селектор, равный нулю, соответствует подпрограмме main() в большой TVM-программе.

- halt (x –), завершает работу операционной системы аналогично bye, но использует *Integer* x в качестве кода выхода,см. 2.3.

- hash (c – x), устаревшая версия hashu. Вместо этого используйте hashu или hashB.

- hashB (c – B), вычисляет хэш представления *Cell* c на основе sha256 (см. [4, 3.1]), который однозначно определяет c и всех его потомков (при условии отсутствия коллизий для sha256), см. 5.4. Результат возвращается в виде значения *Bytes*, состоящего ровно из 32 байт.

- hashu (c – x) вычисляет хэш представления *Cell* c на основе sha256 аналогично hashB, но возвращает результат в виде 256-битного целого числа с большим порядком байтов без знака.

- hold (S x – S'), добавляет к строке S один символ в кодировке UTF-8 с кодовой точкой Юникода x. Эквивалентно chr $+.

- hole (– p), создает новый *Box* p , который не содержит никакого значения, ср. 2.14. Эквивалентно null box.

- i, (b x y – b'), добавляет двоичное представление знакового y-битного целого числа x в *Builder* b, где $0 \leq y \leq 257$, ср. 5.2. Если в b недостаточно места (т.е. если b уже содержит более $1023 - y$ битов

данных), или если целое *число* x не укладывается в $биты\ y$, возникает исключение.

- i>B ($x\ y - B$), хранит подписанное большое байтовое y-битное *целое число* x в *значение Bytes* B, состоящее из ровно $y/8$ байт. Целое число y должно быть кратно восьми в диапазоне $0\ldots 256$, см. 5.6.

- i@ ($s\ x - y$), извлекает знаковое большое порядковое x-битное целое число из первых x бит *Slice* s, см. 5.3. Если s содержит меньше x бит данных, создается исключение.

- i@+ ($s\ x - y\ s'$), извлекает знаковое большое порядковое x-битное целое число из первых x бит Slice s аналогично i@, но возвращает и оставшуюся часть s, см. 5.3.

- i@? ($s\ x - y\ -1$ или 0), извлекает из Slice знаковое целое число биг-эндиана аналогично i@, но впоследствии при успешном нажатии целого числа -1 , см. 5.3. Если в $x\ бит$ осталось меньше s, нажимает целое число 0, чтобы указать на сбой.

- i@?+ ($s\ x - y\ s'\ -1$ или $s\ 0$), извлекает из *Slice s* знаковое целое число с большим порядком байтов и вычисляет оставшуюся часть этого *Slice* аналогично i@+, но после этого нажимает -1 , чтобы указать на успех, см. 5.3. При сбое нажимает на неизмененные $Slice\ s$ и 0, чтобы указать на сбой.

- idict! ($v\ x\ D\ n - D'\ -1$ или $D\ 0$), добавляет новое значение v (представленное *Slice*) с ключом, заданным знаковым биг-эндианом n-битным целым числом x в словарь D с n-битными ключами, и возвращает новый словарь D' и -1 при успешном выполнении, см. 6.3. В противном случае возвращается неизменный словарь D и 0.

- idict!+ ($v\ x\ D\ n - D'\ -1$ или $D\ 0$), добавляет новую пару ключ-значение (x,v) в словарь D аналогично idict!, но терпит неудачу, если ключ уже существует, возвращая неизменный словарь D и 0, см. 6.3.

- idict- ($x\ D\ n - D'\ -1$ или $D\ 0$), удаляет ключ, представленный знаковым биг-эндианом n-битным *целым числом* x из словаря, представленного *ячейкой* D, см. 6.3. Если ключ найден, удаляет его из словаря и возвращает измененный словарь D' и -1. В противном случае возвращает неизмененный словарь D и 0.

- idict@ ($x\ D\ n - v\ -1$ или 0), ищет ключ, представленный знаковым бигендианским n-битным *целым числом* x в словаре, представленном *Cell* или *Null* D, см. 6.3. Если ключ найден, возвращает соответствующее значение в виде *Slice* v и -1. В противном случае возвращает значение 0.

- idict@- ($x\ D\ n - D'\ v\ -1$ или $D\ 0$), ищет ключ, представленный знаковым n-битным целым числом x с большим порядком *байтов* в словаре, представленном ячейкой D, см. 6.3. Если ключ найден, удаляет его из словаря и возвращает измененный словарь D', соответствующее значение в виде *Slice* v и -1. В противном случае возвращает неизмененный словарь D и 0.

- if ($x\ e$ –), выполняет маркер выполнения (т.е. *WordDef*) e, но только если *целое число* x не равно нулю, см. 3.2.

- ifnot ($x\ e$ –), выполняет маркер выполнения e, но только если *целое число* x равно нулю, см. 3.2.

- include (S –), загружает и интерпретирует исходный файл Fift из пути, заданного строкой S, см. 7.1. Если имя файла S не начинается со косой черты, Fift включает путь поиска, обычно взятый из переменной среды FIFTPATH или аргумента командной строки -I интерпретатора Fift (и равного /usr/lib/fift, если оба отсутствуют) для поиска S.

- list (x_1 ... $x_n\ n - l$), строит список l длины n с элементами x_1, ..., x_n, в указанном порядке, см. 2.16. Эквивалентно null ' cons rot times.

- max (*x y – z*), вычисляет максимум $z := \max(x,y)$ двух *целых чисел x* и *y*. Эквивалент minmax nip.
- min (*x y – z*), вычисляет минимум $z := \min(x,y)$ двух *целых чисел x* и *y*. Эквивалентно minmax drop.
- minmax (*x y – z t*), вычисляет как минимальное $z := \min(x,y)$, так и максимальное $t := \max(x,y)$ двух *целых чисел x* и *y*.
- mod (*x y – r* := $x \bmod y$), вычисляет остаток $x \bmod y = x - y \cdot \lfloor x/y \rfloor$ деления *x* на *y*, см. 2.4.
- negate (*x* – −x), изменяет знак Целого *числа*, см. 2.4.
- newkeypair (– *B B0*), генерирует новую пару закрытого/открытого ключей Ed25519 и возвращает как закрытый ключ B, так и открытый ключ *B0* в виде значений 32 байт байта, см. 6.1. Качество ключей достаточно хорошее для целей тестирования. Реальные приложения должны подавать достаточную энтропию в OpenSSL PRNG перед генерацией пар ключей Ed25519.
- nil (– *t*), толкает пустой *Tuple t* = (). Эквивалентно 0 tuple.
- nip (*x y – y*), удаляет вторую запись стека сверху, см. 2.5. Эквивалентно swap drop.
- nop (–), ничего не делает, см. 4.6.
- not (*x* – $-1 - x$), вычисляет побитовое дополнение целого *числа*, см. 2,4.
- now (– *x*), возвращает текущее Unixtime как целое *число*, см. 6.1.
- null (– ⊥), нажимает значение *Null*, см. 2.16
- null! (*p* –), сохраняет значение *Null* в *Box p*. Эквивалент null swap !.
- null? (*x* – ?), проверяет, является *ли x Null*, см. 2.16.

- or ($x\ y$ – $x|y$), вычисляет побитовое OR двух *целых чисел,* см. 2.4.
- over ($x\ y$ – $x\ y\ x$), создает копию второй записи стека сверху над записью верхней части стека, см. 2.5.
- pair ($x\ y$ – t), создает новую пару t = (x,y), см. 2.15. Эквивалентно 2 tuple или | rot , swap ,.
- pfxdict! ($v\ k\ s\ n$ – s' -1 или s 0), добавляет пару ключ-значение (k,v),обе представленные *Slices,* в префиксный словарь s с ключами длиной не более n, см. 6.3. При успешном выполнении возвращает измененный словарь s' и -1. При сбое возвращает исходный словарь s и 0.
- pfxdict!+ ($v\ k\ s\ n$ – s' -1 или s 0), добавляет пару ключ-значение (k,v) в префиксный словарь s аналогично pfxdict!, но не удается, если ключ уже существует, см. 6.3.
- pfxdict@ ($k\ s\ n$ – v -1 или 0), ищет ключ k (представленный *Slice*) в префиксном словаре s с длиной ключей, ограниченной n битами, см. 6.3. При успешном выполнении возвращает значение, найденное как *Slice* v и -1. При сбое возвращает значение 0.
- pick (x_n ... x_0 n – x_n ...x_0 x_n), создает копию n-й записи из верхней части стека, где $n \geq 0$ также передается в стеке, см. 2.5. В частности, 0 pick эквивалентен dup, а 1 pick эквивалентно over.
- priv>pub (B – B'), вычисляет открытый ключ, соответствующий закрытому ключу Ed25519, см. 6.1. Как открытый ключ B' , так и закрытый ключ B представлены 32-байтовыми значениями *Bytes* .
- quit (… –), выходит на самый верхний уровень интерпретатора Fift (без печати ok в интерактивном режиме) и очищает стек, см. 2.3.
- ref, (b c – b'), добавляет к *Builder* b ссылку на *Cell* c, см. 5.2. Если b уже содержит четыре ссылки, создается исключение.

- ref@ (*s* – *c*), извлекает первую ссылку из *Slice s* и возвращает *ячейку c*, на которую ссылается (см. 5.3). Если ссылок не осталось, создается исключение.

- ref@+ (*s* – *s' c*) извлекает первую ссылку из *Slice s* аналогично ref@, но возвращает и остальную часть *s* (см. 5.3).

- ref@? (*s* – *c* −1 или 0), извлекает первую ссылку из *Slice s* аналогично ref@, но использует флаг для указания на сбой вместо создания исключения (см. 5.3).

- ref@?+ (*s* – *s' c* −1 или *s* 0), аналогично ref@+, но использует флаг для указания на сбой вместо того, чтобы выдавать исключение, см. 5.3.

- remaining (*s* – *x y*), возвращает как количество битов данных *x,* так и количество ссылок на ячейки *y*, оставшихся в *Slice s*, см. 5.3.

- reverse (x_1 ... x_n y_1 ... y_m n m – x_n ... x_1 y_1 ... y_m), переворачивает порядок n записей стека, расположенных непосредственно под самыми верхними m элементами, где как 0, так $\leq m,n \leq 255$ передаются в стеке.

- roll (x_n ... x_0 n – x_{n-1} ... x_0 x_n), поворачивает верхние n записей стека, где $n \geq 0$ также передается в стеке, см. 2.5. В частности, 1 roll эквивалентен swap, а 2 roll эквивалентен rot.

- rot (*x y z* – *y z x*), вращает три самые верхние записи стека.

- runvm (... *c c* – ... *x c'*), вызывает новый экземпляр TVM с текущим продолжением cc и специальным регистром c3, инициализированным из *Slice s*, и инициализирует специальный регистр c4 («корень постоянных данных», см. [4, 1.4]) с Ячейкой *c*, см. 6.4. В отличие от runvmdict, не проталкивает неявный ноль в исходный стек TVM; при необходимости он может быть явно передан под *s*. Конечное значение c4 возвращается в верхней части конечного стека Fift как другая *ячейка c'*. Таким образом, можно

эмулировать выполнение смарт-контрактов, которые проверяют или изменяют их постоянное хранилище.

- runvmcode (... s – ... x), вызывает новый экземпляр TVM с текущим продолжением cc, инициализированным из *Slice* s, тем самым выполняя код s в TVM, см. 6.4. Исходный стек Fift (без s) передается полностью как начальный стек нового экземпляра TVM. Когда TVM завершает работу, его результирующий стек используется в качестве нового стека Fift, а код выхода x помещается в его верхнюю часть. Если x не равно нулю, указывая на то, что TVM был завершен необработанным исключением, следующая запись стека сверху содержит параметр этого исключения, а x — код исключения. В этом случае все остальные записи удаляются из стека.

- runvmctx (... c c m – ... x c'), вариант runvm, который также инициализирует c7 («контекстный регистр» TVM) *Tuple t*, см. 6.4.

- runvmdict (... c – ... x), вызывает новый экземпляр TVM с текущим продолжением cc, инициализированным из *Slice* s аналогично runvmcode, но также инициализирует специальный регистр c3 с тем же значением и отправляет ноль в начальный стек TVM перед запуском, см. 6.4. В типичном приложении *Slice* s состоит из кода выбора подпрограммы, который использует integer верхней части стека для выбора подпрограммы для выполнения, что позволяет определять и выполнять несколько взаимно рекурсивных подпрограмм (см. [4, 4.6] и 7.8). Селектор, равный нулю, соответствует подпрограмме main() в большой TVM-программе.

- s, (b s – b'), добавляет биты данных и ссылки, взятые из *Slice* s в *Builder* b, см. 5.2.

- s> (s –), выдает исключение, если *Slice* s не пуст, см. 5.3. Обычно он отмечает конец десериализации ячейки, проверяя, остались ли какие-либо необработанные биты данных или ссылки.

- s>c (s – c), создает ячейку c непосредственно из *Slice* s, см. 5.3. Эквивалент <b swap s, b>.

- sbitrefs (s – x y), возвращает как количество битов данных x, так и количество ссылок на ячейки y, оставшихся в *Slice* s, см. 5.3. Эквивалентно remaining.

- sbits (s – x), возвращает количество битов данных x , оставшихся в *Slice* s, см. 5.3.

- sdict! (v k D n – D' -1 или D 0), добавляет новое значение v (представленное *Slice*) с ключом, заданным *первыми* n битами *Slice* k, в словарь D с n-битными ключами и возвращает новый словарь D' и -1 об успехе, см. 6.3. В противном случае возвращается неизменный словарь D и 0.

- sdict!+ (v k D n – D' -1 или D 0), добавляет новую пару ключ-значение (k,v) в словарь D аналогично sdict!, но терпит неудачу, если ключ уже существует, возвращая неизменный словарь D и 0, см. 6.3.

- sdict- (x D n – D' -1 или D 0), удаляет ключ, заданный первыми n битами данных *Slice* x из словаря, представленного ячейкой D, см. 6.3. Если key найден, удаляет его из словаря и возвращает измененный словарь D' и -1. В противном случае возвращает неизмененный словарь D и 0.

- sdict@ (k D n – v -1 или 0), ищет ключ, заданный первыми n битами данных *Slice* x в словаре, представленном *Cell* или *Null* D, ср. 6.3. Если ключ найден, возвращает соответствующее значение в виде *Slice* v и -1. В противном случае возвращает значение 0.

- sdict@- ($x\ D\ n - D'\ v\ -1$ или $D\ 0$), ищет ключ, заданный первыми n битами данных *Slice* x в словаре, представленном ячейкой D, см. 6.3. Если ключ найден, удаляет его из словаря и возвращает измененный словарь D', соответствующее значение в виде *Slice* v и -1. В противном случае возвращает неизмененный словарь D и 0.

- second ($t - x$), возвращает второй компонент кортежа, см. 2,15. Эквивалент 1 [].

- sgn ($x - y$), вычисляет знак *целого числа* x (т.е. толкает 1, если $x > 0$, -1, если $x < 0$, и 0, если $x = 0$), см. 2.12. Эквивалентно 0 cmp.

- shash ($s - B$), вычисляет хэш представления Slice на основе sha256 , сначала преобразуя его в ячейку, см. 5.4. Эквивалент s>c hashB.

- sign ($S\ x - S'$), добавляет знак минуса "-" к *строке S,* если *целое число* x отрицательно. В противном случае оставляет S нетронутым.

- single ($x - t$), создает новый синглтон $t = (x)$, т.е. одноэлементный *кортеж*. Эквивалентно 1 tuple.

- skipspc (–), пропускает пустые символы из текущей строки ввода до тех пор, пока не будет найден непустый символ или символ конца строки.

- smca>$ ($x\ y\ z - S$), упаковывает стандартный адрес смарт-контракта TON с рабочей цепочкой x (подписанное 32-битное *целое число*) и адресом в рабочей *цепочке* y (беззнаковое 256-битное *целое число*) в 48-символьную строку S (человекочитаемое представление адреса) согласно флагам z, см. 6.2. Возможные отдельные флаги в z : $+1$ для неотправляемых адресов, $+2$ для адресов только тестовой сети и $+4$ для вывода base64url вместо base64.

- space (–), выводит один пробел. Эквивалентно bl emit или ." ".

- sr, (b s – b'), создает новую *ячейку,* содержащую все данные и ссылки из *Slice* s, и добавляет ссылку на эту ячейку в *Builder* b, см. 5.2. Эквивалент s>c ref,.
- srefs (s – x), возвращает количество ссылок на ячейки x , оставшихся в *Slice* s, см. 5.3.
- swap (x y – y x), заменяет две самые верхние записи стека, см. 2.5.
- ten (– 10), выталкивает *целочисленную константу* 10.
- third (t – x), возвращает третий компонент кортежа, см. 2.15. Эквивалент 2 [].
- times (e n –), выполняет маркер выполнения (*WordDef*) e ровно n раз, если $n \geq 0$, см. 3.3. Если n отрицательно, создается исключение.
- triple (x y z – t), создает новый тройной t = (x,y,z), см. 2.15. Эквивалентно 3 tuple.
- true (– -1), толкает -1 в стек, см. 2.11. Эквивалент -1.
- tuck (x y – y x y), эквивалентный swap over, см. 2,5.
- tuple (x_1 ... x_n n – t), создает новый *Tuple* t := (x_1,…,x_n) из $n \geq 0$ самых верхних значений стека, см. 2.15. Эквивалент dup 1 reverse | { swap , } rot times, но более эффективно.
- tuple? (t – ?), проверяет, является ли t кортежем, и возвращает -1 или 0 соответственно.
- type (s –), печатает Строку s , взятую из верхней части стека, в стандартный вывод, см. 2.10.
- u, (b x y – b'), добавляет двоичное представление большого порядкового числа x *в* $Builder$ b y-битного целого *числа* x без знака,

где $0 \leq y \leq 256$, см. 5.2. Если операция невозможна, создается исключение.

- u>B ($x\ y - B$), хранит беззнаковое y-битное *целое число x* с большим порядком байта в *Bytes* значение B, состоящее из точно $y/8$ байт. Целое число y должно быть кратно восьми в диапазоне $0\ldots 256$, см. 5.6.

- u@ ($s\ x - y$), *извлекает из первого* с большим порядком байтов *из* x-битное целое число x *бита* первых x *битов Slice* s, см. 5.3. Если s содержит меньше x бит данных, создается исключение.

- u@+ ($s\ x - y\ s'$), извлекает беззнаковое большое порядковое x-битное целое число из первых x битами *Slice* s аналогично u@, но возвращает и оставшуюся часть s, см. 5.3.

- u@? ($s\ x - y\ -1$ или 0), извлекает из Slice целое число с большим порядком байтов без знака аналогично you@, но впоследствии выталкивает целое число -1 при успехе, см. 5.3. Если в x *бит* осталось меньше s, нажимает целое число 0, чтобы указать на сбой.

- u@?+ ($s\ x - y\ s'\ -1$ или $s\ 0$), извлекает из Slice s целое число с большим порядком байтов без знака и вычисляет оставшуюся часть этого *Slice* аналогично u@+, но затем нажимает -1, чтобы указать на успех,см. 5.3. При сбое нажимает на неизмененные *фрагменты* s и 0, чтобы указать на сбой.

- udict! ($v\ x\ D\ n - D'\ -1$ или $D\ 0$), добавляет новое значение v (представленное *Slice*) с ключом, заданным большим порядком байтов без знака n-битным целым *числом* x в словарь D с n-битными ключами, и возвращает новый словарь D' и -1 об успехе, см. 6.3. В противном случае возвращается неизменный словарь D и 0.

- udict!+ ($v\ x\ D\ n - D'\ -1$ или $D\ 0$), добавляет новую пару ключ-значение (x,v) в словарь D аналогично udict!, но терпит неудачу, если ключ уже существует, возвращая неизменный словарь D и 0, см. 6.3.

- udict- ($x\ D\ n\ –\ D'\ -1$ или $D\ 0$), удаляет ключ, представленный беззнаковым n-битным целым числом x с большим порядком *байтов* x, из словаря, представленного ячейкой D, см. 6.3. Если ключ найден, удаляет его из словаря и возвращает измененный словарь D' и -1. В противном случае возвращает неизмененный словарь D и 0.

- udict@ ($x\ D\ n – v\ -1$ или 0), ищет ключ, представленный беззнаковым бигендианским n-битным *целым числом* x в словаре, представленном *Cell* или *Null* D, см. 6.3. Если ключ найден, возвращает соответствующее значение в виде *Slice* v и -1. В противном случае возвращает значение 0.

- udict@- ($x\ D\ n\ –\ D'\ v\ -1$ или $D\ 0$), ищет ключ, представленный беззнаковым n-битным *целым числом* x в словаре, представленном ячейкой D, см. 6.3. Если ключ найден, удаляет его из словаря и возвращает измененный словарь D', соответствующее значение в виде *Slice* v и -1. В противном случае возвращает неизмененный словарь D и 0.

- ufits ($x\ y\ –\ ?$), проверяет, является ли *целое число* x целым числом без знака y-бит (т. е. является ли $0 \le x < 2y$ для $0 \le y \le 1023$), и возвращает -1 или 0 соответственно.

- uncons ($l\ –\ h\ t$), разлагает непустой список на его голову и хвост, см. 2.16. Эквивалент unpair.

- undef? ⟨*word-name*⟩ (– ?), проверяет, является ли слово ⟨*word-name*⟩ неопределенным во время выполнения, и возвращает соответственно -1 или 0.

- unpair ($t – x\ y$), распаковывает пару $t = (x,y)$, см. 2.15. Эквивалентно 2 untuple.

- unsingle ($t\ –\ x$), распаковывает синглетон $t = (x)$. Эквивалентно 1 untuple.

- until (e –), a to till loop, см. 3.4: выполняет *WordDef* e, затем удаляет целое число верхней части стека и проверяет, равно ли оно нулю. Если это так, то начинается новая итерация цикла путем выполнения e. В противном случае выходит из цикла.

- untriple (t – x y z), распаковывает тройной t = (x,y,z), см. 2.15. Эквивалентно 3 untuple.

- untuple (t n – x_1 ... x_n), возвращает все компоненты *Tuple* t = (x_1,…,x_n),но только в том случае, если его длина равна n, см. 2,15. В противном случае возникает исключение.

- variable (–), сканирует пустое имя слова S из оставшейся части входных данных, выделяет пустое *Box* и определяет новое обычное слово S как константу, которая будет толкать новый *Box* при вызове, см. 2.14. Эквивалентен hole constant.

- while (e e' –), a while loop, см. 3.4: выполняет *WordDef* e, затем удаляет и проверяет верхнее целое число стека. Если он равен нулю, выходит из цикла. В противном случае выполняется *WordDef* e', а затем начинается новая итерация цикла, выполняя e и проверяя условие выхода после этого.

- word (x – s), анализирует слово, разделенное символом, с кодовой точкой Юникода x из оставшейся части текущей входной строки и отправляет результат в виде *String*, см. 2.10. Например, слово bl word abracadabra type будет печатать строку «abracadabra». Если x = 0, пропускает начальные пробелы, а затем сканирует до конца текущую входную строку. Если x = 32, пропускает начальные пробелы перед синтаксическим анализом следующего слова.

- words (–), печатает названия всех слов, определенных в настоящее время в словаре, см. 4.6.

- x. (x –), печатает шестнадцатеричное представление (без префикса 0x) целого *числа* x, за которым следует один пробел. Эквивалентно x._ space.

- x._ (x –), печатает шестнадцатеричное представление (без префикса 0x) целого *числа* x без пробелов. Эквивалентен (x.) type.

- xor ($x\ y$ – $x \oplus y$), вычисляет побитовое eXclusive OR двух *целых чисел,* см. 2.4.

- x{⟨$hex\text{-}data$⟩} (– s), создает *фрагмент* s , не содержащий ссылок и до 1023 бит данных, указанных в ⟨$hex\text{-}data$⟩, см. 5.1. Точнее, каждая шестнадцатеричная цифра из ⟨$hex\text{-}data$⟩ преобразуется в четыре двоичные цифры обычным способом. После этого, если последний символ ⟨$hex\text{-}data$⟩ является подчеркиванием _, то все конечные двоичные нули и двоичный непосредственно предшествующий им удаляются из результирующей двоичной строки (подробнее см. [4, 1.0]). Например, x{6C_} эквивалентен b{01101}.

- { (– l), активное слово, которое увеличивает состояние внутренней переменной на единицу и отправляет новый пустой *WordList* в стек, см. 4.7.

- | (– t), создает пустой *Tuple* t = (), см. 2.15. Эквивалентно nil и 0 tuple.

- |+ ($s\ s'$ – s''), объединяет два *среза* s и s', см. 5.1. Это означает, что биты данных нового *Slice* s'' получаются путем объединения битов данных s и s', а список ссылок на ячейки s'' строится аналогичным образом путем объединения соответствующих списков для s и s'. Эквивалентно <b rot s, swap s, b> <s.

- |_ ($s\ s'$ – s''), учитывая два *среза* s и s', создает новый *Slice* s'', который получается из s путем добавления новой ссылки на *ячейку*, содержащую s', см. 5.1. Эквивалентно <b rot s, swap s>c ref, b> <s.

- } (l – e), активное слово, которое преобразует *WordList* l в *WordDef* (маркер выполнения) e, тем самым делая невозможными все дальнейшие модификации l , и уменьшает состояние внутренней переменной на единицу; затем толкает целое число 1, за которым следует 'nop, см. 4.7. Чистый эффект заключается в преобразовании построенного *WordList* в маркер выполнения и отправке этого маркера выполнения в стек либо немедленно, либо во время выполнения внешнего блока.