2.4 Simple integer arithmetic

When Fift encounters a word that is absent from the dictionary, but which can be interpreted as an integer constant (or "literal"), its value is pushed into the stack (as explained in **2.8** in more detail). Apart from that, several integer arithmetic primitives are defined:

- `+` ($x\ y - x + y$), replaces two *Integer*s x and y passed at the top of the stack with their sum $x + y$. All deeper stack elements remain intact. If either x or y is not an *Integer*, or if the sum does not fit into a signed 257-bit *Integer*, an exception is thrown.

- `-` ($x\ y - x - y$), computes the difference $x - y$ of two *Integer*s x and y. Notice that the first argument x is the second entry from the top of the stack, while the second argument y is taken from the top of the stack.

- `negate` ($x - -x$), changes the sign of an *Integer*.

- `*` ($x\ y - xy$), computes the product xy of two *Integer*s x and y.

- `/` ($x\ y - q := \lfloor x/y \rfloor$), computes the floor-rounded quotient $\lfloor x/y \rfloor$ of two *Integer*s.

- `mod` ($x\ y - r := x \bmod y$), computes the remainder $x \bmod y = x - y \cdot \lfloor x/y \rfloor$ of division of x by y.

- `/mod` ($x\ y - q\ r$), computes both the quotient and the remainder.

- `/c`, `/r` ($x\ y - q$), division words similar to `/`, but using ceiling rounding ($q := \lceil x/y \rceil$) and nearest-integer rounding ($q := \lfloor 1/2 + x/y \rfloor$), respectively.

- `/cmod`, `/rmod` ($x\ y - q\ r := x - qy$), division words similar to `/mod`, but using ceiling or nearest-integer rounding.

- `<<` ($x\ y - x \cdot 2^y$), computes an arithmetic left shift of binary number x by $y \geq 0$ positions, yielding $x \cdot 2^y$.

- `>>` ($x\ y - q := \lfloor x \cdot 2^{-y} \rfloor$), computes an arithmetic right shift by $y \geq 0$ positions.

- `>>c`, `>>r` ($x\ y - q$), similar to `>>`, but using ceiling or nearest-integer rounding.
- `and`, `or`, `xor` ($x\ y - x \oplus y$), compute the bitwise AND, OR, or XOR of two *Integer*s.
- `not` ($x - -1 - x$), bitwise complement of an *Integer*.
- `*/` ($x\ y\ z - \lfloor xy/z \rfloor$), "multiply-then-divide": multiplies two integers x and y producing a 513-bit intermediate result, then divides the product by z.
- `*/mod` ($x\ y\ z - q\ r$), similar to `*/`, but computes both the quotient and the remainder.
- `*/c`, `*/r` ($x\ y\ z - q$), `*/cmod`, `*/rmod` ($x\ y\ z - q\ r$), similar to `*/` or `*/mod`, but using ceiling or nearest-integer rounding.
- `*>>`, `*>>c`, `*>>r` ($x\ y\ z - q$), similar to `*/` and its variants, but with division replaced with a right shift. Compute $q = xy/2^z$ rounded in the indicated fashion (floor, ceiling, or nearest integer).
- `<</`, `<<c`, `<<r` ($x\ y\ z - q$), similar to `*/`, but with multiplication replaced with a left shift. Compute $q = 2^z x/y$ rounded in the indicated fashion (notice the different order of arguments y and z compared to `*/`).

In addition, the word "`.`" may be used to print the decimal representation of an *Integer* passed at the top of the stack (followed by a single space), and "`x.`" prints the hexadecimal representation of the top-of-stack integer. The integer is removed from the stack afterwards.

The above primitives can be employed to use the Fift interpreter in interactive mode as a simple calculator for arithmetic expressions represented in reverse Polish notation (with operation symbols after the operands). For instance,

```
7 4 - .
```

computes $7 - 4 = 3$ and prints "`3 ok`", and

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

computes $2 + 3 \cdot 4 = 14$ and $(2 + 3) \cdot 4 = 20$, and prints "`14 20 ok`".

2.5 Stack manipulation words

Stack manipulation words rearrange one or several values near the top of the stack, regardless of their types, and leave all deeper stack values intact. Some of the most often used stack manipulation words are listed below:

- `dup` (x – x x), duplicates the top-of-stack entry. If the stack is empty, throws an exception.[4]
- `drop` (x –), removes the top-of-stack entry.
- `swap` (x y – y x), interchanges the two topmost stack entries.
- `rot` (x y z – y z x), rotates the three topmost stack entries.
- `-rot` (x y z – z x y), rotates the three topmost stack entries in the opposite direction. Equivalent to `rot rot`.
- `over` (x y – x y x), creates a copy of the second stack entry from the top over the top-of-stack entry.
- `tuck` (x y – y x y), equivalent to `swap over`.
- `nip` (x y – y), removes the second stack entry from the top. Equivalent to `swap drop`.
- `2dup` (x y – x y x y), equivalent to `over over`.
- `2drop` (x y –), equivalent to `drop drop`.
- `2swap` (a b c d – c d a b), interchanges the two topmost pairs of stack entries.
- `pick` (x_n $\ldots x_0$ n – x_n $\ldots x_0$ x_n), creates a copy of the n-th entry from the top of the stack, where $n \geq 0$ is also passed in the stack. In particular, `0 pick` is equivalent to `dup`, and `1 pick` to `over`.
- `roll` ($x_n \ldots x_0$ n – $x_{n-1} \ldots x_0$ x_n), rotates the top n stack entries, where $n \geq 0$ is also passed in the stack. In particular, `1 roll` is equivalent to `swap`, and `2 roll` to `rot`.

[4] Notice that Fift word names are case-sensitive, so one cannot type `DUP` instead of `dup`.

- `-roll` ($x_n \ldots x_0\ n - x_0\ x_n \ldots x_1$), rotates the top n stack entries in the opposite direction, where $n \geq 0$ is also passed in the stack. In particular, `1 -roll` is equivalent to `swap`, and `2 -roll` to `-rot`.

- `exch` ($x_n \ldots x_0\ n - x_0 \ldots x_n$), interchanges the top of the stack with the n-th stack entry from the top, where $n \geq 0$ is also taken from the stack. In particular, `1 exch` is equivalent to `swap`, and `2 exch` to `swap rot`.

- `exch2` ($\ldots n\ m - \ldots$), interchanges the n-th stack entry from the top with the m-th stack entry from the top, where $n \geq 0$, $m \geq 0$ are taken from the stack.

- `?dup` ($x - x\ x$ or 0), duplicates an *Integer* x, but only if it is non-zero. Otherwise leaves it intact.

For instance, "`5 dup * .`" will compute $5 \cdot 5 = 25$ and print "`25 ok`".

One can use the word "`.s`"—which prints the contents of the entire stack, starting from the deepest elements, without removing the elements printed from the stack—to inspect the contents of the stack at any time, and to check the effect of any stack manipulation words. For instance,

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

prints

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

When Fift does not know how to print a stack value of an unknown type, it instead prints `???`.

2.6 Defining new words

In its simplest form, defining new Fift words is very easy and can be done with the aid of three special words: "`{`", "`}`", and "`:`". One simply opens the definition with `{` (necessarily followed by a space), then lists all the words that

constitute the new definition, then closes the definition with `}` (also followed by a space), and finally assigns the resulting definition (represented by a *WordDef* value in the stack) to a new word by writing `:` ⟨*new-word-name*⟩. For instance,

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

defines a new word `square`, which executes `dup` and `*` when invoked. In this way, typing `5 square` becomes equivalent to typing `5 dup *`, and produces the same result (25):

```
5 square .
```

prints "`25 ok`". One can also use the new word as a part of new definitions:

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

prints "`243 ok`", which indeed is 3^5.

If the word indicated after "`:`" is already defined, it is tacitly redefined. However, all existing definitions of other words will continue to use the old definition of the redefined word. For instance, if we redefine `square` after we have already defined `**5` as above, `**5` will continue to use the original definition of `square`.

2.7 Named constants

One can define *(named) constants*—i.e., words that push a predefined value when invoked—by using the defining word `constant` instead of the defining word "`:`" (colon). For instance,

```
1000000000 constant Gram
```

defines a constant `Gram` equal to *Integer* 10^9. In other words, 1000000000 will be pushed into the stack whenever `Gram` is invoked:

```
Gram 2 * .
```

prints "`2000000000 ok`".

Of course, one can use the result of a computation to initialize the value of a constant:

```
Gram 1000 / constant mGram
mGram .
```

prints "`1000000 ok`".

The value of a constant does not necessarily have to be an *Integer*. For instance, one can define a string constant in the same way:

```
"Hello, world!" constant hello
hello type cr
```

prints "`Hello, world!`" on a separate line.

If a constant is redefined, all existing definitions of other words will continue to use the old value of the constant. In this respect, a constant does not behave as a global variable.

One can also store two values into one "double" constant by using the defining word `2constant`. For instance,

```
355 113 2constant pifrac
```

defines a new word `pifrac`, which will push 355 and 113 (in that order) when invoked. The two components of a double constant can be of different types.

If one wants to create a constant with a fixed name within a block or a colon definition, one should use `=:` and `2=:` instead of `constant` and `2constant`:

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

produces

```
3 9 12  ok
7 49 56  ok
```

If one wants to recover the execution-time value of such a "constant", one can prefix the name of the constant with the word `@'`:

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

produces

```
( 3 , 9 ) ok
```

The drawback of this approach is that `@'` has to look up the current definition of constants `x` and `y` in the dictionary each time `showxy` is executed. Variables (cf. **2.14**) provide a more efficient way to achieve similar results.

2.8 Integer and fractional constants, or literals

Fift recognizes unnamed integer constants (called *literals* to distinguish them from named constants) in decimal, binary, and hexadecimal formats. Binary literals are prefixed by `0b`, hexadecimal literals are prefixed by `0x`, and decimal literals do not require a prefix. For instance, `0b1011`, `11`, and `0xb` represent the same integer (11). An integer literal may be prefixed by a minus sign "`-`" to change its sign; the minus sign is accepted both before and after the `0x` and `0b` prefixes.

When Fift encounters a string that is absent from the dictionary but is a valid integer literal (fitting into the 257-bit signed integer type *Integer*), its value is pushed into the stack.

Apart from that, Fift offers some support for decimal and common fractions. If a string consists of two valid integer literals separated by a slash `/`, then Fift interprets it as a fractional literal and represents it by two *Integer*s p and q in the stack, the numerator p and the denominator q. For instance, `-17/12` pushes -17 and 12 into the Fift stack (being thus equivalent to `-17 12`), and `-0x11/0b1100` does the same thing. Decimal, binary, and hexadecimal fractions, such as `2.39` or `-0x11.ef`, are also represented by two integers p and q, where q is a suitable power of the base (10, 2, or 16, respectively). For instance, `2.39` is equivalent to `239 100`, and `-0x11.ef` is equivalent to `-0x11ef 0x100`.

Such a representation of fractions is especially convenient for using them with the scaling primitive `*/` and its variants, thus converting common and decimal fractions into a suitable fixed-point representation. For instance, if we want to represent fractional amounts of Grams by integer amounts of nanograms, we can define some helper words

```
1000000000 constant Gram
{ Gram * } : Gram*
{ Gram swap */r } : Gram*/
```

and then write `2.39 Gram*/` or `17/12 Gram*/` instead of integer literals `2390000000` or `1416666667`.

If one needs to use such Gram literals often, one can introduce a new active prefix word `GR$` as follows:

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

```
  1 'nop
} ::_ GR$
```

makes `GR$3`, `GR$2.39`, and `GR$17/12` equivalent to integer literals `3000000000`, `2390000000`, and `1416666667`, respectively. Such values can be printed in similar form by means of the following words:

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

produces `GR$-17.239 ok`. The above definitions make use of tricks explained in later portions of this document (especially Chapter **4**).

We can also manipulate fractions by themselves by defining suitable "rational arithmetic words":

```
// 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.
```

will output "`31/30 ok`", indicating that $1.7 - 2/3 = 31/30$. Here "`._`" is a variant of "`.`" that does not print a space after the decimal representation of an *Integer*.

2.9 String literals

String literals are introduced by means of the prefix word `"`, which scans the remainder of the line until the next `"` character, and pushes the string thus obtained into the stack as a value of type *String*. For instance, `"Hello, world!"` pushes the corresponding *String* into the stack:

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

2.10 Simple string manipulation

The following words can be used to manipulate strings:

- "⟨*string*⟩" (– S), pushes a *String* literal into the stack.
- ."⟨*string*⟩" (–), prints a constant string into the standard output.
- `type` (S –), prints a *String* S taken from the top of the stack into the standard output.
- `cr` (–), outputs a carriage return (or a newline character) into the standard output.
- `emit` (x –), prints a UTF-8 encoded character with Unicode codepoint given by *Integer* x into the standard output.
- `char` ⟨*string*⟩ (– x), pushes an *Integer* with the Unicode codepoint of the first character of ⟨*string*⟩.
- `bl` (– x), pushes the Unicode codepoint of a space, i.e., 32.
- `space` (–), prints one space, equivalent to `bl emit`.
- `$+` (S S' – $S.S'$), concatenates two strings.
- `$len` (S – x), computes the byte length (not the UTF-8 character length!) of a string.
- `+"`⟨*string*⟩`"` (S – S'), concatenates *String* S with a string literal. Equivalent to `"`⟨*string*⟩`" $+`.
- `word` (x – S), parses a word delimited by the character with the Unicode codepoint x from the remainder of the current input line and pushes the result as a *String*. For instance, `bl word abracadabra type` will print the string "`abracadabra`". If $x = 0$, skips leading spaces, and then scans until the end of the current input line. If $x = 32$, skips leading spaces before parsing the next word.
- `(.)` (x – S), returns the *String* with the decimal representation of *Integer* x.

- `(number)` (S – 0 or x 1 or x y 2), attempts to parse the *String* S as an integer or fractional literal as explained in **2.8**.

For instance, `."*"`, `"*" type`, `42 emit`, and `char * emit` are four different ways to output a single asterisk.

2.11 Boolean expressions, or flags

Fift does not have a separate value type for representing boolean values. Instead, any non-zero *Integer* can be used to represent truth (with -1 being the standard representation), while a zero *Integer* represents falsehood. Comparison primitives normally return -1 to indicate success and 0 otherwise.

Constants `true` and `false` can be used to push these special integers into the stack:

- `true` (– -1), pushes -1 into the stack.
- `false` (– 0), pushes 0 into the stack.

If boolean values are standard (either 0 or -1), they can be manipulated by means of bitwise logical operations `and`, `or`, `xor`, `not` (listed in **2.4**). Otherwise, they must first be reduced to the standard form using `0<>`:

- `0<>` (x – $x \neq 0$), pushes -1 if *Integer* x is non-zero, 0 otherwise.

2.12 Integer comparison operations

Several integer comparison operations can be used to obtain boolean values:

- `<` (x y – ?), checks whether $x < y$ (i.e., pushes -1 if $x < y$, 0 otherwise).
- `>`, `=`, `<>`, `<=`, `>=` (x y – ?), compare x and y and push -1 or 0 depending on the result of the comparison.
- `0<` (x – ?), checks whether $x < 0$ (i.e., pushes -1 if x is negative, 0 otherwise). Equivalent to `0 <`.
- `0>`, `0=`, `0<>`, `0<=`, `0>=` (x – ?), compare x against zero.
- `cmp` (x y – z), pushes 1 if $x > y$, -1 if $x < y$, and 0 if $x = y$.

- `sgn` ($x - y$), pushes 1 if $x > 0$, -1 if $x < 0$, and 0 if $x = 0$. Equivalent to `0 cmp`.

Example:

```
2 3 < .
```

prints "`-1 ok`", because 2 is less than 3.

A more convoluted example:

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

prints "`true false false ok`".

2.13 String comparison operations

Strings can be lexicographically compared by means of the following words:

- `$=` (S S' – ?), returns -1 if strings S and S' are equal, 0 otherwise.
- `$cmp` (S S' – x), returns 0 if strings S and S' are equal, -1 if S is lexicographically less than S', and 1 if S is lexicographically greater than S'.

2.14 Named and unnamed variables

In addition to constants introduced in **2.7**, Fift supports *variables*, which are a more efficient way to represent changeable values. For instance, the last two code fragments of **2.7** could have been written with the aid of variables instead of constants as follows:

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

producing the same output as before:

```
3 9 12 ok
7 49 56 ok
( 3 , 9 ) ok
```

The phrase `variable x` creates a new *Box*, i.e., a memory location that can be used to store exactly one value of any Fift-supported type, and defines `x` as a constant equal to this *Box*:

- `variable` (–), scans a blank-delimited word name S from the remainder of the input, allocates an empty *Box*, and defines a new ordinary word S as a constant, which will push the new *Box* when invoked. Equivalent to `hole constant`.
- `hole` (– p), creates a new *Box* p that does not hold any value. Equivalent to `null box`.
- `box` (x – p), creates a new *Box* containing specified value x. Equivalent to `hole tuck !`.

The value currently stored in a *Box* may be fetched by means of word `@` (pronounced "fetch"), and modified by means of word `!` (pronounced "store"):

- `@` (p – x), fetches the value currently stored in *Box* p.
- `!` (x p –), stores new value x into *Box* p.

Several auxiliary words exist that can modify the current value in a more sophisticated fashion:

- `+!` (x p –), increases the integer value stored in *Box* p by *Integer* x. Equivalent to `tuck @ + swap !`.
- `1+!` (p –), increases the integer value stored in *Box* p by one. Equivalent to `1 swap +!`.
- `0!` (p –), stores *Integer* 0 into *Box* p. Equivalent to `0 swap !`.

In this way we can implement a simple counter:

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

produces

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

After these definitions are in place, we can even forget the definition of `counter` by means of the phrase `forget counter`. Then the only way to access the value of this variable is by means of `reset-counter` and `next-counter`.

Variables are usually created by `variable` with no value, or rather with a *Null* value. If one wishes to create initialized variables, one can use the phrase `box constant`:

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

prints "`18 ok`". One can even define a special defining word for initialized variables, if they are needed often:

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

prints "`18 test ok`".

The variables have so far only one disadvantage compared to the constants: one has to access their current values by means of an auxiliary word `@`. Of course, one can mitigate this by defining a "getter" and a "setter" word for a variable, and use these words to write better-looking code:

```
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
```

prints "`( 3 , 30 ) ( 5 , 56 ) ok`", which are the points $(x, f(x))$ on the graph of $f(x) = x^2 + 5x + 6$ with $x = 3$ and $x = 5$.

Again, if we want to define "getters" for all our variables, we can first define a defining word as explained in **4.8**, and use this word to define both a getter and a setter at the same time:

```
{ 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
```

produces

```
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 Tuples and arrays

Fift also supports *Tuples*, i.e., immutable ordered collections of arbitrary values of stack value types (cf. **2.1**). When a *Tuple* t consists of values x_1, ..., x_n (in that order), we write $t = (x_1, \ldots, x_n)$. The number n is called the *length* of *Tuple* t; it is also denoted by $|t|$. Tuples of length two are also called *pairs*, tuples of length three are *triples*.

- `tuple` ($x_1 \ldots x_n\ n - t$), creates new *Tuple* $t := (x_1, \ldots, x_n)$ from $n \geq 0$ topmost stack values.
- `pair` ($x\ y - t$), creates new pair $t = (x, y)$. Equivalent to `2 tuple`.
- `triple` ($x\ y\ z - t$), creates new triple $t = (x, y, z)$. Equivalent to `3 tuple`.
- `|` ($- t$), creates an empty *Tuple* $t = ()$. Equivalent to `0 tuple`.
- `,` ($t\ x - t'$), appends x to the end of *Tuple* t, and returns the resulting *Tuple* t'.
- `.dump` ($x -$), dumps the topmost stack entry in the same way as `.s` dumps all stack elements.

For instance, both

```
| 2 , 3 , 9 , .dump
```

and

```
2 3 9 triple .dump
```

construct and print triple $(2, 3, 9)$:

```
[ 2 3 9 ] ok
```

Notice that the components of a *Tuple* are not necessarily of the same type, and that a component of a *Tuple* can also be a *Tuple*:

```
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
```

produces

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

Once a *Tuple* has been constructed, we can extract any of its components, or completely unpack the *Tuple* into the stack:

- `untuple` $(t\ n - x_1\ \ldots x_n)$, returns all components of a *Tuple* $t = (x_1, \ldots, x_n)$, but only if its length is equal to n. Otherwise throws an exception.
- `unpair` $(t - x\ y)$, unpacks a pair $t = (x, y)$. Equivalent to `2 untuple`.
- `untriple` $(t - x\ y\ z)$, unpacks a triple $t = (x, y, z)$. Equivalent to `3 untuple`.
- `explode` $(t - x_1 \ldots x_n\ n)$, unpacks a *Tuple* $t = (x_1, \ldots, x_n)$ of unknown length n, and returns that length.
- `count` $(t - n)$, returns the length $n = |t|$ of *Tuple* t.
- `tuple?` $(t - ?)$, checks whether t is a *Tuple*, and returns -1 or 0 accordingly.

- `[]` ($t\ i - x$), returns the $(i+1)$-st component t_{i+1} of *Tuple* t, where $0 \leq i < |t|$.
- `first` ($t - x$), returns the first component of a *Tuple*. Equivalent to `0 []`.
- `second` ($t - x$), returns the second component of a *Tuple*. Equivalent to `1 []`.
- `third` ($t - x$), returns the third component of a *Tuple*. Equivalent to `2 []`.

For instance, we can access individual elements and rows of a matrix:

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

produces

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

Notice that *Tuple*s are somewhat similar to arrays of other programming languages, but are immutable: we cannot change one individual component of a *Tuple*. If we still want to create something like an array, we need a *Tuple* of *Box*es (cf. **2.14**):

- `allot` ($n - t$), creates a *Tuple* that consists of n new empty *Box*es. Equivalent to `| { hole , } rot times`.

For instance,

```
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[]
```

creates an uninitialized array A of length 10, an initialized array B of length 6, and then interchanges some elements of B and prints the first four elements of the resulting B:

```
4 1 1 3  ok
```

2.16 Lists

Lisp-style lists can also be represented in Fift. First of all, two special words are introduced to manipulate values of type *Null*, used to represent the empty list (not to be confused with the empty *Tuple*):

- `null` (– $\perp$), pushes the only value $\perp$ of type *Null*, which is also used to represent an empty list.
- `null?` (x – ?), checks whether x is a *Null*. Can also be used to check whether a list is empty.

After that, `cons` and `uncons` are defined as aliases for `pair` and `unpair`:

- `cons` (h t – l), constructs a list from its head (first element) h and its tail (the list consisting of all remaining elements) t. Equivalent to `pair`.
- `uncons` (l – h t), decomposes a non-empty list into its head and its tail. Equivalent to `unpair`.
- `car` (l – h), returns the head of a list. Equivalent to `first`.
- `cdr` (l – t), returns the tail of a list. Equivalent to `second`.
- `cadr` (l – h'), returns the second element of a list. Equivalent to `cdr car`.
- `list` ($x_1 \ldots x_n$ n – l), constructs a list l of length n with elements x_1, …, x_n, in that order. Equivalent to `null ' cons rot times`.
- `.l` (l –), prints a Lisp-style list l.

For instance,

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

produces

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

Notice that the three-element list (2 3 9) is distinct from the triple $(2, 3, 9)$.

2.17 Atoms

An *Atom* is a simple entity uniquely identified by its name. *Atom*s can be used to represent identifiers, labels, operation names, tags, and stack markers. Fift offers the following words to manipulate *Atom*s:

- `(atom)` ($S\ x - a\ -1$ or 0), returns the only *Atom* a with the name given by *String* S. If there is no such *Atom* yet, either creates it (if *Integer* x is non-zero) or returns a single zero to indicate failure (if x is zero).
- `atom` ($S - a$), returns the only *Atom* a with the name S, creating such an atom if necessary. Equivalent to `true (atom) drop`.
- `‘`⟨*word*⟩ ($-\ a$), introduces an *Atom* literal, equal to the only *Atom* with the name equal to ⟨*word*⟩. Equivalent to `"`⟨*word*⟩`" atom`.
- `anon` ($-\ a$), creates a new unique anonymous *Atom*.
- `atom?` (u – ?), checks whether u is an *Atom*.
- `eq?` ($u\ v$ – ?), checks whether u and v are equal *Integer*s, *Atom*s, or *Null*s. If they are not equal, or if they are of different types, or not of one of the types listed, returns zero.

For instance,

```
‘+ 2 ‘* 3 4 3 list 3 list .l
```

creates and prints the list

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

which is the Lisp-style representation of arithmetical expression $2+3\cdot 4$. An interpreter for such expressions might use `eq?` to check the operation sign (cf. **3.5** for an explanation of recursive functions in 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
```

prints

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

If we load `Lisp.fif` to enable Lisp-style list syntax, we can enter

```
"Lisp.fif" include
( `+ 2 ( `* 3 4 ) ) dup .l cr eval . cr
```

with the same result as before. The word `(`, defined in `Lisp.fif`, uses an anonymous *Atom* created by `anon` to mark the current stack position, and then `)` builds a list from several top stack entries, scanning the stack until the anonymous *Atom* marker is found:

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

2.18 Command line arguments in script mode

The Fift interpreter can be invoked in *script mode* by passing `-s` as a command line option. In this mode, all further command line arguments are not scanned for Fift startup command line options. Rather, the next argument after `-s` is used as the filename of the Fift source file, and all further command line arguments are passed to the Fift program by means of special words `$`n and `$#`:

- `$#` (– x), pushes the total number of command-line arguments passed to the Fift program.
- `$`n (– S), pushes the n-th command-line argument as a *String* S. For instance, `$0` pushes the name of the script being executed, `$1` the first command line argument, and so on.
- `$()` (x – S), pushes the x-th command-line argument similarly to `$`n, but with *Integer* x taken from the stack.

Additionally, if the very first line of a Fift source file begins with the two characters "`#!`", this line is ignored. In this way simple Fift scripts can be written in a *ix system. For instance, if

```
#!/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
```

is saved into a file `cmdline.fif` in the current directory, and its execution bit is set (e.g., by `chmod 755 cmdline.fif`), then it can be invoked from the shell or any other program, provided the Fift interpreter is installed as `/usr/bin/fift`, and its standard library `Fift.fif` is installed as `/usr/lib/fift/Fift.fif`:

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

prints

```
-60
```

when invoked from a *ix shell such as the Bourne–again shell (Bash).

3 Blocks, loops, and conditionals

Similarly to the arithmetic operations, the execution flow in Fift is controlled by stack-based primitives. This leads to an inversion typical of reverse Polish notation and stack-based arithmetic: one first pushes a block representing a conditional branch or the body of a loop into the stack, and then invokes a conditional or iterated execution primitive. In this respect, Fift is more similar to PostScript than to Forth.

3.1 Defining and executing blocks

A block is normally defined using the special words "`{`" and "`}`". Roughly speaking, all words listed between `{` and `}` constitute the body of a new block, which is pushed into the stack as a value of type *WordDef*. A block may be stored as a definition of a new Fift word by means of the defining word "`:`" as explained in **2.6**, or executed by means of the word `execute`:

```
17 { 2 * } execute .
```

prints "`34 ok`", being essentially equivalent to "`17 2 * .`". A slightly more convoluted example:

```
{ 2 * } 17 over execute swap execute .
```

applies "anonymous function" $x \mapsto 2x$ twice to 17, and prints the result $2 \cdot (2 \cdot 17) = 68$. In this way a block is an execution token, which can be duplicated, stored into a constant, used to define a new word, or executed.

The word `'` recovers the current definition of a word. Namely, the construct `'` ⟨*word-name*⟩ pushes the execution token equivalent to the current definition of the word ⟨*word-name*⟩. For instance,

```
' dup execute
```

is equivalent to `dup`, and

```
' dup : duplicate
```

defines `duplicate` as a synonym for (the current definition of) `dup`.

Alternatively, we can duplicate a block to define two new words with the same definition:

```
{ dup * }
dup : square : **2
```

defines both `square` and `**2` to be equivalent to `dup *`.

3.2 Conditional execution of blocks

Conditional execution of blocks is achieved using the words `if`, `ifnot`, and `cond`:

- `if` (x e –), executes e (which must be an execution token, i.e., a *WordDef*),[5] but only if *Integer* x is non-zero.
- `ifnot` (x e –), executes execution token e, but only if *Integer* x is zero.
- `cond` (x e e' –), if *Integer* x is non-zero, executes e, otherwise executes e'.

For instance, the last example in **2.12** can be more conveniently rewritten as

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

still resulting in "`true false false ok`".

Notice that blocks can be arbitrarily nested, as already shown in the previous example. One can write, for example,

```
{ ?dup
  { 0<
    { ."negative " }
    { ."positive " }
    cond
  }
  { ."zero " }
  cond
} : chksign
-17 chksign
```

to obtain "`negative ok`", because −17 is negative.

[5] A *WordDef* is more general than a *WordList*. For instance, the definition of the primitive + is a *WordDef*, but not a *WordList*, because + is not defined in terms of other Fift words.

3.3 Simple loops

The simplest loops are implemented by `times`:

- `times` (e n –), executes e exactly n times, if $n \geq 0$. If n is negative, throws an exception.

For instance,

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

computes and prints 10^{70}.

We can use this kind of loop to implement a simple factorial function:

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

prints "`120 ok`", because $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$.

This loop can be modified to compute Fibonacci numbers instead:

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

computes the sixth Fibonacci number $F_6 = 13$.

3.4 Loops with an exit condition

More sophisticated loops can be created with the aid of `until` and `while`:

- `until` (e –), executes e, then removes the top-of-stack integer and checks whether it is zero. If it is, then begins a new iteration of the loop by executing e. Otherwise exits the loop.

- `while` (e e' –), executes e, then removes and checks the top-of-stack integer. If it is zero, exits the loop. Otherwise executes e', then begins a new loop iteration by executing e and checking the exit condition afterwards.

For instance, we can compute the first two Fibonacci numbers greater than 1000:

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

prints "`1597 2584 ok`".

We can use this word to compute the first 70 decimal digits of the golden ratio $\phi = (1 + \sqrt{5})/2 \approx 1.61803$:

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

prints "161803 . . . 2604 `ok`".

3.5 Recursion

Notice that, whenever a word is mentioned inside a `{ ...}` block, the current (compile-time) definition is included in the *WordList* being created. In this way we can refer to the previous definition of a word while defining a new version of it:

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

produces "`number 2 number 3 5 number 8 ok`". Notice that `print-sum` continues to use the original definition of "`.`", but `print-next` already uses the modified "`.`".

This feature may be convenient on some occasions, but it prevents us from introducing recursive definitions in the most straightforward fashion. For instance, the classical recursive definition of the factorial

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

will fail to compile, because `fact` happens to be an undefined word when the definition is compiled.

A simple way around this obstacle is to use the word `@'` (cf. **4.6**) that looks up the current definition of the next word during the execution time and then executes it, similarly to what we already did in **2.7**:

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

produces "`120 ok`", as expected.

However, this solution is rather inefficient, because it uses a dictionary lookup each time `fact` is recursively executed. We can avoid this dictionary lookup by using variables (cf. **2.14** and **2.7**):

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

This somewhat longer definition of the factorial avoids dictionary lookups at execution time by introducing a special variable `'fact` to hold the final definition of the factorial.[6] Then `fact` is defined to execute whatever *WordDef* is currently stored in `'fact`, and once the body of the recursive definition of the factorial is constructed, it is stored into this variable by means of the phrase `'fact !`, which replaces the more customary phrase `: fact`.

We could rewrite the above definition by using special "getter" and "setter" words for vector variable `'fact` as we did for variables in **2.14**:

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

If we need to introduce a lot of recursive and mutually-recursive definitions, we might first introduce a custom defining word (cf. **4.8**) for simultaneously defining both the "getter" and the "setter" words for anonymous vector variables, similarly to what we did in **2.14**:

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

The first three lines of this fragment define `fact` and `:fact` essentially in the same way they had been defined in the first four lines of the previous fragment.

If we wish to make `fact` unchangeable in the future, we can add a `forget :fact` line once the definition of the factorial is complete:

[6] Variables that hold a *WordDef* to be `executed` later are called *vector variables*. The process of replacing `fact` with `'fact @ execute`, where `'fact` is a vector variable, is called *vectorization*.

```
{ 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 .
```

Alternatively, we can modify the definition of `vector-set` in such a way that `:fact` would forget itself once it is invoked:

```
{ 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 .
```

However, some vector variables must be modified more than once, for instance, to modify the behavior of the comparison word `less` in a merge sort algorithm:

```
{ 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
```

producing the following output:

```
(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 Throwing exceptions

Two built-in words are used to throw exceptions:

- `abort` (S –), throws an exception with an error message taken from *String* S.
- `abort"`⟨*message*⟩`"` (x –), throws an exception with the error message ⟨*message*⟩ if x is a non-zero integer.

The exception thrown by these words is represented by the C++ exception `fift::IntError` with its value equal to the specified string. It is normally handled within the Fift interpreter itself by aborting all execution up to the top level and printing a message with the name of the source file being interpreted, the line number, the currently interpreted word, and the specified error message. For instance:

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

prints "`safe/: Division by zero`", without the usual "`ok`". The stack is cleared in the process.

Incidentally, when the Fift interpreter encounters an unknown word that cannot be parsed as an integer literal, an exception with the message "`-?`" is thrown, with the effect indicated above, including the stack being cleared.

4 Dictionary, interpreter, and compiler

In this chapter we present several specific Fift words for dictionary manipulation and compiler control. The "compiler" is the part of the Fift interpreter that builds lists of word references (represented by *WordList* stack values) from word names; it is activated by the primitive "`{`" employed for defining blocks as explained in **2.6** and **3.1**.

Most of the information included in this chapter is rather sophisticated and may be skipped during a first reading. However, the techniques described here are heavily employed by the Fift assembler, used to compile TVM code. Therefore, this section is indispensable if one wishes to understand the current implementation of the Fift assembler.

4.1 The state of the Fift interpreter

The state of the Fift interpreter is controlled by an internal integer variable called `state`, currently unavailable from Fift itself. When `state` is zero, all words parsed from the input (i.e., the Fift source file or the standard input in the interactive mode) are looked up in the dictionary and immediately executed afterwards. When `state` is positive, the words found in the dictionary are not executed. Instead, they (or rather the references to their current definitions) are *compiled*, i.e., added to the end of the *WordList* being constructed.

Typically, `state` equals the number of the currently open blocks. For instance, after interpreting "`{ 0= { ."zero"`" the `state` variable will be equal to two, because there are two nested blocks. The *WordList* being constructed is kept at the top of the stack.

The primitive "`{`" simply pushes a new empty *WordList* into the stack, and increases `state` by one. The primitive "`}`" throws an exception if `state` is already zero; otherwise it decreases `state` by one, and leaves the resulting

WordList in the stack, representing the block just constructed.[7] After that, if the resulting value of `state` is non-zero, the new block is compiled as a literal (unnamed constant) into the encompassing block.

4.2 Active and ordinary words

All dictionary words have a special flag indicating whether they are *active* words or *ordinary* words. By default, all words are ordinary. In particular, all words defined with the aid of "`:`" and `constant` are ordinary.

When the Fift interpreter finds a word definition in the dictionary, it checks whether it is an ordinary word. If it is, then the current word definition is either executed (if `state` is zero) or "compiled" (if `state` is greater than zero) as explained in **4.1**.

On the other hand, if the word is active, then it is always executed, even if `state` is positive. An active word is expected to leave some values $x_1 \ldots x_n$ n e in the stack, where $n \geq 0$ is an integer, $x_1 \ldots x_n$ are n values of arbitrary types, and e is an execution token (a value of type *WordDef*). After that, the interpreter performs different actions depending on `state`: if `state` is zero, then n is discarded and e is executed, as if a `nip execute` phrase were found. If `state` is non-zero, then this collection is "compiled" in the current *WordList* (located immediately below x_1 in the stack) in the same way as if the `(compile)` primitive were invoked. This compilation amounts to adding some code to the end of the current *WordList* that would push x_1, ..., x_n into the stack when invoked, and then adding a reference to e (representing a delayed execution of e). If e is equal to the special value `'nop`, representing an execution token that does nothing when executed, then this last step is omitted.

4.3 Compiling literals

When the Fift interpreter encounters a word that is absent from the dictionary, it invokes the primitive `(number)` to attempt to parse it as an integer or fractional literal. If this attempt succeeds, then the special value `'nop` is pushed, and the interpretation proceeds in the same way as if an active word

[7] The word `}` also transforms this *WordList* into a *WordDef*, which has a different type tag and therefore is a different Fift value, even if the same underlying C++ object is used by the C++ implementation.

were encountered. In other words, if `state` is zero, then the literal is simply left in the stack; otherwise, `(compile)` is invoked to modify the current *WordList* so that it would push the literal when executed.

4.4 Defining new active words

New active words are defined similarly to new ordinary words, but using "`::`" instead of "`:`". For instance,

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

defines the active word `say`, which scans the next blank-separated word after itself and compiles it as a literal along with a reference to the current definition of `type` into the current *WordList* (if `state` is non-zero, i.e., if the Fift interpreter is compiling a block). When invoked, this addition to the block will push the stored string into the stack and execute `type`, thus printing the next word after `say`. On the other hand, if `state` is zero, then these two actions are performed by the Fift interpreter immediately. In this way,

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

will print "`hello3 ok`", while

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

will print "`hello3 hello6 ok`".

Of course, a block may be used to represent the required action instead of `' type`. For instance, if we want a version of `say` that prints a space after the stored word, we can write

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

to obtain "`hello 3 hello 6 ok`".

Incidentally, the words `"` (introducing a string literal) and `."` (printing a string literal) can be defined as follows:

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

The new defining word "`::_`" defines an active *prefix* word, i.e., an active word that does not require a space afterwards.

4.5 Defining words and dictionary manipulation

Defining words are words that define new words in the Fift dictionary. For instance, "`:`", "`::_`", and `constant` are defining words. All of these defining words might have been defined using the primitive `(create)`; in fact, the user can introduce custom defining words if so desired. Let us list some defining words and dictionary manipulation words:

- `create` ⟨*word-name*⟩ (e –), defines a new ordinary word with the name equal to the next word scanned from the input, using *WordDef* e as its definition. If the word already exists, it is tacitly redefined.

- `(create)` (e S x –), creates a new word with the name equal to *String* S and definition equal to *WordDef* e, using flags passed in *Integer* $0 \le x \le 3$. If bit +1 is set in x, creates an active word; if bit +2 is set in x, creates a prefix word.

- `:` ⟨*word-name*⟩ (e –), defines a new ordinary word ⟨*word-name*⟩ in the dictionary using *WordDef* e as its definition. If the specified word is already present in the dictionary, it is tacitly redefined.

- `forget` ⟨*word-name*⟩ (–), forgets (removes from the dictionary) the definition of the specified word.

- `(forget)` (S –), forgets the word with the name specified in *String* S. If the word is not found, throws an exception.

- `:_` ⟨*word-name*⟩ (e –), defines a new ordinary *prefix* word ⟨*word-name*⟩, meaning that a blank or an end-of-line character is not required by the Fift input parser after the word name. In all other respects it is similar to "`:`".

- `::` ⟨*word-name*⟩ (e –), defines a new *active* word ⟨*word-name*⟩ in the dictionary using *WordDef* e as its definition. If the specified word is already present in the dictionary, it is tacitly redefined.

- `::_` ⟨*word-name*⟩ (e –), defines a new active *prefix* word ⟨*word-name*⟩, meaning that a blank or an end-of-line character is not required by the Fift input parser after the word name. In all other respects it is similar to "`::`".

- `constant` ⟨*word-name*⟩ (x –), defines a new ordinary word ⟨*word-name*⟩ that would push the given value x when invoked.
- `2constant` ⟨*word-name*⟩ (x y –), defines a new ordinary word named ⟨*word-name*⟩ that would push the given values x and y (in that order) when invoked.
- `=:` ⟨*word-name*⟩ (x –), defines a new ordinary word ⟨*word-name*⟩ that would push the given value x when invoked, similarly to `constant`, but works inside blocks and colon definitions.
- `2=:` ⟨*word-name*⟩ (x y –), defines a new ordinary word ⟨*word-name*⟩ that would push the given values x and y (in that order) when invoked, similarly to `2constant`, but works inside blocks and colon definitions.

Notice that most of the above words might have been defined in terms of `(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 Dictionary lookup

The following words can be used to look up words in the dictionary:

- `'` ⟨*word-name*⟩ (– e), pushes the definition of the word ⟨*word-name*⟩, recovered at the compile time. If the indicated word is not found, throws an exception. Notice that `'` ⟨*word-name*⟩ `execute` is always equivalent to ⟨*word-name*⟩ for ordinary words, but not for active words.
- `nop` (–), does nothing.
- `'nop` (– e), pushes the default definition of `nop`—an execution token that does nothing when executed.
- `find` (S – e -1 or e 1 or 0), looks up *String* S in the dictionary and returns its definition as a *WordDef* e if found, followed by -1 for ordinary words or 1 for active words. Otherwise pushes 0.

- `(')` ⟨*word-name*⟩ (– *e*), similar to `'`, but returns the definition of the specified word at execution time, performing a dictionary lookup each time it is invoked. May be used to recover current values of constants inside word definitions and other blocks by using the phrase `(')` ⟨*word-name*⟩ `execute`.

- `@'` ⟨*word-name*⟩ (– *e*), similar to `(')`, but recovers the definition of the specified word at execution time, performing a dictionary lookup each time it is invoked, and then executes this definition. May be used to recover current values of constants inside word definitions and other blocks by using the phrase `@'` ⟨*word-name*⟩, equivalent to `(')` ⟨*word-name*⟩ `execute`, cf. **2.7**.

- `[compile]` ⟨*word-name*⟩ (–), compiles ⟨*word-name*⟩ as if it were an ordinary word, even if it is active. Essentially equivalent to `'` ⟨*word-name*⟩ `execute`.

- `words` (–), prints the names of all words currently defined in the dictionary.

4.7 Creating and manipulating word lists

In the Fift stack, lists of references to word definitions and literals, to be used as blocks or word definitions, are represented by the values of the type *WordList*. Some words for manipulating *WordList*s include:

- `{` (– *l*), an active word that increases internal variable `state` by one and pushes a new empty *WordList* into the stack.

- `}` (*l* – *e*), an active word that transforms a *WordList* *l* into a *WordDef* (an execution token) *e*, thus making all further modifications of *l* impossible, and decreases internal variable `state` by one and pushes the integer 1, followed by a `'nop`. The net effect is to transform the constructed *WordList* into an execution token and push this execution token into the stack, either immediately or during the execution of an outer block.

- `({)` (– *l*), pushes an empty *WordList* into the stack.

- `(})` (*l* – *e*), transforms a *WordList* into an execution token, making all further modifications impossible.

- `(compile)` ($l\ x_1 \ldots x_n\ n\ e - l'$), extends *WordList* l so that it would push $0 \leq n \leq 255$ values $x_1, \ldots, x_n$ into the stack and execute the execution token e when invoked, where $0 \leq n \leq 255$ is an *Integer*. If e is equal to the special value `'nop`, the last step is omitted.

- `does` ($x_1 \ldots x_n\ n\ e - e'$), creates a new execution token e' that would push n values $x_1, \ldots, x_n$ into the stack and then execute e. It is roughly equivalent to a combination of `({)`, `(compile)`, and `(})`.

4.8 Custom defining words

The word `does` is actually defined in terms of simpler words:

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

It is especially useful for defining custom defining words. For instance, `constant` and `2constant` may be defined with the aid of `does` and `create`:

```
{ 1 'nop does create } : constant
{ 2 'nop does create } : 2constant
```

Of course, non-trivial actions may be performed by the words defined by means of such custom defining words. For instance,

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

will print "`hello unknown error ok`", because `hello` is defined by means of a custom defining word `says` to print "`hello`" whenever invoked, and similarly `error` prints "`unknown error`" when invoked. The above definitions are essentially equivalent to

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

However, custom defining words may perform more sophisticated actions when invoked, and preprocess their arguments at compile time. For instance, the message can be computed in a non-trivial fashion:

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

defines word `hw`, which prints "Hello, world!" when invoked. The string with this message is computed once at compile time (when `says` is invoked), not at execution time (when `hw` is invoked).

5 Cell manipulation

We have discussed the basic Fift primitives not related to TVM or the TON Blockchain so far. Now we will turn to TON-specific words, used to manipulate *Cell*s.

5.1 Slice literals

Recall that a (TVM) *Cell* consists of at most 1023 data bits and at most four references to other *Cell*s, a *Slice* is a read-only view of a portion of a *Cell*, and a *Builder* is used to create new *Cell*s. Fift has special provisions for defining *Slice* literals (i.e., unnamed constants), which can also be transformed into *Cells* if necessary.

Slice literals are introduced by means of active prefix words `x{` and `b{`:

- `b{`⟨*binary-data*⟩`}` (– *s*), creates a *Slice* *s* that contains no references and up to 1023 data bits specified in ⟨*binary-data*⟩, which must be a string consisting only of the characters '`0`' and '`1`'.

- `x{`⟨*hex-data*⟩`}` (– *s*), creates a *Slice* *s* that contains no references and up to 1023 data bits specified in ⟨*hex-data*⟩. More precisely, each hex digit from ⟨*hex-data*⟩ is transformed into four binary digits in the usual fashion. After that, if the last character of ⟨*hex-data*⟩ is an underscore `_`, then all trailing binary zeroes and the binary one immediately preceding them are removed from the resulting binary string (cf. [4, 1.0] for more details).

In this way, `b{00011101}` and `x{1d}` both push the same *Slice* consisting of eight data bits and no references. Similarly, `b{111010}` and `x{EA_}` push the same *Slice* consisting of six data bits. An empty *Slice* can be represented as `b{}` or `x{}`.

If one wishes to define constant *Slice*s with some *Cell* references, the following words might be used:

- `|_` ($s\ s' - s''$), given two *Slice*s s and s', creates a new *Slice* s'', which is obtained from s by appending a new reference to a *Cell* containing s'.
- `|+` ($s\ s' - s''$), concatenates two *Slice*s s and s'. This means that the data bits of the new *Slice* s'' are obtained by concatenating the data bits of s and s', and the list of *Cell* references of s'' is constructed similarly by concatenating the corresponding lists for s and s'.

5.2 Builder primitives

The following words can be used to manipulate *Builder*s, which can later be used to construct new *Cell*s:

- `<b` ($- b$), creates a new empty *Builder*.
- `b>` ($b - c$), transforms a *Builder* b into a new *Cell* c containing the same data as b.
- `i,` ($b\ x\ y - b'$), appends the big-endian binary representation of a signed y-bit integer x to *Builder* b, where $0 \le y \le 257$. If there is not enough room in b (i.e., if b already contains more than $1023 - y$ data bits), or if *Integer* x does not fit into y bits, an exception is thrown.
- `u,` ($b\ x\ y - b'$), appends the big-endian binary representation of an unsigned y-bit integer x to *Builder* b, where $0 \le y \le 256$. If the operation is impossible, an exception is thrown.
- `ref,` ($b\ c - b'$), appends to *Builder* b a reference to *Cell* c. If b already contains four references, an exception is thrown.
- `s,` ($b\ s - b'$), appends data bits and references taken from *Slice* s to *Builder* b.
- `sr,` ($b\ s - b'$), constructs a new *Cell* containing all data and references from *Slice* s, and appends a reference to this cell to *Builder* b. Equivalent to `<b swap s, b> ref,`.
- `$,` ($b\ S - b'$), appends *String* S to *Builder* b. The string is interpreted as a binary string of length $8n$, where n is the number of bytes in the UTF-8 representation of S.

- `B,` ($b\ B - b'$), appends *Bytes* B to *Builder* b.
- `b+` ($b\ b' - b''$), concatenates two *Builders* b and b'.
- `bbits` ($b - x$), returns the number of data bits already stored in *Builder* b. The result x is an *Integer* in the range $0 \ldots 1023$.
- `brefs` ($b - x$), returns the number of references already stored in *Builder* b. The result x is an *Integer* in the range $0 \ldots 4$.
- `bbitrefs` ($b - x\ y$), returns both the number of data bits x and the number of references y already stored in *Builder* b.
- `brembits` ($b - x$), returns the maximum number of additional data bits that can be stored in *Builder* b. Equivalent to `bbits 1023 swap -`.
- `bremrefs` ($b - x$), returns the maximum number of additional cell references that can be stored in *Builder* b.
- `brembitrefs` ($b - x\ y$), returns both the maximum number of additional data bits $0 \le x \le 1023$ and the maximum number of additional cell references $0 \le y \le 4$ that can be stored in *Builder* b.

The resulting *Builder* may be inspected by means of the non-destructive stack dump primitive `.s`, or by the phrase `b> <s csr.`. For instance:

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

outputs

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

One can observe that `.s` dumps the internal representation of a *Builder*, with two tag bytes at the beginning (usually equal to the number of cell references already stored in the *Builder*, and to twice the number of complete bytes stored in the *Builder*, increased by one if an incomplete byte is present). On

the other hand, `csr.` dumps a *Slice* (constructed from a *Cell* by `<s`, cf. **5.3**) in a form similar to that used by `x{` to define *Slice* literals (cf. **5.1**).

Incidentally, the word `mkTest` shown above (without the `.s` in its definition) corresponds to the TL-B constructor

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

and may be used to serialize values of this TL-B type.

5.3 Slice primitives

The following words can be used to manipulate values of the type *Slice*, which represents a read-only view of a portion of a *Cell*. In this way data previously stored into a *Cell* may be deserialized, by first transforming a *Cell* into a *Slice*, and then extracting the required data from this *Slice* step-by-step.

- `<s` (c – s), transforms a *Cell* c into a *Slice* s containing the same data. It usually marks the start of the deserialization of a cell.

- `s>` (s –), throws an exception if *Slice* s is non-empty. It usually marks the end of the deserialization of a cell, checking whether there are any unprocessed data bits or references left.

- `i@` (s x – y), fetches a signed big-endian x-bit integer from the first x bits of *Slice* s. If s contains less than x data bits, an exception is thrown.

- `i@+` (s x – y s'), fetches a signed big-endian x-bit integer from the first x bits of *Slice* s similarly to `i@`, but returns the remainder of s as well.

- `i@?` (s x – y -1 or 0), fetches a signed integer from a *Slice* similarly to `i@`, but pushes integer -1 afterwards on success. If there are less than x bits left in s, pushes integer 0 to indicate failure.

- `i@?+` (s x – y s' -1 or s 0), fetches a signed integer from *Slice* s and computes the remainder of this *Slice* similarly to `i@+`, but pushes -1 afterwards to indicate success. On failure, pushes the unchanged *Slice* s and 0 to indicate failure.

- `u@`, `u@+`, `u@?`, `u@?+`, counterparts of `i@`, `i@+`, `i@?`, `i@?+` for deserializing unsigned integers.

- `B@` ($s\ x - B$), fetches first x bytes (i.e., $8x$ bits) from *Slice* s, and returns them as a *Bytes* value B. If there are not enough data bits in s, throws an exception.
- `B@+` ($s\ x - B\ s'$), similar to `B@`, but returns the remainder of *Slice* s as well.
- `B@?` ($s\ x - B\ -1$ or 0), similar to `B@`, but uses a flag to indicate failure instead of throwing an exception.
- `B@?+` ($s\ x - B\ s'\ -1$ or $s\ 0$), similar to `B@+`, but uses a flag to indicate failure instead of throwing an exception.
- `$@`, `$@+`, `$@?`, `$@?+`, counterparts of `B@`, `B@+`, `B@?`, `B@?+`, returning the result as a (UTF-8) *String* instead of a *Bytes* value. These primitives do not check whether the byte sequence read is a valid UTF-8 string.
- `ref@` ($s - c$), fetches the first reference from *Slice* s and returns the *Cell* c referred to. If there are no references left, throws an exception.
- `ref@+` ($s - s'\ c$), similar to `ref@`, but returns the remainder of s as well.
- `ref@?` ($s - c\ -1$ or 0), similar to `ref@`, but uses a flag to indicate failure instead of throwing an exception.
- `ref@?+` ($s - s'\ c\ -1$ or $s\ 0$), similar to `ref@+`, but uses a flag to indicate failure instead of throwing an exception.
- `empty?` ($s - ?$), checks whether a *Slice* is empty (i.e., has no data bits and no references left), and returns -1 or 0 accordingly.
- `remaining` ($s - x\ y$), returns both the number of data bits x and the number of cell references y remaining in *Slice* s.
- `sbits` ($s - x$), returns the number of data bits x remaining in *Slice* s.
- `srefs` ($s - x$), returns the number of cell references x remaining in *Slice* s.
- `sbitrefs` ($s - x\ y$), returns both the number of data bits x and the number of cell references y remaining in *Slice* s. Equivalent to `remaining`.

- `$>s` (S – s), transforms *String* S into a *Slice*. Equivalent to `<b swap $, b> <s`.
- `s>c` (s – c), creates a *Cell* c directly from a *Slice* s. Equivalent to `<b swap s, b>`.
- `csr.` (s –), recursively prints a *Slice* s. On the first line, the data bits of s are displayed in hexadecimal form embedded into an `x{...}` construct similar to the one used for *Slice* literals (cf. **5.1**). On the next lines, the cells referred to by s are printed with larger indentation.

For instance, values of the TL-B type `Test` discussed in **5.2**

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

may be deserialized as follows:

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

prints "`17239 -1000000001 ok`" as expected.

Of course, if one needs to check constructor tags often, a helper word can be defined for this purpose:

```
{ 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 . .
```

We can do even better with the aid of active prefix words (cf. **4.2** and **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 . .
```

A shorter but less efficient solution would be to reuse the previously defined `tag?`:

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

first outputs "`x{F55AA}`", and then throws an exception with the message "constructor tag mismatch".

5.4 Cell hash operations

There are few words that operate on *Cell*s directly. The most important of them computes the *(*SHA256*-based) representation hash* of a given cell (cf. [4, 3.1]), which can be roughly described as the SHA256 hash of the cell's data bits concatenated with recursively computed hashes of the cells referred to by this cell:

- `hash` ($c - B$), computes the SHA256-based representation hash of *Cell* c (cf. [4, 3.1]), which unambiguously defines c and all its descendants (provided there are no collisions for SHA256). The result is returned as a *Bytes* value consisting of exactly 32 bytes.

- `shash` ($s - B$), computes the SHA256-based representation hash of a *Slice* by first transforming it into a cell. Equivalent to `s>c hash`.

5.5 Bag-of-cells operations

A *bag of cells* is a collection of one or more cells along with all their descendants. It can usually be serialized into a sequence of bytes (represented by a *Bytes* value in Fift) and then saved into a file or transferred by network. Afterwards, it can be deserialized to recover the original cells. The TON Blockchain systematically represents different data structures (including the TON Blockchain blocks) as a tree of cells according to a certain TL-B scheme (cf. [5], where this scheme is explained in detail), and then these trees of cells are routinely imported into bags of cells and serialized into binary files.

Fift words for manipulating bags of cells include:

- `B>boc` (B – c), deserializes a "standard" bag of cells (i.e., a bag of cells with exactly one root cell) represented by *Bytes* B, and returns the root *Cell* c.

- `boc+>B` (c x – B), creates and serializes a "standard" bag of cells, containing one root *Cell* c along with all its descendants. An *Integer* parameter $0 \leq x \leq 31$ is used to pass flags indicating the additional options for bag-of-cells serialization, with individual bits having the following effect:

  – +1 enables bag-of-cells index creation (useful for lazy deserialization of large bags of cells).

  – +2 includes the CRC32-C of all data into the serialization (useful for checking data integrity).

  – +4 explicitly stores the hash of the root cell into the serialization (so that it can be quickly recovered afterwards without a complete deserialization).

  – +8 stores hashes of some intermediate (non-leaf) cells (useful for lazy deserialization of large bags of cells).

  – +16 stores cell cache bits to control caching of deserialized cells.

  Typical values of x are $x = 0$ or $x = 2$ for very small bags of cells (e.g., TON Blockchain external messages) and $x = 31$ for large bags of cells (e.g., TON Blockchain blocks).

- `boc>B` (c – B), serializes a small "standard" bag of cells with root *Cell* c and all its descendants. Equivalent to `0 boc+>B`.

For instance, the cell created in **5.2** with a value of TL-B `Test` type may be serialized as follows:

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

outputs "`B5EE9C7201040101000000000900000E4A4357C46535FF ok`". Here `Bx.` is the word that prints the hexadecimal representation of a *Bytes* value.

5.6 Binary file I/O and Bytes manipulation

The following words can be used to manipulate values of type *Bytes* (arbitrary byte sequences) and to read them from or write them into binary files:

- `B{`⟨*hex-digits*⟩`}` (– *B*), pushes a *Bytes* literal containing data represented by an even number of hexadecimal digits.
- `Bx.` (*B* –), prints the hexadecimal representation of a *Bytes* value. Each byte is represented by exactly two uppercase hexadecimal digits.
- `file>B` (*S* – *B*), reads the (binary) file with the name specified in *String S* and returns its contents as a *Bytes* value. If the file does not exist, an exception is thrown.
- `B>file` (*B S* –), creates a new (binary) file with the name specified in *String S* and writes data from *Bytes B* into the new file. If the specified file already exists, it is overwritten.
- `file-exists?` (*S* – ?), checks whether the file with the name specified in *String S* exists.

For instance, the bag of cells created in the example in **5.5** can be saved to disk as `sample.boc` as follows:

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

It can be loaded and deserialized afterwards (even in another Fift session) by means of `file>B` and `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 . .
```

prints "`17239 -1000000001 ok`".

Additionally, there are several words for directly packing (serializing) data into *Bytes* values, and unpacking (deserializing) them afterwards. They can be combined with `B>file` and `file>B` to save data directly into binary files, and load them afterwards.

- `Blen` (B – x), returns the length of a *Bytes* value B in bytes.
- `Bhash` (B – B'), computes the SHA256 hash of a *Bytes* value. The hash is returned as a 32-byte *Bytes* value.
- `B=` (B B' – ?), checks whether two *Bytes* sequences are equal.
- `Bcmp` (B B' – x), lexicographically compares two *Bytes* sequences, and returns -1, 0, or 1, depending on the comparison result.
- `B>i@` (B x – y), deserializes the first $x/8$ bytes of a *Bytes* value B as a signed big-endian x-bit *Integer* y.
- `B>i@+` (B x – B' y), deserializes the first $x/8$ bytes of B as a signed big-endian x-bit *Integer* y similarly to `B>i@`, but also returns the remaining bytes of B.
- `B>u@`, `B>u@+`, variants of `B>i@` and `B>i@+` deserializing unsigned integers.
- `B>Li@`, `B>Li@+`, `B>Lu@`, `B>Lu@+`, little-endian variants of `B>i@`, `B>i@+`, `B>u@`, `B>u@+`.
- `B|` (B x – B' B''), cuts the first x bytes from a *Bytes* value B, and returns both the first x bytes (B') and the remainder (B'') as new *Bytes* values.
- `i>B` (x y – B), stores a signed big-endian y-bit *Integer* x into a *Bytes* value B consisting of exactly $y/8$ bytes. Integer y must be a multiple of eight in the range $0 \ldots 256$.
- `u>B` (x y – B), stores an unsigned big-endian y-bit *Integer* x into a *Bytes* value B consisting of exactly $y/8$ bytes, similarly to `i>B`.
- `Li>B`, `Lu>B`, little-endian variants of `i>B` and `u>B`.
- `B+` (B' B'' – B), concatenates two *Bytes* sequences.

6 TON-specific operations

This chapter describes the TON-specific Fift words, with the exception of the words used for *Cell* manipulation, already discussed in the previous chapter.

6.1 Ed25519 cryptography

Fift offers an interface to the same Ed25519 elliptic curve cryptography used by TVM, described in Appendix A of [5]:

- `now` (– x), returns the current Unixtime as an *Integer*.
- `newkeypair` (– B B'), generates a new Ed25519 private/public key pair, and returns both the private key B and the public key B' as 32-byte *Bytes* values. The quality of the keys is good enough for testing purposes. Real applications must feed enough entropy into OpenSSL PRNG before generating Ed25519 keypairs.
- `priv>pub` (B – B'), computes the public key corresponding to a private Ed25519 key. Both the public key B' and the private key B are represented by 32-byte *Bytes* values.
- `ed25519_sign` (B B' – B''), signs data B with the Ed25519 private key B' (a 32-byte *Bytes* value) and returns the signature as a 64-byte *Bytes* value B''.
- `ed25519_sign_uint` (x B' – B''), converts a big-endian unsigned 256-bit integer x into a 32-byte sequence and signs it using the Ed25519 private key B' similarly to `ed25519_sign`. Equivalent to `swap 256 u>B swap ed25519_sign`. The integer x to be signed is typically computed as the hash of some data.
- `ed25519_chksign` (B B' B'' – ?), checks whether B' is a valid Ed25519 signature of data B with the public key B''.

6.2 Smart-contract address parser

Two special words can be used to parse TON smart-contract addresses in human-readable (base64 or base64url) forms:

- `smca>$` (x y z – S), packs a standard TON smart-contract address with workchain x (a signed 32-bit *Integer*) and in-workchain address y (an unsigned 256-bit *Integer*) into a 48-character string S (the human-readable representation of the address) according to flags z. Possible individual flags in z are: +1 for non-bounceable addresses, +2 for testnet-only addresses, and +4 for base64url output instead of base64.

- `$>smca` (S – x y z -1 or 0), unpacks a standard TON smart-contract address from its human-readable string representation S. On success, returns the signed 32-bit workchain x, the unsigned 256-bit in-workchain address y, the flags z (where $+1$ means that the address is non-bounceable, $+2$ that the address is testnet-only), and -1. On failure, pushes 0.

A sample human-readable smart-contract address could be deserialized and displayed as follows:

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

outputs "`-1 538fa7...0f7d 0`", meaning that the specified address is in workchain -1 (the masterchain of the TON Blockchain), and that the 256-bit address inside workchain -1 is 0x538...f7d.

6.3 Dictionary manipulation

Fift has several words for *hashmap* or *(TVM) dictionary* manipulation, corresponding to values of TL-B type `HashmapE` n X as described in [4, 3.3]. These (TVM) dictionaries are not to be confused with the Fift dictionary, which is a completely different thing. A dictionary of TL-B type `HashmapE` n X is essentially a key-value collection with distinct n-bit keys (where $0 \le n \le 1023$) and values of an arbitrary TL-B type X. Dictionaries are represented by trees of cells (the complete layout may be found in [4, 3.3]) and stored as values of type *Cell* or *Slice* in the Fift stack.

- `dictnew` (– s), pushes a *Slice* that represents a new empty dictionary.

- `idict!` (v x s n – s' -1 or s 0), adds a new value v (represented by a *Slice*) with key given by signed big-endian n-bit integer x into dictionary s with n-bit keys, and returns the new dictionary s' and -1 on success. Otherwise the unchanged dictionary s and 0 are returned.

- `idict!+` (v x s n – s' -1 or s 0), adds a new key-value pair (x, v) into dictionary s similarly to `idict!`, but fails if the key already exists by returning the unchanged dictionary s and 0.

- `b>idict!`, `b>idict!+`, variants of `idict!` and `idict!+` accepting the new value v in a *Builder* instead of a *Slice*.

- `udict!`, `udict!+`, `b>udict!`, `b>udict!+`, variants of `idict!`, `idict!+`, `b>idict!`, `b>idict!+`, but with an unsigned n-bit integer x used as a key.

- `idict@` (x s n – v -1 or 0), looks up the key represented by signed big-endian n-bit *Integer* x in the dictionary represented by *Slice* s. If the key is found, returns the corresponding value as a *Slice* v and -1. Otherwise returns 0.

- `udict@` (x s n – v -1 or 0), similar to `idict@`, but with an *un*signed big-endian n-bit *Integer* x used as a key.

- `dictmap` (s n e – s'), applies execution token e (i.e., an anonymous function) to each of the key-value pairs stored in a dictionary s with n-bit keys. The execution token is executed once for each key-value pair, with a *Builder* b and a *Slice* v (containing the value) pushed into the stack before executing e. After the execution e must leave in the stack either a modified *Builder* b' (containing all data from b along with the new value v') and -1, or 0 indicating failure. In the latter case, the corresponding key is omitted from the new dictionary.

- `dictmerge` (s s' n e – s''), combines two dictionaries s and s' with n-bit keys into one dictionary s'' with the same keys. If a key is present in only one of the dictionaries s and s', this key and the corresponding value are copied verbatim to the new dictionary s''. Otherwise the execution token (anonymous function) e is invoked to merge the two values v and v' corresponding to the same key k in s and s', respectively. Before e is invoked, a *Builder* b and two *Slice*s v and v' representing the two values to be merged are pushed. After the execution e leaves either a modified *Builder* b' (containing the original data from b along with the combined value) and -1, or 0 on failure. In the latter case, the corresponding key is omitted from the new dictionary.

Fift also offers some support for prefix dictionaries:

- `pfxdict!` (v k s n – s' -1 or s 0), adds key-value pair (k, v), both represented by *Slice*s, into a prefix dictionary s with keys of length at

most n. On success, returns the modified dictionary s' and -1. On failure, returns the original dictionary s and 0.

- `pfxdict!+` (v k s n – s' -1 or s 0), adds key-value pair (k, v) into prefix dictionary s similarly to `pfxdict!`, but fails if the key already exists.

- `pfxdict@` (k s n – v -1 or 0), looks up key k (represented by a *Slice*) in the prefix dictionary s with the length of keys limited by n bits. On success, returns the value found v and -1. On failure, returns 0.

6.4 Invoking TVM from Fift

TVM can be linked with the Fift interpreter. In this case, several Fift primitives become available that can be used to invoke TVM with arguments provided from Fift. The arguments can be prepared in the Fift stack, which is passed in its entirety to the new instance of TVM. The resulting stack and the exit code are passed back to Fift and can be examined afterwards.

- `runvmcode` ($\ldots s$ – $\ldots x$), invokes a new instance of TVM with the current continuation `cc` initialized from *Slice* s, thus executing code s in TVM. The original Fift stack (without s) is passed in its entirety as the initial stack of TVM. When TVM terminates, its resulting stack is used as the new Fift stack, with the exit code x pushed at its top. If x is non-zero, indicating that TVM has been terminated by an unhandled exception, the next stack entry from the top contains the parameter of this exception, and x is the exception code. All other entries are removed from the stack in this case.

- `runvmdict` ($\ldots s$ – $\ldots x$), invokes a new instance of TVM with the current continuation `cc` initialized from *Slice* s similarly to `runvmcode`, but also initializes the special register `c3` with the same value, and pushes a zero into the initial TVM stack before the TVM execution begins. In a typical application *Slice* s consists of a subroutine selection code that uses the top-of-stack *Integer* to select the subroutine to be executed, thus enabling the definition and execution of several mutually-recursive subroutines (cf. [4, 4.6] and **7.8**). The selector equal to zero corresponds to the `main()` subroutine in a large TVM program.

- `runvm` ($\ldots s\ c - \ldots x\ c'$), invokes a new instance of TVM with both the current continuation `cc` and the special register `c3` initialized from *Slice* s, and pushes a zero into the initial TVM stack similarly to `runvmdict`, but also initializes special register `c4` (the "root of persistent data", cf. [4, 1.4]) with *Cell* c. The final value of `c4` is returned at the top of the final Fift stack as another *Cell* c'. In this way one can emulate the execution of smart contracts that inspect or modify their persistent storage.

- `gasrunvmcode` ($\ldots s\ z - \ldots x\ z'$), a gas-aware version of `runvmcode` that accepts an extra *Integer* argument z (the original gas limit) at the top of the stack, and returns the gas consumed by this TVM run as a new top-of-stack *Integer* value z'.

- `gasrunvmdict` ($\ldots s\ z - \ldots x\ z'$), a gas-aware version of `runvmdict`.

- `gasrunvm` ($\ldots s\ c\ z - \ldots x\ c'\ z'$), a gas-aware version of `runvm`.

For example, one can create an instance of TVM running some simple code as follows:

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

The TVM stack is initialized by three integers 2, 3, and 9 (in this order; 9 is the topmost entry), and then the *Slice* `x{1221}` containing 16 data bits and no references is transformed into a TVM continuation and executed. By consulting Appendix A of [4], we see that `x{12}` is the code of the TVM instruction `XCHG s1, s2`, and that `x{21}` is the code of the TVM instruction `OVER` (not to be confused with the Fift primitive `over`, which incidentally has the same effect on the stack). The result of the above execution is:

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

Here 0 is the exit code (indicating successful TVM termination), and `3 2 9 2` is the final TVM stack state.

If an unhandled exception is generated during the TVM execution, the code of this exception is returned as the exit code:

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

produces

```
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
```

Notice that TVM is executed with internal logging enabled, and its log is displayed in the standard output.

Simple TVM programs may be represented by *Slice* literals with the aid of the `x{...}` construct similarly to the above examples. More sophisticated programs are usually created with the aid of the Fift assembler as explained in the next chapter.

7 Using the Fift assembler

The *Fift assembler* is a short program (currently less than 30KiB) written completely in Fift that transforms human-readable mnemonics of TVM instructions into their binary representation. For instance, one could write `<{ s1 s2 XCHG OVER }>s` instead of `x{1221}` in the example discussed in **6.4**, provided the Fift assembler has been loaded beforehand (usually by the phrase `"Asm.fif" include`).

7.1 Loading the Fift assembler

The Fift assembler is usually located in file `Asm.fif` in the Fift library directory (which usually contains standard Fift library files such as `Fift.fif`). It is typically loaded by putting the phrase `"Asm.fif" include` at the very beginning of a program that needs to use Fift assembler:

- `include` (S –), loads and interprets a Fift source file from the path given by *String* S. If the filename S does not begin with a slash, the Fift include search path, typically taken from the `FIFTPATH` environment variable or the `-I` command-line argument of the Fift interpreter (and equal to `/usr/lib/fift` if both are absent), is used to locate S.

The current implementation of the Fift assembler makes heavy use of custom defining words (cf. **4.8**); its source can be studied as a good example of how defining words might be used to write very compact Fift programs (cf. also the original edition of [1], where a simple 8080 Forth assembler is discussed).

In the future, almost all of the words defined by the Fift assembler will be moved to a separate vocabulary (namespace). Currently they are defined in the global namespace, because Fift does not support namespaces yet.

7.2 Fift assembler basics

The Fift assembler inherits from Fift its postfix operation notation, i.e., the arguments or parameters are written before the corresponding instructions. For instance, the TVM assembler instruction represented as `XCHG s1,s2` in [4] is represented in the Fift assembler as `s1 s2 XCHG`.

Fift assembler code is usually opened by a special opening word, such as `<{`, and terminated by a closing word, such as `}>` or `}>s`. For instance,

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

compiles two TVM instructions `XCHG s1,s2` and `OVER`, and returns the result as a *Slice* (because `}>s` is used). The resulting *Slice* is displayed by `csr.`, yielding

```
x{1221}
```

One can use Appendix A of [4] and verify that `x{12}` is indeed the (codepage zero) code of the TVM instruction `XCHG s1,s2`, and that `x{21}` is the code of the TVM instruction `OVER` (not to be confused with Fift primitive `over`).

In the future, we will assume that the Fift assember is already loaded and omit the phrase `"Asm.fif" include` from our examples.

The Fift assembler uses the Fift stack in a straightforward fashion, using the top several stack entries to hold a *Builder* with the code being assembled, and the arguments to TVM instructions. For example:

- `<{` (– *b*), begins a portion of Fift assembler code by pushing an empty *Builder* into the Fift stack (and potentially switching the namespace to the one containing all Fift assembler-specific words). Approximately equivalent to `<b`.

- `}>` ($b - b'$), terminates a portion of Fift assembler code and returns the assembled portion as a *Builder* (and potentially recovers the original namespace). Approximately equivalent to `nop` in most situations.

- `}>c` ($b - c$), terminates a portion of Fift assembler code and returns the assembled portion as a *Cell* (and potentially recovers the original namespace). Approximately equivalent to `b>`.

- `}>s` ($b - s$), terminates a portion of Fift assembler code similarly to `}>`, but returns the assembled portion as a *Slice*. Equivalent to `}>c <s`.

- `OVER` ($b - b'$), assembles the code of the TVM instruction `OVER` by appending it to the *Builder* at the top of the stack. Approximately equivalent to `x{21} s,`.

- `s1` ($- s$), pushes a special *Slice* used by the Fift assembler to represent the "stack register" `s1` of TVM.

- `s0...s15` ($- s$), words similar to `s1`, but pushing the *Slice* representing other "stack registers" of TVM. Notice that `s16...s255` must be accessed using the word `s()`.

- `s()` ($x - s$), takes an *Integer* argument $0 \le x \le 255$ and returns a special *Slice* used by the Fift assembler to represent "stack register" `s`(x).

- `XCHG` ($b\ s\ s' - b'$), takes two special *Slice*s representing two "stack registers" `s`(i) and `s`(j) from the stack, and appends to *Builder* b the code for the TVM instruction `XCHG s`(i)`,s`(j).

In particular, note that the word `OVER` defined by the Fift assembler has a completely different effect from Fift primitive `over`.

The actual action of `OVER` and other Fift assembler words is somewhat more complicated than that of `x{21} s,`. If the new instruction code does not fit into the *Builder* b (i.e., if b would contain more than 1023 data bits after adding the new instruction code), then this and all subsequent instructions are assembled into a new *Builder* $\tilde{b}$, and the old *Builder* b is augmented by a reference to the *Cell* obtained from $\tilde{b}$ once the generation of $\tilde{b}$ is finished. In this way long stretches of TVM code are automatically split into chains of valid *Cell*s containing at most 1023 bits each. Because TVM interprets a

lonely cell reference at the end of a continuation as an implicit `JMPREF`, this partitioning of TVM code into cells has almost no effect on the execution.

7.3 Pushing integer constants

The TVM instruction `PUSHINT` x, pushing an *Integer* constant x when invoked, can be assembled with the aid of Fift assembler words `INT` or `PUSHINT`:

- `PUSHINT` $(b\ x - b')$, assembles TVM instruction `PUSHINT` x into a *Builder*.
- `INT` $(b\ x - b')$, equivalent to `PUSHINT`.

Notice that the argument to `PUSHINT` is an *Integer* value taken from the Fift stack and is not necessarily a literal. For instance, `<{ 239 17 * INT }>s` is a valid way to assemble a `PUSHINT 4063` instruction, because $239 \cdot 17 = 4063$. Notice that the multiplication is performed by Fift during assemble time, not during the TVM runtime. The latter computation might be performed by means of `<{ 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
```

produces

```
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
```

Notice that the Fift assembler chooses the shortest encoding of the `PUSHINT` x instruction depending on its argument x.

7.4 Immediate arguments

Some TVM instructions (such as `PUSHINT`) accept immediate arguments. These arguments are usually passed to the Fift word assembling the corresponding instruction in the Fift stack. Integer immediate arguments are usually represented by *Integer*s, cells by *Cell*s, continuations by *Builder*s and *Cell*s, and cell slices by *Slice*s. For instance, `17 ADDCONST` assembles TVM instruction `ADDCONST 17`, and `x{ABCD_} PUSHSLICE` assembles `PUSHSLICE xABCD_`:

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

produces

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

On some occasions, the Fift assembler pretends to be able to accept immediate arguments that are out of range for the corresponding TVM instruction. For instance, `ADDCONST` x is defined only for $-128 \le x < 128$, but the Fift assembler accepts `239 ADDCONST`:

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

produces

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

We can see that "`ADDCONST 239`" has been tacitly replaced by `PUSHINT 239` and `ADD`. This feature is convenient when the immediate argument to `ADDCONST` is itself a result of a Fift computation, and it is difficult to estimate whether it will always fit into the required range.

In some cases, there are several versions of the same TVM instructions, one accepting an immediate argument and another without any arguments.

For instance, there are both `LSHIFT` n and `LSHIFT` instructions. In the Fift assembler, such variants are assigned distinct mnemonics. In particular, `LSHIFT` n is represented by n `LSHIFT#`, and `LSHIFT` is represented by itself.

7.5 Immediate continuations

When an immediate argument is a continuation, it is convenient to create the corresponding *Builder* in the Fift stack by means of a nested `<{ ... }>` construct. For instance, TVM assembler instructions

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

can be assembled and executed by

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

producing

```
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
```

More convenient ways to use literal continuations created by means of the Fift assembler exist. For instance, the above example can be also assembled by

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

or even

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

both producing "`x{710192A70AE4} ok`".

Incidentally, a better way of implementing the above loop is by means of `REPEATEND`:

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

or

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

both produce "`x{7101E7A70A}`" and output "`10000000`" after seven iterations of the loop.

Notice that several TVM instructions that store a continuation in a separate cell reference (such as `JMPREF`) accept their argument in a *Cell*, not in a *Builder*. In such situations, the `<{ ... }>c` construct can be used to produce this immediate argument.

7.6 Control flow: loops and conditionals

Almost all TVM control flow instructions—such as `IF`, `IFNOT`, `IFRET`, `IFNOTRET`, `IFELSE`, `WHILE`, `WHILEEND`, `REPEAT`, `REPEATEND`, `UNTIL`, and `UNTILEND`—can be assembled similarly to `REPEAT` and `REPEATEND` in the examples of **7.5** when applied to literal continuations. For instance, TVM assembler code

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

which computes $3n + 1$ or $n/2$ depending on whether its argument n is odd or even, can be assembled and applied to $n = 7$ by

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

producing

```
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
```

Of course, a more compact and efficient way to implement this conditional expression would be

```
<{ DUP 1 INT AND
```

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

or

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

both producing the same code "`x{2071B093A703A4DCAB00}`".

Fift assembler words that can be used to produce such "high-level" conditionals and loops include `IF:<{`, `IFNOT:<{`, `IFJMP:<{`, `}>ELSE<{`, `}>ELSE:`, `}>IF`, `REPEAT:<{`, `UNTIL:<{`, `WHILE:<{`, `}>DO<{`, `}>DO:`, `AGAIN:<{`, `}>AGAIN`, `}>REPEAT`, and `}>UNTIL`. Their complete list can be found in the source file `Asm.fif`. For instance, an UNTIL loop can be created by `UNTIL:<{ ... }>` or `<{ ... }>UNTIL`, and a WHILE loop by `WHILE:<{ ... }>DO<{ ... }>`.

If we choose to keep a conditional branch in a separate cell, we can use the `<{ ... }>c` construct along with instructions such as `IFJMPREF`:

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

has the same effect as the code from the previous example when executed, but it is contained in two separate cells:

```
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 Macro definitions

Because TVM instructions are implemented in the Fift assembler using Fift words that have a predictable effect on the Fift stack, the Fift assembler is automatically a macro assembler, supporting macro definitions. For instance, suppose that we wish to define a macro definition `RANGE` x y, which checks whether the TVM top-of-stack value is between integer literals x and y (inclusive). This macro definition can be implemented as follows:

```
{ 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 .
```

which produces

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

Notice that `GEQINT` and `LEQINT` are themselves macro definitions defined in `Asm.fif`, because they do not correspond directly to TVM instructions. For instance, x `GEQINT` corresponds to the TVM instruction `GTINT` $x-1$.

Incidentally, the above code can be shortened by two bytes by replacing `IFNOT: DROP ZERO` with `AND`.

7.8 Larger programs and subroutines

Larger TVM programs, such as TON Blockchain smart contracts, typically consist of several mutually recursive subroutines, with one or several of them selected as top-level subroutines (called `main()` or `recv_internal()` for

smart contracts). The execution starts from one of the top-level subroutines, which is free to call any of the other defined subroutines, which in turn can call whatever other subroutines they need.

Such TVM programs are implemented by means of a selector function, which accepts an extra integer argument in the TVM stack; this integer selects the actual subroutine to be invoked (cf. [4, 4.6]). Before execution, the code of this selector function is loaded both into special register `c3` and into the current continuation `cc`. The selector of the main function (usually zero) is pushed into the initial stack, and the TVM execution is started. Afterwards a subroutine can be invoked by means of a suitable TVM instruction, such as `CALLDICT` n, where n is the (integer) selector of the subroutine to be called.

The Fift assembler offers several words facilitating the implementation of such large TVM programs. In particular, subroutines can be defined separately and assigned symbolic names (instead of numeric selectors), which can be used to call them afterwards. The Fift assembler automatically creates a selector function from these separate subroutines and returns it as the top-level assembly result.

Here is a simple example of such a program consisting of several subroutines. This program computes the complex number $(5 + i)^4 \cdot (239 - i)$:

```
"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
```

This program produces:

```
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
```

Some observations and comments based on the previous example follow:

- A TVM program is opened by `PROGRAM{` and closed by either `}END>c` (which returns the assembled program as a *Cell*) or `}END>s` (which returns a *Slice*).
- A new subroutine is declared by means of the phrase `NEWPROC` ⟨*name*⟩. This declaration assigns the next positive integer as a selector for the newly-declared subroutine, and stores this integer into the constant ⟨*name*⟩. For instance, the above declarations define `add`, `sub`, and `mul` as integer constants equal to 1, 2, and 3, respectively.
- Some subroutines are predeclared and do not need to be declared again by `NEWPROC`. For instance, `main` is a subroutine identifier bound to the integer constant (selector) 0.
- Other predefined subroutine selectors such as `recv_internal` (equal to 0) or `recv_external` (equal to −1), useful for implementing TON Blockchain smart contracts (cf. [5, 4.4]), can be declared by means of `constant` (e.g., `-1 constant recv_external`).
- A subroutine can be defined either with the aid of the word `PROC`, which accepts the integer selector of the subroutine and the *Slice* containing the code for this subroutine, or with the aid of the construct ⟨*selector*⟩ `PROC:<{ ... }>`, convenient for defining larger subroutines.

- `CALLDICT` and `JMPDICT` instructions may be assembled with the aid of the words `CALL` and `JMP`, which accept the integer selector of the subroutine to be called as an immediate argument passed in the Fift stack.
- The current implementation of the Fift assembler collects all subroutines into a dictionary with 14-bit signed integer keys. Therefore, all subroutine selectors must be in the range $-2^{13} \dots 2^{13} - 1$.
- If a subroutine with an unknown selector is called during runtime, an exception with code 11 is thrown by the code automatically inserted by the Fift assembler. This code also automatically selects codepage zero for instruction encoding by means of a `SETCP0` instruction.
- The Fift assembler checks that all subroutines declared by `NEWPROC` are actually defined by `PROC` or `PROC:<{` before the end of the program. It also checks that a subroutine is not redefined.

One should bear in mind that very simple programs (including the simplest smart contracts) may be made more compact by eliminating this general subroutine selection machinery in favor of custom subroutine selection code and removing unused subroutines. For instance, the above example can be transformed into

```
<{ 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
```

which produces

```
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 List of Fift words

This Appendix provides an alphabetic list of almost all Fift words—including primitives and definitions from the standard library `Fift.fif`, but excluding Fift assembler words defined in `Asm.fif` (because the Fift assembler is simply an application from the perspective of Fift). Some experimental words have been omitted from this list. Other words may have been added to or removed from Fift after this text was written. The list of all words available in your Fift interpreter may be inspected by executing `words`.

Each word is described by its name, followed by its *stack notation* in parentheses, indicating several values near the top of the Fift stack before and after the execution of the word; all deeper stack entries are usually assumed to be left intact. After that, a text description of the word's effect is provided. If the word has been discussed in a previous section of this document, a reference to this section is included.

Active words and active prefix words that parse a portion of the input stream immediately after their occurrence are listed here in a modified way. Firstly, these words are listed alongside the portion of the input that they parse; the segment of each entry that is actually a Fift word is underlined for emphasis. Secondly, their stack effect is usually described from the user's perspective, and reflects the actions performed during the execution phase of the encompassing blocks and word definitions.

For example, the active prefix word `B{`, used for defining *Bytes* literals (cf. **5.6**), is listed as `B{`⟨*hex-digits*⟩`}`, and its stack effect is shown as (– B) instead of (– B 1 e), even though the real effect of the execution of the active word `B{` during the compilation phase of an encompassing block or word definition is the latter one (cf. **4.2**).

- `!` (x p –), stores new value x into *Box* p, cf. **2.14**.
- `"`⟨*string*⟩`"` (– S), pushes a *String* literal into the stack, cf. **2.9** and **2.10**.
- `#` (x S – x' S'), performs one step of the conversion of *Integer* x into its decimal representation by appending to *String* S one decimal digit representing x mod 10. The quotient $x' := \lfloor x/10 \rfloor$ is returned as well.
- `#>` (S – S'), finishes the conversion of an *Integer* into its human-readable representation (decimal or otherwise) started with `<#` by reversing *String* S. Equivalent to `$reverse`.

- `#s` (x S – x' S'), performs `#` one or more times until the quotient x' becomes non-positive. Equivalent to `{ # over 0<= } until`.

- `$#` (– x), pushes the total number of command-line arguments passed to the Fift program, cf. **2.18**. Defined only when the Fift interpreter is invoked in script mode (with the `-s` command line argument).

- `$(`⟨*string*⟩`)` (– ...), looks up the word `$`⟨*string*⟩ during execution time and executes its current definition. Typically used to access the current values of command-line arguments, e.g., `$(2)` is essentially equivalent to `@' $2`.

- `$()` (x – S), pushes the x-th command-line argument similarly to `$`n, but with *Integer* $x \geq 0$ taken from the stack, cf. **2.18**. Defined only when the Fift interpreter is invoked in script mode (with the `-s` command line argument).

- `$+` (S S' – $S.S'$), concatenates two strings, cf. **2.10**.

- `$,` (b S – b'), appends *String* S to *Builder* b, cf. **5.2**. The string is interpreted as a binary string of length $8n$, where n is the number of bytes in the UTF-8 representation of S.

- `$`n (– S), pushes the n-th command-line argument as a *String* S, cf. **2.18**. For instance, `$0` pushes the name of the script being executed, `$1` the first command line argument, and so on. Defined only when the Fift interpreter is invoked in script mode (with the `-s` command line argument).

- `$=` (S S' – ?), returns -1 if strings S and S' are equal, 0 otherwise, cf. **2.13**. Equivalent to `$cmp 0=`.

- `$>s` (S – s), transforms the *String* S into a *Slice*, cf. **5.3**. Equivalent to `<b swap $, b> <s`.

- `$>smca` (S – x y z -1 or 0), unpacks a standard TON smart-contract address from its human-readable string representation S, cf. **6.2**. On success, returns the signed 32-bit workchain x, the unsigned 256-bit in-workchain address y, the flags z (where $+1$ means that the address is non-bounceable, $+2$ that the address is testnet-only), and -1. On failure, pushes 0.

- `$@` ($s\ x$ – S), fetches the first x bytes (i.e., $8x$ bits) from *Slice* s, and returns them as a UTF-8 *String* S, cf. **5.3**. If there are not enough data bits in s, throws an exception.
- `$@+` ($s\ x$ – $S\ s'$), similar to `$@`, but returns the remainder of *Slice* s as well, cf. **5.3**.
- `$@?` ($s\ x$ – S -1 or 0), similar to `$@`, but uses a flag to indicate failure instead of throwing an exception, cf. **5.3**.
- `$@?+` ($s\ x$ – $S\ s'$ -1 or s 0), similar to `$@+`, but uses a flag to indicate failure instead of throwing an exception, cf. **5.3**.
- `$cmp` ($S\ S'$ – x), returns 0 if strings S and S' are equal, -1 if S is lexicographically less than S', and 1 if S is lexicographically greater than S', cf. **2.13**.
- `$len` (S – x), computes the byte length (not the UTF-8 character length!) of a string, cf. **2.10**.
- `$reverse` (S – S'), reverses the order of UTF-8 characters in *String* S. If S is not a valid UTF-8 string, the return value is undefined and may be also invalid.
- `%1<<` ($x\ y$ – z), computes $z := x \bmod 2^y = x\&(2^y - 1)$ for two *Integer*s x and $0 \le y \le 256$.
- `'` ⟨*word-name*⟩ (– e), returns the execution token equal to the current (compile-time) definition of ⟨*word-name*⟩, cf. **3.1**. If the specified word is not found, throws an exception.
- `'nop` (– e), pushes the default definition of `nop`—an execution token that does nothing when executed, cf. **4.6**.
- `(')` ⟨*word-name*⟩ (– e), similar to `'`, but returns the definition of the specified word at execution time, performing a dictionary lookup each time it is invoked, cf. **4.6**. May be used to recover the current values of constants inside word definitions and other blocks by using the phrase `(')` ⟨*word-name*⟩ `execute`.
- `(-trailing)` ($S\ x$ – S'), removes from *String* S all trailing characters with UTF-8 codepoint x.

- `(.)` (x – S), returns the *String* with the decimal representation of *Integer* x. Equivalent to `dup abs <# #s rot sign #> nip`.

- `(atom)` (S x – a –1 or 0), returns the only *Atom* a with the name given by *String* S, cf. **2.17**. If there is no such *Atom* yet, either creates it (if *Integer* x is non-zero) or returns a single zero to indicate failure (if x is zero).

- `(b.)` (x – S), returns the *String* with the binary representation of *Integer* x.

- `(compile)` (l x_1 ... x_n n e – l'), extends *WordList* l so that it would push $0 \leq n \leq 255$ values x_1, ..., x_n into the stack and execute the execution token e when invoked, where $0 \leq n \leq 255$ is an *Integer*, cf. **4.7**. If e is equal to the special value `'nop`, the last step is omitted.

- `(create)` (e S x –), creates a new word with the name equal to *String* S and definition equal to *WordDef* e, using flags passed in *Integer* $0 \leq x \leq 3$, cf. **4.5**. If bit +1 is set in x, creates an active word; if bit +2 is set in x, creates a prefix word.

- `(def?)` (S – ?), checks whether the word S is defined.

- `(dump)` (x – S), returns a *String* with a dump of the topmost stack value x, in the same format as employed by `.dump`.

- `(execute)` (x_1 ... x_n n e – ...), executes execution token e, but first checks that there are at least $0 \leq n \leq 255$ values in the stack apart from n and e themselves. It is a counterpart of `(compile)` that may be used to immediately "execute" (perform the intended runtime action of) an active word after its immediate execution, as explained in **4.2**.

- `(forget)` (S –), forgets the word with the name specified in *String* S, cf. **4.5**. If the word is not found, throws an exception.

- `(number)` (S – 0 or x 1 or x y 2), attempts to parse the *String* S as an integer or fractional literal, cf. **2.10** and **2.8**. On failure, returns a single 0. On success, returns x 1 if S is a valid integer literal with value x, or x y 2 if S is a valid fractional literal with value x/y.

- `(x.)` (x – S), returns the *String* with the hexadecimal representation of *Integer* x.
- `({)` (– l), pushes an empty *WordList* into the stack, cf. **4.7**
- `(})` (l – e), transforms a *WordList* into an execution token (*WordDef*), making all further modifications impossible, cf. **4.7**.
- `*` (x y – xy), computes the product xy of two *Integer*s x and y, cf. **2.4**.
- `*/` (x y z – $\lfloor xy/z \rfloor$), "multiply-then-divide": multiplies two integers x and y producing a 513-bit intermediate result, then divides the product by z, cf. **2.4**.
- `*/c` (x y z – $\lceil xy/z \rceil$), "multiply-then-divide" with ceiling rounding: multiplies two integers x and y producing a 513-bit intermediate result, then divides the product by z, cf. **2.4**.
- `*/cmod` (x y z – q r), similar to `*/c`, but computes both the quotient $q := \lceil xy/z \rceil$ and the remainder $r := xy - qz$, cf. **2.4**.
- `*/mod` (x y z – q r), similar to `*/`, but computes both the quotient $q := \lfloor xy/z \rfloor$ and the remainder $r := xy - qz$, cf. **2.4**.
- `*/r` (x y z – $q := \lfloor xy/z + 1/2 \rfloor$), "multiply-then-divide" with nearest-integer rounding: multiplies two integers x and y with 513-bit intermediate result, then divides the product by z, cf. **2.4**.
- `*/rmod` (x y z – q r), similar to `*/r`, but computes both the quotient $q := \lfloor xy/z + 1/2 \rfloor$ and the remainder $r := xy - qz$, cf. **2.4**.
- `*>>` (x y z – q), similar to `*/`, but with division replaced with a right shift, cf. **2.4**. Computes $q := \lfloor xy/2^z \rfloor$ for $0 \le z \le 256$. Equivalent to `1<< */`.
- `*>>c` (x y z – q), similar to `*/c`, but with division replaced with a right shift, cf. **2.4**. Computes $q := \lceil xy/2^z \rceil$ for $0 \le z \le 256$. Equivalent to `1<< */c`.
- `*>>r` (x y z – q), similar to `*/r`, but with division replaced with a right shift, cf. **2.4**. Computes $q := \lfloor xy/2^z + 1/2 \rfloor$ for $0 \le z \le 256$. Equivalent to `1<< */r`.

- `*mod` ($x\ y\ z$ – r), similar to `*/mod`, but computes only the remainder $r := xy - qz$, where $q := \lfloor xy/z \rfloor$. Equivalent to `*/mod nip`.
- `+` ($x\ y$ – $x+y$), computes the sum $x+y$ of two *Integer*s x and y, cf. **2.4**.
- `+!` ($x\ p$ –), increases the integer value stored in *Box* p by *Integer* x, cf. **2.14**. Equivalent to `tuck @ + swap !`.
- `+"`⟨*string*⟩`"` (S – S'), concatenates *String* S with a string literal, cf. **2.10**. Equivalent to `"`⟨*string*⟩`" $+`.
- `,` ($t\ x$ – t'), appends x to the end of *Tuple* t, and returns the resulting *Tuple* t', cf. **2.15**.
- `-` ($x\ y$ – $x-y$), computes the difference $x-y$ of two *Integer*s x and y, cf. **2.4**.
- `-!` ($x\ p$ –), decreases the integer value stored in *Box* p by *Integer* x. Equivalent to `swap negate swap +!`.
- `-1` (– -1), pushes *Integer* -1.
- `-1<<` (x – -2^x), computes -2^x for $0 \le x \le 256$. Approximately equivalent to `1<< negate` or `-1 swap <<`, but works for $x = 256$ as well.
- `-roll` ($x_n \ldots x_0\ n$ – $x_0\ x_n \ldots x_1$), rotates the top n stack entries in the opposite direction, where $n \ge 0$ is also passed in the stack, cf. **2.5**. In particular, `1 -roll` is equivalent to `swap`, and `2 -roll` to `-rot`.
- `-rot` ($x\ y\ z$ – $z\ x\ y$), rotates the three topmost stack entries in the opposite direction, cf. **2.5**. Equivalent to `rot rot`.
- `-trailing` (S – S'), removes from *String* S all trailing spaces. Equivalent to `bl (-trailing)`.
- `-trailing0` (S – S'), removes from *String* S all trailing '`0`' characters. Equivalent to `char 0 (-trailing)`.
- `.` (x –), prints the decimal representation of *Integer* x, followed by a single space, cf. **2.4**. Equivalent to `._ space`.

- ."⟨*string*⟩" (–), prints a constant string into the standard output, cf. **2.10**.
- ._ (x –), prints the decimal representation of *Integer* x without any spaces. Equivalent to (.) type.
- .dump (x –), dumps the topmost stack entry in the same way as .s dumps all stack elements, cf. **2.15**. Equivalent to (dump) type space.
- .l (l –), prints a Lisp-style list l, cf. **2.16**.
- .s (–), dumps all stack entries starting from the deepest, leaving them intact, cf. **2.5**. Human-readable representations of stack entries are output separated by spaces, followed by an end-of-line character.
- .sl (–), dumps all stack entries leaving them intact similarly to .s, but showing each entry as a List-style list l as .l does.
- .tc (–), outputs the total number of allocated cells into the standard error stream.
- / (x y – $q := \lfloor x/y \rfloor$), computes the floor-rounded quotient $\lfloor x/y \rfloor$ of two *Integer*s, cf. **2.4**.
- /* ⟨*multiline-comment*⟩ */ (–), skips a multi-line comment delimited by word "*/" (followed by a blank or an end-of-line character), cf. **2.2**.
- // ⟨*comment-to-eol*⟩ (–), skips a single-line comment until the end of the current line, cf. **2.2**.
- /c (x y – $q := \lceil x/y \rceil$), computes the ceiling-rounded quotient $\lceil x/y \rceil$ of two *Integer*s, cf. **2.4**.
- /cmod (x y – q r), computes both the ceiling-rounded quotient $q := \lceil x/y \rceil$ and the remainder $r := x - qy$, cf. **2.4**.
- /mod (x y – q r), computes both the floor-rounded quotient $q := \lfloor x/y \rfloor$ and the remainder $r := x - qy$, cf. **2.4**.
- /r (x y – q), computes the nearest-integer-rounded quotient $\lfloor x/y+1/2 \rfloor$ of two *Integer*s, cf. **2.4**.

- `/rmod` ($x\ y$ – $q\ r$), computes both the nearest-integer-rounded quotient $q := \lfloor x/y + 1/2 \rfloor$ and the remainder $r := x - qy$, cf. **2.4**.
- `0` (– 0), pushes *Integer* 0.
- `0!` (p –), stores *Integer* 0 into *Box* p, cf. **2.14**. Equivalent to `0 swap !`.
- `0<` (x – ?), checks whether $x < 0$ (i.e., pushes -1 if x is negative, 0 otherwise), cf. **2.12**. Equivalent to `0 <`.
- `0<=` (x – ?), checks whether $x \leq 0$ (i.e., pushes -1 if x is non-positive, 0 otherwise), cf. **2.12**. Equivalent to `0 <=`.
- `0<>` (x – ?), checks whether $x \neq 0$ (i.e., pushes -1 if x is non-zero, 0 otherwise), cf. **2.12**. Equivalent to `0 <>`.
- `0=` (x – ?), checks whether $x = 0$ (i.e., pushes -1 if x is zero, 0 otherwise), cf. **2.12**. Equivalent to `0 =`.
- `0>` (x – ?), checks whether $x > 0$ (i.e., pushes -1 if x is positive, 0 otherwise), cf. **2.12**. Equivalent to `0 >`.
- `0>=` (x – ?), checks whether $x \geq 0$ (i.e., pushes -1 if x is non-negative, 0 otherwise), cf. **2.12**. Equivalent to `0 >=`.
- `1` (– 1), pushes *Integer* 1.
- `1+` (x – $x + 1$), computes $x + 1$. Equivalent to `1 +`.
- `1+!` (p –), increases the integer value stored in *Box* p by one, cf. **2.14**. Equivalent to `1 swap +!`.
- `1-` (x – $x - 1$), computes $x - 1$. Equivalent to `1 -`.
- `1-!` (p –), decreases the integer value stored in *Box* p by one. Equivalent to `-1 swap +!`.
- `1<<` (x – 2^x), computes 2^x for $0 \leq x \leq 255$. Equivalent to `1 swap <<`.
- `1<<1-` (x – $2^x - 1$), computes $2^x - 1$ for $0 \leq x \leq 256$. Almost equivalent to `1<< 1-`, but works for $x = 256$.

- `2` (– 2), pushes *Integer* 2.
- `2*` (x – $2x$), computes $2x$. Equivalent to `2 *`.
- `2+` (x – $x + 2$), computes $x + 2$. Equivalent to `2 +`.
- `2-` (x – $x - 2$), computes $x - 2$. Equivalent to `2 -`.
- `2/` (x – $\lfloor x/2 \rfloor$), computes $\lfloor x/2 \rfloor$. Equivalent to `2 /` or to `1 >>`.
- `2=:` ⟨*word-name*⟩ (x y –), an active variant of `2constant`: defines a new ordinary word ⟨*word-name*⟩ that would push the given values x and y when invoked, cf. **2.7**.
- `2constant` (x y –), scans a blank-delimited word name S from the remainder of the input, and defines a new ordinary word S as a double constant, which will push the given values x and y (of arbitrary types) when invoked, cf. **4.5**.
- `2drop` (x y –), removes the two topmost stack entries, cf. **2.5**. Equivalent to `drop drop`.
- `2dup` (x y – x y x y), duplicates the topmost pair of stack entries, cf. **2.5**. Equivalent to `over over`.
- `2over` (x y z w – x y z w x y), duplicates the second topmost pair of stack entries.
- `2swap` (a b c d – c d a b), interchanges the two topmost pairs of stack entries, cf. **2.5**.
- `:` ⟨*word-name*⟩ (e –), defines a new ordinary word ⟨*word-name*⟩ in the dictionary using *WordDef* e as its definition, cf. **4.5**. If the specified word is already present in the dictionary, it is tacitly redefined.
- `::` ⟨*word-name*⟩ (e –), defines a new active word ⟨*word-name*⟩ in the dictionary using *WordDef* e as its definition, cf. **4.5**. If the specified word is already present in the dictionary, it is tacitly redefined.
- `::_` ⟨*word-name*⟩ (e –), defines a new active prefix word ⟨*word-name*⟩ in the dictionary using *WordDef* e as its definition, cf. **4.5**. If the specified word is already present in the dictionary, it is tacitly redefined.

- `:_` ⟨*word-name*⟩ (e –), defines a new ordinary prefix word ⟨*word-name*⟩ in the dictionary using *WordDef* e as its definition, cf. **4.5**. If the specified word is already present in the dictionary, it is tacitly redefined.

- `<` (x y – ?), checks whether $x < y$ (i.e., pushes -1 if *Integer* x is less than *Integer* y, 0 otherwise), cf. **2.12**.

- `<#` (– S), pushes an empty *String*. Typically used for starting the conversion of an *Integer* into its human-readable representation, decimal or in another base. Equivalent to `""`.

- `<<` (x y – $x \cdot 2^y$), computes an arithmetic left shift of binary number x by $y \geq 0$ positions, yielding $x \cdot 2^y$, cf. **2.4**.

- `<</` (x y z – q), computes $q := \lfloor 2^z x/y \rfloor$ for $0 \leq z \leq 256$ producing a 513-bit intermediate result, similarly to `*/`, cf. **2.4**. Equivalent to `1<< swap */`.

- `<</c` (x y z – q), computes $q := \lceil 2^z x/y \rceil$ for $0 \leq z \leq 256$ producing a 513-bit intermediate result, similarly to `*/c`, cf. **2.4**. Equivalent to `1<< swap */c`.

- `<</r` (x y z – q), computes $q := \lfloor 2^z x/y + 1/2 \rfloor$ for $0 \leq z \leq 256$ producing a 513-bit intermediate result, similarly to `*/r`, cf. **2.4**. Equivalent to `1<< swap */r`.

- `<=` (x y – ?), checks whether $x \leq y$ (i.e., pushes -1 if *Integer* x is less than or equal to *Integer* y, 0 otherwise), cf. **2.12**.

- `<>` (x y – ?), checks whether $x \neq y$ (i.e., pushes -1 if *Integer*s x and y are not equal, 0 otherwise), cf. **2.12**.

- `<b` (– b), creates a new empty *Builder*, cf. **5.2**.

- `<s` (c – s), transforms a *Cell* c into a *Slice* s containing the same data, cf. **5.3**. It usually marks the start of the deserialization of a cell.

- `=` (x y – ?), checks whether $x = y$ (i.e., pushes -1 if *Integer*s x and y are equal, 0 otherwise), cf. **2.12**.

- `=:` ⟨*word-name*⟩ (x –), an active variant of `constant`: defines a new ordinary word ⟨*word-name*⟩ that would push the given value x when invoked, cf. **2.7**.

- `>` (x y – ?), checks whether $x > y$ (i.e., pushes -1 if *Integer* x is greater than *Integer* y, 0 otherwise), cf. **2.12**.

- `>=` (x y – ?), checks whether $x \geq y$ (i.e., pushes -1 if *Integer* x is greater than or equal to *Integer* y, 0 otherwise), cf. **2.12**.

- `>>` (x y – $q := \lfloor x \cdot 2^{-y} \rfloor$), computes an arithmetic right shift of binary number x by $0 \leq y \leq 256$ positions, cf. **2.4**. Equivalent to `1<< /`.

- `>>c` (x y – $q := \lceil x \cdot 2^{-y} \rceil$), computes the ceiling-rounded quotient q of x by 2^y for $0 \leq y \leq 256$, cf. **2.4**. Equivalent to `1<< /c`.

- `>>r` (x y – $q := \lfloor x \cdot 2^{-y} + 1/2 \rfloor$), computes the nearest-integer-rounded quotient q of x by 2^y for $0 \leq y \leq 256$, cf. **2.4**. Equivalent to `1<< /r`.

- `?dup` (x – x x or 0), duplicates an *Integer* x, but only if it is non-zero, cf. **2.5**. Otherwise leaves it intact.

- `@` (p – x), fetches the value currently stored in *Box* p, cf. **2.14**.

- `@'` ⟨*word-name*⟩ (– e), recovers the definition of the specified word at execution time, performing a dictionary lookup each time it is invoked, and then executes this definition, cf. **2.7** and **4.6**. May be used to recover current values of constants inside word definitions and other blocks by using the phrase `@'` ⟨*word-name*⟩, equivalent to `(')` ⟨*word-name*⟩ `execute`.

- `B+` (B' B'' – B), concatenates two *Bytes* values, cf. **5.6**.

- `B,` (b B – b'), appends *Bytes* B to *Builder* b, cf. **5.2**. If there is no room in b for B, throws an exception.

- `B=` (B B' – ?), checks whether two *Bytes* sequences are equal, and returns -1 or 0 depending on the comparison outcome, cf. **5.6**.

- `B>Li@` (B x – y), deserializes the first $x/8$ bytes of a `Bytes` value B as a signed little-endian x-bit *Integer* y, cf. **5.6**.

- `B>Li@+` ($B\ x$ – $B'\ y$), deserializes the first $x/8$ bytes of B as a signed little-endian x-bit *Integer* y similarly to `B>Li@`, but also returns the remaining bytes of B, cf. **5.6**.
- `B>Lu@` ($B\ x$ – y), deserializes the first $x/8$ bytes of a `Bytes` value B as an unsigned little-endian x-bit *Integer* y, cf. **5.6**.
- `B>Lu@+` ($B\ x$ – $B'\ y$), deserializes the first $x/8$ bytes of B as an unsigned little-endian x-bit *Integer* y similarly to `B>Lu@`, but also returns the remaining bytes of B, cf. **5.6**.
- `B>boc` (B – c), deserializes a "standard" bag of cells (i.e., a bag of cells with exactly one root cell) represented by *Bytes* B, and returns the root *Cell* c, cf. **5.5**.
- `B>file` ($B\ S$ –), creates a new (binary) file with the name specified in *String* S and writes data from *Bytes* B into the new file, cf. **5.6**. If the specified file already exists, it is overwritten.
- `B>i@` ($B\ x$ – y), deserializes the first $x/8$ bytes of a `Bytes` value B as a signed big-endian x-bit *Integer* y, cf. **5.6**.
- `B>i@+` ($B\ x$ – $B'\ y$), deserializes the first $x/8$ bytes of B as a signed big-endian x-bit *Integer* y similarly to `B>i@`, but also returns the remaining bytes of B, cf. **5.6**.
- `B>u@` ($B\ x$ – y), deserializes the first $x/8$ bytes of a `Bytes` value B as an unsigned big-endian x-bit *Integer* y, cf. **5.6**.
- `B>u@+` ($B\ x$ – $B'\ y$), deserializes the first $x/8$ bytes of B as an unsigned big-endian x-bit *Integer* y similarly to `B>u@`, but also returns the remaining bytes of B, cf. **5.6**.
- `B@` ($s\ x$ – B), fetches the first x bytes (i.e., $8x$ bits) from *Slice* s, and returns them as a *Bytes* value B, cf. **5.3**. If there are not enough data bits in s, throws an exception.
- `B@+` ($s\ x$ – $B\ s'$), similar to `B@`, but returns the remainder of *Slice* s as well, cf. **5.3**.
- `B@?` ($s\ x$ – B -1 or 0), similar to `B@`, but uses a flag to indicate failure instead of throwing an exception, cf. **5.3**.

- `B@?+` ($s\ x$ – $B\ s'\ -1$ or $s\ 0$), similar to `B@+`, but uses a flag to indicate failure instead of throwing an exception, cf. **5.3**.
- `Bcmp` ($B\ B'$ – x), lexicographically compares two *Bytes* sequences, and returns -1, 0, or 1, depending on the comparison result, cf. **5.6**.
- `Bhash` (B – B'), computes the SHA256 hash of a *Bytes* value, cf. **5.6**. The hash is returned as a 32-byte *Bytes* value.
- `Blen` (B – x), returns the length of a *Bytes* value B in bytes, cf. **5.6**.
- `Bx.` (B –), prints the hexadecimal representation of a *Bytes* value, cf. **5.6**. Each byte is represented by exactly two uppercase hexadecimal digits.
- `B{`⟨*hex-digits*⟩`}` (– B), pushes a *Bytes* literal containing data represented by an even number of hexadecimal digits, cf. **5.6**.
- `B|` ($B\ x$ – $B'\ B''$), cuts the first x bytes from a *Bytes* value B, and returns both the first x bytes (B') and the remainder (B'') as new *Bytes* values, cf. **5.6**.
- `Li>B` ($x\ y$ – B), stores a signed little-endian y-bit *Integer* x into a *Bytes* value B consisting of exactly $y/8$ bytes. Integer y must be a multiple of eight in the range $0 \ldots 256$, cf. **5.6**.
- `Lu>B` ($x\ y$ – B), stores an unsigned little-endian y-bit *Integer* x into a *Bytes* value B consisting of exactly $y/8$ bytes. Integer y must be a multiple of eight in the range $0 \ldots 256$, cf. **5.6**.
- `[` (–), opens an internal interpreter session even if `state` is greater than zero, i.e., all subsequent words are executed immediately instead of being compiled.
- `[]` ($t\ i$ – x), returns the $(i+1)$-st component t_{i+1} of *Tuple* t, where $0 \le i < |t|$, cf. **2.15**.
- `[compile]` ⟨*word-name*⟩ (–), compiles ⟨*word-name*⟩ as if it were an ordinary word, even if it is active, cf. **4.6**. Essentially equivalent to `'` ⟨*word-name*⟩ `execute`.

- `]` ($x_1 \ldots x_n\ n$ –), closes an internal interpreter session opened by `[` and invokes `(compile)` or `(execute)` afterwards depending on whether `state` is greater than zero. For instance, `{ [ 2 3 + 1 ] * }` is equivalent to `{ 5 * }`.

- `‘`⟨*word*⟩ (– a), introduces an *Atom* literal, equal to the only *Atom* with the name equal to ⟨*word*⟩, cf. **2.17**. Equivalent to `"`⟨*word*⟩`" atom`.

- `abort` (S –), throws an exception with an error message taken from *String* S, cf. **3.6**.

- `abort"`⟨*message*⟩`"` (x –), throws an exception with the error message ⟨*message*⟩ if the *Integer* x is non-zero, cf. **3.6**.

- `abs` (x – $|x|$), computes the absolute value $|x| = \max(x, -x)$ of *Integer* x. Equivalent to `dup negate max`.

- `allot` (n – t), creates a new array, i.e., a *Tuple* that consists of n new empty *Box*es, cf. **2.15**. Equivalent to `| { hole , } rot times`.

- `and` ($x\ y$ – $x\&y$), computes the bitwise AND of two *Integer*s, cf. **2.4**.

- `anon` (– a), creates a new unique anonymous *Atom*, cf. **2.17**.

- `atom` (S – a), returns the only *Atom* a with the name S, creating such an atom if necessary, cf. **2.17**. Equivalent to `true (atom) drop`.

- `atom?` (u – ?), checks whether u is an *Atom*, cf. **2.17**.

- `b+` ($b\ b'$ – b''), concatenates two *Builders* b and b', cf. **5.2**.

- `b.` (x –), prints the binary representation of an *Integer* x, followed by a single space. Equivalent to `b._ space`.

- `b._` (x –), prints the binary representation of an *Integer* x without any spaces. Equivalent to `(b.) type`.

- `b>` (b – c), transforms a *Builder* b into a new *Cell* c containing the same data as b, cf. **5.2**.

- `b>idict!` ($v\ x\ s\ n$ – s' -1 or s 0), adds a new value v (represented by a *Builder*) with key given by signed big-endian n-bit integer x into dictionary s with n-bit keys, and returns the new dictionary s' and -1 on success, cf. **6.3**. Otherwise the unchanged dictionary s and 0 are returned.

- `b>idict!+` ($v\ x\ s\ n$ – s' -1 or s 0), adds a new key-value pair (x, v) into dictionary s similarly to `b>idict!`, but fails if the key already exists by returning the unchanged dictionary s and 0, cf. **6.3**.

- `b>udict!` ($v\ x\ s\ n$ – s' -1 or s 0), adds a new value v (represented by a *Builder*) with key given by unsigned big-endian n-bit integer x into dictionary s with n-bit keys, and returns the new dictionary s' and -1 on success, cf. **6.3**. Otherwise the unchanged dictionary s and 0 are returned.

- `b>udict!+` ($v\ x\ s\ n$ – s' -1 or s 0), adds a new key-value pair (x, v) into dictionary s similarly to `b>udict!`, but fails if the key already exists by returning the unchanged dictionary s and 0, cf. **6.3**.

- `bbitrefs` (b – $x\ y$), returns both the number of data bits x and the number of references y already stored in *Builder* b, cf. **5.2**.

- `bbits` (b – x), returns the number of data bits already stored in *Builder* b. The result x is an *Integer* in the range $0 \ldots 1023$, cf. **5.2**.

- `bl` (– x), pushes the Unicode codepoint of a space, i.e., 32, cf. **2.10**.

- `boc+>B` ($c\ x$ – B), creates and serializes a "standard" bag of cells, containing one root *Cell* c along with all its descendants, cf. **5.5**. An *Integer* parameter $0 \le x \le 31$ is used to pass flags indicating the additional options for bag-of-cells serialization, with individual bits having the following effect:

  - $+1$ enables bag-of-cells index creation (useful for lazy deserialization of large bags of cells).
  - $+2$ includes the CRC32-C of all data into the serialization (useful for checking data integrity).

- +4 explicitly stores the hash of the root cell into the serialization (so that it can be quickly recovered afterwards without a complete deserialization).
- +8 stores hashes of some intermediate (non-leaf) cells (useful for lazy deserialization of large bags of cells).
- +16 stores cell cache bits to control caching of deserialized cells.

Typical values of x are $x = 0$ or $x = 2$ for very small bags of cells (e.g., TON Blockchain external messages) and $x = 31$ for large bags of cells (e.g., TON Blockchain blocks).

- `boc>B` (c – B), serializes a small "standard" bag of cells with root *Cell* c and all its descendants, cf. **5.5**. Equivalent to `0 boc+>B`.
- `box` (x – p), creates a new *Box* containing specified value x, cf. **2.14**. Equivalent to `hole tuck !`.
- `brefs` (b – x), returns the number of references already stored in *Builder* b, cf. **5.2**. The result x is an *Integer* in the range $0 \ldots 4$.
- `brembitrefs` (b – x y), returns both the maximum number of additional data bits $0 \le x \le 1023$ and the maximum number of additional cell references $0 \le y \le 4$ that can be stored in *Builder* b, cf. **5.2**.
- `brembits` (b – x), returns the maximum number of additional data bits that can be stored in *Builder* b, cf. **5.2**. Equivalent to `bbits 1023 swap -`.
- `bremrefs` (b – x), returns the maximum number of additional cell references that can be stored in *Builder* b, cf. **5.2**.
- `bye` (–), quits the Fift interpreter to the operating system with a zero exit code, cf. **2.3**. Equivalent to `0 halt`.
- `b{`⟨*binary-data*⟩`}` (– s), creates a *Slice* s that contains no references and up to 1023 data bits specified in ⟨*binary-data*⟩, which must be a string consisting only of the characters '`0`' and '`1`', cf. **5.1**.
- `caddr` (l – h''), returns the third element of a list. Equivalent to `cddr car`.

- `cadr` (l – h'), returns the second element of a list, cf. **2.16**. Equivalent to `cdr car`.
- `car` (l – h), returns the head of a list, cf. **2.16**. Equivalent to `first`.
- `cddr` (l – t'), returns the tail of the tail of a list. Equivalent to `cdr cdr`.
- `cdr` (l – t), returns the tail of a list, cf. **2.16**. Equivalent to `second`.
- `char` ⟨*string*⟩ (– x), pushes an *Integer* with the Unicode codepoint of the first character of ⟨*string*⟩, cf. **2.10**. For instance, `char *` is equivalent to `42`.
- `chr` (x – S), returns a new *String* S consisting of one UTF-8 encoded character with Unicode codepoint x.
- `cmp` (x y – z), compares two *Integer*s x and y, and pushes 1 if $x > y$, -1 if $x < y$, and 0 if $x = y$, cf. **2.12**. Approximately equivalent to `-sgn`.
- `cond` (x e e' –), if *Integer* x is non-zero, executes e, otherwise executes e', cf. **3.2**.
- `cons` (h t – l), constructs a list from its head (first element) h and its tail (the list consisting of all remaining elements) t, cf. **2.16**. Equivalent to `pair`.
- `constant` (x –), scans a blank-delimited word name S from the remainder of the input, and defines a new ordinary word S as a constant, which will push the given value x (of arbitrary type) when invoked, cf. **4.5** and **2.7**.
- `count` (t – n), returns the length $n = |t|$ of *Tuple* t, cf. **2.15**.
- `cr` (–), outputs a carriage return (or a newline character) into the standard output, cf. **2.10**.
- `create` (e –), defines a new ordinary word with the name equal to the next word scanned from the input, using *WordDef* e as its definition, cf. **4.5**. If the word already exists, it is tacitly redefined.

- `csr.` (s –), recursively prints a *Slice* s, cf. **5.3**. On the first line, the data bits of s are displayed in hexadecimal form embedded into an `x{...}` construct similar to the one used for *Slice* literals (cf. **5.1**). On the next lines, the cells referred to by s are printed with larger indentation.

- `def?` ⟨*word-name*⟩ (– ?), checks whether the word ⟨*word-name*⟩ is defined at execution time, and returns -1 or 0 accordingly.

- `depth` (– n), returns the current depth (the total number of entries) of the Fift stack as an *Integer* $n \geq 0$.

- `dictmap` (s n e – s'), applies execution token e (i.e., an anonymous function) to each of the key-value pairs stored in a dictionary s with n-bit keys, cf. **6.3**. The execution token is executed once for each key-value pair, with a *Builder* b and a *Slice* v (containing the value) pushed into the stack before executing e. After the execution e must leave in the stack either a modified *Builder* b' (containing all data from b along with the new value v') and -1, or 0 indicating failure. In the latter case, the corresponding key is omitted from the new dictionary.

- `dictmerge` (s s' n e – s''), combines two dictionaries s and s' with n-bit keys into one dictionary s'' with the same keys, cf. **6.3**. If a key is present in only one of the dictionaries s and s', this key and the corresponding value are copied verbatim to the new dictionary s''. Otherwise the execution token (anonymous function) e is invoked to merge the two values v and v' corresponding to the same key k in s and s', respectively. Before e is invoked, a *Builder* b and two *Slice*s v and v' representing the two values to be merged are pushed. After the execution e leaves either a modified *Builder* b' (containing the original data from b along with the combined value) and -1, or 0 on failure. In the latter case, the corresponding key is omitted from the new dictionary.

- `dictnew` (– s), pushes a *Slice* that represents a new empty dictionary, cf. **6.3**. Equivalent to `b{0}`.

- `does` ($x_1 \ldots x_n$ n e – e'), creates a new execution token e' that would push n values $x_1, \ldots, x_n$ into the stack and then execute e when invoked, cf. **4.7**. It is roughly equivalent to a combination of `({)`, `(compile)`, and `(})`.

- `drop` (x –), removes the top-of-stack entry, cf. **2.5**.
- `dup` (x – x x), duplicates the top-of-stack entry, cf. **2.5**. If the stack is empty, throws an exception.
- `ed25519_chksign` (B B' B'' – ?), checks whether B' is a valid Ed25519-signature of data B with the public key B'', cf. **6.1**.
- `ed25519_sign` (B B' – B''), signs data B with the Ed25519 private key B' (a 32-byte *Bytes* value) and returns the signature as a 64-byte *Bytes* value B'', cf. **6.1**.
- `ed25519_sign_uint` (x B' – B''), converts a big-endian unsigned 256-bit integer x into a 32-byte sequence and signs it using the Ed25519 private key B' similarly to `ed25519_sign`, cf. **6.1**. Equivalent to `swap 256 u>B swap ed25519_sign`. The integer x to be signed is typically computed as the hash of some data.
- `emit` (x –), prints a UTF-8 encoded character with Unicode codepoint given by *Integer* x into the standard output, cf. **2.10**. For instance, `42 emit` prints an asterisk "`*`", and `916 emit` prints a Greek Delta "Δ". Equivalent to `chr type`.
- `empty?` (s – ?), checks whether a *Slice* is empty (i.e., has no data bits and no references left), and returns -1 or 0 accordingly, cf. **5.3**.
- `eq?` (u v – ?), checks whether u and v are equal *Integer*s, *Atom*s, or *Null*s, cf. **2.17**. If they are not equal, or if they are of different types, or not of one of the types listed, returns zero.
- `exch` (x_n $\ldots x_0$ n – x_0 $\ldots x_n$), interchanges the top of the stack with the n-th stack entry from the top, where $n \geq 0$ is also taken from the stack, cf. **2.5**. In particular, `1 exch` is equivalent to `swap`, and `2 exch` to `swap rot`.
- `exch2` ($\ldots n$ m – $\ldots$), interchanges the n-th stack entry from the top with the m-th stack entry from the top, where $n \geq 0$, $m \geq 0$ are taken from the stack, cf. **2.5**.
- `execute` (e – $\ldots$), executes the execution token (*WordDef*) e, cf. **3.1**.

- `explode` (t – $x_1 \ldots x_n$ n), unpacks a *Tuple* $t = (x_1, \ldots, x_n)$ of unknown length n, and returns that length, cf. **2.15**.

- `false` (– 0), pushes 0 into the stack, cf. **2.11**. Equivalent to `0`.

- `file-exists?` (S – ?), checks whether the file with the name specified in *String* S exists, cf. **5.6**.

- `file>B` (S – B), reads the (binary) file with the name specified in *String* S and returns its contents as a *Bytes* value, cf. **5.6**. If the file does not exist, an exception is thrown.

- `find` (S – e -1 or e 1 or 0), looks up *String* S in the dictionary and returns its definition as a *WordDef* e if found, followed by -1 for ordinary words or 1 for active words, cf. **4.6**. Otherwise pushes 0.

- `first` (t – x), returns the first component of a *Tuple*, cf. **2.15**. Equivalent to `0 []`.

- `fits` (x y – ?), checks whether *Integer* x is a signed y-bit integer (i.e., whether $-2^{y-1} \leq x < 2^{y-1}$ for $0 \leq y \leq 1023$), and returns -1 or 0 accordingly.

- `forget` (–), forgets (removes from the dictionary) the definition of the next word scanned from the input, cf. **4.5**.

- `gasrunvm` ($\ldots s$ c z – $\ldots x$ c' z'), a gas-aware version of `runvm`, cf. **6.4**: invokes a new instance of TVM with both the current continuation `cc` and the special register `c3` initialized from *Slice* s, and pushes a zero into the initial TVM stack similarly to `runvmdict`, but also initializes special register `c4` (the "root of persistent data", cf. [4, 1.4]) with *Cell* c. Then starts the new TVM instance with the gas limit set to z. The actually consumed gas z' is returned at the top of the final Fift stack, and the final value of `c4` is returned immediately below the top of the final Fift stack as another *Cell* c'.

- `gasrunvmcode` ($\ldots s$ z – $\ldots x$ z'), a gas-aware version of `runvmcode`, cf. **6.4**: invokes a new instance of TVM with the current continuation `cc` initialized from *Slice* s and with the gas limit set to z, thus executing code s in TVM. The original Fift stack (without s) is passed in its entirety as the initial stack of the new TVM instance. When TVM

terminates, its resulting stack is used as the new Fift stack, with the exit code x and the actually consumed gas z' pushed at its top. If x is non-zero, indicating that TVM has been terminated by an unhandled exception, the next stack entry from the top contains the parameter of this exception, and x is the exception code. All other entries are removed from the stack in this case.

- `gasrunvmdict` ($\ldots s\ z - \ldots x\ z'$), a gas-aware version of `runvmdict`, cf. **6.4**: invokes a new instance of TVM with the current continuation `cc` initialized from *Slice* s and sets the gas limit to z similarly to `gasrunvmcode`, but also initializes the special register `c3` with the same value, and pushes a zero into the initial TVM stack before the TVM execution begins. The actually consumed gas is returned as an *Integer* z'. In a typical application *Slice* s consists of a subroutine selection code that uses the top-of-stack *Integer* to select the subroutine to be executed, thus enabling the definition and execution of several mutually-recursive subroutines (cf. [4, 4.6] and **7.8**). The selector equal to zero corresponds to the `main()` subroutine in a large TVM program.

- `halt` (x –), quits to the operating system similarly to `bye`, but uses *Integer* x as the exit code, cf. **2.3**.

- `hash` ($c - B$), computes the SHA256-based representation hash of *Cell* c (cf. [4, 3.1]), which unambiguously defines c and all its descendants (provided there are no collisions for SHA256), cf. **5.4**. The result is returned as a *Bytes* value consisting of exactly 32 bytes.

- `hold` ($S\ x - S'$), appends to *String* S one UTF-8 encoded character with Unicode codepoint x. Equivalent to `chr $+`.

- `hole` ($- p$), creates a new *Box* p that does not hold any value, cf. **2.14**. Equivalent to `null box`.

- `i,` ($b\ x\ y - b'$), appends the big-endian binary representation of a signed y-bit integer x to *Builder* b, where $0 \le y \le 257$, cf. **5.2**. If there is not enough room in b (i.e., if b already contains more than $1023 - y$ data bits), or if *Integer* x does not fit into y bits, an exception is thrown.

- `i>B` ($x\ y - B$), stores a signed big-endian y-bit *Integer* x into a *Bytes* value B consisting of exactly $y/8$ bytes. Integer y must be a multiple of eight in the range $0 \ldots 256$, cf. **5.6**.

- `i@` ($s\ x - y$), fetches a signed big-endian x-bit integer from the first x bits of *Slice* s, cf. **5.3**. If s contains less than x data bits, an exception is thrown.

- `i@+` ($s\ x - y\ s'$), fetches a signed big-endian x-bit integer from the first x bits of *Slice* s similarly to `i@`, but returns the remainder of s as well, cf. **5.3**.

- `i@?` ($s\ x - y\ -1$ or 0), fetches a signed big-endian integer from a *Slice* similarly to `i@`, but pushes integer -1 afterwards on success, cf. **5.3**. If there are less than x bits left in s, pushes integer 0 to indicate failure.

- `i@?+` ($s\ x - y\ s'\ -1$ or $s\ 0$), fetches a signed big-endian integer from *Slice* s and computes the remainder of this *Slice* similarly to `i@+`, but pushes -1 afterwards to indicate success, cf. **5.3**. On failure, pushes the unchanged *Slice* s and 0 to indicate failure.

- `idict!` ($v\ x\ s\ n - s'\ -1$ or $s\ 0$), adds a new value v (represented by a *Slice*) with key given by signed big-endian n-bit integer x into dictionary s with n-bit keys, and returns the new dictionary s' and -1 on success, cf. **6.3**. Otherwise the unchanged dictionary s and 0 are returned.

- `idict!+` ($v\ x\ s\ n - s'\ -1$ or $s\ 0$), adds a new key-value pair (x, v) into dictionary s similarly to `idict!`, but fails if the key already exists by returning the unchanged dictionary s and 0, cf. **6.3**.

- `idict@` ($x\ s\ n - v\ -1$ or 0), looks up key represented by signed big-endian n-bit *Integer* x in the dictionary represented by *Slice* s, cf. **6.3**. If the key is found, returns the corresponding value as a *Slice* v and -1. Otherwise returns 0.

- `if` ($x\ e -$), executes execution token (i.e., a *WordDef*) e, but only if *Integer* x is non-zero, cf. **3.2**.

- `ifnot` ($x\ e -$), executes execution token e, but only if *Integer* x is zero, cf. **3.2**.

- `include` ($S -$), loads and interprets a Fift source file from the path given by *String* S, cf. **7.1**. If the filename S does not begin with a slash, the Fift include search path, typically taken from the `FIFTPATH`

environment variable or the `-I` command-line argument of the Fift interpreter (and equal to `/usr/lib/fift` if both are absent), is used to locate S.

- `list` ($x_1 \ldots x_n\ n - l$), constructs a list l of length n with elements $x_1, \ldots, x_n$, in that order, cf. **2.16**. Equivalent to `null ' cons rot times`.
- `max` ($x\ y - z$), computes the maximum $z := \max(x, y)$ of two *Integer*s x and y. Equivalent to `minmax nip`.
- `min` ($x\ y - z$), computes the minimum $z := \min(x, y)$ of two *Integer*s x and y. Equivalent to `minmax drop`.
- `minmax` ($x\ y - z\ t$), computes both the minimum $z := \min(x, y)$ and the maximum $t := \max(x, y)$ of two *Integer*s x and y.
- `mod` ($x\ y - r := x \bmod y$), computes the remainder $x \bmod y = x - y \cdot \lfloor x/y \rfloor$ of division of x by y, cf. **2.4**.
- `negate` ($x - -x$), changes the sign of an *Integer*, cf. **2.4**.
- `newkeypair` ($-\ B\ B'$), generates a new Ed25519 private/public key pair, and returns both the private key B and the public key B' as 32-byte *Bytes* values, cf. **6.1**. The quality of the keys is good enough for testing purposes. Real applications must feed enough entropy into OpenSSL PRNG before generating Ed25519 keypairs.
- `nil` ($- t$), pushes the empty *Tuple* $t = ()$. Equivalent to `0 tuple`.
- `nip` ($x\ y - y$), removes the second stack entry from the top, cf. **2.5**. Equivalent to `swap drop`.
- `nop` ($-$), does nothing, cf. **4.6**.
- `not` ($x - -1 - x$), computes the bitwise complement of an *Integer*, cf. **2.4**.
- `now` ($- x$), returns the current Unixtime as an *Integer*, cf. **6.1**.
- `null` ($- \perp$), pushes the *Null* value, cf. **2.16**

- `null!` (p –), stores a *Null* value into *Box* p. Equivalent to `null swap !`.
- `null?` (x – ?), checks whether x is *Null*, cf. **2.16**.
- `or` (x y – $x|y$), computes the bitwise OR of two *Integer*s, cf. **2.4**.
- `over` (x y – x y x), creates a copy of the second stack entry from the top over the top-of-stack entry, cf. **2.5**.
- `pair` (x y – t), creates new pair $t = (x, y)$, cf. **2.15**. Equivalent to `2 tuple` or to `| rot , swap ,`.
- `pfxdict!` (v k s n – s' -1 or s 0), adds key-value pair (k, v), both represented by *Slice*s, into a prefix dictionary s with keys of length at most n, cf. **6.3**. On success, returns the modified dictionary s' and -1. On failure, returns the original dictionary s and 0.
- `pfxdict!+` (v k s n – s' -1 or s 0), adds key-value pair (k, v) into prefix dictionary s similarly to `pfxdict!`, but fails if the key already exists, cf. **6.3**.
- `pfxdict@` (k s n – v -1 or 0), looks up key k (represented by a *Slice*) in the prefix dictionary s with the length of keys limited by n bits, cf. **6.3**. On success, returns the value found as a *Slice* v and -1. On failure, returns 0.
- `pick` ($x_n \ldots x_0$ n – $x_n \ldots x_0$ x_n), creates a copy of the n-th entry from the top of the stack, where $n \geq 0$ is also passed in the stack, cf. **2.5**. In particular, `0 pick` is equivalent to `dup`, and `1 pick` to `over`.
- `priv>pub` (B – B'), computes the public key corresponding to a private Ed25519 key, cf. **6.1**. Both the public key B' and the private key B are represented by 32-byte *Bytes* values.
- `quit` (... –), exits to the topmost level of the Fift interpreter (without printing an `ok` in interactive mode) and clears the stack, cf. **2.3**.
- `ref,` (b c – b'), appends to *Builder* b a reference to *Cell* c, cf. **5.2**. If b already contains four references, an exception is thrown.

- `ref@` (s – c), fetches the first reference from the *Slice* s and returns the *Cell* c referred to, cf. **5.3**. If there are no references left, throws an exception.

- `ref@+` (s – s' c), fetches the first reference from the *Slice* s similarly to `ref@`, but returns the remainder of s as well, cf. **5.3**.

- `ref@?` (s – c -1 or 0), fetches the first reference from the *Slice* s similarly to `ref@`, but uses a flag to indicate failure instead of throwing an exception, cf. **5.3**.

- `ref@?+` (s – s' c -1 or s 0), similar to `ref@+`, but uses a flag to indicate failure instead of throwing an exception, cf. **5.3**.

- `remaining` (s – x y), returns both the number of data bits x and the number of cell references y remaining in the *Slice* s, cf. **5.3**.

- `reverse` ($x_1 \ldots x_n$ $y_1 \ldots y_m$ n m – $x_n \ldots x_1$ $y_1 \ldots y_m$), reverses the order of n stack entries located immediately below the topmost m elements, where both $0 \le m, n \le 255$ are passed in the stack.

- `roll` ($x_n \ldots x_0$ n – $x_{n-1} \ldots x_0$ x_n), rotates the top n stack entries, where $n \ge 0$ is also passed in the stack, cf. **2.5**. In particular, `1 roll` is equivalent to `swap`, and `2 roll` to `rot`.

- `rot` (x y z – y z x), rotates the three topmost stack entries.

- `runvmcode` ($\ldots s$ – $\ldots x$), invokes a new instance of TVM with the current continuation `cc` initialized from *Slice* s, thus executing code s in TVM, cf. **6.4**. The original Fift stack (without s) is passed in its entirety as the initial stack of the new TVM instance. When TVM terminates, its resulting stack is used as the new Fift stack, with the exit code x pushed at its top. If x is non-zero, indicating that TVM has been terminated by an unhandled exception, the next stack entry from the top contains the parameter of this exception, and x is the exception code. All other entries are removed from the stack in this case.

- `runvmdict` ($\ldots s$ – $\ldots x$), invokes a new instance of TVM with the current continuation `cc` initialized from *Slice* s similarly to `runvmcode`, but also initializes the special register `c3` with the same value, and

pushes a zero into the initial TVM stack before start, cf. **6.4**. In a typical application *Slice* s consists of a subroutine selection code that uses the top-of-stack *Integer* to select the subroutine to be executed, thus enabling the definition and execution of several mutually-recursive subroutines (cf. [4, 4.6] and **7.8**). The selector equal to zero corresponds to the `main()` subroutine in a large TVM program.

- `runvm` ($\ldots s\ c$ – $\ldots x\ c'$), invokes a new instance of TVM with both the current continuation `cc` and the special register `c3` initialized from *Slice* s, and pushes a zero into the initial TVM stack similarly to `runvmdict`, but also initializes special register `c4` (the "root of persistent data", cf. [4, 1.4]) with *Cell* c, cf. **6.4**. The final value of `c4` is returned at the top of the final Fift stack as another *Cell* c'. In this way one can emulate the execution of smart contracts that inspect or modify their persistent storage.

- `s,` ($b\ s$ – b'), appends data bits and references taken from *Slice* s to *Builder* b, cf. **5.2**.

- `s>` (s –), throws an exception if *Slice* s is non-empty, cf. **5.3**. It usually marks the end of the deserialization of a cell, checking whether there are any unprocessed data bits or references left.

- `s>c` (s – c), creates a *Cell* c directly from a *Slice* s, cf. **5.3**. Equivalent to `<b swap s, b>`.

- `sbitrefs` (s – $x\ y$), returns both the number of data bits x and the number of cell references y remaining in *Slice* s, cf. **5.3**. Equivalent to `remaining`.

- `sbits` (s – x), returns the number of data bits x remaining in *Slice* s, cf. **5.3**.

- `second` (t – x), returns the second component of a *Tuple*, cf. **2.15**. Equivalent to `1 []`.

- `sgn` (x – y), computes the sign of an *Integer* x (i.e., pushes 1 if $x > 0$, -1 if $x < 0$, and 0 if $x = 0$), cf. **2.12**. Equivalent to `0 cmp`.

- `shash` (s – B), computes the SHA256-based representation hash of a *Slice* by first transforming it into a cell, cf. **5.4**. Equivalent to `s>c hash`.

- `sign` (S x – S'), appends a minus sign "-" to *String* S if *Integer* x is negative. Otherwise leaves S intact.

- `single` (x – t), creates new singleton $t = (x)$, i.e., a one-element *Tuple*. Equivalent to `1 tuple`.

- `skipspc` (–), skips blank characters from the current input line until a non-blank or an end-of-line character is found.

- `smca>$` (x y z – S), packs a standard TON smart-contract address with workchain x (a signed 32-bit *Integer*) and in-workchain address y (an unsigned 256-bit *Integer*) into a 48-character string S (the human-readable representation of the address) according to flags z, cf. **6.2**. Possible individual flags in z are: +1 for non-bounceable addresses, +2 for testnet-only addresses, and +4 for base64url output instead of base64.

- `space` (–), outputs a single space. Equivalent to `bl emit` or to `." "`.

- `sr,` (b s – b'), constructs a new *Cell* containing all data and references from *Slice* s, and appends a reference to this cell to *Builder* b, cf. **5.2**. Equivalent to `s>c ref,`.

- `srefs` (s – x), returns the number of cell references x remaining in *Slice* s, cf. **5.3**.

- `swap` (x y – y x), interchanges the two topmost stack entries, cf. **2.5**.

- `ten` (– 10), pushes *Integer* constant 10.

- `third` (t – x), returns the third component of a *Tuple*, cf. **2.15**. Equivalent to `2 []`.

- `times` (e n –), executes execution token (*WordDef*) e exactly n times, if $n \geq 0$, cf. **3.3**. If n is negative, throws an exception.

- `triple` (x y z – t), creates new triple $t = (x, y, z)$, cf. **2.15**. Equivalent to `3 tuple`.

- `true` (– −1), pushes −1 into the stack, cf. **2.11**. Equivalent to `-1`.
- `tuck` (x y – y x y), equivalent to `swap over`, cf. **2.5**.
- `tuple` ($x_1 \ldots x_n$ n – t), creates new *Tuple* $t := (x_1, \ldots, x_n)$ from $n \geq 0$ topmost stack values, cf. **2.15**. Equivalent to `dup 1 reverse | { swap , } rot times`, but more efficient.
- `tuple?` (t – ?), checks whether t is a *Tuple*, and returns −1 or 0 accordingly.
- `type` (s –), prints a *String* s taken from the top of the stack into the standard output, cf. **2.10**.
- `u,` (b x y – b'), appends the big-endian binary representation of an unsigned y-bit integer x to *Builder* b, where $0 \leq y \leq 256$, cf. **5.2**. If the operation is impossible, an exception is thrown.
- `u>B` (x y – B), stores an unsigned big-endian y-bit *Integer* x into a *Bytes* value B consisting of exactly $y/8$ bytes. Integer y must be a multiple of eight in the range $0 \ldots 256$, cf. **5.6**.
- `u@` (s x – y), fetches an unsigned big-endian x-bit integer from the first x bits of *Slice* s, cf. **5.3**. If s contains less than x data bits, an exception is thrown.
- `u@+` (s x – y s'), fetches an unsigned big-endian x-bit integer from the first x bits of *Slice* s similarly to `u@`, but returns the remainder of s as well, cf. **5.3**.
- `u@?` (s x – y −1 or 0), fetches an unsigned big-endian integer from a *Slice* similarly to `u@`, but pushes integer −1 afterwards on success, cf. **5.3**. If there are less than x bits left in s, pushes integer 0 to indicate failure.
- `u@?+` (s x – y s' −1 or s 0), fetches an unsigned big-endian integer from *Slice* s and computes the remainder of this *Slice* similarly to `u@+`, but pushes −1 afterwards to indicate success, cf. **5.3**. On failure, pushes the unchanged *Slice* s and 0 to indicate failure.

- `udict!` ($v\ x\ s\ n$ – s' -1 or s 0), adds a new value v (represented by a *Slice*) with key given by big-endian unsigned n-bit integer x into dictionary s with n-bit keys, and returns the new dictionary s' and -1 on success, cf. **6.3**. Otherwise the unchanged dictionary s and 0 are returned.

- `udict!+` ($v\ x\ s\ n$ – s' -1 or s 0), adds a new key-value pair (x, v) into dictionary s similarly to `udict!`, but fails if the key already exists by returning the unchanged dictionary s and 0, cf. **6.3**.

- `udict@` ($x\ s\ n$ – v -1 or 0), looks up key represented by unsigned big-endian n-bit *Integer* x in the dictionary represented by *Slice* s, cf. **6.3**. If the key is found, returns the corresponding value as a *Slice* v and -1. Otherwise returns 0.

- `ufits` ($x\ y$ – ?), checks whether *Integer* x is an unsigned y-bit integer (i.e., whether $0 \leq x < 2^y$ for $0 \leq y \leq 1023$), and returns -1 or 0 accordingly.

- `uncons` (l – $h\ t$), decomposes a non-empty list into its head and its tail, cf. **2.16**. Equivalent to `unpair`.

- `undef?` ⟨*word-name*⟩ (– ?), checks whether the word ⟨*word-name*⟩ is undefined at execution time, and returns -1 or 0 accordingly.

- `unpair` (t – $x\ y$), unpacks a pair $t = (x, y)$, cf. **2.15**. Equivalent to `2 untuple`.

- `unsingle` (t – x), unpacks a singleton $t = (x)$. Equivalent to `1 untuple`.

- `until` (e –), an until loop, cf. **3.4**: executes *WordDef* e, then removes the top-of-stack integer and checks whether it is zero. If it is, then begins a new iteration of the loop by executing e. Otherwise exits the loop.

- `untriple` (t – $x\ y\ z$), unpacks a triple $t = (x, y, z)$, cf. **2.15**. Equivalent to `3 untuple`.

- `untuple` ($t\ n$ – $x_1\ \ldots x_n$), returns all components of a *Tuple* $t = (x_1, \ldots, x_n)$, but only if its length is equal to n, cf. **2.15**. Otherwise throws an exception.

- `variable` (–), scans a blank-delimited word name S from the remainder of the input, allocates an empty *Box*, and defines a new ordinary word S as a constant, which will push the new *Box* when invoked, cf. **2.14**. Equivalent to `hole constant`.

- `while` (e e' –), a while loop, cf. **3.4**: executes *WordDef* e, then removes and checks the top-of-stack integer. If it is zero, exits the loop. Otherwise executes *WordDef* e', then begins a new loop iteration by executing e and checking the exit condition afterwards.

- `word` (x – s), parses a word delimited by the character with the Unicode codepoint x from the remainder of the current input line and pushes the result as a *String*, cf. **2.10**. For instance, `bl word abracadabra type` will print the string "`abracadabra`". If $x = 0$, skips leading spaces, and then scans until the end of the current input line. If $x = 32$, skips leading spaces before parsing the next word.

- `words` (–), prints the names of all words currently defined in the dictionary, cf. **4.6**.

- `x.` (x –), prints the hexadecimal representation (without the `0x` prefix) of an *Integer* x, followed by a single space. Equivalent to `x._ space`.

- `x._` (x –), prints the hexadecimal representation (without the `0x` prefix) of an *Integer* x without any spaces. Equivalent to `(x.) type`.

- `xor` (x y – $x \oplus y$), computes the bitwise eXclusive OR of two *Integer*s, cf. **2.4**.

- `x{`⟨*hex-data*⟩`}` (– s), creates a *Slice* s that contains no references and up to 1023 data bits specified in ⟨*hex-data*⟩, cf. **5.1**. More precisely, each hex digit from ⟨*hex-data*⟩ is transformed into four binary digits in the usual fashion. After that, if the last character of ⟨*hex-data*⟩ is an underscore `_`, then all trailing binary zeroes and the binary one immediately preceding them are removed from the resulting binary string (cf. [4, 1.0] for more details). For instance, `x{6C_}` is equivalent to `b{01101}`.

- `{` (– l), an active word that increases internal variable `state` by one and pushes a new empty *WordList* into the stack, cf. **4.7**.

- `|` (– t), creates an empty *Tuple* $t = ()$, cf. **2.15**. Equivalent to `nil` and to `0 tuple`.
- `|+` (s s' – s''), concatenates two *Slice*s s and s', cf. **5.1**. This means that the data bits of the new *Slice* s'' are obtained by concatenating the data bits of s and s', and the list of *Cell* references of s'' is constructed similarly by concatenating the corresponding lists for s and s'. Equivalent to `<b rot s, swap s, b> <s`.
- `|_` (s s' – s''), given two *Slice*s s and s', creates a new *Slice* s'', which is obtained from s by appending a new reference to a *Cell* containing s', cf. **5.1**. Equivalent to `<b rot s, swap s>c ref, b> <s`.
- `}` (l – e), an active word that transforms a *WordList* l into a *WordDef* (an execution token) e, thus making all further modifications of l impossible, and decreases internal variable `state` by one; then pushes the integer 1, followed by a `'nop`, cf. **4.7**. The net effect is to transform the constructed *WordList* into an execution token and push this execution token into the stack, either immediately or during the execution of an outer block.