TVM Assembler Phase 1 (Specification) Report

Prepared by Pruvendo at 12/10/21

Executive summary

The present document represents the informal specification for the TVM Assembler module (crate) implemented by TON Labs.
The purpose of this document is to both to describe the features of the module as well as to prepare it for the full-scale formal verification.

Project description

The project being described is a TVM assembler compatible with the following description of the machine. The project allows the manual creation of the assembler code according to the mnemonics described in the document mentioned above, however, its usage is considered as exceptional and no friendly tools are provided. More, **it's not a standalone tool** so it's calling is operated by some external toolchain that is located beyond the current document.

For the standard TonLabs build toolchain it stays between the compiler and the linker.

Reference

The present specification is based on the link https://github.com/tonlabs/ton-labs-assembler with the hash 60e5d3aff02aa5c0aba8061b6a3a96ad02e46ef9.

Installation

To install the library being discussed :

- setup rust
- build the crate using cargo tool[1]

Running

The assembler crate can be used by any Rust application that imports it. The public API is defined in the corresponding section. Alternatively, to get better understanding of the crate features the user can implement and run tests such as:

```
#[test]
    pub fn test_over() {
        let result = compile_code("OVER");
        assert_eq!(result.is_err(), false);
        println!("{}", result.ok().unwrap());
```

[1] For beginners it's advised to use IDEA Ultimate with Rust plugin

```
    }
```

Key terminology[2]

Cells, builders and slices

All the data stored in the TVM (including code)(with some exceptions as a stack or registry) is saved as a **cell** (or, more correctly, as a tree of cells). A cell is a container that keeps up to 1023 bits of arbitrary data and up to four references to other cells. Such, any kind of data can be stored in a form of a tree of cells (throughout the present document the tree of cells is called a cell (referring to the root one)).
The cells are immutable and can not be read directly. The only way to read it is to convert the cell into a **slice**.

Slice is a read-only entity that can be read only once. So, at each step the application can read some amount of data (starting from the beginning) and/or some references (starting from the first one). Upon reading, the consumed data (and references) are removed from the slice, allowing the next portion of data to be read next time. When completely read, the slice becomes empty and does not contain any information anymore. Slices can be roughly compared to the input streams in many traditional languages such as Java.

Builders are entities opposite to slices and intended to create new cells. The newly created builder is usually empty (while some TVM primitives allow creating it with some data and/or references) and then the application can add (to the end only, no insertions in the beginning or in the middle) some additional data and/or references. When all the required information has been written to the builder, it can be converted into the cell.

Types of cells

Each cell can have one of the following types:

- Unknown
- Ordinary
- Prune
- Library reference
- Merkle proof

[2] The present document provides extremely basic explanation of the TVM concepts and entities. For the detailed explanation refer to https://test.ton.org/tvm.pdf

- Merkle update

Most of these types are intended for rather specific usage and called exotic so for the purposes of the present document all the cells must be ordinary.

Cell level

Each cell has a special attribute called a level. For ordinary cells it's just the highest level of their children so for the tree of ordinary cells the level is always zero. For exotic cells it can be different but this discussion is beyond the present document and it can be assumed that this level is always zero.

Serialization and deserialization

Each cell, slice or builder (or tree of cells with a root cell, slice or builder) can be serialized into a string. And, in reverse, each string can be transformed into a cell, slice or builder. The way of serialization and deserialization should be clearly specified[3], but for the purposes of the present project the following statements are considered as true:

- Any string has a single "canonical" representation as a cell (slice or builder)
- Any cell (slice or builder) has a single "canonical" representation as a string
- "Canonical" representation of the "canonical" representation of some string is the string itself

Transformation between cells, slices and builders

- Each slice in any state has the only equivalent as a cell
- Each builder in any state has the only equivalent as a cell

Thus, the cells, slices and builders can be considered as different representations of the same entity.

Continuations

Continuation is a sequence of TVM primitives with parameters, in other words it's a bytecode. The simplest example of continuation is a program itself. However, the other kinds of continuation exist, such as branches of the `IF` family of primitives, bodies of loops,

[3] As a scope of the another project

subprograms (called by `CALL` and `JMP` families of primitives) etc. In these cases the continuations act as parameters of the corresponding primitives.

Registries

TVM has 16 control registries; each of them has its own intention. Only the first eight of them are used while others are reserved for future use. For convenience, all the control registries are named as `C0`, `C1`, …, `C15`. The control registries (together with the stack discussed below) fully define the state of the virtual machine such as a current continuation, global variables, temporary data etc. Some primitives use the registry number as a parameter.

Stack

TVM is a stack-based virtual machine so most of the intermediate operations are performed on a stack. For example, if it's required to calculate a sum of two numbers, the programmer should push both of them into the stack and then execute `ADD` primitive. The following types of data can be located at the stack:

- 257-bit numbers (that can be interpreted either as 256-bit signed (by module) or 257-bit unsigned, depending on the context)
- Builders
- Slices
- Cells
- Continuations
- Tuples
- NULL

While the depth of the stack can be very high, the primitives that operate with the certain element on the stack can use at most the first 18 elements (for a few primitives 256 elements are supported). For convenience, these elements are called `S0`, `S1`, …, `S17`, where `S0` is the top element on the stack.

Representation of bit sequences

TVM is a bit (not byte) based virtual machine. So when the bit sequence has a number of bits not divisible by 8, its representation as a sequence of hexadecimal digits can be impossible. To avoid this limitation the completion tag (`_`) was introduced. So the following sequence is added to the end of the bit sequence: `1` + minimal required number of 0's to make the total length of the sequence divisible by 8. As an example, the empty string of bits will be represented as `x80_`, `10110111011` is `xB770_`.

If the sequence does not contain a completion tag it’s just considered as a sequence of 4-bit octets in a natural order.

Requirements

Types

The following types must be supported by the calling side (it’s to be done automatically by importing `ton_labs_assembler` (such as `use ton_labs_assembler::compile_code;`)):

Type	Desciption	Internal structure
`Cell`	Implementation of a cell paradigm described above	Defined in `ton-labs-types` (file `mod.rs`) project[4]
`BuilderData`	Implementation of a builder paradigm described above	Has the following fields: ● `data` - data for the builder ● `length_in_bits` - length of the data ● `references` - references for cells ● `cell_type` - cell type ● `level_mask` - cell level
`SliceData`	Implementation of a slice paradigm described above	Has the following fields: ● `cell` - `Cell`[5] object ● `data_window` - the range of the unread part ● `references_window` - the range of the unread references
`Position`	Represents position of the file being compiled	Has the following fields:

[4] The specification for types is beyond the present project and will be described by a special one
[5] Described above

		- `filename` - name of the file - `line` - line in the file - `column` - column in the file
`ParameterError`	Enum that the represents types of parameter errors	The values are: - `UnexpectedType` - `NotSupported` - `OutOfRange`
`OperationError`	Enum that represents types of operational errors	The values are: - `Parameter (String, ParameterError)` - `TooManyParameters` - `LogicErrorInParameters(String)` - `MissingRequiredParameters` - `MissingBlock` - `Nested (Box`[6]`<CompileError`[7]`>)` - `NotFitInSlice`
`CompileError`	The high-level enum that represents any kind of compilation errors	The values are: - `Syntax ( Position, String)` - `UnknownOperation( Position,String )` - `Operation (Position, String, OperationError)`
`DbgPos`	Represents the position for the debug information	Has the following fields: - `filename` - file that is being compiled - `line` - the line the

[6] The rust boxes are described here

[7] Described below

		debug information is provided to • `line_code` - line number in a source code (in Solidity or other original language)
`DbgInfo`	The overall debug information	A map that has an cell hash (in a hexadecimal form) as a key and a map as values (offset as a key and `DbgPos` as a value)
`Line`	The line of code with debug information	Has the following fields: • `text` - contest of the line • `pos` - `DbgPos` of the line
`Lines`	The code to be compiled as a vector of `Line`	`Vec`[8] of `Line`

Public API

This section defines the public API for the module being specified as well as the most common features.

API functions

The following functions are defined as a public API.

compile_code

Input parameters:

- `code` - the assembler source code to be compiled as a string

Output:

- The `Result`[9] object that in case of:
    - `OK` - represents the compiled code as a `SliceData`

[8] https://doc.rust-lang.org/std/vec/struct.Vec.html

[9] https://doc.rust-lang.org/std/result/

    - `Error` - represents an error as a `CompileError`

compile_code_to_cell

Input parameters:

- `code` - the assembler source code to be compiled as a string

Output:

- The `Result` object that in case of:
    - `OK` - represents the compiled code as a `Cell`
    - `Error` - represents an error as a `CompileError`

compile_code_to_builder

Input parameters:

- `code` - the assembler source code to be compiled as a string

Output:

- The `Result` object that in case of:
    - `OK` - represents the compiled code as a `BuilderData`
    - `Error` - represents an error as a `CompileError`

compile_code_debuggable

Input parameters:

- `code` - the assembler source code to be compiled as a `Lines`

Output:

- The `Result` object that in case of:
    - `OK` - represents the compiled code as a pair of:
        - compiled code as a `SliceData`
        - debug information as a `DbgInfo`
    - `Error` - represents an error as a `CompileError`

API equivalence

Definitions

As it was mentioned above, cells, slices and builders are bijective to each other. We call all the objects participating in these bijective relationships as EQUAL.

Also we call all the objects having the same set of primitive properties (numbers, strings and hash codes) as SAME.

For `Lines` the STRING is defined as a concatenation of all the `text` in this vector (separated by `\n`).

Requirements

APIEQ.1	For any failed compilation of `compile_code` : • `compile_code_to_cell` MUST also fail for the same input • `compile_code_to_builder` MUST also fail for the same input • `compile_code_debuggable` MUST also fail for the same input as `STRING` (with the allowed differences when the position is used) • `compile_code_to_cell` MUST have the SAME `CompileError` for the same input • `compile_code_to_builder` MUST have the SAME `CompileError` for the same input • `compile_code_debuggable` MUST have the SAME `CompileError` for the same input as `STRING`
APIEQ.2	For any passed compilation of `compile_code` : • `compile_code_to_cell` MUST also pass for the same input • `compile_code_to_builder` MUST also pass for the same input • `compile_code_debuggable` MUST also pass for the same input as `STRING` • `compile_code_to_cell` MUST have the EQUAL `Cell` for the same input • `compile_code_to_builder` MUST have the EQUAL `BuilderData` for the same input • `compile_code_debuggable` MUST have the SAME `SliceData` for the same input as `STRING`

Thus, as the equivalence of the API functions is defined above, **the rest of the specification is related exclusively to the `compile_code_debuggable` as to the most generic one.**

API requirements

CMN.1	The resulting cell[10] is an ordinary cell
CMN.2	The resulting cell has a zero level

Comments

The rest of each line after `;` character (and before `\n` if it exists) is considered as a comment (including `;` character) and excluded from the further parsing. Introducing the following terms:

- The UNCOMMENTED line is the line with the following attributes compared to the original line:
    - `text` - depending on the position of `;` in the `text` of the original line. If `;` :
        - is the first character - empty string
        - exists somewhere in the string - substring from the beginning of the original `text` till its first occurrence (excluding)
        - is absent - original `text`
    - `pos` - original `pos`
- The UNCOMMENTED lines is a STRING of a `Vec` of UNCOMMENTED line

CMT.1	The output of `compile_code_debuggable` for UNCOMMENTED lines is the SAME as for original lines

Taking into account the requirement above **for the rest of the present specification we'll consider exclusively the UNCOMMENTED lines**.

Position

Each character in the input (both in a string form and in a form of lines) has a position that has three attributes: `line`, `column` and `filename`.

[10] Or its slice/builder equivalent, will not be mentioned further

For the lines form:

- `line` - `line_code` value of the `pos` attribute of the corresponding line
- `column` - 1-based index of the character in `text` attribute of the corresponding line
- `filename` - `filename` value of the `pos` attribute of the corresponding line

For the string form:

- `line` - number of the `\n` characters before the specified one increased by 1
- `column` - number of the characters between the last `\n` and the specified one increased by 1
- `filename` - empty string

Whitespaces

The following characters are called whitespaces: ` `, `\t`, `\r`, `\n`.

Let's call lines as NORMALIZED if the following actions were subsequently applied (assuming the lines are not empty):

1. All the `text` attributes are concatenated with `\n` as a separator
2. Convert all the whitespaces into ` `.
3. In case of multiple subsequent whitespaces leave just one
4. Put the resulting string as a `text` attribute of a newly created line
5. Leave `pos` attribute arbitrary
6. Create resulting lines as a one-sized `Vec` from the newly created line
7. Call the resulting lines as NORMALIZED original lines

For the empty lines the NORMALIZED lines are empty lines as well.

WSE.1	In case `compile_code_debuggable` is successful for certain lines, this method is also successful for NORMALIZED lines and the output slices are the SAME
WSE.2	In case `compile_code_debuggable` fails for certain lines, this method also fails for NORMALIZED lines and their `CompileError`'s are the SAME with the following exception: if position information is used it should be mapped accordingly so any position that points to a specific character in the original lines should point to the same (in case it's a whitespace it can be converted to ` `, or even eliminated as a double whitespace, in the latter case the survivor of this sequence of whitespaces must be used) character for the NORMALIZED case

Taking into account the requirements above **for the rest of the present specification, we'll consider exclusively the NORMALIZED lines** (but the importing part specifying the debug information and some other cases where such a case is clearly indicated).

Blocks

Informally speaking, blocks are part of the input STRING that are bracketed by `{` and `}` characters. They are used as parameters for some primitives.

BLOCK COUNTER for the specified character is a difference between the number of occurrences of `{` before the specified character (including itself) and the number of occurrences of `}` before the specified character (including itself).

The BLOCK RUINER is the first character in the input STRING where the BLOCK COUNTER becomes negative.

BLK.1	(**IMPORTANT!!! position from not NORMALIZED input is used here**). In case BLOCK RUINER exists the compilation should fail with `Syntax CompileError` with the following attributes: ● pos - BLOCK RUINER position (in the original not NORMALIZED input) ● explanation - `}`
BLK.2	(**IMPORTANT!!! position from not NORMALIZED input is used here**). If the BLOCK COUNTER for the last character in STRING is positive the compilation should fail with `Syntax CompileError` with the following attributes: ● pos - last character position (in the original not NORMALIZED input) ● explanation - `{`

Lexemes

The STRING for lines (as a remainder UNCOMMENTED NORMALIZED lines are considered) can be considered as a sequence of lexemes possibly separated by the whitespace. The following lexemes are valid:

- Alphanumeric - the sequence of Latin letters (uppercase and lowercase characters are considered as the same), digits (0-9), and the following characters: `-`, `_` and `.`
- Blocks - `{` and `}`
- Commas - `,`

LEX.1	The alphanumeric lexemes are as long as they are valid. For example, `abcdef` is one lexeme and not, say, two (`abc` and `def`)
LEX.2	Whitespace character breaks any lexeme
LEX.3	(**IMPORTANT!!! position from not NORMALIZED input is used here**). In case a character can not be recognized as a part of a valid lexeme, the compilation fails and a `Syntax CompileError` with the following attributes is generated: ● pos - unrecognized character position (in the original not NORMALIZED input) ● explanation - `Bad char`
LEX.4	Result of any compilation with all the characters from the input STRING uppercased is the SAME as the result of the compilation with the original STRING

Taking into account the requirements above, **all the Latin letters are considered as capital for the rest of the present document**.

Grammar

Parsing

Parsing is the process of transforming a lexeme into some internal representation (a sequence of bytes). Each type of parsing is described below.

Parsing of decimal numbers

Generic approach

The specific parsing rules are defined by the following parameters:

- `min_value` - the minimal allowed value

- `max_value` - the maximal allowed value
- `base_value` - the “shifted” zero[11]
- `bits` - a number of bits in the result of the parsing

The alphanumeric lexeme to be parsed must either:

- contain digits only
- start with `-` while all the other characters are digits
- at least one digit must exist

Such a lexeme is called NUMERIC.

The VALUE of NUMERIC lexem is calculated by the following formula: $s \sum_{i=1}^{n} c_i 10^{n-i}$, where:

- s - `-1` if the lexeme starts with `-`, `1` otherwise
- n - number of digits
- c_i - the ith digit in the lexeme

The BIT VALUE of NUMERIC WITH `base_value`, `bits` is a big-endian representation of the VALUE of NUMERIC subtracted by `base_value` with the number of bits is :

- if `bits` is divisible by 8 - `bits`[12]
- otherwise - $bits + 8 - bits \bmod 8$ while all the unused higher bits are zero.

The PARSING OF `numeric` WITH `min_value`, `max_value`, `base_value`, `bits` is:

- if numeric is not NUMERIC then `UnexpectedType ParameterError`
- if VALUE of `numeric` is less than `min_value` then `OutOfRange ParameterError`
- if VALUE of `numeric` is more than `max_value` then `OutOfRange ParameterError`
- Otherwise, it’s a BIT VALUE OF `numeric` WITH `base_value`, `bits`

Kinds of decimal parsing

For the convenience, the following shortcuts are introduced. All the shortcuts have the following meaning: `<SHORTCUT> numeric = PARSING OF numeric WITH <specific min_value>, <specific max_value>, <specific base_value>, <specific bits>`. The full list of shortcuts is defined below:

[11] It can be a bit tricky to understand but consider the following example. You have a four-bit value that never can be 0, so normally you can store just 15 kinds of values. But you don’t want to waste the 16th, so you do the following trick: save 1 as 0, 2 as 1, …, 16 as 15, thus getting all the 16 kinds. The value of such an “effective zero” is called a `base_value`

[12] In can be represented either in signed or unsigned form, depending on the context

Shortcut	`min_value`	`max_value`	`base_value`	`bits`
`U2`	0	3	0	4
`U4-1`	-1	14	-1	4
`U4`	0	15	0	4
`U4+1`	1	16	1	4
`U4+2`	2	17	2	4
`U4-`	0	14	0	4
`U4++1-1`	1	15	0	4
`U4++2-2`	2	14	0	4
`U4++1-2`	1	14	0	4
`U4++1`	1	16	0	4
`U4--5`	-5	10	0	4
`U5`	0	31	0	5
`U6`	0	63	0	6
`U10`	0	1023	0	10
`U11`	0	2047	0	11
`U14`	0	16383	0	14
`U8-15`	-15	239	0	8
`I8`	-128	127	0	8
`U8+1`	1	256	1	8
`U8--15`	0	239	0	8
`I30`	-2^{29}	$2^{29}-1$	0	30

Parsing of hexadecimal numbers

Hexadecimal numbers are represented as a alphanumerical lexem that:

- either starts with `0x`, `0X`, `-0x` or `-0X`
- the rest of characters are case-insensitive hexadecimal digits

Such a lexeme is called HEX.

The VALUE of HEX lexem is calculated by the following formula: $\sum_{i=1}^{n} c_i 16^{n-i}$, where:

- n - number of digits (after `x` or `X`)
- c_i - the ith digit in the lexeme (after `x` or `X`)

The BIT VALUE of HEX WITH `base_value` is a big-endian representation of the VALUE of HEX multiplied by `-1` in case the lexeme starts with `-`, with the number of bits is :

- if `bits` is divisible by 8 - `bits`[13]
- otherwise - $bits\ + 8\ -\ bits\ mod\ 8$ while all the unused higher bits are zero.

The PARSING OF `hex` WITH `min_value`, `max_value`, `bits` is:

- if `hex` is not HEX then `UnexpectedType ParameterError`
- if VALUE of `hex` is less than `min_value` then `OutOfRange ParameterError`
- if VALUE of `hex` is more than `max_value` then `OutOfRange ParameterError`
- Otherwise, it’s a BIT VALUE OF `hex` WITH `bits`

Kinds of hexadecimal parsing

For the convenience, the following shortcuts are introduced. All the shortcuts have the following meaning: `<SHORTCUT> hex = PARSING OF hex WITH <specific min_value>, <specific max_value>, <specific bits>`. The full list of shortcuts is defined below:

Shortcut	**`min_value`**	**`max_value`**	**`bits`**
`HU--5`	-5	A	4
`HI8`	-80	7F	8
`HI16`	-8000	7FFF	16
`HI30`	-2^{1D}	2^{1D}-1	30

[13] In can be represented either in signed or unsigned form, depending on the context

Parsing of control and stack registries

Generic approach

The alphanumeric lexeme to be parsed must:

- Start with `c`, `C`, `s` or `S`
- The rest of the lexeme must be NUMERIC

Such a lexeme is called REGISTRY, where the first character is called REGISTRY TYPE and the rest of the lexeme - REGISTRY NUMBER.

PARSED REGISTRY `registry` WITH `min_value`, `max_value, bits` means:

- If `registry` is not REGISTRY then `UnexpectedType ParameterError`
- Otherwise, PARSING OF REGISTRY NUMBER WITH `min_value`, `max_value`, `min_value`, `bits`

Parsing of control registries

The alphanumeric lexeme to be parsed must:

- Start with `c`, `C`
- It's a REGISTRY lexeme

CR `registry` means:

- IF `registry` does not start with `c`, `C` then `UnexpectedType ParameterError`
- Otherwise, PARSED REGISTRY `registry` WITH `0, 15, 4`

Parsing of stack registries

The alphanumeric lexeme to be parsed must:

- Start with `s`, `S`
- It's a REGISTRY lexeme

STACK REGISTRY `registry` WITH `min_value`, `max_value, bits` means:

- IF `registry` does not start with `s`, `S` then `UnexpectedType ParameterError`
- Otherwise, PARSED REGISTRY `registry` WITH `min_value`, `max_value, bits`

For the convenience, the following shortcuts are introduced. All the shortcuts have the following meaning: `<SHORTCUT> registry = STACK REGISTRY registry WITH <specific min_value>, <specific max_value>`. The full list of shortcuts is defined below:

Shortcut	min_value	max_value	bits
SR	0	15	4
SR++1	1	15	4
SR++2	2	15	4
SR-1	-1	14	4
SR-2	-2	13	4
SR+	0	255	8

Parsing of slices

Slices are the expected parameters for some primitives. The alphanumeric lexeme that represents them must:

- Start with x or X
- May end with _
- All other characters must be hexadecimal digits (case insensitive)

Such lexeme is called SLICE.

PSLICE `slice , bits` means:

- If `slice` is not SLICE then `UnexpectedType ParameterError`
- Otherwise, the **completed** sequence of bits as described above with a number `bits` assumed to be preceding of `slice`. Thus the total length of PSLICE + number of `bits` is always divisible by 8.

Parsing of `PLDUZ` parameter

The `PLDUZ` primitive has a parameter that must be divisible by 32. So, it should be an alphanumeric lexeme that:

- is NUMERIC
- PARSED OF `lexeme` WITH `32`, `256`, `32`, `8` is divisible by 8

Such a lexeme is called PLDUZ.

PPLDUZ `plduz` means:

- IF PARSED OF `lexeme` WITH `32, 256, 32, 8` is not divisible by 8 then `OutOfRange ParameterError`
- In case of failure - the result of PARSED OF `lexeme` WITH `32, 256, 32, 8`
- In case of success - the result of PARSED OF `lexeme` WITH `32, 256, 32, 8` divided by 32

Parsing of strings

This kind of parsing is used for debug primitives. The lexeme to be parsed is a generic alphanumerical one.

PSTRING `lexeme, min_size, max_size` means:

- If lexeme starts with `x` or `X`, contains at least one more character and all the characters but the first one are hexadecimal digits - the bit sequence that is a big-endian representation of the hexadecimal number of all the characters but the first one
- Otherwise, the natural representation of the `lexeme` as a bit string
- if lexeme has less bytes than `min_size` or more bytes then `max_size` then `OutOfRange ParameterError`

Generic grammar

The high level grammar of the language can be described by the following rules (the `+` character stands for concatenation of lexemes).

GRM.1	FULL PROGRAM is a SET OF PRIMITIVES
GRM.2	SET OF PRIMITIVES is either: ● SET OF PRIMITIVES + PRIMITIVE ● NOTHING
GRM.3	PRIMITIVE is a COMMAND + PARAMETERS
GRM.4	COMMAND is a one of the strings defined below
GRM.5	PARAMETERS is either: ● PARAMETERS + `,` + PARAMETER ● PARAMETER ● nothing

GRM.6	PARAMETER is either: - hexadecimal lexeme - BLOCK
GRM.7	BLOCK is `{` + SET OF PRIMITIVES + `}`

Error handling

In case the grammar rules mentioned above are not met the compilation should fail with the following errors.

GRE.1	In case of an extra comma the compilation should fail with `Syntax CompileError` having the following attributes: - pos - comma position (in the original not NORMALIZED input) - explanation - `,`
GRE.2	In case of a missing comma the compilation should fail with `Syntax CompileError` having the following attributes: - pos - position where the comma was expected (in the original not NORMALIZED input) - explanation - `Missing comma`

Parsing of primitives

Each primitive has:

- A name - case insensitive alphanumeric string (can contain `-` or `.`)
- Strict set of parameters (variative in some cases) specific for this particular primitive (possibly empty)

Each parameter should be parsable into a sequence of bits by one of rules discussed in the Parsing section.

PRE.1	If the command is unknown (is absent in the list below) the compilation should fail with `UnknownOperation CompileError` with the following attributes: - `pos` - position where the command is located (in the original not NORMALIZED input) - `explanation` - the unrecognized command
PRE.2	If the primitive has too few parameters the compilation should fail with

	Operation CompileError with the following attributes: - pos - position where the parameter was expected (in the original not NORMALIZED input) - explanation - the command being parsed - OperationError - MissingRequiredParameters
PRE.3	If the primitive has too many parameters the compilation should fail with Operation CompileError with the following attributes: - pos - position where the extra parameter was found (in the original not NORMALIZED input) - explanation - the command being parsed - OperationError - TooManyParameters
PRE.4	If the primitive failed to be parsed with the required parsing rules (with some exceptions discussed below) the compilation should fail with Operation CompileError with the following attributes: - pos - position where the invalid parameter was found (in the original not NORMALIZED input) - explanation - the command being parsed - OperationError - Parameter with : ○ str - parameter value ○ ParameterError - received ParameterError

Code generation

Generic approach

Each command represents one (in some cases, more, such a case will be discussed below) TVM primitive that can be saved as a sequence of bits. After this, each parsed parameter is saved in the form of a sequence of bits as well as saved next to the previous parameter or to primitive itself.

Some commands stay for a few primitives. In these cases the exact primitive is defined by the set of parameters so they need to be sequentially tried getting a CompileError as discussed here, until the right primitive will be found.

In some cases other bits are inserted in the middle or end of the sequence, such cases are highlighted separately.

For convenience, bitwise concatenation is marked by ░, while different options of primitives - by bullets.

GEE.1	The generated code for each primitive should not be larger than 1023 bits otherwise `Operation CompileError` with the following attributes: ● `pos` - position where the primitive was ended (in the original not NORMALIZED input) ● `explanation` - the command being parsed ● `OperationError` - `NotFitInSlice`
GEE.2	If the command allows a few options for TVM primitives and none of them fits `Operation CompileError` with the following attributes: ● `pos` - position where the command located (in the original not NORMALIZED input) ● `explanation` - the command being parsed ● `OperationError` - last received `OperationError`
GEE.3	If the parameter is a block then it is compiled as an independent program. If this compilation fails the main compilation should fail as well with `Operation CompileError` with the following attributes: ● `pos` - position where the block located (in the original not NORMALIZED input) ● `explanation` - the command being parsed ● `OperationError` - `Nested` with received inner `CompileError` as a parameter
GEE.4	If the parameter is a block and it's not found then `Operation CompileError` with the following attributes: ● `pos` - position where the block should be located (in the original not NORMALIZED input) ● `explanation` - the command being parsed ● `OperationError` - `MissingBlock`
GEN.1	If the parameter is a block then it is compiled as an independent program. If this compilation passes the result should be returned as a cell. The length of such a cell is not counted as a part of primitive data but rather will be used as a cell reference

If the parameter is a block th

Table of commands

Command	Generated code

-ROLL	55 U4+1 0
-ROLLX	62
-ROT	59
.BLOB	PSLICE 0
.CELL	BLOCK
2DROP	5B
2DUP	5C
2OVER	5D
2ROT	5513
2SWAP	5A
ABS	B60B
ACCEPT	F800
ADD	A0
ADDCONST	A6 I8
ADDRAND	F815
AGAIN	EA
AGAINBRK	E31A
AGAINEND	EB
AGAINENDBRK	E31B
AND	B0
ATEXIT	EDF3
ATEXITALT	EDF4
BALANCE	F827
BBITREFS	CF33
BBITS	CF31
BCHECKBITREFS	CF3B

BCHECKBITREFSQ	CF3F
BCHKBITS	● CF38 U8+1 ● CF39
BCHKBITSQ	● CF3C U8+1 ● CF3D
BCHKREFS	CF3A
BCHKREFSQ	CF3E
BDEPTH	CF30
BINDUMP	FE12
BINPRINT	FE13
BITSIZE	B602
BLESS	ED1E
BLESSARGS	EE U4 U4-1
BLESSNUMARGS	EE0 U4-
BLESSVARARGS	ED1F
BLKDROP	5F0 U4
BLKDROP2	6C U4++1-1 U4
BLKPUSH	5F U4++1-1 U4
BLKSWAP	55 U4+1 U4+1
BLKSWX	63
BLOCKLT	F824
BOOLAND	EDF0
BOOLEVAL	EDF9
BOOLOR	EDF1
BRANCH	DB32
BREFS	CF32
BREMBITS	CF35

BREMBITREFS	CF37
BREMREFS	CF36
BUYGAS	F802
CADR	6FB4
CADDR	6FD4
CALL	● F0 U8 ● F1 (00 bits) U14
CALLCC	DB34
CALLCCARGC	DB36 U4 U4-1
CALLCCVARARGS	DB3B
CALLDICT	● F0 U8 ● F1 (00 bits) U14
CALLREF	● DB3C ● DB3C BLOCK
CALLX	D8
CALLXARGS	● DA U4 U4 ● DB0 U4
CALLXVARARGS	DB38
CAR	6F10
CDATASIZE	F941
CDATASIZEQ	F940
CDDR	6FB5
CDDDR	6FD5
CDEPTH	D765
CDR	DF11
CHANGELIB	FB07
CHKBOOL	B400
CHKBIT	B500

CHKDEPTH	69
CHKNAN	C5
CHKSIGNS	F911
CHKSIGNU	F910
CHKTUPLE	6F30
CMP	BF
COMMA	6F8C
COMMIT	F80F
COMPOS	EDF0
COMPOSALT	EDF1
COMPOSBOTH	EDF2
CONDSEL	E304
CONDSELCHK	E305
CONFIGROOT	F829
CONFIGDICT	F830
CONFIGPARAM	F832
CONFIGOPTPARAM	F833
CONS	6F02
CTOS	D0
DEC	A5
DEBUG	FE U8--15
DEBUGOFF	FE1E
DEBUGON	FE1F
DEBUGSTR	FEF (4 bits - length of PSTRING in bytes - 1) PSTRING 1,16
DEPTH	68
DICTADD	F432

DICTADDB	F451
DICTADDGET	F431
DICTADDGETB	F455
DICTADDGETREF	F43B
DICTADDREF	F433
DICTDEL	F459
DICTDELGET	F462
DICTDELGETREF	F463
DICTEMPTY	6E
DICTGET	F40A
DICTGETNEXT	F474
DICTGETNEXTEQ	F475
DICTGETOPTREF	F469
DICTGETPREV	F476
DICTGETPREVEQ	F477
DICTGETREF	F40B
DICTIADD	F434
DICTIADDB	F452
DICTIADDGET	F43C
DICTIADDGETB	F456
DICTIADDGETREF	F43D
DICTIADDREF	F435
DICTIDEL	F45A
DICTIDELGET	F464
DICTIDELGETREF	F465
DICTIGET	F40C

DICTIGETEXEC	F4A2
DICTIGETEXECZ	F4BE
DICTIGETJMP	F4A0
DICTIGETJMPZ	F4BC
DICTIGETNEXT	F478
DICTIGETNEXTEQ	F479
DICTIGETOPTREF	F46A
DICTIGETPREV	F47A
DICTIGETPREVEQ	F47B
DICTIGETREF	F40D
DICTIMAX	F48C
DICTIMAXREF	F48D
DICTIMIN	F484
DICTIMINREF	F485
DICTIREMMAX	F49C
DICTIREMMAXREF	F49D
DICTIREMMIN	F494
DICTIREMMINREF	F495
DICTIREPLACE	F424
DICTIREPLACEB	F44A
DICTIREPLACEGET	F42C
DICTIREPLACEGETB	F44E
DICTIREPLACEGETREF	F42D
DICTITREPLACEREF	F425
DICTISET	F414
DICTISETB	F442

DICTISETGET	F41C
DICTISETGETB	F446
DICTISETGETOPTREF	F46E
DICTISETGETREF	F41D
DICTISETREF	F415
DICTMAX	F48A
DICTMAXREF	F48B
DICTMIN	F482
DICTMINREF	F483
DICTPUSHCONST	F4A4 (6 zero bits) U10
DICTREMMAX	F49A
DICTREMMAXREF	F49B
DICTREMMIN	F492
DICTREMMINREF	F493
DICTREPLACE	F422
DICTREPLACEB	F449
DICTREPLACEGET	F42A
DICTREPLACEGETB	F44D
DICTREPLACEGETREF	F42B
DICTREPLACEREF	F423
DICTSET	F412
DICTSETB	F441
DICTSETGET	F41A
DICTSETGETB	F445
DICTSETGETOPTREF	F46D
DICTSETGETREF	F41B

DICTSETREF	F413
DICTUADD	F436
DICTUADDB	F453
DICTUADDGET	F43E
DUCTUADDGETB	F457
DICTUADDGETREF	F43F
DICTUADDREF	F437
DICTUDEL	F45B
DICTUDELGET	F466
DICTUDELGETREF	F467
DICTUGET	F40E
DICTUGETEXEC	F4A3
DICTUGETEXECZ	F4BF
DICTUGETJMP	F4A1
DICTUGETJMPZ	F4BD
DICTUGETNEXT	F47C
DICTUGETNEXTEQ	F47D
DICTUGETOPTREF	F46B
DICTUGETPREV	F47E
DICTUGETPREVEQ	F47F
DICTUGETREF	F40F
DICTUMAX	F48E
DICTUMAXREF	F48F
DICTUMIN	F486
DICTUMINREF	F487
DICTUREMMAX	F49E

DICTUREMMAXREF	F49F
DICTUREMMIN	F496
DICTUREMMINREF	F497
DICTUREPLACE	F426
DICTUREPLACEB	F44B
DICTUREPLACEGET	F42E
DICTUREPLACEGETB	F44F
DICTUREPLACEGETREF	F42F
DICTUREPLACEREF	F427
DICTUSET	F416
DICTUSETB	F443
DICTUSETGET	F41E
DICTUSETGETB	F447
DICTUSETGETOPTREF	F46F
DICTUSETGETREF	F41F
DICTUSETREF	F417
DIV	A904
DIVC	A906
DIVR	A905
DIVMOD	A90C
DIVMODC	A90E
DIVMODR	A90D
DROP	30
DROPX	65
DROP2	5B
DUMP	FE2 U4-

DUMPSTK	FE00
DUMPSTKTOP	FE0 U4++1-2
DUMPTOSFMT	FEF (4 bits - length of PSTRING in bytes - 1) PSTRING 1,16
DUP	20
DUP2	5C
ENDC	C9
ENDCST	CD
ENDS	D1
ENDXC	CF23
EQUAL	BA
EQINT	C0 I8
EXECUTE	D8
EXPLODE	6F4 U4
EXPLODEVAR	6F84
FALSE	70
FIRST	6F10
FITS	B4 U4+1
FITSX	B600
GASTOGRAM	F805
GEQ	BE
GETGLOBVAR	F840
GETGLOB	F84 (010 bits) U5
GETPARAM	F82 U4
GRAMTOGAS	F804
GREATER	BC
GTINT	C2 I8

HASHCU	F900
HASHSU	F901
HEXDUMP	FE10
HEXPRINT	FE11
IF	DE
IFBITJMP	E3 (100 bits) U5
IFBITJMPREF	E3 (110 bits) U5
IFELSE	E2
IFELSEREF	● E30E ● E30EBLOCK
IFJMP	E0
IFJMPREF	● E302 ● E302 BLOCK
IFNBITJMP	E3 (101 bits) U5
IFNBITJMPRED	E3 (111 bits) U5
IFNOT	DF
IFNOTJMP	E1
IFNOTJMPREF	● E303 ● E303 BLOCK
IFNOTREF	● E301 ● E301 BLOCK
IFNOTRET	DD
IFNOTRETALT	E309
IFRET	DC
IFRETALT	E308
IFREF	● E300 ● E300 BLOCK
IFREFELSE	● E30D ● E30D BLOCK

IFREFELSEREF	E30F
INC	A4
INTSORT2	B60A
INVERT	EDF8
INDEX	6F1 U4
INDEXQ	6F6 U4
INDEXVAR	6F81
INDEXVARQ	6F86
INDEX2	6FB U2 U2
INDEX3	6F (11 bits) U2 U2 U2
INITCODEHASH	F82B
ISNAN	C4
ISNEG	C100
ISNNEG	C2FF
ISNPOS	C101
ISNULL	6E
ISPOS	C200
ISTUPLE	6F8A
ISZERO	C000
JMP	F1 (01 bits) U14
JMPDICT	F1 (01 bits) U14
JMPREF	● D83D ● D83D BLOCK
JMPX	D9
JMPXARGS	D81 U4
JMPXDATA	DB3E
LAST	6F8B

LDCONT	D766
LDI	D2 U8+1
LDDICT	F404
LDDICTS	F402
LDDICTQ	F406
LDGRAMS	FA00
LDILE4	D750
LDILE4Q	D758
LDILE8	D752
LDILE8Q	D75A
LDIQ	D70C U8+1
LDIX	D700
LDXQ	D704
LDMSGADDR	FA40
LFMSGADDRQ	FA41
LDONES	D761
LDOPTREF	F404
LDREF	D4
LDREFRTOS	D5
LDSAME	D762
LDSLICE	D6 U8+1
LDSLICEQ	D71E U8+1
LDSLICEX	D718
LDSLICEXQ	D71A
LDU	D3 U8+1
LDULE4	D751

LDULE4Q	D759
LDULE8	D753
LDULE8Q	D75B
LDUQ	D70D U8+1
LDUX	D701
LDUXQ	D705
LDVARINT16	FA01
LDVARINT32	FA05
LDVARUINT16	FA00
LDVARUINT32	FA04
LDZEROES	D760
LEQ	BB
LESS	B9
LESSINT	C1 I8
LOGFLUSH	FEF000
LOGSTR	FEF (4 bits - length of PSTRING in bytes) 00 PSTRING 0,15
LSHIFT	● AA U8+1 ● AC
LSHIFTDIV	● A9C4 ● A9D4 U8+1
LSHIFTDIVC	● A9C7 ● A9D7 U8+1
LSHIFTDIVMOD	● A9CC ● A9DC U8+1
LSHIFTDIVMODC	● A9CE ● A9DE U8+1
LSHIFTDIVMODR	● A9CD ● A9DD U8+1
LSHIFTDIVR	● A9C5 ● A9D5 U8+1

LSHIFTMOD	● A9C8 ● A9D8 U8+1
LSHIFTMODC	● A9CA ● A9DA U8+1
LSHIFTMODR	● A9C9 ● A9D9 U8+1
LTIME	F825
MAX	B609
MIN	B608
MINMAX	B60A
MOD	A908
MODC	A90A
MODPOW2	● A928 ● A938 U8+1
MODPOW2C	● A92A ● A93A U8+1
MODPOW2R	● A929 ● A939 U8+1
MODR	A909
MUL	A8
MULCONST	A7 I8
MULDIV	A984
MULDIVC	A986
MULDIVR	A985
MULDIVMOD	A98C
MULDIVMODC	A98E
MULDIVMODR	A98D
MULMOD	A988
MULMODC	A98A

MULMODPOW2	● A9A8 ● A9B8 U8+1
MULMODPOW2C	● A9AA ● A9BA U8+1
MULMODPOW2R	● A9A9 ● A9B9 U8+1
MULMODR	A989
MULRSHIFT	● A9A4 ● A9B4 U8+1
MULRSHIFTC	● A9A6 ● A9B6 U8+1
MULSHIFTMOD	● A9AC ● A9BC U8+1
MULSHIFTMODC	● A9AE ● A9BE U8+1
MULSHIFTMODR	● A9AD ● A9BD U8+1
MULRSHIFTR	● A9A5 ● A9B5 U8+1
MYADDR	F828
MYCODE	F82A
NEGATE	A3
NEQ	BD
NEQINT	C3 I8
NEWC	C8
NEWDICT	6D
NIL	6F00
NIP	31
NOP	00
NOT	B3

NOW	F823
NULL	6D
NULLROTRIF	6FA2
NULLROTRIF2	6FA6
NULLROTRIFNOT	6FA3
NULLROTRIFNOT2	6FA7
NULLSWAPIF	6FA0
NULLSWAPIF2	6FA4
NULLSWAPIFNOT	6FA1
NULLSWAPIFNOT2	6FA5
ONE	71
OR	B1
OVER	21
OVER2	5D
ONLYTOPX	6A
ONLYX	6B
PAIR	6F02
PARSEMSGADDR	FA42
PARSEMSGADDRQ	FA43
PFXDICTADD	F472
PFXDICTCONSTGETJMP	F4A (11 bits) U10
PFXDICTDEL	F473
PFXDICTGET	F4A9
PFXDICTGETEXEC	F4AB
PFXDICTGETJMP	F4AA
PFXDICTGETQ	F4A8

PFXDICTREPLACE	F471
PFXDICTSET	F470
PFXDICTSWITCH	F4A (11 bits) U10
PLDDICT	F405
PLDDICTS	F403
PLDDICTQ	F407
PLDI	D70A U8+1
PLDILE4	D754
PLDILE4Q	D75C
PLDILE8	D756
PLDILE8Q	D75E
PLDIQ	D70E U8+1
PLDIX	D702
PLDIXQ	D706
PLDSLICE	D71D U8+1
PLDSLICEQ	D71F U8+1
PLDOPTREF	F405
PLDREF	D74C
PLDREFIDX	D74 (11 bits) U2
PLDREFVAR	F748
PLDSLICEX	D719
PLDSLICEXQ	D71B
PLDU	D70B U8+1
PLDULE4	D755
PLDULE4Q	D75D
PLDULE8	D757

PLDULE8Q	D75F
PLDUQ	D70F U8+1
PLDUX	D703
PLDUXQ	D707
PLDUZ	D71 (0 bit) PPLDUZ
PICK	60
PUSHX	60
POP	● 3 SR ● 57 SR+ ● ED5 CR
POPCTR	ED5 CR
POPCTRSAVE	ED9 CR
POPCTRX	EDE1
POPROOT	ED54
POPSAVE	ED9 CR
POW2	AE
PREPARE	F1 (10 bits) U14
PREPAREDICT	F1 (10 bits) U14
PRINT	FE3 U4-
PRINTSTR	FEF (4 bits - length of PSTRING in bytes) PSTRING 0,15
PU2XC	546 U4 U4+1 U4+2
PUSH	● 2 SR ● 56 SR+ ● ED4 CR
PUSH2	53 U4 U4
PUSH3	547 U4 U4 U4
PUSHCONT	if BLOCK has 0 references and less than 16 bytes (rounded up) then: ● 9 (4 bits - length of BLOCK in bytes) (data of BLOCK with zeroes in the end to make the bit length divisible

	by 8) else: • 8 (111 bits) (2 bits - number of references in BLOCK) (7 bits -) length of BLOCK in bytes) (data of BLOCK (or first 127 bytes if it's larger than 128 bytes) with zeroes in the end to make the bit length divisible by 8)
PUSHCTR	ED4 CR
PUSHCTRX	EDE0
PUSHINT	• 7 U--5 • 7 HU--5 • 80 I8 • 80 HI8 • 81 I16 • 81 HI16 • 82 (3 bits - number of valued bits in I30) (valued bits of I30) • 82 (3 bits - number of valued bits in HI30) (valued bits of HI30)
PUSHNAN	83FF
PUSHNEGPOW2	85 U8+1
PUSHNULL	6D
PUSHPOW2	83 U8+1
PUSHPOW2DEC	84 U8+1
PUSHREF	• 88 • 88 BLOCK
PUSHREFCONT	• 8A • 8A BLOCK
PUSHREFSLICE	• 89 • 89 BLOCK
PUSHROOT	ED44
PUSHSLICE[14]	• 8B (four bits as length of PSLICE 4 - 4 / 8) PSLICE 4 • 8D (000 bits) (seven bits as length of PSLICE 10 - 6 / 8) PSLICE 10

[14] Looks like variants of PUSHSLICE with references are not supported. Please see the corresponding bug. These variants don't present in the current specification as well

PUXC	52 SR SR-1
PUXC2	544 SR SR-1 SR-1
PUXCPU	545 SR SR-1 SR-1
QABS	B7B60B
QADD	B7A0
QADDCONST	87A6 I8
QAND	B7B0
QBITSIZE	B7B602
QCMP	B7BF
QDEC	B7A5
QDIV	B7A904
QDIVC	B7A906
QDIVR	B70905
QDIVMOD	B7090C
QDIVMODC	B7A90E
QDIVMODR	B7A90D
QEQINT	B7C0 I8
QEQUAL	B7BA
QFITS	B7B4 U8+1
QFITSX	B7B600
QGEQ	B7BE
QGREATER	B7BC
OGTINT	B7 C2 I8
QINC	B7A4
QINTSORT2	B7B60A
QMAX	B7B609

QMIN	B7B608
QMINMAX	B7B60A
QMOD	B7A908
QMODC	B7A90A
QMODR	B7A909
QMUL	B7A8
QMULCONST	B7A7 I8
QMULDIV	B7A984
QMULDIVC	B7A986
QMULDIVR	B7A985
QMULDIVMOD	B7A98C
QMULDIVMODC	B7A98E
QMULDIVMODR	B7A98D
QMULMOD	B7A988
QMULMODC	B7A98A
QMULMODR	B7A989
QLESS	B7B9
QLESSINT	B7C1 I8
QLEQ	B7BB
QLSHIFT	● B7AA U8+1 ● B7AC
QLSHIFTDIV	● B7A9C4 ● B7A9D4 U8+1
QLSHIFTDIVC	● B7A9C7 ● B7A9D7 U8+1
QLSHIFTDIVMOD	● B7A9CC ● B7A9DC U8+1
QLSHIFTDIVMODC	● B7A9CE ● B7A9DE U8+1

QLSHIFTDIVMODR	● B7A9CD ● B7A9DD U8+1
QLSHIFTDIVR	● B7A9C5 ● B7A9D5 U8+1
QLSHIFTMOD	● B7A9C8 ● B7A9D8 U8+1
QLSHIFTMODC	● B7A9CA ● B7A9DA U8+1
QLSHIFTMODR	● B7A9C9 ● B7A9D9 U8+1
QMULMODPOW2	● B7A9A8 ● B7A9B8 U8+1
QMULMODPOW2C	● B7A9AA ● B7A9BA U8+1
QMULMODPOW2R	● B7A9A9 ● B7A9B9 U8+1
QMULRSHIFT	● B7A9A4 ● B7A9B4 U8+1
QMULRSHIFTC	● B7A9A6 ● B7A9B6 U8+1
QMULSHIFTMOD	● B7A9AC ● B7A9BC U8+1
QMULSHIFTMODC	● B7A9AE ● B7A9BE U8+1
QMULSHIFTMODR	● B7A9AD ● B7A9BD U8+1
QMULRSHIFTR	● B7A9A5 ● B7A9B5 U8+1
QMODPOW2	● B7A928 ● B7A938 U8+1
QMODPOW2C	● B7A92A ● B7A93A U8+1
QMODPOW2R	● B7A929 ● B7A939 U8+1

QNEGATE	B7A3
QNEQ	B7BD
QNEQINT	B7C3 I8
QNOT	B7B3
QOR	B7B1
QPOW2	B7AE
QRSHIFT	● B7AD ● B7AB U8+1
QRSHIFTC	● B7A926 ● B7A936 U8+1
QRSHIFTR	● B7A925 ● B7A935 U8+1
QRSHIFTMOD	● B7A92C ● B7A93C U8+1
QRSHIFTMODC	● B7A92D ● B7A93D U8+1
QRSHIFTMODR	● B7A92E ● B7A93E U8+1
QSGN	B7B8
QSUB	B7A1
QSUBR	B7A2
QTLEN	6F89
QUBITSIZE	B7B603
QUFITS	B7B5 U8+1
QUFITSX	B7B601
QXOR	B7B2
RAND	F811
RANDSEED	F826
RANDU256	F810

RAWRESERVE	FB02
RAWRESERVEX	FB03
REPEAT	E4
REPEATBRK	E314
REPEATEND	E5
REPEATENDBRK	E315
RET	DB30
RETALT	DB31
RETARGS	DB2 U4
RETBOOL	DB32
RETDATA	DB3F
RETFALSE	DB31
RETTRUE	DB30
RETURNARGS	ED0 U4
RETURNVARARGS	ED10
RETVARARGS	DB39
REVERSE	5E U4+2 U4
REVX	64
REWRITESTDADDR	FA44
REWRITESTDADDRQ	FA45
REWRITEVARADDR	FA46
REWRITEVARADDRQ	FA47
ROT	58
ROT2	5513
ROTREV	59
ROLL	550 U4+1

ROLLREV	55 U4+1 0
ROLLX	61
ROLLREVX	62
RSHIFT	● AD ● AB U8+1
RSHIFTC	● A926 ● A936 U8+1
RSHIFTR	● A925 ● A935 U8+1
RSHIFTMOD	● A92C ● A93C U8+1
RSHIFTMODC	● A92D ● A93D U8+1
RSHIFTMODR	● A92E ● A93E U8+1
SAMEALT	EDFA
SAMEALTSAV	EDFB
SAVE	EDA CR
SAVEALT	EDB CR
SAVEALTCTR	EDB CR
SAVEBOTH	EDC CR
SAVEBOTHCTR	EDC CR
SAVECTR	EDA CR
SBITS	D749
SBITREFS	D74B
SCHKBITS	D741
SCHKBITREFS	D743
SCHKBITREFSQ	D747
SCHKBITSQ	D745

SCHKREFS	D742
SCHKREFSQ	D746
SCUTFIRST	D730
SCUTLAST	D732
SDATASIZE	F943
SDATASIZEQ	F942
SDBEGINS	● if the parameter is ‘0’ then: ○ D72802 ● if the parameter is ‘1’ then: ○ D72806 ● else: ○ D7 (001010 bits) (seven bits of length of PSLICE 13 - 3 / 8) PSLICE 13
SDBEGINSQ	D7 (001011 bits) (seven bits of length of PSLICE 13 - 3 / 8) PSLICE 13
SDBEGINSX	D726
SDBEGINSXQ	D727
SDCNTLEAD0	C710
SDCNTLEAD1	C711
SDCNTTRAIL0	C712
SDCNTTRAIL1	C713
SDCUTFIRST	D720
SDCUTLAST	D722
SDEMPTY	C701
SDEPTH	D764
SDEQ	C705
SDFIRST	C703
SDPFX	C708
SDPFXREV	C709
SDPPFX	C70A

SDPPFXREV	C70B
SDPSFX	C70E
SDPSFXREV	C70F
SDSFX	C70C
SDSFXREV	C70D
SDLEXCMP	C704
SDSKIPFIRST	D721
SDSKIPLAST	D723
SDSUBSTR	D724
SECOND	6F11
SEMPTY	C700
SENDRAWMSG	FB00
SETALTCTR	ED8 CR
SETCODE	FB04
SETCONT	ED6 CR
SETCONTARGS	● EC U4 F ● EC U4 U4--1
SETCONTCTR	ED6 CR
SETCONTCTRX	EDE2
SETCONTVARARGS	ED11
SETCP	FF U8--15
SETCP0	FF00
SETCPX	FFF0
SETEXITALT	EDF5
SETGASLIMIT	F801
SETGLOBVAR	F860
SETGLOB	F8 (011 bits) U5

SETFIRST	6F50
SETINDEX	6F5 U4
SETINDEXQ	6F7 U4
SETINDEXVAR	6F85
SETINDEXVARQ	6F87
SETLIBCODE	FB06
SETNUMARGS	EC0 U4--1
SETNUMVARARGS	ED12
SETRAND	F814
SETRETCTR	ED7 CR
SETSECOND	6F51
SETTHIRD	6F52
SGN	B8
SHA256U	F902
SINGLE	6F01
SKIPDICT	F401
SKIPOPTREF	F401
SPLIT	D736
SPLITQ	D737
SREFS	D74A
SREMPTY	C702
SSKIPFIRST	D731
SSKIPLAST	D733
STB	CF13
STBQ	CF1B
STBR	CF17

STBREF	CF11
STBREFQ	CF19
STBREFR	CD
STBREFRQ	CF1D
STBRQ	CF1F
STGRAMS	FA02
STDICT	F400
STDICTS	CE
STI	CA U8+1
STILE4	CF28
STILE8	CF2A
STIQ	CF0C U8+1
STIR	CF0A U8+1
STIRQ	CF0E U8+1
STIX	CF00
STIXQ	CF04
STIXR	CF02
STIXRQ	CF06
STONE	CF83
STONES	CF41
STOPTREF	F400
STRDUMP	FE14
STRPRINT	FE15
STREF	CC
STREF2CONST	CF21
STREF3CONST	CFE2

STREFCONST	CF20
STREFQ	CF18
STREFR	CF14
STREFRQ	CF1C
STSAME	CF42
STSLICE	CE
STSLICECONST	● if the parameter is ‘0’ then: ○ CF81 ● if the parameter is ‘1’ then: ○ CF83 ● else ○ CF (100 bits) (three bits of length of PSLICE 6 - 2 / 8) PSLICE 6
STSLICEQ	CF1A
STSLICER	CF16
STSLICERQ	CF1E
STU	CB U8+1
STULE4	CF29
STULE8	CF2B
STUQ	CF0D U8+1
STUR	CF0B U8+1
STURQ	CF0F U8+1
STUX	CF01
STUXQ	CF05
STUXR	CF03
STUXRQ	CF07
STVARINT16	FA03
STVARINT32	FA07
STVARUINT16	FA02

STVARUINT32	FA06
STZERO	CF81
STZEROES	CF40
STCONT	CF43
SUB	A1
SUBDICTGET	F4B1
SUBDICTIGET	F4B2
SUBDICTIRPGET	F4B6
SUBDICTRPGET	F4B5
SUBDICTUGET	F4B3
SUBDICTURPGET	F4B7
SUBR	A2
SUBSLICE	D734
SWAP	01
SWAP2	5A
TEN	7A
THENRET	EDF6
THENRETALT	EDF7
THIRD	6F12
THROW	● F2 (01 bits) U6 ● F2C (1 bits) U11
THROWIF	● F2 (10 bits) U6 ● F2D (1 bits) U11
THROWIFNOT	● F2 (11 bits) U6 ● F2E (1 bits) U11
THROWANY	F2F0
THROWANYIF	F2F2
THROWANYIFNOT	F2F4

THROWARG	F2C (1 bit) U11
THROWARGANY	F2F1
THROWARGANYIF	F2F3
THROWARGANYIFNOT	F2F5
THROWARGIF	F2D (1 bit) U11
THROWARGIFNOT	F2E (1 bit) U11
TLEN	6F88
TPOP	6F8D
TPUSH	6F8C
TRIPLE	6F03
TRUE	7F
TRY	F2FF
TRYARGS	F3 U4 U4
TUCK	66
TUPLE	6F0 U4
TUPLEVAR	6F80
TWO	72
UBITSIZE	B603
UFITS	B5 U8+1
UFITSX	B601
UNCONS	6F22
UNPACKFIRST	6F3 U4
UNPACKFIRSTVAR	6F83
UNPAIR	6F22
UNSINGLE	6F21
UNTIL	E6

UNTILBRK	E316
UNTILEND	E7
UNTILENDBRK	E317
UNTRIPLE	6F23
UNTUPLE	6F2 U4
UNTUPLEVAR	6F82
WHILE	E8
WHILEBRK	E318
WHILEEND	E9
WHILEENDBRK	E319
XC2PU	541 SR SR SR
XCHG	● 01 ● 0 SR++1 ● if the first parameter is `s0` or `S0` then (considering the second parameter only): ○ 0 SR++1 ● if the first parameter is `s1` or `S1` then (considering the second parameter only): ○ 1 SR++2 ● 10 SR++1 SR++2 ● 11 SR+
XCHG2	50 SR SR
XCHG3	4 SR SR SR
XCHGX	67
XCPU	51 SR SR
XCPU2	543 SR SR SR
XCPUXC	542 SR SR SR-1
XCTOS	D739
XLOAD	D73A
XLOADQ	D73B

XOR	B2
ZERO	70
ZEROROTRIF	6F92
ZEROROTRIF2	6F96
ZEROROTRIFNOT	6F93
ZEROROTRIFNOT2	6F97
ZEROSWAPIF	6F90
ZEROSWAPIF2	6F94
ZEROSWAPIFNOT	6F91
ZEROSWAPIFNOT2	6F95

Specific bugs

GEE.4	If the command is `PUSHCONT` the BLOCK should have less than four references, otherwise `Operation CompileError` with the following attributes: ● `pos` - position where the primitive was ended (in the original not NORMALIZED input) ● `explanation` - the command being parsed ● `OperationError` - `NotFitInSlice`
GEE.5	if the parsed value for PUSHINT is less than -2^{29} or greater than 2^{29}-1 then `Operation CompileError` with the following attributes: ● `pos` - position where the primitive was ended (in the original not NORMALIZED input) ● `explanation` - the command being parsed ● `OperationError` - `ParameterError` with: ○ string - `arg0` ○ reason - `OutOfRange`
GEE.6	If XCHG has two parameters and the second parameter represents the index higher or equal than the first parameter, then: ● `pos` - position where the primitive was ended (in the original not NORMALIZED input)

	● `explanation` - the command being parsed ● `OperationError` - `LogicErrorInParameters` with: ○ string - `arg 1 should be greater than arg 0`

Code finalization

In case all the previous steps were successful the code (as a set of primitives in a binary form according to rules defined in the previous chapter) is finalized into a chain of cells keeping the following rules:

GEN.2	For the first primitive the new cell is created (root cell). This cell becomes a current cell
GEN.3	If possible, each next primitive is added to the current cell. It should fit into data (1023 bits) and leave at least one reference
GEN.4	Otherwise, the next primitive is added to a new, empty cell that becomes a new current cell. The previous current cell sets its last unused reference to a new current one
GEN.5	The root cell is returned as a result of the compilation

Debug information

Debug information is generated in case of successful compilation using `compile_code_debuggable` in the form of a DbgInfo entity.

DBG.1	The resulting `DbgInfo` is a map where: ● keys - hashes of the resulting cells ● values - maps where: ○ keys - offsets of the compiled commands (bit number of the command's start in the cell's data) ○ values - original `DbgPos` of the uncompiled command in the input `Lines` entity
DBG.2	Each command has to have corresponding record in the resulting `DbgInfo`

Possible bugs

During the creation of the present document the following bug candidates were found. The developers were informed but discussion still did not happen. The list is below:

№	Severity	Description
BUG.1	MAJOR	In case the last line has a comment not separated with the last lexeme by a whitespace, the last lexeme is ignored
BUG.2	MAJOR	Both `\r` and `\n` characters are considered as line terminators that can lead to doubling line numbers if used under Windows
BUG.3	MAJOR	If the final block is not closed by `}` no error is generated
BUG.4	MAJOR	Incorrect code for `LSHIFTDIVR` primitive. The mask should be `11010101`
BUG.5	MAJOR	Variants of `PUSHSLICE` and `STSLICECONST` with references are not supported.