Fift：簡介

Nikolai Durov
譯者：Dr. Awesome Doge

November 1, 2025

Abstract

本文旨在簡要介紹 Fift，這是一種專門為建立和管理 TON 區塊鏈智慧合約而設計的新程式語言，以及其用於與 TON 虛擬機 [4] 和 TON 區塊鏈 [5] 互動的功能。

引言

本文件簡要介紹 Fift，這是一種基於堆疊的通用程式語言，經過最佳化以建立、除錯和管理 TON 區塊鏈智慧合約。

Fift 專門設計用於與 TON 虛擬機（TON VM 或 TVM）[4] 和 TON 區塊鏈 [5] 互動。特別是，它原生支援與 TVM 共享的 257 位元整數運算和 TVM 單元操作，以及 TON 區塊鏈採用的基於 Ed25519 的密碼學介面。Fift 發行版中還包含用於編寫新智慧合約的 TVM 程式碼巨集組譯器。

作為一種基於堆疊的語言，Fift 與 Forth 類似。由於本文的簡潔性，對 Forth 的一些了解可能有助於理解 Fift。[1]然而，這兩種語言之間存在顯著差異。例如，Fift 強制執行執行時期型別檢查，並在其堆疊中保留不同型別的值（不僅僅是整數）。

Fift 中定義的詞彙（內建函數或原語）列表及其簡要說明，呈現在附錄 A 中。

請注意，本文件的當前版本描述了 Fift 的初步測試版本；某些細節可能會在將來發生變化。

[1]有很好的 Forth 入門書籍；我們推薦 [1]。

Contents

1 概述

Fift 是一種簡單的基於堆疊的程式語言，設計用於測試和除錯 TON 虛擬機 [4] 和 TON 區塊鏈 [5]，但也可能適用於其他用途。當 Fift 被呼叫時（通常透過執行名為 `fift` 的二進位檔案），它要麼讀取、解析並解釋命令列中指示的一個或多個來源檔案，要麼進入互動模式並解釋從標準輸入讀取和解析的 Fift 命令。還有一種「腳本模式」，透過命令列開關 `-s` 啟動，在此模式下，除第一個參數外的所有命令列參數透過變數 $\$n$ 和 `$#` 傳遞給 Fift 程式。這樣，Fift 既可用於互動式實驗和除錯，也可用於編寫簡單的腳本。

Fift 操作的所有資料都保存在一個（後進先出）堆疊中。每個堆疊條目都附加了一個型別標籤，它明確地確定儲存在相應堆疊條目中的值的型別。Fift 支援的值型別包括 *Integer*（表示帶符號的 257 位元整數）、*Cell*（表示 TVM 單元，由最多 1023 個資料位元和最多四個對其他單元的參考組成，如 [4] 中所述）、*Slice*（用於解析單元的 *Cell* 部分視圖）和 *Builder*（用於建構新單元）。這些資料型別（及其實作）與 TVM [4] 共享，並且可以在必要時安全地從 Fift 堆疊傳遞到 TVM 堆疊，反之亦然（例如，當使用 Fift 原語（如 `runvmcode`）從 Fift 呼叫 TVM 時）。

除了與 TVM 共享的資料型別外，Fift 還引入了一些獨特的資料型別，例如 *Bytes*（任意位元組序列）、*String*（UTF-8 字串）、*WordList* 和 *WordDef*（由 Fift 用於建立新的「詞彙」並操作其定義）。事實上，Fift 可以擴展以操作任意「物件」（由通用型別 *Object* 表示），前提是它們在當前實作中派生自 C++ 類別 `td::CntObject`。

Fift 來源檔案和函式庫通常保存在後綴為`.fif` 的文字檔案中。函式庫和包含檔案的搜尋路徑透過 `-I` 命令列參數或 `FIFTPATH` 環境變數傳遞給 Fift 執行檔。如果兩者都未設定，則使用預設函式庫搜尋路徑 `/usr/lib/fift`。

在啟動時，標準 Fift 函式庫從檔案 `Fift.fif` 讀取，然後才解釋任何其他來源。它必須存在於函式庫搜尋路徑中，否則 Fift 執行將失敗。

Fift 的基本資料結構是其全域字典，包含詞彙——或者更準確地說，詞彙定義——對應於內建原語和函數以及使用者定義的函數。[2]在 Fift 中，只需在互動模式下輸入詞彙的名稱（不帶空格字元的 UTF-8 字串）即可執行該詞彙。當 Fift 啟動時，一些詞彙（原語）已經被定義（在當前實作中透過一些 C++ 程式碼）；其他詞彙在標準函式庫 `Fift.fif` 中定義。之後，使用者可以透過定義新詞彙或重新定義舊詞彙來擴展字典。

[2]Fift 詞彙通常比其他程式語言的函數或子程式更短。可以在 [2] 中找到不錯的討論和一些指南（針對 Forth 詞彙）。

字典應該被分割成多個詞彙表或命名空間；然而，命名空間尚未實作，因此所有詞彙目前都在同一個全域命名空間中定義。

Fift 輸入來源檔案和標準輸入（在互動模式下）的解析器相當簡單：輸入逐行讀取，然後跳過空白字元，並偵測並從輸入行中移除剩餘行的最長前綴（即字典詞彙的名稱）。[3]之後，執行找到的詞彙，並重複此過程直到行尾。當輸入行耗盡時，從當前輸入檔案或標準輸入讀取後續行。

為了被偵測，大多數詞彙需要在它們之後立即有一個空白字元或行尾；這透過在字典中的詞彙名稱後附加空格來反映。其他詞彙，稱為前綴詞彙，不需要在它們之後立即有空白字元。

如果沒有找到詞彙，則將輸入行的第一個剩餘字元直到下一個空白或行尾字元組成的字串解釋為 *Integer* 並推入堆疊。例如，如果我們呼叫 Fift，輸入 `2 3 + .`（並按 Enter），Fift 首先將等於 2 的 *Integer* 常數推入其堆疊，然後是等於 3 的另一個整數常數。之後，內建原語「`+`」被解析並在字典中找到；當呼叫時，它從堆疊中取出兩個最頂層的元素並用它們的和（在我們的例子中為 5）替換它們。最後，「`.`」是一個原語，它列印堆疊頂部 *Integer* 的十進位表示，後跟一個空格。結果，我們觀察到 Fift 解釋器在標準輸出中列印「`5 ok`」。字串「`ok`」由解釋器在互動模式下完成解釋從標準輸入讀取的行時列印。

內建詞彙列表可以在附錄 A 中找到。

2 Fift 基礎

本章介紹 Fift 程式語言的基本功能。討論最初是非正式和不完整的，但逐漸變得更正式和更精確。在某些情況下，後面的章節和附錄 A 提供了關於本章首次提及的詞彙的更多細節；類似地，一些將在後面章節中詳細解釋的技巧已經在此處適當使用。

2.1 Fift 堆疊值型別列表

目前，以下資料型別的值可以保存在 Fift 堆疊中：

- *Integer* — 帶符號的 257 位元整數。在堆疊記號中通常用 x、y 或 z 表示（當描述 Fift 詞彙的堆疊效果時）。
- *Cell* — TVM 單元，由最多 1023 個資料位元和最多 4 個對其他單元的參考組成（參見 [4]）。通常用 c 或其變體表示，例如 c' 或 c_2。

[3]請注意，與 Forth 不同，Fift 詞彙名稱區分大小寫：`dup` 和 `DUP` 是不同的詞彙。

- *Slice* — TVM 單元的部分視圖，用於從 *Cell* 中解析資料。通常用 s 表示。
- *Builder* — 部分建構的 *Cell*，包含最多 1023 個資料位元和最多四個參考；可用於建立新的 *Cell*。通常用 b 表示。
- *Null* — 具有唯一「空」值的型別。用於初始化新的 *Box*。通常用 $\perp$ 表示。
- *Tuple* — 這些型別中任何型別的值的有序集合（不一定相同）；可用於表示任意代數資料型別和 Lisp 風格列表的值。
- *String* — （通常是可列印的）UTF-8 字串。通常用 S 表示。
- *Bytes* — 任意 8 位元位元組序列，通常用於表示二進位資料。通常用 B 表示。
- *WordList* — （部分建立的）詞彙參考列表，用於建立新的 Fift 詞彙定義。通常用 l 表示。
- *WordDef* — 執行權杖，通常表示現有 Fift 詞彙的定義。通常用 e 表示。
- *Box* — 記憶體中可用於儲存一個堆疊值的位置。通常用 p 表示。
- *Atom* — 由其名稱（字串）唯一標識的簡單實體。可用於表示識別符號、標籤、操作名稱、標記和堆疊標記。通常用 a 表示。
- *Object* — 派生自基底類別 `td::CntObject` 的任何類別的任意 C++ 物件；可由 Fift 擴充功能用於操作其他資料型別並與其他 C++ 函式庫介面。

上面列出的前六種型別與 TVM 共享；其餘的是 Fift 特定的。請注意，並非所有 TVM 堆疊型別都出現在 Fift 中。例如，TVM *Continuation* 型別未被 Fift 明確識別；如果此型別的值最終出現在 Fift 堆疊中，它將作為通用 *Object* 進行操作。

2.2 註解

Fift 識別兩種註解：「`//` 」（後面必須跟一個空格）開啟單行註解直到行尾，`/*` 定義多行註解直到 `*/`。詞彙 `//` 和 `/*` 都在標準 Fift 函式庫（`Fift.fif`）中定義。

2.3 終止 Fift

詞彙 `bye` 以零退出碼終止 Fift 解釋器。如果需要非零退出碼（例如，在 Fift 腳本中），可以使用詞彙 `halt`，它以給定的退出碼（作為堆疊頂部的 *Integer* 傳遞）終止 Fift。相反，`quit` 不退出到作業系統，而是退出到 Fift 解釋器的頂層。

2.4 簡單整數運算

當 Fift 遇到字典中不存在但可以解釋為整數常數（或「字面值」）的詞彙時，其值被推入堆疊（如 **2.8** 中更詳細地解釋的那樣）。除此之外，定義了幾個整數算術原語：

- `+`（$x\ y - x+y$），用堆疊頂部傳遞的兩個 *Integer* x 和 y 的和 $x+y$ 替換它們。所有更深的堆疊元素保持不變。如果 x 或 y 不是 *Integer*，或者如果和不適合帶符號的 257 位元 *Integer*，則拋出異常。
- `-`（$x\ y - x-y$），計算兩個 *Integer* x 和 y 的差 $x-y$。請注意，第一個參數 x 是距堆疊頂部的第二個條目，而第二個參數 y 從堆疊頂部取得。
- `negate`（$x - -x$），改變 *Integer* 的符號。
- `*`（$x\ y - xy$），計算兩個 *Integer* x 和 y 的積 xy。
- `/`（$x\ y - q := \lfloor x/y \rfloor$），計算兩個 *Integer* 的向下取整商 $\lfloor x/y \rfloor$。
- `mod`（$x\ y - r := x \bmod y$），計算 x 除以 y 的餘數 $x \bmod y = x - y \cdot \lfloor x/y \rfloor$。
- `/mod`（$x\ y - q\ r$），同時計算商和餘數。
- `/c`、`/r`（$x\ y - q$），類似於 `/` 的除法詞彙，但分別使用向上取整（$q := \lceil x/y \rceil$）和最接近整數取整（$q := \lfloor 1/2 + x/y \rfloor$）。
- `/cmod`、`/rmod`（$x\ y - q\ r := x - qy$），類似於 `/mod` 的除法詞彙，但使用向上取整或最接近整數取整。
- `<<`（$x\ y - x \cdot 2^y$），計算二進位數 x 的算術左移 $y \geq 0$ 個位置，產生 $x \cdot 2^y$。
- `>>`（$x\ y - q := \lfloor x \cdot 2^{-y} \rfloor$），計算算術右移 $y \geq 0$ 個位置。

- `>>c`、`>>r`（$x\ y - q$），類似於 `>>`，但使用向上取整或最接近整數取整。
- `and`、`or`、`xor`（$x\ y - x \oplus y$），計算兩個 *Integer* 的位元 AND、OR 或 XOR。
- `not`（$x - -1 - x$），*Integer* 的位元補數。
- `*/`（$x\ y\ z - \lfloor xy/z \rfloor$），「先乘後除」：將兩個整數 x 和 y 相乘產生 513 位元中間結果，然後將乘積除以 z。
- `*/mod`（$x\ y\ z - q\ r$），類似於 `*/`，但同時計算商和餘數。
- `*/c`、`*/r`（$x\ y\ z - q$）、`*/cmod`、`*/rmod`（$x\ y\ z - q\ r$），類似於 `*/` 或 `*/mod`，但使用向上取整或最接近整數取整。
- `*>>`、`*>>c`、`*>>r`（$x\ y\ z - q$），類似於 `*/` 及其變體，但用右移替換除法。計算以指示方式（向下、向上或最接近整數）取整的 $q = xy/2^z$。
- `<</`、`<</c`、`<</r`（$x\ y\ z - q$），類似於 `*/`，但用左移替換乘法。計算以指示方式取整的 $q = 2^z x/y$（注意與 `*/` 相比，參數 y 和 z 的順序不同）。

此外，詞彙「`.`」可用於列印堆疊頂部傳遞的 *Integer* 的十進位表示（後跟單個空格），「`x.`」列印堆疊頂部整數的十六進位表示。之後整數從堆疊中移除。

上述原語可用於在互動模式下使用 Fift 解釋器作為以逆波蘭記號（運算符號在運算元之後）表示的算術表示式的簡單計算器。例如，

```
7 4 - .
```

計算 $7 - 4 = 3$ 並列印「`3 ok`」，和

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

計算 $2 + 3 \cdot 4 = 14$ 和 $(2 + 3) \cdot 4 = 20$，並列印「`14 20 ok`」。

2.5 堆疊操作詞彙

堆疊操作詞彙重新排列堆疊頂部附近的一個或多個值，無論其型別如何，並保持所有更深的堆疊值不變。下面列出了一些最常用的堆疊操作詞彙：

- `dup`（$x - x\ x$），複製堆疊頂部條目。如果堆疊為空，則拋出異常。[4]

[4]請注意，Fift 詞彙名稱區分大小寫，因此不能輸入 `DUP` 而不是 `dup`。

- `drop`（x –），移除堆疊頂部條目。
- `swap`（$x\ y$ – $y\ x$），交換兩個最頂層的堆疊條目。
- `rot`（$x\ y\ z$ – $y\ z\ x$），旋轉三個最頂層的堆疊條目。
- `-rot`（$x\ y\ z$ – $z\ x\ y$），以相反方向旋轉三個最頂層的堆疊條目。等同於 `rot rot`。
- `over`（$x\ y$ – $x\ y\ x$），在堆疊頂部條目之上建立距頂部第二個堆疊條目的副本。
- `tuck`（$x\ y$ – $y\ x\ y$），等同於 `swap over`。
- `nip`（$x\ y$ – y），移除距頂部的第二個堆疊條目。等同於 `swap drop`。
- `2dup`（$x\ y$ – $x\ y\ x\ y$），等同於 `over over`。
- `2drop`（$x\ y$ – ），等同於 `drop drop`。
- `2swap`（$a\ b\ c\ d$ – $c\ d\ a\ b$），交換兩對最頂層的堆疊條目。
- `pick`（x_n …$x_0\ n$ – x_n …$x_0\ x_n$），建立距堆疊頂部第 n 個條目的副本，其中 $n \geq 0$ 也在堆疊中傳遞。特別是，`0 pick` 等同於 `dup`，`1 pick` 等同於 `over`。
- `roll`（x_n …$x_0\ n$ – x_{n-1} …$x_0\ x_n$），旋轉前 n 個堆疊條目，其中 $n \geq 0$ 也在堆疊中傳遞。特別是，`1 roll` 等同於 `swap`，`2 roll` 等同於 `rot`。
- `-roll`（x_n …$x_0\ n$ – $x_0\ x_n$ …x_1），以相反方向旋轉前 n 個堆疊條目，其中 $n \geq 0$ 也在堆疊中傳遞。特別是，`1 -roll` 等同於 `swap`，`2 -roll` 等同於 `-rot`。
- `exch`（x_n …$x_0\ n$ – x_0 …x_n），將堆疊頂部與距頂部第 n 個堆疊條目交換，其中 $n \geq 0$ 也從堆疊中取得。特別是，`1 exch` 等同於 `swap`，`2 exch` 等同於 `swap rot`。
- `exch2`（…$n\ m$ – …），將距頂部第 n 個堆疊條目與距頂部第 m 個堆疊條目交換，其中 $n \geq 0$、$m \geq 0$ 從堆疊中取得。
- `?dup`（x – $x\ x$ 或 0），複製 *Integer* x，但僅當它非零時。否則保持不變。

例如，「5 dup * .」將計算 $5 \cdot 5 = 25$ 並列印「25 ok」。

可以使用詞彙「.s」——它列印整個堆疊的內容，從最深的元素開始，而不從堆疊中移除列印的元素——隨時檢查堆疊的內容，並檢查任何堆疊操作詞彙的效果。例如，

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

列印

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

當 Fift 不知道如何列印未知型別的堆疊值時，它會列印???。

2.6 定義新詞彙

最簡單的形式，定義新的 Fift 詞彙非常容易，可以藉助三個特殊詞彙來完成：「{」、「}」和「:」。只需用 { （後面必須跟一個空格）打開定義，然後列出構成新定義的所有詞彙，然後用}（也跟一個空格）關閉定義，最後透過編寫: ⟨*new-word-name*⟩ 將結果定義（由堆疊中的 *WordDef* 值表示）分配給新詞彙。例如，

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

定義了一個新詞彙 square，它在呼叫時執行 dup 和 *。這樣，輸入 5 square 就等同於輸入 5 dup *，並產生相同的結果（25）：

```
5 square .
```

列印「25 ok」。還可以將新詞彙用作新定義的一部分：

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

列印「243 ok」，這確實是 3^5。

如果「:」之後指示的詞彙已經定義，它將被默默地重新定義。然而，所有其他詞彙的現有定義將繼續使用重新定義詞彙的舊定義。例如，如果我們在已經如上定義 **5 之後重新定義 square，**5 將繼續使用 square 的原始定義。

2.7 命名常數

可以透過使用定義詞彙 constant 而不是定義詞彙「:」（冒號）來定義（命名）常數——即在呼叫時推入預定義值的詞彙。例如，

```
1000000000 constant Gram
```

定義了等於 *Integer* 10^9 的常數 Gram。換句話說，每當呼叫 Gram 時，1000000000 將被推入堆疊：

```
Gram 2 * .
```

列印「2000000000 ok」。

當然，可以使用計算結果來初始化常數的值：

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

列印「1000000 ok」。

常數的值不一定必須是 *Integer*。例如，可以用相同的方式定義字串常數：

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

在單獨一行上列印「Hello, world!」。

如果重新定義常數，所有其他詞彙的現有定義將繼續使用常數的舊值。在這方面，常數的行為不像全域變數。

還可以使用定義詞彙 2constant 將兩個值儲存到一個「雙」常數中。例如，

```
355 113 2constant pifrac
```

定義了一個新詞彙 pifrac，它在呼叫時將推入 355 和 113（按此順序）。雙常數的兩個組成部分可以是不同型別。

如果想在區塊或冒號定義中建立具有固定名稱的常數，應使用 =: 和 2=: 而不是 constant 和 2constant：

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

產生

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

如果想恢復這種「常數」的執行時期值，可以在常數名稱前加上詞彙 `@'`：

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

產生

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

這種方法的缺點是 `@'` 每次執行 `showxy` 時都必須在字典中查找常數 `x` 和 `y` 的當前定義。變數（參見 **2.14**）提供了一種更有效的方式來實現類似的結果。

2.8 整數和分數常數，或字面值

Fift 識別十進位、二進位和十六進位格式的未命名整數常數（稱為字面值以將它們與命名常數區分開來）。二進位字面值以 `0b` 為前綴，十六進位字面值以 `0x` 為前綴，十進位字面值不需要前綴。例如，`0b1011`、`11` 和 `0xb` 表示相同的整數（11）。整數字面值可以以減號「`-`」為前綴來改變其符號；減號在 `0x` 和 `0b` 前綴之前和之後都被接受。

當 Fift 遇到字典中不存在但是有效整數字面值（適合 257 位元帶符號整數型別 *Integer*）的字串時，其值被推入堆疊。

除此之外，Fift 對十進位和普通分數提供一些支援。如果字串由兩個由斜線 `/` 分隔的有效整數字面值組成，則 Fift 將其解釋為分數字面值，並在堆疊中用兩個 *Integer* p 和 q 表示它，分子 p 和分母 q。例如，`-17/12` 將 -17 和 12 推入 Fift 堆疊（因此等同於 `-17 12`），`-0x11/0b1100` 做同樣的事情。十進位、二進位和十六進位分數，例如 `2.39` 或 `-0x11.ef`，也由兩個整數 p 和 q 表示，其中 q 是基數（分別為 10、2 或 16）的適當次方。例如，`2.39` 等同於 `239 100`，`-0x11.ef` 等同於 `-0x11ef 0x100`。

這種分數表示特別方便用於縮放原語 `*/` 及其變體，從而將普通和十進位分數轉換為適當的定點表示。例如，如果我們想用 nanogram 的整數數量表示 Gram 的分數數量，可以定義一些輔助詞彙

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

然後編寫 `2.39 Gram*/` 或 `17/12 Gram*/` 而不是整數字面值 `2390000000` 或 `1416666667`。

如果經常需要使用這樣的 Gram 字面值，可以如下引入新的主動前綴詞彙 `GR$`：

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

使 `GR$3`、`GR$2.39` 和 `GR$17/12` 分別等同於整數字面值 `3000000000`、`2390000000` 和 `1416666667`。這樣的值可以透過以下詞彙以類似形式列印：

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

產生 `GR$-17.239 ok`。上述定義使用了本文件後面部分（特別是第 **4** 章）中解釋的技巧。

我們還可以透過定義適當的「有理數算術詞彙」來自己操作分數：

```
// 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.
```

將輸出「`31/30 ok`」，表示 $1.7 - 2/3 = 31/30$。這裡「`._`」是「`.`」的變體，它在 *Integer* 的十進位表示後不列印空格。

2.9 字串字面值

字串字面值透過前綴詞彙`"` 引入，它掃描該行的其餘部分直到下一個`"` 字元，並將由此獲得的字串作為型別 *String* 的值推入堆疊。例如，`"Hello, world!"` 將相應的 *String* 推入堆疊：

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

2.10 簡單字串操作

以下詞彙可用於操作字串：

- "⟨*string*⟩"（– *S*），將 *String* 字面值推入堆疊。
- ."⟨*string*⟩"（– ），將常數字串列印到標準輸出。
- type（*S* – ），將從堆疊頂部取得的 *String S* 列印到標準輸出。
- cr（– ），將回車（或換行字元）輸出到標準輸出。
- emit（*x* – ），將具有由 *Integer x* 給出的 Unicode 碼點的 UTF-8 編碼字元列印到標準輸出。
- char ⟨*string*⟩（– *x*），推入帶有⟨*string*⟩ 第一個字元的 Unicode 碼點的 *Integer*。
- bl（– *x*），推入空格的 Unicode 碼點，即 32。
- space（– ），列印一個空格，等同於 bl emit。
- $+（*S S′* – *S.S′*），串接兩個字串。
- $len（*S* – *x*），計算字串的位元組長度（不是 UTF-8 字元長度 ⁉）。
- +"⟨*string*⟩"（*S* – *S′*），將 *String S* 與字串字面值串接。等同於"⟨*string*⟩" $+。
- word（*x* – *S*），從當前輸入行的其餘部分解析由 Unicode 碼點 *x* 的字元分隔的詞彙，並將結果作為 *String* 推入。例如，bl word abracadabra type 將列印字串「abracadabra」。如果 $x = 0$，跳過前導空格，然後掃描到當前輸入行的末尾。如果 $x = 32$，在解析下一個詞彙之前跳過前導空格。
- (.)（*x* – *S*），返回帶有 *Integer x* 十進位表示的 *String*。
- (number)（*S* – 0 或 *x* 1 或 *x y* 2），嘗試將 *String S* 解析為 **2.8** 中解釋的整數或分數字面值。

例如，."*"、"*" type、42 emit 和 char * emit 是輸出單個星號的四種不同方式。

2.11 布林表示式，或旗標

Fift 沒有用於表示布林值的單獨值型別。相反，任何非零 *Integer* 可用於表示真（-1 是標準表示），而零 *Integer* 表示假。比較原語通常返回 -1 表示成功，否則返回 0。

常數 `true` 和 `false` 可用於將這些特殊整數推入堆疊：

- `true`（– -1），將 -1 推入堆疊。
- `false`（– 0），將 0 推入堆疊。

如果布林值是標準的（0 或 -1），可以透過 **2.4** 中列出的位元邏輯運算 `and`、`or`、`xor`、`not` 來操作它們。否則，必須首先使用 `0<>` 將它們轉換為標準形式：

- `0<>`（x – $x \neq 0$），如果 *Integer* x 非零則推入 -1，否則推入 0。

2.12 整數比較運算

幾個整數比較運算可用於獲得布林值：

- `<`（x y – ?），檢查 $x < y$ 是否成立（即如果 $x < y$ 則推入 -1，否則推入 0）。
- `>`、`=`、`<>`、`<=`、`>=`（x y – ?），比較 x 和 y 並根據比較結果推入 -1 或 0。
- `0<`（x – ?），檢查 $x < 0$ 是否成立（即如果 x 為負則推入 -1，否則推入 0）。等同於 `0 <`。
- `0>`、`0=`、`0<>`、`0<=`、`0>=`（x – ?），將 x 與零比較。
- `cmp`（x y – z），如果 $x > y$ 則推入 1，如果 $x < y$ 則推入 -1，如果 $x = y$ 則推入 0。
- `sgn`（x – y），如果 $x > 0$ 則推入 1，如果 $x < 0$ 則推入 -1，如果 $x = 0$ 則推入 0。等同於 `0 cmp`。

範例：

```
2 3 < .
```

列印「-1 ok」，因為 2 小於 3。

一個更複雜的範例：

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

列印「true false false ok」。

2.13 字串比較運算

字串可以透過以下詞彙進行字典序比較：

- $= ($S\ S' - ?$)，如果字串 S 和 S' 相等則返回 -1，否則返回 0。
- $cmp ($S\ S' - x$)，如果字串 S 和 S' 相等則返回 0，如果 S 在字典序上小於 S' 則返回 -1，如果 S 在字典序上大於 S' 則返回 1。

2.14 命名和未命名變數

除了 **2.7** 中介紹的常數外，Fift 支援變數，這是表示可變值的更有效方式。例如， **2.7** 的最後兩個程式碼片段可以使用變數而不是常數來編寫，如下所示：

```
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
```

產生與之前相同的輸出：

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

短語 variable x 建立一個新的 *Box*，即一個可用於儲存任何 Fift 支援型別的單一值的記憶體位置，並將 x 定義為等於此 *Box* 的常數：

- variable（– ），從輸入的其餘部分掃描以空白分隔的詞彙名稱 S，分配一個空的 *Box*，並將新的普通詞彙 S 定義為常數，它在呼叫時將推入新的 *Box*。等同於 hole constant。

- `hole`（– *p*），建立一個不包含任何值的新 *Box p*。等同於 `null box`。
- `box`（*x* – *p*），建立一個包含指定值 *x* 的新 *Box*。等同於 `hole tuck !`。

當前儲存在 *Box* 中的值可以透過詞彙 `@`（發音為「fetch」）取得，並可以透過詞彙`!`（發音為「store」）修改：

- `@`（*p* – *x*），取得當前儲存在 *Box p* 中的值。
- `!`（*x p* – ），將新值 *x* 儲存到 *Box p* 中。

存在幾個輔助詞彙可以更複雜的方式修改當前值：

- `+!`（*x p* – ），將儲存在 *Box p* 中的整數值增加 *Integer x*。等同於 `tuck @ + swap !`。
- `1+!`（*p* – ），將儲存在 *Box p* 中的整數值增加一。等同於 `1 swap +!`。
- `0!`（*p* – ），將 *Integer* 0 儲存到 *Box p* 中。等同於 `0 swap !`。

這樣我們可以實作一個簡單的計數器：

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

產生

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

在這些定義就位後，我們甚至可以透過短語 `forget counter` 忘記 `counter` 的定義。那麼存取此變數值的唯一方法是透過 `reset-counter` 和 `next-counter`。

變數通常由 `variable` 建立而沒有值，或者更確切地說具有 *Null* 值。如果想建立已初始化的變數，可以使用短語 `box constant`：

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

列印「`18 ok`」。如果經常需要已初始化的變數，甚至可以定義特殊的定義詞彙：

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

列印「18 test ok」。

到目前為止，變數與常數相比只有一個缺點：必須透過輔助詞彙 @ 存取它們的當前值。當然，可以透過為變數定義「getter」和「setter」詞彙來緩解這一點，並使用這些詞彙編寫更好看的程式碼：

```
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
```

列印「(3 , 30) (5 , 56) ok」，這是 $f(x) = x^2 + 5x + 6$ 圖形上的點 $(x, f(x))$，其中 $x = 3$ 和 $x = 5$。

同樣，如果我們想為所有變數定義「getter」，可以首先定義一個定義詞彙，如 **4.8** 中所述，並使用此詞彙同時定義 getter 和 setter：

```
{ 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
```

產生

```
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 元組和陣列

Fift 也支援 *Tuple*，即堆疊值型別的任意值的不可變有序集合（參見 **2.1**）。當 *Tuple* t 由值 x_1、…、x_n（按此順序）組成時，我們寫作 $t = (x_1, \ldots, x_n)$。數字 n 稱為 *Tuple* t 的長度；它也用 $|t|$ 表示。長度為二的元組也稱為對，長度為三的元組稱為三元組。

- `tuple`（x_1 …x_n n – t），從 $n \geq 0$ 個最頂層堆疊值建立新的 *Tuple* $t := (x_1, \ldots, x_n)$。
- `pair`（x y – t），建立新的對 $t = (x, y)$。等同於 `2 tuple`。
- `triple`（x y z – t），建立新的三元組 $t = (x, y, z)$。等同於 `3 tuple`。
- `|`（– t），建立空的 *Tuple* $t = ()$。等同於 `0 tuple`。
- `,`（t x – t'），將 x 附加到 *Tuple* t 的末尾，並返回結果 *Tuple* t'。
- `.dump`（x – ），以與`.s` 傾印所有堆疊元素相同的方式傾印最頂層的堆疊條目。

例如，

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

和

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

都建構並列印三元組 $(2, 3, 9)$：

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

請注意，*Tuple* 的組成部分不一定是同一型別，並且 *Tuple* 的組成部分也可以是 *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
```

產生

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

一旦建構了 *Tuple*，我們可以提取其任何組成部分，或將 *Tuple* 完全解包到堆疊中：

- `untuple`（$t\ n - x_1\ \ldots x_n$），返回 *Tuple* $t = (x_1, \ldots, x_n)$ 的所有組成部分，但僅當其長度等於 n 時。否則拋出異常。
- `unpair`（$t - x\ y$），解包對 $t = (x, y)$。等同於 `2 untuple`。
- `untriple`（$t - x\ y\ z$），解包三元組 $t = (x, y, z)$。等同於 `3 untuple`。
- `explode`（$t - x_1\ \ldots x_n\ n$），解包未知長度 n 的 *Tuple* $t = (x_1, \ldots, x_n)$，並返回該長度。
- `count`（$t - n$），返回 *Tuple* t 的長度 $n = |t|$。
- `tuple?`（$t - ?$），檢查 t 是否為 *Tuple*，並相應地返回 -1 或 0。
- `[]`（$t\ i - x$），返回 *Tuple* t 的第 $(i + 1)$ 個組成部分 t_{i+1}，其中 $0 \leq i < |t|$。
- `first`（$t - x$），返回 *Tuple* 的第一個組成部分。等同於 `0 []`。
- `second`（$t - x$），返回 *Tuple* 的第二個組成部分。等同於 `1 []`。
- `third`（$t - x$），返回 *Tuple* 的第三個組成部分。等同於 `2 []`。

例如，我們可以存取矩陣的單個元素和行：

```
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
```

產生

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

請注意，*Tuple* 在某種程度上類似於其他程式語言的陣列，但它是不可變的：我們無法改變 *Tuple* 的單個元件。如果我們仍然想建立類似陣列的東西，我們需要一個由 *Box* 組成的 *Tuple*（參見 **2.14**）：

- `allot`（n – t），建立一個由 n 個新的空 *Box* 組成的 *Tuple*。等同於 `| { hole , } rot times`。

例如，

```
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[]
```

建立了一個長度為 10 的未初始化陣列 A、一個長度為 6 的已初始化陣列 B，然後交換 B 的某些元素，並印出結果 B 的前四個元素：

```
4 1 1 3  ok
```

2.16 串列

Fift 也可以表示 Lisp 風格的串列。首先，引入兩個特殊詞彙來操作 *Null* 型別的值，用於表示空串列（不要與空 *Tuple* 混淆）：

- `null`（– ⊥），推入 *Null* 型別的唯一值 ⊥，也用於表示空串列。
- `null?`（x – ?），檢查 x 是否為 *Null*。也可用於檢查串列是否為空。

之後，`cons` 和 `uncons` 被定義為 `pair` 和 `unpair` 的別名：

- `cons`（h t – l），從其頭部（第一個元素）h 和尾部（由所有剩餘元素組成的串列）t 建構串列。等同於 `pair`。
- `uncons`（l – h t），將非空串列分解為其頭部和尾部。等同於 `unpair`。
- `car`（l – h），回傳串列的頭部。等同於 `first`。
- `cdr`（l – t），回傳串列的尾部。等同於 `second`。
- `cadr`（l – h'），回傳串列的第二個元素。等同於 `cdr car`。
- `list`（x_1 …x_n n – l），建構一個長度為 n 的串列 l，元素依序為 x_1、…、x_n。等同於 `null ' cons rot times`。

- `.l` (l –)，印出 Lisp 風格的串列 l。

例如，

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

產生

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

請注意，三元素串列 (2 3 9) 不同於三元組 $(2, 3, 9)$。

2.17 原子

Atom（原子）是由其名稱唯一識別的簡單實體。*Atom* 可用於表示識別符號、標籤、運算名稱、標籤和堆疊標記。Fift 提供以下詞彙來操作 *Atom*：

- `(atom)`（S x – a -1 或 0），回傳由 *String* S 給定名稱的唯一 *Atom* a。如果尚不存在這樣的 *Atom*，則在 *Integer* x 非零時建立它，或在 x 為零時回傳單一零以表示失敗。
- `atom`（S – a），回傳名稱為 S 的唯一 *Atom* a，必要時建立這樣的原子。等同於 `true (atom) drop`。
- `` `⟨word⟩ ``（– a），引入 *Atom* 字面值，等於名稱等於 ⟨*word*⟩ 的唯一 *Atom*。等同於`"⟨word⟩" atom`。
- `anon`（– a），建立一個新的唯一匿名 *Atom*。
- `atom?`（u – ?），檢查 u 是否為 *Atom*。
- `eq?`（u v – ?），檢查 u 和 v 是否為相等的 *Integer*、*Atom* 或 *Null*。如果它們不相等，或者它們屬於不同型別，或者不屬於列出的型別之一，則回傳零。

例如，

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

建立並印出串列

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

這是算術表示式 $2 + 3 \cdot 4$ 的 Lisp 風格表示。這類表示式的解譯器可能使用 `eq?` 來檢查運算符號（參見 **3.5** 關於 Fift 中遞迴函數的說明）：

```
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
```

印出

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

如果我們載入 `Lisp.fif` 以啟用 Lisp 風格的串列語法，我們可以輸入

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

得到與之前相同的結果。在 `Lisp.fif` 中定義的詞彙 `(`，使用由 `anon` 建立的匿名 *Atom* 來標記當前堆疊位置，然後) 從數個頂層堆疊項目建立串列，掃描堆疊直到找到匿名 *Atom* 標記：

```
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 腳本模式中的命令列引數

Fift 解譯器可以通過在命令列選項中傳遞 `-s` 來以腳本模式呼叫。在此模式下，所有後續的命令列引數都不會被掃描以尋找 Fift 啟動命令列選項。相反，`-s` 之後的下一個引數用作 Fift 來源檔案的檔名，所有後續的命令列引數通過特殊詞彙 $\texttt{\$}n$ 和 `$#` 傳遞給 Fift 程式：

- `$#` $(- x)$，推入傳遞給 Fift 程式的命令列引數總數。
- $\texttt{\$}n$ $(- S)$，將第 n 個命令列引數作為 *String* S 推入。例如，`$0` 推入正在執行的腳本名稱，`$1` 推入第一個命令列引數，依此類推。
- `$()` $(x - S)$，類似於 $\texttt{\$}n$ 推入第 x 個命令列引數，但 *Integer* x 從堆疊中取得。

此外，如果 Fift 來源檔案的第一行以兩個字元「`#!`」開頭，則此行將被忽略。這樣就可以在 *ix 系統中編寫簡單的 Fift 腳本。例如，如果

```
#!/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
```

被儲存到當前目錄中的檔案 `cmdline.fif`，並設定其執行位元（例如，通過 `chmod 755 cmdline.fif`），那麼它可以從 shell 或任何其他程式呼叫，前提是 Fift 解譯器安裝為 `/usr/bin/fift`，其標準函式庫 `Fift.fif` 安裝為 `/usr/lib/fift/Fift.fif`：

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

印出

```
-60
```

當從 *ix shell（如 Bourne-again shell（Bash））呼叫時。

3 區塊、迴圈和條件式

與算術運算類似，Fift 中的執行流程由基於堆疊的基本操作控制。這導致了反向波蘭表示法和基於堆疊的算術的典型反轉：首先將表示條件分支或迴圈主體的區塊推入堆疊，然後呼叫條件式或迭代執行基本操作。在這方面，Fift 更類似於 PostScript 而不是 Forth。

3.1 定義和執行區塊

區塊通常使用特殊詞彙「`{`」和「`}`」來定義。粗略地說，在 `{` 和`}` 之間列出的所有詞彙構成新區塊的主體，該區塊作為 *WordDef* 型別的值推入堆疊。區塊可以通過定義詞彙「`:`」儲存為新 Fift 詞彙的定義（如 **2.6** 中所述），或通過詞彙 `execute` 執行：

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

印出「`34 ok`」，本質上等同於「`17 2 * .`」。一個稍微複雜的例子：

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

將「匿名函數」$x \mapsto 2x$ 兩次應用於 17，並印出結果 $2 \cdot (2 \cdot 17) = 68$。這樣，區塊就是執行令牌，可以複製、儲存到常數中、用於定義新詞彙或執行。

詞彙`'` 恢復詞彙的當前定義。也就是說，構造`'` ⟨*word-name*⟩ 推入等同於詞彙 ⟨*word-name*⟩ 當前定義的執行令牌。例如，

```
' dup execute
```

等同於 `dup`，而

```
' dup : duplicate
```

將 `duplicate` 定義為 `dup`（當前定義）的同義詞。

或者，我們可以複製一個區塊來定義兩個具有相同定義的新詞彙：

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

將 `square` 和 `**2` 都定義為等同於 `dup *`。

3.2 區塊的條件式執行

區塊的條件式執行使用詞彙 `if`、`ifnot` 和 `cond` 實現：

- `if`（$x\ e\ –$），執行 e（必須是執行令牌，即 *WordDef*），[5]但僅當 *Integer* x 非零時執行。
- `ifnot`（$x\ e\ –$），執行執行令牌 e，但僅當 *Integer* x 為零時執行。
- `cond`（$x\ e\ e'\ –$），如果 *Integer* x 非零，則執行 e，否則執行 e'。

例如，**2.12** 中的最後一個例子可以更方便地改寫為

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

仍然產生「`true false false ok`」。

請注意，如前例所示，區塊可以任意巢狀。例如，可以編寫

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

得到「`negative ok`」，因為 -17 是負數。

3.3 簡單迴圈

最簡單的迴圈由 `times` 實現：

- `times`（$e\ n\ –$），如果 $n \geq 0$，則恰好執行 e n 次。如果 n 為負數，則拋出例外。

例如，

[5] *WordDef* 比 *WordList* 更通用。例如，基本運算 `+` 的定義是 *WordDef*，但不是 *WordList*，因為 `+` 不是用其他 Fift 詞彙定義的。

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

計算並印出 10^{70}。

我們可以使用這種迴圈來實現簡單的階乘函數：

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

印出「`120 ok`」，因為 $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$。

這個迴圈可以修改為計算費波那契數：

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

計算第六個費波那契數 $F_6 = 13$。

3.4 具有退出條件的迴圈

可以藉助 `until` 和 `while` 建立更複雜的迴圈：

- `until`（e –），執行 e，然後移除堆疊頂端整數並檢查它是否為零。如果是，則通過執行 e 開始迴圈的新迭代。否則退出迴圈。
- `while`（e e' –），執行 e，然後移除並檢查堆疊頂端整數。如果為零，則退出迴圈。否則執行 e'，然後通過執行 e 並在之後檢查退出條件來開始新的迴圈迭代。

例如，我們可以計算大於 1000 的前兩個費波那契數：

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

印出「`1597 2584 ok`」。

我們可以使用這個詞彙來計算黃金比例 $\phi = (1 + \sqrt{5})/2 \approx 1.61803$ 的前 70 位十進位數字：

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

印出「161803…2604 `ok`」。

3.5 遞迴

請注意，每當在 { …} 區塊內提及詞彙時，當前（編譯時）的定義就會包含在正在建立的 *WordList* 中。這樣，我們可以在定義詞彙的新版本時參照其先前的定義：

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

產生「number 2 number 3 5 number 8 ok」。請注意，print-sum 繼續使用「.」的原始定義，但 print-next 已經使用了修改後的「.」。

這個特性在某些情況下可能很方便，但它阻止我們以最直接的方式引入遞迴定義。例如，階乘的經典遞迴定義

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

將無法編譯，因為在編譯定義時 fact 恰好是未定義的詞彙。

繞過這個障礙的一個簡單方法是使用詞彙 @'（參見 **4.6**），它在執行時查找下一個詞彙的當前定義，然後執行它，類似於我們在 **2.7** 中已經做的：

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

如預期產生「120 ok」。

然而，這個解決方案相當低效，因為每次遞迴執行 fact 時都會使用字典查找。我們可以通過使用變數（參見 **2.14** 和 **2.7**）來避免這種字典查找：

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

這個稍長的階乘定義通過引入一個特殊變數'fact 來保存階乘的最終定義，從而避免了執行時的字典查找。[6]然後 fact 被定義為執行當前儲存在'fact 中的任何 *WordDef*，一旦構造了階乘遞迴定義的主體，就通過短語'fact ! 將其儲存到此變數中，該短語替換了更常見的短語: fact。

我們可以通過為向量變數'fact 使用特殊的「取值器」和「設定器」詞彙來重寫上述定義，就像我們在 **2.14** 中為變數所做的那樣：

[6]保存稍後要 execute 的 *WordDef* 的變數稱為向量變數。將 fact 替換為'fact @ execute（其中'fact 是向量變數）的過程稱為向量化。

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

如果我們需要引入大量遞迴和相互遞迴的定義，我們可能首先引入一個自訂定義詞彙（參見 **4.8**）來同時定義匿名向量變數的「取值器」和「設定器」詞彙，類似於我們在 **2.14** 中所做的：

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

此片段的前三行定義 `fact` 和`:fact`，本質上與前一片段的前四行定義它們的方式相同。

如果我們希望使 `fact` 在未來不可改變，我們可以在階乘的定義完成後新增一行 `forget :fact`：

```
{ 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 .
```

或者，我們可以修改 `vector-set` 的定義，使`:fact` 在被呼叫一次後會忘記自己：

```
{ 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 .
```

然而，某些向量變數必須被修改多次，例如，在合併排序演算法中修改比較詞彙 `less` 的行為：

```
{ 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
```

產生以下輸出：

```
(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 拋出例外

兩個內建詞彙用於拋出例外：

- `abort`（S – ），拋出一個例外，錯誤訊息取自 *String* S。
- `abort"`⟨*message*⟩`"`（x – ），如果 x 是非零整數，則拋出一個例外，錯誤訊息為 ⟨*message*⟩。

這些詞彙拋出的例外由 C++ 例外 `fift::IntError` 表示，其值等於指定的字串。它通常在 Fift 解譯器內部處理，通過中止所有執行直到頂層，並印出包含正在解譯的來源檔名稱、行號、當前解譯的詞彙和指定的錯誤訊息的訊息。例如：

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

印出「`safe/: Division by zero`」，沒有通常的「`ok`」。在此過程中堆疊被清空。

順便一提，當 Fift 解譯器遇到無法解析為整數字面值的未知詞彙時，會拋出訊息為「`-?`」的例外，效果如上所述，包括堆疊被清空。

4 字典、解譯器和編譯器

在本章中，我們介紹幾個用於字典操作和編譯器控制的特定 Fift 詞彙。「編譯器」是 Fift 解譯器的一部分，它從詞彙名稱建立詞彙參照串列（由 *WordList* 堆疊值表示）；它由用於定義區塊的基本運算「`{`」啟動，如 **2.6** 和 **3.1** 中所述。

本章包含的大部分資訊相當複雜，在第一次閱讀時可以跳過。然而，這裡描述的技術被 Fift 組譯器大量使用，用於編譯 TVM 程式碼。因此，如果想要理解 Fift 組譯器的當前實現，本節是不可或缺的。

4.1 Fift 解譯器的狀態

Fift 解譯器的狀態由一個稱為 `state` 的內部整數變數控制，目前無法從 Fift 本身存取。當 `state` 為零時，所有從輸入（即 Fift 來源檔案或互動模式下的標準輸入）解析的詞彙都會在字典中查找，然後立即執行。當 `state` 為正數時，在字典中找到的詞彙不會執行。相反，它們（或更確切地說，對其當前定義的參照）被編譯，即新增到正在建構的 *WordList* 的末尾。

通常，state 等於當前開啟區塊的數量。例如，在解譯「{ 0= { ."zero"」之後，state 變數將等於二，因為有兩個巢狀區塊。正在建構的 *WordList* 保存在堆疊頂部。

基本運算「{」只是將一個新的空 *WordList* 推入堆疊，並將 state 加一。基本運算「}」在 state 已經為零時拋出例外；否則將 state 減一，並將結果 *WordList* 留在堆疊中，表示剛剛建構的區塊。[7]之後，如果 state 的結果值非零，則新區塊作為字面值（未命名常數）編譯到外圍區塊中。

4.2 活躍詞彙和普通詞彙

所有字典詞彙都有一個特殊旗標，指示它們是活躍詞彙還是普通詞彙。預設情況下，所有詞彙都是普通的。特別是，所有使用「:」和 constant 定義的詞彙都是普通的。

當 Fift 解譯器在字典中找到詞彙定義時，它會檢查該詞彙是否為普通詞彙。如果是，則當前詞彙定義要麼被執行（如果 state 為零），要麼被「編譯」（如果 state 大於零），如 **4.1** 中所述。

另一方面，如果詞彙是活躍的，則即使 state 為正數，它也總是被執行。活躍詞彙預期在堆疊中留下一些值 x_1 ...x_n n e，其中 $n \geq 0$ 是整數，$x_1 \ldots x_n$ 是 n 個任意型別的值，e 是執行令牌（*WordDef* 型別的值）。之後，解譯器根據 state 執行不同的動作：如果 state 為零，則丟棄 n 並執行 e，就像找到 nip execute 短語一樣。如果 state 非零，則此集合被「編譯」到當前 *WordList*（位於堆疊中 x_1 的正下方）中，其方式與呼叫 (compile) 基本運算相同。此編譯相當於在當前 *WordList* 的末尾新增一些程式碼，當呼叫時會將 x_1、...、x_n 推入堆疊，然後新增對 e 的參照（表示延遲執行 e）。如果 e 等於特殊值'nop（表示執行時不執行任何操作的執行令牌），則省略最後一步。

4.3 編譯字面值

當 Fift 解譯器遇到字典中不存在的詞彙時，它會呼叫基本運算 (number) 嘗試將其解析為整數或分數字面值。如果此嘗試成功，則推入特殊值'nop，並且解譯以與遇到活躍詞彙相同的方式繼續進行。換句話說，如果 state 為零，則字面值只是留在堆疊中；否則，呼叫 (compile) 來修改當前 *WordList*，使其在執行時推入字面值。

[7]詞彙} 還將此 *WordList* 轉換為 *WordDef*，後者具有不同的型別標籤，因此是不同的 Fift 值，即使 C++ 實現使用相同的底層 C++ 物件。

4.4 定義新的活躍詞彙

新的活躍詞彙的定義與新的普通詞彙類似，但使用「::」而不是「:」。例如，

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

定義活躍詞彙 say，它掃描其後的下一個以空格分隔的詞彙，並將其作為字面值與當前定義的 type 的參照一起編譯到當前 *WordList* 中（如果 state 非零，即如果 Fift 解譯器正在編譯區塊）。當呼叫時，區塊的此新增部分將把儲存的字串推入堆疊並執行 type，從而印出 say 之後的下一個詞彙。另一方面，如果 state 為零，則這兩個動作由 Fift 解譯器立即執行。這樣，

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

將印出「hello3 ok」，而

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

將印出「hello3 hello6 ok」。

當然，可以使用區塊來表示所需的動作，而不是' type。例如，如果我們想要一個在儲存的詞彙後印出空格的 say 版本，我們可以寫

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

得到「hello 3 hello 6 ok」。

順便一提，詞彙"（引入字串字面值）和."（印出字串字面值）可以定義如下：

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

新的定義詞彙「::_」定義活躍前綴詞彙，即不需要後面有空格的活躍詞彙。

4.5 定義詞彙和字典操作

定義詞彙是在 Fift 字典中定義新詞彙的詞彙。例如，「:」、「::_」和 constant 都是定義詞彙。所有這些定義詞彙都可以使用基本運算 (create) 定義；實際上，如果需要，使用者可以引入自訂定義詞彙。讓我們列出一些定義詞彙和字典操作詞彙：

- create ⟨*word-name*⟩ (e –)，使用從輸入掃描的下一個詞彙作為名稱，使用 *WordDef* e 作為其定義來定義新的普通詞彙。如果該詞彙已存在，則會被默默地重新定義。
- (create) (e S x –)，建立一個名稱等於 *String* S 且定義等於 *WordDef* e 的新詞彙，使用在 *Integer* $0 \leq x \leq 3$ 中傳遞的旗標。如果在 x 中設定了位元 $+1$，則建立活躍詞彙；如果設定了位元 $+2$，則建立前綴詞彙。
- : ⟨*word-name*⟩ (e –)，使用 *WordDef* e 作為其定義，在字典中定義新的普通詞彙 ⟨*word-name*⟩。如果指定的詞彙已存在於字典中，則會被默默地重新定義。
- forget ⟨*word-name*⟩ (–)，忘記（從字典中移除）指定詞彙的定義。
- (forget) (S –)，忘記名稱在 *String* S 中指定的詞彙。如果找不到該詞彙，則拋出例外。
- :_ ⟨*word-name*⟩ (e –)，定義新的普通前綴詞彙 ⟨*word-name*⟩，這意味著 Fift 輸入解析器在詞彙名稱之後不需要空格或行尾字元。在所有其他方面，它與「:」類似。
- :: ⟨*word-name*⟩ (e –)，使用 *WordDef* e 作為其定義，在字典中定義新的活躍詞彙 ⟨*word-name*⟩。如果指定的詞彙已存在於字典中，則會被默默地重新定義。
- ::_ ⟨*word-name*⟩ (e –)，定義新的活躍前綴詞彙 ⟨*word-name*⟩，這意味著 Fift 輸入解析器在詞彙名稱之後不需要空格或行尾字元。在所有其他方面，它與「::」類似。
- constant ⟨*word-name*⟩ (x –)，定義新的普通詞彙 ⟨*word-name*⟩，當呼叫時會推入給定的值 x。
- 2constant ⟨*word-name*⟩ (x y –)，定義新的普通詞彙 ⟨*word-name*⟩，當呼叫時會推入給定的值 x 和 y（依此順序）。

- =: ⟨*word-name*⟩（x – ），定義新的普通詞彙 ⟨*word-name*⟩，當呼叫時會推入給定的值 x，與 constant 類似，但在區塊和冒號定義內有效。

- 2=: ⟨*word-name*⟩（x y – ），定義新的普通詞彙 ⟨*word-name*⟩，當呼叫時會推入給定的值 x 和 y（依此順序），與 2constant 類似，但在區塊和冒號定義內有效。

請注意，上述大多數詞彙都可以根據 (create) 來定義：

```
{ 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 字典查找

以下詞彙可用於在字典中查找詞彙：

- ' ⟨*word-name*⟩（– e），推入在編譯時恢復的詞彙 ⟨*word-name*⟩ 的定義。如果找不到指定的詞彙，則拋出例外。請注意，對於普通詞彙，' ⟨*word-name*⟩ execute 總是等同於⟨*word-name*⟩，但對於活躍詞彙則不然。

- nop（– ），不執行任何操作。

- 'nop（– e），推入 nop 的預設定義——一個執行時不執行任何操作的執行令牌。

- find（S – e -1 或 e 1 或 0），在字典中查找 *String* S，如果找到，則將其定義作為 *WordDef* e 回傳，後跟普通詞彙的 -1 或活躍詞彙的 1。否則推入 0。

- (') ⟨*word-name*⟩（– e），與' 類似，但在執行時回傳指定詞彙的定義，每次呼叫時執行字典查找。可用於通過使用短語 (') ⟨*word-name*⟩ execute 來恢復詞彙定義和其他區塊內常數的當前值。

- `@'` ⟨***word-name***⟩（– *e*），與 `(')` 類似，但在執行時恢復指定詞彙的定義，每次呼叫時執行字典查找，然後執行此定義。可用於通過使用短語 `@'` ⟨***word-name***⟩（等同於 `(')` ⟨***word-name***⟩ `execute`）來恢復詞彙定義和其他區塊內常數的當前值，參見 **2.7**。

- `[compile]` ⟨***word-name***⟩（– ），編譯 ⟨*word-name*⟩，就像它是普通詞彙一樣，即使它是活躍的。本質上等同於`'` ⟨***word-name***⟩ `execute`。

- `words`（– ），印出字典中當前定義的所有詞彙的名稱。

4.7 建立和操作詞彙串列

在 Fift 堆疊中，詞彙定義和字面值的參照串列由 *WordList* 型別的值表示，用作區塊或詞彙定義。一些用於操作 *WordList* 的詞彙包括：

- `{`（– *l*），一個活躍詞彙，將內部變數 `state` 加一，並將一個新的空 *WordList* 推入堆疊。

- `}`（*l* – *e*），一個活躍詞彙，將 *WordList l* 轉換為 *WordDef*（執行令牌）*e*，從而使對 *l* 的所有進一步修改變得不可能，並將內部變數 `state` 減一，推入整數 1，後跟`'nop`。其淨效果是將建構的 *WordList* 轉換為執行令牌，並將此執行令牌推入堆疊，無論是立即還是在外部區塊的執行期間。

- `({)`（– *l*），將一個空 *WordList* 推入堆疊。

- `(})`（*l* – *e*），將 *WordList* 轉換為執行令牌，使所有進一步修改變得不可能。

- `(compile)`（*l* x_1 …x_n *n e* – *l*′），擴展 *WordList l*，使其在呼叫時將 $0 \le n \le 255$ 個值 x_1、…、x_n 推入堆疊並執行執行令牌 *e*，其中 $0 \le n \le 255$ 是 *Integer*。如果 *e* 等於特殊值`'nop`，則省略最後一步。

- `does`（x_1 …x_n *n e* – *e*′），建立一個新的執行令牌 *e*′，它會將 *n* 個值 x_1、…、x_n 推入堆疊，然後執行 *e*。它大致等同於 `(compile)`、`(compile)` 和 `(})` 的組合。

4.8 自訂定義詞彙

詞彙 `does` 實際上是根據更簡單的詞彙定義的：

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

它對於定義自訂定義詞彙特別有用。例如，constant 和 2constant 可以藉助 does 和 create 來定義：

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

當然，通過這種自訂定義詞彙定義的詞彙可以執行非平凡的動作。例如，

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

將印出「hello unknown error ok」，因為 hello 是通過自訂定義詞彙 says 定義的，每當呼叫時會印出「hello」，類似地，error 在呼叫時印出「unknown error」。上述定義本質上等同於

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

然而，自訂定義詞彙在呼叫時可以執行更複雜的動作，並在編譯時預處理其引數。例如，可以以非平凡的方式計算訊息：

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

定義詞彙 hw，當呼叫時印出「Hello, world!」。此訊息的字串在編譯時（當 says 被呼叫時）計算一次，而不是在執行時（當 hw 被呼叫時）。

5 單元操作

到目前為止，我們已經討論了與 TVM 或 TON 區塊鏈無關的基本 Fift 基本運算。現在我們將轉向用於操作 *Cell* 的 TON 特定詞彙。

5.1 切片字面值

回想一下，（TVM）*Cell* 由最多 1023 個資料位元和最多四個對其他 *Cell* 的參照組成，*Slice* 是 *Cell* 一部分的唯讀視圖，*Builder* 用於建立新 *Cell*。Fift 對定義 *Slice* 字面值（即未命名常數）有特殊規定，如果需要，這些字面值也可以轉換為 *Cell*。

Slice 字面值通過活躍前綴詞彙 `x{` 和 `b{` 來引入：

- `b{`⟨***binary-data***⟩`}`（– *s*），建立一個 *Slice s*，它不包含參照，並包含最多 1023 個資料位元，這些位元在 ⟨*binary-data*⟩ 中指定，⟨*binary-data*⟩ 必須是僅由字元「`0`」和「`1`」組成的字串。

- `x{`⟨***hex-data***⟩`}` （– *s*），建立一個 *Slice s*，它不包含參照，並包含最多 1023 個資料位元，這些位元在 ⟨*hex-data*⟩ 中指定。更確切地說，⟨*hex-data*⟩ 中的每個十六進位數字以通常的方式轉換為四個二進位數字。之後，如果 ⟨*hex-data*⟩ 的最後一個字元是底線 `_`，則從結果二進位字串中移除所有尾隨二進位零和緊接在它們之前的二進位一（有關更多詳細資訊，請參見 [4, 1.0]）。

這樣，`b{00011101}` 和 `x{1d}` 都推入由八個資料位元和零個參照組成的相同 *Slice*。類似地，`b{111010}` 和 `x{EA_}` 推入由六個資料位元組成的相同 *Slice*。空 *Slice* 可以表示為 `b{}` 或 `x{}`。

如果想要定義帶有一些 *Cell* 參照的常數 *Slice*，可以使用以下詞彙：

- `|_` （*s s′* – *s″*），給定兩個 *Slice s* 和 *s′*，建立一個新的 *Slice s″*，它從 *s* 獲得，方法是附加一個對包含 *s′* 的 *Cell* 的新參照。

- `|+` （*s s′* – *s″*），連接兩個 *Slice s* 和 *s′*。這意味著新 *Slice s″* 的資料位元通過連接 *s* 和 *s′* 的資料位元獲得，*s″* 的 *Cell* 參照串列類似地通過連接 *s* 和 *s′* 的對應串列來建構。

5.2 建構器基本運算

以下詞彙可用於操作 *Builder*，稍後可用於建構新 *Cell*：

- `<b` （– *b*），建立一個新的空 *Builder*。

- `b>` （*b* – *c*），將 *Builder b* 轉換為包含與 *b* 相同資料的新 *Cell c*。

- `i,` （*b x y* – *b′*），將有號 *y* 位元整數 *x* 的大端序二進位表示附加到 *Builder b*，其中 $0 \leq y \leq 257$。如果 *b* 中沒有足夠的空間（即，如果 *b*

已包含超過 $1023 - y$ 個資料位元），或者如果 *Integer* x 不適合 y 位元，則拋出例外。

- `u,` （$b\ x\ y - b'$），將無號 y 位元整數 x 的大端序二進位表示附加到 *Builder* b，其中 $0 \leq y \leq 256$。如果操作不可能，則拋出例外。
- `ref,` （$b\ c - b'$），將對 *Cell* c 的參照附加到 *Builder* b。如果 b 已包含四個參照，則拋出例外。
- `s,` （$b\ s - b'$），將從 *Slice* s 取得的資料位元和參照附加到 *Builder* b。
- `sr,` （$b\ s - b'$），建構一個包含來自 *Slice* s 的所有資料和參照的新 *Cell*，並將對此單元的參照附加到 *Builder* b。等同於 `<b swap s, b> ref,`。
- `$,` （$b\ S - b'$），將 *String* S 附加到 *Builder* b。字串被解釋為長度為 $8n$ 的二進位字串，其中 n 是 S 的 UTF-8 表示中的位元組數。
- `B,` （$b\ B - b'$），將 *Bytes* B 附加到 *Builder* b。
- `b+` （$b\ b' - b''$），連接兩個 *Builder* b 和 b'。
- `bbits` （$b - x$），回傳已儲存在 *Builder* b 中的資料位元數。結果 x 是範圍 $0 \ldots 1023$ 中的 *Integer*。
- `brefs` （$b - x$），回傳已儲存在 *Builder* b 中的參照數。結果 x 是範圍 $0 \ldots 4$ 中的 *Integer*。
- `bbitrefs` （$b - x\ y$），回傳已儲存在 *Builder* b 中的資料位元數 x 和參照數 y。
- `brembits` （$b - x$），回傳可在 *Builder* b 中儲存的額外資料位元的最大數量。等同於 `bbits 1023 swap -`。
- `bremrefs` （$b - x$），回傳可在 *Builder* b 中儲存的額外單元參照的最大數量。
- `brembitrefs` （$b - x\ y$），回傳可在 *Builder* b 中儲存的額外資料位元的最大數量 $0 \leq x \leq 1023$ 和額外單元參照的最大數量 $0 \leq y \leq 4$。

結果 *Builder* 可以通過非破壞性堆疊傾印基本運算`.s` 或短語 `b> <s csr.` 進行檢查。例如：

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

輸出

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

可以觀察到.s 傾印 *Builder* 的內部表示，開頭有兩個標籤位元組（通常等於已儲存在 *Builder* 中的單元參照數，以及儲存在 *Builder* 中的完整位元組數的兩倍，如果存在不完整位元組則加一）。另一方面，csr. 以類似於 x{ 定義 *Slice* 字面值所使用的形式傾印 *Slice*（由 <s 從 *Cell* 建構，參見 **5.3**）（參見 **5.1**）。

順便一提，上面顯示的詞彙 mkTest（其定義中沒有.s）對應於 TL-B 建構器

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

並可用於序列化此 TL-B 型別的值。

5.3 切片基本運算

以下詞彙可用於操作 *Slice* 型別的值，它表示 *Cell* 一部分的唯讀視圖。這樣，先前儲存到 *Cell* 中的資料可以被反序列化，方法是首先將 *Cell* 轉換為 *Slice*，然後從此 *Slice* 逐步提取所需的資料。

- <s（$c - s$），將 *Cell* c 轉換為包含相同資料的 *Slice* s。它通常標記單元反序列化的開始。
- s>（$s -$ ），如果 *Slice* s 非空，則拋出例外。它通常標記單元反序列化的結束，檢查是否還有未處理的資料位元或參照。
- i@（$s\ x - y$），從 *Slice* s 的前 x 個位元中提取有號大端序 x 位元整數。如果 s 包含少於 x 個資料位元，則拋出例外。
- i@+（$s\ x - y\ s'$），類似於 i@ 從 *Slice* s 的前 x 個位元提取有號大端序 x 位元整數，但也回傳 s 的剩餘部分。

- `i@?`（$s\ x - y\ -1$ 或 0），類似於 `i@` 從 *Slice* 提取有號整數，但在成功時之後推入整數 -1。如果 s 中剩餘的位元少於 x 個，則推入整數 0 表示失敗。

- `i@?+`（$s\ x - y\ s'\ -1$ 或 $s\ 0$），從 *Slice* s 提取有號整數並計算此 *Slice* 的剩餘部分，類似於 `i@+`，但之後推入 -1 表示成功。失敗時，推入未改變的 *Slice* s 和 0 表示失敗。

- `u@`、`u@+`、`u@?`、`u@?+`，`i@`、`i@+`、`i@?`、`i@?+` 的對應版本，用於反序列化無號整數。

- `B@`（$s\ x - B$），從 *Slice* s 提取前 x 個位元組（即 $8x$ 個位元），並將它們作為 *Bytes* 值 B 回傳。如果 s 中沒有足夠的資料位元，則拋出例外。

- `B@+`（$s\ x - B\ s'$），類似於 `B@`，但也回傳 *Slice* s 的剩餘部分。

- `B@?`（$s\ x - B\ -1$ 或 0），類似於 `B@`，但使用旗標表示失敗，而不是拋出例外。

- `B@?+`（$s\ x - B\ s'\ -1$ 或 $s\ 0$），類似於 `B@+`，但使用旗標表示失敗，而不是拋出例外。

- `$@`、`$@+`、`$@?`、`$@?+`，`B@`、`B@+`、`B@?`、`B@?+` 的對應版本，將結果作為（UTF-8）*String* 而不是 *Bytes* 值回傳。這些基本運算不檢查讀取的位元組序列是否為有效的 UTF-8 字串。

- `ref@`（$s - c$），從 *Slice* s 提取第一個參照，並回傳所參照的 *Cell* c。如果沒有剩餘參照，則拋出例外。

- `ref@+`（$s - s'\ c$），類似於 `ref@`，但也回傳 s 的剩餘部分。

- `ref@?`（$s - c\ -1$ 或 0），類似於 `ref@`，但使用旗標表示失敗，而不是拋出例外。

- `ref@?+`（$s - s'\ c\ -1$ 或 $s\ 0$），類似於 `ref@+`，但使用旗標表示失敗，而不是拋出例外。

- `empty?`（$s - ?$），檢查 *Slice* 是否為空（即沒有剩餘資料位元和參照），並相應地回傳 -1 或 0。

- `remaining`（$s - x\ y$），回傳 *Slice* s 中剩餘的資料位元數 x 和單元參照數 y。
- `sbits`（$s - x$），回傳 *Slice* s 中剩餘的資料位元數 x。
- `srefs`（$s - x$），回傳 *Slice* s 中剩餘的單元參照數 x。
- `sbitrefs`（$s - x\ y$），回傳 *Slice* s 中剩餘的資料位元數 x 和單元參照數 y。等同於 `remaining`。
- `$>s`（$S - s$），將 *String* S 轉換為 *Slice*。等同於 `<b swap $, b> <s`。
- `s>c`（$s - c$），直接從 *Slice* s 建立 *Cell* c。等同於 `<b swap s, b>`。
- `csr.`（$s -$ ），遞迴印出 *Slice* s。在第一行，s 的資料位元以十六進位形式顯示，嵌入在類似於用於 *Slice* 字面值的 `x{…}` 構造中（參見 **5.1**）。在接下來的行中，s 參照的單元以更大的縮排印出。

例如， **5.2** 中討論的 TL-B 型別 `Test` 的值

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

可以如下反序列化：

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

如預期印出「`17239 -1000000001 ok`」。

當然，如果經常需要檢查建構器標籤，可以為此目的定義一個輔助詞彙：

```
{ 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 . .
```

藉助活躍前綴詞彙（參見 **4.2** 和 **4.4**），我們可以做得更好：

```
{ 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 . .
```

一個更短但效率較低的解決方案是重用先前定義的 `tag?`：

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

首先輸出「`x{F55AA}`」，然後拋出訊息為「constructor tag mismatch」的例外。

5.4 單元雜湊運算

很少有直接對 *Cell* 進行操作的詞彙。其中最重要的計算給定單元的（基於 *SHA256* 的）表示雜湊值（參見 [4, 3.1]），可以粗略地描述為單元的資料位元與此單元參照的單元的遞迴計算雜湊值連接後的 SHA256 雜湊：

- `hashB`（$c - B$），計算 *Cell* c 的基於 SHA256 的表示雜湊值（參見 [4, 3.1]），它明確定義了 c 及其所有後代（前提是 SHA256 沒有碰撞）。結果作為恰好由 32 個位元組組成的 *Bytes* 值回傳。
- `hashu`（$c - x$），如上計算 c 的基於 SHA256 的表示雜湊值，但將結果作為大端序無號 256 位元 *Integer* 回傳。
- `shash`（$s - B$），通過首先將 *Slice* 轉換為單元來計算其基於 SHA256 的表示雜湊值。等同於 `s>c hashB`。

5.5 單元集合運算

單元集合（bag of cells）是一個或多個單元及其所有後代的集合。它通常可以序列化為位元組序列（在 Fift 中由 *Bytes* 值表示），然後儲存到檔案或通過網路傳輸。之後，它可以被反序列化以恢復原始單元。TON 區塊鏈系統

地根據某個 TL-B 方案將不同的資料結構（包括 TON 區塊鏈區塊）表示為單元樹（參見 [5]，其中詳細說明了此方案），然後這些單元樹被例行匯入單元集合並序列化為二進位檔案。

用於操作單元集合的 Fift 詞彙包括：

- `B>boc`（$B - c$），反序列化由 *Bytes* B 表示的「標準」單元集合（即恰好有一個根單元的單元集合），並回傳根 *Cell* c。
- `boc+>B`（$c\ x - B$），建立並序列化一個「標準」單元集合，包含一個根 *Cell* c 及其所有後代。*Integer* 參數 $0 \le x \le 31$ 用於傳遞指示單元集合序列化額外選項的旗標，各個位元具有以下效果：
  - $+1$ 啟用單元集合索引建立（對於大型單元集合的延遲反序列化有用）。
  - $+2$ 在序列化中包含所有資料的 CRC32-C（對於檢查資料完整性有用）。
  - $+4$ 將根單元的雜湊值明確儲存到序列化中（以便之後可以快速恢復而無需完整反序列化）。
  - $+8$ 儲存一些中間（非葉）單元的雜湊值（對於大型單元集合的延遲反序列化有用）。
  - $+16$ 儲存單元快取位元以控制反序列化單元的快取。

  x 的典型值對於非常小的單元集合（例如，TON 區塊鏈外部訊息）是 $x = 0$ 或 $x = 2$，對於大型單元集合（例如，TON 區塊鏈區塊）是 $x = 31$。
- `boc>B`（$c - B$），序列化帶有根 *Cell* c 及其所有後代的小型「標準」單元集合。等同於 `0 boc+>B`。

例如，在 **5.2** 中建立的具有 TL-B `Test` 型別值的單元可以如下序列化：

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

輸出「`B5EE9C7201040101000000000900000E4A4357C46535FF ok`」。這裡 `Bx.` 是印出 *Bytes* 值的十六進位表示的詞彙。

5.6 二進位檔案 I/O 和位元組操作

以下詞彙可用於操作 *Bytes* 型別的值（任意位元組序列），並從二進位檔案讀取它們或將它們寫入二進位檔案：

- `B{`⟨*hex-digits*⟩`}`（– *B*），推入包含由偶數個十六進位數字表示的資料的 *Bytes* 字面值。
- `Bx.`（*B* –），印出 *Bytes* 值的十六進位表示。每個位元組由恰好兩個大寫十六進位數字表示。
- `file>B`（*S* – *B*），讀取名稱在 *String S* 中指定的（二進位）檔案，並將其內容作為 *Bytes* 值回傳。如果檔案不存在，則拋出例外。
- `B>file`（*B S* –），建立一個名稱在 *String S* 中指定的新（二進位）檔案，並將來自 *Bytes B* 的資料寫入新檔案。如果指定的檔案已存在，則會被覆寫。
- `file-exists?`（*S* – ?），檢查名稱在 *String S* 中指定的檔案是否存在。

例如，在 **5.5** 的例子中建立的單元集合可以如下儲存到磁碟為 `sample.boc`：

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

之後可以通過 `file>B` 和 `B>boc` 載入並反序列化（即使在另一個 Fift 會話中）：

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

印出「`17239 -1000000001 ok`」。

此外，還有幾個詞彙用於直接將資料打包（序列化）到 *Bytes* 值中，並在之後解包（反序列化）它們。它們可以與 `B>file` 和 `file>B` 結合使用，直接將資料儲存到二進位檔案中，並在之後載入它們。

- `Blen`（*B* – *x*），回傳 *Bytes* 值 *B* 的長度（以位元組為單位）。
- `BhashB`（*B* – *B*′），計算 *Bytes* 值的 SHA256 雜湊。雜湊作為 32 位元組 *Bytes* 值回傳。

- `Bhashu`（$B - x$），計算 *Bytes* 值的 SHA256 雜湊，並將雜湊作為無號 256 位元大端序整數回傳。
- `B=`（$B\ B' - ?$），檢查兩個 *Bytes* 序列是否相等。
- `Bcmp`（$B\ B' - x$），按字典順序比較兩個 *Bytes* 序列，並根據比較結果回傳 -1、0 或 1。
- `B>i@`（$B\ x - y$），將 *Bytes* 值 B 的前 $x/8$ 個位元組反序列化為有號大端序 x 位元 *Integer* y。
- `B>i@+`（$B\ x - B'\ y$），類似於 `B>i@` 將 B 的前 $x/8$ 個位元組反序列化為有號大端序 x 位元 *Integer* y，但也回傳 B 的剩餘位元組。
- `B>u@`、`B>u@+`，反序列化無號整數的 `B>i@` 和 `B>i@+` 變體。
- `B>Li@`、`B>Li@+`、`B>Lu@`、`B>Lu@+`，`B>i@`、`B>i@+`、`B>u@`、`B>u@+` 的小端序變體。
- `B|`（$B\ x - B'\ B''$），從 *Bytes* 值 B 中切割前 x 個位元組，並將前 x 個位元組（B'）和剩餘部分（B''）都作為新的 *Bytes* 值回傳。
- `i>B`（$x\ y - B$），將有號大端序 y 位元 *Integer* x 儲存到恰好由 $y/8$ 個位元組組成的 *Bytes* 值 B 中。整數 y 必須是範圍 $0 \ldots 256$ 中的八的倍數。
- `u>B`（$x\ y - B$），類似於 `i>B`，將無號大端序 y 位元 *Integer* x 儲存到恰好由 $y/8$ 個位元組組成的 *Bytes* 值 B 中。
- `Li>B`、`Lu>B`，`i>B` 和 `u>B` 的小端序變體。
- `B+`（$B'\ B'' - B$），連接兩個 *Bytes* 序列。

6 TON 特定運算

本章描述 TON 特定的 Fift 詞彙，但不包括前一章已討論的用於 *Cell* 操作的詞彙。

6.1 Ed25519 密碼學

Fift 提供了與 TVM 使用的相同 Ed25519 橢圓曲線密碼學的介面，在 [5] 的附錄 A 中描述：

- `now`（– x），將當前 Unixtime 作為 *Integer* 回傳。
- `newkeypair`（– B B'），生成新的 Ed25519 私鑰/公鑰對，並將私鑰 B 和公鑰 B' 都作為 32 位元組 *Bytes* 值回傳。金鑰的品質足以用於測試目的。實際應用程式必須在生成 Ed25519 金鑰對之前向 OpenSSL PRNG 提供足夠的熵。
- `priv>pub`（B – B'），計算與私有 Ed25519 金鑰對應的公鑰。公鑰 B' 和私鑰 B 都由 32 位元組 *Bytes* 值表示。
- `ed25519_sign`（B B' – B''），使用 Ed25519 私鑰 B'（32 位元組 *Bytes* 值）簽署資料 B，並將簽章作為 64 位元組 *Bytes* 值 B'' 回傳。
- `ed25519_sign_uint`（x B' – B''），將大端序無號 256 位元整數 x 轉換為 32 位元組序列，並使用 Ed25519 私鑰 B' 對其進行簽署，類似於 `ed25519_sign`。等同於 `swap 256 u>B swap ed25519_sign`。要簽署的整數 x 通常計算為某些資料的雜湊值。
- `ed25519_chksign`（B B' B'' – ?），檢查 B' 是否為使用公鑰 B'' 對資料 B 的有效 Ed25519 簽章。

6.2 智慧合約地址解析器

兩個特殊詞彙可用於解析人類可讀（base64 或 base64url）形式的 TON 智慧合約地址：

- `smca>$`（x y z – S），根據旗標 z 將帶有工作鏈 x（有號 32 位元 *Integer*）和工作鏈內地址 y（無號 256 位元 *Integer*）的標準 TON 智慧合約地址打包到 48 字元字串 S（地址的人類可讀表示）中。z 中的可能單獨旗標為：+1 表示不可反彈地址，+2 表示僅測試網地址，+4 表示 base64url 輸出而不是 base64。
- `$>smca`（S – x y z -1 或 0），從其人類可讀字串表示 S 解包標準 TON 智慧合約地址。成功時，回傳有號 32 位元工作鏈 x、無號 256 位元工作鏈內地址 y、旗標 z（其中 +1 表示地址不可反彈，+2 表示地址僅限測試網）和 -1。失敗時，推入 0。

人類可讀的智慧合約地址範例可以如下反序列化和顯示：

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

輸出「`-1 538fa7…0f7d 0`」，意味著指定的地址在工作鏈 -1（TON 區塊鏈的主鏈）中，並且工作鏈 -1 內的 256 位元地址是 0x538...f7d。

6.3 字典操作

Fift 有幾個用於雜湊映射或（*TVM*）字典操作的詞彙，對應於 [4, 3.3] 中描述的 TL-B 型別 `HashmapE` n X 的值。這些（TVM）字典不要與 Fift 字典混淆，後者是完全不同的東西。TL-B 型別 `HashmapE` n X 的字典本質上是一個鍵值集合，具有不同的 n 位元金鑰（其中 $0 \leq n \leq 1023$）和任意 TL-B 型別 X 的值。字典由單元樹表示（完整佈局可在 [4, 3.3] 中找到），並作為 *Cell* 或 *Slice* 型別的值儲存在 Fift 堆疊中。有時空字典由 *Null* 值表示。

- `dictnew`（– D），推入表示新空字典的 *Null* 值。
- `idict!`（v x D n – D' -1 或 D 0），將由有號大端序 n 位元整數 x 給定金鑰的新值 v（由 *Slice* 表示）新增到具有 n 位元金鑰的字典 D（由 *Cell* 或 *Null* 表示）中，成功時回傳新字典 D' 和 -1。否則回傳未改變的字典 D 和 0。
- `idict!+`（v x D n – D' -1 或 D 0），類似於 `idict!` 將新鍵值對 (x, v) 新增到字典 D 中，但如果金鑰已存在則失敗，回傳未改變的字典 D 和 0。
- `b>idict!`、`b>idict!+`，`idict!` 和 `idict!+` 的變體，接受 *Builder* 而不是 *Slice* 中的新值 v。
- `udict!`、`udict!+`、`b>udict!`、`b>udict!+`，`idict!`、`idict!+`、`b>idict!`、`b>idict!+` 的變體，但使用無號 n 位元整數 x 作為金鑰。
- `sdict!`、`sdict!+`、`b>sdict!`、`b>sdict!+`，`idict!`、`idict!+`、`b>idict!`、`b>idict!+` 的變體，但使用 *Slice* x 的前 n 個資料位元作為金鑰。
- `idict@`（x D n – v -1 或 0），在由 *Cell* D 表示的字典中查找由有號大端序 n 位元 *Integer* x 表示的金鑰。如果找到金鑰，則將對應值作為 *Slice* v 和 -1 回傳。否則回傳 0。

- `idict@-`（x D n – D' v -1 或 D 0），在由 *Cell* D 表示的字典中查找由有號大端序 n 位元 *Integer* x 表示的金鑰。如果找到金鑰，則從字典中刪除它，並回傳修改後的字典 D'、對應值作為 *Slice* v 和 -1。否則回傳未修改的字典 D 和 0。
- `idict-`（x D n – D' -1 或 D 0），類似於 `idict@-` 從字典 D 中刪除整數金鑰 x，但不回傳舊字典 D 中與 x 對應的值。
- `udict@`、`udict@-`、`udict-`，`idict@`、`idict@-`、`idict-` 的變體，但使用無號大端序 n 位元 *Integer* x 作為金鑰。
- `sdict@`、`sdict@-`、`sdict-`，`idict@`、`idict@-`、`idict-` 的變體，但金鑰由 *Slice* k 的前 n 個位元提供。
- `dictmap`（D n e – s'），將執行令牌 e（即匿名函數）應用於儲存在具有 n 位元金鑰的字典 D 中的每個鍵值對。執行令牌對每個鍵值對執行一次，在執行 e 之前將 *Builder* b 和 *Slice* v（包含值）推入堆疊。執行後 e 必須在堆疊中留下修改後的 *Builder* b'（包含來自 b 的所有資料以及新值 v'）和 -1，或 0 表示失敗。在後一種情況下，對應的金鑰從新字典中省略。
- `dictmerge`（D D' n e – D''），將兩個具有 n 位元金鑰的字典 D 和 D' 組合成一個具有相同金鑰的字典 D''。如果金鑰僅存在於字典 D 和 D' 中的一個，則此金鑰和對應值逐字複製到新字典 D''。否則，呼叫執行令牌（匿名函數）e 來合併分別對應於 D 和 D' 中相同金鑰 k 的兩個值 v 和 v'。在呼叫 e 之前，推入 *Builder* b 和兩個表示要合併的兩個值的 *Slice* v 和 v'。執行後 e 留下修改後的 *Builder* b'（包含來自 b 的原始資料以及組合值）和 -1，或 0 表示失敗。在後一種情況下，對應的金鑰從新字典中省略。

Fift 還為前綴字典提供了一些支援：

- `pfxdict!`（v k s n – s' -1 或 s 0），將鍵值對 (k, v)（都由 *Slice* 表示）新增到金鑰長度最多為 n 的前綴字典 s 中。成功時，回傳修改後的字典 s' 和 -1。失敗時，回傳原始字典 s 和 0。
- `pfxdict!+`（v k s n – s' -1 或 s 0），類似於 `pfxdict!` 將鍵值對 (k, v) 新增到前綴字典 s 中，但如果金鑰已存在則失敗。
- `pfxdict@`（k s n – v -1 或 0），在金鑰長度限制為 n 位元的前綴字典 s 中查找金鑰 k（由 *Slice* 表示）。成功時，回傳找到的值 v 和 -1。失敗時，回傳 0。

6.4 從 Fift 呼叫 TVM

TVM 可以與 Fift 解譯器連結。在這種情況下，有幾個 Fift 基本運算可用，可用於使用從 Fift 提供的引數呼叫 TVM。引數可以在 Fift 堆疊中準備，該堆疊完整地傳遞給 TVM 的新實例。結果堆疊和退出代碼傳回 Fift，之後可以檢查。

- `runvmcode`（$\ldots s - \ldots x$），使用從 *Slice* s 初始化的當前延續 `cc` 呼叫 TVM 的新實例，從而在 TVM 中執行程式碼 s。原始 Fift 堆疊（不包含 s）完整地作為 TVM 的初始堆疊傳遞。當 TVM 終止時，其結果堆疊用作新的 Fift 堆疊，退出代碼 x 推入其頂部。如果 x 非零，表示 TVM 已被未處理的例外終止，則從頂部開始的下一個堆疊項目包含此例外的參數，x 是例外代碼。在這種情況下，所有其他項目都從堆疊中移除。

- `runvmdict`（$\ldots s - \ldots x$），類似於 `runvmcode` 使用從 *Slice* s 初始化的當前延續 `cc` 呼叫 TVM 的新實例，但也使用相同的值初始化特殊暫存器 `c3`，並在 TVM 執行開始之前將零推入初始 TVM 堆疊。在典型應用中，*Slice* s 由子例程選擇程式碼組成，該程式碼使用堆疊頂部 *Integer* 來選擇要執行的子例程，從而啟用幾個相互遞迴子例程的定義和執行（參見 [4, 4.6] 和 **7.8**）。等於零的選擇器對應於大型 TVM 程式中的 `main()` 子例程。

- `runvm`（$\ldots s\ c - \ldots x\ c'$），類似於 `runvmdict`（不將額外的零推入初始 TVM 堆疊；如有必要，可以在 s 下明確推入），使用從 *Slice* s 初始化的當前延續 `cc` 和特殊暫存器 `c3` 呼叫 TVM 的新實例，並且還使用 *Cell* c 初始化特殊暫存器 `c4`（「持久性資料的根」，參見 [4, 1.4]）。`c4` 的最終值在最終 Fift 堆疊的頂部作為另一個 *Cell* c' 回傳。這樣可以模擬檢查或修改其持久性儲存的智慧合約的執行。

- `runvmctx`（$\ldots s\ c\ t - \ldots x\ c'$），`runvm` 的變體，也使用 *Tuple* t 初始化 `c7`（「上下文」）。這樣，如果將正確的上下文載入到 `c7` 中，可以完全模擬 TON 區塊鏈內 TVM 智慧合約的執行（參見 [5, 4.4.10]）。

- `gasrunvmcode`（$\ldots s\ z - \ldots x\ z'$），`runvmcode` 的 gas 感知版本，在堆疊頂部接受額外的 *Integer* 引數 z（原始 gas 限制），並將此 TVM 執行消耗的 gas 作為新的堆疊頂部 *Integer* 值 z' 回傳。

- `gasrunvmdict`（$\ldots s\ z - \ldots x\ z'$），`runvmdict` 的 gas 感知版本。

- `gasrunvm`（$\ldots s\ c\ z - \ldots x\ c'\ z'$），`runvm` 的 gas 感知版本。

- `gasrunvmctx` ($...s\ c\ t\ z - ...x\ c'\ z'$)，`runvmctx` 的 gas 感知版本。

例如，可以建立執行一些簡單程式碼的 TVM 實例，如下所示：

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

TVM 堆疊由三個整數 2、3 和 9（依此順序；9 是最頂端項目）初始化，然後包含 16 個資料位元且無參照的 *Slice* `x{1221}` 轉換為 TVM 延續並執行。通過查閱 [4] 的附錄 A，我們看到 `x{12}` 是 TVM 指令 `XCHG s1, s2` 的程式碼，`x{21}` 是 TVM 指令 `OVER` 的程式碼（不要與 Fift 基本運算 `over` 混淆，順便說一下，它對堆疊有相同的效果）。上述執行的結果是：

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

這裡 0 是退出代碼（表示 TVM 成功終止），`3 2 9 2` 是最終 TVM 堆疊狀態。

如果在 TVM 執行期間生成未處理的例外，則此例外的代碼作為退出代碼回傳：

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

產生

```
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
```

請注意，TVM 在啟用內部日誌記錄的情況下執行，其日誌顯示在標準輸出中。

簡單的 TVM 程式可以藉助 `x{...}` 構造用 *Slice* 字面值表示，類似於上述範例。更複雜的程式通常藉助下一章說明的 Fift 組譯器建立。

7 使用 Fift 組譯器

Fift 組譯器是一個完全用 Fift 編寫的短程式（目前不到 30KiB），它將 TVM 指令的人類可讀助憶符號轉換為其二進位表示。例如，可以在 **6.4** 中討論的範例中寫 `<{ s1 s2 XCHG OVER }>s` 而不是 `x{1221}`，前提是事先載入了 Fift 組譯器（通常通過短語`"Asm.fif" include`）。

7.1 載入 Fift 組譯器

Fift 組譯器通常位於 Fift 函式庫目錄（通常包含標準 Fift 函式庫檔案，如 `Fift.fif`）中的檔案 `Asm.fif` 中。它通常通過在需要使用 Fift 組譯器的程式的最開始放置短語`"Asm.fif" include` 來載入：

- `include`（*S* – ），從由 *String S* 給定的路徑載入並解譯 Fift 來源檔案。如果檔名 *S* 不以斜線開頭，則使用 Fift 包含搜尋路徑（通常取自 `FIFTPATH` 環境變數或 Fift 解譯器的 `-I` 命令列引數，如果兩者都不存在則等於 `/usr/lib/fift`）來定位 *S*。

Fift 組譯器的當前實現大量使用自訂定義詞彙（參見 **4.8**）；它的原始碼可以作為如何使用定義詞彙編寫非常緊湊的 Fift 程式的良好範例（另請參見 [1] 的原始版本，其中討論了一個簡單的 8080 Forth 組譯器）。

將來，Fift 組譯器定義的幾乎所有詞彙都將移到單獨的詞彙表（命名空間）中。目前它們在全域命名空間中定義，因為 Fift 尚不支援命名空間。

7.2 Fift 組譯器基礎

Fift 組譯器從 Fift 繼承了其後綴運算表示法，即引數或參數寫在相應指令之前。例如，在 [4] 中表示為 `XCHG s1,s2` 的 TVM 組譯器指令在 Fift 組譯器中表示為 `s1 s2 XCHG`。

Fift 組譯器程式碼通常由特殊的開啟詞彙（如 `<{`）開啟，並由關閉詞彙（如`}>` 或`}>s`）終止。例如，

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

編譯兩個 TVM 指令 `XCHG s1,s2` 和 `OVER`，並將結果作為 *Slice* 回傳（因為使用了`}>s`）。結果 *Slice* 由 `csr.` 顯示，產生

```
x{1221}
```

可以使用 [4] 的附錄 A 並驗證 `x{12}` 確實是 TVM 指令 `XCHG s1,s2` 的（代碼頁零）程式碼，`x{21}` 是 TVM 指令 `OVER` 的程式碼（不要與 Fift 基本運算 `over` 混淆）。

將來，我們將假設 Fift 組譯器已載入，並從範例中省略短語`"Asm.fif" include`。

Fift 組譯器以直接的方式使用 Fift 堆疊，使用頂部的幾個堆疊項目來保存帶有正在組譯的程式碼的 *Builder*，以及 TVM 指令的引數。例如：

- `<{` ($- b$)，通過將空 *Builder* 推入 Fift 堆疊（並可能將命名空間切換到包含所有 Fift 組譯器特定詞彙的命名空間）來開始 Fift 組譯器程式碼的一部分。大致等同於 `<b`。
- `}>` ($b - b'$)，終止 Fift 組譯器程式碼的一部分，並將組譯部分作為 *Builder* 回傳（並可能恢復原始命名空間）。在大多數情況下大致等同於 `nop`。
- `}>c` ($b - c$)，終止 Fift 組譯器程式碼的一部分，並將組譯部分作為 *Cell* 回傳（並可能恢復原始命名空間）。大致等同於 `b>`。
- `}>s` ($b - s$)，類似於`}>` 終止 Fift 組譯器程式碼的一部分，但將組譯部分作為 *Slice* 回傳。等同於`}>c <s`。
- `OVER` ($b - b'$)，通過將其附加到堆疊頂部的 *Builder* 來組譯 TVM 指令 `OVER` 的程式碼。大致等同於 `x{21} s,`。
- `s1` ($- s$)，推入 Fift 組譯器用來表示 TVM 的「堆疊暫存器」`s1` 的特殊 *Slice*。
- `s0…s15` ($- s$)，與 `s1` 類似的詞彙，但推入表示 TVM 其他「堆疊暫存器」的 *Slice*。請注意，`s16…s255` 必須使用詞彙 `s()` 存取。
- `s()` ($x - s$)，取 *Integer* 引數 $0 \leq x \leq 255$，並回傳 Fift 組譯器用來表示「堆疊暫存器」$\texttt{s}(x)$ 的特殊 *Slice*。
- `XCHG` ($b\ s\ s' - b'$)，從堆疊中取兩個表示兩個「堆疊暫存器」$\texttt{s}(i)$ 和 $\texttt{s}(j)$ 的特殊 *Slice*，並將 TVM 指令 `XCHG` $\texttt{s}(i)$,$\texttt{s}(j)$ 的程式碼附加到 *Builder* b。

特別要注意，Fift 組譯器定義的詞彙 `OVER` 與 Fift 基本運算 `over` 的效果完全不同。

`OVER` 和其他 Fift 組譯器詞彙的實際動作比 `x{21} s,` 的動作更複雜。如果新指令程式碼不適合 *Builder* b（即，如果在新增新指令程式碼後 b 將包含超過 1023 個資料位元），則此指令和所有後續指令被組譯到新的 *Builder* $\tilde{b}$ 中，並且一旦 $\tilde{b}$ 的生成完成，舊 *Builder* b 通過對從 $\tilde{b}$ 獲得的 *Cell* 的參照來增強。這樣，長段的 TVM 程式碼被自動分割成有效 *Cell* 的鏈，每個最多包含 1023 個位元。因為 TVM 將延續末尾的單獨單元參照解釋為隱式 `JMPREF`，所以將 TVM 程式碼分割成單元對執行幾乎沒有影響。

7.3 推入整數常數

TVM 指令 PUSHINT x（呼叫時推入 *Integer* 常數 x）可以藉助 Fift 組譯器詞彙 INT 或 PUSHINT 組譯：

- PUSHINT（$b\ x - b'$），將 TVM 指令 PUSHINT x 組譯到 *Builder* 中。
- INT（$b\ x - b'$），等同於 PUSHINT。

請注意，PUSHINT 的引數是從 Fift 堆疊中取得的 *Integer* 值，不一定是字面值。例如，<{ 239 17 * INT }>s 是組譯 PUSHINT 4063 指令的有效方式，因為 $239 \cdot 17 = 4063$。請注意，乘法由 Fift 在組譯時執行，而不是在 TVM 執行時執行。後者的計算可以通過 <{ 239 INT 17 INT MUL }>s 執行：

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

產生

```
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
```

請注意，Fift 組譯器根據其引數 x 選擇 PUSHINT x 指令的最短編碼。

7.4 立即引數

某些 TVM 指令（如 PUSHINT）接受立即引數。這些引數通常在 Fift 堆疊中傳遞給組譯相應指令的 Fift 詞彙。整數立即引數通常由 *Integer* 表示，單元由 *Cell* 表示，延續由 *Builder* 和 *Cell* 表示，單元切片由 *Slice* 表示。例如，17 ADDCONST 組譯 TVM 指令 ADDCONST 17，x{ABCD_} PUSHSLICE 組譯 PUSHSLICE xABCD_：

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

產生

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

在某些情況下，Fift 組譯器假裝能夠接受超出相應 TVM 指令範圍的立即引數。例如，ADDCONST x 僅為 $-128 \leq x < 128$ 定義，但 Fift 組譯器接受 239 ADDCONST：

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

產生

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

我們可以看到「ADDCONST 239」已被默默替換為 PUSHINT 239 和 ADD。當 ADDCONST 的立即引數本身是 Fift 計算的結果，並且很難估計它是否總是適合所需範圍時，此功能很方便。

在某些情況下，同一 TVM 指令有幾個版本，一個接受立即引數，另一個沒有任何引數。例如，有 LSHIFT n 和 LSHIFT 指令。在 Fift 組譯器中，這些變體被分配了不同的助憶符號。特別是，LSHIFT n 由n LSHIFT# 表示，LSHIFT 由其本身表示。

7.5 立即延續

當立即引數是延續時，通過巢狀的 <{ ...}> 構造在 Fift 堆疊中建立相應的 *Builder* 很方便。例如，TVM 組譯器指令

```
PUSHINT 1
SWAP
PUSHCONT {
```

```
    MULCONST 10
}
REPEAT
```

可以通過以下方式組譯和執行

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

產生

```
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
```

存在更方便的方式來使用通過 Fift 組譯器建立的字面值延續。例如，上述範例也可以通過以下方式組譯

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

或甚至

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

兩者都產生「`x{710192A70AE4} ok`」。

順便一提，實現上述迴圈的更好方式是使用 `REPEATEND`：

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

或

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

兩者都產生「x{7101E7A70A}」，並在迴圈的七次迭代後輸出「10000000」。

請注意，幾個將延續儲存在單獨單元參照中的 TVM 指令（如 JMPREF）接受 *Cell* 而不是 *Builder* 中的引數。在這種情況下，可以使用 <{ … }>c 構造來產生此立即引數。

7.6 控制流：迴圈和條件式

幾乎所有 TVM 控制流指令——如 IF、IFNOT、IFRET、IFNOTRET、IFELSE、WHILE、WHILEEND、REPEAT、REPEATEND、UNTIL 和 UNTILEND——在應用於字面值延續時，可以類似於 **7.5** 範例中的 REPEAT 和 REPEATEND 組譯。例如，TVM 組譯器程式碼

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

根據其引數 n 是奇數還是偶數來計算 $3n+1$ 或 $n/2$，可以組譯並應用於 $n=7$：

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

產生

```
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
```

當然，實現此條件表示式的更緊湊和高效的方式是

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

或

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

兩者都產生相同的程式碼「x{2071B093A703A4DCAB00}」。

可用於產生這種「高階」條件式和迴圈的 Fift 組譯器詞彙包括 IF:<{、IFNOT:<{、IFJMP:<{、}>ELSE<{、}>ELSE:、}>IF、REPEAT:<{、UNTIL:<{、WHILE:<{、}>DO<{、}>DO:、AGAIN:<{、}>AGAIN、}>REPEAT 和}>UNTIL。它們的完整列表可以在原始檔 Asm.fif 中找到。例如，UNTIL 迴圈可以通過 UNTIL:<{ … }> 或 <{ … }>UNTIL 建立，WHILE 迴圈可以通過 WHILE:<{ … }>DO<{ … }> 建立。

如果我們選擇將條件分支保存在單獨的單元中，我們可以使用 <{ … }>c 構造以及諸如 IFJMPREF 之類的指令：

```
<{ DUP 1 INT AND
   <{ 3 MULCONST INC }>c IFJMPREF
   1 RSHIFT#
```

```
}>s dup csr.
3 swap runvmcode .s
```

執行時與前一個範例的程式碼具有相同的效果，但它包含在兩個單獨的單元中：

```
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 巨集定義

因為 TVM 指令在 Fift 組譯器中使用對 Fift 堆疊具有可預測效果的 Fift 詞彙實現，所以 Fift 組譯器自動成為巨集組譯器，支援巨集定義。例如，假設我們希望定義巨集定義 `RANGE` x y，它檢查 TVM 堆疊頂部值是否在整數字面值 x 和 y（包含）之間。此巨集定義可以如下實現：

```
{ 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 .
```

這產生

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

```
execute IFRET
66
```

注意 GEQINT 和 LEQINT 本身是在 Asm.fif 中定義的巨集定義，因為它們不直接對應於 TVM 指令。例如，x GEQINT 對應於 TVM 指令 GTINT $x-1$。

順帶一提，上述程式碼可以透過將 IFNOT: DROP ZERO 替換為 AND 來縮短兩個位元組。

7.8 更大的程式和子例程

更大的 TVM 程式，例如 TON 區塊鏈智慧合約，通常由數個相互遞迴的子例程組成，其中一個或數個被選為頂層子例程（對於智慧合約稱為 main() 或 recv_internal()）。執行從其中一個頂層子例程開始，該子例程可自由呼叫任何其他已定義的子例程，而這些子例程又可以呼叫它們需要的任何其他子例程。

此類 TVM 程式是透過選擇器函數實現的，該函數在 TVM 堆疊中接受一個額外的整數引數；這個整數選擇要呼叫的實際子例程（參見 [4, 4.6]）。在執行之前，此選擇器函數的程式碼被載入到特殊暫存器 c3 和目前接續 cc 中。主函數的選擇器（通常為零）被推入初始堆疊中，然後啟動 TVM 執行。之後可以透過適當的 TVM 指令呼叫子例程，例如 CALLDICT n，其中 n 是要呼叫的子例程的（整數）選擇器。

Fift 組譯器提供了數個詞彙來促進此類大型 TVM 程式的實現。特別是，子例程可以分別定義並指派符號名稱（而非數字選擇器），之後可以使用這些名稱來呼叫它們。Fift 組譯器會自動從這些分離的子例程建立選擇器函數，並將其作為頂層組譯結果返回。

以下是一個由數個子例程組成的程式的簡單範例。此程式計算複數 $(5+i)^4 \cdot (239-i)$：

```
"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
```

此程式產生：

```
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
```

基於前一個範例的一些觀察和評論如下：

- TVM 程式由 `PROGRAM{` 開啟，並由`}END>c`（將組譯的程式作為 *Cell* 返回）或`}END>s`（返回 *Slice*）關閉。
- 新子例程透過片語 `NEWPROC` ⟨*name*⟩ 宣告。此宣告將下一個正整數指派為新宣告子例程的選擇器，並將此整數儲存到常數 ⟨*name*⟩ 中。例如，上述宣告將 `add`、`sub` 和 `mul` 定義為分別等於 1、2 和 3 的整數常數。
- 某些子例程是預先宣告的，不需要透過 `NEWPROC` 再次宣告。例如，`main` 是綁定到整數常數（選擇器）0 的子例程識別符。
- 其他預定義的子例程選擇器，例如 `recv_internal`（等於 0）或 `recv_external`（等於 −1），對於實現 TON 區塊鏈智慧合約很有用（參見 [5, 4.4]），可以透過 `constant` 宣告（例如，`-1 constant recv_external`）。

- 子例程可以透過詞彙 PROC（接受子例程的整數選擇器和包含此子例程程式碼的 *Slice*）定義，或透過結構⟨*selector*⟩ PROC:<{ … }>（方便定義較大的子例程）定義。
- CALLDICT 和 JMPDICT 指令可以透過詞彙 CALL 和 JMP 組譯，這些詞彙接受要呼叫的子例程的整數選擇器作為在 Fift 堆疊中傳遞的立即引數。
- Fift 組譯器的目前實現將所有子例程收集到具有 14 位元有號整數金鑰的字典中。因此，所有子例程選擇器必須在 $-2^{13} \ldots 2^{13} - 1$ 範圍內。
- 如果在執行期間呼叫具有未知選擇器的子例程，則 Fift 組譯器自動插入的程式碼會拋出程式碼為 11 的例外。此程式碼還會透過 SETCP0 指令自動為指令編碼選擇第零代碼頁。
- Fift 組譯器檢查由 NEWPROC 宣告的所有子例程在程式結束前是否確實由 PROC 或 PROC:<{ 定義。它還檢查子例程是否未被重新定義。

應該記住，非常簡單的程式（包括最簡單的智慧合約）可以透過消除這種通用子例程選擇機制，改用自訂子例程選擇程式碼並移除未使用的子例程來使其更加緊湊。例如，上述範例可以轉換為

```
<{ 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
```

這產生

```
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. 可於 `https://www.forth.com/starting-forth/` 取得。

[2] L. Brodie, *Thinking Forth: A language and philosophy for solving problems*, Prentice Hall, 1984. 可於 `http://thinking-forth.sourceforge.net/` 取得。

[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 Fift 詞彙列表

本附錄提供幾乎所有 Fift 詞彙的字母順序列表——包括來自標準函式庫 `Fift.fif` 的基本詞彙和定義，但不包括在 `Asm.fif` 中定義的 Fift 組譯器詞彙（因為從 Fift 的角度來看，Fift 組譯器只是一個應用程式）。某些實驗性詞彙已從此列表中省略。在本文撰寫後，可能已向 Fift 新增或移除其他詞彙。您的 Fift 解譯器中可用的所有詞彙列表可以透過執行 `words` 來檢查。

每個詞彙由其名稱描述，後面是括號中的堆疊記法，指示在詞彙執行前後 Fift 堆疊頂部附近的數個值；通常假設所有更深層的堆疊項目保持不變。之後，提供詞彙效果的文字說明。如果該詞彙已在本文件的前面章節中討論過，則包含對該章節的參考。

活動詞彙和活動前綴詞彙（在其出現後立即解析輸入串流的一部分）以修改的方式在此列出。首先，這些詞彙與它們解析的輸入部分一起列出；為了強調，每個項目中實際上是 Fift 詞彙的部分會加上底線。其次，它們的堆疊效果通常從使用者的角度描述，並反映在包含區塊和詞彙定義的執行階段期間執行的動作。

例如，用於定義位元組字面值的活動前綴詞彙 `B{`(參見**5.6**)列為`B{`⟨*hex-digits*⟩`}`，其堆疊效果顯示為 $(-B)$ 而不是 $(-B\ 1\ e)$，儘管在包含區塊或詞彙定義的編譯階段期間執行活動詞彙 `B{` 的真實效果是後者（參見**4.2**）。

- `!` $(x\ p\ -\ )$，將新值 x 儲存到 *Box* p 中，參見**2.14**。
- `"`⟨*string*⟩`"` $(\ -\ S)$，將 *String* 字面值推入堆疊，參見**2.9** 和 **2.10**。
- `#` $(x\ S\ -\ x'\ S')$，執行將 *Integer* x 轉換為其十進位表示的一個步驟，方法是向 *String* S 附加一個表示 $x \bmod 10$ 的十進位數字。商 $x' := \lfloor x/10 \rfloor$ 也會返回。
- `#>` $(S\ -\ S')$，透過反轉 *String* S 來完成使用 `<#` 開始的將 *Integer* 轉換為其可讀表示（十進位或其他）的轉換。等同於 `$reverse`。
- `#s` $(x\ S\ -\ x'\ S')$，執行 `#` 一次或多次，直到商 x' 變為非正數。等同於 `{ # over 0<= } until`。
- `$#` $(\ -\ x)$，推送傳遞給 Fift 程式的命令列引數總數，參見**2.18**。僅在 Fift 解譯器以腳本模式呼叫時（使用 `-s` 命令列引數）定義。
- `$(`⟨*string*⟩`)` $(\ -\ \ldots)$，在執行期間查詢詞彙 `$`⟨*string*⟩ 並執行其目前定義。通常用於存取命令列引數的目前值，例如，`$(2)` 本質上等同於 `@' $2`。

- `$()` $(x - S)$，類似於 `$n` 推送第 x 個命令列引數，但 *Integer* $x \geq 0$ 從堆疊中取得，參見**2.18**。僅在 Fift 解譯器以腳本模式呼叫時（使用 `-s` 命令列引數）定義。
- `$+` $(S\ S' - S.S')$，串接兩個字串，參見**2.10**。
- `$,` $(b\ S - b')$，將 *String* S 附加到 *Builder* b，參見**5.2**。該字串被解釋為長度為 $8n$ 的二進位字串，其中 n 是 S 的 UTF-8 表示中的位元組數。
- `$`n $(- S)$，將第 n 個命令列引數作為 *String* S 推送，參見**2.18**。例如，`$0` 推送正在執行的腳本名稱，`$1` 推送第一個命令列引數，依此類推。僅在 Fift 解譯器以腳本模式呼叫時（使用 `-s` 命令列引數）定義。
- `$=` $(S\ S' - ?)$，如果字串 S 和 S' 相等則返回 -1，否則返回 0，參見**2.13**。等同於 `$cmp 0=`。
- `$>s` $(S - s)$，將 *String* S 轉換為 *Slice*，參見**5.3**。等同於 `<b swap $, b> <s`。
- `$>smca` $(S - x\ y\ z\ -1 \text{ or } 0)$，從其可讀字串表示 S 解包標準 TON 智慧合約地址，參見**6.2**。成功時，返回有號 32 位元工作鏈 x、無號 256 位元工作鏈內地址 y、旗標 z（其中 $+1$ 表示地址不可彈回，$+2$ 表示地址僅限測試網），以及 -1。失敗時，推送 0。
- `$@` $(s\ x - S)$，從 *Slice* s 提取前 x 個位元組（即 $8x$ 位元），並將它們作為 UTF-8 *String* S 返回，參見**5.3**。如果 s 中沒有足夠的資料位元，則拋出例外。
- `$@+` $(s\ x - S\ s')$，類似於 `$@`，但也返回 *Slice* s 的剩餘部分，參見**5.3**。
- `$@?` $(s\ x - S\ -1 \text{ or } 0)$，類似於 `$@`，但使用旗標來指示失敗而不是拋出例外，參見**5.3**。
- `$@?+` $(s\ x - S\ s'\ -1 \text{ or } s\ 0)$，類似於 `$@+`，但使用旗標來指示失敗而不是拋出例外，參見**5.3**。
- `$cmp` $(S\ S' - x)$，如果字串 S 和 S' 相等則返回 0，如果 S 在字典序上小於 S' 則返回 -1，如果 S 在字典序上大於 S' 則返回 1，參見**2.13**。

- `$len` $(S - x)$，計算字串的位元組長度（而非 UTF-8 字元長度 ⁉），參見**2.10**。
- `$pos` $(S\ S' - x \text{ or } -1)$，返回子字串 S' 在字串 S 中第一次出現的位置（位元組偏移量） x，或 -1。
- `$reverse` $(S - S')$，反轉 *String* S 中 UTF-8 字元的順序。如果 S 不是有效的 UTF-8 字串，返回值未定義且也可能無效。
- `%1<<` $(x\ y - z)$，對於兩個 *Integer*s x 和 $0 \leq y \leq 256$，計算 $z := x \bmod 2^y = x\&(2^y - 1)$。
- `'` ⟨***word-name***⟩ $(- e)$，返回等於 ⟨*word-name*⟩ 的目前（編譯期）定義的執行權杖，參見**3.1**。如果找不到指定的詞彙，則拋出例外。
- `'nop` $(- e)$，推送 `nop` 的預設定義——一個在執行時不執行任何操作的執行權杖，參見**4.6**。
- `('')` ⟨***word-name***⟩ $(- e)$，類似於`'`，但在執行期間返回指定詞彙的定義，每次呼叫時執行字典查詢，參見**4.6**。可用於透過使用片語 `('')` ⟨***word-name***⟩ `execute` 在詞彙定義和其他區塊內復原常數的目前值。
- `(-trailing)` $(S\ x - S')$，從 *String* S 中移除所有具有 UTF-8 程式碼點 x 的尾隨字元。
- `(.)` $(x - S)$，返回具有 *Integer* x 的十進位表示的 *String*。等同於 `dup abs <# #s rot sign #> nip`。
- `(atom)` $(S\ x - a\ -1 \text{ or } 0)$，返回名稱由 *String* S 給出的唯一 *Atom* a，參見**2.17**。如果尚無此 *Atom*，則建立它（如果 *Integer* x 為非零）或返回單個零以指示失敗（如果 x 為零）。
- `(b.)` $(x - S)$，返回具有 *Integer* x 的二進位表示的 *String*。
- `(compile)` $(l\ x_1\ ...x_n\ n\ e - l')$，擴展 *WordList* l，使其在呼叫時將 $0 \leq n \leq 255$ 個值 x_1、...、x_n 推入堆疊並執行執行權杖 e，其中 $0 \leq n \leq 255$ 是 *Integer*，參見**4.7**。如果 e 等於特殊值`'nop`，則省略最後一步。
- `(create)` $(e\ S\ x -)$，建立一個新詞彙，名稱等於 *String* S，定義等於 *WordDef* e，使用在 *Integer* $0 \leq x \leq 3$ 中傳遞的旗標，參見**4.5**。如果 x 中設定了位元 $+1$，則建立活動詞彙；如果設定了位元 $+2$，則建立前綴詞彙。

- `(def?)` $(S - ?)$，檢查詞彙 S 是否已定義。
- `(dump)` $(x - S)$，返回一個 *String*，其中包含最頂部堆疊值 x 的傾印，格式與`.dump` 使用的格式相同。
- `(execute)` $(x_1 \ldots x_n\ n\ e - \ldots)$，執行執行權杖 e，但首先檢查堆疊中除 n 和 e 本身外是否至少有 $0 \leq n \leq 255$ 個值。它是 `(compile)` 的對應詞，可用於在活動詞彙的立即執行後立即「執行」（執行其預期的執行期動作），如**4.2** 中所述。
- `(forget)` $(S -)$，忘記在 *String* S 中指定名稱的詞彙，參見**4.5**。如果找不到詞彙，則拋出例外。
- `(number)` $(S - 0 \text{ or } x\ 1 \text{ or } x\ y\ 2)$，嘗試將 *String* S 解析為整數或分數字面值，參見**2.10** 和 **2.8**。失敗時，返回單個 0。成功時，如果 S 是值為 x 的有效整數字面值，則返回 x 1，或者如果 S 是值為 x/y 的有效分數字面值，則返回 $x\ y\ 2$。
- `(x.)` $(x - S)$，返回具有 *Integer* x 的十六進位表示的 *String*。
- `({)` $(- l)$，將空的 *WordList* 推入堆疊，參見**4.7**
- `(})` $(l - e)$，將 *WordList* 轉換為執行權杖（*WordDef*），使所有進一步的修改不可能，參見**4.7**。
- `*` $(x\ y - xy)$，計算兩個 *Integer*s x 和 y 的乘積 xy，參見**2.4**。
- `*/` $(x\ y\ z - \lfloor xy/z \rfloor)$，「先乘後除」：將兩個整數 x 和 y 相乘產生 513 位元中間結果，然後將乘積除以 z，參見**2.4**。
- `*/c` $(x\ y\ z - \lceil xy/z \rceil)$，帶有向上取整的「先乘後除」：將兩個整數 x 和 y 相乘產生 513 位元中間結果，然後將乘積除以 z，參見**2.4**。
- `*/cmod` $(x\ y\ z - q\ r)$，類似於 `*/c`，但同時計算商 $q := \lceil xy/z \rceil$ 和餘數 $r := xy - qz$，參見**2.4**。
- `*/mod` $(x\ y\ z - q\ r)$，類似於 `*/`，但同時計算商 $q := \lfloor xy/z \rfloor$ 和餘數 $r := xy - qz$，參見**2.4**。
- `*/r` $(x\ y\ z - q := \lfloor xy/z + 1/2 \rfloor)$，帶有最接近整數取整的「先乘後除」：將兩個整數 x 和 y 相乘產生 513 位元中間結果，然後將乘積除以 z，參見**2.4**。

- `*/rmod` $(x\ y\ z - q\ r)$，類似於 `*/r`，但同時計算商 $q := \lfloor xy/z + 1/2 \rfloor$ 和餘數 $r := xy - qz$，參見**2.4**。
- `*>>` $(x\ y\ z - q)$，類似於 `*/`，但將除法替換為右移，參見**2.4**。對於 $0 \leq z \leq 256$ 計算 $q := \lfloor xy/2^z \rfloor$。等同於 `1<< */`。
- `*>>c` $(x\ y\ z - q)$，類似於 `*/c`，但將除法替換為右移，參見**2.4**。對於 $0 \leq z \leq 256$ 計算 $q := \lceil xy/2^z \rceil$。等同於 `1<< */c`。
- `*>>r` $(x\ y\ z - q)$，類似於 `*/r`，但將除法替換為右移，參見**2.4**。對於 $0 \leq z \leq 256$ 計算 $q := \lfloor xy/2^z + 1/2 \rfloor$。等同於 `1<< */r`。
- `*mod` $(x\ y\ z - r)$，類似於 `*/mod`，但僅計算餘數 $r := xy - qz$，其中 $q := \lfloor xy/z \rfloor$。等同於 `*/mod nip`。
- `+` $(x\ y - x + y)$，計算兩個 *Integer*s x 和 y 的和 $x + y$，參見**2.4**。
- `+!` $(x\ p -)$，將儲存在 *Box* p 中的整數值增加 *Integer* x，參見**2.14**。等同於 `tuck @ + swap !`。
- `+"`⟨*string*⟩`"` $(S - S')$，將 *String* S 與字串字面值串接，參見**2.10**。等同於`"`⟨*string*⟩`" $+`。
- `,` $(t\ x - t')$，將 x 附加到 *Tuple* t 的末尾，並返回結果 *Tuple* t'，參見**2.15**。
- `-` $(x\ y - x - y)$，計算兩個 *Integer*s x 和 y 的差 $x - y$，參見**2.4**。
- `-!` $(x\ p -)$，將儲存在 *Box* p 中的整數值減少 *Integer* x。等同於 `swap negate swap +!`。
- `-1` $(- -1)$，推送 *Integer* -1。
- `-1<<` $(x - -2^x)$，對於 $0 \leq x \leq 256$ 計算 -2^x。大約等同於 `1<< negate` 或 `-1 swap <<`，但對於 $x = 256$ 也能運作。
- `-roll` $(x_n\ \ldots x_0\ n - x_0\ x_n\ \ldots x_1)$，將頂部 n 個堆疊項目向相反方向旋轉，其中 $n \geq 0$ 也在堆疊中傳遞，參見**2.5**。特別是，`1 -roll` 等同於 `swap`，`2 -roll` 等同於 `-rot`。
- `-rot` $(x\ y\ z - z\ x\ y)$，將最頂部的三個堆疊項目向相反方向旋轉，參見**2.5**。等同於 `rot rot`。

- `-trailing` ($S - S'$)，從 *String* S 中移除所有尾隨空格。等同於 `bl (-trailing)`。
- `-trailing0` ($S - S'$)，從 *String* S 中移除所有尾隨 '0' 字元。等同於 `char 0 (-trailing)`。
- `.` ($x -$)，列印 *Integer* x 的十進位表示，後面跟著一個空格，參見**2.4**。等同於`._ space`。
- `."`⟨*string*⟩`"` (–)，將常數字串列印到標準輸出，參見**2.10**。
- `._` ($x -$)，列印 *Integer* x 的十進位表示，不包含任何空格。等同於`(.) type`。
- `.dump` ($x -$)，以與`.s` 傾印所有堆疊元素相同的方式傾印最頂部的堆疊項目，參見**2.15**。等同於`(dump) type space`。
- `.l` ($l -$)，列印 Lisp 風格列表 l，參見**2.16**。
- `.s` (–)，從最深處開始傾印所有堆疊項目，保持它們不變，參見**2.5**。堆疊項目的可讀表示以空格分隔輸出，後面跟著換行字元。
- `.sl` (–)，類似於`.s` 傾印所有堆疊項目並保持它們不變，但像`.l` 一樣將每個項目顯示為 List 風格列表 l。
- `.tc` (–)，將分配的單元總數輸出到標準錯誤串流。
- `/` ($x\ y - q := \lfloor x/y \rfloor$)，計算兩個 *Integer*s 的向下取整商 $\lfloor x/y \rfloor$，參見**2.4**。
- `/*` ⟨*multiline-comment*⟩ `*/` (–)，跳過由詞彙「`*/`」（後面跟著空白或換行字元）分隔的多行註釋，參見**2.2**。
- `//` ⟨*comment-to-eol*⟩ (–)，跳過單行註釋直到當前行結束，參見**2.2**。
- `/c` ($x\ y - q := \lceil x/y \rceil$)，計算兩個 *Integer*s 的向上取整商 $\lceil x/y \rceil$，參見**2.4**。
- `/cmod` ($x\ y - q\ r$)，同時計算向上取整商 $q := \lceil x/y \rceil$ 和餘數 $r := x - qy$，參見**2.4**。

- `/mod` $(x\ y - q\ r)$，同時計算向下取整商 $q := \lfloor x/y \rfloor$ 和餘數 $r := x - qy$，參見**2.4**。
- `/r` $(x\ y - q)$，計算兩個 *Integer*s 的最接近整數取整商 $\lfloor x/y + 1/2 \rfloor$，參見**2.4**。
- `/rmod` $(x\ y - q\ r)$，同時計算最接近整數取整商 $q := \lfloor x/y + 1/2 \rfloor$ 和餘數 $r := x - qy$，參見**2.4**。
- `0` $(- 0)$，推送 *Integer* 0。
- `0!` $(p -)$，將 *Integer* 0 儲存到 *Box* p 中，參見**2.14**。等同於 `0 swap !`。
- `0<` $(x - ?)$，檢查 $x < 0$（即如果 x 為負數則推送 -1，否則推送 0），參見**2.12**。等同於 `0 <`。
- `0<=` $(x - ?)$，檢查 $x \leq 0$（即如果 x 為非正數則推送 -1，否則推送 0），參見**2.12**。等同於 `0 <=`。
- `0<>` $(x - ?)$，檢查 $x \neq 0$（即如果 x 為非零則推送 -1，否則推送 0），參見**2.12**。等同於 `0 <>`。
- `0=` $(x - ?)$，檢查 $x = 0$（即如果 x 為零則推送 -1，否則推送 0），參見**2.12**。等同於 `0 =`。
- `0>` $(x - ?)$，檢查 $x > 0$（即如果 x 為正數則推送 -1，否則推送 0），參見**2.12**。等同於 `0 >`。
- `0>=` $(x - ?)$，檢查 $x \geq 0$（即如果 x 為非負數則推送 -1，否則推送 0），參見**2.12**。等同於 `0 >=`。
- `1` $(- 1)$，推送 *Integer* 1。
- `1+` $(x - x + 1)$，計算 $x + 1$。等同於 `1 +`。
- `1+!` $(p -)$，將儲存在 *Box* p 中的整數值增加一，參見**2.14**。等同於 `1 swap +!`。
- `1-` $(x - x - 1)$，計算 $x - 1$。等同於 `1 -`。
- `1-!` $(p -)$，將儲存在 *Box* p 中的整數值減少一。等同於 `-1 swap +!`。

- `1<<` $(x - 2^x)$，對於 $0 \leq x \leq 255$ 計算 2^x。等同於 `1 swap <<`。
- `1<<1-` $(x - 2^x - 1)$，對於 $0 \leq x \leq 256$ 計算 $2^x - 1$。幾乎等同於 `1<< 1-`，但對於 $x = 256$ 也能運作。
- `2` $(- 2)$，推送 *Integer* 2。
- `2*` $(x - 2x)$，計算 $2x$。等同於 `2 *`。
- `2+` $(x - x + 2)$，計算 $x + 2$。等同於 `2 +`。
- `2-` $(x - x - 2)$，計算 $x - 2$。等同於 `2 -`。
- `2/` $(x - \lfloor x/2 \rfloor)$，計算 $\lfloor x/2 \rfloor$。等同於 `2 /` 或 `1 >>`。
- `2=:` ⟨***word-name***⟩ $(x\ y -)$，`2constant` 的活動變體：定義一個新的普通詞彙 ⟨*word-name*⟩，當呼叫時將推送給定值 x 和 y，參見**2.7**。
- `2constant` $(x\ y -)$，從輸入的其餘部分掃描一個以空白分隔的詞彙名稱 S，並將新的普通詞彙 S 定義為雙常數，當呼叫時將推送給定值 x 和 y（任意類型），參見**4.5**。
- `2drop` $(x\ y -)$，移除最頂部的兩個堆疊項目，參見**2.5**。等同於 `drop drop`。
- `2dup` $(x\ y - x\ y\ x\ y)$，複製最頂部的堆疊項目對，參見**2.5**。等同於 `over over`。
- `2over` $(x\ y\ z\ w - x\ y\ z\ w\ x\ y)$，複製第二頂部的堆疊項目對。
- `2swap` $(a\ b\ c\ d - c\ d\ a\ b)$，交換最頂部的兩對堆疊項目，參見**2.5**。
- `:` ⟨***word-name***⟩ $(e -)$，使用 *WordDef* e 作為其定義在字典中定義新的普通詞彙 ⟨*word-name*⟩，參見**4.5**。如果指定的詞彙已存在於字典中，則會被默認重新定義。
- `::` ⟨***word-name***⟩ $(e -)$，使用 *WordDef* e 作為其定義在字典中定義新的活動詞彙 ⟨*word-name*⟩，參見**4.5**。如果指定的詞彙已存在於字典中，則會被默認重新定義。
- `::_` ⟨***word-name***⟩ $(e -)$，使用 *WordDef* e 作為其定義在字典中定義新的活動前綴詞彙 ⟨*word-name*⟩，參見**4.5**。如果指定的詞彙已存在於字典中，則會被默認重新定義。

- `:_` ⟨*word-name*⟩ $(e -)$，使用 *WordDef* e 作為其定義在字典中定義新的普通前綴詞彙 ⟨*word-name*⟩，參見**4.5**。如果指定的詞彙已存在於字典中，則會被默認重新定義。

- `<` $(x\ y - ?)$，檢查 $x < y$（即如果 *Integer* x 小於 *Integer* y 則推送 -1，否則推送 0），參見**2.12**。

- `<#` $(- S)$，推送空的 *String*。通常用於開始將 *Integer* 轉換為其可讀表示，十進位或其他基數。等同於`""`。

- `<<` $(x\ y - x \cdot 2^y)$，計算二進位數字 x 左移 $y \geq 0$ 個位置的算術左移，產生 $x \cdot 2^y$，參見**2.4**。

- `<</` $(x\ y\ z - q)$，對於 $0 \leq z \leq 256$ 計算 $q := \lfloor 2^z x/y \rfloor$，產生 513 位元中間結果，類似於 `*/`，參見**2.4**。等同於 `1<< swap */`。

- `<</c` $(x\ y\ z - q)$，對於 $0 \leq z \leq 256$ 計算 $q := \lceil 2^z x/y \rceil$，產生 513 位元中間結果，類似於 `*/c`，參見**2.4**。等同於 `1<< swap */c`。

- `<</r` $(x\ y\ z - q)$，對於 $0 \leq z \leq 256$ 計算 $q := \lfloor 2^z x/y + 1/2 \rfloor$，產生 513 位元中間結果，類似於 `*/r`，參見**2.4**。等同於 `1<< swap */r`。

- `<=` $(x\ y - ?)$，檢查 $x \leq y$（即如果 *Integer* x 小於或等於 *Integer* y 則推送 -1，否則推送 0），參見**2.12**。

- `<>` $(x\ y - ?)$，檢查 $x \neq y$（即如果 *Integer*s x 和 y 不相等則推送 -1，否則推送 0），參見**2.12**。

- `<b` $(- b)$，建立新的空 *Builder*，參見**5.2**。

- `<s` $(c - s)$，將 *Cell* c 轉換為包含相同資料的 *Slice* s，參見**5.3**。它通常標記單元反序列化的開始。

- `=` $(x\ y - ?)$，檢查 $x = y$（即如果 *Integer*s x 和 y 相等則推送 -1，否則推送 0），參見**2.12**。

- `=:` ⟨*word-name*⟩ $(x -)$，`constant` 的活動變體：定義一個新的普通詞彙 ⟨*word-name*⟩，當呼叫時將推送給定值 x，參見**2.7**。

- `>` $(x\ y - ?)$，檢查 $x > y$（即如果 *Integer* x 大於 *Integer* y 則推送 -1，否則推送 0），參見**2.12**。

- `>=` $(x\ y - ?)$，檢查 $x \geq y$（即如果 *Integer* x 大於或等於 *Integer* y 則推送 -1，否則推送 0），參見**2.12**。
- `>>` $(x\ y - q := \lfloor x \cdot 2^{-y} \rfloor)$，計算二進位數字 x 右移 $0 \leq y \leq 256$ 個位置的算術右移，參見**2.4**。等同於 `1<< /`。
- `>>c` $(x\ y - q := \lceil x \cdot 2^{-y} \rceil)$，對於 $0 \leq y \leq 256$ 計算 x 除以 2^y 的向上取整商 q，參見**2.4**。等同於 `1<< /c`。
- `>>r` $(x\ y - q := \lfloor x \cdot 2^{-y} + 1/2 \rfloor)$，對於 $0 \leq y \leq 256$ 計算 x 除以 2^y 的最接近整數取整商 q，參見**2.4**。等同於 `1<< /r`。
- `?dup` $(x - x\ x \text{ or } 0)$，複製 *Integer* x，但僅在其為非零時，參見**2.5**。否則保持不變。
- `@` $(p - x)$，提取當前儲存在 *Box* p 中的值，參見**2.14**。
- `@'` ⟨*word-name*⟩ $(- e)$，在執行期間復原指定詞彙的定義，每次呼叫時執行字典查詢，然後執行此定義，參見**2.7** 和 **4.6**。可用於透過使用片語 `@'` ⟨*word-name*⟩ 在詞彙定義和其他區塊內復原常數的目前值，等同於 `(')` ⟨*word-name*⟩ `execute`。
- `B+` $(B'\ B'' - B)$，串接兩個 *Bytes* 值，參見**5.6**。
- `B,` $(b\ B - b')$，將 *Bytes* B 附加到 *Builder* b，參見**5.2**。如果 b 中沒有足夠的空間容納 B，則拋出例外。
- `B=` $(B\ B' - ?)$，檢查兩個 *Bytes* 序列是否相等，並根據比較結果返回 -1 或 0，參見**5.6**。
- `B>Li@` $(B\ x - y)$，將 `Bytes` 值 B 的前 $x/8$ 個位元組反序列化為有號小端序 x 位元 *Integer* y，參見**5.6**。
- `B>Li@+` $(B\ x - B'\ y)$，類似於 `B>Li@` 將 B 的前 $x/8$ 個位元組反序列化為有號小端序 x 位元 *Integer* y，但也返回 B 的剩餘位元組，參見**5.6**。
- `B>Lu@` $(B\ x - y)$，將 `Bytes` 值 B 的前 $x/8$ 個位元組反序列化為無號小端序 x 位元 *Integer* y，參見**5.6**。
- `B>Lu@+` $(B\ x - B'\ y)$，類似於 `B>Lu@` 將 B 的前 $x/8$ 個位元組反序列化為無號小端序 x 位元 *Integer* y，但也返回 B 的剩餘位元組，參見**5.6**。

- `B>boc` $(B - c)$，反序列化由 *Bytes* B 表示的「標準」單元集合（即恰好具有一個根單元的單元集合），並返回根 *Cell* c，參見**5.5**。
- `B>file` $(B\ S -)$，建立一個名稱由 *String* S 指定的新（二進位）檔案，並將來自 *Bytes* B 的資料寫入新檔案，參見**5.6**。如果指定的檔案已存在，則會被覆寫。
- `B>i@` $(B\ x - y)$，將 `Bytes` 值 B 的前 $x/8$ 個位元組反序列化為有號大端序 x 位元 *Integer* y，參見**5.6**。
- `B>i@+` $(B\ x - B'\ y)$，類似於 `B>i@` 將 B 的前 $x/8$ 個位元組反序列化為有號大端序 x 位元 *Integer* y，但也返回 B 的剩餘位元組，參見**5.6**。
- `B>u@` $(B\ x - y)$，將 `Bytes` 值 B 的前 $x/8$ 個位元組反序列化為無號大端序 x 位元 *Integer* y，參見**5.6**。
- `B>u@+` $(B\ x - B'\ y)$，類似於 `B>u@` 將 B 的前 $x/8$ 個位元組反序列化為無號大端序 x 位元 *Integer* y，但也返回 B 的剩餘位元組，參見**5.6**。
- `B@` $(s\ x - B)$，從 *Slice* s 提取前 x 個位元組（即 $8x$ 位元），並將它們作為 *Bytes* 值 B 返回，參見**5.3**。如果 s 中沒有足夠的資料位元，則拋出例外。
- `B@+` $(s\ x - B\ s')$，類似於 `B@`，但也返回 *Slice* s 的剩餘部分，參見**5.3**。
- `B@?` $(s\ x - B\ -1 \text{ or } 0)$，類似於 `B@`，但使用旗標來指示失敗而不是拋出例外，參見**5.3**。
- `B@?+` $(s\ x - B\ s'\ -1 \text{ or } s\ 0)$，類似於 `B@+`，但使用旗標來指示失敗而不是拋出例外，參見**5.3**。
- `Bcmp` $(B\ B' - x)$，按字典序比較兩個 *Bytes* 序列，並根據比較結果返回 -1、0 或 1，參見**5.6**。
- `Bhash` $(B - x)$，`Bhashu` 的已棄用版本。請使用 `Bhashu` 或 `BhashB` 代替。
- `BhashB` $(B - B')$，計算 *Bytes* 值的 SHA256 雜湊，參見**5.6**。雜湊作為 32 位元組 *Bytes* 值返回。

- `Bhashu` $(B - x)$，計算 *Bytes* 值的 SHA256 雜湊，參見**5.6**。雜湊作為大端序無號 256 位元 *Integer* 值返回。
- `Blen` $(B - x)$，返回 *Bytes* 值 B 的長度（以位元組為單位），參見**5.6**。
- `Bx.` $(B -)$，列印 *Bytes* 值的十六進位表示，參見**5.6**。每個位元組恰好由兩個大寫十六進位數字表示。
- `B{`⟨*hex-digits*⟩`}` $(- B)$，推送包含由偶數個十六進位數字表示的資料的 *Bytes* 字面值，參見**5.6**。
- `B|` $(B\ x - B'\ B'')$，從 *Bytes* 值 B 中切出前 x 個位元組，並將前 x 個位元組（B'）和剩餘部分（B''）作為新的 *Bytes* 值返回，參見**5.6**。
- `Li>B` $(x\ y - B)$，將有號小端序 y 位元 *Integer* x 儲存到恰好由 $y/8$ 個位元組組成的 *Bytes* 值 B 中。整數 y 必須是 $0 \ldots 256$ 範圍內的八的倍數，參見**5.6**。
- `Lu>B` $(x\ y - B)$，將無號小端序 y 位元 *Integer* x 儲存到恰好由 $y/8$ 個位元組組成的 *Bytes* 值 B 中。整數 y 必須是 $0 \ldots 256$ 範圍內的八的倍數，參見**5.6**。
- `[` $(-)$，即使 `state` 大於零也開啟內部解譯器會話，即所有後續詞彙都會立即執行而不是被編譯。
- `[]` $(t\ i - x)$，返回 *Tuple* t 的第 $(i+1)$ 個分量 t_{i+1}，其中 $0 \le i < |t|$，參見**2.15**。
- `[compile]` ⟨*word-name*⟩ $(-)$，編譯 ⟨*word-name*⟩，就好像它是普通詞彙一樣，即使它是活動的，參見**4.6**。本質上等同於`'` ⟨*word-name*⟩ `execute`。
- `]` $(x_1\ \ldots x_n\ n -)$，關閉由 `[` 開啟的內部解譯器會話，並根據 `state` 是否大於零之後呼叫 `(compile)` 或 `(execute)`。例如，`{ [ 2 3 + 1 ] * }` 等同於 `{ 5 * }`。
- `` `⟨*word*⟩ `` $(- a)$，引入 *Atom* 字面值，等於名稱等於 ⟨*word*⟩ 的唯一 *Atom*，參見**2.17**。等同於`"`⟨*word*⟩`" atom`。
- `abort` $(S -)$，拋出一個錯誤訊息取自 *String* S 的例外，參見**3.6**。

- abort"⟨*message*⟩" $(x -)$，如果 *Integer* x 為非零，則拋出一個錯誤訊息為 $\langle message \rangle$ 的例外，參見**3.6**。
- abs $(x - |x|)$，計算 *Integer* x 的絕對值 $|x| = \max(x, -x)$。等同於 `dup negate max`。
- allot $(n - t)$，建立新陣列，即由 n 個新空 *Box*es 組成的 *Tuple*，參見**2.15**。等同於 `| { hole , } rot times`。
- and $(x\ y - x\&y)$，計算兩個 *Integer*s 的位元 AND，參見**2.4**。
- anon $(- a)$，建立一個新的唯一匿名 *Atom*，參見**2.17**。
- atom $(S - a)$，返回唯一名稱為 S 的 *Atom* a，必要時建立此原子，參見**2.17**。等同於 `true (atom) drop`。
- atom? $(u - ?)$，檢查 u 是否為 *Atom*，參見**2.17**。
- b+ $(b\ b' - b'')$，串接兩個 *Builders* b 和 b'，參見**5.2**。
- b. $(x -)$，列印 *Integer* x 的二進位表示，後面跟著一個空格。等同於 `b._ space`。
- b._ $(x -)$，列印 *Integer* x 的二進位表示，不包含任何空格。等同於 `(b.) type`。
- b> $(b - c)$，將 *Builder* b 轉換為包含與 b 相同資料的新 *Cell* c，參見**5.2**。
- b>idict! $(v\ x\ D\ n - D'\ -1 \text{ or } D\ 0)$，將新值 v（由 *Builder* 表示）與由有號大端序 n 位元整數 x 給定的金鑰加入到具有 n 位元金鑰的字典 D 中，並在成功時返回新字典 D' 和 -1，參見**6.3**。否則返回未更改的字典 D 和 0。
- b>idict!+ $(v\ x\ D\ n - D'\ -1 \text{ or } D\ 0)$，類似於 b>idict! 將新金鑰-值對 (x, v) 加入字典 D，但如果金鑰已存在則失敗，返回未更改的字典 D 和 0，參見**6.3**。
- b>sdict! $(v\ k\ D\ n - D'\ -1 \text{ or } D\ 0)$，將新值 v（由 *Builder* 表示）與由 *Slice* k 的前 n 個位元給定的金鑰加入到具有 n 位元金鑰的字典 D 中，並在成功時返回新字典 D' 和 -1，參見**6.3**。否則返回未更改的字典 D 和 0。

- `b>sdict!+` $(v\ k\ D\ n - D'\ -1 \text{ or } D\ 0)$，類似於 `b>sdict!` 將新金鑰-值對 (k, v) 加入字典 D，但如果金鑰已存在則失敗，返回未更改的字典 D 和 0，參見**6.3**。

- `b>udict!` $(v\ x\ D\ n - D'\ -1 \text{ or } D\ 0)$，將新值 v（由 *Builder* 表示）與由無號大端序 n 位元整數 x 給定的金鑰加入到具有 n 位元金鑰的字典 D 中，並在成功時返回新字典 D' 和 -1，參見**6.3**。否則返回未更改的字典 D 和 0。

- `b>udict!+` $(v\ x\ D\ n - D'\ -1 \text{ or } D\ 0)$，類似於 `b>udict!` 將新金鑰-值對 (x, v) 加入字典 D，但如果金鑰已存在則失敗，返回未更改的字典 D 和 0，參見**6.3**。

- `bbitrefs` $(b - x\ y)$，返回已儲存在 *Builder* b 中的資料位元數 x 和參照數 y，參見**5.2**。

- `bbits` $(b - x)$，返回已儲存在 *Builder* b 中的資料位元數。結果 x 是範圍 $0 \ldots 1023$ 內的 *Integer*，參見**5.2**。

- `bl` $(- x)$，推送空格的 Unicode 程式碼點，即 32，參見**2.10**。

- `boc+>B` $(c\ x - B)$，建立並序列化「標準」單元集合，包含一個根 *Cell* c 及其所有後代，參見**5.5**。*Integer* 參數 $0 \le x \le 31$ 用於傳遞指示單元集合序列化附加選項的旗標，個別位元具有以下效果：

  - +1 啟用單元集合索引建立（對於大型單元集合的惰性反序列化很有用）。
  - +2 將所有資料的 CRC32-C 包含在序列化中（對於檢查資料完整性很有用）。
  - +4 明確地將根單元的雜湊儲存到序列化中（以便之後可以快速復原而無需完整反序列化）。
  - +8 儲存某些中間（非葉）單元的雜湊（對於大型單元集合的惰性反序列化很有用）。
  - +16 儲存單元快取位元以控制反序列化單元的快取。

  對於非常小的單元集合（例如，TON 區塊鏈外部訊息），x 的典型值為 $x = 0$ 或 $x = 2$，對於大型單元集合（例如，TON 區塊鏈區塊），則為 $x = 31$。

- `boc>B` $(c - B)$，序列化具有根 *Cell* c 及其所有後代的小型「標準」單元集合，參見**5.5**。等同於 `0 boc+>B`。
- `box` $(x - p)$，建立包含指定值 x 的新 *Box*，參見**2.14**。等同於 `hole tuck !`。
- `brefs` $(b - x)$，返回已儲存在 *Builder* b 中的參照數，參見**5.2**。結果 x 是範圍 $0 \ldots 4$ 內的 *Integer*。
- `brembitrefs` $(b - x\ y)$，返回可儲存在 *Builder* b 中的最大附加資料位元數 $0 \leq x \leq 1023$ 和最大附加單元參照數 $0 \leq y \leq 4$，參見**5.2**。
- `brembits` $(b - x)$，返回可儲存在 *Builder* b 中的最大附加資料位元數，參見**5.2**。等同於 `bbits 1023 swap -`。
- `bremrefs` $(b - x)$，返回可儲存在 *Builder* b 中的最大附加單元參照數，參見**5.2**。
- `bye` $(-)$，以零退出碼退出 Fift 解譯器到作業系統，參見**2.3**。等同於 `0 halt`。
- `b{`⟨*binary-data*⟩`}` $(- s)$，建立 *Slice* s，它不包含參照且包含最多 1023 個資料位元，在 ⟨*binary-data*⟩ 中指定，它必須是僅由字元 '0' 和 '1' 組成的字串，參見**5.1**。
- `caddr` $(l - h'')$，返回列表的第三個元素。等同於 `cddr car`。
- `cadr` $(l - h')$，返回列表的第二個元素，參見**2.16**。等同於 `cdr car`。
- `car` $(l - h)$，返回列表的頭部，參見**2.16**。等同於 `first`。
- `cddr` $(l - t')$，返回列表的尾部的尾部。等同於 `cdr cdr`。
- `cdr` $(l - t)$，返回列表的尾部，參見**2.16**。等同於 `second`。
- `char` ⟨*string*⟩ $(- x)$，推送具有⟨*string*⟩ 第一個字元的 Unicode 程式碼點的 *Integer*，參見**2.10**。例如，`char *` 等同於 `42`。
- `chr` $(x - S)$，返回由一個 UTF-8 編碼字元組成的新 *String* S，其 Unicode 程式碼點為 x。

- `cmp` $(x\ y - z)$，比較兩個 *Integer*s x 和 y，如果 $x > y$ 則推送 1，如果 $x < y$ 則推送 -1，如果 $x = y$ 則推送 0，參見**2.12**。大約等同於 `- sgn`。

- `cond` $(x\ e\ e' -)$，如果 *Integer* x 為非零，則執行 e，否則執行 e'，參見**3.2**。

- `cons` $(h\ t - l)$，從其頭部（第一個元素）h 和其尾部（由所有剩餘元素組成的列表） t 建構列表，參見**2.16**。等同於 `pair`。

- `constant` $(x -)$，從輸入的其餘部分掃描一個以空白分隔的詞彙名稱 S，並將新的普通詞彙 S 定義為常數，當呼叫時將推送給定值 x（任意類型），參見**4.5** 和 **2.7**。

- `count` $(t - n)$，返回 *Tuple* t 的長度 $n = |t|$，參見**2.15**。

- `cr` $(-)$，將回車（或換行字元）輸出到標準輸出，參見**2.10**。

- `create` $(e -)$，定義一個新的普通詞彙，名稱等於從輸入掃描的下一個詞彙，使用 *WordDef* e 作為其定義，參見**4.5**。如果詞彙已存在，則會被默認重新定義。

- `csr.` $(s -)$，遞迴地列印 *Slice* s，參見**5.3**。在第一行上，s 的資料位元以十六進位形式顯示，嵌入到類似於用於 *Slice* 字面值的 `x{…}` 結構中（參見**5.1**）。在接下來的幾行上，s 所參照的單元以更大的縮排列印。

- `def?` ⟨*word-name*⟩ $(- ?)$，在執行期間檢查詞彙 ⟨*word-name*⟩ 是否已定義，並相應地返回 -1 或 0。

- `depth` $(- n)$，將 Fift 堆疊的目前深度（項目總數）作為 *Integer* $n \geq 0$ 返回。

- `dictmap` $(D\ n\ e - D')$，將執行權杖 e（即匿名函數）應用於儲存在具有 n 位元金鑰的字典 D 中的每個金鑰-值對，參見**6.3**。執行權杖為每個金鑰-值對執行一次，在執行 e 之前將 *Builder* b 和 *Slice* v（包含值）推入堆疊。執行後 e 必須在堆疊中留下修改後的 *Builder* b'（包含來自 b 的所有資料以及新值 v'）和 -1，或 0 指示失敗。在後一種情況下，從新字典中省略相應的金鑰。

- `dictmerge` $(D\ D'\ n\ e - D'')$，將兩個具有 n 位元金鑰的字典 D 和 D' 組合成一個具有相同金鑰的字典 D''，參見**6.3**。如果金鑰僅存在於字典 D 和 D' 中的一個中，則此金鑰和相應值會逐字複製到新字典 D''。否則呼叫執行權杖（匿名函數）e 來合併 D 和 D' 中對應於相同金鑰 k 的兩個值 v 和 v'。在呼叫 e 之前，推入 *Builder* b 和兩個 *Slice*s v 和 v'，表示要合併的兩個值。執行後 e 留下修改後的 *Builder* b'（包含來自 b 的原始資料以及組合值）和 -1，或在失敗時留下 0。在後一種情況下，從新字典中省略相應的金鑰。

- `dictnew` $(-D)$，推送表示新空字典的 *Null* 值，參見**6.3**。等同於 `null`。

- `does` $(x_1\ \ldots x_n\ n\ e - e')$，建立一個新執行權杖 e'，當呼叫時將 n 個值 x_1、…、x_n 推入堆疊，然後執行 e，參見**4.7**。它大致等同於 `({)`、`(compile)` 和 `(})` 的組合。

- `drop` $(x -)$，移除堆疊頂部項目，參見**2.5**。

- `dup` $(x - x\ x)$，複製堆疊頂部項目，參見**2.5**。如果堆疊為空，則拋出例外。

- `ed25519_chksign` $(B\ B'\ B'' - ?)$，檢查 B' 是否為使用公鑰 B'' 對資料 B 的有效 Ed25519 簽章，參見**6.1**。

- `ed25519_sign` $(B\ B' - B'')$，使用 Ed25519 私鑰 B'（一個 32 位元組 *Bytes* 值）對資料 B 進行簽名，並將簽名作為 64 位元組 *Bytes* 值 B'' 返回，參見**6.1**。

- `ed25519_sign_uint` $(x\ B' - B'')$，將大端序無號 256 位元整數 x 轉換為 32 位元組序列，並使用 Ed25519 私鑰 B' 類似於 `ed25519_sign` 進行簽名，參見**6.1**。等同於 `swap 256 u>B swap ed25519_sign`。要簽名的整數 x 通常計算為某些資料的雜湊。

- `emit` $(x -)$，將由 *Integer* x 給出的 Unicode 程式碼點的 UTF-8 編碼字元列印到標準輸出，參見**2.10**。例如，`42 emit` 列印星號「*」，`916 emit` 列印希臘字母 Delta「Δ」。等同於 `chr type`。

- `empty?` $(s - ?)$，檢查 *Slice* 是否為空（即沒有資料位元和剩餘參照），並相應地返回 -1 或 0，參見**5.3**。

- `eq?` $(u\ v - ?)$，檢查 u 和 v 是否為相等的 *Integer*s、*Atom*s 或 *Null*s，參見**2.17**。如果它們不相等，或者它們的類型不同，或者不是列出的類型之一，則返回零。

- `exch` $(x_n\ \ldots x_0\ n - x_0\ \ldots x_n)$，將堆疊頂部與從頂部開始的第 n 個堆疊項目交換，其中 $n \geq 0$ 也從堆疊中取得，參見**2.5**。特別是，`1 exch` 等同於 `swap`，`2 exch` 等同於 `swap rot`。

- `exch2` $(\ldots n\ m - \ldots)$，將從頂部開始的第 n 個堆疊項目與從頂部開始的第 m 個堆疊項目交換，其中 $n \geq 0$、$m \geq 0$ 從堆疊中取得，參見**2.5**。

- `execute` $(e - \ldots)$，執行執行權杖（*WordDef*）e，參見**3.1**。

- `explode` $(t - x_1\ \ldots x_n\ n)$，解包未知長度 n 的 *Tuple* $t = (x_1, \ldots, x_n)$，並返回該長度，參見**2.15**。

- `false` $(- 0)$，將 0 推入堆疊，參見**2.11**。等同於 `0`。

- `file-exists?` $(S - ?)$，檢查名稱由 *String* S 指定的檔案是否存在，參見**5.6**。

- `file>B` $(S - B)$，讀取名稱由 *String* S 指定的（二進位）檔案，並將其內容作為 *Bytes* 值返回，參見**5.6**。如果檔案不存在，則拋出例外。

- `find` $(S - e\ -1 \text{ or } e\ 1 \text{ or } 0)$，在字典中查詢 *String* S，如果找到，則將其定義作為 *WordDef* e 返回，對於普通詞彙後面跟著 -1，對於活動詞彙後面跟著 1，參見**4.6**。否則推送 0。

- `first` $(t - x)$，返回 *Tuple* 的第一個分量，參見**2.15**。等同於 `0 []`。

- `fits` $(x\ y - ?)$，檢查 *Integer* x 是否為有號 y 位元整數（即對於 $0 \leq y \leq 1023$，是否為 $-2^{y-1} \leq x < 2^{y-1}$），並相應地返回 -1 或 0。

- `forget` $(-)$，忘記（從字典中移除）從輸入掃描的下一個詞彙的定義，參見**4.5**。

- `gasrunvm` $(\ldots s\ c\ z - \ldots x\ c'\ z')$，`runvm` 的 gas 感知版本，參見**6.4**：呼叫一個新的 TVM 實例，將目前接續 `cc` 和特殊暫存器 `c3` 都從 *Slice* s 初始化，並使用 *Cell* c 初始化特殊暫存器 `c4`（「持久資料根」，參見 [4, 1.4]）。然後以 gas 限制設定為 z 啟動新的 TVM 實例。實際消耗的 gas z' 在最終 Fift 堆疊頂部返回，`c4` 的最終值在最終 Fift 堆疊頂部下方作為另一個 *Cell* c' 返回。

- `gasrunvmcode` $(...s\ z - ...x\ z')$，`runvmcode` 的 gas 感知版本，參見**6.4**：呼叫一個新的 TVM 實例，從 *Slice* s 初始化目前接續 `cc` 並將 gas 限制設定為 z，從而在 TVM 中執行程式碼 s。原始 Fift 堆疊（不含 s）完整傳遞為新 TVM 實例的初始堆疊。當 TVM 終止時，其結果堆疊用作新的 Fift 堆疊，退出碼 x 和實際消耗的 gas z' 推入其頂部。如果 x 為非零，表示 TVM 已被未處理的例外終止，則從頂部開始的下一個堆疊項目包含此例外的參數，x 是例外代碼。在這種情況下，所有其他項目都從堆疊中移除。

- `gasrunvmctx` $(...s\ c\ t\ z - ...x\ c'\ z')$，`runvmctx` 的 gas 感知版本，參見**6.4**。與 `gasrunmv` 的不同之處在於它使用 *Tuple* t 初始化 `c7`。

- `gasrunvmdict` $(...s\ z - ...x\ z')$，`runvmdict` 的 gas 感知版本，參見**6.4**：呼叫一個新的 TVM 實例，從 *Slice* s 初始化目前接續 `cc` 並將 gas 限制設定為 z，類似於 `gasrunvmcode`，但也使用相同值初始化特殊暫存器 `c3`，並在 TVM 執行開始之前將零推入初始 TVM 堆疊。實際消耗的 gas 作為 *Integer* z' 返回。在典型應用中，*Slice* s 由子例程選擇程式碼組成，該程式碼使用堆疊頂部 *Integer* 來選擇要執行的子例程，從而實現數個相互遞迴子例程的定義和執行（參見 [4, 4.6] 和 **7.8**）。等於零的選擇器對應於大型 TVM 程式中的 `main()` 子例程。

- `halt` $(x -)$，類似於 `bye` 退出到作業系統，但使用 *Integer* x 作為退出碼，參見**2.3**。

- `hash` $(c - x)$，`hashu` 的已棄用版本。請使用 `hashu` 或 `hashB` 代替。

- `hashB` $(c - B)$，計算 *Cell* c 的基於 SHA256 的表示雜湊（參見 [4, 3.1]），它明確定義 c 及其所有後代（假設 SHA256 沒有碰撞），參見**5.4**。結果作為恰好由 32 個位元組組成的 *Bytes* 值返回。

- `hashu` $(c - x)$，類似於 `hashB` 計算 *Cell* c 的基於 SHA256 的表示雜湊，但將結果作為大端序無號 256 位元 *Integer* 返回。

- `hold` $(S\ x - S')$，向 *String* S 附加一個 Unicode 程式碼點為 x 的 UTF-8 編碼字元。等同於 `chr $+`。

- `hole` $(- p)$，建立一個不持有任何值的新 *Box* p，參見**2.14**。等同於 `null box`。

- `i,` $(b\ x\ y - b')$，將有號 y 位元整數 x 的大端序二進位表示附加到 *Builder* b，其中 $0 \leq y \leq 257$，參見**5.2**。如果 b 中沒有足夠的空間（即如果 b 已包含超過 $1023 - y$ 個資料位元），或如果 *Integer* x 不適合 y 位元，則拋出例外。

- `i>B` $(x\ y - B)$，將有號大端序 y 位元 *Integer* x 儲存到恰好由 $y/8$ 個位元組組成的 *Bytes* 值 B 中。整數 y 必須是 $0 \ldots 256$ 範圍內的八的倍數，參見**5.6**。

- `i@` $(s\ x - y)$，從 *Slice* s 的前 x 個位元中提取有號大端序 x 位元整數，參見**5.3**。如果 s 包含少於 x 個資料位元，則拋出例外。

- `i@+` $(s\ x - y\ s')$，類似於 `i@` 從 *Slice* s 的前 x 個位元中提取有號大端序 x 位元整數，但也返回 s 的剩餘部分，參見**5.3**。

- `i@?` $(s\ x - y\ -1 \text{ or } 0)$，類似於 `i@` 從 *Slice* 中提取有號大端序整數，但在成功時之後推送整數 -1，參見**5.3**。如果 s 中剩餘少於 x 個位元，則推送整數 0 以指示失敗。

- `i@?+` $(s\ x - y\ s'\ -1 \text{ or } s\ 0)$，從 *Slice* s 中提取有號大端序整數並計算此 *Slice* 的剩餘部分，類似於 `i@+`，但之後推送 -1 以指示成功，參見**5.3**。失敗時，推送未更改的 *Slice* s 和 0 以指示失敗。

- `idict!` $(v\ x\ D\ n - D'\ -1 \text{ or } D\ 0)$，將新值 v（由 *Slice* 表示）與由有號大端序 n 位元整數 x 給定的金鑰加入到具有 n 位元金鑰的字典 D 中，並在成功時返回新字典 D' 和 -1，參見**6.3**。否則返回未更改的字典 D 和 0。

- `idict!+` $(v\ x\ D\ n - D'\ -1 \text{ or } D\ 0)$，類似於 `idict!` 將新金鑰-值對 (x, v) 加入字典 D，但如果金鑰已存在則失敗，返回未更改的字典 D 和 0，參見**6.3**。

- `idict-` $(x\ D\ n - D'\ -1 \text{ or } D\ 0)$，從由 *Cell* D 表示的字典中刪除由有號大端序 n 位元 *Integer* x 表示的金鑰，參見**6.3**。如果找到金鑰，則從字典中刪除它並返回修改後的字典 D' 和 -1。否則返回未修改的字典 D 和 0。

- `idict@` $(x\ D\ n - v\ -1 \text{ or } 0)$，在由 *Cell* 或 *Null* D 表示的字典中查詢由有號大端序 n 位元 *Integer* x 表示的金鑰，參見**6.3**。如果找到金鑰，則將相應值作為 *Slice* v 和 -1 返回。否則返回 0。

- `idict@-` $(x\ D\ n - D'\ v\ -1 \text{ or } D\ 0)$，在由 *Cell* D 表示的字典中查詢由有號大端序 n 位元 *Integer* x 表示的金鑰，參見**6.3**。如果找到金鑰，則從字典中刪除它並返回修改後的字典 D'、作為 *Slice* v 的相應值和 -1。否則返回未修改的字典 D 和 0。

- `if` $(x\ e -)$，執行執行權杖（即 *WordDef*）e，但僅當 *Integer* x 為非零時，參見**3.2**。

- `ifnot` $(x\ e -)$，執行執行權杖 e，但僅當 *Integer* x 為零時，參見**3.2**。

- `include` $(S -)$，從由 *String* S 給出的路徑載入並解譯 Fift 原始檔，參見**7.1**。如果檔案名稱 S 不以斜槓開頭，則使用 Fift 包含搜尋路徑（通常取自 `FIFTPATH` 環境變數或 Fift 解譯器的 `-I` 命令列引數，如果兩者都不存在則等於 `/usr/lib/fift`）來定位 S。

- `list` $(x_1\ \ldots x_n\ n - l)$，依序建構長度為 n、元素為 x_1、…、x_n 的列表 l，參見**2.16**。等同於 `null ' cons rot times`。

- `max` $(x\ y - z)$，計算兩個 *Integer*s x 和 y 的最大值 $z := \max(x, y)$。等同於 `minmax nip`。

- `min` $(x\ y - z)$，計算兩個 *Integer*s x 和 y 的最小值 $z := \min(x, y)$。等同於 `minmax drop`。

- `minmax` $(x\ y - z\ t)$，計算兩個 *Integer*s x 和 y 的最小值 $z := \min(x, y)$ 和最大值 $t := \max(x, y)$。

- `mod` $(x\ y - r := x \bmod y)$，計算 x 除以 y 的餘數 $x \bmod y = x - y \cdot \lfloor x/y \rfloor$，參見**2.4**。

- `negate` $(x - -x)$，改變 *Integer* 的符號，參見**2.4**。

- `newkeypair` $(- B\ B')$，產生新的 Ed25519 私鑰/公鑰對，並將私鑰 B 和公鑰 B' 都作為 32 位元組 *Bytes* 值返回，參見**6.1**。金鑰的品質對於測試目的來說足夠好。實際應用必須在產生 Ed25519 金鑰對之前向 OpenSSL PRNG 提供足夠的熵。

- `nil` $(- t)$，推送空 *Tuple* $t = ()$。等同於 `0 tuple`。

- `nip` $(x\ y - y)$，從頂部移除第二個堆疊項目，參見**2.5**。等同於 `swap drop`。

- `nop` (–)，什麼都不做，參見**4.6**。
- `not` $(x - -1 - x)$，計算 *Integer* 的位元補數，參見**2.4**。
- `now` (– x)，將目前 Unixtime 作為 *Integer* 返回，參見**6.1**。
- `null` (– $\perp$)，推送 *Null* 值，參見**2.16**
- `null!` $(p -)$，將 *Null* 值儲存到 *Box* p 中。等同於 `null swap !`。
- `null?` $(x - ?)$，檢查 x 是否為 *Null*，參見**2.16**。
- `or` $(x\ y - x|y)$，計算兩個 *Integer*s 的位元 OR，參見**2.4**。
- `over` $(x\ y - x\ y\ x)$，在堆疊頂部項目上方建立從頂部開始的第二個堆疊項目的複本，參見**2.5**。
- `pair` $(x\ y - t)$，建立新對 $t = (x, y)$，參見**2.15**。等同於 `2 tuple` 或 `| rot , swap ,`。
- `pfxdict!` $(v\ k\ s\ n - s'\ -1 \text{ or } s\ 0)$，將金鑰-值對 (k, v)（都由 *Slice*s 表示）加入到金鑰長度最多為 n 的前綴字典 s 中，參見**6.3**。成功時，返回修改後的字典 s' 和 -1。失敗時，返回原始字典 s 和 0。
- `pfxdict!+` $(v\ k\ s\ n - s'\ -1 \text{ or } s\ 0)$，類似於 `pfxdict!` 將金鑰-值對 (k, v) 加入前綴字典 s，但如果金鑰已存在則失敗，參見**6.3**。
- `pfxdict@` $(k\ s\ n - v\ -1 \text{ or } 0)$，在金鑰長度限制為 n 位元的前綴字典 s 中查詢金鑰 k（由 *Slice* 表示），參見**6.3**。成功時，將找到的值作為 *Slice* v 和 -1 返回。失敗時，返回 0。
- `pick` $(x_n\ \ldots x_0\ n - x_n\ \ldots x_0\ x_n)$，建立從堆疊頂部開始的第 n 個項目的複本，其中 $n \geq 0$ 也在堆疊中傳遞，參見**2.5**。特別是，`0 pick` 等同於 `dup`，`1 pick` 等同於 `over`。
- `priv>pub` $(B - B')$，計算對應於私密 Ed25519 金鑰的公鑰，參見**6.1**。公鑰 B' 和私鑰 B 都由 32 位元組 *Bytes* 值表示。
- `quit` $(\ldots -)$，退出到 Fift 解譯器的最頂層（在互動模式下不列印 `ok`）並清除堆疊，參見**2.3**。
- `ref,` $(b\ c - b')$，向 *Builder* b 附加對 *Cell* c 的參照，參見**5.2**。如果 b 已包含四個參照，則拋出例外。

- `ref@` $(s - c)$，從 *Slice* s 中提取第一個參照並返回所參照的 *Cell* c，參見**5.3**。如果沒有剩餘參照，則拋出例外。

- `ref@+` $(s - s'\ c)$，類似於 `ref@` 從 *Slice* s 中提取第一個參照，但也返回 s 的剩餘部分，參見**5.3**。

- `ref@?` $(s - c\ -1 \text{ or } 0)$，類似於 `ref@` 從 *Slice* s 中提取第一個參照，但使用旗標來指示失敗而不是拋出例外，參見**5.3**。

- `ref@?+` $(s - s'\ c\ -1 \text{ or } s\ 0)$，類似於 `ref@+`，但使用旗標來指示失敗而不是拋出例外，參見**5.3**。

- `remaining` $(s - x\ y)$，返回 *Slice* s 中剩餘的資料位元數 x 和單元參照數 y，參見**5.3**。

- `reverse` $(x_1\ ...x_n\ y_1\ ...y_m\ n\ m - x_n\ ...x_1\ y_1\ ...y_m)$，反轉位於最頂部 m 個元素正下方的 n 個堆疊項目的順序，其中 $0 \le m, n \le 255$ 都在堆疊中傳遞。

- `roll` $(x_n\ ...x_0\ n - x_{n-1}\ ...x_0\ x_n)$，旋轉頂部 n 個堆疊項目，其中 $n \ge 0$ 也在堆疊中傳遞，參見**2.5**。特別是，`1 roll` 等同於 `swap`，`2 roll` 等同於 `rot`。

- `rot` $(x\ y\ z - y\ z\ x)$，旋轉最頂部的三個堆疊項目。

- `runvm` $(...s\ c - ...x\ c')$，呼叫一個新的 TVM 實例，將目前接續 `cc` 和特殊暫存器 `c3` 都從 *Slice* s 初始化，並使用 *Cell* c 初始化特殊暫存器 `c4`（「持久資料根」，參見 [4, 1.4]），參見**6.4**。與 `runvmdict` 相反，不會將隱式零推入初始 TVM 堆疊；如果需要，可以在 s 下方明確傳遞。`c4` 的最終值在最終 Fift 堆疊頂部作為另一個 *Cell* c' 返回。透過這種方式，可以模擬檢查或修改其持久儲存的智慧合約的執行。

- `runvmcode` $(...s - ...x)$，呼叫一個新的 TVM 實例，從 *Slice* s 初始化目前接續 `cc`，從而在 TVM 中執行程式碼 s，參見**6.4**。原始 Fift 堆疊（不含 s）完整傳遞為新 TVM 實例的初始堆疊。當 TVM 終止時，其結果堆疊用作新的 Fift 堆疊，退出碼 x 推入其頂部。如果 x 為非零，表示 TVM 已被未處理的例外終止，則從頂部開始的下一個堆疊項目包含此例外的參數，x 是例外代碼。在這種情況下，所有其他項目都從堆疊中移除。

- `runvmctx` $(\ldots s\ c\ t - \ldots x\ c')$，`runvm` 的變體，也使用 *Tuple* t 初始化 `c7`（TVM 的「上下文暫存器」），參見**6.4**。
- `runvmdict` $(\ldots s - \ldots x)$，呼叫一個新的 TVM 實例，類似於 `runvmcode` 從 *Slice* s 初始化目前接續 `cc`，但也使用相同值初始化特殊暫存器 `c3`，並在開始之前將零推入初始 TVM 堆疊，參見**6.4**。在典型應用中，*Slice* s 由子例程選擇程式碼組成，該程式碼使用堆疊頂部 *Integer* 來選擇要執行的子例程，從而實現數個相互遞迴子例程的定義和執行（參見 [4, 4.6] 和 **7.8**）。等於零的選擇器對應於大型 TVM 程式中的 `main()` 子例程。
- `s,` $(b\ s - b')$，將從 *Slice* s 取得的資料位元和參照附加到 *Builder* b，參見**5.2**。
- `s>` $(s -)$，如果 *Slice* s 為非空，則拋出例外，參見**5.3**。它通常標記單元反序列化的結束，檢查是否還有未處理的資料位元或參照。
- `s>c` $(s - c)$，直接從 *Slice* s 建立 *Cell* c，參見**5.3**。等同於 `<b swap s, b>`。
- `sbitrefs` $(s - x\ y)$，返回 *Slice* s 中剩餘的資料位元數 x 和單元參照數 y，參見**5.3**。等同於 `remaining`。
- `sbits` $(s - x)$，返回 *Slice* s 中剩餘的資料位元數 x，參見**5.3**。
- `sdict!` $(v\ k\ D\ n - D'\ -1 \text{ or } D\ 0)$，將新值 v（由 *Slice* 表示）與由 *Slice* k 的前 n 個位元給定的金鑰加入到具有 n 位元金鑰的字典 D 中，並在成功時返回新字典 D' 和 -1，參見**6.3**。否則返回未更改的字典 D 和 0。
- `sdict!+` $(v\ k\ D\ n - D'\ -1 \text{ or } D\ 0)$，類似於 `sdict!` 將新金鑰-值對 (k, v) 加入字典 D，但如果金鑰已存在則失敗，返回未更改的字典 D 和 0，參見**6.3**。
- `sdict-` $(x\ D\ n - D'\ -1 \text{ or } D\ 0)$，從由 *Cell* D 表示的字典中刪除由 *Slice* x 的前 n 個資料位元給定的金鑰，參見**6.3**。如果找到金鑰，則從字典中刪除它並返回修改後的字典 D' 和 -1。否則返回未修改的字典 D 和 0。
- `sdict@` $(k\ D\ n - v\ -1 \text{ or } 0)$，在由 *Cell* 或 *Null* D 表示的字典中查詢由 *Slice* x 的前 n 個資料位元給定的金鑰，參見**6.3**。如果找到金鑰，則將相應值作為 *Slice* v 和 -1 返回。否則返回 0。

- `sdict@-` $(x\ D\ n - D'\ v\ -1 \text{ or } D\ 0)$，在由 *Cell* D 表示的字典中查詢由 *Slice* x 的前 n 個資料位元給定的金鑰，參見**6.3**。如果找到金鑰，則從字典中刪除它並返回修改後的字典 D'、作為 *Slice* v 的相應值和 -1。否則返回未修改的字典 D 和 0。
- `second` $(t - x)$，返回 *Tuple* 的第二個分量，參見**2.15**。等同於 `1 []`。
- `sgn` $(x - y)$，計算 *Integer* x 的符號（即如果 $x > 0$ 則推送 1，如果 $x < 0$ 則推送 -1，如果 $x = 0$ 則推送 0），參見**2.12**。等同於 `0 cmp`。
- `shash` $(s - B)$，透過首先將 *Slice* 轉換為單元來計算其基於 SHA256 的表示雜湊，參見**5.4**。等同於 `s>c hashB`。
- `sign` $(S\ x - S')$，如果 *Integer* x 為負數，則向 *String* S 附加減號「-」。否則保持 S 不變。
- `single` $(x - t)$，建立新單例 $t = (x)$，即一個元素的 *Tuple*。等同於 `1 tuple`。
- `skipspc` $(\ -\ )$，從當前輸入行跳過空白字元，直到找到非空白或行尾字元。
- `smca>$` $(x\ y\ z - S)$，根據旗標 z，將具有工作鏈 x（有號 32 位元 *Integer*）和工作鏈內地址 y（無號 256 位元 *Integer*）的標準 TON 智慧合約地址打包到 48 字元字串 S（地址的可讀表示）中，參見**6.2**。z 中可能的個別旗標為：+1 用於不可彈回地址，+2 用於僅限測試網地址，+4 用於 base64url 輸出而非 base64。
- `space` $(\ -\ )$，輸出單個空格。等同於 `bl emit` 或 `." "`。
- `sr,` $(b\ s - b')$，建構包含來自 *Slice* s 的所有資料和參照的新 *Cell*，並向 *Builder* b 附加對此單元的參照，參見**5.2**。等同於 `s>c ref,`。
- `srefs` $(s - x)$，返回 *Slice* s 中剩餘的單元參照數 x，參見**5.3**。
- `swap` $(x\ y - y\ x)$，交換最頂部的兩個堆疊項目，參見**2.5**。
- `ten` $(\ -\ 10)$，推送 *Integer* 常數 10。
- `third` $(t - x)$，返回 *Tuple* 的第三個分量，參見**2.15**。等同於 `2 []`。
- `times` $(e\ n -)$，如果 $n \geq 0$，則執行執行權杖（*WordDef*）e 恰好 n 次，參見**3.3**。如果 n 為負數，則拋出例外。

- `totalcsize` $(c - x\ y\ z)$，遞迴地計算以 *Cell* c 為根的單元樹中唯一單元的總數 x、資料位元 y 和單元參照 z。
- `totalssize` $(s - x\ y\ z)$，遞迴地計算以 *Slice* s 為根的單元樹中唯一單元的總數 x、資料位元 y 和單元參照 z。
- `triple` $(x\ y\ z - t)$，建立新三元組 $t = (x, y, z)$，參見**2.15**。等同於 `3 tuple`。
- `true` $(- -1)$，將 -1 推入堆疊，參見**2.11**。等同於 `-1`。
- `tuck` $(x\ y - y\ x\ y)$，等同於 `swap over`，參見**2.5**。
- `tuple` $(x_1\ ...x_n\ n - t)$，從 $n \geq 0$ 個最頂部堆疊值建立新 *Tuple* $t := (x_1, \ldots, x_n)$，參見**2.15**。等同於 `dup 1 reverse | { swap , } rot times`，但更有效率。
- `tuple?` $(t - ?)$，檢查 t 是否為 *Tuple*，並相應地返回 -1 或 0。
- `type` $(s -)$，將從堆疊頂部取得的 *String* s 列印到標準輸出，參見**2.10**。
- `u,` $(b\ x\ y - b')$，將無號 y 位元整數 x 的大端序二進位表示附加到 *Builder* b，其中 $0 \leq y \leq 256$，參見**5.2**。如果運算不可能，則拋出例外。
- `u>B` $(x\ y - B)$，將無號大端序 y 位元 *Integer* x 儲存到恰好由 $y/8$ 個位元組組成的 *Bytes* 值 B 中。整數 y 必須是 $0 \ldots 256$ 範圍內的八的倍數，參見**5.6**。
- `u@` $(s\ x - y)$，從 *Slice* s 的前 x 個位元中提取無號大端序 x 位元整數，參見**5.3**。如果 s 包含少於 x 個資料位元，則拋出例外。
- `u@+` $(s\ x - y\ s')$，類似於 `u@` 從 *Slice* s 的前 x 個位元中提取無號大端序 x 位元整數，但也返回 s 的剩餘部分，參見**5.3**。
- `u@?` $(s\ x - y\ -1 \text{ or } 0)$，類似於 `u@` 從 *Slice* 中提取無號大端序整數，但在成功時之後推送整數 -1，參見**5.3**。如果 s 中剩餘少於 x 個位元，則推送整數 0 以指示失敗。
- `u@?+` $(s\ x - y\ s'\ -1 \text{ or } s\ 0)$，從 *Slice* s 中提取無號大端序整數並計算此 *Slice* 的剩餘部分，類似於 `u@+`，但之後推送 -1 以指示成功，參見**5.3**。失敗時，推送未更改的 *Slice* s 和 0 以指示失敗。

- `udict!` $(v\ x\ D\ n - D'\ -1 \text{ or } D\ 0)$，將新值 v（由 *Slice* 表示）與由大端序無號 n 位元整數 x 給定的金鑰加入到具有 n 位元金鑰的字典 D 中，並在成功時返回新字典 D' 和 -1，參見**6.3**。否則返回未更改的字典 D 和 0。

- `udict!+` $(v\ x\ D\ n - D'\ -1 \text{ or } D\ 0)$，類似於 `udict!` 將新金鑰-值對 (x, v) 加入字典 D，但如果金鑰已存在則失敗，返回未更改的字典 D 和 0，參見**6.3**。

- `udict-` $(x\ D\ n - D'\ -1 \text{ or } D\ 0)$，從由 *Cell* D 表示的字典中刪除由無號大端序 n 位元 *Integer* x 表示的金鑰，參見**6.3**。如果找到金鑰，則從字典中刪除它並返回修改後的字典 D' 和 -1。否則返回未修改的字典 D 和 0。

- `udict@` $(x\ D\ n - v\ -1 \text{ or } 0)$，在由 *Cell* 或 *Null* D 表示的字典中查詢由無號大端序 n 位元 *Integer* x 表示的金鑰，參見**6.3**。如果找到金鑰，則將相應值作為 *Slice* v 和 -1 返回。否則返回 0。

- `udict@-` $(x\ D\ n - D'\ v\ -1 \text{ or } D\ 0)$，在由 *Cell* D 表示的字典中查詢由無號大端序 n 位元 *Integer* x 表示的金鑰，參見**6.3**。如果找到金鑰，則從字典中刪除它並返回修改後的字典 D'、作為 *Slice* v 的相應值和 -1。否則返回未修改的字典 D 和 0。

- `ufits` $(x\ y - ?)$，檢查 *Integer* x 是否為無號 y 位元整數（即對於 $0 \leq y \leq 1023$，是否為 $0 \leq x < 2^y$），並相應地返回 -1 或 0。

- `uncons` $(l - h\ t)$，將非空列表分解為其頭部和尾部，參見**2.16**。等同於 `unpair`。

- `undef?` $\langle$*word-name*$\rangle$ $(- ?)$，在執行期間檢查詞彙 $\langle$*word-name*$\rangle$ 是否未定義，並相應地返回 -1 或 0。

- `unpair` $(t - x\ y)$，解包對 $t = (x, y)$，參見**2.15**。等同於 `2 untuple`。

- `unsingle` $(t - x)$，解包單例 $t = (x)$。等同於 `1 untuple`。

- `until` $(e -)$，一個 until 迴圈，參見**3.4**：執行 *WordDef* e，然後移除堆疊頂部整數並檢查它是否為零。如果是，則透過執行 e 開始迴圈的新迭代。否則退出迴圈。

- `untriple` $(t - x\ y\ z)$，解包三元組 $t = (x, y, z)$，參見**2.15**。等同於 `3 untuple`。

- `untuple` $(t\ n - x_1\ \ldots x_n)$，返回 *Tuple* $t = (x_1, \ldots, x_n)$ 的所有分量，但僅當其長度等於 n 時，參見**2.15**。否則拋出例外。

- `variable` $(\ -\ )$，從輸入的其餘部分掃描一個以空白分隔的詞彙名稱 S，分配一個空 *Box*，並將新的普通詞彙 S 定義為常數，當呼叫時將推送新的 *Box*，參見**2.14**。等同於 `hole constant`。

- `while` $(e\ e' -)$，一個 while 迴圈，參見**3.4**：執行 *WordDef* e，然後移除並檢查堆疊頂部整數。如果為零，則退出迴圈。否則執行 *WordDef* e'，然後透過執行 e 並在之後檢查退出條件來開始新的迴圈迭代。

- `word` $(x - s)$，從當前輸入行的其餘部分解析由 Unicode 程式碼點為 x 的字元分隔的詞彙，並將結果作為 *String* 推送，參見**2.10**。例如，`bl word abracadabra type` 將列印字串「`abracadabra`」。如果 $x = 0$，則跳過前導空格，然後掃描直到當前輸入行結束。如果 $x = 32$，則在解析下一個詞彙之前跳過前導空格。

- `words` $(\ -\ )$，列印字典中當前定義的所有詞彙的名稱，參見**4.6**。

- `x.` $(x -)$，列印 *Integer* x 的十六進位表示（不含 `0x` 前綴），後面跟著一個空格。等同於 `x._ space`。

- `x._` $(x -)$，列印 *Integer* x 的十六進位表示（不含 `0x` 前綴），不包含任何空格。等同於 `(x.) type`。

- `xor` $(x\ y - x \oplus y)$，計算兩個 *Integer*s 的位元互斥 OR，參見**2.4**。

- `x{`⟨***hex-data***⟩`}` $(\ -\ s)$，建立 *Slice* s，它不包含參照且包含最多 1023 個資料位元，在 ⟨*hex-data*⟩ 中指定，參見**5.1**。更準確地說，來自 ⟨*hex-data*⟩ 的每個十六進位數字以通常的方式轉換為四個二進位數字。之後，如果 ⟨*hex-data*⟩ 的最後一個字元是底線 `_`，則從結果二進位字串中移除所有尾隨二進位零以及緊接在它們之前的二進位一（參見 [4, 1.0] 以取得更多詳細資訊）。例如，`x{6C_}` 等同於 `b{01101}`。

- `{` $(\ -\ l)$，一個活動詞彙，它將內部變數 `state` 增加一，並將新的空 *WordList* 推入堆疊，參見**4.7**。

- `|` $(\ -\ t)$，建立空 *Tuple* $t = ()$，參見**2.15**。等同於 `nil` 和 `0 tuple`。

- `|+` $(s\ s' - s'')$，串接兩個 *Slice*s s 和 s'，參見**5.1**。這意味著新 *Slice* s'' 的資料位元透過串接 s 和 s' 的資料位元獲得，s'' 的 *Cell* 參照列表類似地透過串接 s 和 s' 的相應列表建構。等同於 `<b rot s, swap s, b> <s`。

- `|_` $(s\ s' - s'')$，給定兩個 *Slice*s s 和 s'，建立新 *Slice* s''，它從 s 獲得，透過附加對包含 s' 的 *Cell* 的新參照，參見**5.1**。等同於 `<b rot s, swap s>c ref, b> <s`。

- `}` $(l - e)$，一個活動詞彙，它將 *WordList* l 轉換為 *WordDef*（執行權杖）e，從而使對 l 的所有進一步修改不可能，並將內部變數 `state` 減少一；然後推送整數 1，後面跟著`'nop`，參見**4.7**。淨效果是將建構的 *WordList* 轉換為執行權杖並將此執行權杖推入堆疊，立即或在外部區塊的執行期間。