DARK FORTH

Kodai　著

2019-09-17 版　　発行

まえがき

FORTH は不思議なプログラミング言語だ。低レイヤでの手続きを記述するための至極単純な構文 (文法・表現) をもち、実行時に自らの意味論[*1]を拡張できる。その特性ゆえに、ドメイン固有言語[*2](domain-specific language、DSL) を開発するための汎用言語として機能する。

FORTH は 1970 年頃から開発され、多くのプログラミング言語[*3]に影響を与えながら、その仕様を洗練させてきた。処理系が非常にコンパクトであり、メモリ効率が良いため、現在も組み込み系やロボット制御において活用されている。

本書では、すでに何かしらの言語を用いてプログラミングを経験している人に向けて、FORTH による「スタック指向」プログラミングの基礎やメタプログラミングの初歩について解説していく。

FORTH の多彩な意味論と計算機の織りなす世界に足を踏み入れるあなたに、一体何が待ち受けているのだろう。本書が、あなたにとって良い旅の道しるべとなることを願っている。

FORTH と共にあらんことを。

[*1] ここでは、プログラムがどのような動作・結果を持つかを定式化したものを指す。

[*2] 特定のタスク向けに設計されたコンピュータ言語

[*3] 日本語プログラミング言語「Mind」, オブジェクト指向言語「Mops」「Factor」など

目次

v

目次

第 1 章

はじめに

この章では、本書の対象読者と目的について説明した後、処理系のインストールなどの準備について述べる。最後に FORTH の略史や役立つリファレンスを紹介する。

1

1.1 対象の読者

FORTH に触れたことはないが、逐次実行・条件分岐・繰り返しと言った手続き型プログラミングの基礎的な概念をある程度理解している人を対象としている。また、FORTH における独特な用語はできる限り説明した上で導入するようにしているが、配列・スタックといった基本的なデータ構造についての説明は省いている。

1.2 本書の目的

本書での具体的な目的は「FORTH の特徴的な言語機能とその活用方法を理解して、仕様書等を参照しながら探求していく土台をつくること」である。この目的のために、あえて FORTH の算術・文字列操作などの細かな API には深入りせず、よりコアな言語機能の説明を優先している。その関

係上、はじめは辞書的な読み方ではなく連続したチュートリアルとして、先頭の章から順番に読み進めることを推奨する。

1.3　Gforth のインストール作業

本書では、FORTH 処理系のひとつである GNU forth (以降、Gforth) を使用する。著者が使用している Gforth のバージョンは `0.7.9` である。`http://gforth.org` (図 1.1) でインストール方法に関する情報やチュートリアル (英語) が提供されている。

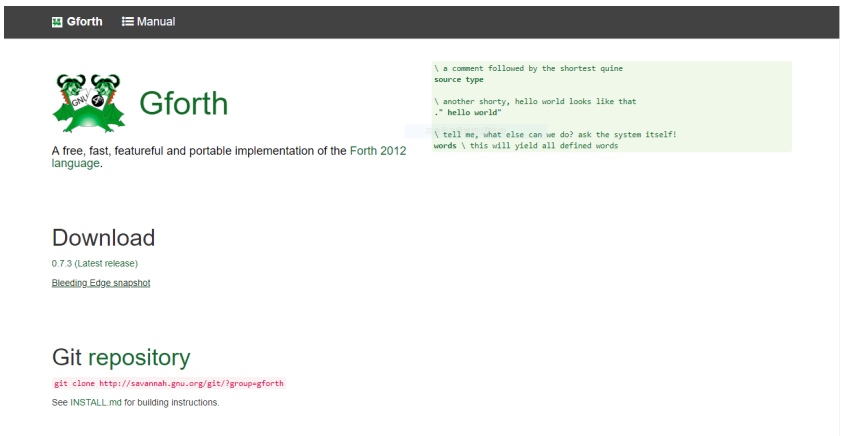

図 1.1: Gforth 公式サイト

Windows

`http://www.complang.tuwien.ac.at/forth/gforth/` から、「Win32 self installing」と記載されたインストーラを入手し、それを実行して指示に従ってインストールする。

macOS

Homebrew 経由で導入できる。

```
brew install gforth
```

Ubuntu

apt コマンドでインストールできる。

```
sudo apt install gforth
```

Arch linux

AUR [*1]で提供されているので、yay 等を利用してインストールする。

[*1] パッケージ詳細ページ: https://aur.archlinux.org/packages/gforth/

```
yay -S gforth
```

1.4 FORTH の歴史と標準

FORTH は、1958 年からチャールズ・ムーアが個人開発していたプログラミングシステムから考案された。1968 年に、このソフトウェアをミニコンピュータ上で FORTRAN で実装したものが FORTH の原型とされている。FORTH が他のプログラマに対して最初に公開されたのは 1970 年代で、アメリカ国立天文台 (NARO) にいたエリザベス・ラザーによって始められたものである。NARO にいた 2 人は制御用ソフト開発において FORTH を完成させ、後に FORTH, Inc. を設立した (1973 年)。ムーアの仕事の関係上、言語開発の初期段階では移植性の高さが求められていた。当時の FORTH は現在のスクリプト言語のような立ち位置ではなく、FORTH 処理系自体が軽量 OS として用いられていた。そういった処理系のひとつである MacFORTH は、アップル Macintosh の最初の常駐開発システムとして採用されていた。

その後も FORTH は様々な環境に移植され、1979 年に言語仕様の標準化が始まった。1979 年に成分化された FORTH-79、1983 年に成分化された FORTH-83 は「事実上の標準[*2]」である。これらの標準は 1994 年に ANSI によって統合され、ANS Forth と名付けられた。1979 年には ISO/IEC においても標準化されている[*3]。

本書で扱う Gforth は Forth 2012 Standard という規格に沿って現在も盛んに開発されており、Forth 2012 Standard は Forth 200x 標準化委員会

[*2] デファクトスタンダードとも呼ばれる。標準化機関等が定めた規格ではなく、市場における競争や広く採用された結果として「事実上標準化した基準」を指している。

[*3] ISO/IEC 15145:1997 https://www.iso.org/standard/26479.html

によって策定されている。

1.5　役立つリファレンス

Gforth Manual

`http://gforth.org/manual/`

FORTH と Gforth の仕様を学べるマニュアル

Forth Standard

`https://forth-standard.org/`

Forth 2012 Standard に関する情報が共有され、さらなる技術仕様 (プロポーザル) がまとめられている。

Starting FORTH

`https://forth.com/starting-forth`

FORTH 入門者向けに、基本からコンパイラ変形まで網羅的に解説している本

iMops-forth @Wiki

`https://w.atwiki.jp/imops-forth/`

FORTH からの派生言語である「iMops」について、様々な記事が公開されている Wiki

第 2 章

FORTH 序論

この章では、FORTH の学習を始める上で把握しておくべき周辺の基礎概念と、FORTH の中核をなす諸概念について扱っていく。

7

2.1　ワードとインタプリタ

FORTH のシンタックスは、要は「文字列を空白文字で区切って並べただけ」だ。この極限まで単純化されたシンタックスが、後述する多彩な意味論と相まって、高い融通性を発揮する。

これから少しの間、Gforth を「電卓」として使っていく。Gforth を起動し、リスト 2.1 を書き写してみよう。入力が終わったら、改行してみてほしい。

リスト 2.1: はじめての FORTH プログラム

```
6 3 4 + * .
```

正しく動いていれば、たった今書き込んだコードのすぐ左に `42 ok 0` と表示されたはずだ。`ok` と出力されたのは、入力されたコードを Gforth が

すべて解釈実行し終えたからだ。Gforth は、この瞬間にどんな流れで、どうリスト 2.1 を解釈したのだろうか。

Gforth は手続きに必要なパラメータをスタック上で管理しながら、ソースコードを先頭から一直線に解釈する。リスト 2.1 を構成している文字列は 6, 3, 4, +, *, . の 6 つだ。

- 最初の 3 つは 32 ビット符号付き整数値として解釈され、それぞれ順番にスタックにプッシュされる。
- あと 3 つの、数値として認識できない文字列は「**ワード** (word)」として解釈される。

ワードは、簡単に言えば手続きに名前をつけたものだ。これは、他の言語における関数やサブルーチンのようなものとして理解される。

FORTH 処理系がインタプリタとして動作しているときにワードを認識すると、対応する**解釈時意味論** (interpretation semantics) に従って呼び出しが発生する。後述するが、「〜時意味論」のようにワードには複数の意味論が存在し、呼び出し時の処理系の状態によって、ワードの挙動は異なる(ように作為的にワードを定義できる)。

FORTH 処理系はデフォルトではインタプリタとして動作しており、この段階では解釈時意味論だけ考慮していれば問題ない。

リスト 2.1 でのインタプリタの挙動は以下のようになる。

1. 6 3 4 まで解釈した直後には、スタックにはそのまま底から 6, 3, 4 が積まれている。
2. + を読み込んだインタプリタは、ワード名とその定義を紐付けて保存している**辞書** (dictionary) からその定義を見つけ出し、呼び出す。+ ワードはスタックの一番上に積まれている 2 つの数 3, 4 をポップし、和 7 をプッシュする。
3. * を読み込んだインタプリタは同様に定義を見つけ出して呼び出す。* ワードはスタックの一番上に積まれている 2 つの数 6, 7 をポップ

し、積 42 をプッシュする。

4. . ワードの解釈時意味論は「スタックのトップの要素をポップして出力すること」であるから、スタックは空になり、 42 が出力される。

つまり、リスト 2.1 は $(3+4)\times 6$ を評価させて、結果を出力させるプログラムだったわけだ。FORTH において算術式の評価を表現すると、自然に「被演算子を先に並べて、最後に演算子を書く記法」になる。この記法は一般的には**逆ポーランド記法**と呼ばれており、スタックとの相性が良い。

今後は、特に指定のない限り、Gforth を起動した直後の状態を想定してサンプルコードを扱っていく。BYE[*1] ワードを呼び出せば Gforth を終了できる。

スタックの要素数と各要素の値を出力する .S ワードと、改行コードを出力する CR ワードも使用して、リスト 2.1 でのスタックの変化を探ってみよう。(リスト 2.2)

9

リスト 2.2: スタックの変化を探る

```
CR .S 6 3 4 CR .S + CR .S * CR .S
```

実行結果

```
<0>
<3> 6 3 4
<2> 6 7
<1> 42
```

スタック上で正しく計算されていることがわかる。

[*1] アルファベットの大文字小文字の区別はないため、bye と入力しても構わない。

2.2 基本的な算術

+ ワードや * ワードと同様に、以下のワードを用いて基本的な算術を行う。

- (マイナス)
: 減算

/ (スラッシュ)
: 除算

MOD
: 剰余

/MOD
: 「剰余」「商」を順にプッシュする

NEGATE
: 符号を反転する

ABS
: 絶対値

リスト 2.3: サンプル

```
23 8 - 3 / .
7 4 MOD .
CR 7 4 /MOD .S CR
2 NEGATE .
-42 ABS .
```

実行結果

```
5 3
<2> 3 1
-2 42
```

2.3 スタック操作ワード

FORTH にはスタックの内容を操作するための組み込みワードが用意されている。その 1 つが DUP ワードだ。スタックのトップに積まれている要素を複製する。

リスト 2.4: DUP ワードの利用

```
8 DUP * .
```

リスト 2.4 を実行すると、すぐ右側に 64 ok 0 と表示される。.(ドット) ワードを呼び出すと、スタックのトップの要素をポップして出力する。つまり DUP ワードによって複製が行われスタックにはふたつの 8 が残り、それらは * ワードによってポップされ 2 数の積 64 が残る。

他にもいろいろなスタック操作用のワードが用意されている。

DROP

トップの要素を破棄する。

NIP

トップから 2 つ目の要素を破棄する。

SWAP

トップの 2 つの要素の順序を入れ替える。

OVER

トップから 2 つ目の要素をコピーして、それをプッシュする。

ROT

トップから 3 つの要素を逆順にする。

スタック上の 2 つの要素をまとめて操作するためのワードも用意されている。

2DROP

トップから 2 つの要素を破棄する。

2SWAP

スタック上の (底の方から) a b c d を、 c d a b に並び替える。

2OVER

スタック上の a b c d のうち a b をコピーしてプッシュする。プッシュ操作が完了すると a b c d a b のようになっている。

2DUP

トップから 2 つの要素をコピーして、同じ順でプッシュする。

2SWAP ワードと 2OVER ワードの説明では、スタック上の要素に識別名をつけた上で、呼び出しの前後でそれがどう変化するかを示している。こういった記法については、後の「2.7 スタック表記法」でより厳密に定義して使っていくことにする。

2.4　コメント

リスト 2.5: コメントの書き方

```
\ コメント
( コメント )
```

上のような書き方をすれば、**コメント**の部分は無視される。\ と (はそれぞれひとつのワードとして独立に定義されており、これらを呼び出すために、後ろに空白文字が必要になる。\ ワードは、後続するソースコードを行末まで読み飛ばす。(ワードは、後続するソースコードを) まで読み飛ばす。この 2 つのワードのように、後ろに続いているソースコードをパースして引数に取ることが可能なワードも存在している。後述するが、そのようなワードをユーザが定義することも可能である。

2.5 セル

FORTH で扱われるデータのサイズの最小単位は**セル** (cell) である。スタックに積まれる要素のサイズもすべて 1 セルとなっている。1 セルあたりの実際のサイズは処理系依存だが、標準仕様に含まれている CELLS ワードを用いて、使用中の処理系でのサイズを知ることができる。

リスト 2.6: 1 セルあたりのバイト数を調べる

```
1 CELLS .
```

CELLS ワードは、スタック上の整数値 1 つをポップし、それに 1 セルあたりのバイト数を掛けてプッシュする。例えば 64 ビット OS 向けの Gforth では 1 セル = 8 バイトなので、リスト 2.6 を実行すると 8 と出力されるはずだ。

2.6 文字列・印字

FORTH においては、文字列は「書き込み先の先頭アドレス」「文字数」の 2 つの値で表現される。メモリ上に文字列を書き込んでこの 2 つをスタックに積む役割をもつのが S" ワードだ。書き込まれる文字列はソースコード上に直接記述し、 " をデリミタとしている。スタック上の 2 値をポップして文字列として出力するには TYPE ワードを用いる。

リスト 2.7: S" ワードの動作

```
S" HELLO WORLD" TYPE
```

実行結果

```
HELLO WORLD
```

ダブルクォートやタブ文字、バックスペース文字といった文字も扱いたい場合、エスケープシーケンスに対応している S\" ワードを用いる。

リスト 2.8: S\" ワードの動作とエスケープシーケンス

```
S\" \l\"Hello,\lFORTH\x21\"" TYPE
```

実行結果

```
"Hello,
FORTH!"
```

扱えるエスケープシーケンスについては 表 2.1 にまとめてある。

表 2.1: エスケープシーケンス一覧

記号	意味
\a	アラート
\b	バックスペース
\e	エスケープ
\f	改ページ
\l	改行 (LF)
\m	改行 (CR/LF)
\q または \"	ダブルクォート
\x[16 進数]	ASCII 文字 (16 進数で ASCII コードを指定)
\\	バックスラッシュ

15

単にソースコード中の文字列を切り出して出力するだけの ." ワードもある。いわゆる printf デバッグに用いられることが多い。(リスト 2.9)

リスト 2.9: ." ワードの動作

```
." DEBUG"
```

実行結果

```
DEBUG
```

実際にメモリ上に文字列が格納されていることを確認するには、DUMP ワードが便利だ。「バイト数」と「先頭アドレス」をスタックから順にポップし、その領域の内容をダンプしてくれる。出力形式については規定されておらず処理系依存となる。以下に、筆者の環境でテストした結果を載せて

おく。

リスト 2.10: 文字列 ABC を書き込み、ダンプさせるサンプル

```
S" ABC" 2DUP DUMP TYPE
```

実行結果

```
60008EB40: 41 42 43                    -                              ABC
ABC
```

3 バイトの領域に、A, B, C それぞれの ASCII コードが書き込まれていることがわかる。

2.7 スタック表記法

ワードを呼び出した際のスタックへの影響を記述するための記法を紹介する。これは、 FORTH のリファレンス等でワードの挙動を説明する際に多用される。基本的には、以下のようなフォーマットになっている。

リスト 2.11: スタック表記法

```
( before -- after )
```

`before`, `after` の部分には、スペース区切りで**シンボル**が 0 個以上並ぶ。シンボルは、FORTH におけるデータ型や、その値のサイズを表現するための記号である。Forth 2012 Standard において用いられているシンボルについて、表 2.2 にまとめておく。現段階では扱っていない言語機能に関するシンボルも含まれているが、今後参照する資料として一旦読み流しておいてほ

しい。

表 2.2: スタック表記法で用いられるシンボル

シンボル	データ型	サイズ
flag	フラグ	1 セル
char	文字	1 セル
n	符号付き整数	1 セル
u	符号無し整数	1 セル
x	指定のない 1 セル分のデータ	1 セル
xt	エグゼキューショントークン	1 セル
addr	アドレス	1 セル
a-addr	アラインメントされた領域のアドレス	1 セル
c-addr	文字用にアライメントされた領域のアドレス	1 セル
d	2 セル符号付き整数	2 セル
ud	2 セル符号無し整数	2 セル
i * x, j * x, k * x	任意	0 セル以上

2.8　比較・論理演算

先にスタックに積んだ 2 つの数値をポップして比較・論理演算を行うためのワードが用意されている。ただし、FORTH において偽は 0 、真は 0 以外の数で表されるので注意が必要だ。真を表す値としては、 -1 がよく用いられる[*2]。

=

(x1 x2 -- flag)

2 数が等しいなら真、等しくないなら偽を返す

<>

[*2] -1 をビット反転すると 0 になる。0 をビット反転すると -1 になる。

`( x1 x2 -- flag )`

2 数が等しくないなら真、等しければ偽を返す

<, >, <=, >=

`( n1 n2 -- flag )`

大小比較

AND, OR, XOR

`( x1 x2 -- x3 )`

ビットごとの論理積 / 論理和 / 排他的論理和を求める

INVERT

`( x1 -- x2 )`

ビットを反転する

2.9 ワード定義の基本

ユーザがワード定義する、つまり辞書に新たな項目を追加する方法を紹介する。基本的には、: ワードを使用する。

リスト 2.12: ワード定義

```
: square ( n -- n ) DUP * ;
```

リスト 2.12 では、新たに square というワードを定義している。(n -- n) の部分はコメントとして扱われ、プログラマに対して square ワードの挙動をスタック表記法で示す「ドキュメント」としての役割を担っている。

リスト 2.13: square ワードの利用

```
8 square .
```

square ワードの解釈時意味論は「スタックのトップの要素を 1 度複製し、上 2 つの要素を掛け合わせること」となるため、実際に square ワードを呼び出す リスト 2.13 を実行すると 64 が出力される。

: ワードは、呼び出されるとまず後続するソースコードを空白文字の直前まで読み込む。それを名前として保存した後、処理系を**コンパイル状態** (compilation state) に移行させる。コンパイル状態のときにワードを認識すると、先述した解釈時意味論ではなく**コンパイル時意味論** (compilation semantics) に従って呼び出しが発生する。

一部のワードを除いて、多くのワードのコンパイル時意味論は「自らの解釈時意味論を定義に追加すること」である。ただし、解釈時意味論と「他のワードの呼び出し中に間接的に呼び出されたときの挙動」が異なる場合には、後者を特に**実行時意味論** (execution semantics) と呼ぶ。

2.10　ソースファイルの利用

Gforth でソースファイルを指定して実行する

ソースファイルに書き込まれたソースコードを Gforth で実行できる。例えば file1.4th というソースファイルと、file2.4th というソースファイルを順に実行したい場合、次のようなコマンドとなる。

```
gforth file1.4th file2.4th
```

■コラム: ソースファイルの拡張子

一般的には .fs, .fth, .4th, .forth などが用いられる。筆者は

.4th を用いることが多い。(.fs だと F# のソースファイルの拡張子と被ってしまう)

実行中に別のソースファイルを読み込む

先にソースファイルを文字列で指定し、INCLUDED ワード（ i * x c-addr u -- j * x ）を呼び出すことで、そのソースファイルを開いて全体を解釈実行させられる。

リスト 2.14: file.4th を読み込む

```
S" file.4th" INCLUDED
```

第 3 章

制御構造

通常「条件分岐」や「繰り返し」はコンパイル状態でのスタックを活用して実現される。この章では、コンパイル状態で利用できる制御フロー関連の機能に触れていく。

3.1 条件分岐

条件分岐を作り出すためには IF, ELSE, THEN ワードを用いる。他のプログラミング言語の具象構文と見比べて語順が異なっている。この 3 つのワードはコンパイル状態でしか呼び出すことができず、解釈時意味論は未定義である。その代わり、それぞれ実行時意味論が定義されている。

リスト 3.1 を見てほしい。これは、スタックに積まれた 1 つの数が 2 の倍数の場合に `Multiple of 2` と出力するワード `word3-1` の定義だ。

リスト 3.1: IF THEN による条件分岐

```
: word3-1  ( n -- )  2 MOD 0 = IF ." Multiple of 2" THEN ;
```

IF, THEN ワードの組が特別扱いされているわけではなく、それぞれ独立

したワードとして定義されている。IF ワードは「先にスタックに積まれた 1 つの数を条件として、それが偽だった場合のジャンプ先を定めないまま、真だった場合のコンパイルを開始する」という役割を持つ。そして THEN ワードは「未解決だったジャンプ先を解決する」という役割を持つ。リスト 3.2 では、実際に word3-1 ワードを解釈させて挙動を確認している。1 行目を解釈させても何も主力されないが、2 行目を解釈させると Multiple of 2 と出力されたはずだ。

リスト 3.2: word3-1 の動作確認

```
3 word3-1
4 word3-1
```


IF, THEN ワードは、「コンパイル状態で呼び出されるとコンパイラに働きかけて、実際に定義されるワードの挙動に特別な影響を与える」ワードである。このようなワードによって実現される意味論を**走行時意味論** (runtime semantics) という。

ELSE ワードは「未解決だった『条件が偽の場合のジャンプ先』を解決し、『条件が真の状態で ELSE ワードに達したときのジャンプ先』を未解決にしたままコンパイルを続行する」という役割を持つ。リスト 3.3 で定義している word3-2 ワードは、スタック上の 2 数のうち小さい方をスタックに残して、もう一方をスタックから取り除く。

リスト 3.3: ELSE を用いた条件分岐

```
: word3-2  ( n1 n2 -- n3 )  2DUP < IF DROP ELSE NIP THEN ;
7 10 word3-2 .
10 7 word3-2 .
```

スタック上の値の違いに応じた多方向への分岐を実現するには、CASE,

OF, ENDOF, ENDCASE ワードを用いる。リスト 3.4 で定義している word3-3 ワードでは、先にスタックに積まれていた数が 1, 2, 3 の場合に、それぞれ異なる文字列を出力する。それ以外の値の場合には、Other number:[数値] の形式で出力する。

リスト 3.4: CASE による多方向への分岐

```
: word3-3  ( n1 n2 -- n3 )
  CASE
    1 OF ." It is one." ENDOF
    2 OF ." It is two." ENDOF
    3 OF ." It is three." ENDOF
    ." Other number: " DUP .
  ENDCASE
;
1 word3-3
2 word3-3
3 word3-3
4 word3-3
```


CASE ワードの呼び出しまでにスタックのトップに積まれた値 (以下、「入力値」) が、OF ワードの前に積んだ値 (以下、「基準値」) と一致した場合には、その後の ENDOF ワードまでを実行して ENDCASE ワードまでジャンプする。OF, ENDOF ワードを用いないことで、それまで挙げたどの基準値にも一致しなかった場合の処理を記述できる。

OF ワードの走行時意味論は「直前に積まれた入力値と基準値のうち、基準値だけをポップすること」であり、ENDCASE ワードの走行時意味論は「入力値をポップして実行を続けること」である。そのため、スタック上の入力値を消費する操作をする場合には、入力値を複製しておくことが必要となる。word3-3 ワードでも、Other number: を出力した後に入力値を出力するために、予め DUP ワードを呼び出している。

3.2 繰り返し

ループ

ここでは回数を指定してループを発生させる方法についてのみ述べる。確定ループを作り出すには DO, LOOP ワードを用いる。DO (n1 n2 --) ワードの走行時意味論としては、スタックから 2 数を取り出し、順番にループの「境界値 n1」「初期値 n2」とする。ループカウンタの値は I ワードを用いて取得する。ネストする場合は J ワードを用いて内側のループでのカウンタの値を取得できるが、そもそも 1 つのワード定義のなかでネストした制御構造をまとめて記述することは推奨しない。可読性に欠けるからだ。

これから、確定ループを扱う練習としてインクリメンタルに「スタックに積まれた数の階乗を求める」ワードを定義していく。まずは「1 から 10 までの数を出力する」1to10 ワードと、「0 から 9 までの数を出力する」0to9 ワードを リスト 3.5 で定義してみる。

2
4

リスト 3.5: 1 から 10 までの数を出力する

```
: 1to10  ( -- )
  11 1 DO I . LOOP
;

: 0to9 ( -- )
  10 0 DO I . LOOP
;
```

LOOP ワードに達してループカウンタを更新したときに、境界値と初期値が一致していればループは終了する。そのため 10 回確定でループさせたい場合は「境界値 - 初期値 = 10」となるように境界値と初期値を設定すればいい。

次は 1, 2, 3 の階乗を求める fact-1, fact-2, fact-3 ワードを定義する。

リスト 3.6: 1, 2, 3 の階乗を求める

```
: fact-1  ( -- n )
  1 2 1 DO I * LOOP
;
: fact-2  ( -- n )
  1 3 1 DO I * LOOP
;
: fact-3  ( -- n )
  1 4 1 DO I * LOOP
;
fact-1 .
fact-2 .
fact-3 .
```

25

実行結果

```
1 2 6
```

n の階乗を求めるために、1 から n + 1 までの確定ループのなかで、最初に積んだ 1 に対して毎回ループカウンタを掛けている。先にスタックのトップに積まれた数を求めるには、それがここでの n として扱われるようにすればいい。それを実装したものが リスト 3.7 の fact-n ワードである。DO ワードを呼び出す直前に、スタックに「ループカウンタを掛けられる数」「境界値」「初期値」を順に積んでおく必要があるので、SWAP ワードを用いて順序を直している。

リスト 3.7: 1 以上の数の階乗を求める

```
: fact-n  ( n1 -- n2 )
  1 SWAP 1 + 1 DO I * LOOP
;
```

しかし fact-n ワードには欠陥がある。0 以下の数をスタックのトップに積んだ状態で呼び出すと無限ループが発生する。この問題は、DO ループのカウンタがずっと境界値に一致しないことに起因する。

これを解決するには DO ワードの代わりに ?DO ワードを使用する。?DO ワードは基本的には DO ワードと同じ挙動だが、初期値が境界値以下だった場合 1 度もループしない、という特徴がある。これを使って、 0 以下の数値の入力に対しては 1 を返すように定義すればいい。最終的に、「スタックに積まれた数の階乗を求める」ワード fact の定義はリスト 3.8 のようになった。

リスト 3.8: 階乗を求める

```
: fact  ( n1 -- n2 )
  1 SWAP 1 + 1 ?DO I * LOOP
;
0 fact .
1 fact .
2 fact .
3 fact .
4 fact .
```

実行結果

```
1 1 2 6 24
```

ループからの脱出

「ワードの呼び出しを終了する」という走行時意味論を持った EXIT ワードを用いると、どこからでも定義内容の最後までジャンプできる。IF ワードと EXIT ワードを組み合わせることで、特定の条件を満たさない場合に呼び出しを終了し、満たしていた場合は続行するという挙動を定義できる。これは可読性を向上させる上で有効な手段であり、一般的には**早期リターン**と呼ばれている。早期リターンを用いた一例を リスト 3.9 に示す。

リスト 3.9: 早期リターン

```
: word-for-pos  ( n -- )
  DUP 0 <
  IF
    DROP EXIT
  THEN
  ." positive number: " . CR
;

-2 word-for-pos
-1 word-for-pos
1 word-for-pos
2 word-for-pos
```

実行結果

```
positive number: 1
positive number: 2
```

EXIT ワードの注意点として、「DO, ?DO, LOOP ワードによるループからの脱出のために EXIT ワード単体で使用してはいけない」ということが挙げら

れる。その理由については 3.4 節「リターンスタック」で述べる。代替策として UNLOOP ワードが用意されており、EXIT ワードを呼び出す直前に呼び出すことでループから正常に脱出できる。ループがネストしている場合は、その個数分だけ UNLOOP ワードを呼び出す必要がある。

リスト 3.10: UNLOOP ワードの使用例

```
: less-than-4  ( n -- )
  CR 1 + 1
  DO
    I DUP ." Checked: " . CR 4 =
    IF
      ." Equal to or greater than 4" CR UNLOOP EXIT
    THEN
  LOOP
  ." Less than 4" CR
;

2 less-than-4
10 less-than-4
```


実行結果

```
Checked: 1
Checked: 2
Less than 4

Checked: 1
Checked: 2
Checked: 3
Checked: 4
Equal to or greater than 4
```

リスト 3.10 で定義している less-than-4 ワードでは、スタックから取

り出した数に応じて DO ループを発生させ、ループカウンタが 4 まで増加するとその時点で UNLOOP EXIT によってループを脱出する。

3.3 再帰

RECURSE ワードを用いると、呼び出されている最中に自らの呼び出しを行うようなワードを定義できる。リスト 3.11 では、RECURSE ワードを用いて先述した階乗関数を再定義している。当然 RECURSE ワードは : によるワード定義の最中にしか使うことができない。

リスト 3.11: RECURSE ワードを用いて階乗関数を定義する

```
: fact  ( n1 -- n2 )
  DUP 0 <=
  IF
    DROP 1
  ELSE
    DUP 1 - RECURSE *
  THEN
;
```


3.4 リターンスタック

今まで「スタックは 1 つだけ」のように説明してきたが、実はもう 1 つある。**リターンスタック**という、「ワードの呼び出しが完了した後にジャンプする先」を記録するためのスタックである。今まで扱ってきたスタックは、今後リターンスタックと区別して**データスタック**と呼ぶことにする。

実は FORTH では、このリターンスタックの内容を操作することが可能である。以下ではリターンスタックの操作に関する組み込みワードである >R, R>, R@ ワードを紹介しているが、データスタックのスタック表記に加え

て、(R: before -- after) の形式でリターンスタックのスタック表記も記述している。

>R

解釈時: 未定義動作
実行時:(x --) (R: -- x)
データスタックから 1 つ要素をポップし、それをリターンスタックにプッシュする。

R>

解釈時: 未定義動作
実行時:(-- x) (R: x --)
リターンスタックから 1 つ要素をポップし、それをデータスタックにプッシュする。

R@

解釈時: 未定義動作
実行時:(-- x) (R: x -- x)
リターンスタックの一番上の要素をコピーして、データスタックにプッシュする。

リターンスタックは「1 ワード定義内に限って利用可能な、値の仮置場」として使われることを想定して、アクセス可能になっている。裏を返せば、それ以外の使い方をした場合の挙動は保証されていない。リスト 3.12 では、FORTH における禁忌を犯している。

リスト 3.12: 2 重になったワード呼び出しの順序を変更する

```
: don't-do  ( -- ) ( R: n1 n2 -- n2 n1 )
  R> R> SWAP >R >R
;

: foo  ( -- )  CR ." FOO-BEGIN" don't-do CR ." FOO-END" ;
```

```
: bar  ( -- )  CR ." BAR-BEGIN" foo CR ." BAR-END" ;

bar
```

実行結果

```
BAR-BEGIN
FOO-BEGIN
BAR-END
FOO-END
```


呼び出し順序が変わっていることがわかる。まず bar ワードの呼び出しが始まった直後に、リターンスタックには「bar ワードの呼び出し後のジャンプ先アドレス」が積まれる。その後 foo ワードの呼び出しが始まり、「foo ワードの呼び出し後のジャンプ先アドレス」が積まれる。同様に don't-do ワードの呼び出し後のジャンプ先アドレスが積まれるわけだが、リターンスタックが操作されてトップの 2 つの要素が入れ替わる。つまり don't-do ワードの呼び出しが終わると、もともと foo ワードのために積まれていたアドレスにジャンプしてしまう。その結果 CR ." BAR-END" が実行され、もともと don't-do ワードのために積まれていたアドレスにジャンプし、CR ." FOO-END" が実行される。

リターンスタックを「値の仮置場」として安全に扱うには、「1 ワード定義中での >R ワードの使用回数と R> ワードの使用回数を一致させること」を意識する必要がある。リスト 3.13 では、この考え方に従って 2SWAP ワードを実装している。

リスト 3.13: リターンスタックの安全な使い方

```
: 2swap' ( n1 n2 n3 n4 -- n3 n4 n1 n2 )
  ROT >R ROT R>
;
```

3.2 節「繰り返し」で挙げた「DO ループから抜け出すときに EXIT ワードを単体で使ってはいけない」という問題は、まさにこの節で扱っているリターンスタックに起因する。Forth Standard での DO ワードの解説[*1]にある通り、DO ワードは走行時意味論としてリターンスタックに値をひとつプッシュする。そして LOOP ワードの走行時意味論では、リターンスタックからひとつ値をポップする。DO ワードがすでに呼び出されていて LOOP ワードがまだ呼び出されていないという状況で EXIT すると、リターンスタックに (元々は DO ワードがプッシュした) 余分な要素が残ってしまい、処理系がクラッシュしてしまう。

[*1] https://forth-standard.org/standard/core/DO

第 4 章

データ・辞書

ここまではスタック上のデータの操作や制御構造について触れてきたが、この章では「データを保存する手段」や「呼び出し時の挙動を追加する手段」に関して解説していく。

4.1 辞書

辞書は「すでに定義されたワードのデータとコードの置き場所」である。実際には辞書はメモリ中にロードされており、FORTH インタプリタ・コンパイラに関わるコアなワードの定義も含めて、そこに書き込まれている。辞書に名前とコードを対にして登録すれば、当然それ以降は名前をワードとして扱える。辞書は、定数や変数を管理するグローバル環境としても利用される。

ここからは、実際に定数や変数、ワードを辞書に書き込ませてみながら、辞書の扱い方や意義について探っていくことにしよう。

定数

定数は、**呼び出されるとその値をデータスタックに積むワード**として表現される。そのようなワードを定義するには、CONSTANT (x --) ワードを用いる。先に初期値をデータスタックにプッシュしておいて、直後のソースコード中に定数名を記述する。CONSTANT ワードには、後ろの 1 単語を切り出すパーサとしての挙動もあわせて定義されている。

リスト 4.1: 定数定義

```
100 CONSTANT foo
foo 2 * .
```


実行結果

```
200
```

変数

変数は、**呼び出されるとデータスタックに参照先アドレスを積むワード**として表現される。そのようなワードを定義するには、VALUE (x --) ワードを用いる。CONSTANT ワードと同様に、データスタックのトップの要素を初期値、直後のソースコード中の 1 単語を変数名として変数を宣言・初期化する。「データスタックのトップの要素をアドレスとしてポップして、書き込まれている値をプッシュする」@ (a-addr -- x) ワードを通じて、代入されている値を取得できる。

リスト 4.2 では、270 を初期値として変数 bar を宣言・初期化している。

代入には !(x a-addr --) ワードを用いる。! ワードは、a-addr で指定したメモリ番地に x で指定した値を書き込む。

リスト 4.2: VALUE ワードによる変数宣言と初期化

```
270 VALUE bar
bar @ .   \ 変数 bar の値を出力
42 bar !  \ 変数 bar に 42 を代入
bar @ .
```

実行結果

```
270 42
```


変数の宣言だけをして、初期化を後回しにするには VARIABLE ワードを用いる。VALUE ワードと異なるのは、先にスタックに初期値を積んでおく必要がない点だ。

```
VARIABLE baz
80 baz !
baz @ .
```

CREATE と辞書書き込み

CONSTANT, VARIABLE, VALUE ワードとは異なる、単に「未使用のメモリ空間の先頭アドレスを返すワードを定義する」だけの低レベルな機能も提供されている。その機能を提供するのが CREATE ワードだ。CREATE ワードは、後続する 1 単語をパースして、その名前で実行時意味論が「CREATE の呼び

出し時点での空きデータ域の先頭アドレスを返す」のワードを定義する。実用上は、未使用のメモリ空間の先頭から、指定した容量を確保する ALLOT (n --) ワードと併用される。慣用的には、リスト 4.4 のように CREATE ワードの呼び出しと ALLOT ワードの呼び出しを連続させることが多い。参照先アドレスを得るためのワードの名前と、実際に確保されているメモリ領域のサイズが見てすぐに把握できるからだ。

リスト 4.4: CREATE, ALLOT ワードを用いて 1 セル変数を生成する

```
CREATE var 1 CELLS ALLOT
500 var !
var 1 CELLS DUMP
var @ .
```

実行結果

```
6FFFF87DDD8: F4 01 00 00  00 00 00 00 -
500
```

空きデータ域の先頭アドレスを取得するだけであれば HERE (-- addr) ワードで呼び出せば可能であるから、リスト 4.5 では実際に CREATE された名前に対する値と、HERE ワードの返す値が等しいことを確かめている。

リスト 4.5: CREATE ワードによって定義されるワードの動作確認

```
CREATE foo
foo . CR
HERE .
```

CREATE ワードの定義したワードが返すアドレスに対して、参照先の値を

更新するには、先述した ! ワードを用いる方法と、新たに登場する , (カンマ) ワードを用いる方法がある。リスト 4.6 では、後者の方法で参照先の値を更新している。

```
CREATE foo
74 ,
foo @ .
```

実行結果

```
74
```


, (x --) ワードは、空きデータ域の先頭部分に x で指定した 1 セル分のデータを書き込み、HERE ワードの返す値を 1 セル分増加させる。つまり x の書き込み先は , ワードを呼び出すたびにズレていくため、2 回以上参照先の値を更新するために使うことはできない。

配列の表現

配列は「要素の値が連続して書き込まれたメモリ領域の先頭アドレス」として表現される。要素数をメモリ上に保持する機構は備わってない。リスト 4.7 では要素が 3 つの配列 array を定義して値を書き込んでいる。

リスト 4.7: 配列の生成と要素の取得

```
CREATE array 3 CELLS ALLOT

1 array !
2 array 1 CELLS + !
```

```
3 array 2 CELLS + !

array 3 CELLS DUMP

array @
array 1 CELLS + @
array 2 CELLS + @
* * .
```

実行結果

```
6FFFF87DDD8: 01 00 00 00  00 00 00 00 - 02 00 00 00  00 00 00 00
6FFFF87DDE8: 03 00 00 00  00 00 00 00 -
6
```


4.2　匿名ワードと XT

XT と間接的呼び出し

定義済みのワードをデータとして扱い、識別するためのフォーマットが **XT** (eXecution Token) だ。’ (ティック) ワードを用いて既定義ワードの XT を取得できる。XT をもとに実際に呼び出しを発生させるには EXECUTE (i * x xt -- j * x) ワードを用いる。

リスト 4.8: XT の取得と間接的な呼び出し

```
: hello  ( -- )  ." HELLO!" ;

’ hello EXECUTE
```

実行結果

```
HELLO!
```

’ ワードの実行時意味論は「後続するソースコードから 1 単語をパースして、その名前がついたワードの XT をプッシュすること」であるから、リスト 4.9 のような高等的なワードを定義することが可能である。

リスト 4.9: 間接的な呼び出しを複数回発生させる

```
: hello ." HELLO!" ;

: 2times  ( xt -- i * x )
  DUP EXECUTE EXECUTE
;

: ntimes  ( xt n -- i * x )
  0 ?DO DUP EXECUTE LOOP DROP
;

’ hello 2times CR
’ hello 4 ntimes
```

実行結果

```
HELLO!HELLO!
HELLO!HELLO!HELLO!HELLO!
```

匿名ワード

リスト 4.9 では動作確認のためだけに hello ワードを定義したが、実用上「その場で名前のない (というより名付ける必要のない) ワードを定義して XT だけを得る」という操作が必要になることが多い。

こういった要望に応える言語機能として**匿名ワード**という機能が用意されている。その名の通り「名前のない」ワードを生成できる機能だ。実際に匿名ワードを生成するには :NONAME ワードを用いる。:NONAME の後ろに、: と同じようにワード定義の本体を記述して ; で定義を終了する。すると :NONAME は XT をプッシュするので、それをどこかに保持しておけば、いつでも EXECUTE できる。

:NONAME ワードを用いると、ntimes の最後の 2 行は リスト 4.10 のように書き換えられる。

リスト 4.10: :NONAME ワードの利用

```
:NONAME ." HELLO!" 2times CR
:NONAME ." HELLO!" 4 ntimes
```

名前の仮取得と XT の割り当て

これでワードの定義内容をデータとして持ち回る方法を知ったわけだが、定義内容のない名前を辞書に先に登録しておく操作も可能である。その操作

には DEFER (--) ワードを用いる。

定義内容の定まっていない「名前だけ先に辞書登録した項目」に対して、定義内容としての XT を紐付けるには IS (xt --) ワードを用いる。

リスト 4.11: 名前の仮取得と XT 割り当て

```
DEFER will-be-defined
:NONAME ." It works!" IS will-be-defined
will-be-defined
```

実行結果

```
It works!
```


DEFER ワードで名前だけを登録したが、リスト 4.12 のように、まだ定義内容を紐付けていないワードを呼び出そうとするとどうなるのだろうか。

リスト 4.12: 定義内容の紐付いていないワードを呼び出す実験

```
DEFER foo
foo
```

実行すると、deferred word int-execute is uninitialized というエラーが発生する。

4.3 DOES>

ここまでは定数・変数を用意したり、確保したメモリ領域の先頭アドレスと名前を紐付けて辞書に登録する方法を紹介してきた。これらは「データを

保存する手段」であり、これらの操作によって生成されるワードは「保存されているデータを取得する」ためのものだ。

ただ、複雑なデータ構造をメモリ上に構築して管理する場合などを考慮すると、実用上は単に保存されているデータを返すだけでは不便だ。保存されているデータに手を加えた上で返すようなワードを定義できたほうが便利である。

それを可能にする 1 つの方法として、ここでは DOES> ワードを紹介する。

DOES> ワードは、直前に辞書に追加された項目の実行時意味論を上書きする。かなり難解なワードなので、まずは リスト 4.13 を通して基本的な動作を確認してほしい。

リスト 4.13: DOES> ワードの基本

```
CREATE foo 57 ,

foo @ . CR  \ 参照先に 57 が書き込まれていることを確認

DOES>  \ コンパイル状態に移行
@      \ 「参照先の値を取得する」実行時意味論を追加
;      \ コンパイル状態からインタプリタに移行する

foo .  \ foo ワードの挙動が変わっているか確認
```

実行結果

```
57
57
```

注目すべきは DOES> から ; の部分だ。DOES> ワードは最初に、処理系をコンパイル状態に移行させる。その後 ; によってコンパイル状態からインタプリタに移行するまで、上書きしたい実行時意味論を記述していく。

foo ワードの実行時意味論は、 DOES> 〜 ; の前後で「参照先のアドレス」ではなく「参照先の値」を返すように変化している。

DOES> ワードを用いて上書きする実行時意味論では、たとえ DOES> 〜 ; の間に何も記述しなくても、最初に「参照先のアドレスをプッシュする」という意味論が追加される。つまり DOES> 〜 ; の間では、最初に参照先アドレスがプッシュされるという前提のもとで、呼び出し時の処理を記述していくことになる。リスト 4.13 でも、「参照先の値を取得する」という実行時意味論で上書きするために DOES> と ; の間で (先に参照先アドレスがプッシュされているという前提のもとで) @ ワードを記述している。

DOES> ワードの活用法のひとつとして、「配列の要素を取り出しやすくする」方法を紹介する。

リスト 4.14: 配列を、添字を指定して要素を取得するワードとして再定義する

```
: nth  DOES> SWAP CELLS + @ ;

CREATE array 3 CELLS ALLOT
1 array !
2 array 1 CELLS + !
3 array 2 CELLS + !

nth

0 array .
1 array .
2 array .
```

実行結果

```
1 2 3
```

CREATE array 3 CELLS ALLOT を実行した時点では、array ワードは単

に参照先アドレスを返すワードだ。スタック表記は (-- a-addr) のようになる。だが nth ワードを呼び出すと array ワードの実行時意味論は変わる。先に添字 (0 始まり) をデータスタックに積んでおくと、それによって指定される要素の値を取得して返すワードになる。スタック表記は (n -- x) のようになる。

もう 1 つ、DOES> ワードの利用例を紹介する。リスト 4.15 では、呼び出すたびにカウントアップされた値を返すワードを定義している。

リスト 4.15: 毎回カウントアップされた値を返すワード

```
CREATE counter -1 ,
DOES> DUP DUP @ 1 + SWAP ! @ ;
counter .
counter .
counter .
counter .
```


実行結果

```
0 1 2 3
```

DOES> ワードの追加する実行時意味論である「参照先アドレスの取得」を利用して、呼び出すたびに参照先の値を更新することも可能である。そのため参照先のメモリ領域は、そのワードにとっての環境 (各局所変数とその値の対応) として利用できる。counter ワードは、いわばクロージャ (関数閉包) だったというわけだ。

VARIABLE, VALUE ワードによって定義されたワードでも、DOES> ワードによる実行時意味論の上書きが可能である。

リスト 4.16: VALUE によって作られる変数に対しての DOES>

```
57 VALUE prime-num
DOES> @ 3 / ;
prime-num .
```

実行結果

```
19
```

第 5 章

コンパイラ変形

ここまで扱ってきた内容を総合すると以下のようになる。

47

- ワードの意味論は処理系の状態に依存して切り替わる
- 制御構造を作り出すワードがコンパイル状態で利用できる
- メモリへのデータ書き込み・読み出しのための手段が用意されている

: や ; 、DOES> ワードを除くと原則「処理系の状態に直接影響を与えることのない」機能である。処理系がコンパイラ・インタプリタのどちらとして動作しているかに従って、事前に定義されたワードの定義内容を実行する、ということに変わりはない。

コンパイラ「変形」と題した本章では、コンパイル状態の処理系に対して直接的に影響を与えるいくつかの手法について述べる。ここで述べられる内容はどれもアドホックなコンパイルをするための断片的な機能だが、それぞれの特性を知り上手く組み合わせることで、いわゆるメタプログラミングが可能になる。

5.1　メタプログラミングとは

ロジックを直接記述するのではなく、あるパターンを持ったロジックを生成する高位ロジックを用いて実装していく手法である。FORTH においては「コンパイル状態の処理系に対して影響を与えるワードを用いて、パターンに従ったワードを定義させる手法」と対応する。

5.2　IMMEDIATE

コンパイル状態の処理系に対して作用するワードを定義するには、**コンパイル状態であっても認識されたときに呼び出しが発生する**という特性を持ったワードが定義できる必要がある。それを可能にするのが IMMEDIATE ワードだ。

IMMEDIATE ワードは、直前に定義されたワードに対して作用し、そのワードがコンパイル状態でも呼び出されるようにできる。IMMEDIATE ワードによってコンパイル状態でも呼び出せるようになったワードを **即時ワード**と呼ぶ。組み込みワードのなかにも即時ワードは含まれており、中でも一番我々にとって馴染みが深いのが ; ワードだ。; ワードが即時ワードであるから「ワードのコンパイルを終了し解釈状態に戻る」という意味論を実現できている。

リスト 5.1 を、ソースファイルに記述するのではなく対話環境で 1 行ずつ入力して実行してみてほしい。

```
: say-hello  ." HELLO" ; IMMEDIATE
: new-word
  say-hello
;
```

new-word のコンパイル中、say-hello を入力して確定したときに HELLO と出力されるはずだ。コンパイル状態で呼び出しが発生していることがわかる。

この new-word ワードは、解釈状態でも呼び出せる。即時ワードで、自らが呼び出されているときにコンパイル状態なのか解釈状態なのかを判別するには STATE (-- a-addr) ワードを用いる。STATE ワードが返すアドレスは「処理系の状態を表す値の書き込み先」であり、@ ワードで参照先の値を取り出してその時々での処理系の状態を取得できる。-1 ならばコンパイル状態であり、その他の値については処理系依存である。つまり STATE ワードで確実に判別できるのは「コンパイル状態であるか否か」だけである。

4
9

リスト 5.2: STATE ワードで処理系の状態を取得する

```
: check-state
  STATE @ -1 =
  IF
    ." Compilation state"
  ELSE
    ." Other state"
  THEN
; IMMEDIATE

check-state

: new-word  check-state ;
```

実行結果

```
Other state
Compilation state
```

5.3 POSTPONE

即時ワードの処理を抑制して、定義中のワードの実行時意味論に含まれるように「延期」するのが POSTPONE ワードだ。POSTPONE ワード自体も即時ワードであり、直後のソースコードから 1 単語をワード名としてパースして、それを延期して呼び出すようにコンパイルさせる。

リスト 5.3 では、リスト 5.2 で定義した check-state ワードを用いて POSTPONE ワードの挙動を確認している。

リスト 5.3: POSTPONE

```
: after-compilation  POSTPONE check-state ;
: new-word  after-compilation ;

new-word CR
after-compilation
```

実行結果

```
Other state
Other state
```

5.4 EVALUATE とパーサ

文字列をそのままソースコードとして解釈実行できてしまうのが EVALUATE ワードだ。スタックに文字列 (メモリ上の先頭アドレス + 文字数) を積んでおけば動作するが、(VALUE や CONSTANT ワードのように) 直後のソースコードをパースして利用する方法もある。

```
S\" .\" evaluated\"" EVALUATE
```

実行結果

```
evaluated
```

```
PARSE WORD
```

付録 A

forth-jp について

筆者は、「FORTH 言語に興味がある技術者同士で情報を共有する場所」としてコミュニティ **forth-jp** を運営している。活動場所・内容は以下の通りである。

Slack ワークスペース

参加メンバー同士で、FORTH 言語の仕様や作成したプログラム、参考になるサイトなどについて共有する場所として使用している。

Scrapbox プロジェクト

`https://scrapbox.io/forth-jp`
FORTH に関する知見を蓄積していく場所として使用している。

参加するには、招待リンク (`https://bit.ly/2ZzTiYf`) から直接 Slack ワークスペースに入るか、または筆者の Twitter アカウント (`@0918nobita`) に連絡をとってほしい。

あとがき

いや〜同人誌書くの初めてだったんですが、めっっっっちゃ大変ですね！

え？ いきなり文体が変わってる？ いいじゃないですか、あとがきくらいラフにいきましょうよ。それはさておき、こんな拙い文章をここまで読んでいただき、本当にありがとうございます。FORTH 言語のことを少しでも面白いと思っていただけたなら幸いです。

FORTH 言語って本当に不思議な言語だと思います。シンプルで低レイヤ向けなのにわりと抽象的に記述できてしまうの凄いですよね。今回は基本的な言語機能の解説に留めましたが、またいつかメタプログラミングがメインの本を書いてみようと企んでいます。

正直「コンパイラ変形」の章については勉強不足感が否めなかったです。でも「いつになっても頒布されない神本」より、「頒布されたクソ本」のほうがよっぽどマシだという精神で、今まで蓄積してきた知見を体系化して文章に起こしてみました。「コンパイル状態でデータスタック/リターンスタックに積まれる値にはどんな意味があるのか」、「FORTH で内部 DSL を作るにはどんな機能が役に立つのか」など、まだまだ課題は残っています。

本書の内容についてのご意見・ご感想、またはご指摘などがあれば Twitter などで言及してもらえると、筆者としてはとても励みになります。

本書をきっかけに、FORTH があなたのプログラミングの世界観に彩りを加えることができたなら、これ以上嬉しいことはありません。

それじゃ、またどこかでお会いしましょう。

索引

索引

DARK FORTH

2019 年 9 月 17 日　発行
著　者　Kodai
印刷所　株式会社 日光企画