お試し版

役に立たない

シェルスクリプト読本

やまぐちたかのり

これは技術書典 21 で頒布する『役に立たないシェルスクリプト読本』全 4 章のうち 1 章だけを収録したお試し版である。
完全版 (紙版、PDF 版) は 2026/11/23 の技術書典 21 オフライン会場「せ 32」、または以下の URL から入手できる。

https://techbookfest.org/product/h1M6K5h2NhytQvYMyTcvd9

技術書典 21 の会期終了後も、電子本であれば完全版を上記 URL から入手できる。物理本が欲しい場合は 𝕏 @yamaya に問い合わせてほしい。

序

本書は『役に立たないシェルスクリプト読本』である。「読本」というと軽い読みもののように感じられるかもしれないが、扱っている内容はどちらかというと怪文書に近い。高度なテクニックを駆使して役に立たないシェルスクリプトを書こうというのが本書の主旨である。いや、「高度なテクニック」というのもまだ語弊がある。極限まで変態的な手法を追求して実用性ゼロのスクリプトを書く本である。これを読んで得られるものは何もないし、あなたの仕事に生かせるようなことは書かれていない。

いまどきは、役に立つスクリプトを書く必要があるなら AI に任せておけばいいのだ。そんなのは人間のやる仕事ではない。ちゃんとしたスクリプトを書くのは AI にまかせて、人間は生産性を `/dev/null` に投げ捨てることに全力を尽くすべきだ。それはまさに、AI にはできない人間だからこその営みなのである。

目次

序 ・・ iii
第 1 章　算術式の闇 ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 1
1.1　算術式とは ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 2
1.2　算術式でできること ・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 3
1.3　算術式のチューリング完全性 ・・・・・・・・・・・・・・・・・・・・・・ 6
1.4　外部からの算術式注入 ・・・・・・・・・・・・・・・・・・・・・・・・・・ 7
1.5　算術式からのコマンド実行 ・・・・・・・・・・・・・・・・・・・・・・・ 9
1.6　罠を回避する方法 ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 12
1.7　算術式で Brainfuck インタプリタ ・・・・・・・・・・・・・・・・・・・ 13
1.8　まとめ ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 18
算術式と `expr` の終了ステータス ・・・・・・・・・・・・・・・・・・・ 19
第 2 章　!”#$%&’()=~| ・・・・・・・・・・・・・・・・・・・・・・・・・・・ 21
2.1　Hello, World! までの長い道 ・・・・・・・・・・・・・・・・・・・・・・・ 22
2.2　記号だけでできることできないこと ・・・・・・・・・・・・・・・・・・ 23
2.2.1　できること ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 23
2.2.2　できないこと ・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 25
2.3　記号から数字を得る ・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 26
2.4　記号と数字からアルファベットを得る ・・・・・・・・・・・・・・・・・ 28
2.4.1　うまくいかない方法 ・・・・・・・・・・・・・・・・・・・・・・・・ 28
2.4.2　ワイルドカードの利用 ・・・・・・・・・・・・・・・・・・・・・・・ 29
2.4.3　エラーメッセージの利用 ・・・・・・・・・・・・・・・・・・・・・ 41
2.4.4　算術式の利用 ・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 44

2.5 zsh の場合 ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 47
2.6 POSIX シェルの場合・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 48
2.7 記号だけで Brainfuck インタプリタ ・・・・・・・・・・・・・・・・・・・・ 51
2.8 まとめ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 57
💬 最低限必要な記号の種類・・・・・・・・・・・・・・・・・・・・・・・・ 58
第 3 章 明日を紡ぐコード ・・・・・・・・・・・・・・・・・・・・・・・・・・・ 59
3.1 素数を数える ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 60
3.2 逐次読み込み ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 61
3.3 キメラの回避 ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 66
3.3.1 ファイルを削除する・・・・・・・・・・・・・・・・・・・・・・・・・・ 66
3.3.2 ファイルを移動する・・・・・・・・・・・・・・・・・・・・・・・・・・ 70
3.3.3 一括読み込みさせる・・・・・・・・・・・・・・・・・・・・・・・・・・ 72
3.4 Re: 素数を数える・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 73
3.4.1 カウンタと剰余演算・・・・・・・・・・・・・・・・・・・・・・・・・・ 73
3.4.2 二重ループ ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 75
3.5 FizzBuzz ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 76
3.6 bash 拡張を使わない例 ・・・・・・・・・・・・・・・・・・・・・・・・・・ 76
3.7 自己追記で Brainfuck インタプリタ ・・・・・・・・・・・・・・・・・・・・ 78
3.8 まとめ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 78
💬 逐次読み込みが有益な用途 ・・・・・・・・・・・・・・・・・・・・・・・ 79
第 4 章 お絵描きしよう・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 81
4.1 画像フォーマット・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 82
4.1.1 ベクター画像・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 82
4.1.2 ビットマップ画像 ・・・・・・・・・・・・・・・・・・・・・・・・・・ 84
4.2 フィボナッチ数 ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 87
4.2.1 フィボナッチ螺旋 ・・・・・・・・・・・・・・・・・・・・・・・・・・ 87
4.2.2 フィボナッチ列フラクタル ・・・・・・・・・・・・・・・・・・・・・・ 89
4.3 マンデルブロ集合・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 92
4.4 L-system ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 95

4.4.1 コッホ曲線 ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 95
4.4.2 フィボナッチ列フラクタル ・・・・・・・・・・・・・・・・・・・・・・ 97
4.4.3 ドラゴン曲線・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 98
4.4.4 Gosper 曲線 ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 100
4.4.5 フラクタル植物 ・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 101
4.5 セルオートマトン・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 104
4.5.1 ライフゲーム・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 104
4.5.2 シェルピンスキーのギャスケット ・・・・・・・・・・・・・・・・・・・ 107
4.6 まとめ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 110
💬 表紙について・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 111
あとがき ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 112

1 算術式の闇

あまり知られていないが、bash は算術式だけでチューリング完全である。

この本は『役に立たないシェルスクリプト読本』である。いきなりその 1 章から算術式がチューリング完全な強力な道具だという有用な情報を書いたら主旨に反するのではないかと思うかもしれない。しかし心配ない。たしかに算術式はチューリング完全だが、役には立たない。

より正確に言えば、チューリング完全という性質が役に立つような使い方をしてはいけない。最悪の場合、任意コード実行などのセキュリティホールになりうる危険な道具であり、いかにしてそのような危険な動作をさせないスクリプトを書くかを考えなければならない。順を追って説明していこう。

1.1　算術式とは

算術式 (arithmetic expression) は、シェルの中で四則演算などの簡単な計算をおこなうための道具である。POSIX で規定されているのは、`$(( ... ))` の形式の算術式展開 (arithmetic expansion) としての利用だけである[1)]。

```
$ echo $((1+1))
2
```

ksh、bash、zsh など POSIX から機能を拡張したシェルでは、これ以外にも以下のようなさまざまな場所で算術式を利用できる。

まず、`(( ... ))` や `let ...` で算術式単体を文とすることができる。

```
$ ((a=1+1))
$ echo $a
2
$ let b=2+3
$ echo $b
5
```

この文は計算結果が 0 (偽) の場合に終了コードが 1 (コマンド異常終了) になり、それ以外 (真) の場合は 0 (正常終了) になるため、`if` や `while` と組み合わせて使われることが多い。

```
while ((++i<10)); do
    ...
done
```

数値型に宣言した変数に対する代入は、すべて算術式である。

```
$ typeset -i a          # 変数aは整数型
$ a=3+2
$ b=3+2
$ echo $a $b
5 3+2
```

1)　bash と zsh では `$[ ... ]` でも算術式展開が可能。

この例では $a、$b に同じ 3+2 という値を代入しているが、整数型宣言した $a のほうだけ算術式評価されている。zsh では整数型だけでなく浮動小数点型に宣言した変数も同様に算術式として扱われる。bash と zsh では declare でも型宣言できる (ksh は typeset のみ)。

配列変数の添字も算術式である。

```
$ a=(A B C D E)
$ echo ${a[2]}  ${a[1+1]}
C C
```

条件判断の [[...]] で数値比較演算子が使われた場合には文字列ではなく算術式として扱われる[2)]。

```
$ [[ 2+2 -eq 2*2 ]] && echo true || echo false
true
```

そのほか、以下の場面で算術式が使われる。

- for((expr1;expr2;expr3)) という形式のループにおける expr1 ... expr3
- 部分文字列の取得に使われる ${var:offset:length} における offset と length
- ksh と zsh ビルトインの書式つき数値出力 (printf %d expr) における expr[3)]

1.2　算術式でできること

さて、算術式では四則演算などの簡単な計算をおこなうことができるが、四則演算しかできないわけではない。

まず、変数への代入と参照ができる。

2)　-eq や -le などの数値として比較する演算子を使った場合のみ。= や =~ など文字列として比較する演算子では算術式評価はおこなわれない。また、[[...]] ではなく [...] や test ... を使う場合は -eq などの数値比較演算子でも算術式評価はおこなわれない。ただし、ksh は [...] や test ... でも数値比較演算子を使っていれば算術式として評価される。

3)　%d のほか、%x や %f など数値を引数とする書式が指定された場合に算術式評価される。%s など文字列を引数とする書式は算術式として評価されない。bash の printf や、ビルトインではない /usr/bin/printf などは数値からの変換でも算術式評価はされない。

```
$ ((a=1+1))
$ echo $a
2
$ echo $((a*a))
4
```

これは 1 行目で `1+1` の計算結果を `$a` に代入している。算術式で使われる変数はシェルの変数そのものなので、いったん代入されれば算術式の外でも参照できる (逆に、算術式の外で定義された変数を算術式の中から参照することももちろん可能)。算術式の中では変数の値を `$a` ではなく `a` で参照できることに注目してほしい。これは些細なことのようにも見えるが、実は後で述べるように重要な特徴である。

算術式の中で、条件判断ができる。

```
$ a=10
$ (( (a>100) && (a=100) ))
$ echo $a
10
$ a=200
$ (( (a>100) && (a=100) ))
$ echo $a
100
```

これは「`$a` の値が 100 より大きければ 100 にする」という処理をしている。`&&` や `||` といった論理演算子が利用可能なほか、`>` や `==` といった条件判断演算子、“`式 ? 式 : 式`” という三項演算子も用意されている。

ここまでは POSIX の範囲内の機能である。ここから先は POSIX で定義されない ksh 由来の拡張機能で、ksh と bash、zsh で利用可能なものである。

`,` で区切ることによって算術式の中に複数の算術式を書くことができる。一般的なプログラミング言語における `;` のような役割と思ってよい。

```
$ echo $((a=1+1, a*a))
4
$ echo $a
2
```

これは、`a=1+1` と `a*a` を続けて算術式評価することになる。算術式展開の結果と

しては最後の式の値が使われる。

算術式の中に文字列があらわれた場合、その文字列が算術式とみなされて再評価される。再評価は 1 回かぎりではなく、再帰的におこなわれる。

```
$ a=b
$ b=c
$ c=1
$ echo $((a))
1
```

変数 $a の中身は b という文字列であり、変数 $b の中身は c であり、$c の中身は 1 という数値である。よって、$((a)) という算術式展開は、まず a が評価されて b になり、b が評価されて c になり、最後に c が評価されて 1 という結果が得られる。

```
$ a="2+3"
$ echo $((a*2))         # bashでは、aの値は2+3=5なので5*2が計算される
10
$ echo $(($a*2))        # $aの値は"2+3"なので、2+3*2が計算される
8
$ echo $((($a)*2))      # $aの値は"2+3"なので、(2+3)*2が計算される
10
```

この例で、$a の値は 2+3 という文字列である。$((a*2)) という算術式は、ksh や bash など算術式の再帰評価をおこなうシェルでは、まず文字列 a が算術式として評価され (2＋3＝5 である)、続いて 5*2 という算術式が評価されて 10 という結果が得られる。一方、POSIX にほぼ準拠するシェルでは 2+3 はあくまで文字列であり、算術式としての再帰評価はおこなわれない (たとえば ash[4]や dash[5]などではエラーになる)。

$(($a*2)) はまず $a が文字列として展開されて 2+3 になり、続いて 2+3*2 が計算されて 8 という結果になる。こちらは再帰評価ではなく単なる変数展開であることに注意してほしい。再帰評価ではないので POSIX 準拠のシェルでも機能拡張されたシェルでも同じ動作になるが、演算子の優先度の都合上、直感的ではない結果

4)　FreeBSD や NetBSD で /bin/sh として使われるシェル。
5)　Debian や Ubuntu で/bin/sh として使われるシェル。

になる。(`$a`)`*2` のように変数をカッコでくくれば拡張の有無にかかわらず `10` という結果を得ることができる。

このように、算術式の中で変数の値を参照するのに `a` とするか `$a` とするかで計算結果が異なることがある。

1.3　算術式のチューリング完全性

さて、冒頭で bash の算術式はチューリング完全であると述べた。詳細かつ厳密な説明は役に立たない本書では割愛して役に立つ別の書籍に譲るが、簡単にいえば以下の 3 つの機能を持つ計算モデルはチューリング完全であり、どんなアルゴリズムでも記述できる能力を持つ。

1. 状態の保存
2. 条件分岐
3. 無条件ジャンプ

状態の保存は、メモリやレジスタ、ファイルなどに対する出力ができることを意味する。条件分岐は現在の状態や入力された値によって処理の流れを変更する能力である。ジャンプができれば継続的な状態遷移が可能になる。

たとえばループを実現するには、繰り返しの先頭に戻るための無条件ジャンプが必要であり、さらに、現在の状態を保存し、その状態に応じてループを継続するか終了するかを判断して分岐できれば「意味のあるループ」ができることになる。

この 3 つをシェルの算術式の機能にあてはめてみよう。

1. 状態の保存: 算術式の中で変数に代入できる
2. 条件分岐: `&&` や `||` による短絡評価ができる
3. 無条件ジャンプ: 再帰評価により評価中の式から新たに別の式を評価できる

条件をすべて満たしている。bash は、算術式だけでチューリング完全なのだ。なお、POSIX シェルの範囲では無条件ジャンプができないのでチューリング完全ではない。あくまで、ksh や bash などの機能拡張されたシェルの算術式だけで

ある。

例として、算術式だけでフィボナッチ数列を生成してみよう。言うまでもなく、フィボナッチ数とは以下の式であらわされる数列で、$1, 1, 2, 3, 5, 8, 13, ...$ という値になる。

$$F_0 = 0$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2} \qquad (n \geq 2)$$

スクリプトはリスト 1.1 のようになる[6)]。

▼ リスト 1.1　算術式によるフィボナッチ数列

```
1 #!/bin/bash
2 fib='i++, F[i]=F[i-1]+F[i-2], i<20 && fib'    # 算術式としての文字列を定義
3 ((i=1, F[0]=0, F[1]=1, fib))                  # 算術式評価
4 echo ${F[@]}                                  # 配列の全要素を出力
```

2 行目で文字列を変数 `fib` として定義し、3 行目で初期値を設定した後で算術式として評価している。変数 `fib` の最後には「条件 (`i<20`) を満たしていれば自分自身 (`fib`) を評価する」という式が含まれていて、再帰評価によるループを実現している。

これを実行すると、以下のように F_0 から F_{20} までが出力される。

```
1 $ bash fib.sh
2 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765
```

1.4　外部からの算術式注入

このように、bash の算術式は POSIX シェルとは異なり、いくつかの機能拡張がなされたことによってチューリング完全になっている。しかし、これを便利だと

6)　古いバージョンの bash (macOS の `/bin/bash` など) にはバグがあって、このスクリプトはうまく動かない。

思ってはならない。拡張された機能のうち、とくに再帰評価が極めて厄介で、避けるべきものである。

たとえば、リスト 1.2 のようなスクリプト (arithtest.sh) を考える。

▼ リスト 1.2　arithtest.sh

```
#!/bin/bash
a="hoge"
if [[ $1 -eq 1 ]]; then
    b="fuga"
else
    b="piyo"
fi
echo "a=$a, b=$b"
```

arithtest.sh 1 のように引数として 1 を与えて実行すると、b=fuga が代入され、それ以外の場合は b=piyo が代入される。実際、そのように動いているように見える。

```
$ bash arithtest.sh
a=hoge, b=piyo
$ bash arithtest.sh 0
a=hoge, b=piyo
$ bash arithtest.sh 1
a=hoge, b=fuga
```

しかし、ここで arithtest.sh a=10 のように実行するとどうなるだろうか。

```
$ bash arithtest.sh a=10
a=10, b=piyo
```

なんと、a="hoge"と代入していたにもかかわらず、コマンドライン引数で指定されていた a=10 が実行されて上書きされてしまっている。

前述のとおり、[[...]] で数値を比較するとき、その引数は暗黙的に算術式として評価される。このスクリプトでは、[[$1 -eq 1]] として使っている。これは「コマンドライン引数で指定した文字列が 1 かどうかを調べる」ではない。「コマンドライン引数で指定した文字列を算術式として再帰評価した上で、その結果が

1 かどうかを調べる」である。a=10 は正しい算術式なので、この代入文が実行され、$a の値が破壊されてしまうのだ。

1.5　算術式からのコマンド実行

ここまでで、算術式の再帰評価を悪用することによって、脆弱なシェルスクリプトに対して任意の算術式を注入できることを示した。

算術式はチューリング完全でどんなアルゴリズムも記述できるが、とはいえできることには限界がある。入出力ができないので、既存のファイルを書き換えたり、ファイルを読み取って外部に送信したりといったことはできない。脆弱性により任意の算術式を注入できたとしても、せいぜい変数の値を破壊したり、set -e が設定されているスクリプトをエラー停止させたりできる程度でしかない (変数を外部から変更できるだけでも十分に脆弱なことには留意すべし)。

しかし、算術式それ自体にできることが限られているからといって軽視していいわけではない。残念ながら算術式の中から OS コマンドを起動する方法が存在するのだ。

bash などの機能拡張されたシェルは配列をサポートしている (先述のフィボナッチ数列のスクリプトが配列を利用した例である)。そして、配列の添字は算術式であり、同時にコマンド置換の対象なのだ。

```
$ a='x[$(echo 1)]'      # $aに文字列を定義
$ echo $a               # 文字列として出力すると、echo 1 は実行されない
x[$(echo 1)]
$ x[1]=2
$ echo $((a))           # 算術式として評価すると、echo 1 が実行される
2
```

この例では $a は文字列なので、単に echo するとその文字列がただ表示されるだけである。そして、この文字列は算術式として正しい式である。算術式を評価するとまず echo 1 が実行される。stdout に 1 が出力され、この出力がコマンド置換で配列の添字として使われる。つまり、x[$(echo 1)] は x[1] と解釈されることになる。

算術式は前述のとおり入出力機能を持たない。リスト 1.1 のフィボナッチ数列のスクリプトでも、結果の出力は算術式の外側でおこなっていた。しかし、このコマンド置換を利用することで間接的に可能になる。たとえば、リスト 1.3 は配列添字のコマンド置換を利用した FizzBuzz スクリプトである (実行結果は見慣れた FizzBuzz なので省略して問題ないだろう)。

▼ リスト 1.3　算術式による FizzBuzz

```
#!/bin/bash
fb='i%15==0 ?
      x[$(echo FizzBuzz >&3)] :
      ( i%3==0 ?
        x[$(echo Fizz >&3)] :
        ( i%5==0 ?
          x[$(echo Buzz >&3)] :
          x[$(echo $i >&3)]
        )
      ),
    ++i > 100 || fb'
((i=1,fb)) 3>&1
```

stdout に出力すると配列の添字として使われてしまうため、この例ではいったんファイル記述子 3 に出力 (`>&3`) した後で、それを stdout にリダイレクト (`3>&1`) している。

算術式の中でこのようなコマンド置換がおこなわれるのは配列の添字で使われた場合のみである。それ以外の場所でコマンド置換をさせようとしてもエラーになる。

```
$ a='$(echo 1)'          # $aに文字列を定義
$ echo $a                # 文字列として出力するとそのまま表示される
$(echo 1)
$ echo $((a))            # 算術式として評価するとエラー
bash: $(echo 1): arithmetic syntax error: operand expected (error token is "$(echo 1)")
```

さきほどの `arithtest.sh` (リスト 1.2) を以下のように実行してみよう。

```
$ bash arithtest.sh 'x[$(echo hello world >&2)]'
hello world
a=hoge, b=piyo
```

`$(...)` で指定したコマンドがそのまま実行されてしまっている。外部から算術式を注入できるということは、任意のシェルコマンドを注入できることと同義なのだ。

もうひとつ例を示そう。リスト 1.4 は “**品目, 単価, 個数**” という形式の CSV を読んで、“**品目, 合計金額**” の形式で出力するスクリプトである。

▼ リスト 1.4　掛け算スクリプト (amount.sh)

```
#!/bin/sh
while IFS=, read item price num; do
    echo "$item,$((price*num))"
done
```

ただ掛け算をするだけの単純なスクリプトだ。

```
$ cat fruit.csv
apple,200,5
orange,100,8
$ bash amount.sh < fruit.csv
apple,1000
orange,800
```

しかし、このスクリプトに入力する CSV ファイルに算術式が含まれていた場合はどうなるだろうか。

```
$ cat vuln.csv
apple,200,x[$(echo hello world >&2)]
$ bash amount.sh < vuln.csv
hello world
apple,0
```

算術式が評価され、中に含まれるコマンドが実行されてしまった。

このシェルスクリプト自体は POSIX の範囲内で書かれていて、bash の拡張機能は一切使っていない (念のため、`$(( ... ))` による算術式展開は POSIX の範囲内である)。そして、ash や dash などの拡張機能を実装していないシェルであれば、CSV ファイルに含まれるコマンドは実行されず、安全である (エラーになる)。POSIX 互換の安全なスクリプトが、機能拡張されたシェルで実行されると一転して脆弱になるのだ。

1.6　罠を回避する方法

算術式の再帰評価は有効無効を任意で切り替えられるオプションではない。常に有効であり、無効にする方法は存在しない[7)]。

そのため、このような脆弱性を避けるためには、まずはどんな場面で算術式評価がおこなわれるのかを正確に把握しておかなければならない。POSIX では `$(( ... ))` だけだったが、bash などではさまざまな場面で算術式が使えるように (よけいな) 拡張がされている。1.1 節で列挙しているので、あらためて確認してもらいたい。

とくに、リスト 1.2 の例のように `[[ ... ]]` による条件判断はこの問題に該当しやすいため、どうしても必要がなければ `[ ... ]` の方を使うことを推奨する (ただし、ksh では `[ ... ]` でも算術式評価がおこなわれるので回避策にならない)。

型宣言するプログラミング言語に慣れていると、数値を格納する変数を `typeset` や `declare` で整数型に宣言したくなるかもしれない。整数型に宣言しておくと、文字列が代入されたときにエラーになって検知できる、というのは正しい理解ではない。文字列であっても算術式として正しければエラーにならず、代入した瞬間に算術式評価されてしまう。型宣言しなくても算術式による計算はできる。あえて型宣言するのはリスクを増やすだけである。どうしてもそうしなければならない理由 (筆者には思いつかない) がなければ型宣言すべきではない。

```
1 #!/bin/bash
2 declare -i n    # 変数nは整数型
3 n="$1"         # コマンドライン引数から算術式インジェクションできる
4 ...
```

そして、外部から入力される値 (コマンドライン引数、標準入力、環境変数など) をそのまま算術式評価せず、文字列が数字だけで構成されているかをチェックすることが必要になる。リスト 1.5 はその例である (負の数を考慮していない例である

7)　`set -o posix` を設定したり、`bash --posix` で起動しても効果はない。posix mode は POSIX と非互換の部分の動作を変更するものであって、上位互換 (と bash の開発者が考えているもの) の動作を変えるものではないからだ。

ことに留意すべし)。

▼ リスト 1.5　数字だけで構成されてるかの判定

```
is_valid_number(){
    case "$1" in
    *[!0-9]*) return 1;;  # 0-9 以外の文字が含まれている場合
    *)        return 0;;  # それ以外(0-9だけで構成されている文字列)の場合
    esac
}

is_valid_number "$1" && [[ $1 -eq 1 ]]
```

計算が必要な場面でも算術式を使わず、あえて expr コマンドを使うのも回避策のひとつである。expr は計算機能を持たなかった POSIX 以前の Bourne シェルの時代に使われていたもので、現代ではバッドプラクティスのひとつとされることが多いが、算術式注入による脆弱性を避ける意味では有用である (算術式より expr を使うべきだと言っているわけではない。前述のシェル関数でバリデーションするなどして算術式を使うほうが速度や可読性などさまざまな点でメリットが大きい)。

bc や dc も計算コマンドであるが、どちらもチューリング完全な「言語」であり、さらに GNU 版の dc は外部コマンドの実行も可能なので、これらのコマンドに渡す計算式も算術式と同じ注意を払わなければ同じ脆弱性が起きうる。もちろん Python などのスクリプト言語も同様である。

ash や dash の算術式では再帰評価がおこなわれないのでこのような問題は起きない。しかし、ash、dash で動かすことを想定したシェルスクリプトが他の環境に転用されることもありうるので、問題のない環境で実行する前提のスクリプトであっても配慮した記述をするのが望ましい。

念のため補足しておくと、算術式を一切使うべきでないと言っているわけではない。再帰評価されるような算術式は避ける、外部からの入力を算術式評価しない。こういった点に注意していれば算術式を使ってもとくに問題はない。

1.7　算術式で Brainfuck インタプリタ

最後に、算術式を使ったスクリプトの例を挙げよう。

Brainfuck は難解プログラム言語 (Esoteric Programming Language; Esolang) として知られている。この言語は極めてプリミティブな 8 つの命令 (表 1.1) しか持たない。たとえば、`Hello, World!` を出力する Brainfuck プログラムはリスト 1.6 のようになる。

▼ 表 1.1　Brainfuck の命令

命令	動作	C 言語での記述
`>`	ポインタをインクリメント	`ptr++;`
`<`	ポインタをデクリメント	`ptr--;`
`+`	ポインタが指す値をインクリメント	`(*ptr)++;`
`-`	ポインタが指す値をデクリメント	`(*ptr)--;`
`.`	ポインタが指す値を出力	`putchar(*ptr);`
`,`	ポインタが指す値に 1 バイト入力	`*ptr = getchar();`
`[`	ポインタが指す値が 0 なら対応する `]` の次にジャンプ	`while(*ptr){`
`]`	ポインタが指す値が非 0 なら対応する `[` の次にジャンプ	`}`

▼ リスト 1.6　hello.bf

```
1 +++++++++[>++++++++>++++++++++>+++++>+<<<<-]>.>++.+++++++..+++.
2 >-.------------.<<++++++++++++++++.>.+++.------.--------.>+.>+.
```

Brainfuck は機能がかぎられていて可読性も低く、現実的にはまったく実用性がない。役に立たないという意味では実に本書向きな言語だが、この言語でプログラミングしようという話ではない。ここでは、Brainfuck が単純な言語仕様ながらチューリング完全であるという点が重要である。

1.3 節である計算モデルがチューリング完全であるための条件として状態の保存、条件分岐、ジャンプが可能なことを挙げたが、別の方法でもチューリング完全であることを証明できる。それは「ある計算モデルでチューリング完全な別の計算モデルのエミュレータを実装できるなら、そのモデルはチューリング完全である」というものである。つまり、チューリング完全な言語である Brainfuck のインタプリタを算術式で実装できれば、算術式がチューリング完全であることを証明できるということだ。

証明しよう。

リスト 1.7 が算術式で実装された Brainfuck インタプリタである。

▼ リスト 1.7　算術式による Brainfuck インタプリタ

```
#!/bin/bash

# 文字→文字コード変換用テーブル
ch=({0..255})
# Brainfuckの各命令に対応する算術式
incptr='skip || p++, bf[++i]'
decptr='skip || p--, bf[++i]'
inc='skip || b[p]++, bf[++i]'
dec='skip || b[p]--, bf[++i]'
putchar='skip || x[$(dc -e${b[p]}P >&3)], bf[++i]'
getchar='skip || (b[p]=ch[$(dd bs=1 count=1 2>/dev/null | od -An -d)]), bf[++i]'
begin='skip ? ++depth : (b[p] ? jump[j++]=i : (++depth, skip=1)), bf[++i]'
end='depth ? (--depth || (skip=0), ++i) : ((b[p] ? i=jump[--j] : (--j, ++i)), skip
=0), bf[i]'

# Brainfuckのコードを読み込んで配列に格納
bf=(
    $(tr -dc '<>[],.+-' < "$1" |\
      sed 's/</decptr /g
```

```
            s/>/incptr /g
            s/\+/inc /g
            s/-/dec /g
            s/\./putchar /g
            s/,/getchar /g
            s/\[/begin /g
            s/]/end /g')
)
# コード実行
i=0; p=0; b=(); jump=(); j=0; skip=0; depth=0
((bf[0])) 3>&1
```

算術式には入出力の機能がないので、. (出力) と , (入力) のふたつの命令は純粋な算術式だけでは実装できず、算術式の中から外部コマンドを呼び出すことで実現している。しかし、入出力ができることはチューリング完全であるための条件にも含まれていないので、それ以外の 6 命令を算術式で実装できればチューリング完全であることの証明には十分である。そして、実際にそれを実装している。

各命令は `${bf[@]}` という配列変数に格納されている。その命令の最後で次に実行すべき命令を再帰評価することで、順番にコードを実行していく。

基本的には配列の次の要素を実行するだけだが (`bf[++i]`)、`[` と `]` のふたつの命令は特別な扱いをしている。条件にマッチしなかった場合に `[` から対応する `]` までジャンプするため、その間にあるコードを実行させないためのフラグ (`skip`) と、ネストの深さ (`depth`)、`]` 命令からの帰還先アドレス (`jump`) を管理し、条件分岐とジャンプを実現している。

意図したとおり動作するか確認するため、リスト 1.6 のコードを実行してみよう。

```
$ bash bf.sh hello.bf
Hello, World!
```

見事、`Hello, World!` が出力された。

もうひとつ実行例を挙げよう。リスト 1.8 は、`cat` コマンドのように stdin から入力された内容をそのまま stdout に出力するだけの Brainfuck コードである。

▼ リスト 1.8　cat.bf

```
,[.,]
```

実行してみよう。

```
$ bash bf.sh cat.bf < bf.sh
#!/bin/bash

# 文字→文字コード変換用テーブル
ch=({0..255})
# Brainfuckの各命令に対応する算術式
incpbf.sh: line 29: p: expression recursion level exceeded (error token is "p")
bf.sh: line 29: ++i: expression recursion level exceeded (error token is "i")
```

途中までは出力されているが、エラーで止まってしまった。

実は、bash の算術式は再帰が可能とはいえ、その再帰の深さはかなり低い値 (1000 回ちょっと) までに制限されている。

```
$ a='++i,a'
$ ((a))
bash: ((: ++i,a: expression recursion level exceeded (error token is "i,a")
$ echo $i
1022
```

`cat.bf` の例でわずか 130 バイトほど出力しただけで上限に達してしまったことからもわかるとおり、この程度ではまったく実用にならない[8)]。Brainfuck 自体が非実用的というのは置いておいても、それ以外の用途で算術式を使おうにも中途半端すぎて使いどころがかぎられてしまう。その一方で、算術式を利用したコマンドインジェクションをするには十分以上である。実用には制限が厳しすぎるが、攻撃者にとっては制限がないに等しいのが算術式なのだ。

せめて再帰の上限回数をオプションで変更できれば、本格的な用途では上限を緩和したり、逆に外部からの算術式インジェクションを防ぐために上限をゼロに設定したりとかなり便利になるのだが、そんな機能は存在しない。あきらめよう。

8)　`https://qiita.com/akinomyoga/items/2dd3f341cf15dd9c330b` ではこの上限を突破する方法を提案している。この記事は本章で述べている内容の多くについても網羅していて、また、本家チューリングマシンのエミュレータも実装している。算術式で遊ぶなら必見の記事である。

1.8　まとめ

算術式が再帰評価され、またコマンド実行可能というのはきわめて非直感的であり、こういう問題があることを知らなければ回避するコードを書くという発想にすら至らないだろう。しかし、bash などのシェルではそれが仕様であり、意図された挙動である。

とはいえ、いくら仕様どおりであっても仕様自体が脆弱ならばそれは脆弱性なのではと考えて、筆者は 2019 年 4 月に IPA に脆弱性報告をした。しかし、その回答は「スクリプトを書く人が対処するべき問題であって、シェルそのものの脆弱性ではない」という判断だった。

開発・実装している側でも認識していないわけではないようで、mksh というシェル[9]では man に警告がある。

> *Warning*: This also affects implicit conversion to integer, for example as done by the `let` command. Never use unchecked user input, e.g. from the environment, in an arithmetic context!

bash や zsh などメジャーな実装では man にこのような警告はされていないが、スクリプトを書く側が確実に対処しなければならない。

悪用もされている。

今年 (2026 年)、Ivanti Endpoint Manager Mobile という製品で報告された CVE-2026-1281、CVE-2026-1340[10]、および BeyondTrust Remote Support/Privileged

9）OpenBSD から派生した MirOS BSD (MirBSD) という OS で `/bin/sh` として採用されていたシェル。もともと商用だった ksh のパブリックドメイン版クローンで OpenBSD の `/bin/sh` である pdksh を、MirBSD で採用するにあたってさらに修正したもの。現在 MirBSD 自体は事実上の開発停止状態にあるが、mksh は Android OS の `/system/bin/sh` に採用されるなど、現在もメンテナンスされている。

10）`https://hub.ivanti.com/s/article/Security-Advisory-Ivanti-Endpoint-Manager-Mobile-EPMM-CVE-2026-1281-CVE-2026-1340`
`https://labs.watchtowr.com/someone-knows-bash-far-too-well-and-we-love-it-ivanti-epmm-pre-auth-rces-cve-2026-1281-cve-2026-1340/`

Remote Access の CVE-2026-1731[11]は、いずれも外部から認証不要で任意コマンドの実行を許してしまうという脆弱性で、まさに本章で説明した算術式インジェクションが利用される。深刻度スコアは Ivanti が CVSS 9.8、BeyondTrust が 9.9 で、どちらも最悪レベルである。実際の攻撃も観測されている。

また現実の脆弱性ではないが、ペネトレーションテストのスキル向上のためのオンライン学習プラットフォームである Hack The Box[12]の CTF 課題にも、算術式インジェクションを利用してコード実行させるものがいくつかあるようだ。いまや学習コンテンツのひとつとなるほどに、攻撃者にとっては知っていて当然の手法となっている。

bash の算術式はチューリング完全だが、それは便利なのではなく、ただの厄介ものである。役立ててはならない。

ただし、本書は『役に立たないシェルスクリプト読本』である。次章以降では気にせず再帰評価する算術式を使う。読者は真似するべきではない。

算術式と expr の終了ステータス

1.1 節で説明したとおり、`(( ... ))` や `let ...` は計算結果によって終了ステータス (`$?`) が異なる。

これは `set -e`、つまりスクリプト中のコマンドがエラーで終了したらその時点でシェルスクリプトを終了させるというオプションを設定していた場合に、`if` や `while` などと組み合わせずに算術式を文として使って評価結果が 0 になるとスクリプトが止まってしまうということでもある。

```
1 ((count++))        # set -e されていると $count が 0 のとき異常終了する
```

一方、`$(( ... ))` は算術式展開であって文ではないので、計算結果が 0 でも終了ステータスに影響を与えない。

11） https://www.beyondtrust.com/trust-center/security-advisories/bt26-02
https://www.rapid7.com/blog/post/ra-cve-2026-1731-analysis/

12） https://www.hackthebox.com

```
$ a=$((3-3))
$ echo $?
0
$ ((a=3-3))
$ echo $?
1
```

基本的には `(( ... ))` の形式の算術式文は `if` や `while` などとともに条件判断でのみ使い、それ以外の用途では `$(( ... ))` の形式の算術式展開を利用するべきである。

```
count=$((count+1))  # set -e でも $count の値にかかわらず成功
: $((count++))      # 何もしない ":" の引数として算術式展開してもよい
```

算術式ではなく、`expr` コマンドも同様に計算結果が 0 になると終了ステータスが 1 になる。

```
$ a=$(expr 3 - 3)
$ echo $?
1
```

`expr` は計算以外にも文字列を正規表現にマッチさせる機能を持つ*)。

```
$ expr abcd : '.\(..\)'
bc
```

そして最悪なことに、正規表現にマッチした部分文字列がたまたま 0 (に解釈できる文字列) になった場合も終了ステータスが 1 になって `set -e` なスクリプトをエラー終了させてしまう。バグとしか思えない動作だが、そういう仕様のようだ。

```
$ expr a00b : '.\(..\)'
00
$ echo $?
1
```

---

*)　3 章のリスト 3.10、3.11 に使用例がある。