

# Snap! のこと II

齋藤文康

2023 年 7 月 24 日

Snap! Build Your Own Blocks(ver. 8.2.3) の使い方について、Scratch に準じているので、基本的なことは省いて、私が説明できることだけを取り上げました。マニュアルのようにすべての事柄を説明することはできません。

この文書中のスクリプトは、Debian GNU Linux の Chromium ウェブ・ブラウザ上の Snap! から script pic... または result pic... で得た画像を使用しています。他の OS やウェブ・ブラウザを使用する場合とは違った表示になっているかもしれません。

Snap! は短い周期で更新されていますので記述が合っていない箇所があるかもしれません。私の理解不足で間違っているところがある可能性があります。正しい内容になるように努めましたが、スクリプトを含め無保証です。

# 目 次

<b>1 Continuation 繼続</b>	<b>3</b>
1.1 w/continuation . . . . .	3
1.2 並列処理について . . . . .	8
<b>2 再帰</b>	<b>17</b>
2.1 再帰の例 . . . . .	17
2.1.1 階乗 . . . . .	17
2.1.2 ハノイの塔 . . . . .	17
2.2 再帰の使用 . . . . .	18
2.2.1 繰り返し . . . . .	18
2.2.2 カウントダウンとカウントアップ . . . . .	19
2.2.3 my length . . . . .	19
2.2.4 my contains . . . . .	21
2.2.5 リスト要素の巡回 . . . . .	22
2.2.6 reverse 逆順リスト . . . . .	24
2.2.7 クイックソート（整列 / 並べ替え） . . . . .	25
2.2.8 再帰呼び出しをする局所的な定義ブロック . . . . .	28
2.2.9 末尾再帰 . . . . .	31
<b>3 高階関数</b>	<b>34</b>
3.1 カリー化 . . . . .	43
3.2 OOP オブジェクト指向プログラミング . . . . .	44
<b>4 APL ライブライバー</b>	<b>47</b>
4.1 形 . . . . .	47
4.2 配列の連結 . . . . .	50
4.3 配列要素の配置転換 . . . . .	50
4.4 ベクトル、配列の範囲指定、選択 . . . . .	51
4.5 配列要素に対する演算 . . . . .	53
4.6 outer product . . . . .	55
4.7 inner product . . . . .	57
4.8 ソート、順位付け . . . . .	64

# 1 Continuation 繰続

「Continuation 繰続」は、プログラムの実行過程でその後に行われる処理と説明されます。Snap!では継続を目で見ることができるので、すこし理解しやすいかもしれません。

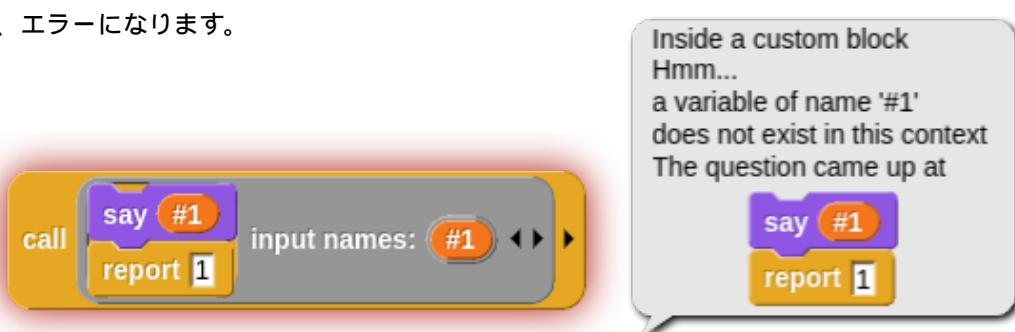
## 1.1 w/continuation

Snap!には、run w/continuation と call w/continuation があります。これは、with continuation の略です。continuation 付きの run や call ということです。

report 1 を、ただの call と call w/continuation ですると、結果は同じになります。



しかし、call の方はリング端の三角をクリックしてフォーマルパラメーターを出してやってみると、エラーになります。



これは外側の入力スロットからの入力を受け取るためのものなので、入力がないためエラーになります。外側に入力をセットするところなります。



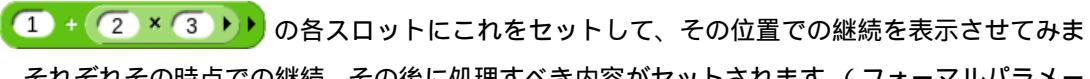
これに対して、call w/continuation の場合は、リング端の三角をクリックしてフォーマルパラメーターを出してやってみてもエラーになりません。

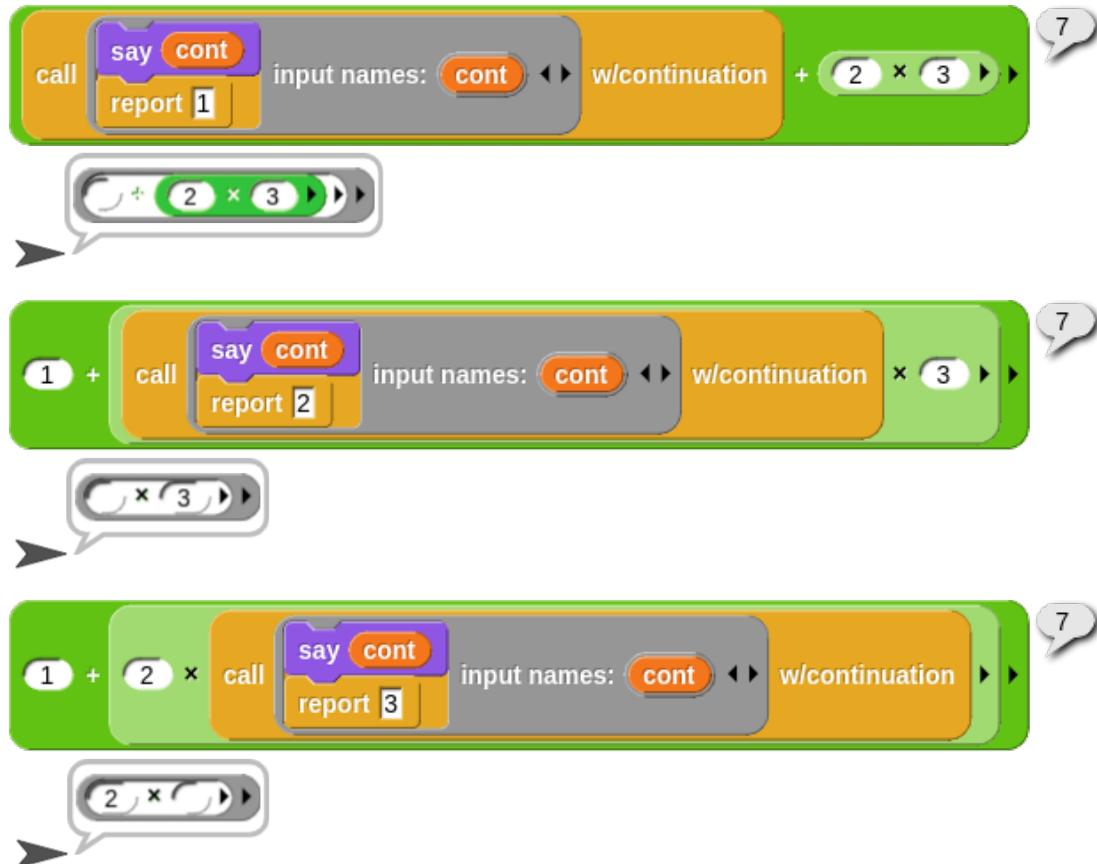


フォーマルパラメーターを表示させてみます。

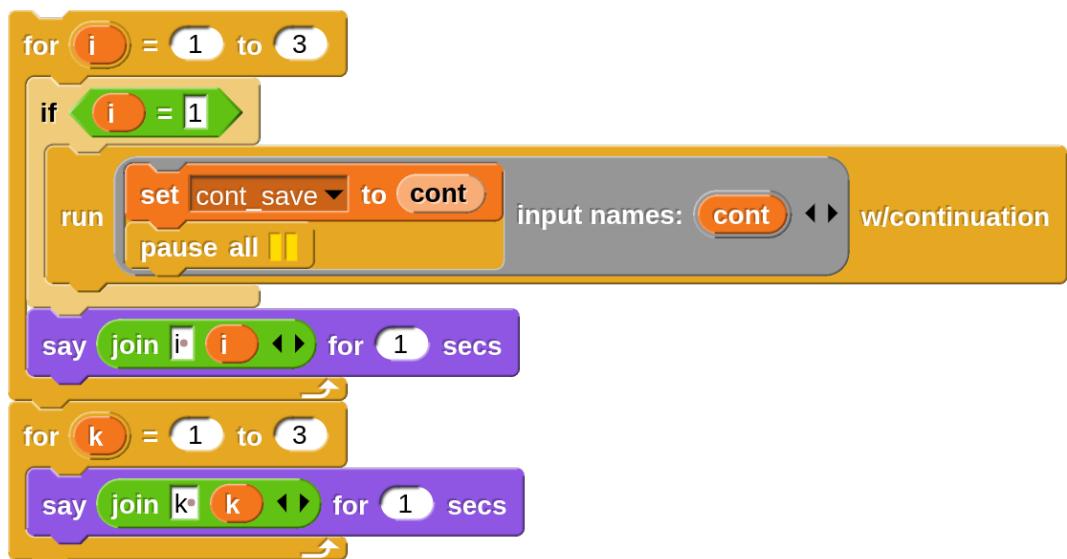


と、空のリングが表示されます。このフォーマルパラメーターには continuation ( 繰続 ) がセットされます。この場合はこの後に処理すべきものが何もないで空です。

 各スロットにこれをセットして、その位置での継続を表示させてみます。それぞれその時点での継続、その後に処理すべき内容がセットされます。( フォーマルパラメーターを `cont` にリネームしています。)



同様にスクリプト中の継続もあります。`cont_save` の変数を作成し、



を実行してください。if ブロックの部分がなければ、i 1, i 2 ,i 3, k 1, k 2, k 3 と表示するスクリプトです。変数が表示されるようになっていると、

`cont_save` `say join i i for 1 secs`

のようにセットされた継続が表示されます。pause の状態になっているので、ここで  で終了させてください。

`run cont_save`

をクリックすると、継続部分、つまりスクリプトの最後まで実行されます。

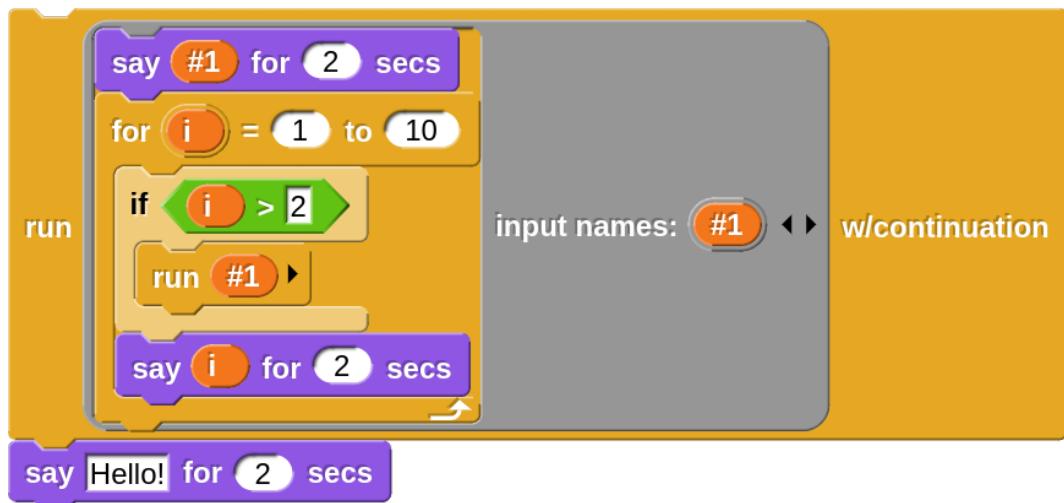
ここではわかりやすい用途として、繰り返しなどのブロックからの脱出について説明します。

`run`

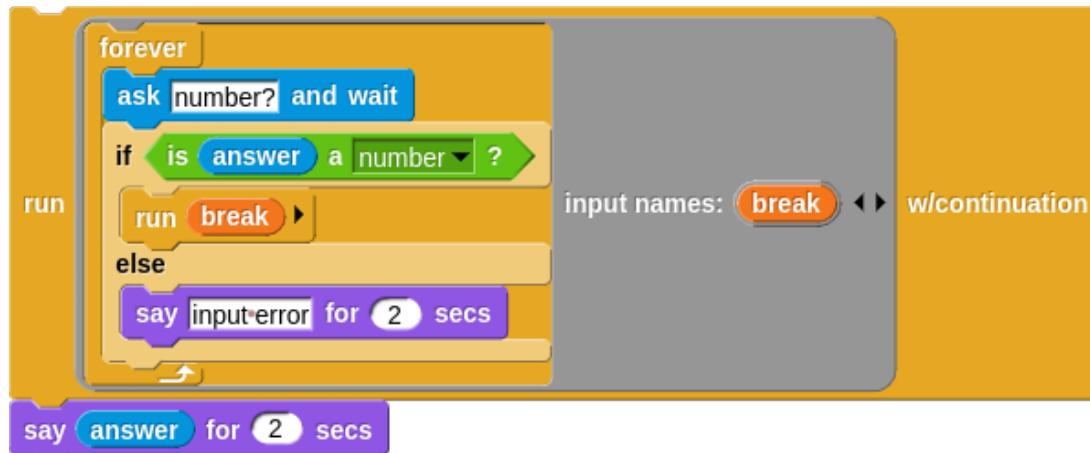
`input names: #1 w/continuation`

中に繰り返しのブロックを入れてやります。その内でフォーマルパラメーター #1 にセットされる継続（このブロックに続くスクリプト）を run すると、繰り返しブロックから脱出して継続するスクリプト動作に移行します。

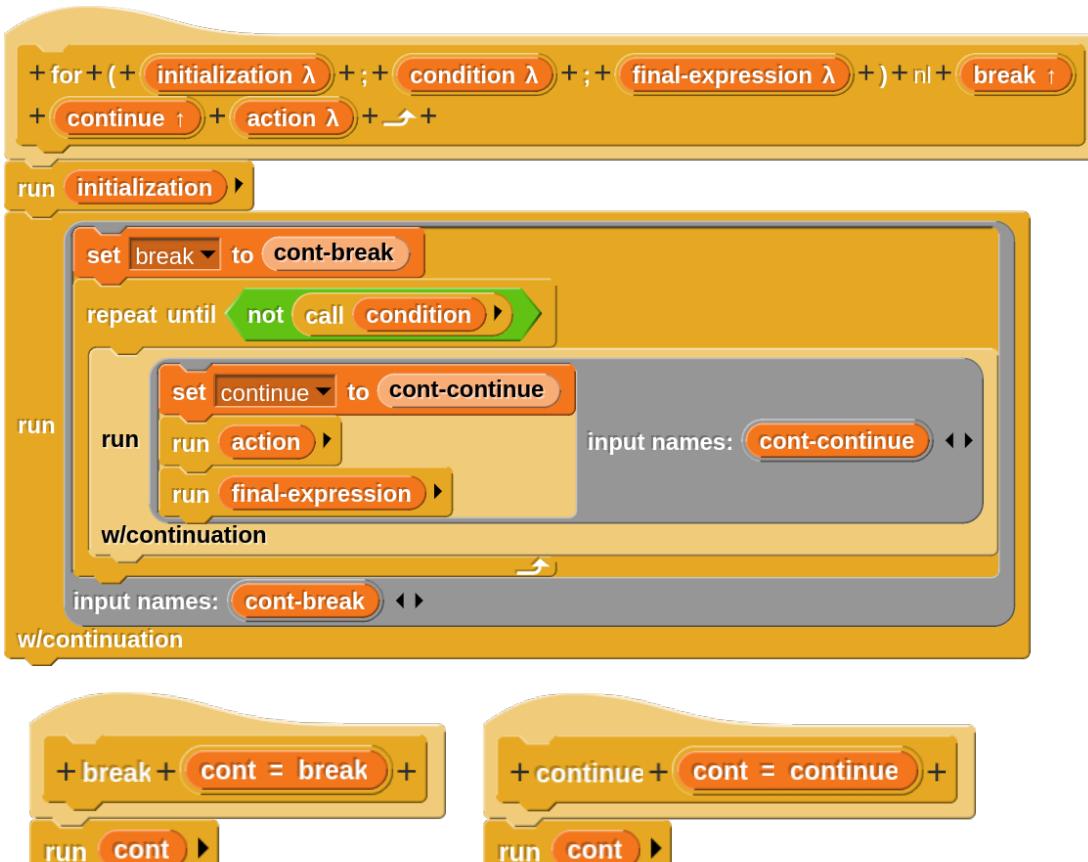
このスクリプトでは、i の値が 3 以上になったら継続スクリプトの実行に移行、つまり繰り返しブロックから脱出します。（最初にフォーマルパラメーター #1 にセットされる継続部分を表示します。）



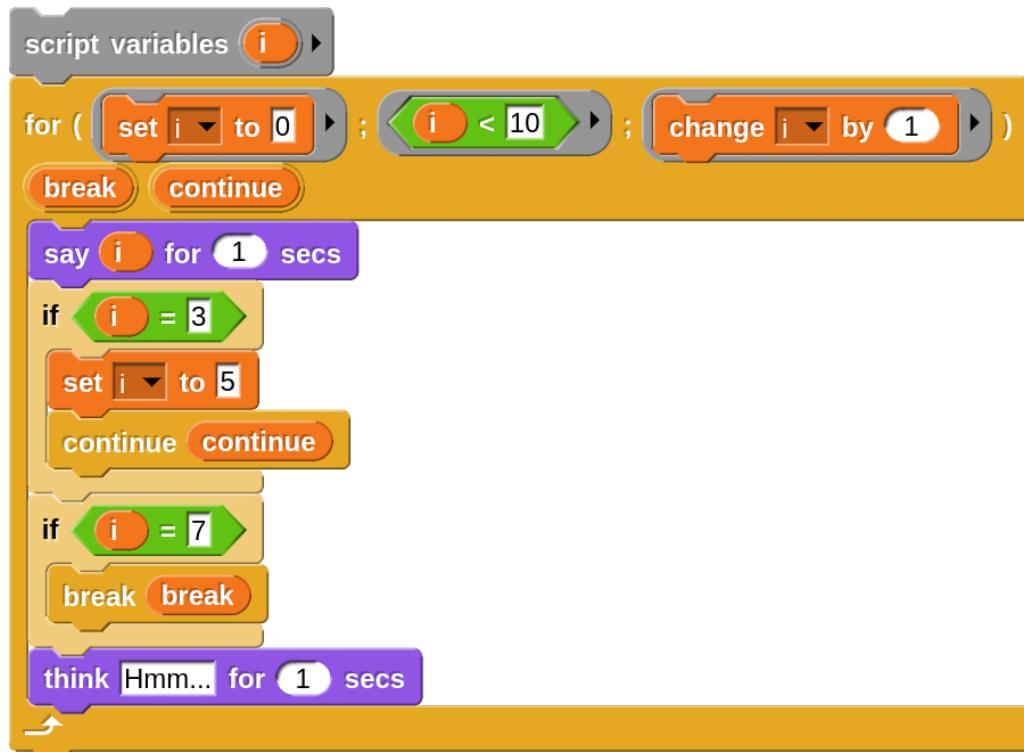
同じような例です。これを使わなくても repeat until でできますが。



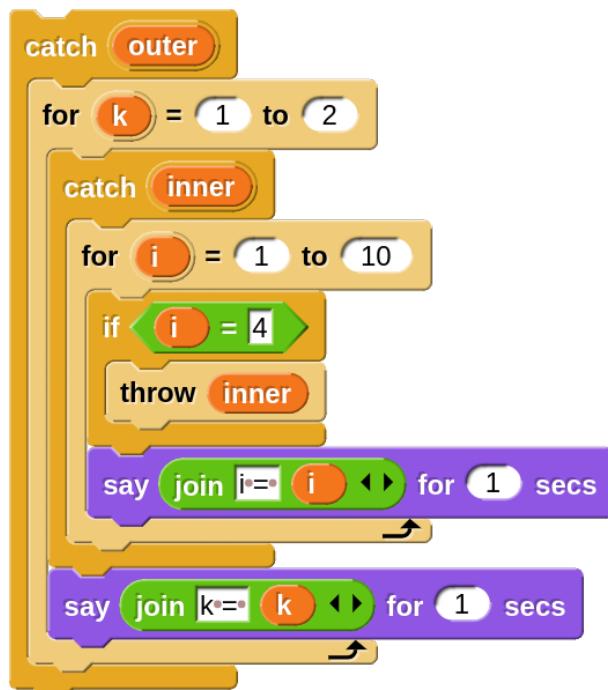
継続の機能を利用すると、C 言語風 for ( ; ; ) ループに break や continue の機能を持たせることができます。break はループから脱出する機能で、continue はループの先頭に戻る機能です。



break と continue は、upvar オプションの変数です。したがって、名前を変更することができます。二重三重のループの場合は、break1, break2 のように使用してください。ジャンプ用のラベルのように外側のループの break, continue を指定することもできます。cont は Any type 変数です。既定値としてそれぞれ文字の break, continue がセットしてありますが、for ( ; ; ) ブロックからそれぞれ break, continue 変数をドロップして使用します。



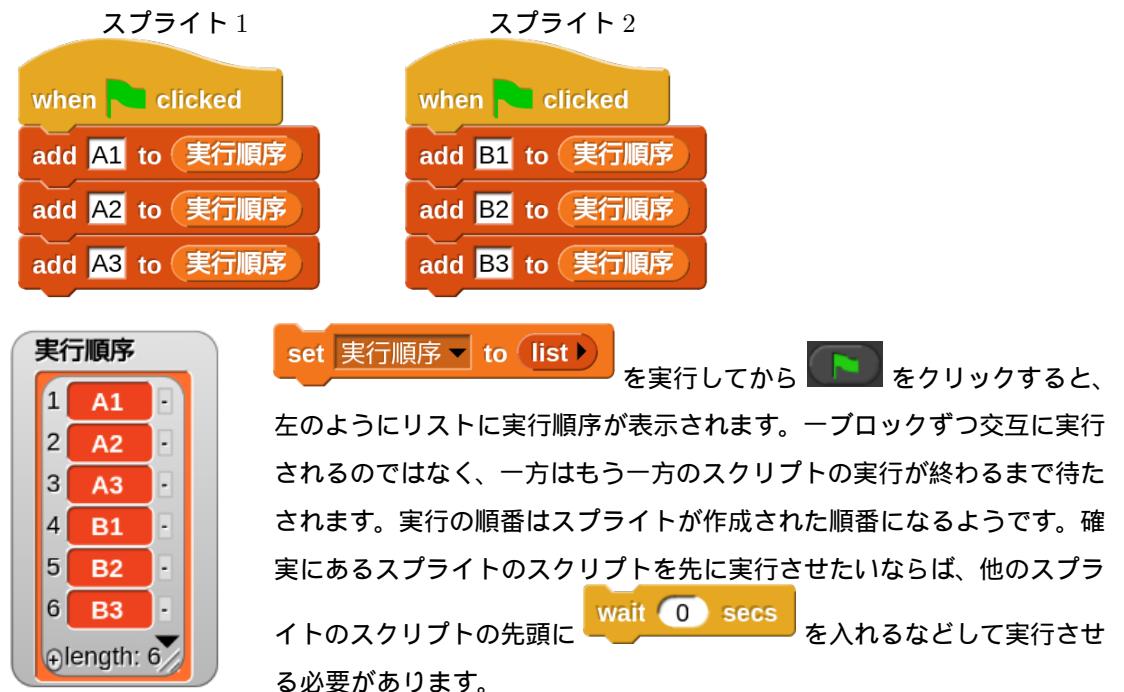
継続の機能をループからの汎用の脱出に応用するのが、Libraries の iteration-composition にある throw と catch です。



これを実行すると、*i* の値が 4 になった時に内側のループから脱出し、  
catch inner の次のスクリプトに進みます。throw outer になると、外側の  
ループから脱出し、catch outer の次のスクリプトに進みます。ある条件の時に、  
throw で指定したラベルの付いた catch ブロックの外側に脱出するという仕組です。

## 1.2 並列処理について

Snap! では、各スプライトに **when green flag clicked** のスクリプトを設定すれば  をクリックすることでそれらの各スクリプトが同時に実行されるように見えます。しかし、実際にはそれぞれを順番に実行しています。**実行順序** の変数を作成し、スプライトを二つ用意してそれぞれに次のスクリプトを作成してください。



スプライト 1  
スプライト 2

実行順序

1	A1
2	A2
3	A3
4	B1
5	B2
6	B3
+ length:	6

set **実行順序** **to** **list** を実行してから  をクリックすると、左のようにリストに実行順序が表示されます。一ブロックずつ交互に実行されるのではなく、一方はもう一方のスクリプトの実行が終わるまで待たれます。実行の順番はスプライトが作成された順番になるようです。確実にあるスプライトのスクリプトを先に実行させたいならば、他のスプライトのスクリプトの先頭に **wait 0 secs** を入れるなどして実行させる必要があります。

スプライト内に複数の **when green flag clicked** のスクリプトがあれば、基本的にはそれがすべて完了してから他のスプライトのスクリプトの実行に移ります。他のスクリプトへの処理移行のタイミングは繰り返し処理、つまり C 型ブロック内の末端に到達した時にも起こります。



スプライト 1  
スプライト 2

実行順序

1	A1
2	B1
3	A2
4	B2
5	A3
6	B3
+ length:	6

for の C 型ブロックにより、A? と B? が交互にリストに加えられました。（? は 1 ~ 3 の数値を表します。）

スプライト 1

```

when green flag clicked
set [実行順序 v] to [list]
for [i = 1 to 3]
  add [join [A1] [i]] to [実行順序]
  add [join [A2] [i]] to [実行順序]

```

スプライト 2

```

when green flag clicked
for [i = 1 to 3]
  add [join [B1] [i]] to [実行順序]
  add [join [B2] [i]] to [実行順序]

```

右のように、今回は A1? A2? と B1? B2? が交互にリストに加えられました。

実行順序	
1	A11
2	A21
3	B11
4	B21
5	A12
6	A22
7	B12
8	B22
9	A13
10	A23
11	B13
12	B23
+length: 12	

次のように、repeat 1 の C 型ブロックで囲んだり wait 0 を入れることでも処理を一ブロックずつ交互に行なうことができるようです。

スプライト 1

```

when green flag clicked
set [実行順序 v] to [list]
for [i = 1 to 3]
  repeat (1)
    add [join [A1] [i]] to [実行順序]
  add [join [A2] [i]] to [実行順序]

```

スプライト 2

```

when green flag clicked
for [i = 1 to 3]
  add [join [B1] [i]] to [実行順序]
  wait (0) secs
  add [join [B2] [i]] to [実行順序]

```

実行順序	
1	A11
2	B11
3	A21
4	B21
5	A12
6	B12
7	A22
8	B22
9	A13
10	B13
11	A23
12	B23
+ length: 12	

右のように、今回は A1? B1? A2? B2? と交互にリストに加えられました。

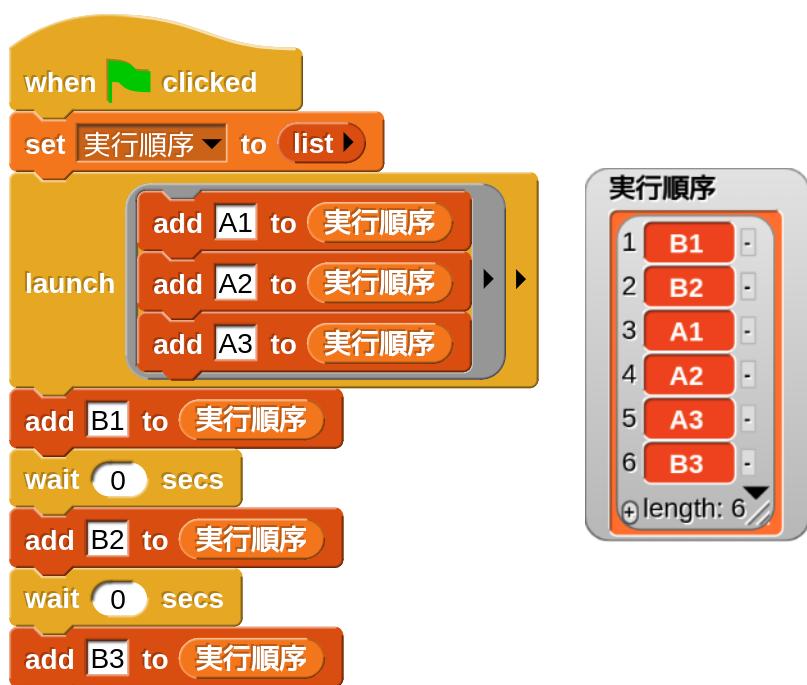
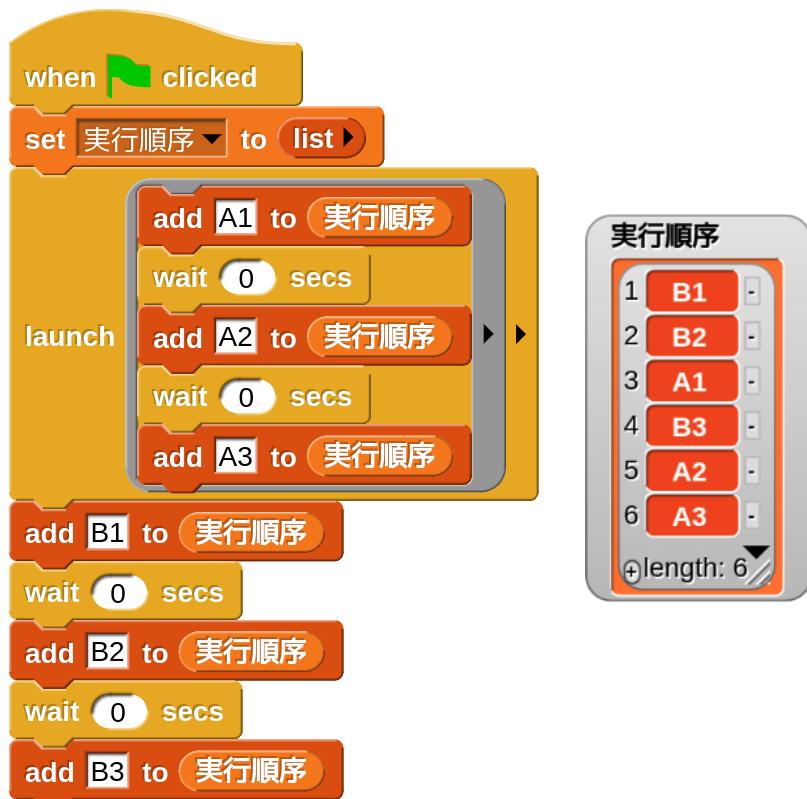
launch での動作です。



実行順序	
1	B1
2	B2
3	B3
4	A1
5	A2
6	A3
+ length: 6	

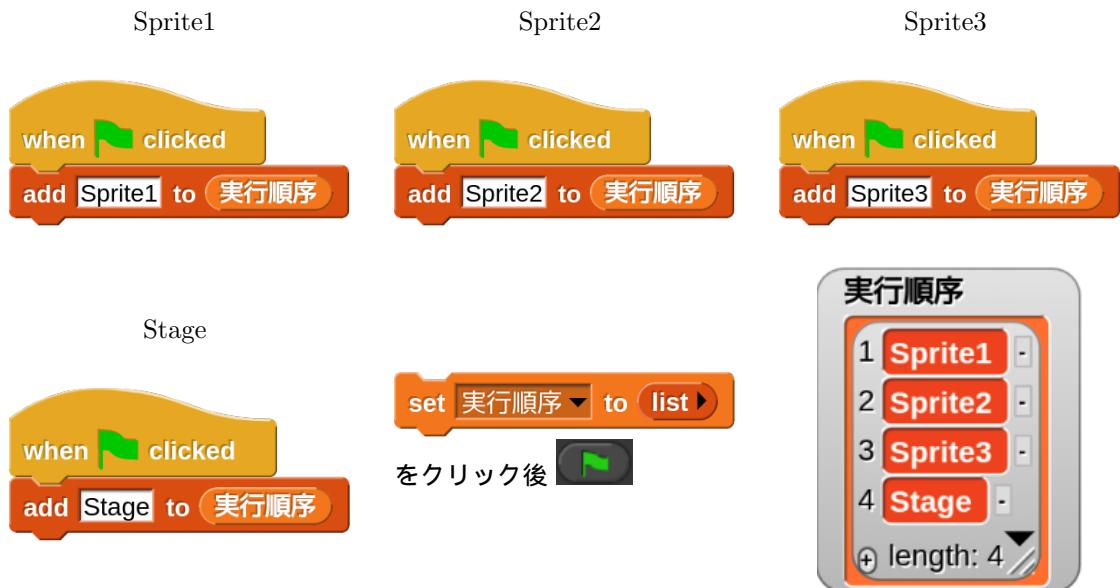


実行順序	
1	B1
2	B2
3	A1
4	B3
5	A2
6	A3
+ length: 6	



スクリプトが思うような動作にならない場合はこのような並列処理が原因になるかもしれません。

C 言語などでは、プログラムは main の関数が実行されます。他の関数などは main から使用されることで実行されます。Snap! では main 関数の役割を Stage に持たせるやり方があります。各スプライトのスクリプトを見てもメインとなる重要なスクリプトが発見できず、Stage のところに隠れている場合があります。プログラムによっていろいろなスプライトが用いられますが、Stage は必ずどんなプログラムにも存在するので、ここにメインのスクリプトを置く理はあります。しかし、Stage は限られたブロックしか使用できないし、次のようにスクリプトの実行順序も最下位になってしまいます。



Stage で初期設定をすると、他のスプライトのスクリプトが実行されてから Stage のスクリプトが実行されるので次のような結果になってしまいます。



したがって、他のスプライトのスクリプトでは **when green flag clicked** を使わないなどの対策が必要になります。

個人的には Snap! 起動時に作成済みの Sprite をメインのスクリプトとするのが無理のないやり方のように思えます。Stage のスクリプトは Stage に関することにのみ使用したほうが分かりやすいです。

ここまで見てきたように、並列処理と言っても Snap! のやり方で順番に実行されているということです。この実行の順番や処理の移行のタイミングを継続を使用してプログラミングすることができます。並列処理される一つのスクリプトを thread (スレッド) と言います。

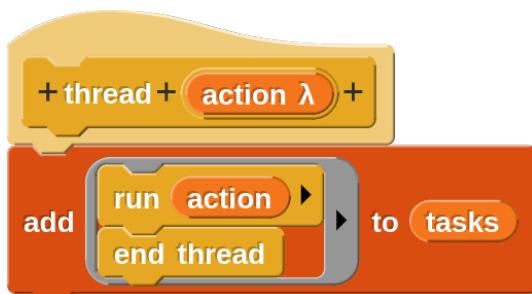
処理するスクリプトの記憶場所として **tasks** の変数を作成します。以下に示す 3 個の定義ブロックは後で例示するスクリプトで共通に使用します。

### thread

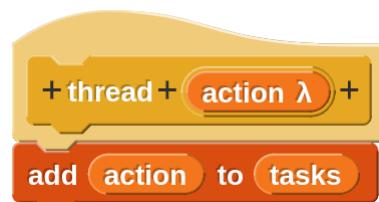
プロックです。スレッドを設定するプロックです。( 設定されたスレッドを tasks リストに加えていきます。)

このプロック自体はスレッドを記憶させるだけで、実行はしません。

action は Command(C-shape) 型です。



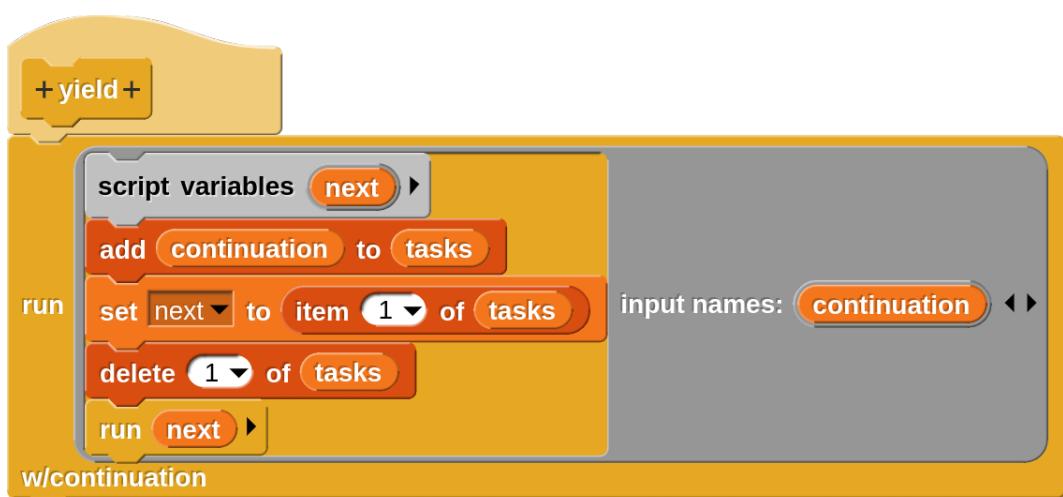
左はマニュアルに掲載されている定義ですが、後で例示するスクリプトの動作においては下の定義でも動くようです。( 非推奨 )



### yield

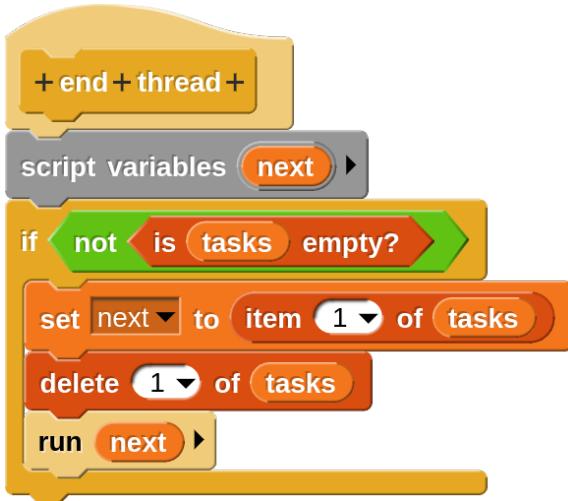
プロックです。( 継続を tasks リストに加え、tasks リストの先頭のスクリプトを取り出して実行します。)

次のスレッドの実行へ移行させます。



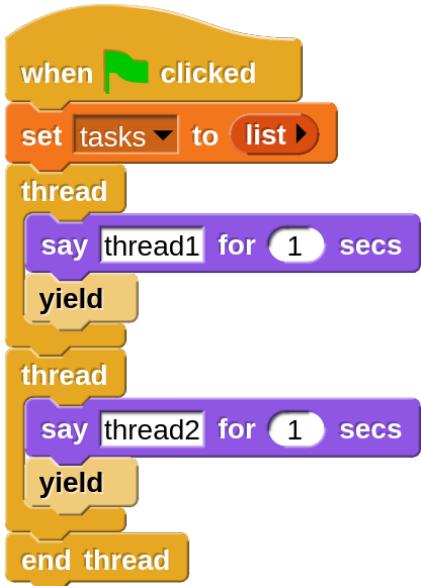
**end thread** ブロックです。( tasks リストが空でなかったら tasks リストの先頭のスクリプトを取り出して実行します。)

これ以上スレッドが無いことを示します。これによって設定された複数のスレッドの実行が始まります。つまり、これを置かないとスレッドが実行されません。



thread ブロックでスレッドを設定して (yield ブロックを挿入することで処理の移行のタイミングを指定します)、スレッドの並びの終わりを示す end thread ブロックを置くというプログラミングになります。継続などを意識しなくても定型的な操作でスレッド処理ができると思います。

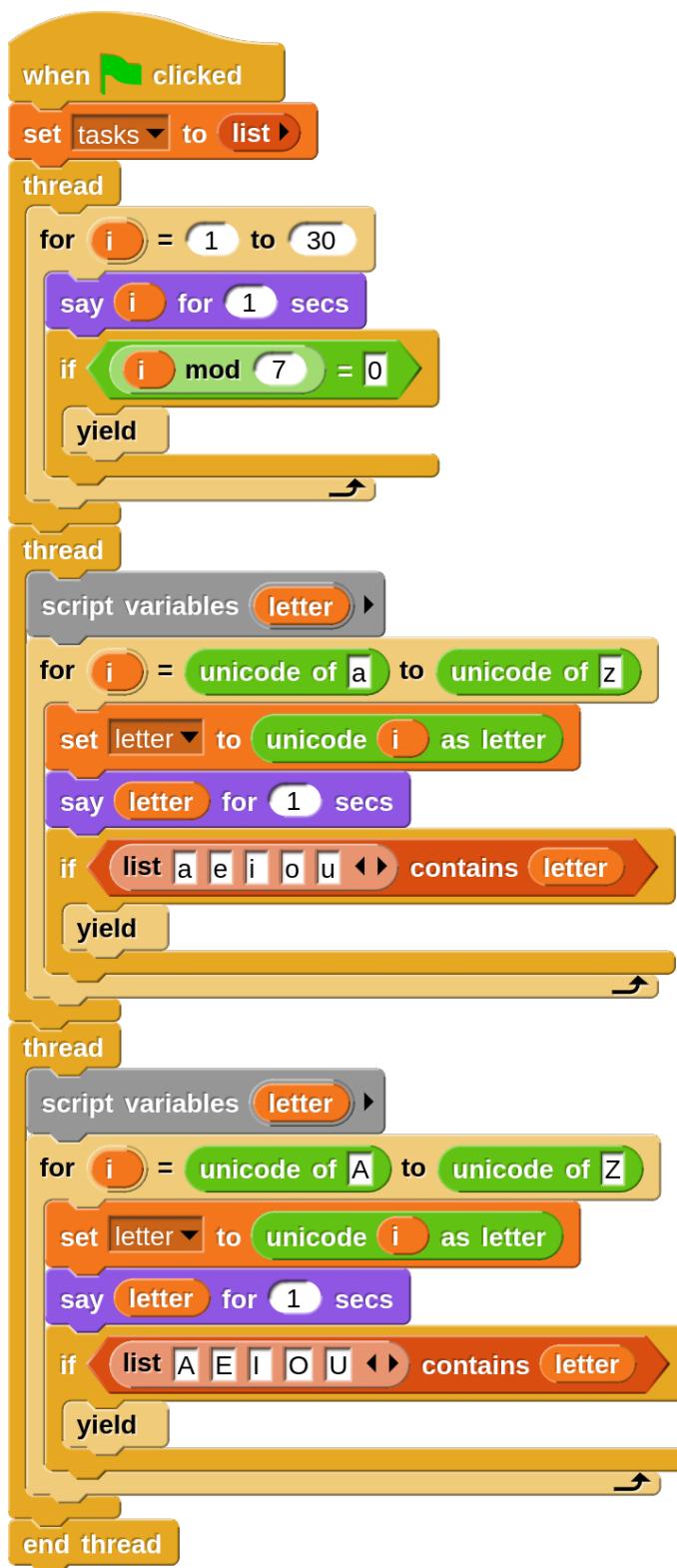
次のスクリプトはスレッド内に繰り返しがないのであまり意味はありませんが、使い方の例です。



番号を表示するだけの 3 個のスレッドを実行してみます。



マニュアルに掲載されているスクリプトを元にした例です。指定の条件の時に他の処理に移行します。



## 2 再帰

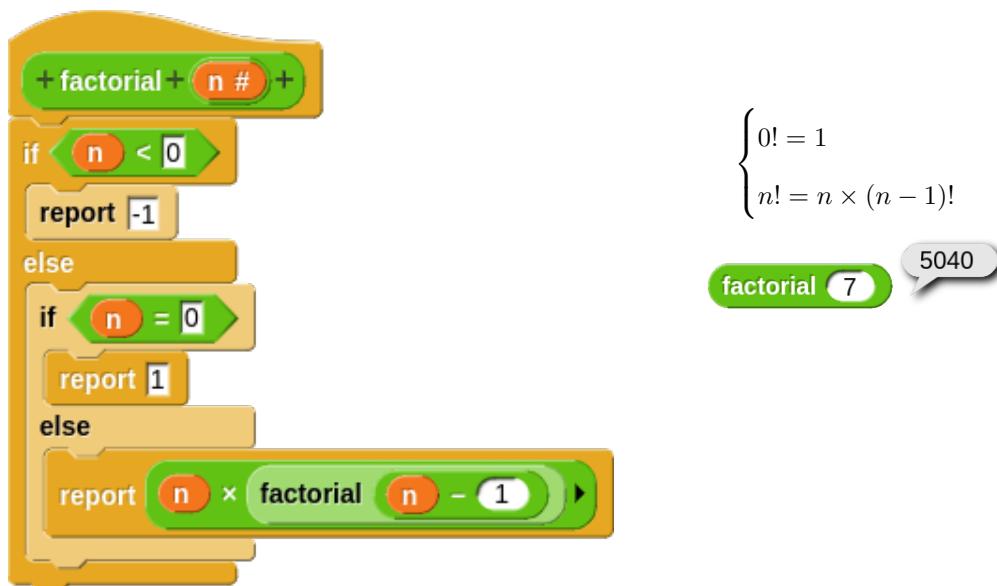
再帰、再帰呼出し（recursive call）は作成したブロックの中で自分自身を呼び出す（実行する）ものです。関数型プログラミングでは繰り返し処理の手法として再帰を使うことは一般的で、効率的だったりします。

### 2.1 再帰の例

Scratch では値を返せなかったので、階乗やフィボナッチ数列はできませんでした。

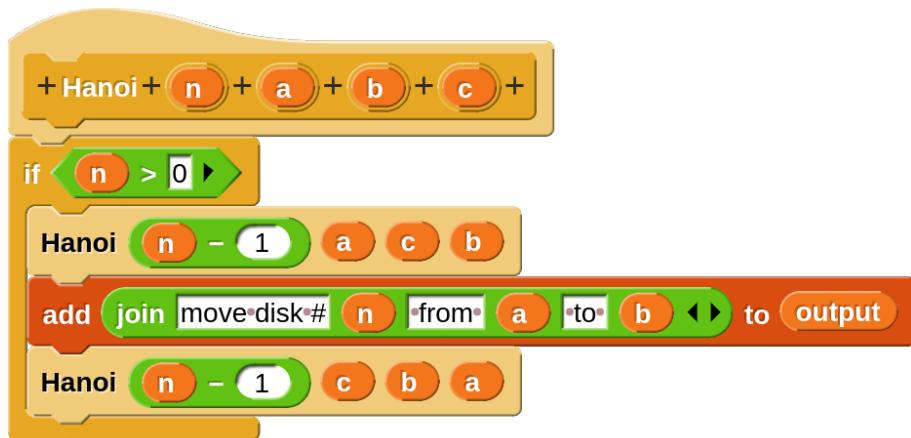
#### 2.1.1 階乗

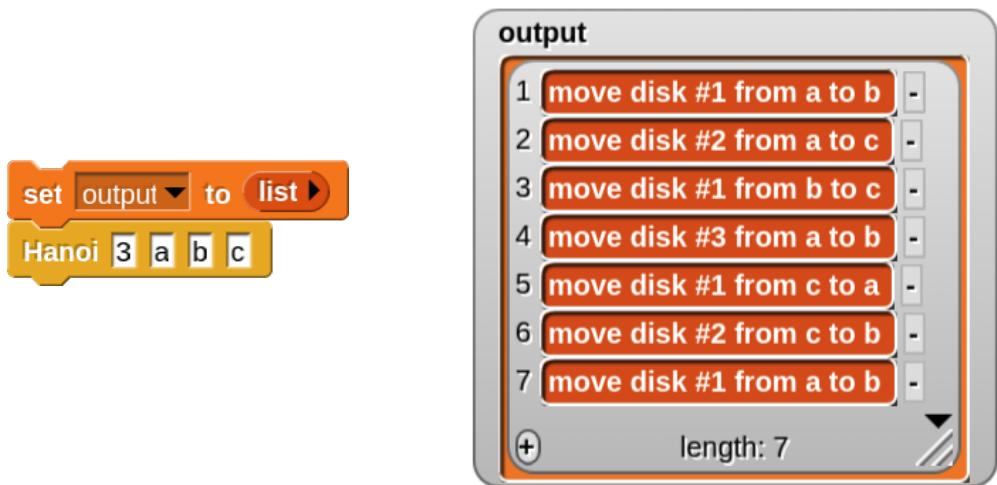
factorial 階乗は再帰の例としてよく使用されます。



#### 2.1.2 ハノイの塔

C 言語などで書かれたプログラムも出力を工夫すれば Snap! スクリプトにすることができます。

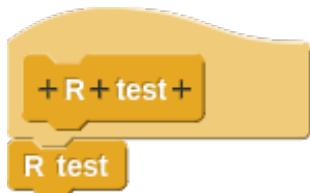




## 2.2 再帰の使用

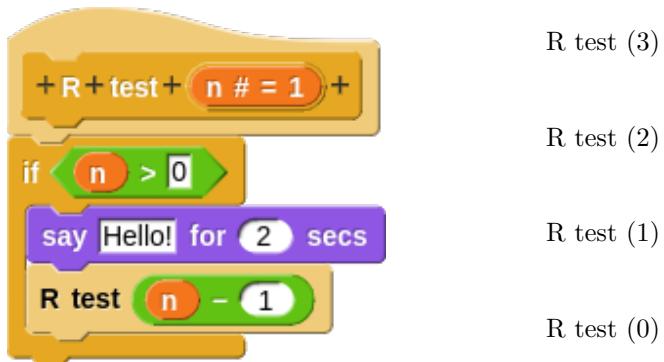
### 2.2.1 繰り返し

まずはただ自分自身を呼び出してみます。(実行しないでください)

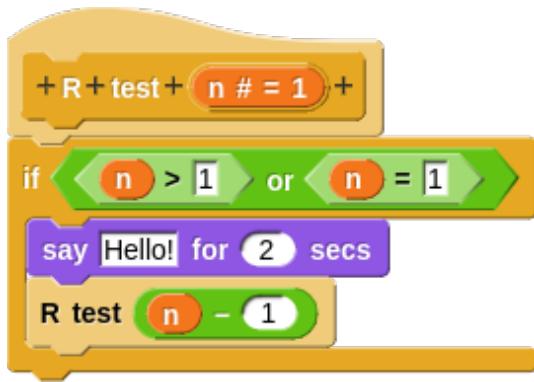


再帰呼び出しが終了するコードブロックがないので無限ループになります。普通の処理系ではスタックオーバーフローでエラー終了かフリーズします。

次は指定した回数だけ「Hello!」と言う定義ブロックです。n の値を減らしながら以下の矢印(→)のように自分自身を呼び出していくきます。この定義ブロックは 0 以下だと何もしないで以下の矢印(↑)のように呼び出し元に戻ります。呼び出し元に戻るを繰り返し、一番最初の呼び出し元に戻ったら終了です。



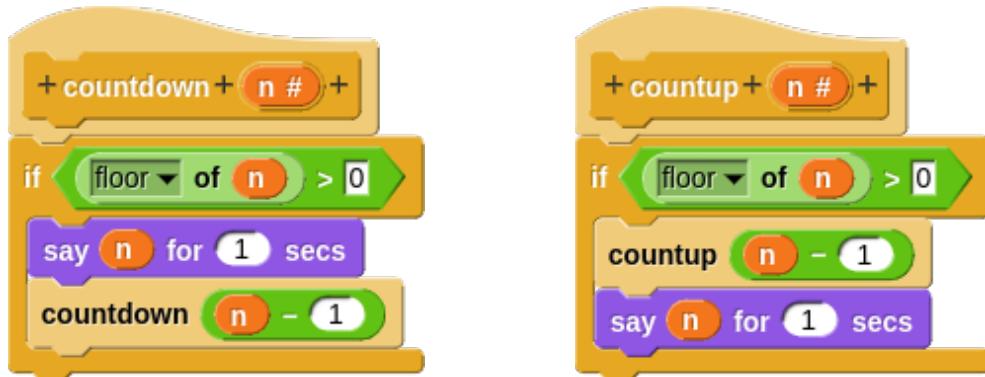
回数に 0.5 を指定しても実行されるので終了条件を変更します。



### 2.2.2 カウントダウンとカウントアップ

再帰を使ってカウントダウンとカウントアップを実行してみます。終了条件の指定方法を少し変えています。使用しているブロックはどちらも同じなのですが、組み合わせる順序によってダウンにもアップにもなります。

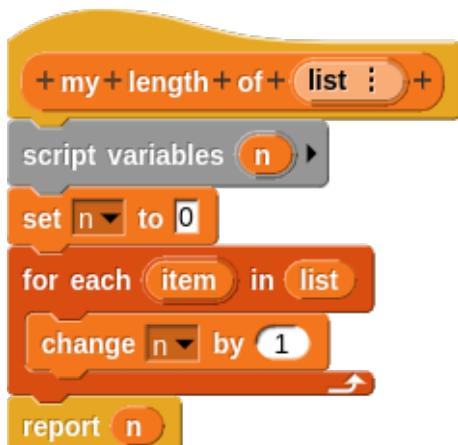
`say` ブロックのところに処理したいコードブロックを持ってくれば繰り返しの処理ができます。



### 2.2.3 my length

リストの要素数を求める `length` ブロックを作ってみます。

普通に考えると `for each` ブロックを使うやり方になると思います。



my length of list 0 my length of numbers from 1 to 3 3

処理にかかる時間を表示します。

reset timer  
say my length of numbers from 1 to 1000  
report timer 16.8

これを repeat until ブロックを使ってやってみます。要素数が 0 になるまで要素を一つずつ削除しながらカウントすることで求めます。

+ my + length + of + list :  
script variables n  
set n to 0  
repeat until is list empty?  
  change n by 1  
  set list to all but first of list  
report n

処理にかかる時間を表示します。

reset timer  
say my length of numbers from 1 to 1000  
report timer 16.7

再帰版です。カウント用の変数が無いので理解しにくいですが、report が返す値がカウント用変数の役割を果たしています。 my length of all but first of list でリストが空になるまで再帰呼び出しされて、0, 1, 2, ... と、report が返す値+1 を積み重ねて、結果的に 0 からのカウントアップで要素数を求めることができます。

[リストの先頭要素を処理する。2番目以降のリストを引数として自分自身を呼び出す] ということをリストが空になるまで、つまり、リストのすべての要素に対して処理を行うのが再帰処理の基本です。

この場合の処理は、リストの要素の値は使用せずにリポートされた値に 1 を加えているだけですが。

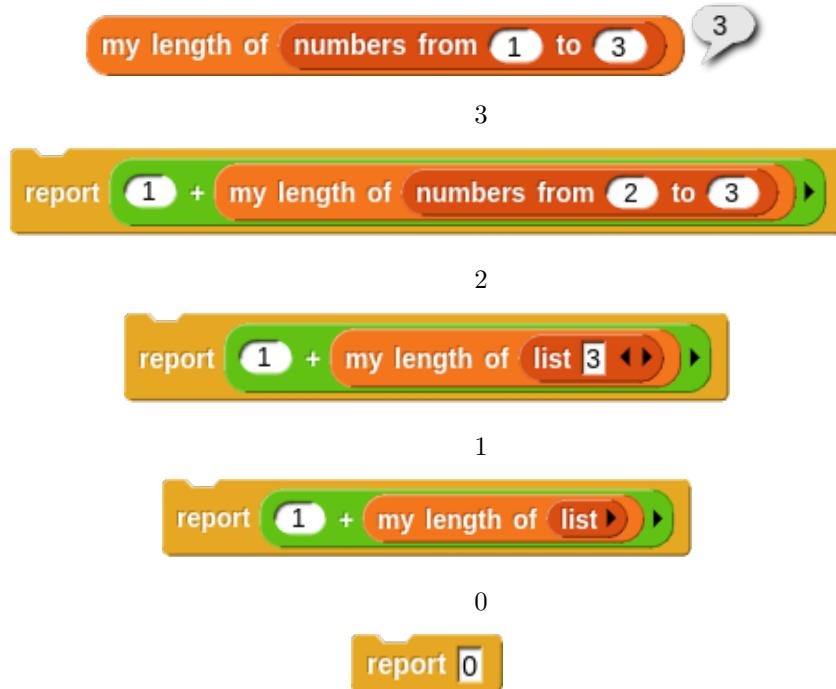


処理にかかる時間を表示します。



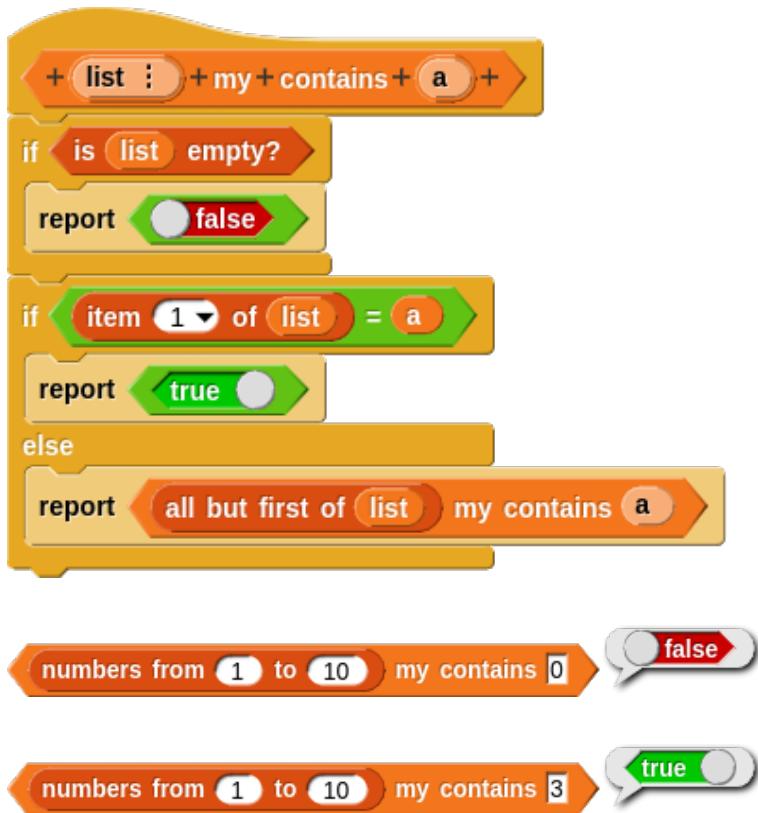
再帰呼び出しを使わないものに比べてとても効率が良いことがわかります。

実行される様子を見てみます。 で示す順に再帰呼び出しが実行され、0と値が確定すると、  
で示す順に返された値に1を加えて呼び出し元に値を返していきます。最終的に値は3になります。



#### 2.2.4 my contains

リストの中に指定の要素が存在するかを求める contains ブロックを作ってみます。



### 2.2.5 リスト要素の巡回

要素にリストを含むリストに対して length を使用すると、内部のリストの分はカウントしません。



これは my length も同様です。



再帰を使って内部の要素に対してもアクセスしてみます。

処理の内容は次のようにになります。

- もしリストが空ならば 0 をリポートする。
- もし先頭の要素がリストならば、そのリストに my length2 をしたものと残りに対して my length2 をしたものを加える。
- そうじゃなかったら、残りに対して my length2 をしたものに 1 を加える。

```

+ my length2 of list +
if is list empty?
report 0
if is item 1 of list a list ?
report
my length2 of item 1 of list + my length2 of all but first of list
else
report 1 + my length2 of all but first of list

```

my length2 of list list 1 2 3 list 4 5 list 6 7 8 8

1を加えるのではなく、要素の値を加えるようにすると合計を求めることができます。

```

+ sum of list +
if is list empty?
report 0
if is item 1 of list a list ?
report sum of list item 1 of list + sum of list all but first of list
else
report item 1 of list + sum of list all but first of list

```

sum of list list 1 2 3 list 4 5 list 6 7 list 8 9 10 55

再帰処理は理解しにくいと思いますが、my length2 のような処理の場合、再帰処理を使わいでやる方法を考えるのは難しい気がします。

## 2.2.6 reverse 逆順リスト

リスト要素の逆順リストリポートを再帰呼出しで行うことができます。

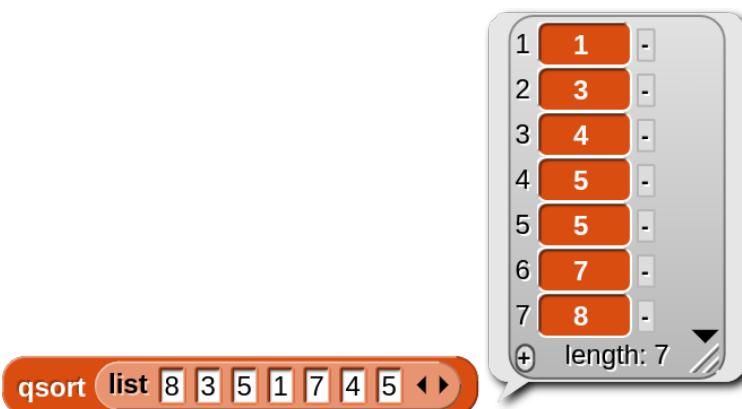
リストから先頭の要素を取り出して〔残りのリスト〕の最後尾に加えます。その〔残りのリスト〕に対して同じ操作をしていけば逆順のリストが得られます。

```
[1, 2, 3, 4, 5] -> [2, 3, 4, 5] + [1]
                      [3, 4, 5] + [2]
                      [4, 5] + [3]
                      [5] + [4]
[ ] + [5]
[5, 4, 3, 2, 1]
```



### 2.2.7 クイックソート（整列 / 並べ替え）

クイックソートのアルゴリズムは有名でよく題材として扱われています。リストの中から任意の値を選び、それよりも小さい値のグループ、その値、大きい値のグループに振り分ければ選択された値の位置付けができます。この操作を小さい値のグループ、大きい値のグループに対して再帰的に繰り返していくば最終的に並べ替えが完了します。任意の値の選び方として random ブロックを使いましたが、先頭の値でも構いません。Snap! には keep ブロックがあるので、アルゴリズムをそのまま表したように割と分かりやすいスクリプトが作れます。



#### [参考文献]

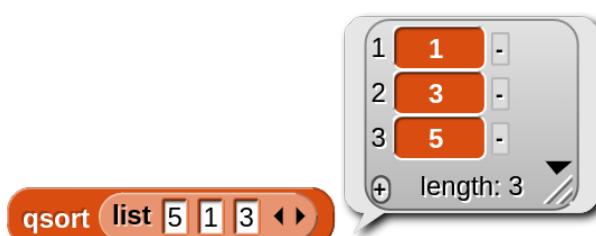
『Programming in Haskell, 2nd edition』

Graham Hutton 著 山本和彦 訳 ラムダノート株式会社 刊

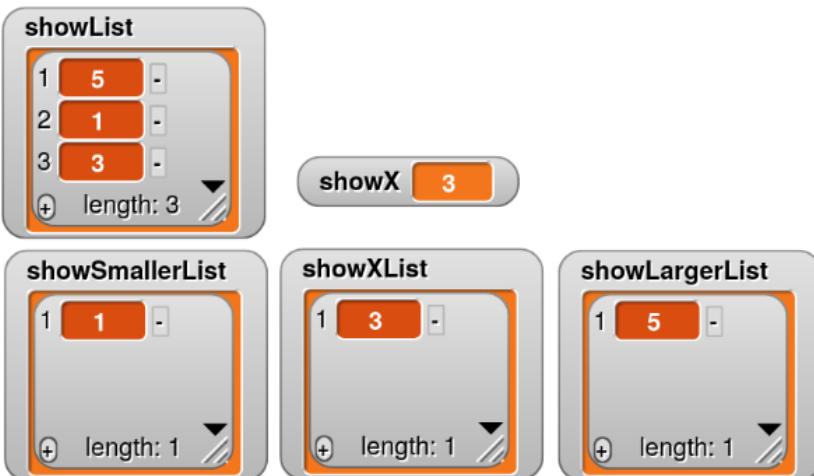
次のように作成したリストを通して値を表示すると操作の様子が見られます。



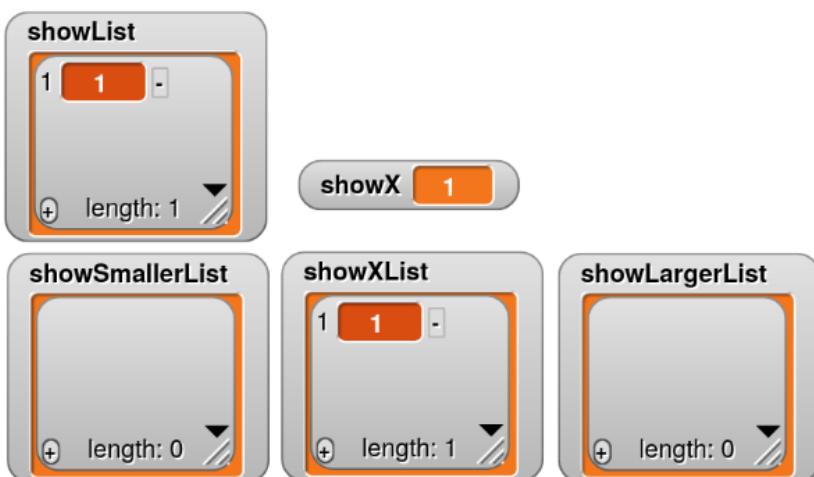
リストの値表示のたびに pause all になります。 をクリックして続行してください。



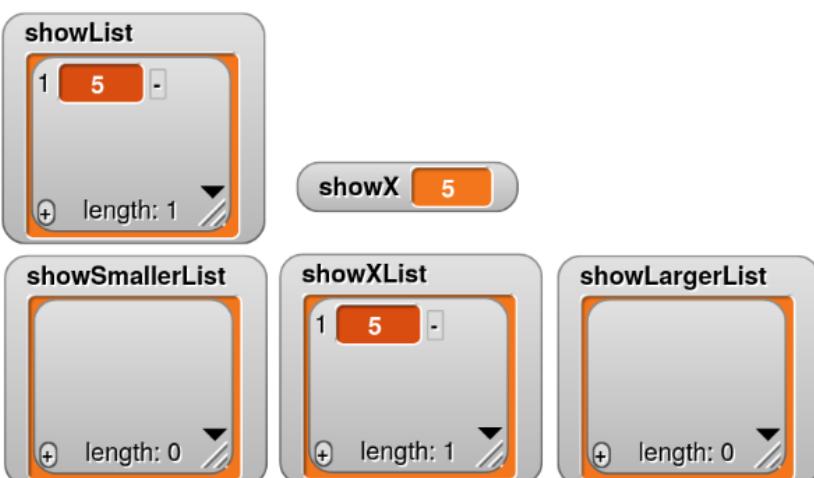
最初に選択された値が 3 だった場合です。



3 より小さい値のグループ処理



3 より大きい値のグループ処理

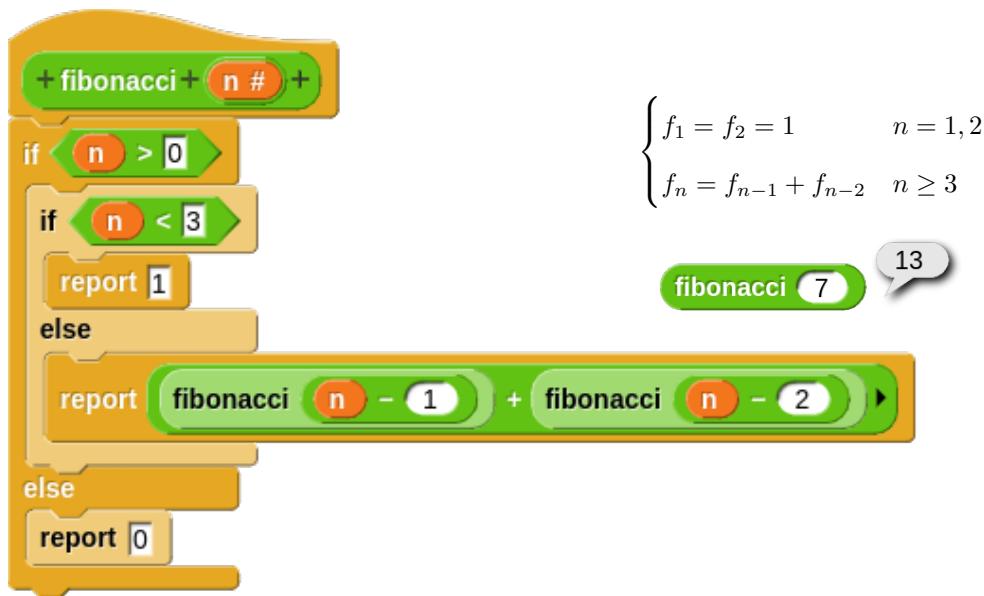


この場合要素数は 1 でしたが、それぞれのグループに対して再帰的に処理されます。

## 2.2.8 再帰呼び出しをする局所的な定義ブロック

Snap! ではスクリプト変数にスクリプトブロックを設定することができます。それを局所的な定義ブロックとして使用することができます。局所的というのは、正式に定義ブロックとして登録するのではなく、その場所だけで利用するということです。

再帰の例としてよく示されるフィボナッチ数列で、局所的な定義ブロックを使用してみます。まずは一般的なやり方です。

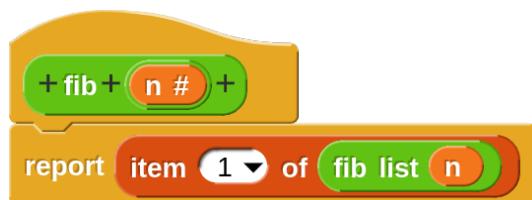


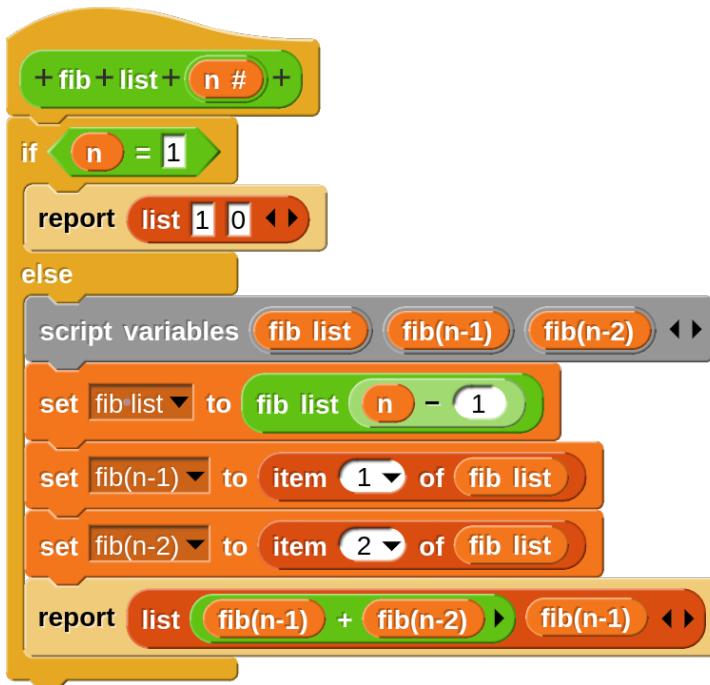
これは  $(n - 1)$  と  $(n - 2)$  を引数にして 2 度再帰呼び出しています。そのため、 $n$  が大きくなると時間がかかるってしまいます。

例えば  $\text{fib}(6)$  を求める場合は、右のようになりますが、 $\text{fib}(6)$  で必要とする  $\text{fib}(4)$  は  $\text{fib}(5)$  で処理済みで、 $\text{fib}(5)$  で必要とする  $\text{fib}(3)$  は  $\text{fib}(4)$  で処理済み … ということで、処理済みのものはその値を渡してやれば済むのでその分の再帰呼び出しの必要がなくなります。

$$\begin{aligned} \text{fib}(6) &= \text{fib}(5) + \text{fib}(4) \\ \text{fib}(5) &= \text{fib}(4) + \text{fib}(3) \\ \text{fib}(4) &= \text{fib}(3) + \text{fib}(2) \\ \text{fib}(3) &= \text{fib}(2) + \text{fib}(1) \\ \text{fib}(2) &= 1 \\ \text{fib}(1) &= 1 \end{aligned}$$

$\text{fib}(n)$  と  $\text{fib}(n-1)$  の値をリストにしてリポートするブロックを使用するバージョンです。再帰呼び出しその定義ブロックです。`fib ()` で、受け取った  $\text{fib}(n)$  の値をリポートするという二段構えになっています。

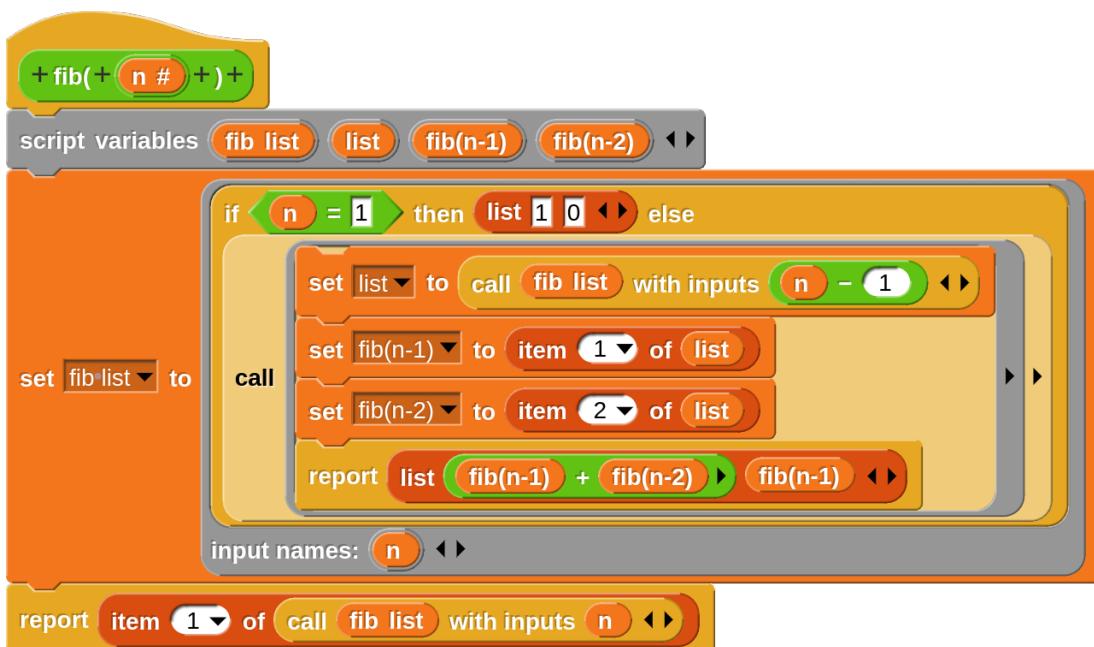




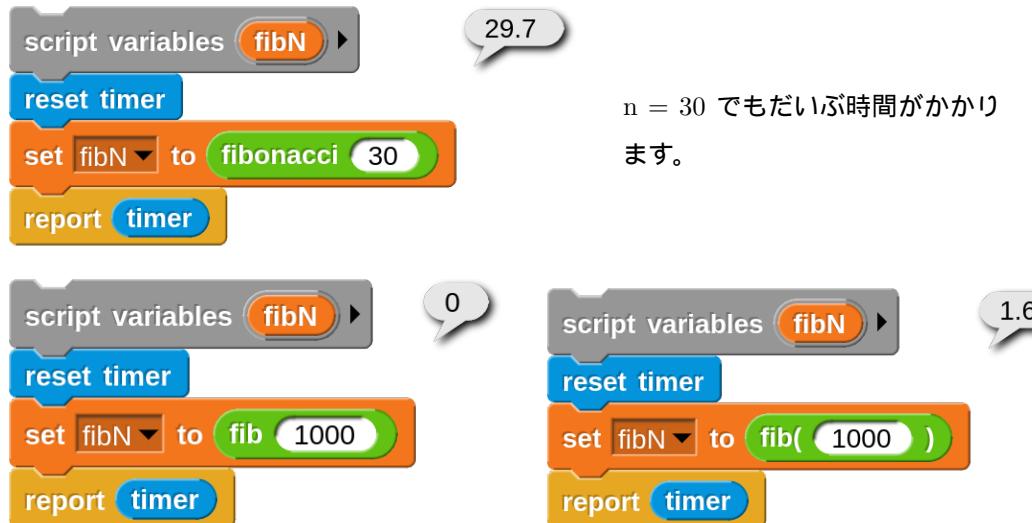
右の表は  $n$  の値に対する  $\text{fib}$  と  $\text{fib list}$  の値です。( は前の項のという意味です。)  $\text{fib}$  の値は、 $\text{fib list}$  がリポートする値であるリストの 1 番目の値になります。その 2 番目の値は  $\text{fib}(n - 1)$  の値です。これによって処理済みの  $\text{fib}$  値を渡していきます。

n	fib	fib list
1	1	list(1, 0)
2	1	list(1, 1) ( 1 + 0, 1)
3	2	list(2, 1) ( 1 + 1, 1)
4	3	list(3, 2) ( 2 + 1, 2)
5	5	list(5, 3) ( 3 + 2, 3)
6	8	list(8, 5) ( 5 + 3, 5)

$\text{fib list}()$  の定義のスクリプトを  $\text{fib}()$  の中でローカル変数にセットすれば局所的な定義プロックになります。



三種類のそれぞれにかかる時間を表示してみます。ただし、fibonacci はとても時間がかかるので  $n = 30$  です。

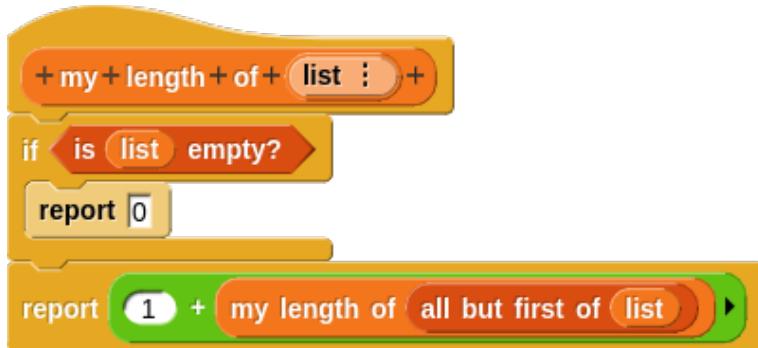


#### [ 参考文献 ]

『プログラミング in OCaml  
～関数型プログラミングの基礎から GUI 構築まで～』  
五十嵐淳 著  
技術評論社 刊

### 2.2.9 末尾再帰

19 ページで my length を扱いました。



この定義ブロックでは `report (1) + (my length of all but first of list)` のように再帰呼出しが計算式に含まれるので、再帰呼出しによる値が確定すると、順々に確定した値による計算値を戻しながら一番最初のところまで戻ることになります。

$L(1, 2, 3)$

[

1 + L(2, 3)

1 + L(3)

1 + L()

1 + 0

1 + 1

1 + 2

3

]

これに対して、次のように再帰呼出しの結果がそのままその定義ブロックの値になるようにします。



引数  $(n + 1)$  を渡していくことでカウントしています。

これを呼び出す本体定義は次のようにになります。



引数 n の値を 0 にすることで、カウンターの値を初期化します。



`L(1, 2, 3)(0)`

[

`L(2, 3)(1)`

`L(3)(2)`

`L()(3)`

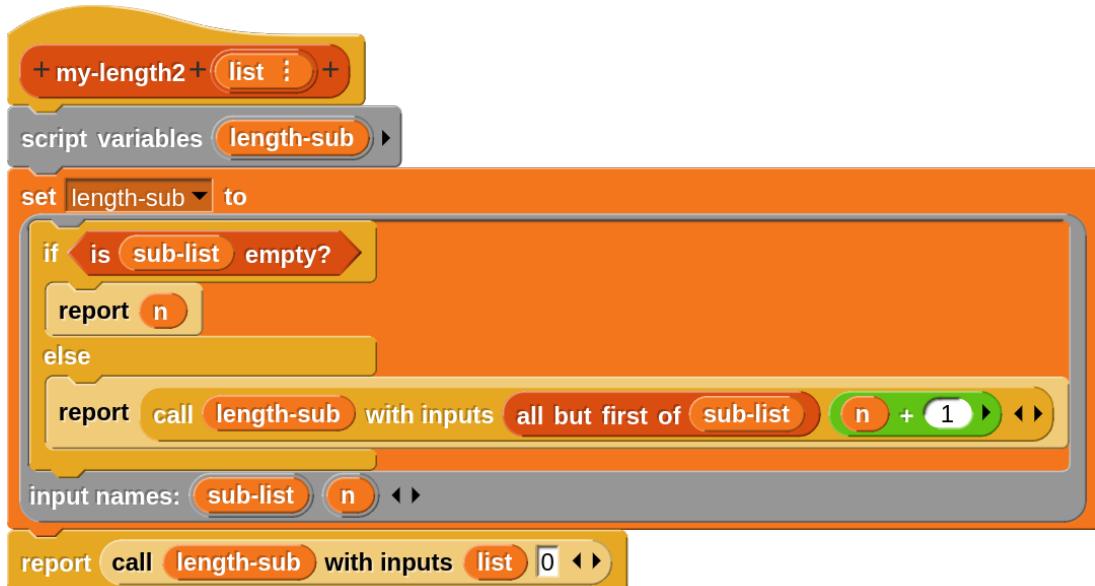
`3`

]

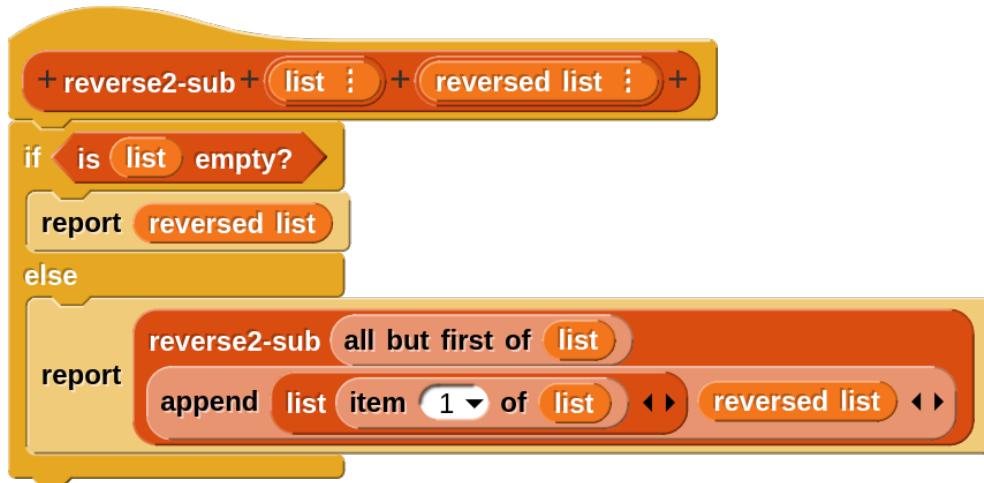
このように、再帰呼出しの結果がそのまま定義ブロックの値になる形を末尾再帰といいます。

Scheme では、末尾再帰はループに最適化されるので効率がよくなります。

my-length-sub はここでしか使ないので局所定義ブロックにすると、次のようにになります。



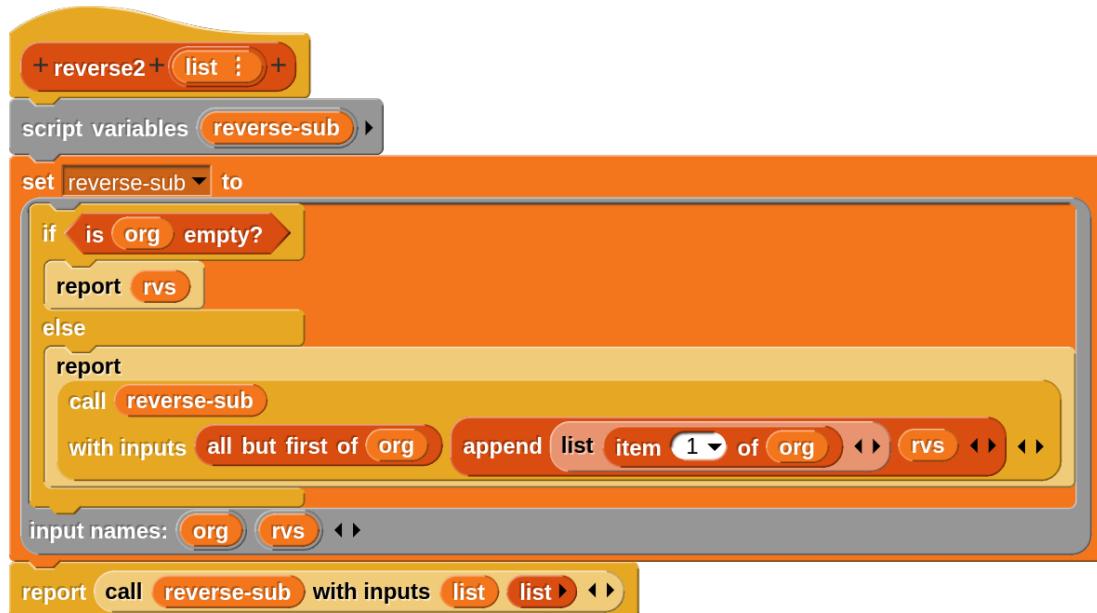
reverse の末尾再帰版です。



これを呼び出す本体定義は次のようにになります。



局所定義ブロック版です。

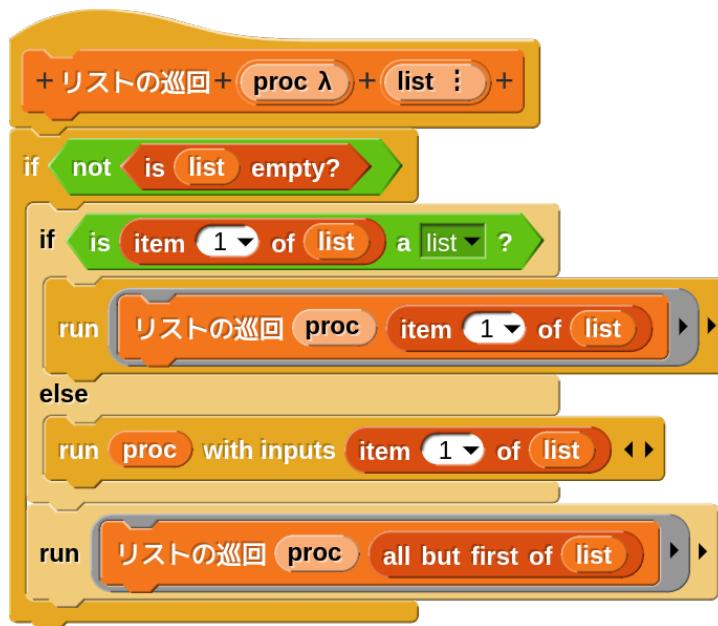


### 3 高階関数

関数の引数や戻り値に関数を指定できるものを高階関数と言います。Snap! のユーザー定義ブロックはこの性質を持ちます。マニュアルでは first class と表現されています。リスト操作に使用する map, keep, find, combine ブロックは入力スロットにリングがあり、引数にスクリプトブロックを指定することを示しています。スクリプトブロックを ringify リングで囲ってやればそのスクリプトブロック自体を戻り値としてリポートすることができます。

入力スロットにスクリプトブロックを指定する例を示してみます。

22 ページで「リスト要素の巡回」を扱いました。やり方を変えることで要素数を求めたり要素の値の合計値を求めることができます。引数で操作を指定すれば同じ定義ブロックでいろいろな用途に使えます。定義ブロック自体は Command 型で、引数 proc は Command(inline) 型です。proc は引数を一つ使用することができます。



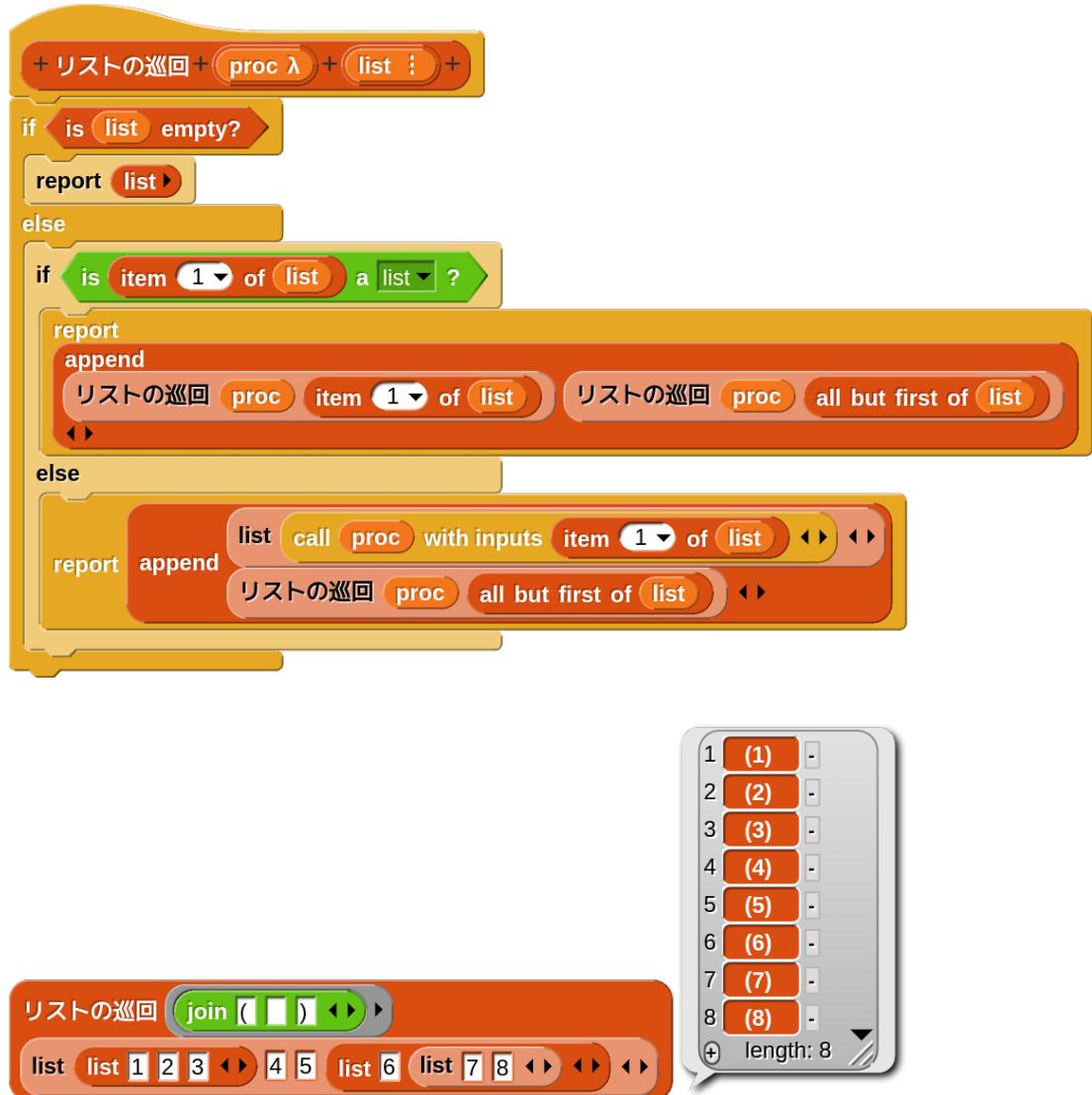
リストの要素数を求めます。



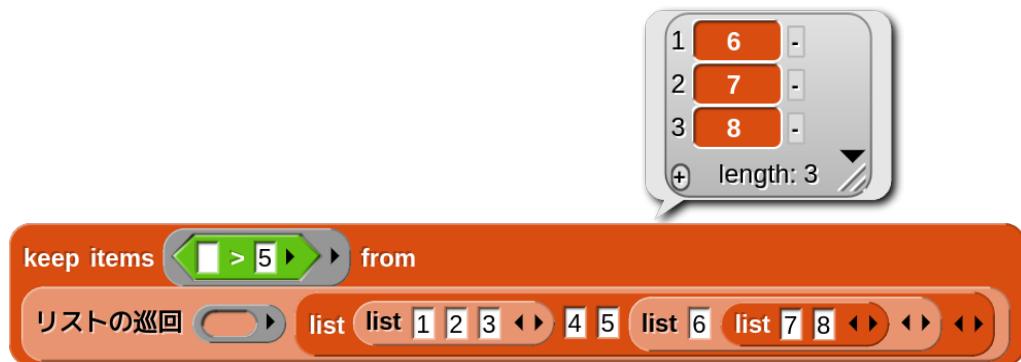
リストの要素の値の合計値を求めます。



リポーター版です。引数 proc は Reporter 型です。



proc を空にすると、リストだけを渡すことになります。

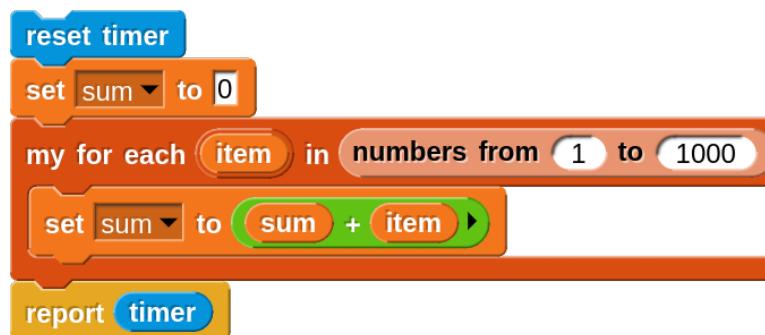
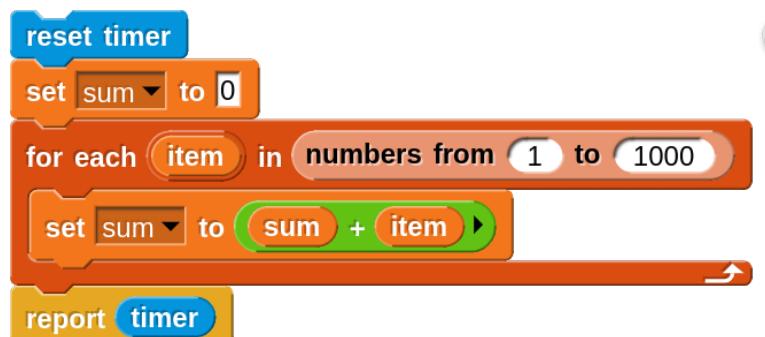
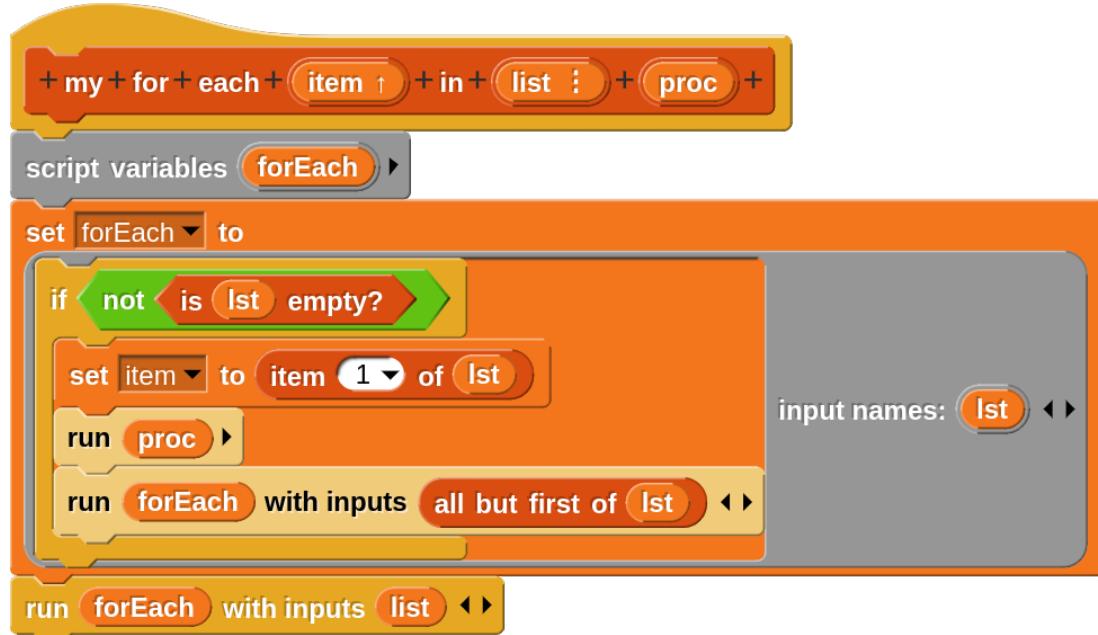


for each item in 目

→ を再帰処理で行うと高速にできます。

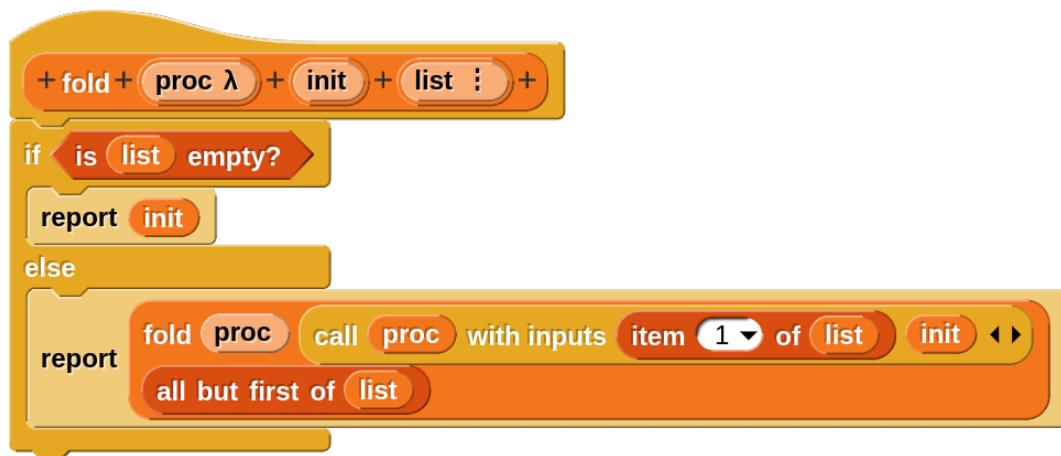
item は [ Upvar-make internal variable visible to caller ] のオプションで作成します。

proc は [ Command (c-shape) ] で作成します。



Gauche で組み込みになっている fold というブロックを作成してみます。

引数 proc は Reporter 型です。



これは、proc が二つの引数をとる操作ブロックで、init と [list の先頭要素] に対して操作します。その値を init にし、[list の次の要素] を引数として proc を実行 ... ということを list の終わりまで行います。

fold の場合、

init	list の先頭	値
0	1	1
1	2	3
3	3	6
6	空	

init(0) + item(1) = 値 1 が次の init の値になる  
init(1) + item(2) = 値 3 が次の init の値になる  
init(3) + item(3) = 値 6 が次の init の値になる  
list が空なので終了 : init の値 6 をリポートする

fold となります。fold では、操作の結果の値が次の操作への第二引数になります。 6

これは、combine list 1 2 3 using 6 と同じ結果ですが、

6 と、操作内容は違います。  
 演算子を使うと違いはあきらかになります。

combine numbers from 1 to 3 using -4

これは、 -4 ということです。

fold 2

これは、 2 ということです。

append を使用すると、次のようにになります。

A Scratch script consisting of the following blocks:

- fold
- append
- list [ ]
- list [ ]
- numbers from 1 to 3

The list [ ] block contains three orange rectangles labeled 1, 2, and 3. A callout box shows the list with the value 3 at index 1, 2 at index 2, and 1 at index 3, with a total length of 3.

fold が（リストの要素 演算子 init）という引数の配置を取るためです。

リストの結合を ++ で表してみます。

fold			
item	-	init	値
1	-	0	1
2	-	1	1
3	-	1	2
		2	2

fold			
item	++	init	値
1	++		1
2	++	1	21
3	++	21	321
		321	321

fold の [ proc ] の操作ブロックは通常は二つの引数を使用します。指定しない場合、その引数は次のようにセットされます。

A Scratch script consisting of the following blocks:

- fold
- append
- list [item init]
- input names: item init
- list [1 2 3]

次のように指定を変更することもできます。

A Scratch script consisting of the following blocks:

- fold
- append
- init
- list [item]
- input names: item init
- list [1 2 3]

次のようにすると、リストの要素の値は使用しないで要素数 length を求めることになります。

A Scratch script consisting of the following blocks:

- fold
- init
- + [1 v]
- input names: item init
- 0
- list [a b c]

A speech bubble contains the number 3.

次のようにすると最大値を求めることができます。

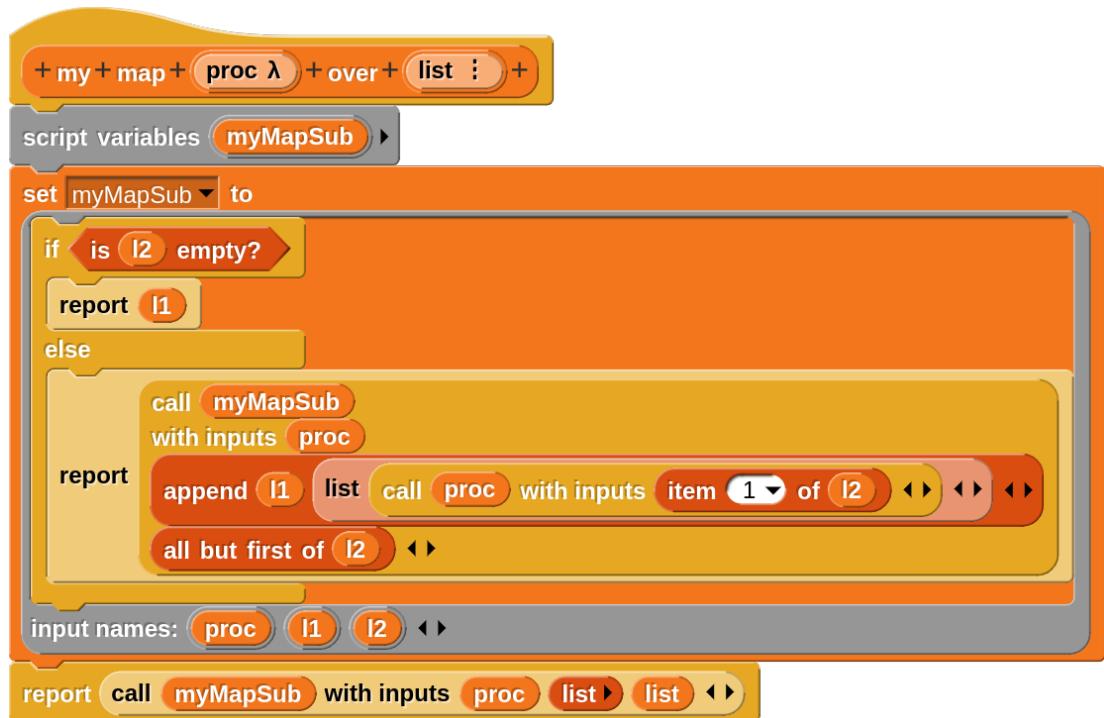
A Scratch script consisting of the following blocks:

- fold
- if <item> > <init>
- then
- item
- else
- init
- input names: item init
- 999
- list [8 3 12 10 5]

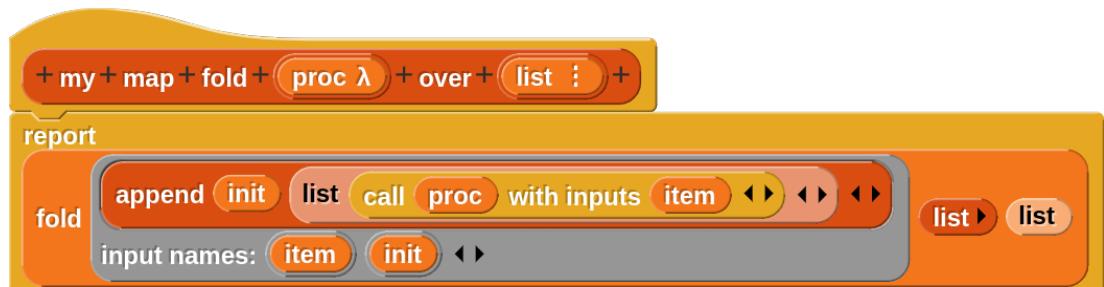
A speech bubble contains the number 12.

map ブロックを作成してみます。引数 proc は Reporter 型です。

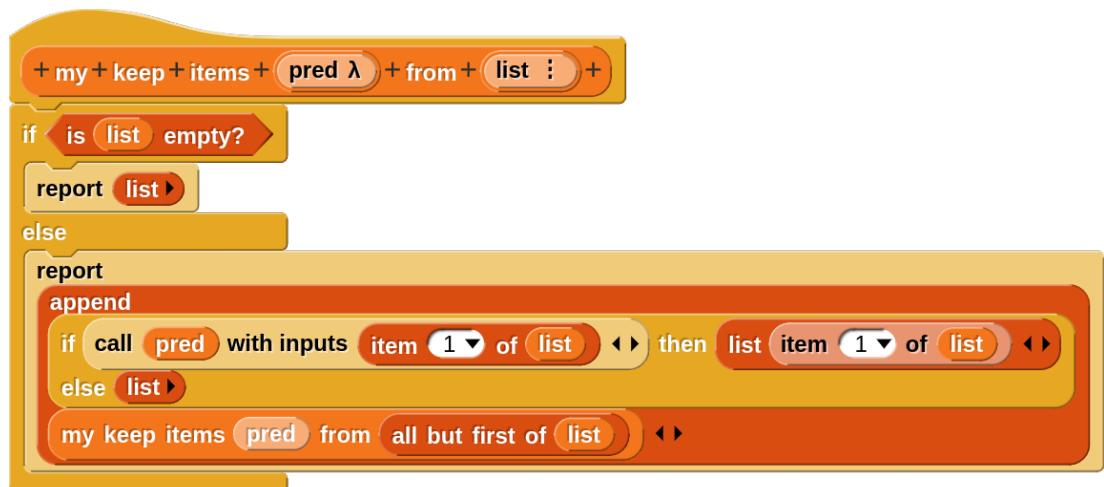
空のリストにリストの要素に対して操作で得た結果を結合するということをリストの最後まで行います。



fold 版です。append への引数の配置を変更しています。



keep ブロックを作成してみます。引数 pred は、Predicate 型です。

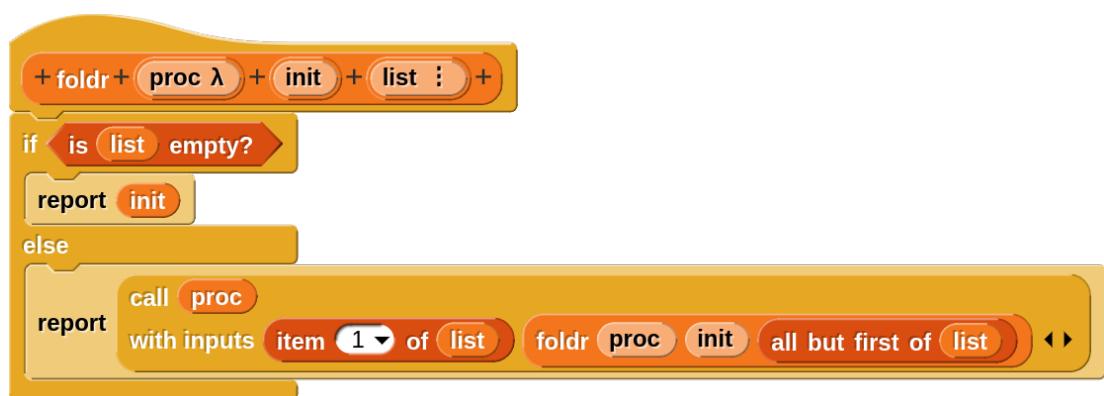
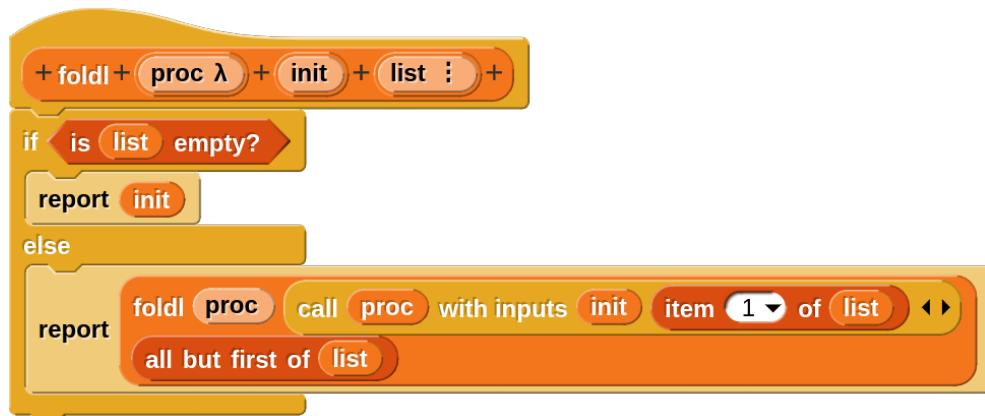


fold 版です。



再帰呼出しを使ってリスト処理をすれば効率の良いプログラムになりますが、なかなかたいへんです。fold を使用してなんでも作成できるわけではありませんが、自動的に再帰呼出しを使った処理してくれます。

因みに Haskell には foldl と foldr があります。



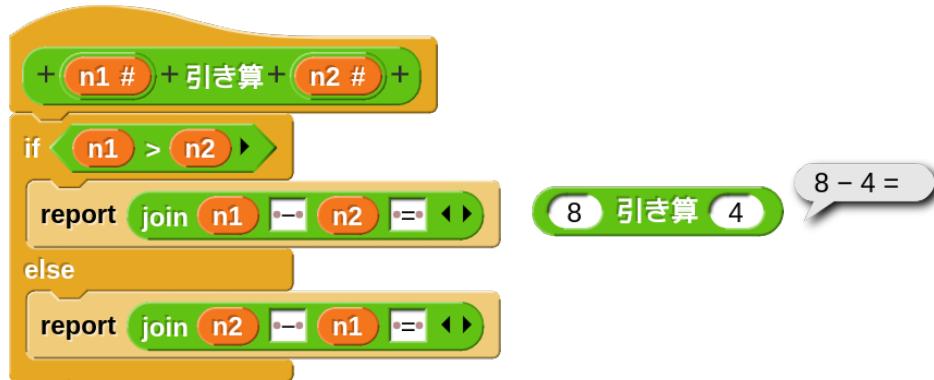
foldl				foldl				foldr				foldr			
init	-	item	値	init	++	item	値	item	-	init	値	item	++	init	値
0	-	1	-1		++	1	1	3	-	0	3	3	++		3
-1	-	2	-3	1	++	2	12	2	-	3	-1	2	++	3	23
-3	-	3	-6	12	++	3	123	1	-	-1	2	1	++	23	123
-6			-6	123			123			2	2			123	123

ブロックがリポートする値ではなくブロック自体を戻り値とする例を示してみます。仕組みを説明するためのものなのであまり意味はありませんが。

まずは基本となる 2 つの引数を取り、足し算の式として表示するものです。



同じように引き算、掛け算、割り算の式として表示するものです。引き算と割り算には引数の扱いに工夫が必要です。

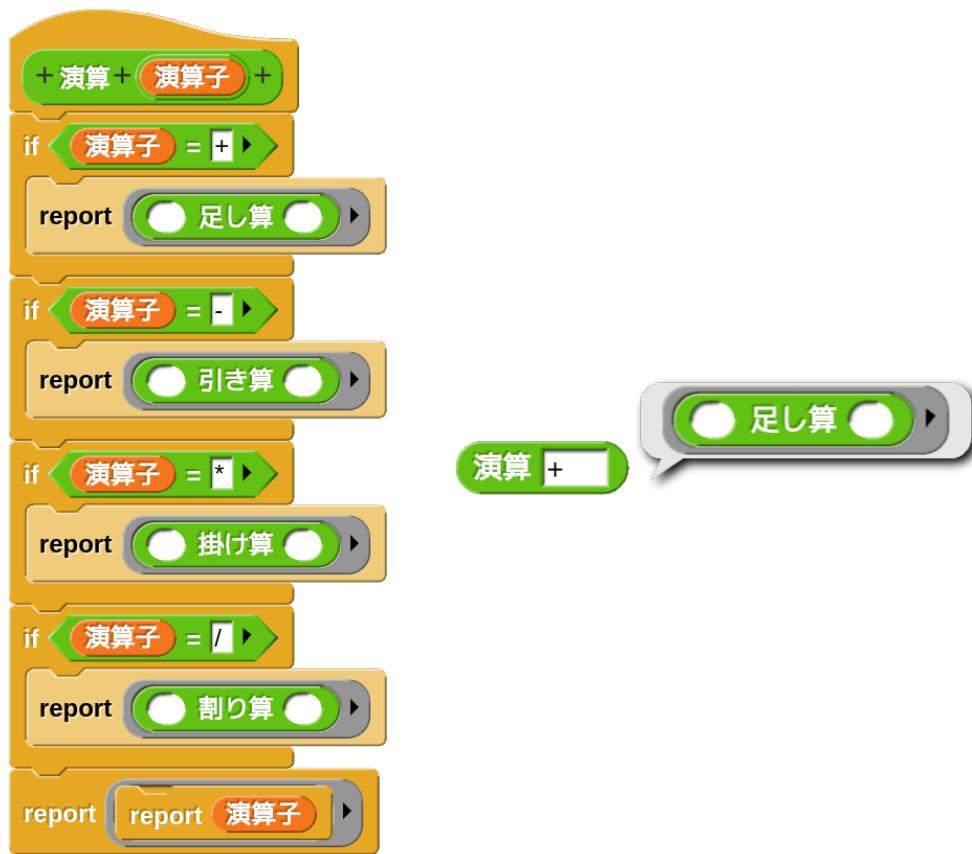


足し算 を ringify してリポートすると、

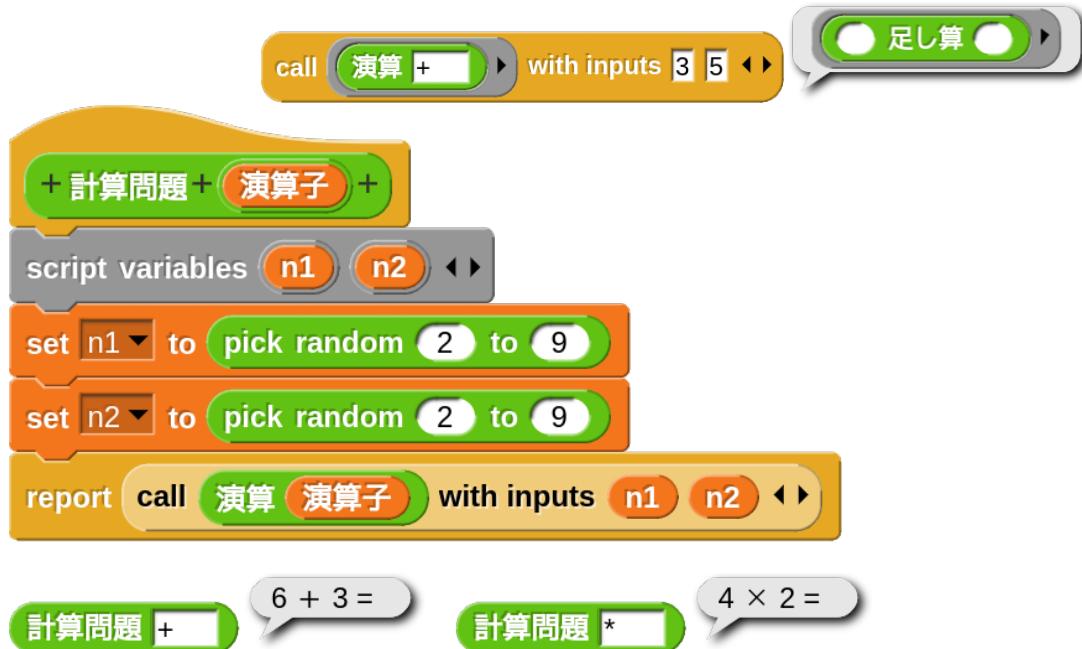


のようにスクリプトブロックをリポートすることができます。

指定された演算子によってこれらのブロック自体を返す定義ブロックです。不明な演算子はそのまま返します。



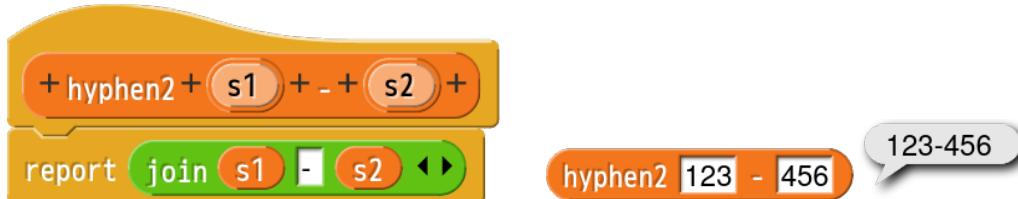
それを呼び出す本体のブロックです。call の入力スロットに **演算 演算子** を入れると ringify **演算 演算子** の状態になります。**演算 演算子** のブロックは ringify されたものを返すものなので、それをまた ringify してしまうと call で実行しても ringify されたものを表示するだけになります。演算のところを右クリックして unringify してください。



### 3.1 カリー化

関数を返す例として「カリー化」ということがよく題材として取り上げられます。複数個の引数に対する処理を一つの引数に対して処理をすることを連ねて行うやり方があります。その「一つの引数に対して処理をする」を関数化することをカリー化と言い、関数を返す関数になります。

Snap! では、一つの引数と外側で指定した値を使って処理するブロックとしてシミュレートできます。以下の定義は入力スロットで指定された二値をハイフンでつなげるものです。この戻り値は文字列です。



これをカリー化してみます。必要な二つの引数をプロトタイプ部分の入力スロットで指定された値 `s1` と `with input` で指定された値 `s2` から得るように変更します。`join` ブロックを `ringify` して、`s2` を受け取るようにしています。上記のスクリプトとの違いは二つ目の引数の受け取り方と、リポートされるものが文字列ではなくスクリプトブロックだということです。



スクリプトブロックを実行するには `call` を使いますが、このままリング付きの `call` で実行するとブロック自体が表示されるだけになってしまいます。`hyphen` がリング付きのスクリプトブロックを返すものなので、それをリング付きの入力スロットに入れたためです。



`call` に入力した `hyphen` ブロックを `unringify` してください。



カリー化したものをユーザー定義ブロックで使用してみます。

例えば、東京都の電話番号に市外局番 03 を付ける定義ブロックです。



### 3.2 OOP オブジェクト指向プログラミング

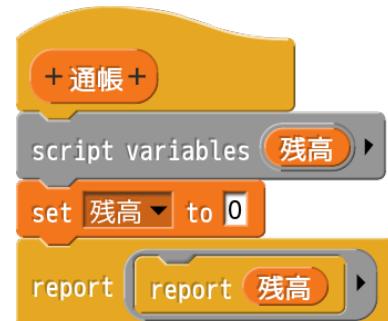
オブジェクト指向プログラミングでは、Class という型を作り、変数や操作（メソッド）を設定します。それを基に作られた変数などの実態にメッセージという形で指令を送って結果を得ます。Snap! には Class のような表立ってオブジェクト指向プログラミングの仕組みはありませんが、高階関数を使うことで同じようなことができます。

預貯金通帳を OOP 的に作ってみます。

預貯金額の変数や通帳で行う操作のスクリプトを通帳自身に持たせます。

通帳を最初に作ると、変数である残高を 0 円にして、残高をリポートする操作を設定するスクリプトです。

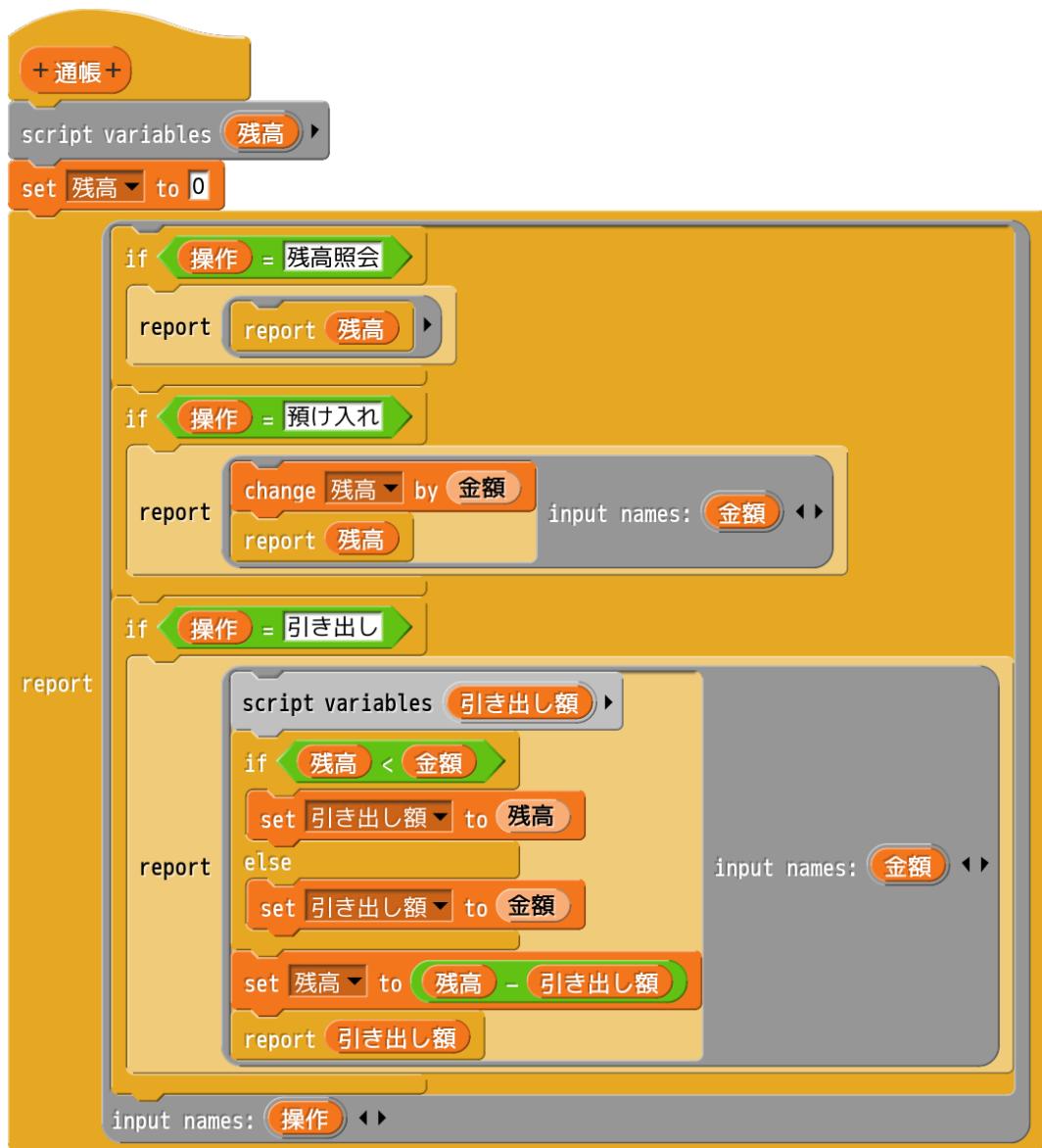
この型を基に以下のようにして通帳を作ります。



「太郎の通帳」を表示させてみます。変数「残高」は「太郎の通帳」が削除されるまで有効です。ローカル変数ですので「花子の通帳」の変数「残高」とは別物です。



通帳への操作として「残高照会」「預け入れ」「引き出し」ができるようにしてみます。「預け入れ」は引数として指定された金額を残高に加算し残高を返します。負数の金額の入力には対応していません。「引き出し」の場合は、残高以内の金額ならばそのまま減算すればいいですが、残高を上回った時は残高を0にします。引き出せた金額を返します。借金は許しません。



定義を変更したので set ブロックの実行が必要です。

set 太郎の通帳 to 通帳

「残高照会」「預け入れ」「引き出し」が「操作」のリクエストであるメッセージになります。メッセージを送るために call が二段構えになっています。

call call 太郎の通帳 with inputs 残高照会 0

call call 太郎の通帳 with inputs 預け入れ 10000 with inputs 10000 10000

call call 太郎の通帳 with inputs 引き出し ▶▶ with inputs 1000 ▶▶ 1000

call call 太郎の通帳 with inputs 残高照会 ▶▶ 9000

使いやすいように ATM ブロックを定義します。「操作」はマウスで選択できるように設定します。「通帳名」は Reporter、「操作」は Text にしてあります。

実は、「通帳名」は「通帳」型の変数を受け取れればいいので Any type でも大丈夫みたいです。

+ ATM + 通帳名 λ + 操作 + 金額 # +  
report call call 通帳名 with inputs 操作 ▶▶ with inputs 金額 ▶▶

ATM 太郎の通帳 残高照会 ▾ 0

ATM 太郎の通帳 預け入れ ▾ 10000

ATM 太郎の通帳 引き出し ▾ 3000

ATM 太郎の通帳 残高照会 ▾ 7000

set 花子の通帳 ▾ to 通帳

ATM 花子の通帳 残高照会 ▾ 0

ATM 花子の通帳 預け入れ ▾ 10000

ATM 花子の通帳 引き出し ▾ 50000

ATM 花子の通帳 残高照会 ▾ 0

通帳をこのようにシミュレートするのは不適切だと思います。 class の仕組みの例として提示してみました。

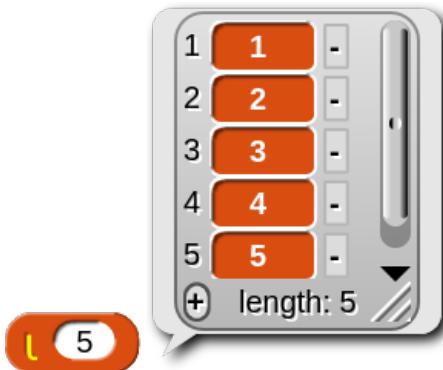
## 4 APL ライブライバー

APL ライブライバーをインポートすると、APL 言語的なブロックが使用できるようになります。基本的なことがらについてだけ説明します。( 使用には JavaScript extensions の設定が必要 )

### 4.1 形

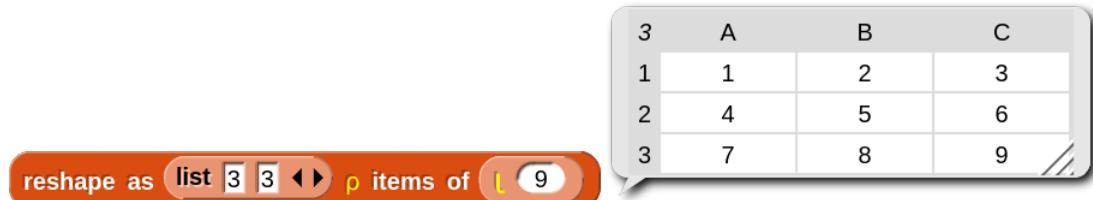
リストではない、ただの一個の数値や文字のデータをスカラーと言います。スカラーを一列に並べたものをベクトルまたはベクターと言います。要素にリストなどを含まないリストです。数学などで扱う、方向の要素を持ったベクトルではありません。( 行列では要素ではなく成分と言うようです。 )

ベクトルを生成するのに、

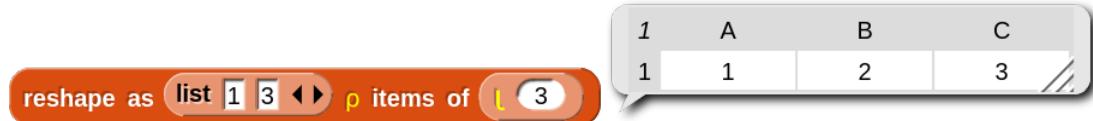


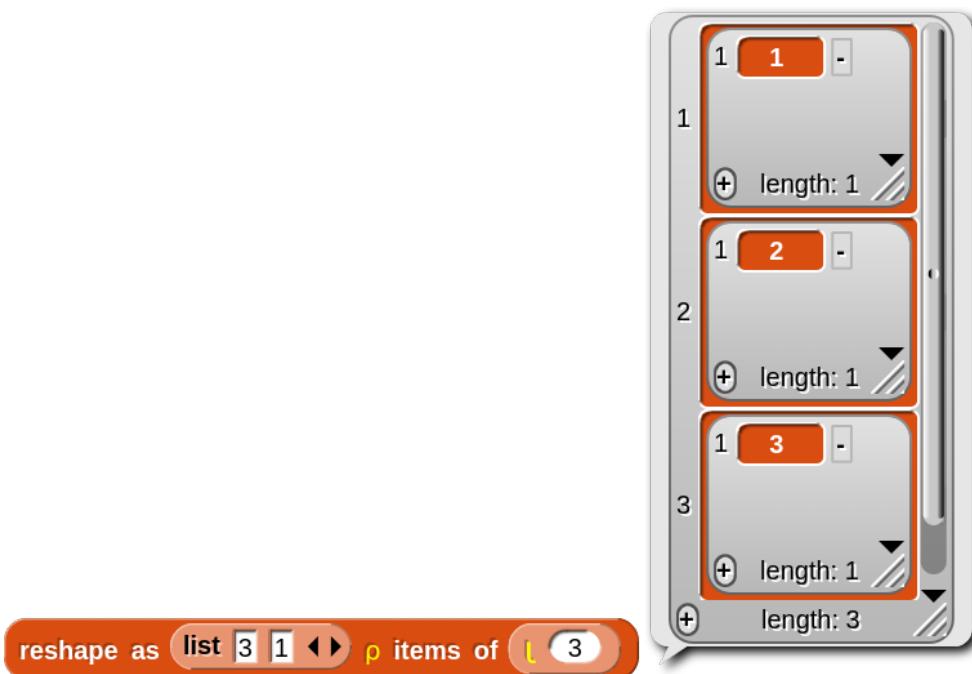
とすれば、1 から指定の数値までのベクトルが得られます。ちなみにこの記号のようなものはギリシャ文字の  $\iota$  イオタです。

ベクトルの形 (shape) を変えて (reshape)、表のように二次元の形にしたものをマトリックス (配列) と言います。reshape ブロックを使って、1~9 のベクトルを 3 行 3 列の配列にすることができます。なお、 $\rho$  はローと読みます。



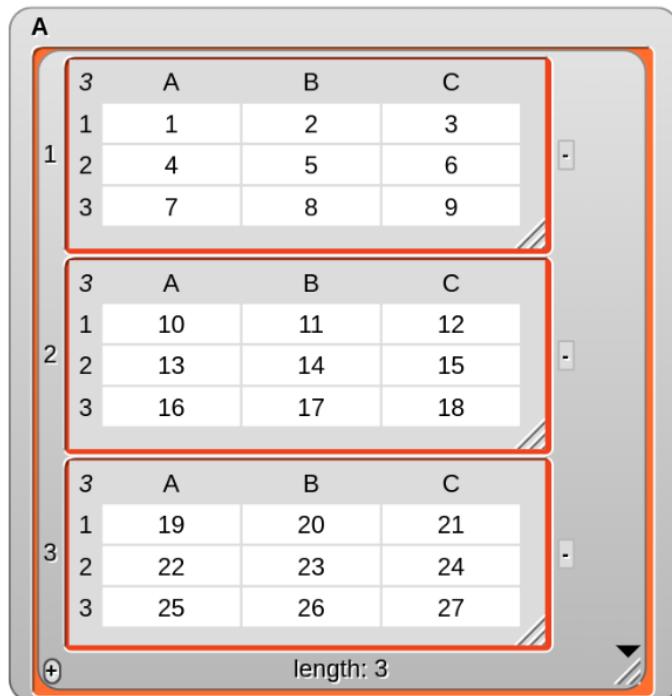
形はリストを使って指定します。次のように 1 行 3 列や、





のように3行1列にすることもできます。

`reshape`のリストを次のようにすると、



3行3列の配列が三次元的に三個連なります。イオタブロックで指定した値よりも配列の要素数が少ない場合は、ベクトルの残りは使用されません。逆に配列の要素数よりも少ない場合は、不足分をベクトルの先頭に戻って供給します。

reshape as list [3 3 ⏪ p items of l 4]

	A	B	C
1	1	2	3
2	4	1	2
3	3	4	1

shape of p 目 は指定されたものの形をリストで返します。

スカラーの形 shape は空です。

shape of p [1]

[+]	length: 0
-----	-----------

shape of p [l 5]

1	5	[+]	length: 1
---	---	-----	-----------

shape of p reshape as list [3 3 ⏪ p items of l 3]

1	3	-
2	3	-
[+]	length: 2	▼

shape of p reshape as list [3 3 3 ⏪ p items of l 3]

1	3	-
2	3	-
3	3	-
[+]	length: 3	▼

形 shape とは別に rank rank of pp 目 というものがあります。これは、一次元、二次元、三次元のような階層を表す数値です。スカラーは 0 になります。

rank of pp [1]

0
---

rank of pp [l 5]

1
---

rank of pp reshape as list [3 3 ⏪ p items of l 3]

2
---

rank of pp reshape as list 3 3 3 ⏪ p items of l 3 3

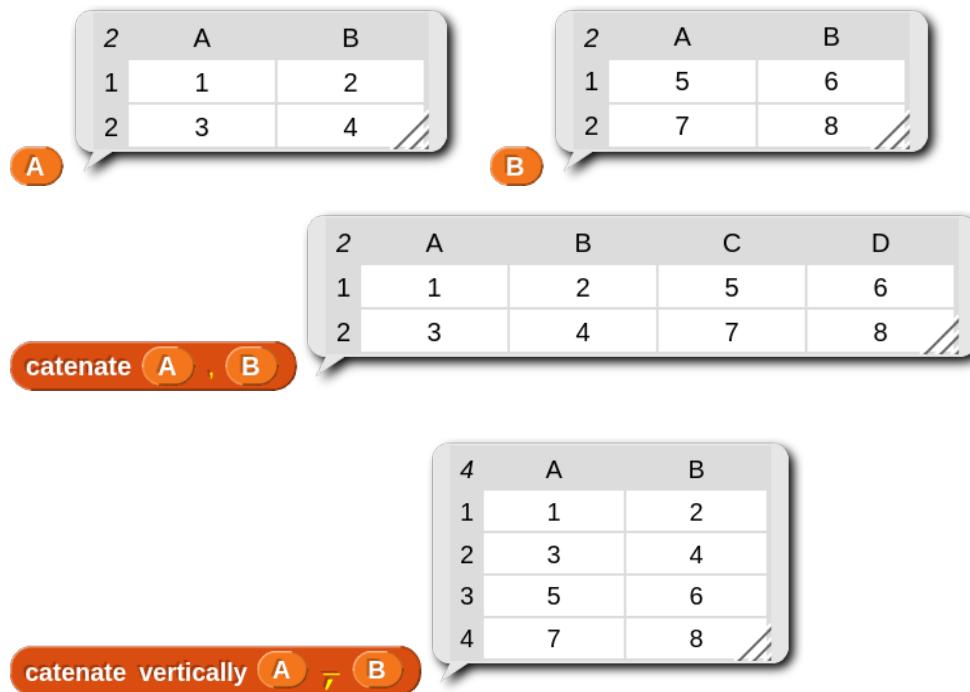
rank を 1 のベクトルにするブロックもあります。

1	1	-
2	2	-
3	3	-
4	4	-
5	5	-
6	6	-
7	7	-
8	8	-
9	9	-
+ length: 9		

flatten (ravel) , reshape as list 3 3 ⏪ p items of l 9

## 4.2 配列の連結

変数 A,B それぞれ 2 行 2 列の配列を作り、横方向、縦方向に連結してみます。



連結される箇所の双方の要素数が同じである必要があります。

## 4.3 配列要素の配置転換

配列内の要素の位置を入れ替える演算子があります。変数 A に 3 行 3 列の配列をセットします。

set A ▾ to reshape as list 3 3 ⏪ p items of l 9

3	A	B	C
1	1	2	3
2	4	5	6
3	7	8	9

A

- 記号が示しているように上下方向、垂直方向に入れ替えます。

3	A	B	C
1	7	8	9
2	4	5	6
3	1	2	3

reverse row order (column contents) ⊖ A

- 記号が示しているように左右方向、水平方向に入れ替えます。

3	A	B	C
1	3	2	1
2	6	5	4
3	9	8	7

reverse column order (row contents) ⚡ A

- 記号が示しているように対角線を軸にして入れ替えます。( 転置 )

3	A	B	C
1	1	4	7
2	2	5	8
3	3	6	9

transpose ◎ A

#### 4.4 ベクトル、配列の範囲指定、選択

take は、ベクトルの先頭から指定した個数の要素のリストをリポートします。負の数で指定すると、ベクトルの最後尾から指定された絶対値の個数の要素のリストをリポートします。

1	1	-	▼
2	2	-	
3	3	-	
⊕	length: 3		

take 3 ↑ from l 5

1	3	-	▼
2	4	-	
3	5	-	
⊕	length: 3		

take -3 ↑ from l 5

`drop` は、ベクトルの先頭から指定した個数の要素を除外し、残りの要素のリストをリポートします。負の数で指定すると、ベクトルの最後尾から指定された絶対値の個数の要素を除外し、残りの要素のリストをリポートします。

drop 3 from [5 v]

drop -3 from [5 v]

配列が指定された場合は、行単位の指定になるようです。

take 2 from [A v]

2	A	B	C
1	1	2	3
2	4	5	6

0(非選択)、1(選択)で指定リストにして要素を選択することができます。指定リストの要素数は選択される側の要素数と一致させる必要があります。

list [1, 2, 3, 4, 5]

select rows (compress columns)

list [1, 0, 0, 1, 0]

map (mod 2 over [5 v])

これを使用すると奇数番の要素を選択できます。

select rows (compress columns)

map (mod 2 over [5 v]) / [5 v]

5行1列なので `select rows` を使用しました。1行5列に対しては `select columns` を使用します。それを偶数番でやってみます。

```

select columns (compress rows) map mod 2 = 0 over l 5
reshape as list 1 5 p items of l 5

```

配列に対して使用する場合は、選択したい行または列の要素数に合わせて指定リストを作成する必要があります。

```

A
select rows (compress columns) list 1 0 1 / A

```

## 4.5 配列要素に対する演算

APL では、配列に対してスカラーの演算をすることができます。

```

reshape as list 3 3 p items of l 3 + 10

```

これは、スカラーの値 10 を `reshape as list 3 3 p items of 10` のように同じ形の配列に変換(整合)して、対応する配列要素同士で演算するようになっています。

`combine ⌈ using` の配列版が用意されています。

3	A	B	C
1	1	2	3
2	4	5	6
3	7	8	9

A

combine in rows (reduce by column vectors)  / 目

配列の 1 行内の各列の要素に対して指定された演算をします。

1	6	.
2	15	.
3	24	.
+	length: 3	▼

combine in rows (reduce by column vectors)  / A

$$(1 + 2 + 3 \quad 4 + 5 + 6 \quad 7 + 8 + 9)$$

この場合の演算子は + なので、1 行内の各列要素の合計になります。

reduce(減らす)... 配列マトリックスだったものがベクトルにランクが減るということです。

combine in columns (reduce by row vectors)  ✖ 目

配列の 1 列内の各行の要素に対して指定された演算をします。

1	A	B	C
1	12	15	18

combine in columns (reduce by row vectors)  ✖ A

$$(1 + 4 + 7 \quad 2 + 5 + 8 \quad 3 + 6 + 9)$$

## 4.6 outer product

次のようにすると、九九の表の一部ができます。( **list [1 2 3 ←→ ]** を **[l 9]** に変更すれば全体表示 )

3	A	B	C
1	1	2	3
2	2	4	6
3	3	6	9

**outer product** **list [1 2 3 ←→ ] o.** **○ × ○ →** **list [1 2 3 ←→ ]**

計算方法を表示させてみます。

3	A	B	C
1	$[a1]x[b1]$	$[a1]x[b2]$	$[a1]x[b3]$
2	$[a2]x[b1]$	$[a2]x[b2]$	$[a2]x[b3]$
3	$[a3]x[b1]$	$[a3]x[b2]$	$[a3]x[b3]$

**outer product** **list [a1 a2 a3 ←→ ] o.** **join [ [ ] x [ ] ←→ ]** **list [b1 b2 b3 ←→ ]**

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \cdot \times \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 \times 1 & 1 \times 2 & 1 \times 3 \\ 2 \times 1 & 2 \times 2 & 2 \times 3 \\ 3 \times 1 & 3 \times 2 & 3 \times 3 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{bmatrix}$$

単位行列なども作成できます。

4	A	B	C	D
1	true	false	false	false
2	false	true	false	false
3	false	false	true	false
4	false	false	false	true

**outer product** **[l 4 o.** **← = →** **[l 4 ]**

0, 1 で表すには、

4	A	B	C	D
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1

**outer product** **[l 4 o.** **← = →** **[l 4 ] x [1 ] →**

1	2	-
2	3	-
3	5	-
4	7	-
5	11	-
6	13	-
7	17	-
8	19	-
9	23	-
10	29	-
(+)	length: 10	▼

2 から 30 までの整数の素数を求めてみます。  
 「2 から」ではなく「1 から」とした場合は、  
 map の演算子を  $2 = \square$  とする必要があります。

```

set [N] to [30]
set [N] to [drop (1) from [1..N]]
repeat
    select rows (compress columns)
        map [1 = mod 1 over]
    end
end
report [combine in rows (reduce by column vectors) (+) /] / [N]
outer product [N o.]
    [0 = mod N]
end

```

2 からその数までの整数で順に mod を使って余りが 0 のものを調べます。combine で集計して余りが 0 だった個数が 1、つまり、その数だけでしか割り切れないものが素数であるということです。map  $1 = \square$  で素数の位置をポイントし、select でそのポイントから数値をピックアップします。

一番内側の outer product ブロックから、結果を表示させながら順に一層ずつブロックを構築していくと、スクリプトの仕組みが理解できるかもしれません。分かりやすいように 1 から 5 の範囲で内側のブロックから順にやってみます。まずは N に 1 から 5 のリストをセットします。

```

set [N] to [1..5]
outer product [N o.]
    [0 = mod N]
end

```

	A	B	C	D	E
1	true	false	false	false	false
2	true	true	false	false	false
3	true	false	true	false	false
4	true	true	false	true	false
5	true	false	false	false	true

A の列の意味は 1 から 5 の値に対する 1 で割った余りが 0 かどうかを表しています。これはすべて割り切れるので全部 true です。B の列の意味は 1 から 5 の値に対する 2 で割った余りが 0 かどうかを表しています。この場合、2 と 4 の箇所で割った余りが 0、つまり true になります。他の C, D, E の列では行と列が同じ値の時しか割った余りが 0、つまり true なりません。

次の処理で N の値に対する各行の true の合計を求めます。

The screenshot shows APL code and its execution results. The code consists of three nested blocks:

- outer product**: Takes a scalar  $N$  and an array  $0 = \dots \mod N$ . The result is a vector of length  $N$  where each element is  $\text{length} \ 5$ .
- combine in rows (reduce by column vectors)**: Reduces the previous result by summing across columns.
- map**: Maps the condition  $2 = \square$  over the reduced result.

The right side shows the resulting data for  $N=5$ :

1	1
2	2
3	2
4	3
5	2
+	length: 5

素数は 1 と自分自身でしか割り切れないものなので、true の合計が 2 のものを調べます。

The screenshot shows APL code and its execution results. The code consists of three nested blocks:

- outer product**: Takes a scalar  $N$  and an array  $0 = \dots \mod N$ . The result is a vector of length  $N$  where each element is  $\text{length} \ 5$ .
- combine in rows (reduce by column vectors)**: Reduces the previous result by summing across columns.
- map**: Maps the condition  $2 = \square$  over the reduced result.

The right side shows the resulting data for  $N=5$ :

1	false
2	true
3	true
4	false
5	true
+	length: 5

2, 3, 5 の位置が true になりました。その位置に対応する  $N$  の値のリストを求めます。

The screenshot shows APL code and its execution results. The code consists of three nested blocks:

- outer product**: Takes a scalar  $N$  and an array  $0 = \dots \mod N$ . The result is a vector of length  $N$  where each element is  $\text{length} \ 5$ .
- combine in rows (reduce by column vectors)**: Reduces the previous result by summing across columns.
- select rows (compress columns)**: Selects rows based on the truth values from the previous step.

The right side shows the resulting data for  $N=5$ :

1	2
2	3
3	5
+	length: 3

### [ 参考文献 ]

『基礎からの APL 解説と例題例解』

西川利男/日本アイビー・エム 共著 サイエンスハウス 刊

## 4.7 inner product

inner product の機能を表示してみます。「?」「??」のところにはそれぞれ左側、右側の演算子が入ります。

The screenshot shows APL code for setting up two 2x2 matrices:

- set A to reshape as list 2 2**: Reshapes a list of 4 items [a b c d] into a 2x2 matrix.
- set B to reshape as list 2 2**: Reshapes a list of 4 items [e f g h] into a 2x2 matrix.

2	A	B
1	a??e?b??g	a??f?b??h
2	c??e?d??g	c??f?d??h

The screenshot shows APL code for calculating the inner product of matrices A and B:

- inner product A**: Calculates the inner product of matrix A.
- join**: Joins the result of the inner product of A with the inner product of B.
- join**: Joins the result of the previous join with the inner product of B.

	A	B
1	$axe+bxg$	$axf+bxh$
2	$cxe+dxg$	$cxh+dxh$

## inner product A

join  +  <  >

join    B

配列要素同士の掛け算とは別に、線形代数では行列と行列の積というものが定義されています。

## inner product

A green calculator icon with a white display screen showing a plus sign (+) and a multiplication sign (x). The calculator has a grey frame and a black numeric keypad below it.

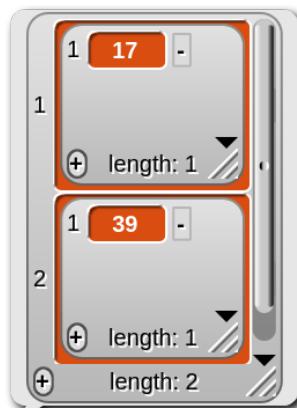
プロセスで求めることによって本末

3行3列配列同士では次のような計算方法になります

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} a \times e + b \times g & a \times f + b \times h \\ c \times e + d \times g & c \times f + d \times h \end{bmatrix}$$

左側の配列の列数と右側の配列の行数は同じ必要があります。

$$\begin{aligned}
 &= \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 \\ 6 \end{bmatrix} \\
 &= \begin{bmatrix} 1 \times 5 + 2 \times 6 \\ 3 \times 5 + 4 \times 6 \end{bmatrix} \\
 &= \begin{bmatrix} 17 \\ 39 \end{bmatrix}
 \end{aligned}$$



inner product

list list 1 2 ◀ ▶

list 3 4 < > << >>

10

list 5 list 6

$$\begin{aligned}
 & \left[ \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \right] \begin{bmatrix} 5 & 6 \\ 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} \\
 &= \begin{bmatrix} 1 \times 5 + 2 \times 7 + 3 \times 9 + 4 \times 11 & 1 \times 6 + 2 \times 8 + 3 \times 10 + 4 \times 12 \\ 90 & 100 \end{bmatrix}
 \end{aligned}$$

The Scratch script shows an inner product operation. At the top, there is a data block with values A=90 and B=100. Below it, the script consists of two main parts: 
 1. An orange **inner product** block with two **list** inputs. The first list contains elements 1, 2, 3, 4, followed by two green **+ x** blocks. The second list contains elements 5, 6, followed by two green **+ x** blocks.
 2. Another orange **inner product** block with two **list** inputs. The first list contains elements 7, 8, followed by two green **+ x** blocks. The second list contains elements 9, 10, followed by two green **+ x** blocks.

行列の積の利用例として座標変換があります。座標  $(x, y)$  の点を原点  $(0, 0)$  を中心にして反時計回りに  $90^\circ$  回転させた座標を求めるものです。

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

web で「回転行列」を検索すると大学系などのサイトで数学的な解説が見られます。

一番わかりやすい例として、 $(1, 0)$  の点を  $90^\circ$  反時計回りに回転させると  $(0, 1)$  の点になります。  
左側の配列、回転させるため係数を作成するブロックです。

The Scratch script creates a rotation matrix for  $90^\circ$  counter-clockwise. It starts with a **set [θ] to [90]** control block. Then, it uses a **set [R] to [list]** control block with two parallel **list** blocks. The top list block contains **cos [θ]**, **neg [sin [θ]]**, and **sin [θ]**. The bottom list block contains **sin [θ]** and **cos [θ]**.

それを表示すると、次のような値の要素の配列になります。 $6.123233995736766e-17$  は 0 とみなします。

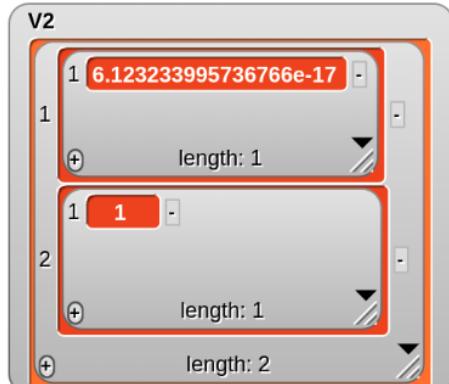
The data block **R** contains a 2x2 matrix with the following values:  
 Row 1: [0, -1]  
 Row 2: [1, 0]  
 To the right of the matrix, there is a mathematical representation:  
 $\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$

この配列  $\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$  と座標  $(x, y)$   $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$  の積  
 $\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 \times 1 + (-1) \times 0 \\ 1 \times 1 + 0 \times 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 \\ 1 \end{bmatrix}$  を求めます。

```

set V0 to list 1
list 0
set V2 to inner product R + × V0

```



結果として、座標  $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$  を 90 度回転させた座標  $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$  が求められました。

注意点として、 $(x, y)$  の座標の表し方が 2 行 1 列の形で指定する必要があることです。

三角形を回転してみます。前準備として指定した座標  $\begin{bmatrix} x \\ y \end{bmatrix}$  のリストの点を一筆書きするユーザー定義ブロックを作成します。

```

+ draw + xyList + xyList ;
pen up
go to x: item 1 of item 1 of item 1 of xyList y:
item 1 of item 2 of item 1 of xyList
pen down
for each item in xyList
  go to x: item 1 of item 1 of item y:
  item 1 of item 2 of item
pen up

```

```

hide
clear
set V0 to list list 0 0 list 20 100 list 40 0 list 0 0
set V0 to map reshape as list 2 1 p items of over V0
set pen color to blue
draw xyList V0

```



これを実行すると、 が表示されます。

```

set theta to 90
set R to list list cos of theta neg of sin of theta
list sin of theta cos of theta
set V2 to
map inner product R + . . x xyList over V0
input names: xyList
set pen color to magenta
draw xyList V2

```



これを実行すると、 回転後の図形が追加表示されます。

座標  $(x, y)$  は 2 行 1 列  $\begin{bmatrix} x \\ y \end{bmatrix}$  で指定する必要があるので、1 行 2 列で指定した各 xy 座標を次のようにして 2 行 1 列の座標に変換していました。

```

set V0 to list list 0 0 list 10 40 list 20 0 list 0 0
set V0 to map reshape as list 2 1 p items of over V0

```

行列の積の定義により、左側の配列による変換が右側のそれぞれの列に対して行われます。

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} a \times e + b \times g & a \times f + b \times h \\ c \times e + d \times g & c \times f + d \times h \end{bmatrix}$$

つまり、右側の配列は列を増やしていくとそれぞれの列に対して同じように変換されるということです。先の例では三角形の座標 (x, y) のリストから各点の座標を取り出して個別に変換していましたが、

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \end{bmatrix}$$

で、一括して変換することができます。ただし、このような形の配列で指定するには、x 座標, y 座標を 2 行に分けて指定する必要があります。

2	A	B	C	D
1	x1	x2	x3	x4
2	y1	y2	y3	y4

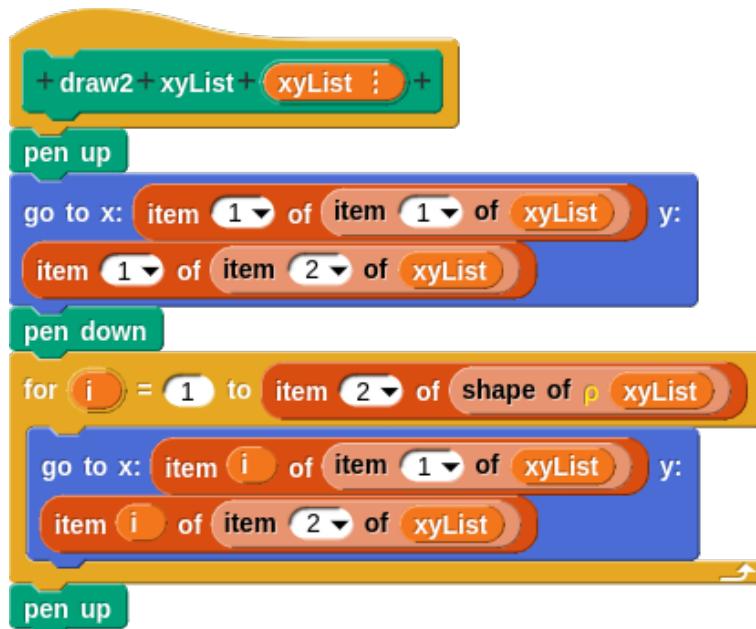
  


transpose を使用すると、前と同じ座標リスト指定から変換することができます。

2	A	B	C	D
1	x1	x2	x3	x4
2	y1	y2	y3	y4


また、このような形の配列で指定された座標の図形を描画するには draw xyList も変更しなればなりません。



変換前の図形描画スクリプトです。

```

hide
clear
set V0 to list [list [0 0] [list [20 100] [list [40 0] [list [0 0]]]]]
set V0 to transpose [V0]
set pen color to blue
draw2 xyList [V0]

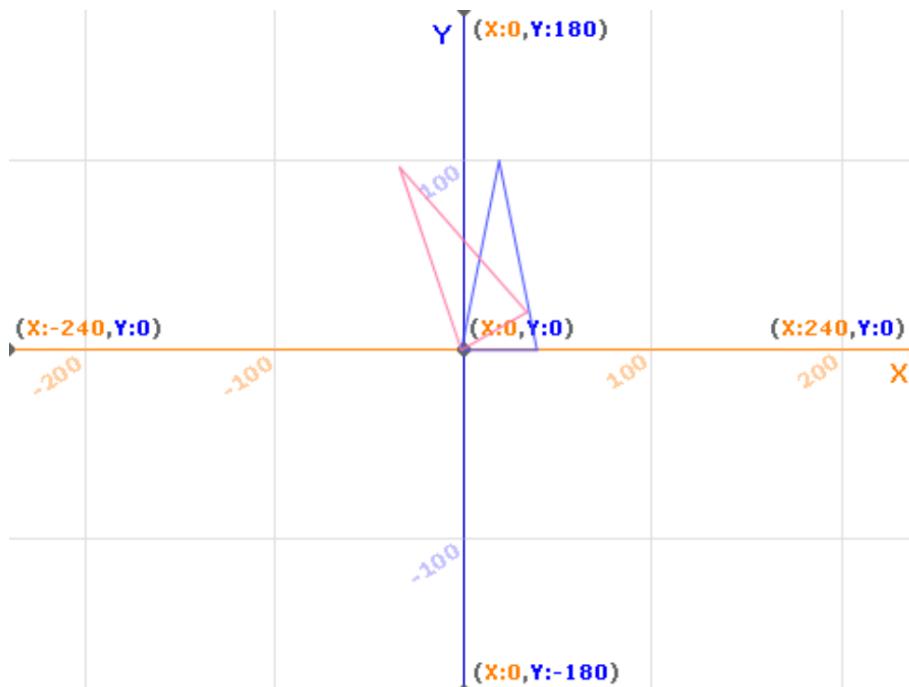
```

30 度反時計回りに回転するよう変換して図形描画するスクリプトです。

```

set θ to 30
set R to list [list [cos of θ [neg of sin of θ]] [list [sin of θ [cos of θ]]]]
set V2 to inner product [R] [ + [x [V0]]] [x [V0]]
set pen color to magenta
draw2 xyList [V2]

```



### [ 参考文献 ]

『Python ではじめる数学の冒険

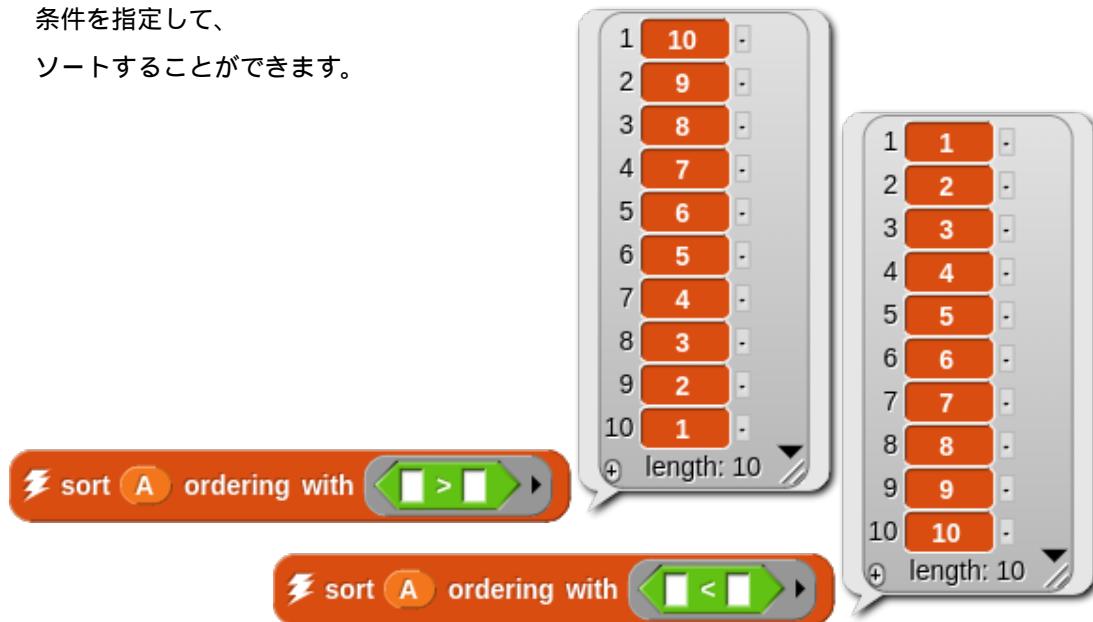
プログラミングで図解する代数、幾何学、三角関数』

Peter Farrell 著 鈴木幸敏 訳 オライリー・ジャパン 刊

## 4.8 ソート、順位付け

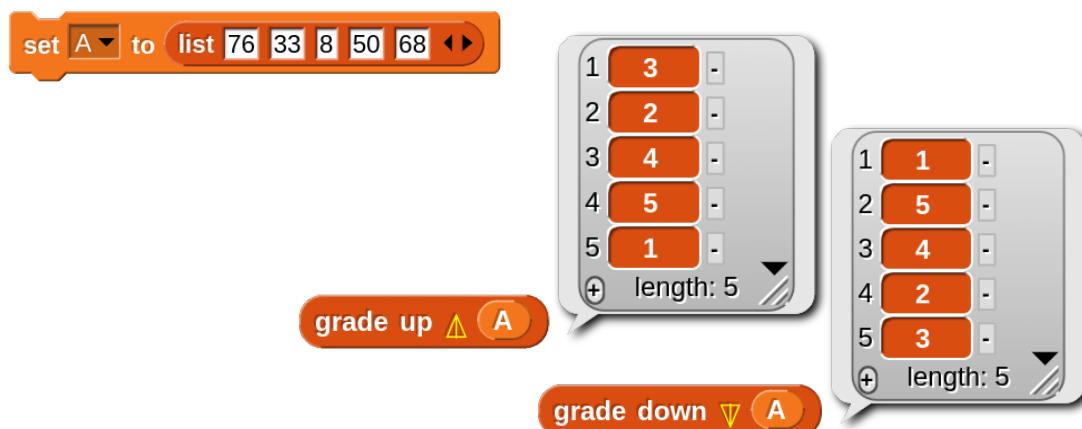
set A ▾ to list 1 8 2 6 3 5 4 7 10 9 ⏪とした場合、

条件を指定して、  
ソートすることができます。



これに対して、指定されたリストの昇順または降順の位置をリストにするブロックがあります。

grade up ⬆ 目 grade down ⬇ 目 記号が示しているように、昇順降順です。



grade up(昇順) の場合は 1 番小さいのは A リストの 3 番目の 8、2 番めは A リストの 2 番目の 33 なので、リストは (3, 2, 4, ..) となります。grade down(降順) の場合は 1 番大きいのは A リストの 1 番目の 76、2 番めは A リストの 5 番目の 68 なので、リストは (1, 5, 4, ..) となります。

これを使ってソートされたリストを表示するのであれば次のようにする必要があります。

```
for each item in grade up ⬆ A
  say item item of A for 2 secs
```

これを使うと、表計算ソフトの RANK 関数のような働きをさせることができます。

grade up ▲ grade up ▲ A

grade up ▲ grade down ▼ A

で、それぞれ A リストの要素が全体の何番目に小さいまたは大きいかの順位付けされたリストが求められます。値のリストの下に全体の順位付けのリストを並べて表示してみます。

2	A	B	C	D	E
1	76	33	8	50	68
2	5	2	1	3	4

reshape as list 2 5 ↵ ⚡ items of catenate A , grade up ▲ grade up ▲ A

2	A	B	C	D	E
1	76	33	8	50	68
2	1	4	5	3	2

reshape as list 2 5 ↵ ⚡ items of  
catenate A , grade up ▲ grade down ▼ A

## 索引

- all but first of, 20–23, 31
- APL, 47
- break, 6
- catch, 7
- continuation, 3
- continue, 6
- factorial, 17
- fold, 37
- iteration-composition, 7
- keep, 25
- map, 39
- rank, 49
- reshape, 47
- reverse, 24
- thread, 13
- throw, 7
- w/continuation, 3
- 階乗, 17
- 回転行列, 59
- 形, 47
- カリー化, 43
- 行列の積, 58
- クイックソート, 25
- 継続, 3
- 高階関数, 34
- 再帰, 17
- 座標変換, 59
- スカラー, 47
- スレッド, 13
- 素数, 56
- 単位行列, 55
- 転置, 51
- 配置転換, 50
- 配列, 47, 50, 51, 53
- ハノイの塔, 17
- フィボナッチ数列, 28
- 並列処理, 8
- ベクター, 47
- ベクトル, 47
- 末尾再帰, 31
- リスト要素の巡回, 22, 34
- 連結, 50