

# Snap! のこと II

齋藤文康

2025 年 12 月 27 日

容量の関係で、「Snap! のこと」で扱っていた「再帰呼び出し」以降の項目を別文書にしました。

私の理解不足で間違っているところがある可能性もあります。正しい内容になるように努めました。スクリプトを含め無保証です。

# 目 次

<b>1</b>	<b>再帰呼び出し</b>	<b>3</b>
1.1	再帰呼出しの例 . . . . .	3
1.1.1	階乗 . . . . .	3
1.1.2	ハノイの塔 . . . . .	3
1.2	再帰呼び出しの使用 . . . . .	4
1.2.1	繰り返し . . . . .	4
1.2.2	my length . . . . .	5
1.2.3	リストをリポート . . . . .	7
1.2.4	take と drop . . . . .	8
1.2.5	my contains . . . . .	9
1.2.6	my uniques . . . . .	9
1.2.7	my index of ( ) in ( ) . . . . .	10
1.2.8	リスト要素の巡回 . . . . .	11
1.2.9	指定の要素に対する delete と replace . . . . .	13
1.3	= と identical . . . . .	15
1.3.1	クイックソート ( 整列 / 並べ替え ) . . . . .	17
1.3.2	フィボナッチ数列 . . . . .	20
1.3.3	末尾再帰 . . . . .	22
<b>2</b>	<b>高階関数</b>	<b>24</b>
2.1	高階関数型ブロックの基本 . . . . .	24
2.2	高階関数型の take と drop . . . . .	26
2.3	操作を指定するリストの巡回 . . . . .	27
2.4	foldl, foldr . . . . .	29
2.5	unfold . . . . .	34
2.6	カーリー化 . . . . .	36

# 1 再帰呼び出し

再帰、再帰呼び出し (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再帰呼び出しが繰り返し処理の手法になっています。

## 1.1 再帰呼び出しの例

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。

### 1.1.1 階乗

factorial 階乗は再帰呼び出しの例としてよく使用されます。

The Scratch script for the factorial function is as follows:

```
+ factorial + n # +
if n < 0
  report -1
else if n = 0
  report 1
else if
  report n × factorial n - 1
```

The mathematical definition of factorial is shown as:

$$\begin{cases} 0! = 1 \\ n! = n \times (n-1)! \end{cases}$$

A call to the function is shown: `factorial 7` returns the value 5040.

### 1.1.2 ハノイの塔

ハノイの塔のパズルも再帰呼び出しの例としてよく使用されます。

The Scratch script for the Tower of Hanoi puzzle is as follows:

```
+ hanoi + n # + a + b + c +
if n = 0
  report list
else
  report append
    hanoi n - 1 a c b
    join move-# n from a to b in front of
    hanoi n - 1 c b a
```

A list of moves for 3 disks is shown:

- move #1 from a to b
- move #2 from a to c
- move #1 from b to c
- move #3 from a to b
- move #1 from c to a
- move #2 from c to b
- move #1 from a to b

The list has a length of 7.

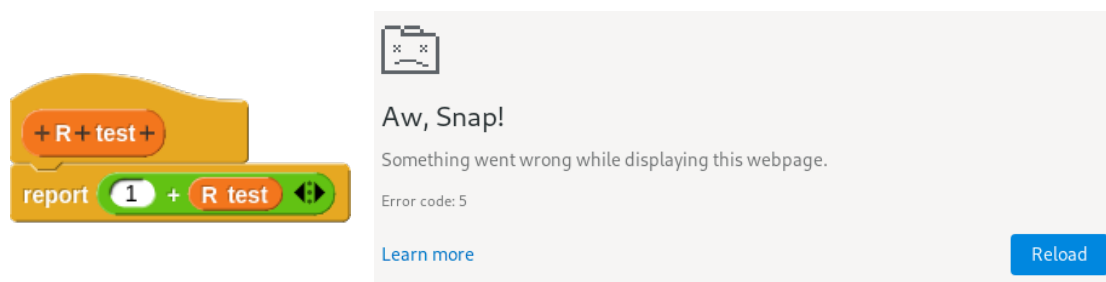
## 1.2 再帰呼び出しの使用

スクリプトの表示面積を小さくするために、3つのブロックを定義して使用します。

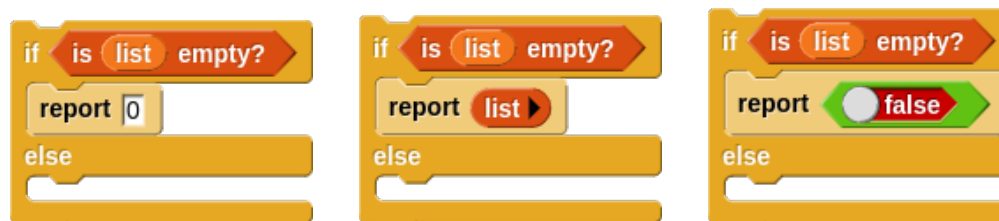


### 1.2.1 繰り返し

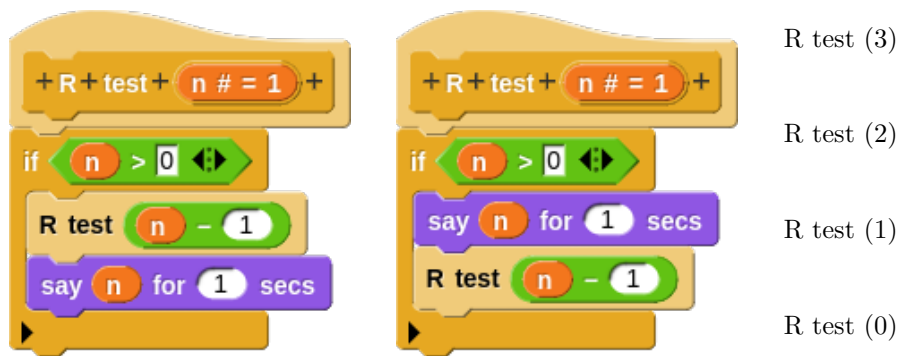
これはただ自分自身を呼び出すものです。(エラーになるので実行しないでください)



定義ブロックはどこかのスクリプトから呼び出されて実行されるわけですが、実行終了後に呼び出し元に帰る必要があります。呼び出し元のスクリプトの実行を継続するための情報を保存しておき、それを取り出してそこに復帰します。上記のスクリプトでは自分自身への無限呼び出しになり、情報を保存するための場所がなくなってしまいます。ただの無限ループというだけの問題ではありません。したがって、再帰呼び出しでは再帰呼び出しを終了するためのスクリプトが必要です。指定の回数繰り返す処理ならば、回数が0が終了条件です。リスト操作では、空リストが終了条件になります。リスト操作で Reporter 型のブロックを作成する場合、数値型では0を、リスト型では空リストを、Predicate 型のブロックを作成する場合は false を原則としてリポートして再帰を終了させます。空リストの場合は、**list** でも同じことです。



次は指定した回数だけ何かを行う定義ブロックです。 $n = 0$  が終了条件になります。 $n$  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きます。この定義ブロックは0以下だと何もしないで呼び出し元にに戻ります。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。再帰呼び出しブロックの位置により  $n$  の値はカウントアップにもカウントダウンにもなります。



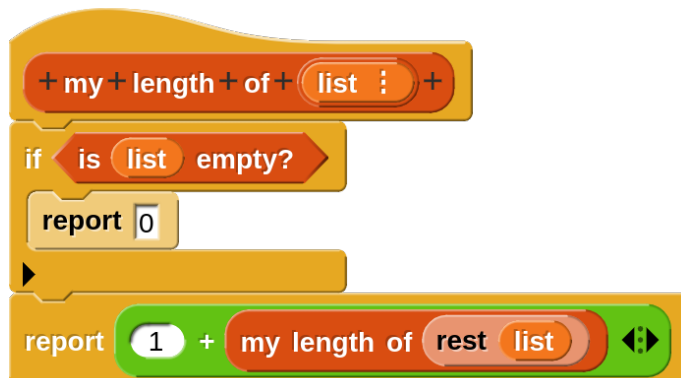
### 1.2.2 my length

リストの要素数を求める length ブロックを repeat until ブロックを使って作ってみます。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。

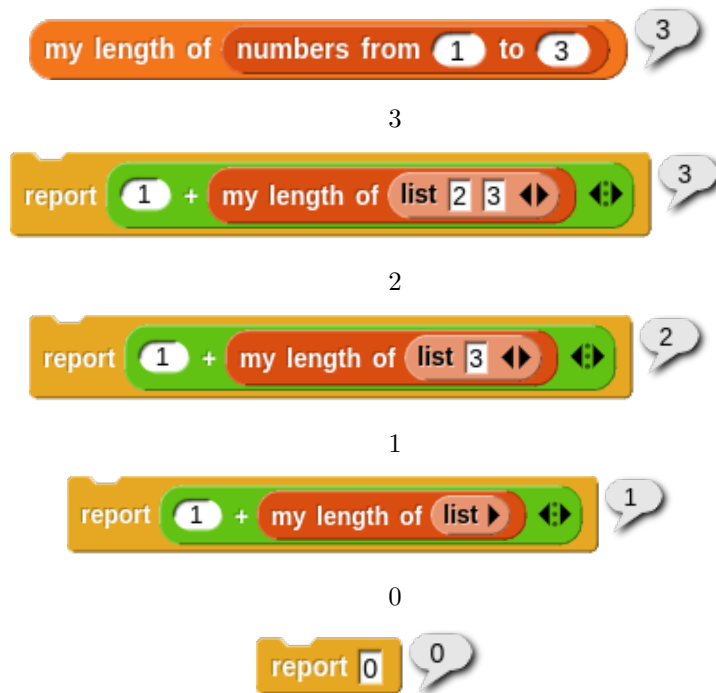


再帰版です。カウント用の変数が無いので理解しにくいですが、report が返す値がカウント用変数の役割を果たしています。my length of rest list でリストが空になるまで再帰呼び出しされて、0, 1, 2, ... と、report が返す値 +1 を積み重ねて、結果的に 0 からのカウントアップで要素数を求めることができます。自分自身を呼び出すことを除けば repeat until 版と同じやり方になります。

〔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。〕ということリストが空になるか条件が成立するまで、処理を行うのが再帰処理の基本です。この場合の処理は、レポートされた値（要素数）に 1 を加えているだけです。



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。



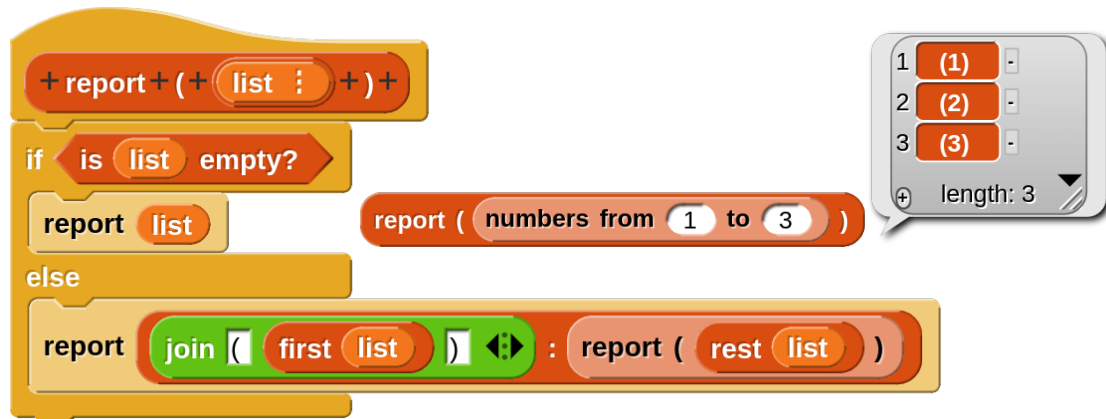
デバッグモードで 1 ステップ実行してみると、リストが空になると次のように、my length of () が 0 を返り値としてこの場所に戻ってきます。それぞれの呼び出しで 1 を加えているのでカウントアップされていきます。



$1 + (1 + (1 + 0)) = 3$  のようにして、返ってきた値に対して加算が行われて終了します。

### 1.2.3 リストをリポート

リストの先頭の要素に対して操作し、その残りのリストに対して再帰呼び出しをした結果を加えることをリストが空になるまで行います。map を行ったような結果になります。

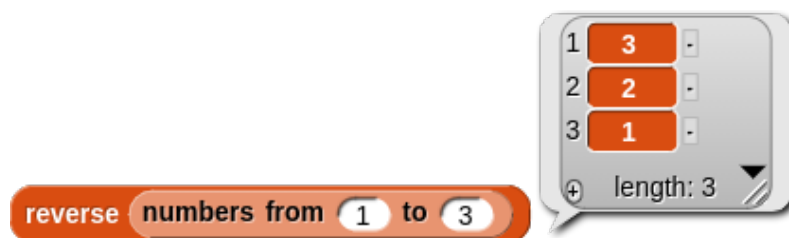


リストの先頭の要素から操作したものを順に加えているのでこのようになりました。

残りのリスト操作の結果に先頭の要素を加えるようにすると、逆順リストが得られます。



append はリスト同士を一つにするものなので、先頭の要素を `list` の中に入れてリストにしています。( ver.11 では要素だけでもよくなりました。 )

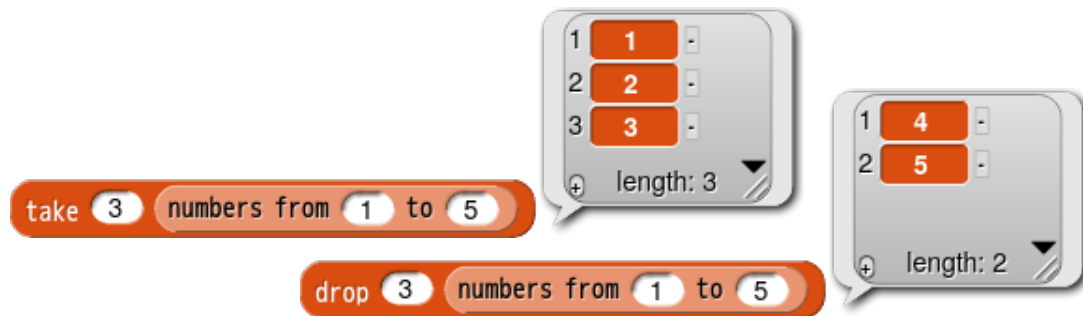


reverse (1 2 3)	{ L2 } + (1)	L3 = (3 2 1)
reverse (2 3)	{ L1 } + (2)	L2 = (3 2)
reverse (3)	{ L0 } + (3)	L1 = (3)
reverse ()	()	L0 = ()

デバッグモードで 1 ステップ実行して眺めてみると、実行の様子が見られておもしろいです。

### 1.2.4 take と drop

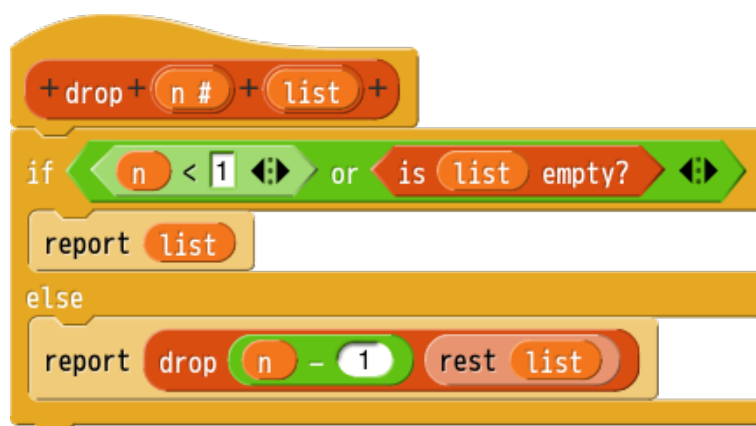
Haskell 言語にはリストの先頭から  $n$  個の要素を取り出す `take` と、リストから先頭の  $n$  個の要素を取り除いたリストを返す `drop` があります。以下のような動作になります。



定義です。take では先頭の要素に  $n$  が 0 になるまで次の要素を追加することを再帰を使って繰り返します。 $n$  が 0 の時と引数のリストが空の時は空リストを返します。



drop では  $n$  の値が 0 になるまで  $n$  の値を  $-1$  しながら先頭以降のリストに対して再帰を繰り返します。 $n$  が 0 の時に返すのは引数であるリスト、つまり求めるリストです。引数であるリストが空の場合は空リストを返すことになります。

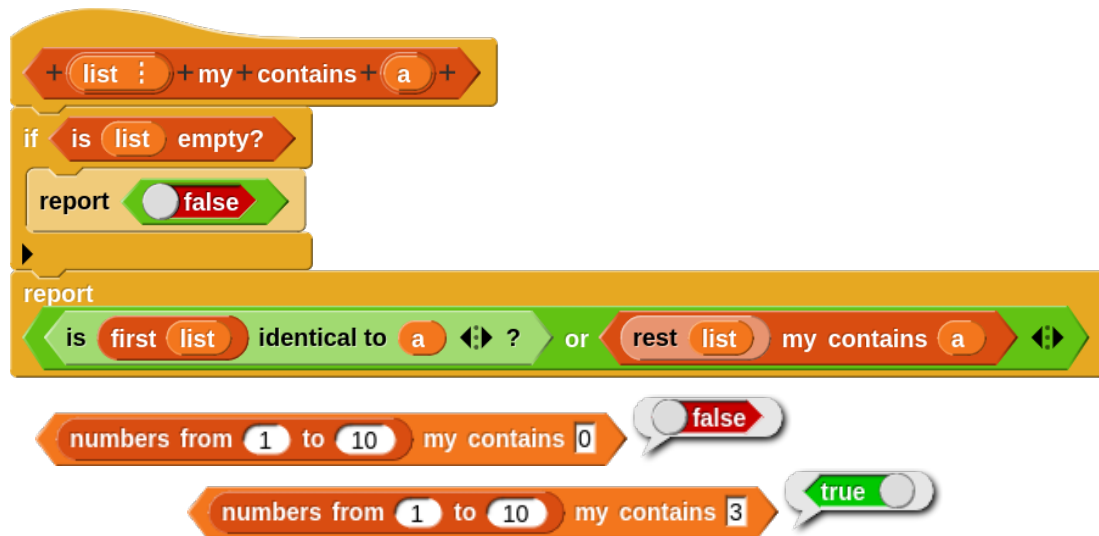




### 1.2.5 my contains

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。

リストの先頭が指定の要素ならば true を返します。そうでないならば、残りのリストに対して同じ操作を繰り返します。



### 1.2.6 my uniques

リストから重複を取り除く `uniques` of を再帰呼び出しで定義してみます。大文字小文字を区別するようにしています。

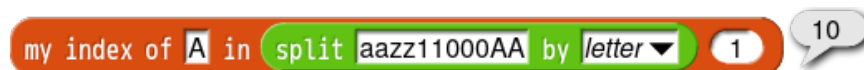
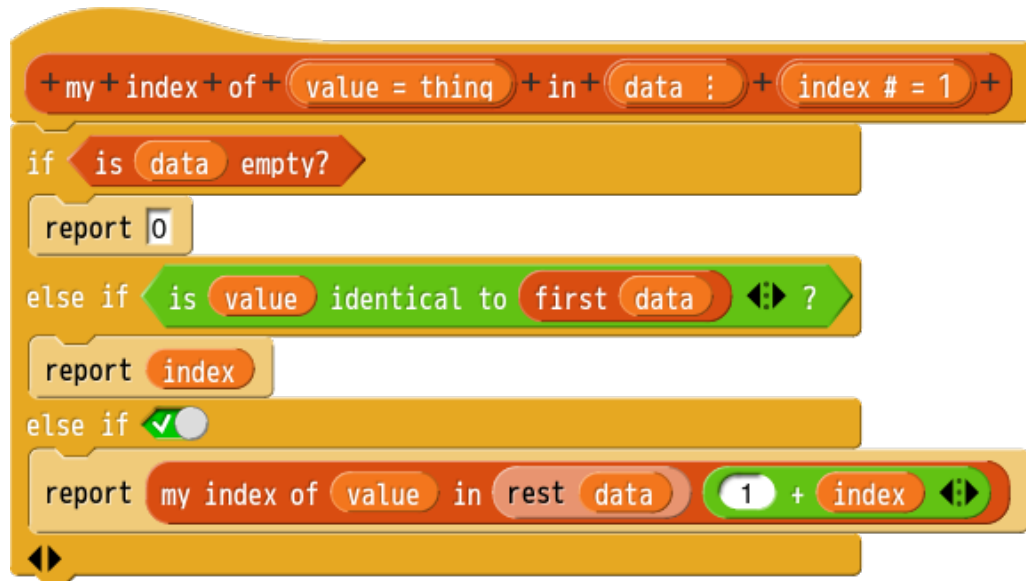


keep を使って先頭の要素と同じでないものを集めることを再帰的にを行っています。

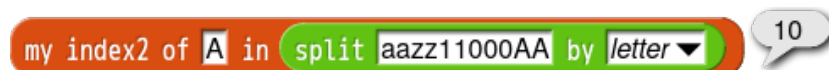
因みに、`not is identical to first list ?` を  
`not mod first list = 0` にして、`numbers from 2 to 100` を引  
数として実行すると 100 までの素数列が求められます。

### 1.2.7 my index of ( ) in ( )

`index of thing in` の大文字小文字を区別する版を再帰呼び出しで作ってみます。インデックス値を +1しながら走査する必要があるの値を渡していきます。

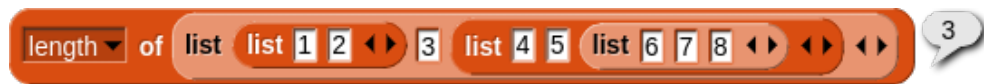


次のように局所定義ブロックにすると、プリミティブブロックと同じかたちになります。



### 1.2.8 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。



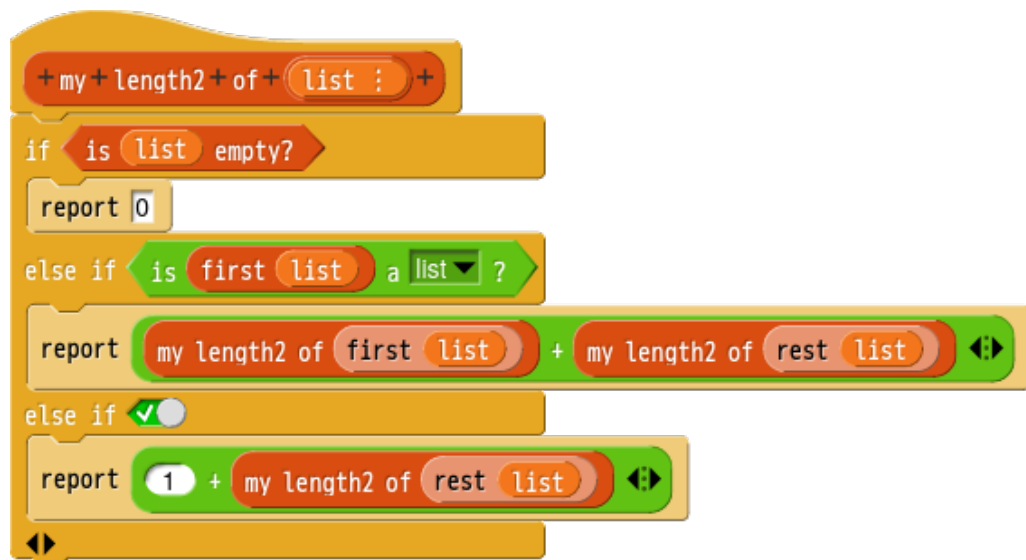
これは my length も同様です。



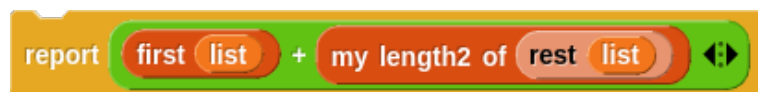
**flatten** of 後に length でもできますが、再帰を使って内部の要素に対してもアクセスしてみます。

処理の内容は次のようになります。

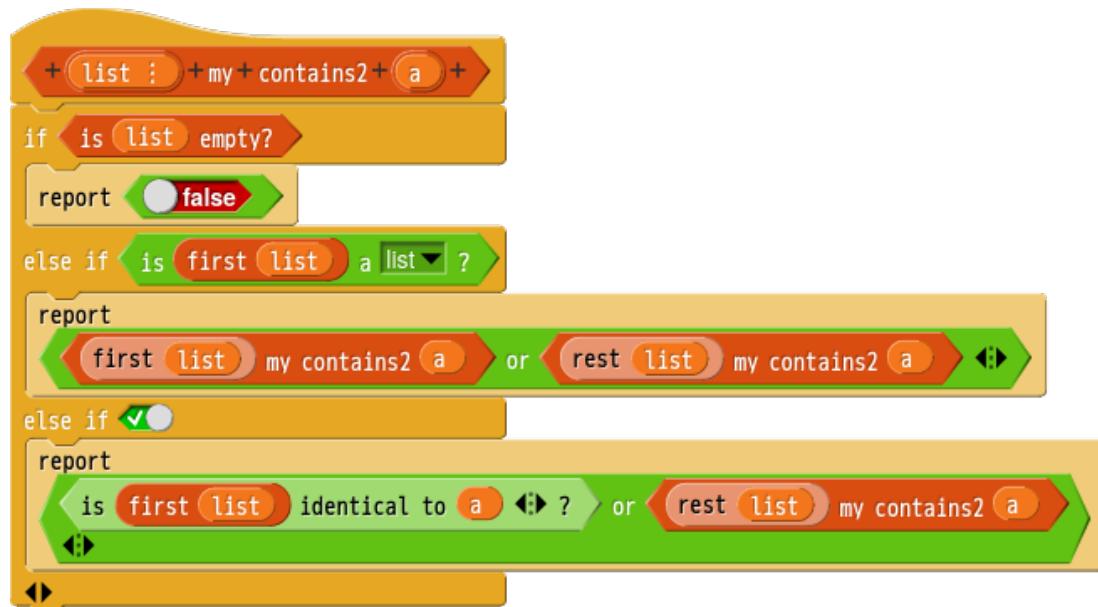
- もしリストが空ならば 0 をレポートする。
- もし先頭の要素がリストならば、そのリストに my length2 をしたものと残りに対して my length2 をしたものを加える。
- そうじゃなかったら、残りに対して my length2 をしたものに 1 を加える。



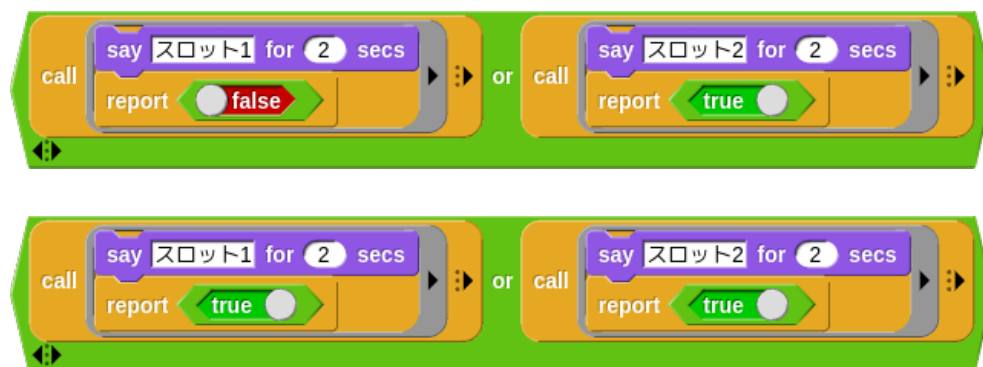
最後の report ブロックで 1 を加えるのではなく、要素の値を加えるようにすると合計を求めることができます。その場合のブロック名は sum of のようなものになると思いますが。



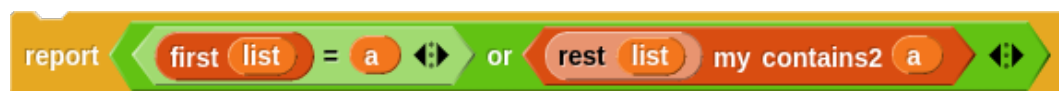
my length2 ブロックを応用すると my contains2 ブロックを作成することができます。true か false かを扱うので演算子は「or」を使用します。なお、is identical to ブロックの代わりに = ブロックを使用すると、マニュアル XI. Metaprogramming A.Reading a block 内の callers of ブロックで使用されている deep contains ブロックになります。



「and」ブロックでは左側のスロットから順にテストして、false ならば残りのスロットのテストは行いません。それに対して、「or」ブロックでは左側のスロットから順にテストして、true ならば残りのスロットのテストは行いません。



そのため、リストの先頭からテストしていった、指定の要素が見つければリストの残りはテストせずにそこで true を返して終了になります。



### 1.2.9 指定の要素に対する delete と replace

リストからある要素を取り除いたリストを求めることは keep を使えばできます。しかし、リストの要素がリストの場合はその内部までは対応しません。

3	A	B
1	3	
2	1	2
3	9	

keep items ☐ not ☒ = 2 ☐ from list 2 3 list 1 2 9 2

my length2 を応用した再帰版 delete です。

```

+delete+ a +of+ list : +
if is list empty?
  report list
else if is first list a list ?
  report delete a of first list : delete a of rest list
else if is first list identical to a ?
  report delete a of rest list
else if ✓
  report first list : delete a of rest list

```

1	3	-
1	1	-
2		-
	length: 1	
3	9	-
	length: 3	

delete 2 of list 2 3 list 1 2 9 2

リストを含まないリストの要素の入れ替えは `map` を使えばできますが、次のように内部のリストまで 2 を 7 に入れ替えることはできません。

5	A	B
1	7	
2	3	
3	1	2
4	9	
5	7	

`map` `if` `= 2` `then` `7` `else`  `over` `list` `2 3` `list` `1 2` `9 2`

指定の要素だった場合、それをコピーしないでリストの残りを続行すれば `delete` になりますが、置き換える要素をコピーすれば `replace` になります。

```


+replace+ old +with+ new +of+ list : +
if is list empty?
  report list
else if is first list a list ?
  report
    replace old with new of first list :
    replace old with new of rest list
else if is first list identical to old ?
  report new : replace old with new of rest list
else if
  report first list : replace old with new of rest list

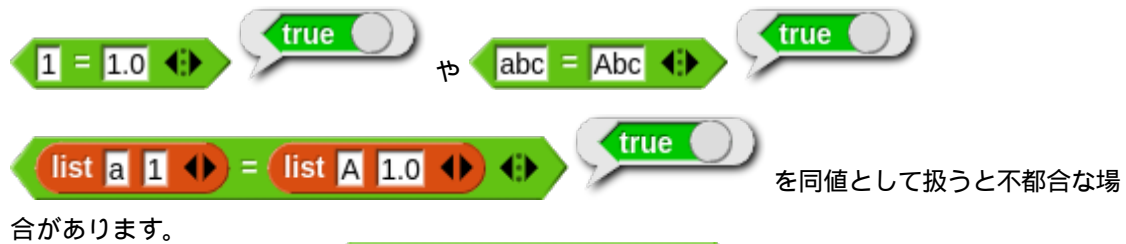
```

5	A	B
1	7	
2	3	
3	1	7
4	9	
5	7	

`replace` `2` `with` `7` `of` `list` `2 3` `list` `1 2` `9 2`

### 1.3 = と identical

二つの値が同じかどうかを調べる  ですが、



その場合に使えるのが、  ?



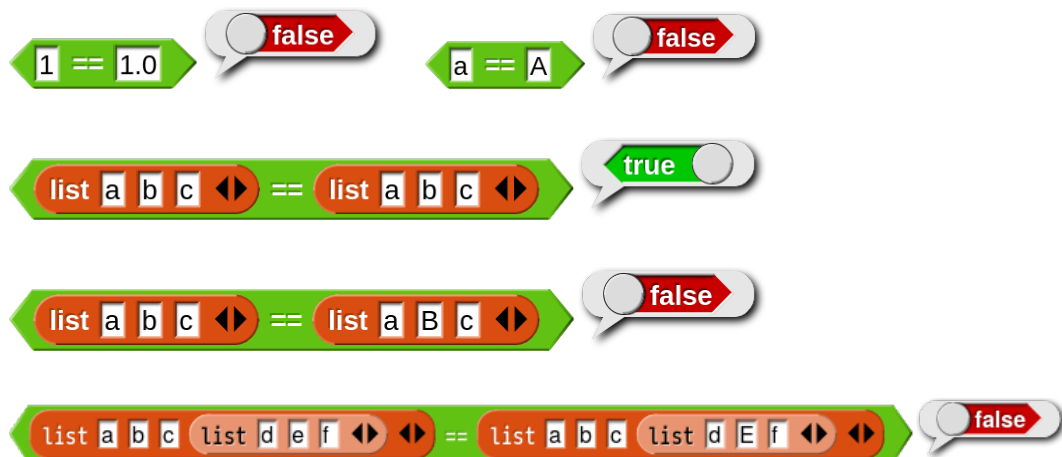
上記は両入力スロットの値についてテストしましたが、リストに対する場合はリストの在り処が同じかどうかのテストをするようです。次は要素値は同じですが、別々に存在するリストです。




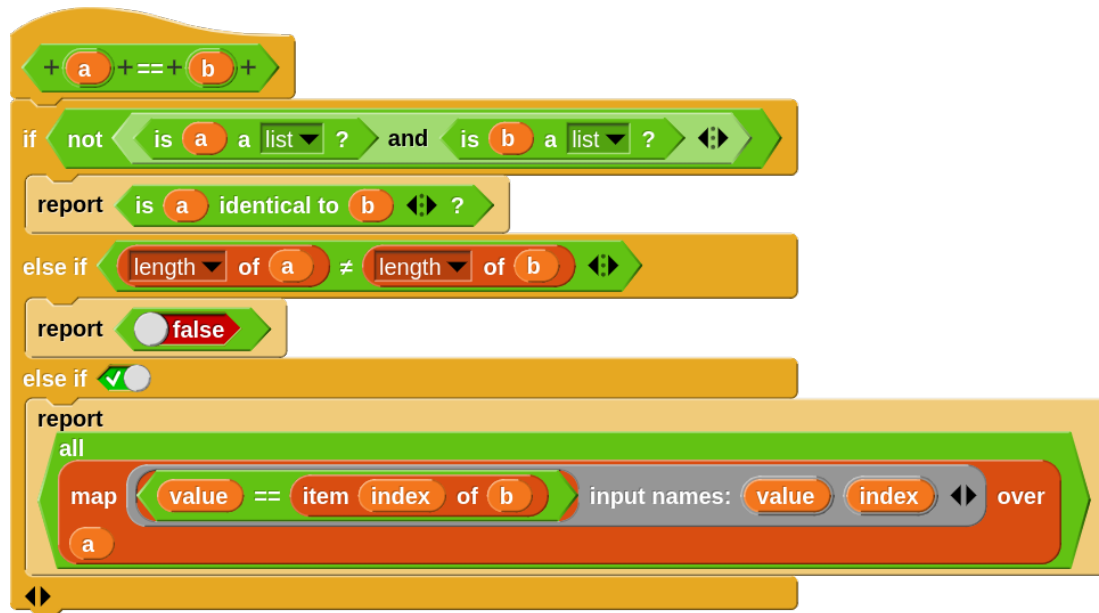
リストの要素を変える場合は元のリストを修正するのではなく、新しくリストを作成してそれを指すようにするようです。a は元のリストを指したままなので `false` になります。


リストに関しても値の同値性だけをチェックするブロックを作ってみます。

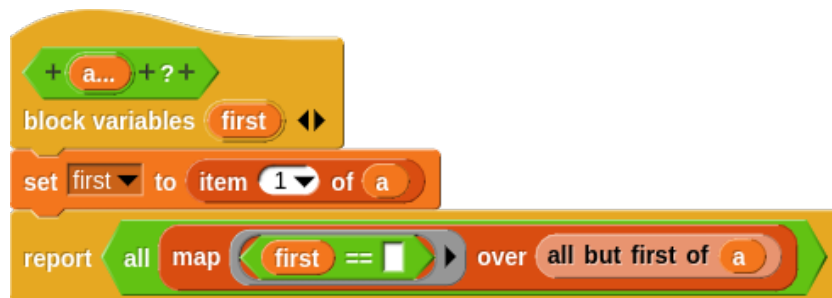
「`identical`」ではなく「`==`」としてみました。



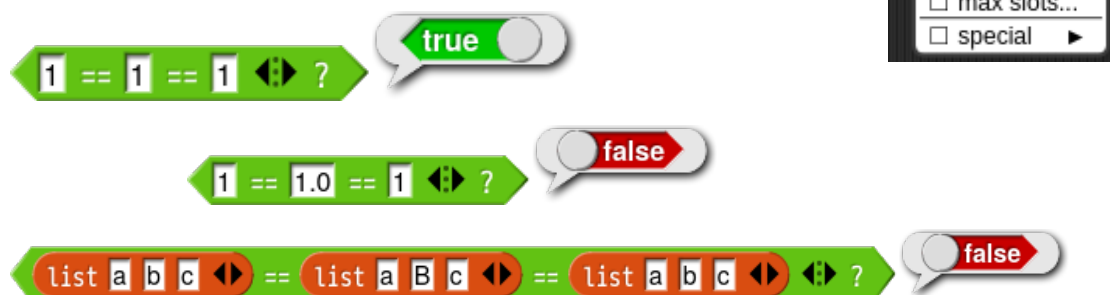
スクリプトです。a と b は Any type 型です。リストの要素にリストを含むものに対処するために再帰呼び出しを使用しています。map を使うとすべての要素をテストすることになります。なお、all は  の右端の三角マークの部分に map をドロップしたものです



is ( ) identical to ( ) では入力スロットを追加できます。  を使って次のようにすると、同じことができます。a を Any type 型 Multiple inputs (value is list of inputs) にします。



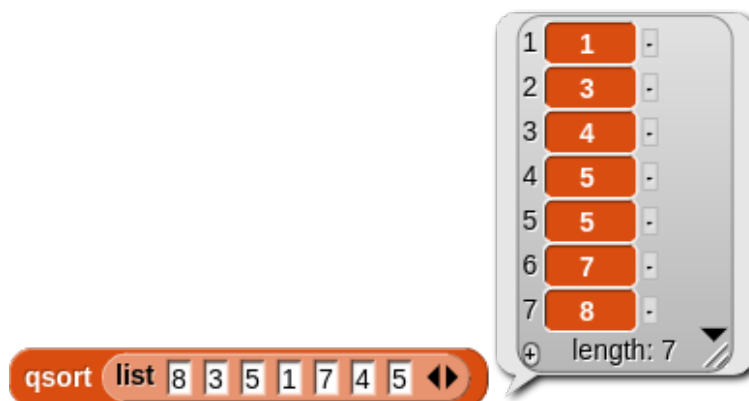
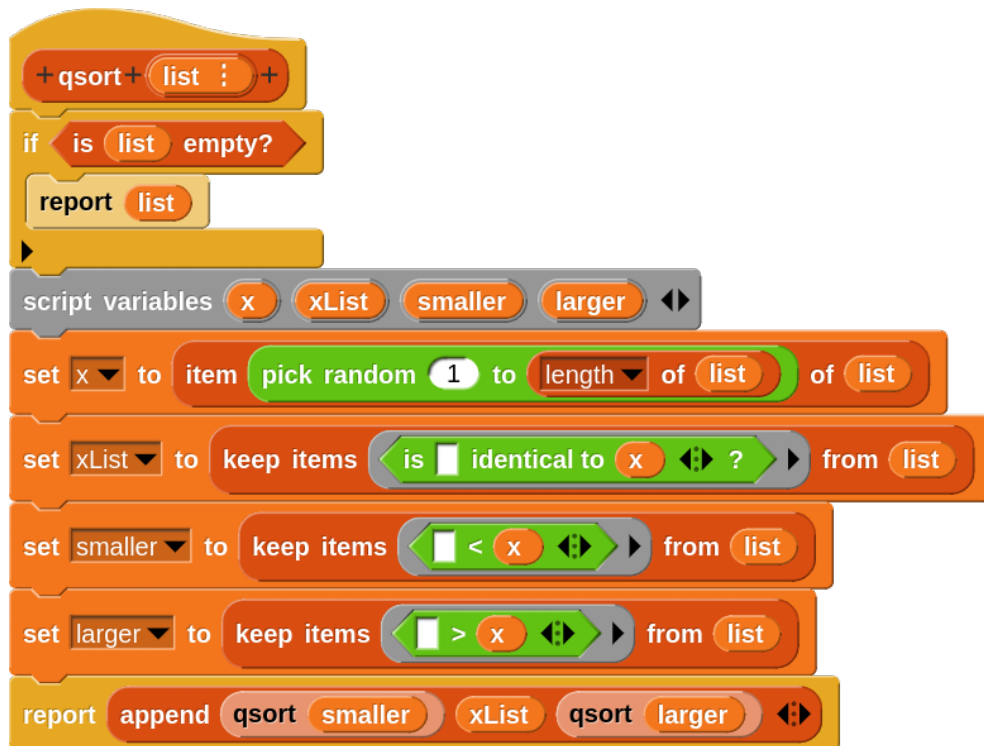
変数の設定で、入力スロットを増やす時に間に挿入する文字列 separator を == に、入力スロット数 0 の場合の文字列 collapse を all == に、初期値としての入力スロット数 initial slots を 2 にします。





### 1.3.1 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け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、リスト要素の入れ替えなどに煩わされず、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。

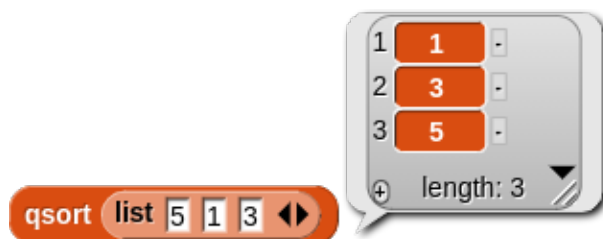


このように自分自身を複数参照することを多重再帰と言います。

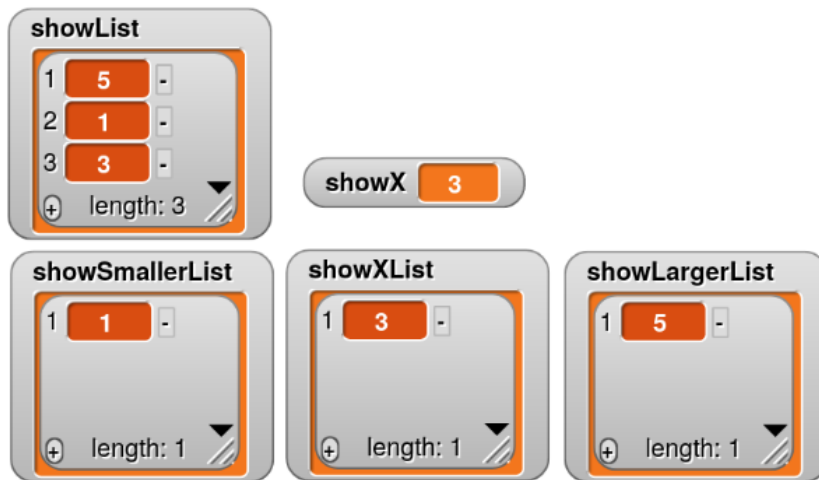
次のように showList, showX, showXList, showSmallerList, showLargerList のグローバル変数を作成し、リストを通して値を表示すると操作の様子が見られます。



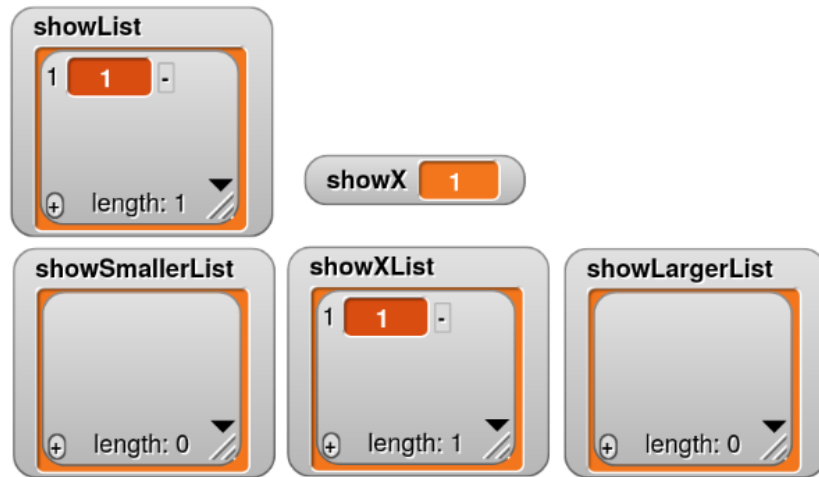
リストの値表示のたびに pause all になります。▶ をクリックして続行してください。



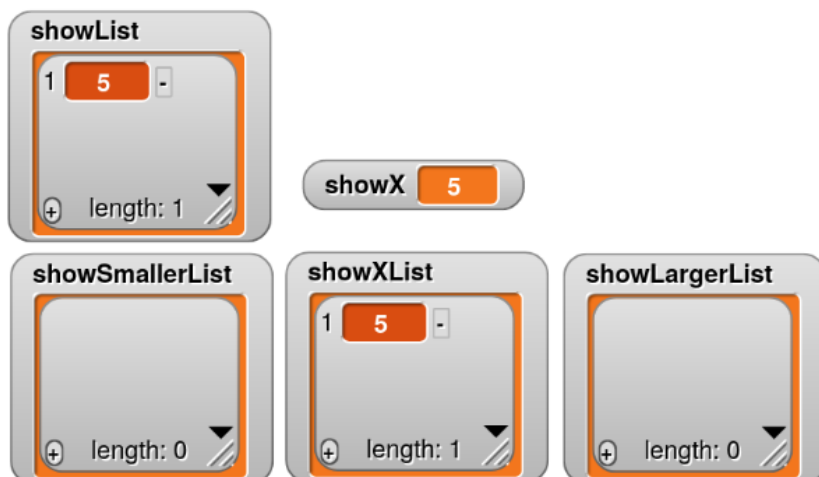
最初に選択された値が 3 だった場合です。



3 より小さい値のグループ処理



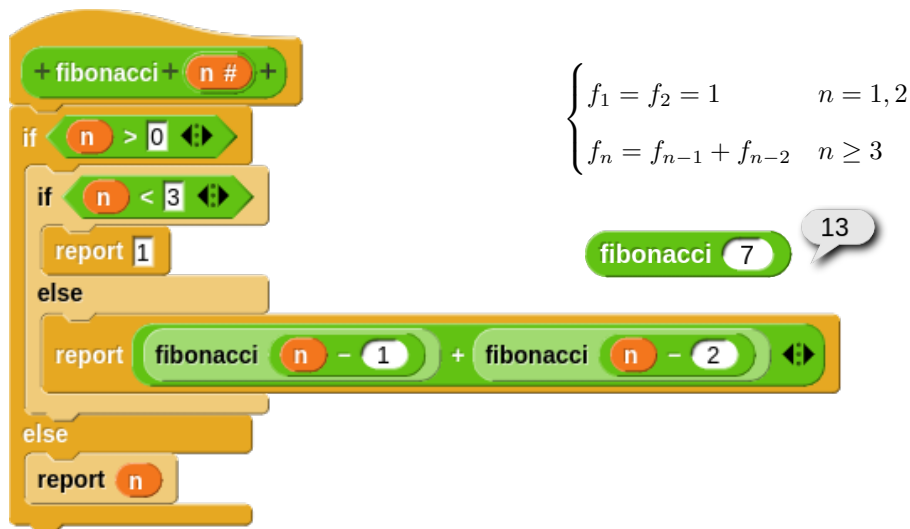
3 より大きい値のグループ処理



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。

### 1.3.2 フィボナッチ数列

再帰の例としてよく示されるフィボナッチ数列です。

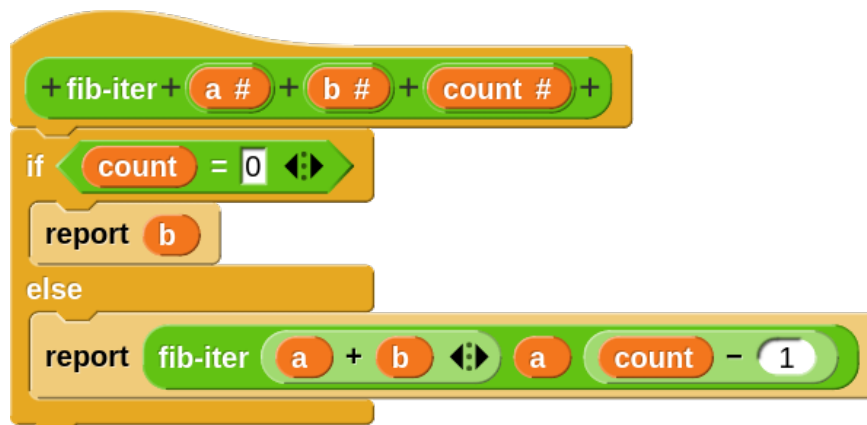


これは  $(n-1)$  と  $(n-2)$  を引数にして 2 度再帰呼び出ししています (多重再帰)。そのため、 $n$  が大きくなると時間がかかってしまいます。

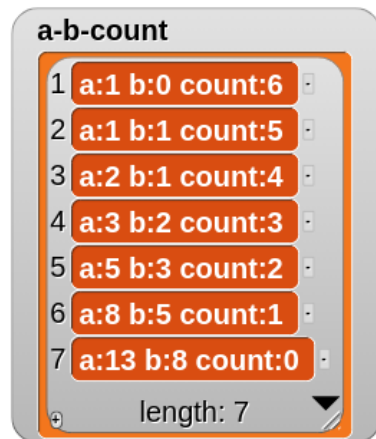
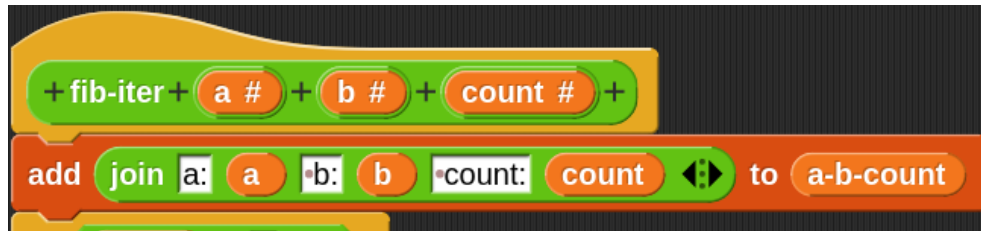
例えば  $\text{fib}(6)$  を求める場合は、右のようになりますが、 $\text{fib}(6)$  で必要とする  $\text{fib}(4)$  は  $\text{fib}(5)$  で処理済みで、 $\text{fib}(5)$  で必要とする  $\text{fib}(3)$  は  $\text{fib}(4)$  で処理済み ... ということで、処理済みのものはその値を渡してやれば済むのでその分の再帰呼び出しの必要がなくなります。

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

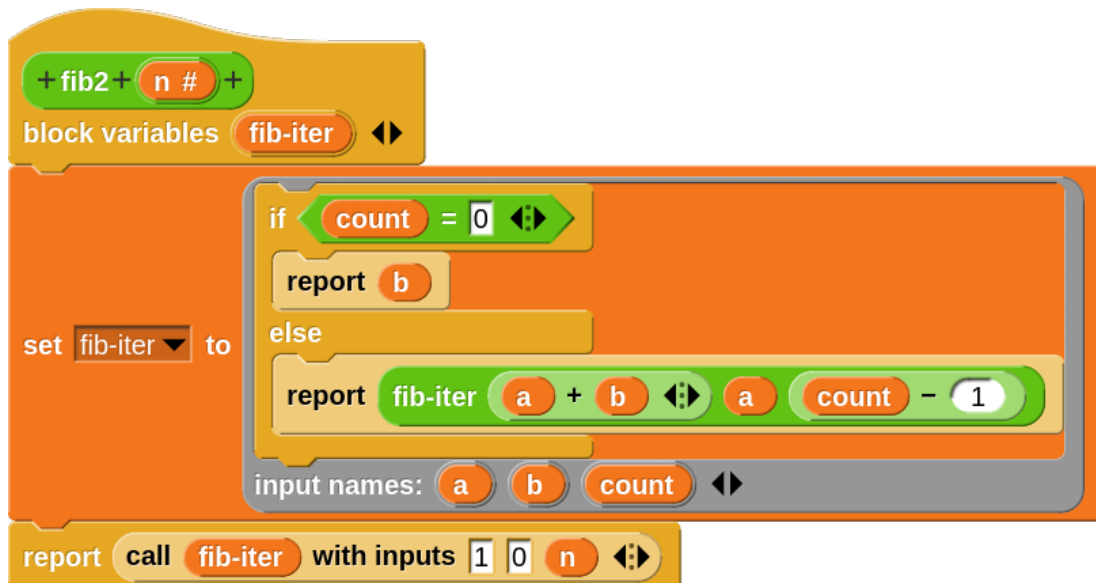
SICP の非公式日本語版翻訳改訂版 真鍋宏史氏訳 の 39 ページに反復プロセス版が載っていました。前二項の値を渡すことで余分な再帰呼び出しを避けています。



次のようにブロックを追加して a, b, count の値を表示させると動作が分かります。

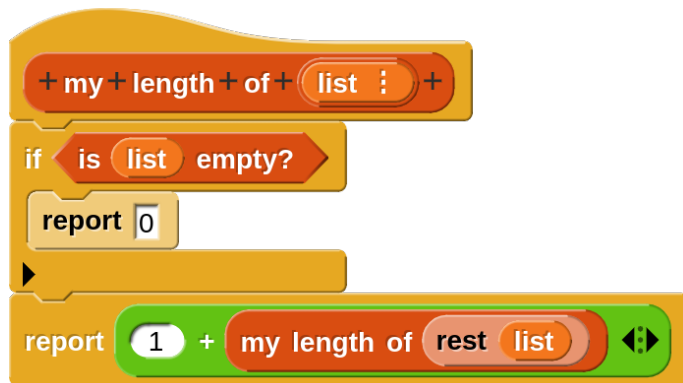


fib-iter はここでしか使わないので、本体内部で変数にセットすると局所定義ブロックにすることができます。



### 1.3.3 末尾再帰

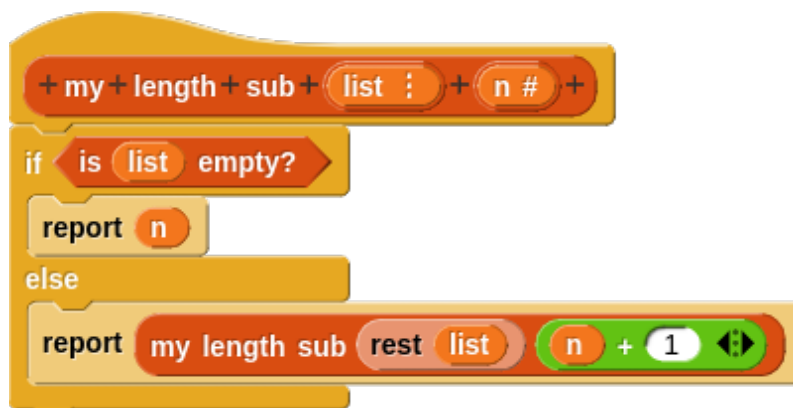
5 ページで my length を扱いました。



この定義ブロックでは、計算式に再帰呼出しが含まれていて、再帰呼出しによる値が確定しないと計算することができません。リストが空になると確定した 0 の値を使って順々に計算した値を戻しながら一番最初のところまで戻って、ようやく値 3 を得ることになります。my length of (1 2 3) を L(1 2 3) と表してみます。

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、次のようにすると再帰呼出しだけを行うという形にすることができます。  
引数 (n + 1) を渡していくことでカウントしています。

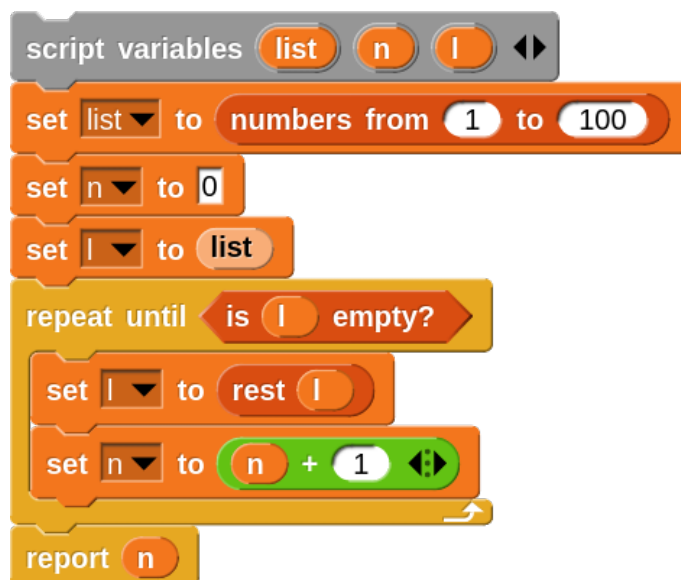


これ呼び出す本体定義は次のようになります。引数 `n` の値を 0 にすることで、カウンターの値を初期化しています。



```
my length3 of (1 2 3)
  my length sub (1 2 3)(0)
    my length sub (2 3)(1)
      my length sub (3)(2)
        my length sub ()(3)
          3
        3
      3
    3
  3
```

このように、一番最後の再帰呼出しの戻り値がそのまま定義ブロックの値になり、再帰呼び出しから戻る時は単に値をそのまま渡すだけです。このような処理の最後を単独の再帰呼び出しにする形を末尾再帰といいます。末尾再帰のスク립トは再帰を使わない反復のスク립トに変換することができます。



Scheme や Haskell などでは、末尾再帰はループに最適化されるので高速になりますが、Snap! ではさほど変わらないようなので末尾再起にこだわる必要もないようです。

## 2 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。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、引数に関数型ブロックを指定する高階関数型のブロックです。

### 2.1 高階関数型ブロックの基本

半径を入力して、円の面積をレポートする関数型ブロックです。



たとえば、`円の面積 2 + 5` や `list 円の面積 1 円の面積 2` のように入力に関数型ブロックを使用しても、それは結果としての値を使用しているだけで高階関数型とは言えません。つまり、それぞれ `12.56 + 5` や `list 3.14 12.56` の意味になります。

`map` のような高階関数型ブロックの場合は、リスト要素の処理方法（スクリプトブロック）を指定するという意味になります。



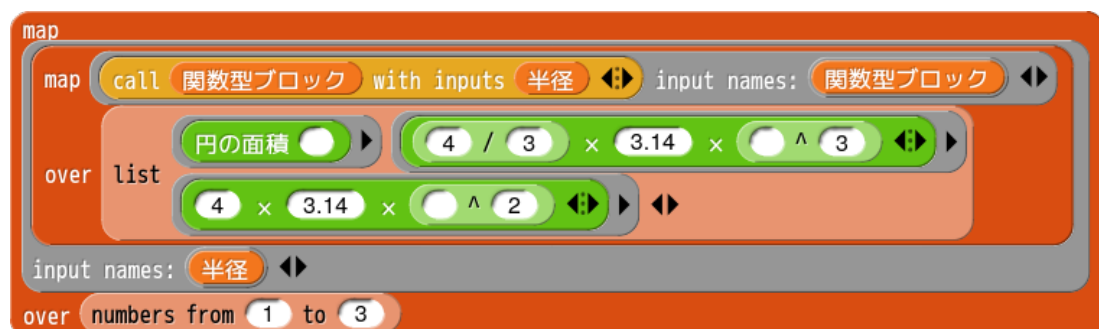
リストを半径とする球の体積を求めます。



リストを半径とする球の表面積を求めます。



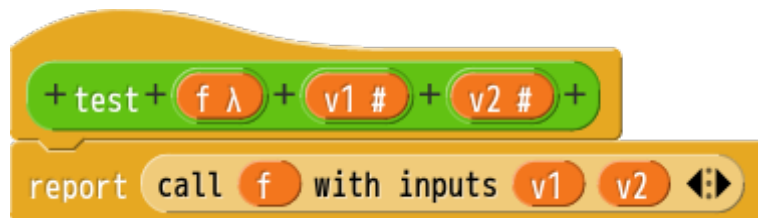
`map` を二重にするとこんな使い方もできなくはありません。





Snap! のユーザー定義ブロックでは高階関数型のものを作成することができます。入力スロットでスクリプトブロックを受け取ることができますし、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。

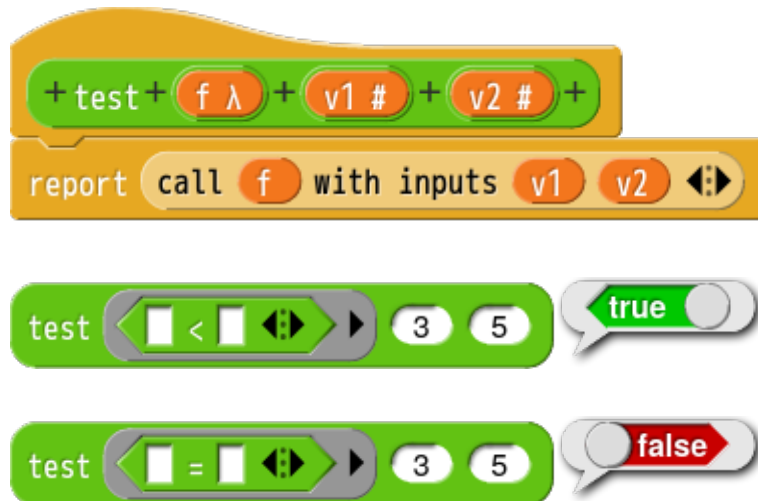
次のブロック定義 test は二つの引数を必要とする関数型ブロックを実行して値を返すものです。入力 f は Reporter 型です。v1 と v2 の入力は Number 型です。関数型ブロックの実行は call ブロックを使用します。これが高階関数型の定義ブロックの基本となります。



入力スロット f に関数型ブロックを入れて処理を指定します。



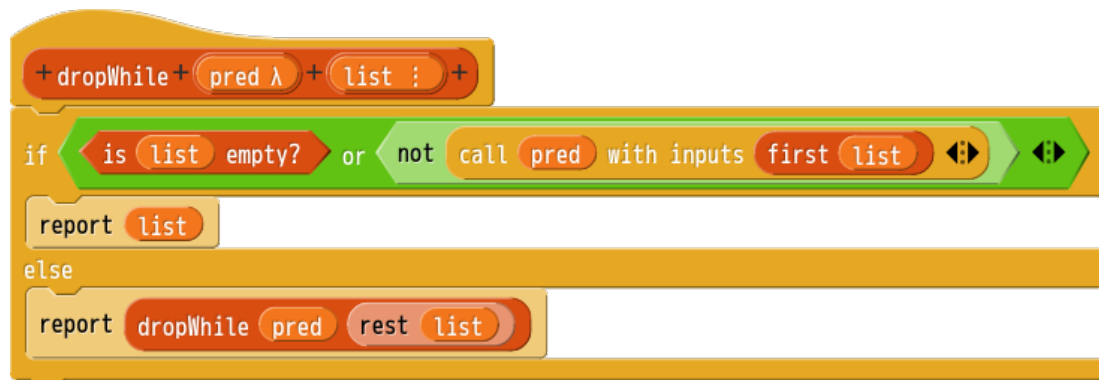
入力 f を Predicate 型にすると、次のように変更されます。(ブロック定義自体の見た目は同じですが)



高階関数型として入力に使用するタイプは Reporter, Any(unevaluated), Predicate, Boolean(unevaluated) が利用できます。実は、Any type や Boolean(T/F) でも入力を ringify してやれば同じ機能になります。

## 2.2 高階関数型の take と drop

8 ページで要素の個数を指定する take と drop を定義しました。高階関数型ブロックの例として、リストの先頭から条件に一致する要素だけを take や drop するブロック takeWhile と dropWhile を作成してみます。不一致の要素が見つかった時点で操作は終了です。条件をチェックするブロックを受け取る変数 pred は Predicate 型です。



## 2.3 操作を指定するリストの巡回

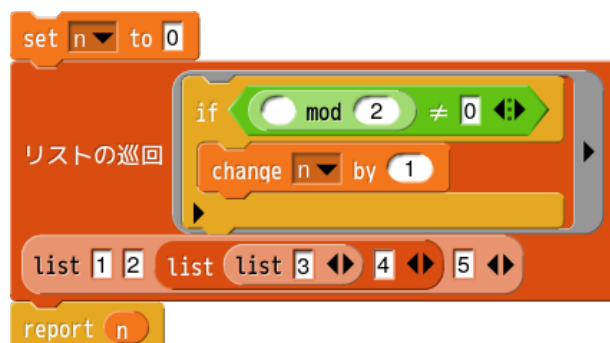
11 ページで「リスト要素の巡回」を扱いました。引数で操作を指定すれば同じ定義ブロックでいろいろな用途に役立ちます。定義ブロック自体は Command 型で、引数 proc は Command(inline) 型です。proc は引数の一つを使用します。



リストの要素数を求めます。引数としてリストの要素が用意されますが、[change (n) by (1)] と (1) でふさがっているので要素は使用されずに 1 でのカウントになります。

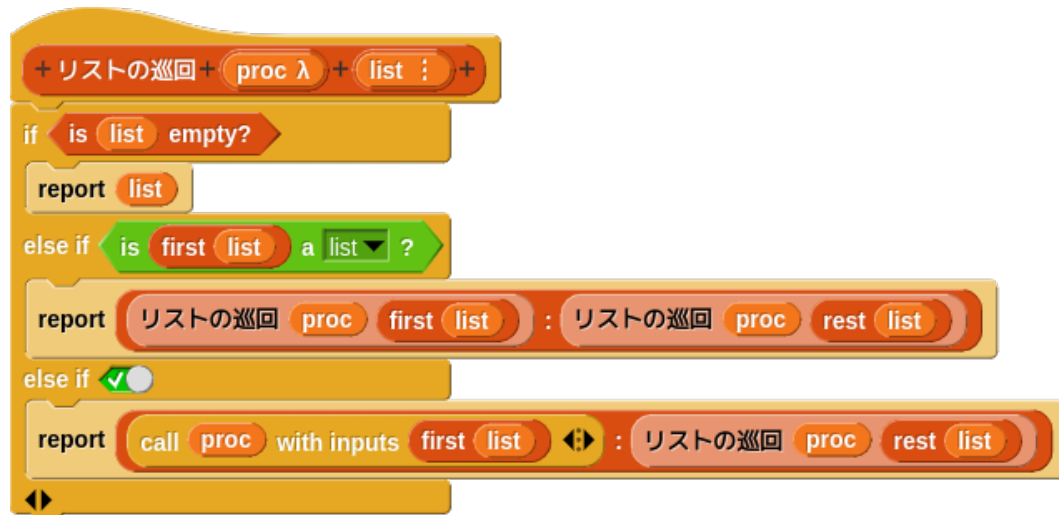


リストの要素の値の合計値を求めます。上とは違い [change (n) by ( )] と引数のスロットにリストの要素が入り加算されます。

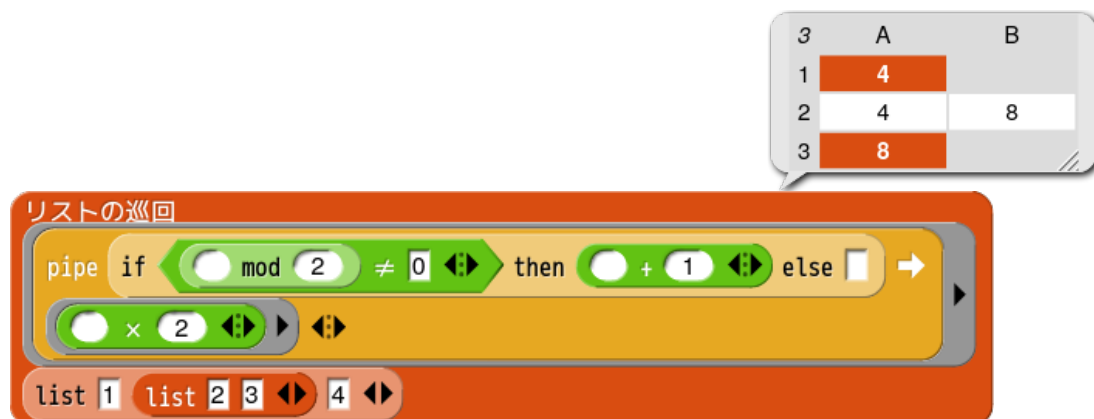


リスト中の奇数要素の数を求めます。

リポーター版です。引数 proc は Reporter 型です。



意味のある例ではありませんが、要素が奇数の時は 1 を加えます。そして各要素を 2 倍します。



## 2.4 foldl, foldr

Haskell 言語には `foldl` と `foldr` というリスト操作関数があります。

これは、二つの引数をとる関数 `proc` (変数型は `Reporter`) と、初期値 `init` と `list` の合計3つを引数として取る関数です。初期値 `init` と `item` (`list` の先頭要素) の二つの値を引数として `proc` が何らかの処理をします。得られた値を新たな `init` とします。その `init` と `item` (`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。 `init` の値が最終値になります。 `foldl` はリストを左側から順に操作し、 `foldr` はリストを右側から順に操作するものです。

`foldl` から見ていきます。



操作内容は、 のようになります。

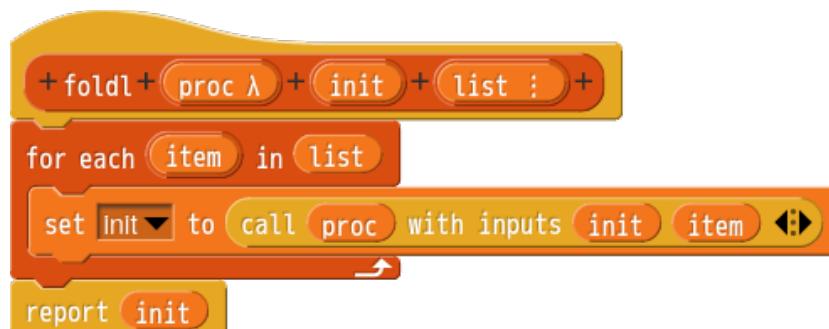
フォーマルパラメータを使用すると次のようになります。



フォーマルパラメータの順序は右下の表のように `init`, `item` の順になります。( #1, #2 から名前の変更をしています )

init	+	item	値
0	+	1	1
1	+	2	3
3	+	3	6
6			6

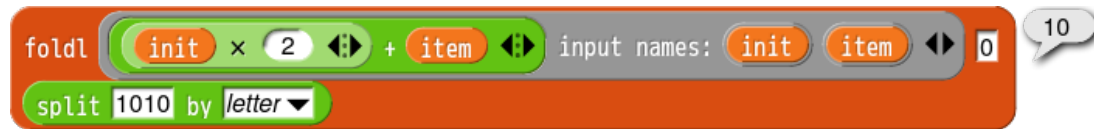
非再帰版だとこうなります。



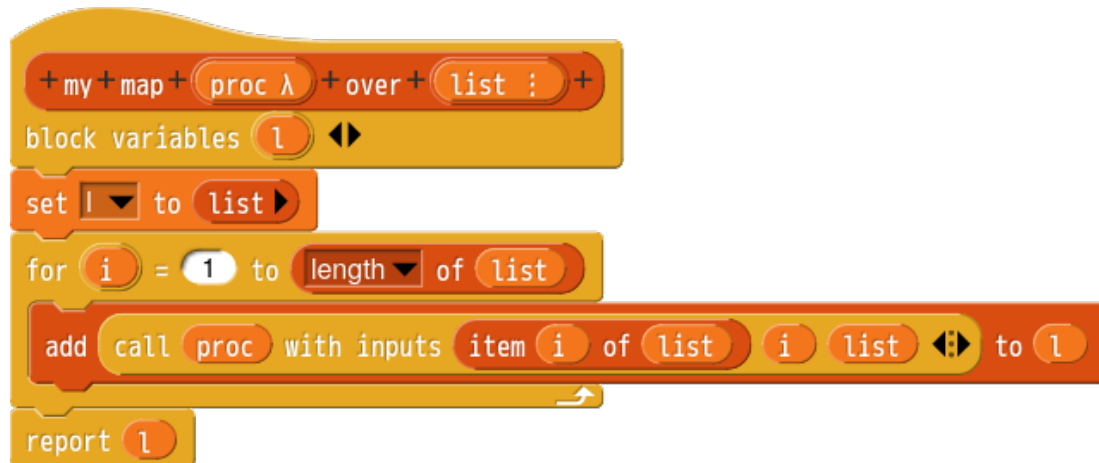
次のようにすると、リストの要素の値は使用しないで要素数 length を求めることになります。



次のようにすると文字列の二進数 (1010) を十進数に変換します。



map ブロックを再帰呼び出しを使用しないで作成してみます。引数 proc は Reporter 型です。プリミティブのもののように value, index, list 値を利用可です。



再帰呼び出しを使用して作成してみます。value, index, list 値は利用できません。



foldl 版です。



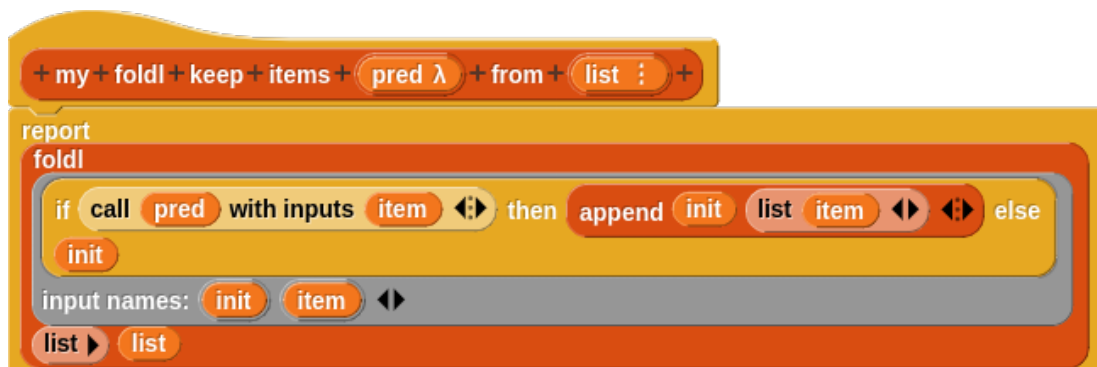
keep ブロックを再帰呼び出しを使用しないで作成してみます。引数 pred は、Predicate 型です。



再帰呼び出しを使用して作成してみます。



foldl 版です。



foldr です。



操作内容は、 のようになります。

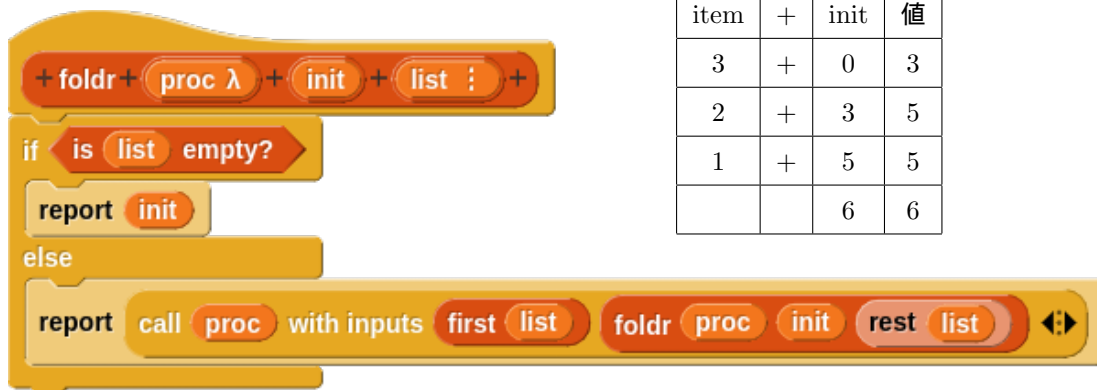
フォーマルパラメータを使用すると次のようになります。



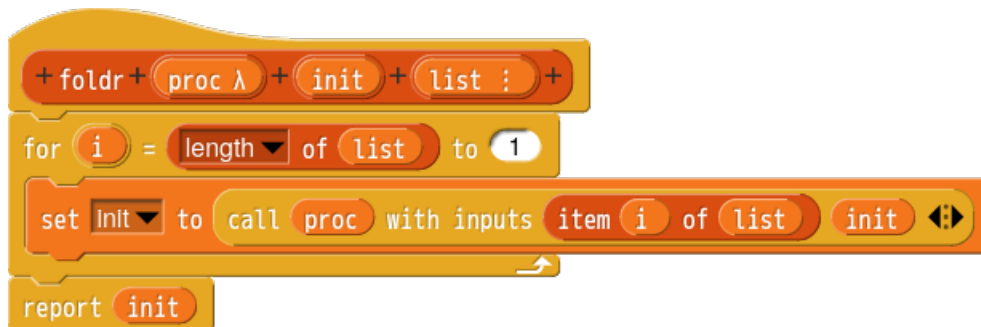
フォーマルパラメータの順序は右下の表のように、foldl とは違う item, init の順になります。

foldr

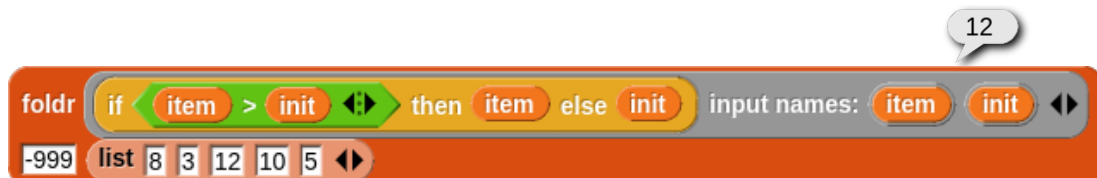
item	+	init	値
3	+	0	3
2	+	3	5
1	+	5	5
		6	6



非再帰版だとこうなります。

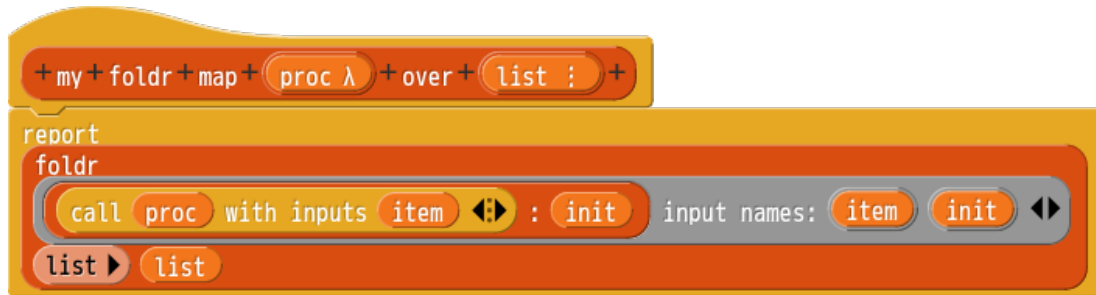


次のようにすると最大値を求めることができます。





foldr を使っても foldl と同じようなことができます。



foldr がリストを右側から操作することを利用した append です。

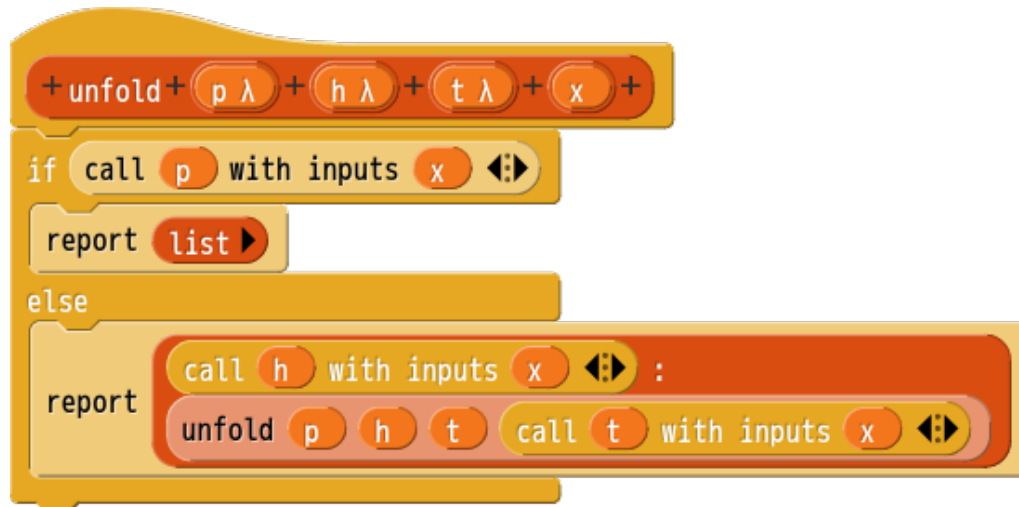


foldl で作成する場合には、リストを逆順にする操作が必要になります。



## 2.5 unfold

fold は、リストに対して操作を行い値を求めるものでした。逆に、ある値に操作を加えてリストを作成するのが unfold です。



`p` は、Predicate 型で処理（再帰呼び出し）の終了条件を指定するものです。`h` は、指定された値 `x` に操作を加えてリストの要素となる値を取り出す Reporter 型のブロックです。`t` は、次の要素を求めるために `x` を処理するための Reporter 型ブロックです。`unfold` はリストの要素を左から並べていきます。`p`, `h`, `t` はそれぞれ関数型のブロックなので、`unfold` は高階関数型のブロックです。

二進数をリストで表したもののから十進数に変換するには、`foldl` を使用して次のようにできます。



逆の操作は `unfold` を使用すると、次のようにできます。



値を 2 で割ると商と余りが求められます。余りが最下位桁の値になります。商は次の桁を求めるための値になります。この操作を商が 0 になるまで行います。リストは逆順で作られるので `reverse` する必要があります。

リストの要素を右から並べていく `unfoldr` を作成するならば、次のようになります。

The script defines a function `unfoldr` that takes a lambda expression `p λ + h λ + t λ + x +` as input. It uses an `if` block to check the result of `call p with inputs x`. If it's truthy, it reports the `list`. Otherwise, it reports an `append` block containing `unfold p h t call t with inputs x` and `list call h with inputs x`. Below the script, the `unfoldr` block is shown with arguments: a counter starting at 0, a modulo 2 operation, a floor division by 2, and the number 10. The output monitor shows a list of 4 elements: 1, 0, 1, 0.

```

+unfoldr+ p λ + h λ + t λ + x +
if call p with inputs x
  report list
else
  report append
    unfold p h t call t with inputs x
    list call h with inputs x

```

unfoldr [ ] = 0 mod 2 floor of / 2 10

1 1  
2 0  
3 1  
4 0  
length: 4

`unfold` を利用して `map` を作成することもできます。

The script defines a function `my unfold map` that takes a lambda expression `f λ + list :` as input. It reports an `unfold` block with arguments: `is empty?`, `call f with inputs first`, `rest`, and `list`. Below the script, the `my unfold map` block is shown with arguments: an addition of 2 and the range 'numbers from 1 to 5'. The output monitor shows a list of 5 elements: 3, 4, 5, 6, 7.

```

+my+unfold+map+ f λ + list : +
report
  unfold is empty? call f with inputs first rest list

```

my unfold map [ ] + 2 numbers from 1 to 5

1 3  
2 4  
3 5  
4 6  
5 7  
length: 5

## 2.6 カリー化

複数の引数が必要な関数を、一つの引数だけ受け取って処理をすることを連ねて行うやり方があります。そのように 1 引数の関数化することをカリー化と言います。カリー化された関数は関数を返す高階関数です。

ある文字を文字列に加えるには、join ブロックを使って次のようにできます。



カリー化したブロック join-c は入力スロットから文字列を受け取り ringify されたブロックを返します。そのブロックは実行されると加える文字を引数として受け取り、結果として合成された文字列を返します。



リポーターブロックを実行するには call ブロックを使用しますが、call ブロックにはリングが装備されています。そこに ringify されたブロックを渡されると次のようになってしまいます。



join-c ブロック部分を右クリックして unringify する必要があります。

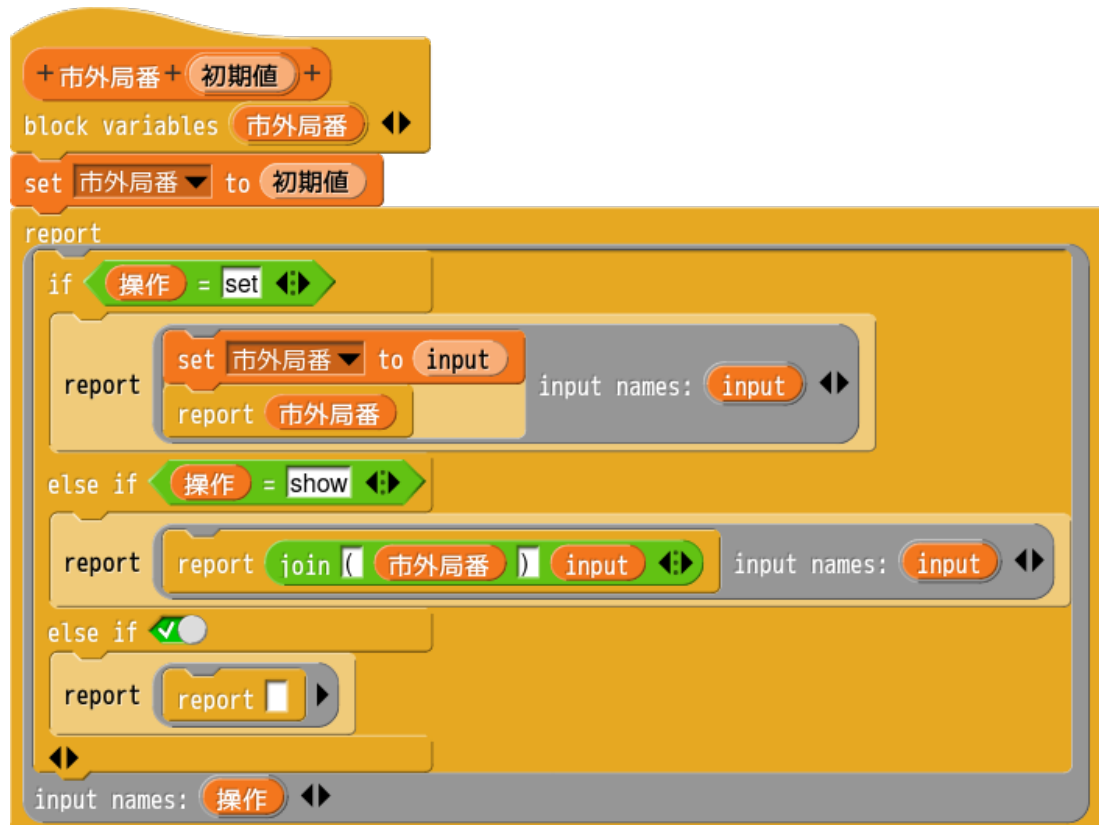


set 東京(03) to join-c (03) のようにすると、部分適用することもできます。



変数「東京 (03)」のように引数を含めて閉じ込められた関数（関数を返す高階関数）は「1234-5678」のように外側から与えられた引数にアクセスすることができます。このような環境をクロージャーと言います。

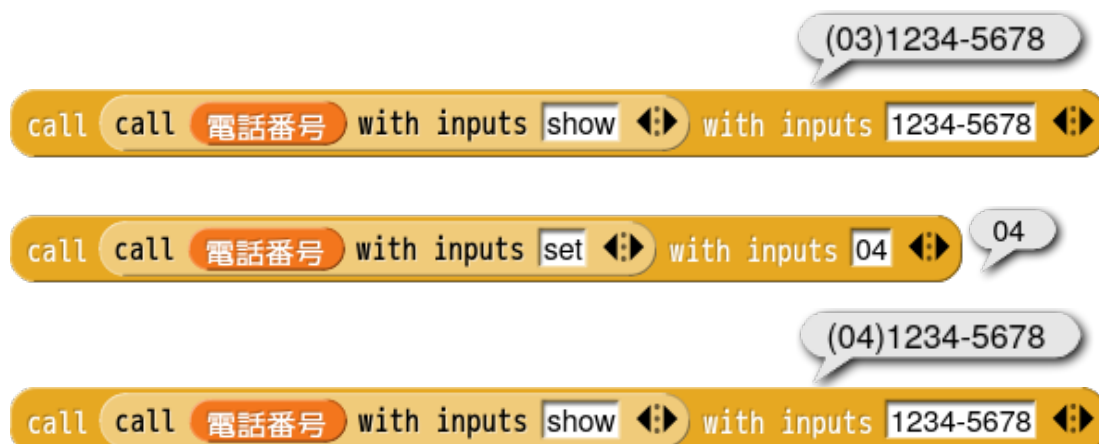
次のようにすると、クロージャー内のローカル変数「市外局番」に対して変更ができます。  
 report ブロックの中に操作に応じてそれぞれに report ブロックがあります。report ブロックが返すのは ringify されたブロックなので、これを実行するには二重の call ブロックが必要です。



変数「電話番号」をクロージャーにします。



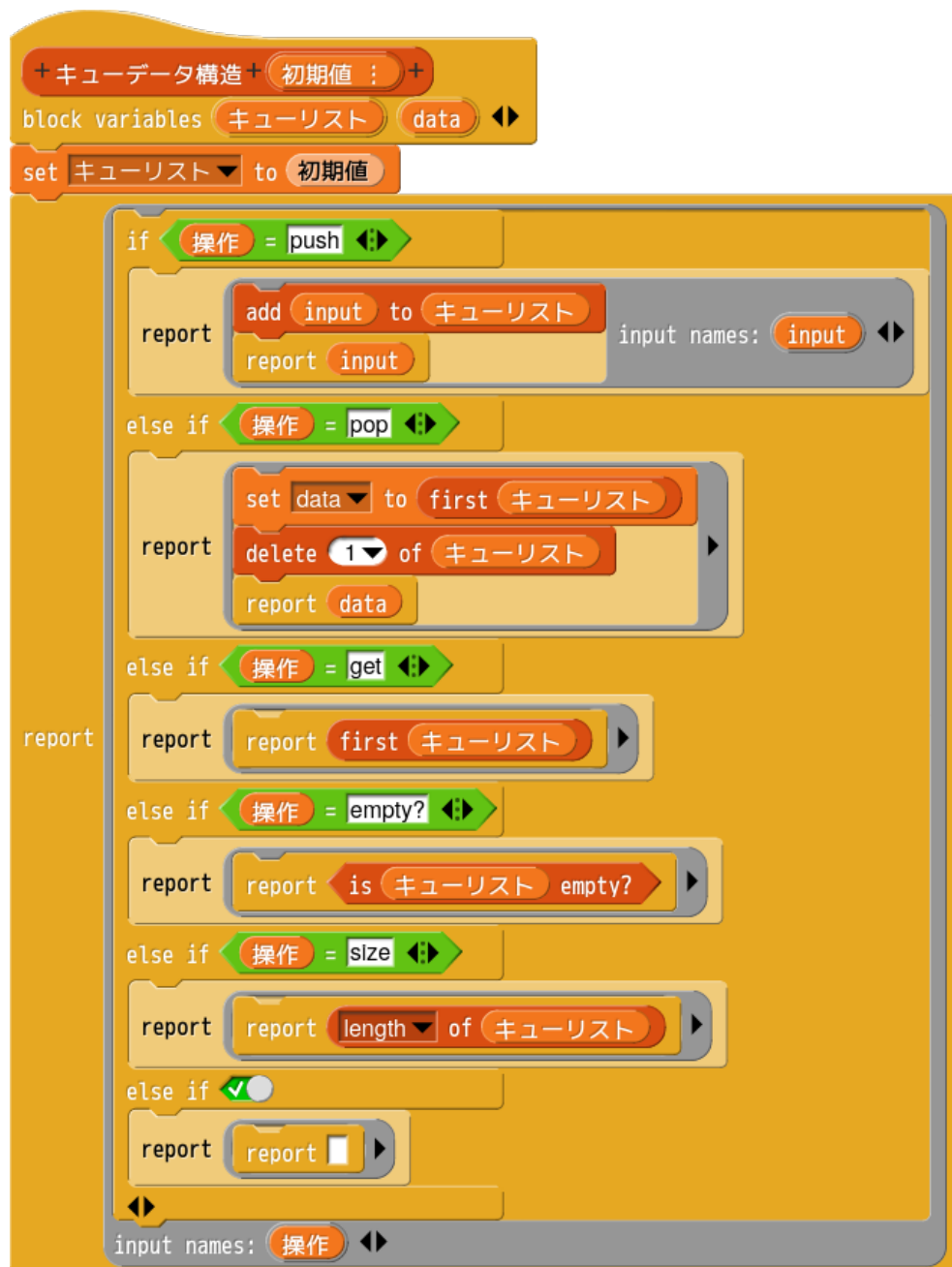
そうすると、「電話番号」を操作することができるようになります。



「市外局番」は外部から set ブロックで変更することができないので、安全な仕組みになります。

応用すると、キューやスタック、辞書などのデータ構造を作成することができます。

キューは待ち行列です。データの格納 (push) は列の後尾から、取り出し (pop) は列の先頭からになります。先頭データの参照 (get)、キューが空かどうか (empty?)、キューのサイズ (size) の機能を加えます。



変数「キュー」をクロージャーにします。



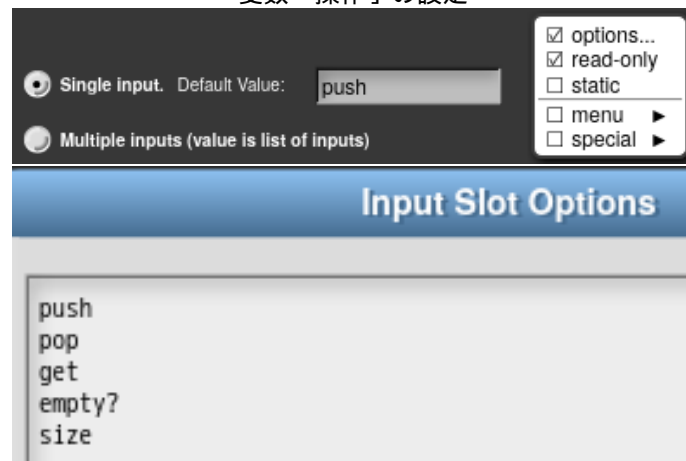
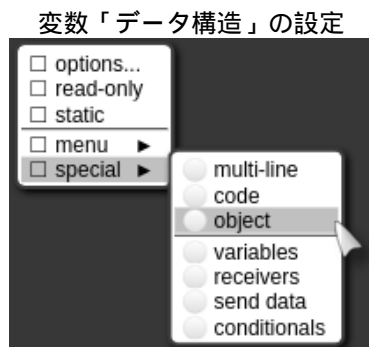
call call キュー with inputs push with inputs 2 2

call call キュー with inputs pop with inputs 1

操作ブロックを作成します。操作は options...(push, pop, get, empty?, size)、read-only で指定するようにします。Default value は push です。変数「データ構造」を special -> object に設定し、キューデータ構造やスタックデータ構造を持つ変数を指定して使えるようにします。

+コレクション+ データ構造 >> + 操作 = push + data +  
report call call データ構造 with inputs 操作 with inputs data

#### 変数「操作」の設定



set キュー to キューデータ構造 list

コレクション キュー push 1 1

コレクション キュー push 2 2

コレクション キュー push 3 3

三番目の入力スロットは空になっていますが、値を入れたとしても使用されません。

コレクション キュー pop 1 1

コレクション キュー pop 2 2

コレクション キュー pop 3

コレクション キュー pop

コレクション キュー empty? true

キューを使って九九のドリルを作ってみます。2 ~ 9 の問題をシャフルしてばらばらの順番にします。シャフルしなければ昇順になりますし、リバース又は 9 ~ 2 にすれば降順の問題になります。問題をキューにセットします。「pop した間に正解できなければその問をキューに push する」をキューが空になるまで繰り返します。その段をクリアしないと終わりません。間違えた問には「!」を付けます。

```
when clicked
  script variables
  set 段 to 3
  set c to 1
  set 段問題 to キューデータ構造 shuffled of numbers from 2 to 9
  repeat until (コレクション 段問題 empty?)
    set 残り to コレクション 段問題 size
    set 問 to コレクション 段問題 pop
    set 答え to 段 x 問
    set チェック to if (c > 8) then "!" else ""
    ask join (残り) チェック 段 "x" 問 "=?" and wait
    if (answer = 答え)
      say よくできました for 2 secs
    else
      say よくかんがえよう for 2 secs
      say join 段 "x" 問 "=? 答え "です for 2 secs
    set dummy to コレクション 段問題 push 問
    change c by 1
```



スタックはデータを積み上げるイメージで、取り出しは積み上げたデータのトップからになります。使用できる機能はキューと同じにしています。

+スタックデータ構造+

block variables スタックリスト data

set スタックリスト to list

if 操作 = push

report

add input to スタックリスト

report input

input names: input

else if 操作 = pop

set data to item last of スタックリスト

report

delete last of スタックリスト

report data

else if 操作 = get

report

report item last of スタックリスト

else if 操作 = empty?

report

report is スタックリスト empty?

else if 操作 = size

report

report length of スタックリスト

else if

report

report

input names: 操作

set スタック to スタックデータ構造

コレクション スタック push 1

コレクション スタック push 2

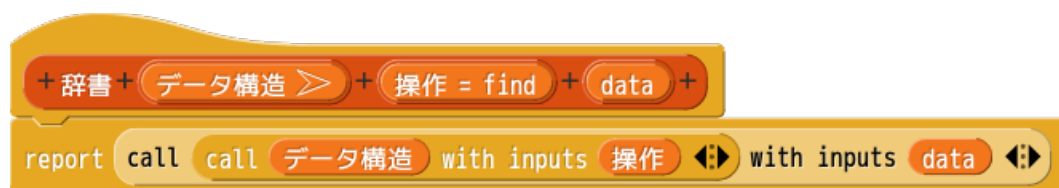
コレクション スタック push 3

三番目の入力スロットは空になっていますが、値を入れたとしても使用されません。

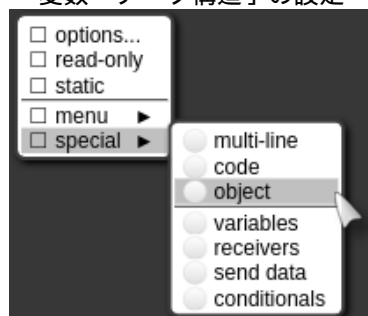


辞書は、キーとデータが対になったものを格納します。連想リストという言い方もあります。検索と削除はキーの値を指定して行います。

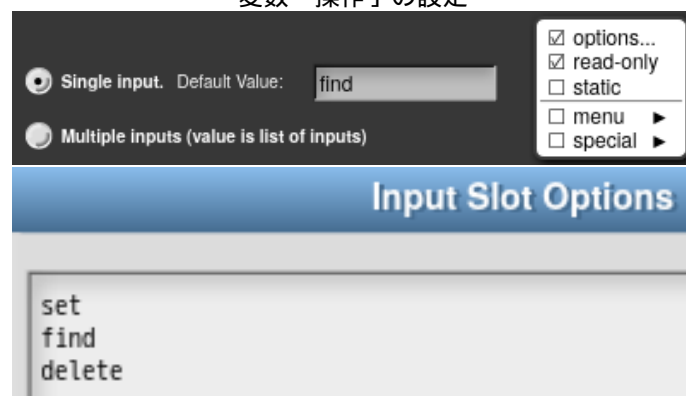
紙面の都合で、操作の定義ブロックを先に示します。変数「操作」は options...(set, find, delete)、read-only で指定するようにします。Default value は find です。変数「データ構造」を special -> object に設定し、辞書データ構造を持つ変数を指定して使えるようにします。



変数「データ構造」の設定



変数「操作」の設定



+ 辞書データ構造 + 初期値 : +

block variables 辞書リスト index data

set 辞書リスト to 初期値

report

if 操作 = set

report

if is input a list?

set index to

index of find first item first 目 = first input in 辞書リスト in 辞書リスト

if index > 0

delete index of 辞書リスト

insert input at 1 of 辞書リスト

report input

else

report

input names: input

else if 操作 = find

report

set data to find first item first 目 = input in 辞書リスト

report if data = then else item 2 of data

input names: input

else if 操作 = delete

report

set index to

index of find first item first 目 = input in 辞書リスト in 辞書リスト

if index > 0

set data to item index of 辞書リスト

delete index of 辞書リスト

report data

else

report

input names: input

else if

report report

input names: 操作

set 辞書 to 辞書データ構造 list

1 one  
2 1  
length: 2

辞書 辞書 set list one 1

1 two  
2 2  
length: 2

辞書 辞書 set list two 2

1 three  
2 3  
length: 2

辞書 辞書 set list three 3

辞書 辞書 find two 2

1 two  
2 2  
length: 2

辞書 辞書 delete two

辞書 辞書 find two

キュー、スタック、辞書の使用例として「逆ポーランド記法計算」を示します。

[参考文献]

『プログラミング言語 AWK』 第2版 発行：オライリー・ジャパン  
Alfred V. Aho  
Brian W. Kernighan 著 千住 治郎 訳  
Peter J. Weinberger

一般的な「中置記法」による計算式  $(1 + 2) \times 3$  は「逆ポーランド記法」（「後置記法」）で記述すると、 $1\ 2\ +\ 3\ \times$  になります。カッコを必要としません。要素が数値の時はスタックに積み、演算子の時は必要な個数の数値をスタックから取り出して、演算の結果をスタックに積みます。この操作を式の終わりまで繰り返すと演算結果がスタック上に求められます。（キーボードには「 $\times$ 」がないので「\*」を使用します。）「変数=」でスタックから変数に、「変数」で値がスタックに置かれます。変数の管理は辞書を使用し、初期値として PI を登録しています。

逆ポーランド記法計算 list 1 2 + 3 \* 9

逆ポーランド記法計算 list 2 r= PI r 2 ^ \* 12.56

逆ポーランド記法計算 list 1 + 引数不足エラー

逆ポーランド記法計算 list 1 2 3 + 引数過多エラー

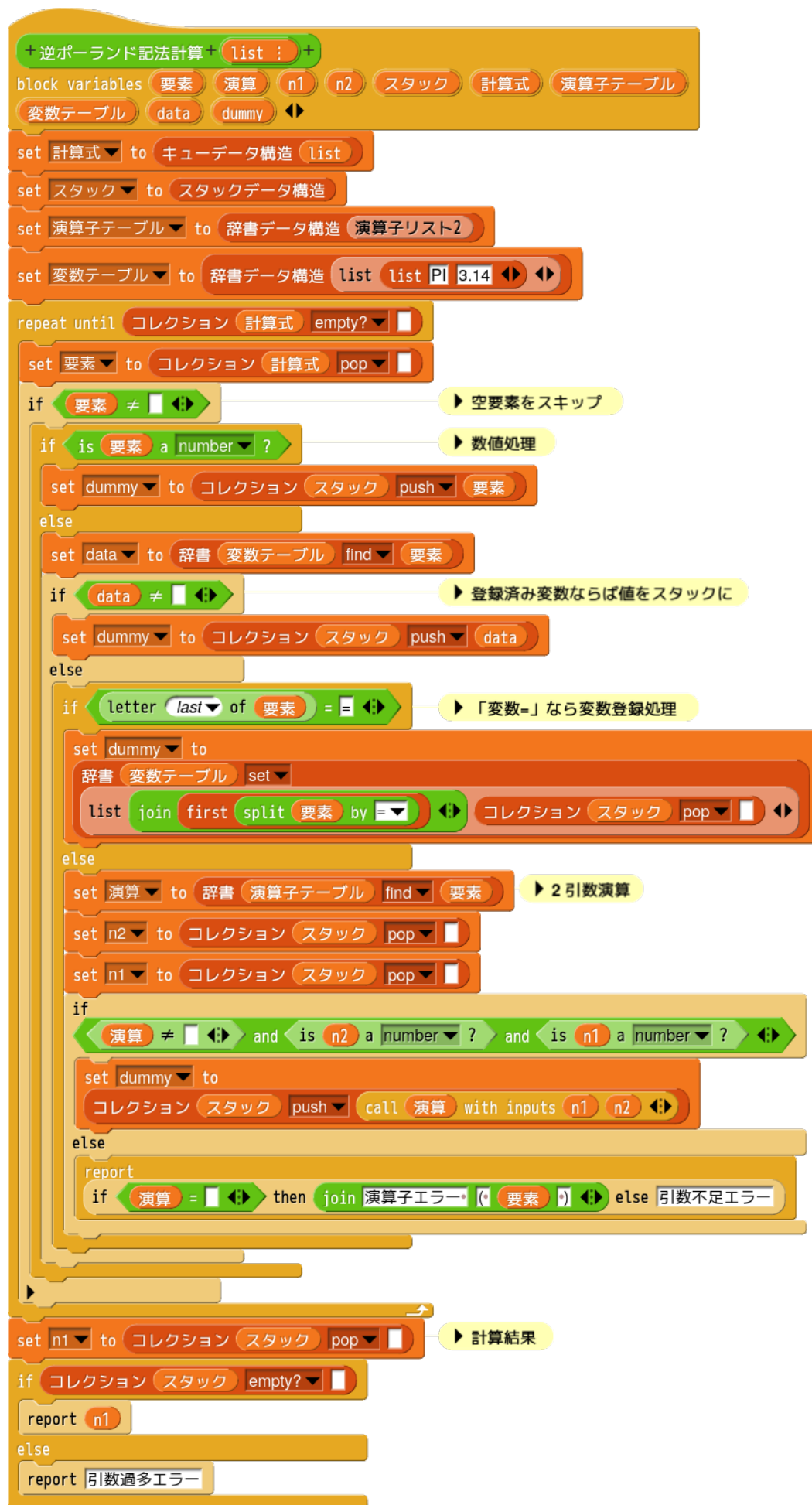
逆ポーランド記法計算 list 1 2 @ 演算子エラー (@)

演算子テーブル設定用の辞書リストです。指定できるのは入力スロットが二つのものだけです。演算子をキーとして検索するとそれを実行するための演算ブロックが得られます。

+演算子リスト2+

list	operator
list +	+
list -	-
list *	*
list /	/
list ^	^
list mod	mod
list min	min
list max	max

本体の定義です。紙面の都合で機能限定版です。後で示す foldl 版のような機能を持たせることもできます。



foldl を使用すると、その仕様をキューやスタックの機能として利用することができます。foldl に与えられたリストからはキューのように要素が供給されます。init は演算の結果で更新されるものですが、これをスタックのように使用して、数値や数値を取り出して得た演算結果を積んでいきます。

FORTH 言語の DUP、SWAP、OVER、ROT 機能を追加し、1 引数の演算もできるようにしています。ただし、細かいエラー処理はできていません。辞書リストにはスタック処理用の辞書、引数 1 用の辞書、引数 2 用の辞書、変数用の辞書のリストがセットされます。

```

+ 逆ポーランド記法計算foldl + list : +
block variables 辞書リスト
set 辞書リスト to
  辞書データ構造 演算子リスト0 辞書データ構造 演算子リスト1
  list 辞書データ構造 演算子リスト2 辞書データ構造 list list Pi 3.14
report
  pipe list →
    foldl func 目 辞書リスト list 目
    if length of 目 = 1 then first 目 else エラー
  
```

逆ポーランド記法計算foldl list 10 3 / floor 3

SWAP はスタックトップから一番目と二番目を入れ替えます。

逆ポーランド記法計算foldl list 2 3 SWAP - 1

DUP はスタックトップの値をコピーします。

逆ポーランド記法計算foldl list 2 DUP + 4

OVER はスタック二番目の値をコピーします。 3 4 コピー

逆ポーランド記法計算foldl list 3 4 over + \* 21

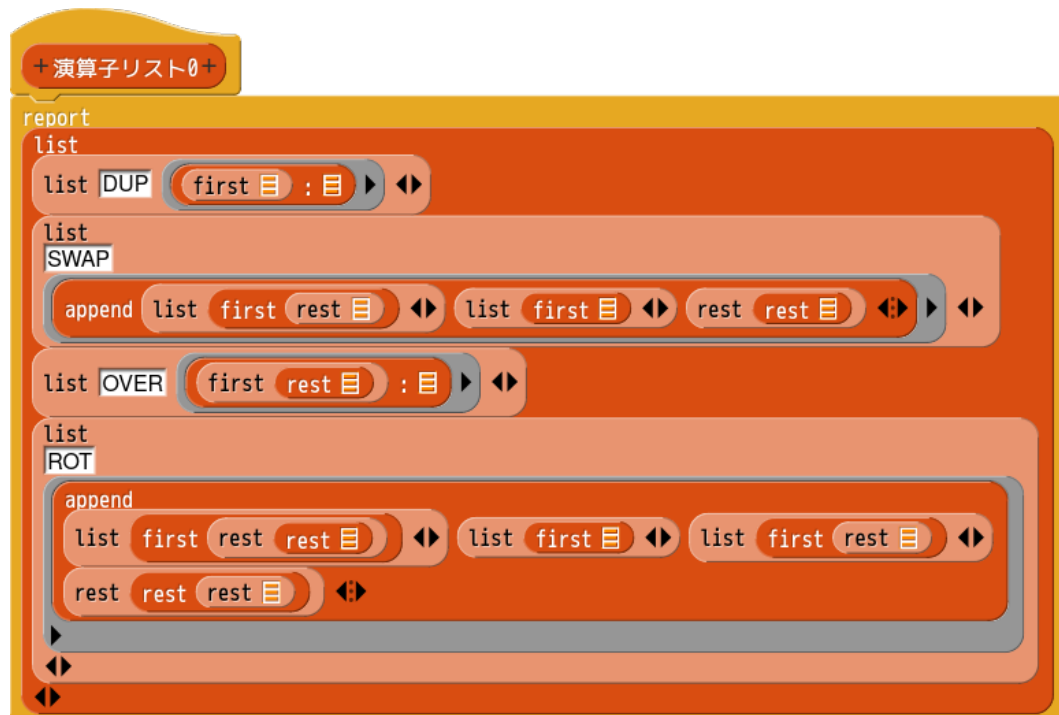
3 x 4 + 3 21 の意味を持ちます。

ROT はスタックトップ 3 の要素を循環して入れ替えます。 3 <- 4 <- 5 移動

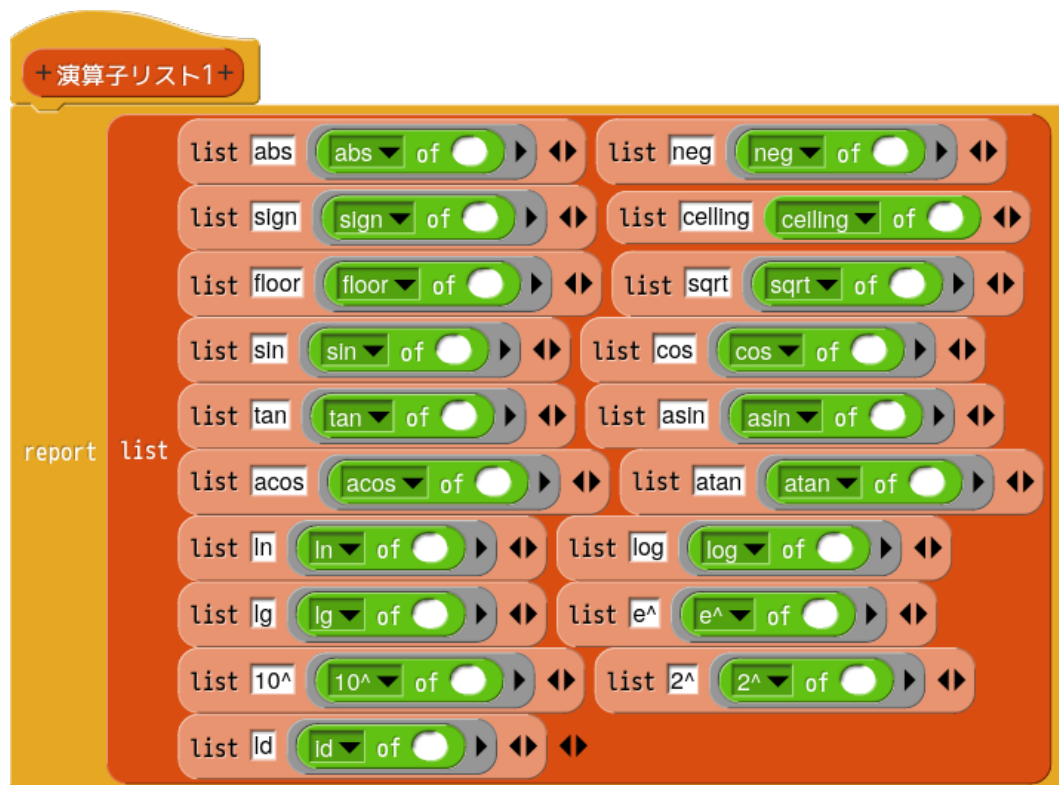
逆ポーランド記法計算foldl list 3 4 5 ROT - \* 8

4 x 5 - 3 8 の意味を持ちます。

スタック演算用辞書設定ブロックです。



1 引数演算用辞書設定ブロックです。

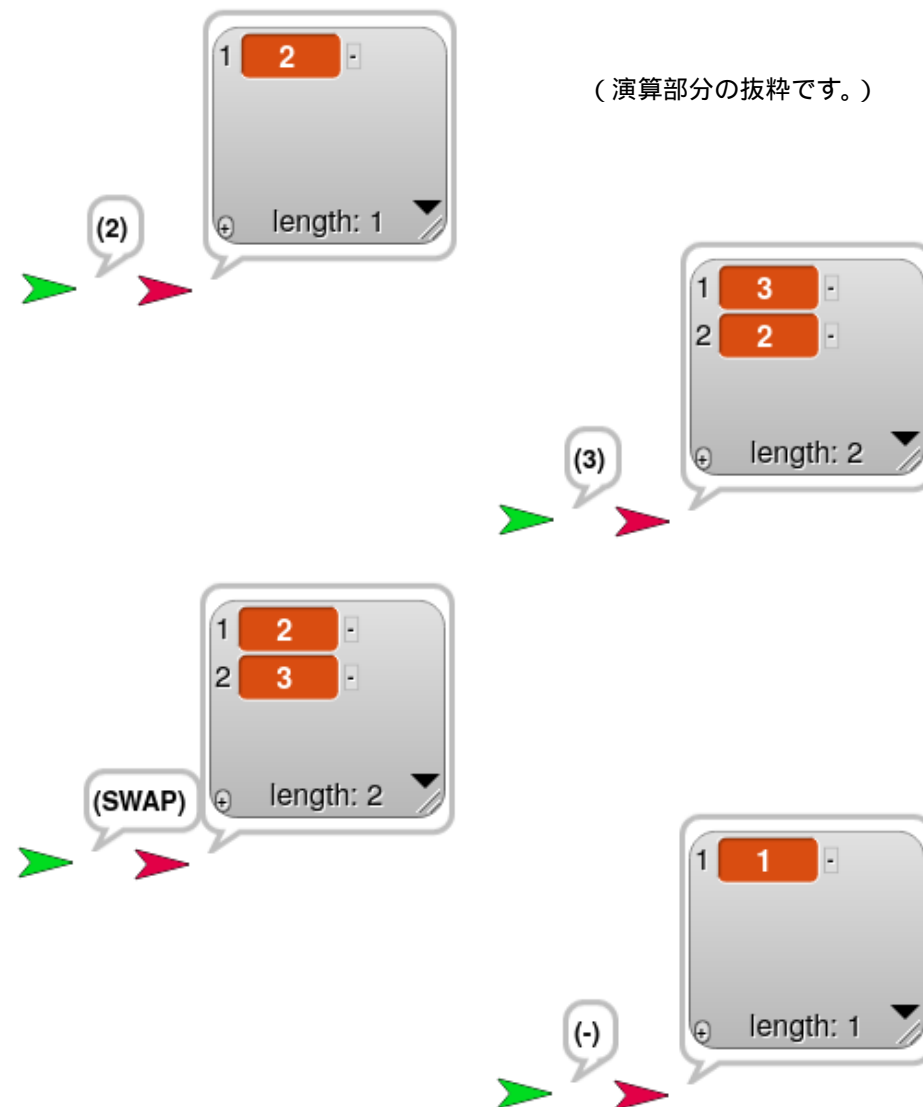
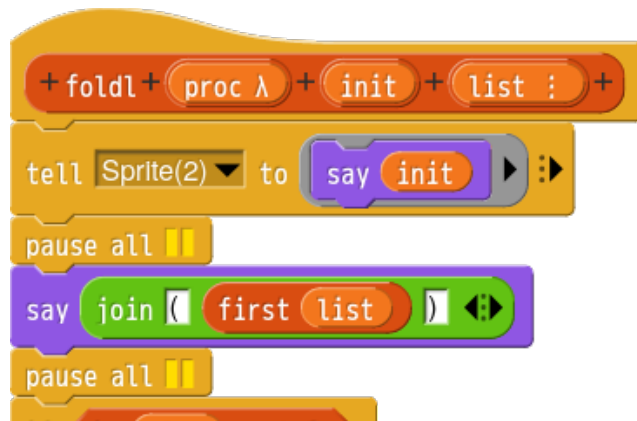


foldl に指定する計算処理を行うブロックです。





Sprite(2) を追加して、次のように foldl 定義に say を挿入すると、処理中の式の要素やスタックとして使用している init の様子が見られます。式の実行は Sprite 側から行います。



## 索引

: ブロック, 4

all, 16

collapse, 16

deep contains, 12

delete, 13

drop, 8

dropWhile, 26

factorial, 3

first ブロック, 4

identical, 15

initial slots, 16

keep, 9, 13, 17, 31

map, 14, 16, 30

multiple inputs, 16

qsort, 17

replace, 14

rest ブロック, 4

separator, 16

take, 8

takeWhile, 26

unfold, 34

unique, 9

階乗, 3

カーリー化, 36

逆ポーランド記法, 45

キュー, 38

クイックソート, 17

クロージャー, 36

高階関数, 24

再帰呼び出し, 3

辞書, 42

スタック, 41

素数, 9

多重再帰, 17, 20

ハノイの塔, 3

フィボナッチ数列, 20

部分適用, 36

末尾再帰, 22

リスト要素の巡回, 11, 27