

# Snap! のこと

齋藤文康

2025 年 10 月 2 日

Snap! Build Your Own Blocks (ver. 11) 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もあります。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

1 始め方	6
2 画面まわり	6
2.1 エリア . . . . .	7
2.1.1 ステージエリア . . . . .	7
2.1.2 スプライトコラル . . . . .	8
2.1.3 スクリプトエリア . . . . .	8
2.1.4 パレットエリア . . . . .	8
2.2 エリアの大きさ . . . . .	9
2.3 実行に関するボタン . . . . .	9
2.4 ブロック表示 . . . . .	12
3 キーボード入力	13
4 変数	14
4.1 for all sprites 全部のスプライトで使えるグローバル変数 . . . . .	14
4.2 for this sprite only スプライト変数 . . . . .	16
4.3 script variables スクリプト変数 . . . . .	17
4.4 Upvar 変数 . . . . .	18
4.5 リスト操作用ブロック . . . . .	18
4.5.1 numbers form ( ) to ( ) . . . . .	20
4.5.2 ( ) in front of ( ) . . . . .	21
4.5.3 all but first of ( ) . . . . .	21
4.5.4 index of ( ) in ( ) . . . . .	22
4.5.5 delete ( ) of ( ) . . . . .	22
4.5.6 append ( ) ( ) . . . . .	22
4.5.7 for each ( ) in ( ) . . . . .	22
4.5.8 reshape ( ) to ( ) ( ) . . . . .	23
4.5.9 map ( ) over ( ) . . . . .	24
4.5.10 keep items ( ) from ( ) . . . . .	26
4.5.11 find first item ( ) in ( ) . . . . .	27
4.5.12 combine ( ) using ( ) . . . . .	27
4.5.13 combinations ( ) ( ) . . . . .	28
4.6 オプションのリスト操作 . . . . .	30
4.6.1 rank of ( ), dimensions of ( ) . . . . .	30
4.6.2 flatten of ( ) . . . . .	31
4.6.3 columns of ( ) 転置行列 . . . . .	31
4.6.4 uniques of ( ) . . . . .	32
4.6.5 distribution of ( ) . . . . .	32

4.6.6	sorted of ( ) . . . . .	32
4.6.7	shuffled of ( ) . . . . .	33
4.6.8	reverse of ( ) . . . . .	33
4.6.9	$\Sigma$ of ( ) . . . . .	33
4.6.10	text of ( ) . . . . .	33
4.6.11	lines of ( ) . . . . .	34
4.6.12	csv of ( ) . . . . .	34
4.6.13	json of ( ) . . . . .	34
4.7	リストの演算 . . . . .	34
4.8	変数に入れられるもの . . . . .	36
<b>5</b>	<b>Control 制御</b>	<b>38</b>
5.1	リポーターの if < > then ( ) else ( ) . . . . .	38
5.2	stop ボタンがクリックされた時の終了処理 . . . . .	38
5.3	run ブロック . . . . .	39
5.4	call ブロック . . . . .	40
5.5	launch ブロック . . . . .	42
5.6	broadcast ブロック, tell to ブロック, request ブロック . . . . .	45
5.7	pipe . . . . .	47
<b>6</b>	<b>ブロックを作成する</b>	<b>48</b>
6.1	定数の作成 . . . . .	48
6.2	( ) ( ) ブロック . . . . .	50
6.3	help 説明文の作成 . . . . .	53
6.4	for (i) = (start) to (end) step (step) . . . . .	53
6.5	block variables . . . . .	66
<b>7</b>	<b>ブロック定義について</b>	<b>67</b>
7.1	プルダウン入力 . . . . .	67
7.2	Title Text とシンボル . . . . .	72
7.3	Input name オプションについて . . . . .	73
7.3.1	Reporter 型 . . . . .	74
7.3.2	Predicate 型 . . . . .	78
7.3.3	Command 型 . . . . .	79
<b>8</b>	<b>その他</b>	<b>83</b>
8.1	デバッグ . . . . .	83
8.2	getc . . . . .	85
8.3	= と identical . . . . .	87
8.4	Event Hat ブロック . . . . .	89
8.5	プリミティブブロックの編集 . . . . .	91

8.6	連想配列、辞書 . . . . .	93
8.7	read-only 入力スロット . . . . .	94
8.8	入力スロットへのリスト指定 . . . . .	95
8.9	入力スロットの追加 . . . . .	96
8.10	ask . . . . .	98
8.11	broadcast の検索オプション . . . . .	99
8.12	クローン . . . . .	99
8.12.1	テンポラリクローン . . . . .	99
8.12.2	パーマネントクローン . . . . .	103
8.13	flat line ends . . . . .	104
8.14	角度 . . . . .	107
8.15	anchor アンカー . . . . .	109
8.16	JavaScript function (オプション 6 ページ参照) . . . . .	111
8.17	時計 . . . . .	116
8.18	並列実行について . . . . .	120
8.19	行列の積 . . . . .	122
8.19.1	inner product . . . . .	122
8.19.2	outer product . . . . .	129
<b>9</b>	<b>Continuation 繙続</b>	<b>132</b>
9.1	run と call の with continuation バージョン . . . . .	132
9.2	catch と throw . . . . .	135
9.3	map からの脱出 . . . . .	135
9.4	break と continue . . . . .	137
9.5	thread . . . . .	139
<b>10</b>	<b>再帰呼び出し</b>	<b>143</b>
10.1	再帰呼び出しの例 . . . . .	143
10.1.1	階乗 . . . . .	143
10.1.2	ハノイの塔 . . . . .	143
10.2	再帰呼び出しの使用 . . . . .	144
10.2.1	繰り返し . . . . .	144
10.2.2	my length . . . . .	145
10.2.3	リストをリポート . . . . .	147
10.2.4	take と drop . . . . .	148
10.2.5	my contains . . . . .	149
10.2.6	my uniques . . . . .	149
10.2.7	my index of ( ) in ( ) . . . . .	150
10.2.8	リスト要素の巡回 . . . . .	151
10.2.9	指定の要素に対する delete と replace . . . . .	153

10.2.10 クイックソート（整列 / 並べ替え）	155
10.2.11 フィボナッチ数列	158
10.2.12 末尾再帰	160
<b>11 高階関数</b>	<b>162</b>
11.1 高階関数型ブロックの基本	162
11.2 高階関数型の take と drop	164
11.3 操作を指定するリストの巡回	165
11.4 foldl, foldr	167
11.5 unfold	173
11.6 concat	175
11.7 カリー化	177
11.8 ブロックの合成	179

## 1 始め方

Snap! のサイトは <https://snap.berkeley.edu/> です。Snap! も Scratch と同じように Web 上でプログラミングする方法と、オフライン版をダウンロードして使用する方法があります。オフライン版も Web ブラウザを利用するので、OS を問わずに使用することができます。

オフライン版は、Snap! のサイトの一番下にある Offline Version のリンクからオフライン版に関するページに移り、Simple Steps: の下の文中のダウンロードサイト

<https://github.com/jmoenig/Snap/releases/latest>

のリンクをクリックすると、オフライン版があるところにたどり着きますから、Source code(zip) か Source code(tar.gz) をダウンロードしてください。

ダウンロード後、展開し、その中にある snap.html を Web ブラウザで開いてください。

オフライン版はアップデートを自分でチェックする必要があります。また、コスチュームやライブラリーは Snap! のサイトからではなく、オフライン版のフォルダーにある Costumes、libraries から Import します。

オンライン版は、Run Snap! Now をクリックすれば使用できます。

日本語化もできますが、英語のままのほうがブロック表示がマニュアルやヘルプの表示と同じで対応がわかりやすいと思います。この文書では英語版のまま使用します。



日本語にするには、 のボタンをクリックすると表示される設定メニューの中で Language... をクリックして日本語を選べば変更することができます。  
また、Zoom blocks... をクリックすると、ブロックの大きさを変更することができます。  
JavaScript extensions がチェックされていると JavaScript ブロックが使用可能になります。

## 2 画面まわり

Snap! の画面にデスクトップやフォルダーからファイルをドロップすると、対応するファイル拡張子ならばそれに応じた処理をしてくれます。

- Snap! のプロジェクト (.xml) の場合は、プロジェクトとして開いてくれます。

- 画像ファイル (.png, .jpeg など) の場合は、その時点で対象になっているスプライトのコスチュームまたはステージの背景としてインポートし、ワードローブまたはバックグラウンドに入れります。
- サウンドファイル (.mp3 など) の場合は、その時点で対象になっているスプライトのサウンドとしてインポートし、ジュークボックスに入れます。
- テキストファイル (.txt) の場合は、変数を作成して読み込みます。
- .csv や .json のファイルは変数を作成し、リストとして読み込みます。

読み込むための変数が作成される場合は、拡張子を除いたファイル名が変数名になります。日本語のファイル名だと日本語の変数名になります。

英語版でも変数名やデータの内容など日本語が使えます。

## 2.1 エリア

各エリアには次のような名前がついています。



### 2.1.1 ステージエリア

ここにはスプライトが動く様子や pen で描いた軌跡などが表示されます。変数の値をリポートする変数ウォッチャーも表示されます。このエリアにマウスポインターを合わせ、右クリックすると次のメニューが出ます。



- edit は、ステージ用のスクリプトの作成です。操作対象を Stage にします。

- show all は、非表示設定になっているものも含めてスプライトを全部表示します。ステージ外に行ってしまったものもステージ内に連れ戻します。変数ウォッチャーも表示されます。不要な変数ウォッチャーは、パレットエリアにドロップしてください。
- pic... は、ステージのスクリーンショットを撮ります。画像はダウンロードフォルダーへ。
- pen trails は、pen や stamp で描いた軌跡を選択されているスプライトのためのコスチュームとしてワードローブに、またはステージのための背景としてバックグラウンドに追加します。このコスチュームの中心は、pen trails された時点の座標になります。

### 2.1.2 スプライトコラル



ここには使用するスプライトやステージが表示されます。 をクリックすると新しいスプライトを生成できます。すでにあるスプライトを右クリックして出てくるメニューからコピーやクローンを作ることもできます。

### 2.1.3 スクリプトエリア

スクリプトエリアは、スクリプトを作成する場所です。ただし、スクリプトエリアの部分はスクリプトを扱う時はスクリプトエリアで、コスチュームを扱う時はワードローブエリア、サウンドを扱う時はジュークボックスと呼び方が変わります。また、ステージの時はワードローブではなくバックグラウンドです。

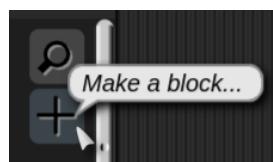
スクリプトエリアの右上には、 と があります。

ブロックを組んでいて、いらないと思ってパレットエリアに移してしまったものが必要だった場合は、 をクリックすれば元に戻せます。これを使うと になり、その変更を元に戻すことができます。

は、キーボードを使ってスクリプトを作成するモードへのスイッチです。

### 2.1.4 パレットエリア

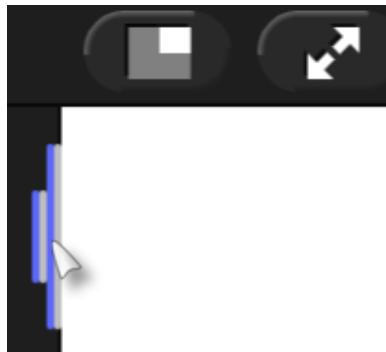
ここからスクリプト作成のためのブロックを持ってきます。要らなくなったブロックを戻す場所もあります。上部にある 8 個のボタンから選択して、使用する機能のブロックを表示します。



のボタンはカスタムブロックを作成する時に使います。カスタムブロック(ユーザー定義ブロック)とは、ブロックエディターで作成、修正可能なブロックです。それに対して、プリミティブブロックは、Snap! に備わっているブロックです。

その上のボタンはブロック検索用です。クリックすると検索窓がでます。アルファベットを入力すると該当するブロックが表示されます。

## 2.2 エリアの大きさ



の部分でステージの大きさ、結果的にスクリプトエリアの大きさを変えることができます。 のボタンのクリックで変わります。 のボタンは画面の表示をステージのみにして発表モードにするものです。左図のマウスポインターが置かれて色が薄い紫色になっているところをドラッグしても大きさを変えることができます。



左図のマウスポインターが置かれて色が薄い紫色になっているところをドラッグすると、パレットエリアの大きさを変えて横幅のあるブロックの全体を表示させることができます。

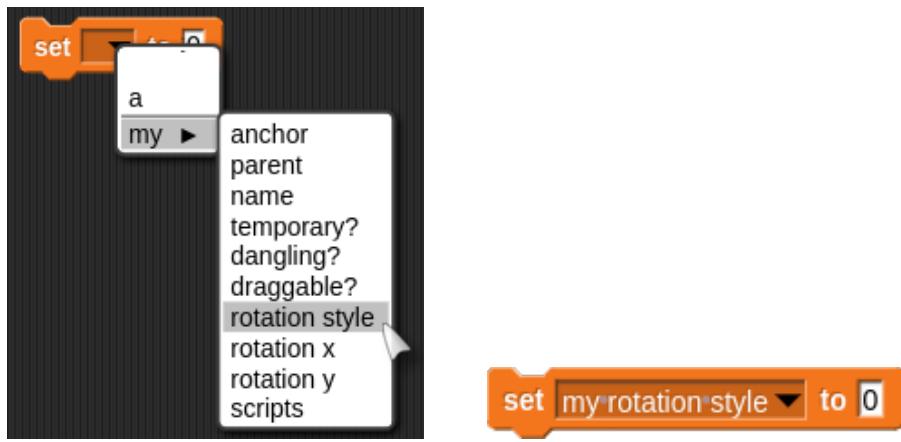
## 2.3 実行に関するボタン

ステージエリアの上には のボタンがあります。これは に接続されたスクリプトを実行するボタンです。 はスクリプトの実行を終了させるボタンです。 はスクリプトの実行を一時停止させるボタンです。 のボタンをクリックすると実行中のブロックをハイライトさせてゆっくり実行させたりできます。 のマウスポインターが置かれているところをドラッグすると実行のスピードが調整できます。左端だと 1 動作ごとのステップ実行になります。デバッグの時に使えます。(83 ページ参照) デバッグモードでは、実行中のブロックをハイライトしたり、ブロック中の変数やリストの値を表示してくれます。 のクリックで実行再開です。スクリプトを止めて確認したいところに を入れても、そこで一時停止させることができます。

スクリプトエリアの上に次のようなボタンがあります。



これは、スプライトの回転を可能にするかを設定します。1番上は、可能(1)。真ん中は、左右のみ(2)。1番下は、回転不可(0)。set ブロックを使って rotation style に()の中の数値(0, 1, 2)を入れてやると、スクリプト上で設定することができます。



スプライトをマウスでドラッグできるかを設定するのが、 drammable です。スクリプトで設定するのが、 です。こちらは、true か false で設定します。

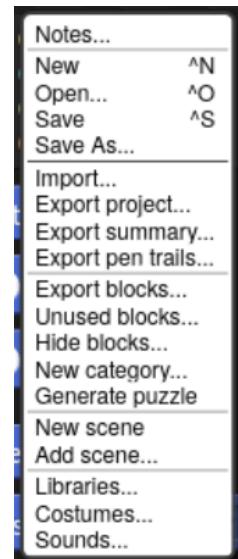
 をクリックすると右のメニューが出てきます。

Notes... にはプロジェクトの覚書、注釈が書けます。

New で新しいプロジェクトの開始、Open... で保存してあるプロジェクトの読み込み、Save と Save As... でプロジェクトの保存です。

scene は、プロジェクト内にサブプロジェクトを置くものです。新規作成または既存のプロジェクトをオープンしてサブプロジェクトとして加えます。 で次のプロジェクトに移行することができます。

Libraries... でライブラリーからいろいろなブロックを取り込むことができます。 Costumes... でコスチュームを取り込むことができます。



必要なコスチュームを Import し終えたなら Cancel をクリックします。

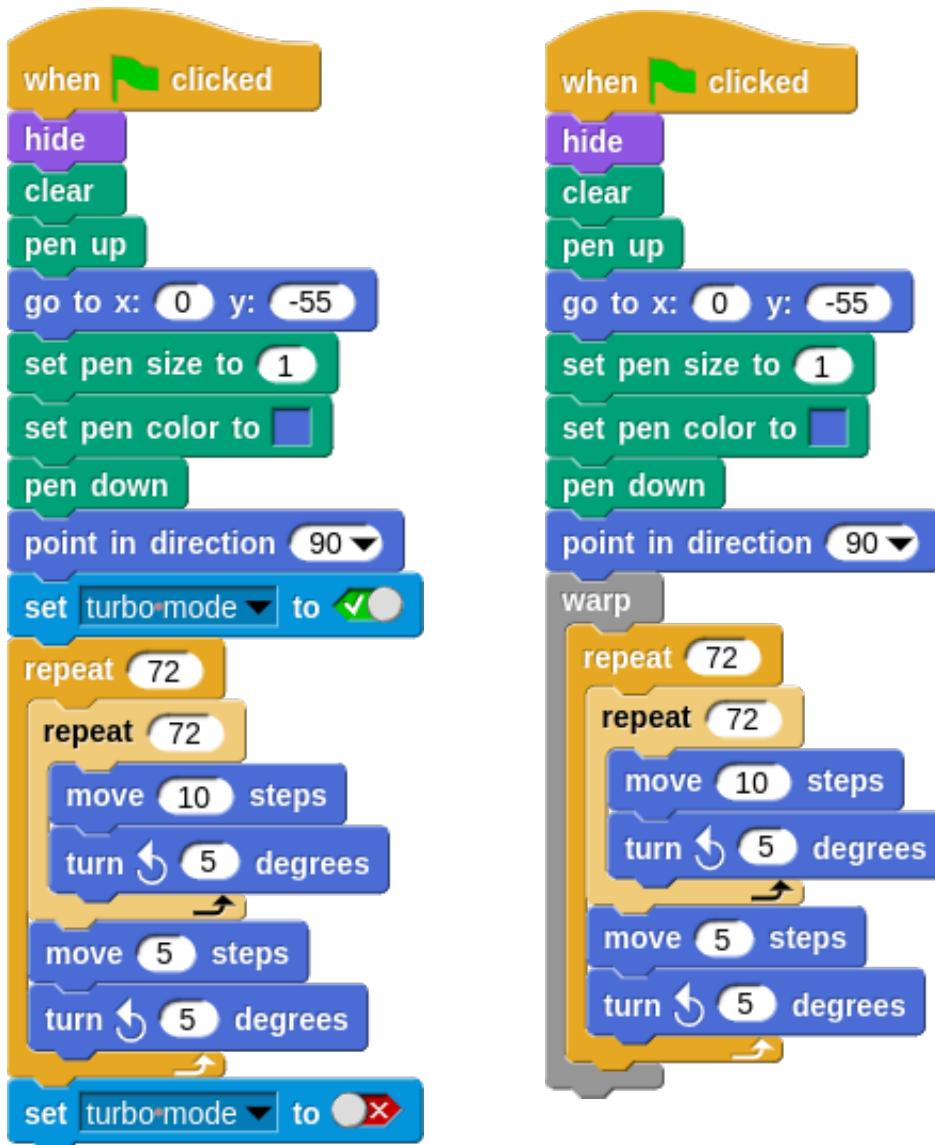
Scratch で高速に実行するためのターボモードが Snap! にもあります。

[Shift] を押しながら  のボタンをクリックすると  (ターボモード) になり、これをクリックすると描画のスピードが速くなります。

Sensing パレットに  ブロックがあります。これを turbo mode にして六角形の部分をクリックすることで、 ターボモードをオンにしたり  オフにすることができます。スクリプト内で自在にターボモードの切り替えができます。

ワープブロックでスクリプトを囲むと、その処理に専念するためにとても速く処理することができますが、処理できる量にも限界があるようでスムーズにいかないこともあります。

描画のスピードを比較するために、ターボモードで実行する場合とワープを使用した場合のスクリプトを示します。



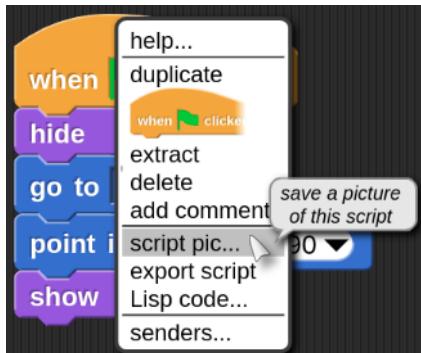
## 2.4 ブロック表示

$$y = \frac{1000}{\sqrt{\sqrt{(x - 50)^2 + (z + 50)^2} + 100}} - \frac{1000}{\sqrt{\sqrt{(x + 50)^2 + (z - 50)^2} + 100}}$$

このような長い式を Snap! でスクリプトにすると、

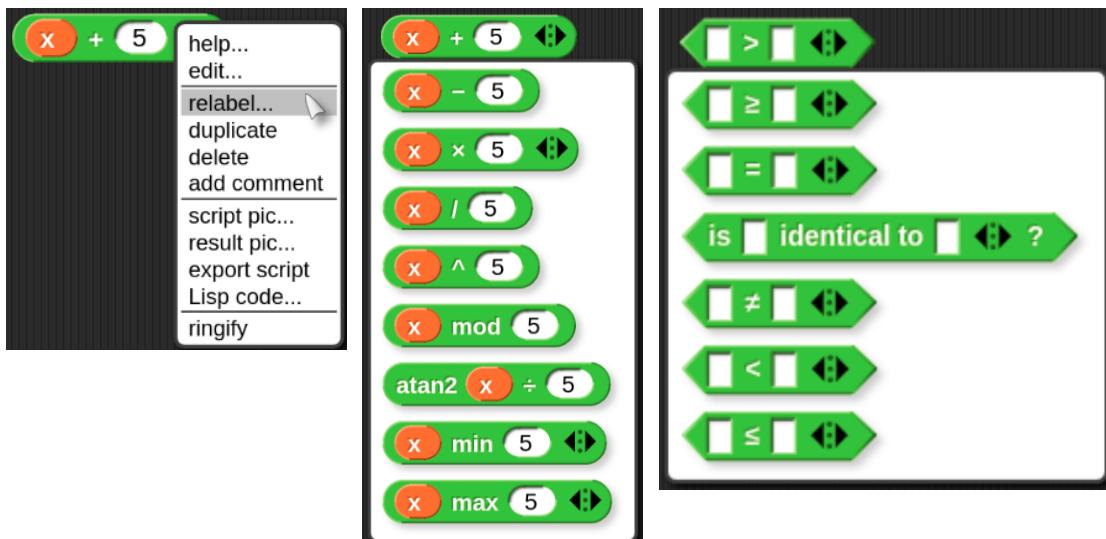


となります。長くなった場合は、自動的に折り畳まれます。また、同じパレットのブロックが重なった場合は、色合いが交互に変化して表示されます。ゼブラカラーリングだそうです。



スクリプトをプリンターで印刷したい場合は、スクリプトを右クリックすると、script pic ... で、画像ファイルとしてダウンロードフォルダーへエクスポートされます。  
ファイル名はプロジェクト名 + script pic + (番号) + 画像ファイルを表す拡張子になります。

ブロックを組んでいて、間違えたとか違う種類の方にしたい時があります。そういう時はそのブロックを右クリックすると出てくるメニューから、relabel... をクリックすると欲しいものが得られる場合があります。パレットエリアには無いものも使用できます。



### 3 キー ボード 入力

スクリプトエリアには  があります。これをクリックするか、Shift + ⌘ でキー ボード を 

使ってスクリプトを作成できるモードになります。 をクリックしてください。すると、スクリプトエリアに白い線が点滅されます。すでにいくつかスクリプトがある場合は、Tab を押すと別のスクリプトのところへ移動します。

二つの計算結果を表示するスクリプトを作ってみます。



スクリプトが何もない状態から始めます。

 を出すために、「when」の最初の文字  を打ちます。すると、パレットエリアに左図のように表示されるので、 と  で選んでください。

次に、 です。「say」から   と打つと欲しいものが得られます。スクリプトエリアにセットされると、入力スロットの部分が白く(ハロ)囲われています。



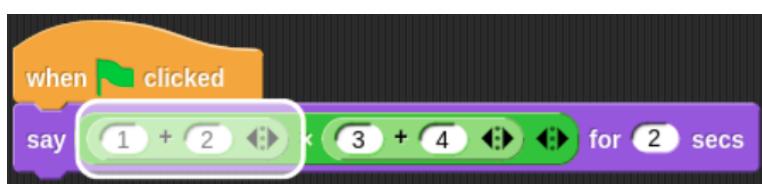
ここで  を押すと、ハロが消えて初期値としての「Hello!」を修正または別なテキストを入力することができます。  を押してハロが出ている状態で文字を打ち込むと、ブロックや変数を候補として出してくれます。数字や「( ) + - \* / < = >」を打ち込むと、数式の入力ができます。

「(1+2)\*(3+4)」と入れてみてください。パレットエリアに入力に応じて式のブロックが表示されます。  で決定です。

16/4/2



16/(4/2)

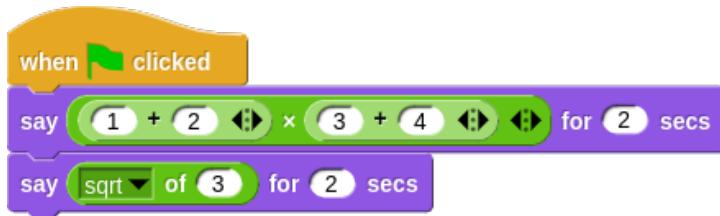


  で入力スロットを移動して変更できます。  で次のブロックに移ります。

次に、ルート 3 の値を表示させてみます。ルートは sqrt で求めます。

また、say Hello! for 2 secs を出してください。

say の最初の入力スロットの部分で、□(of の「o」です)と打ちこんで sqrt □ of 10 を選びます。この場合は sqrt になっているのでこのままでいいですが、別な演算を選ぶ場合は □ を押すと選択肢が表示されます。後は 3 をセットすれば終了です。できたスクリプトは [Control]+[Shift]+□ で実行できます。



どこかをクリックするか [Esc] などでキーボード入力モードから抜けます。

## 4 変数

変数を作成するにはいくつか方法があります。

- Make variables をクリックする。
- script variables □ を利用する。
- block variables を利用する。( ブロック内変数 66 ページ参照 )
- for ブロックなどの変数を利用する。
- .txt .csv .json などのファイルを Snap! 画面にドロップする。

英語版のままでも変数名に日本語も使えますし、値として日本語を扱うこともできます。次のように半角全角の混じった文字列でも正しく処理されるようです。



### 4.1 for all sprites 全部のスプライトで使えるグローバル変数

変数は、有効になる(変数が見える)範囲によって種類が分かれます。パレットエリアにある



[ Make a variable ] をクリックすると、



が出ますから for all sprites を選んで、varAll



という変数を作ります。そうすると、パレットエリアに  
うして作られた変数は全部のスプライトで使用できます。このようにどこからでも使用できる変数  
をグローバル変数とか大域変数と言います。

左のチェックボックスにチェックを入れると変数の値を表示する変数ウォッチャーがステージエリ  
アに表示されます。



変数を削除する場合は、**varAll** で Delete a variable をクリックすると登録さ  
れている変数が表示されるので、そこから選んで削除します。

名前を変更する場合はその変数のところを右クリックして、



します。all の方を選ぶと、この変数を使用しているすべての個所を変更します。



Make a variable で作成した変数は、右クリックで  
transient のオプションが表示されます。これはプロジェクトの保存の時にこの変数の値を保存さ  
せないためのものです。以下は、transient の効果を示すものです。

スクリプトが何もない状態で、変数 var を for all sprites を選んで作成します。この段階では var は、 0 です。



これを実行させない状態で save します。すると、ファイルの容量は 8.5KB でした。これを実行すると、変数 var に 10000 個の要素のリストが入ります。

この状態で save します。すると、ファイルの容量は 59.8KB でした。この保存したプロジェクトを改めて Open... で読み込むと、保存された時の変数の値になっています。つまり、何もしないと変数が値ごと保存されてプロジェクトファイルの容量が大きくなるということです。transient をチェックして save すると、ファイルの容量は 10.9KB でした。この数値は実行環境など状況によって違うかもしれません。この保存したプロジェクトを改めて Open... で読み込むと、変数 var の値は初期値 0 になっています。これが transient の機能です。このスクリプトの場合、プログラム開始時に 10000 個の要素を持つリストが作成されるのでプロジェクトファイルに値を含ませるのは無駄なことです。transient をうまく使うとプロジェクトファイルをスリムにすることができます。

グローバル変数はどこからでも使って便利なのですが、あるスプライトが使用中の変数を別なスプライトによってかってに変更されてしまうと具合が悪いです。ある範囲の中だけで有効なローカル変数というものがあります。ローカル変数にはその範囲によって種類があります。

## 4.2 for this sprite only スプライト変数



〔 Make a variable 〕 をクリックして、



for this sprite only を選んで varSprite を作



成します。すると、パレットエリアに varSprite が表示されます。varSprite という名前

の左にロケーションピンアイコンが表示されて、これがこのスプライト専用の変数であることを表します。この変数はこのスプライトのスクリプトエリア内ならばどのスクリプトからも使用することができます。このスプライト限定になりますがカスタムブロック内で使用することもできます。



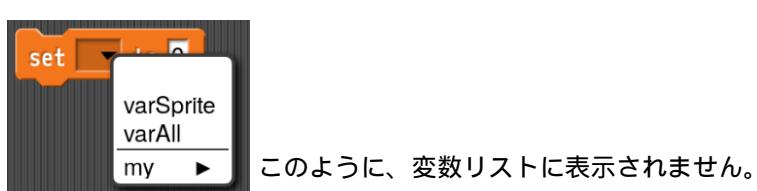
#### 4.3 script variables スクリプト変数

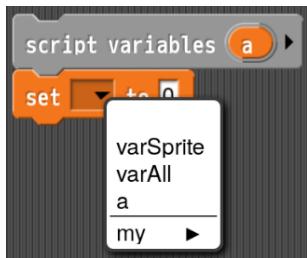
script variables a で、その下に接続された範囲だけで有効なスクリプト変数が使えるようになります。変数名は a のところをクリックすると変えられます。



左向きの三角をクリックすると削除できます。

スクリプト変数は script variables a をスクリプトエリアに持ってくるだけでは使えません。



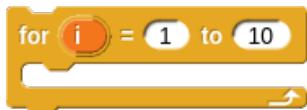


このように接続されてからでないと使えません。



このようにその前でも有効になりません。

#### 4.4 Upvar 変数

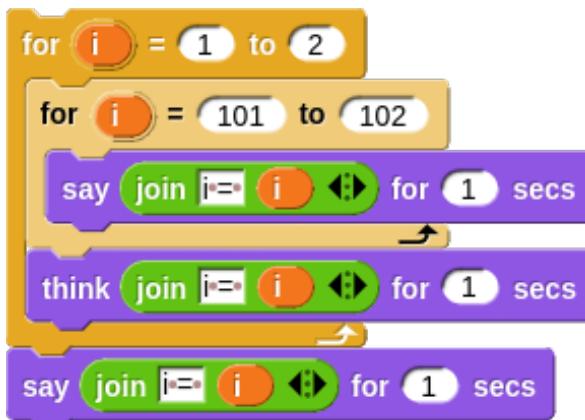


for ループのようにあらかじめ用意されている変数があります。

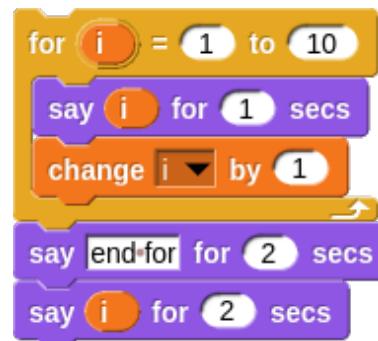
これはスクリプト変数のように変数名を変更することができます。また、この変数をいくつでもドラッグ&ドロップして使用することができます。for ループ用の変数は、ループする間に 1 ずつ増加または減少していくので、その変化していく変数の値をスクリプトで使用するということですが。



この例は、i の値を 3 から 1 に 1 ずつ減らしながら C 型ブロック内のスクリプトを実行します。3, 2, 1 と表示します。



ループする回数自体は合っていますが、thing ブロックは外側ではなく内側の i の値を表示します。変数は適切に使用しましょう。



このようにすれば増減を操作することができます。ループ変数はループ外でも参照できるようです。

#### 4.5 リスト操作用ブロック

Snap! には Scratch のようなリスト専用の変数はありません。変数に数値や文字列を入れるよう、リストを入れればリストを記憶する変数になります。



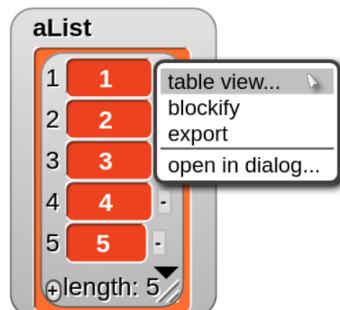
**aList** で **aList** という変数を作成すると、ステージエリアに **aList 0** が表示されます。 **list [ ]** の左向きの三角をクリックして **set aList to [list]** とすると、空のリストができます。 **list [ ]** の右向きの三角をクリックして **set aList to [list 1 2 3]** で値を入れてやると(左端の入力スロットに 1 を入れて、**Tab** キーを押すと次の入力スロットに移



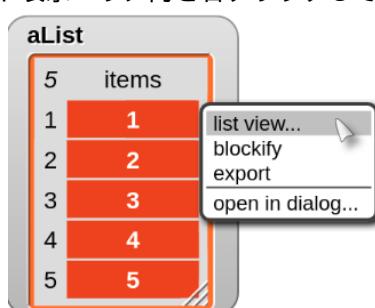
れます)、**aList** は要素を持つリストになり、ステージエリアに



**set aList to [numbers from 1 to 5]** を使うと、  
**set aList to [list]**  
**for [i = 1 to 5]**  
**add [i] to [aList]** で同じことができます。



変数ウォッチャーのリスト表示エリア内を右クリックして



**table view...** を選択すると  
 クリックして **list view...** を選択すると元に戻ります。

に変更することができます。また右



list view 内では、要素をクリックすると値を変更することができます。左下の プラスマークをクリックすると要素を増やせますし、要素の右の マイナスマークをクリックするとその要素を削除できます。



変数ウォッチャーの内部を右クリックして、blockify をクリックすると、リストブロックとして取り出すことができます。



open in dialog... をクリックすると、ステージ外にリストのTable view を表示することができます。

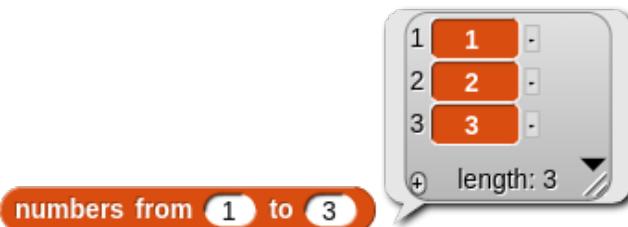
**export** をクリックすると、変数の内容をテキストファイルとして保存できます。リストの場合は .csv 形式、複雑なリストの場合は .json 形式のテキストファイルとして書き出されます。

変数ブロックや演算ブロックのような橿円形のブロックは、リポーター ブロックと言ってクリックするかスクリプト内で使用されると値を返して（リポートして）くれます。



#### 4.5.1 numbers form ( ) to ( )

これは指定された範囲のリストをリポートするものです。



指定された範囲内で 1 ずつ増加するリストをリポート



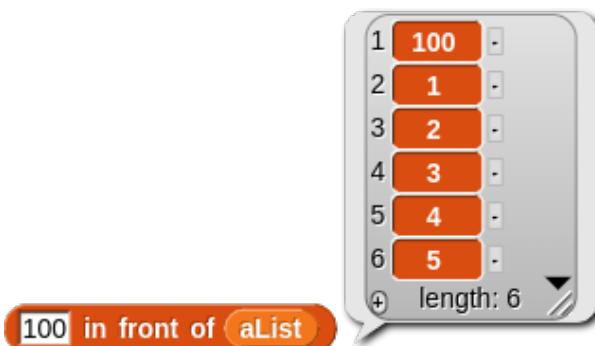
指定された範囲内で 1 ずつ減少するリストをリポート



1 ずつの増加または減少なので指定された値が含まれないこともあります。



#### 4.5.2 ( ) in front of ( )



これは、指定された値を指定されたリストの先頭に挿入したリストをリポートします。指定されたリスト自体は変更しません。

#### 4.5.3 all but first of ( )



これは、指定されたリストの先頭を除いたリストをリポートします。指定されたリスト自体は変更しません。

#### 4.5.4 index of ( ) in ( )

これは、指定された要素がリストの中に存在するか調べて、もしあればそのインデックスをリポートします。もしなかったら 0 をリポートします。



#### 4.5.5 delete ( ) of ( )

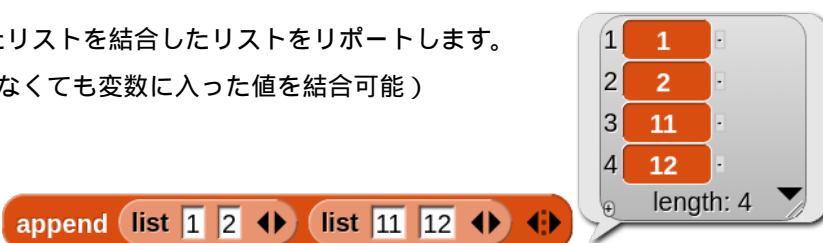
これは、指定されたリストから指定された位置の要素を削除します。



位置が要素数よりも大きい場合は何もしません。ただし、位置を 0 とすると last を指定したのと同じになってしまいます。-1 にすると、最後から -1 番目、-2 にすると、最後から -2 番目ということになります。要素数 -1 以上の値を負数にした値、要素数 5 の aList では -4 以下の数値を指定すると、1 番目の要素を削除することになります。

#### 4.5.6 append ( ) ( )

これは、指定されたリストを結合したリストをリポートします。  
(ver.11 からリストでなくとも変数に入った値を結合可能)



変数などにリストをセットする場合、  
"set [bList v] to [aList]" とすると bList は aList と同じリストを参照します。そのため、

"replace item [2] of [aList] with [x]" のようなことを行うと aList, bList 共に変更されます。

"set [bList v] to [append [aList]]" のようにすると、bList には aList のコピーがセットされるので変更されません。

#### 4.5.7 for each ( ) in ( )

実行してみると動作が分かりますが、リストの各要素を item に入れながら全要素分指定されたスクリプトを実行します。右が、for ループで同じことをするものです。



#### 4.5.8 reshape( ) to( )( )

リストの要素にリストを含むものを配列、縦横二次元的なものを行列と言います。

4	A	B	C
1	1	2	3
2	4	5	6
3	7	8	9
4	10	11	12

list [list [1 2 3] ←] [list [4 5 6] ←] [list [7 8 9] ←] [list [10 11 12] ←] ←

reshape ブロックに、要素を指定するリストと行数と列数を指定することで希望の形の配列を作成することができます。ただし、指定した行数 × 列数個以上のリストの要素は無視されます。

4	A	B	C
1	1	2	3
2	4	5	6
3	7	8	9
4	10	11	12

reshape [numbers from 1 to 20 to 4 3] ←

指定したリストの要素数が行数 × 列数よりも少ない場合はリストの先頭に戻ってその値が使用されます。このことを利用すると、値が 0 ( または他の値 ) の配列を作成することができます。

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

reshape [list 0 ← to 4 3] ←

1	0
2	0
3	0
4	0
5	0

+ length: 5

reshape [list 0 ← to 5] ←

例えば、1 行目の 3 列目の値を設定したり読み出したりするには次のようにします。

set aList to reshape [list 0 ← to 4 3] ←

replace item 3 of item 1 of aList with 99

#### 4.5.9 map ( ) over ( )



これは、指定されたリストの各要素に対して指定された操作を行ったリストをリポートします。

1	11	.
2	12	.
3	13	.
+	length: 3	▼



「+」の左側の入力スロットが空になっています。ここにリストの要素が順番に入って、計算された結果をリストとしてリポートします。+ 演算なので **10 +** でも同じです。

入れ子になったリストも map を入れ子にすることで対応できます。

2	A	B	C
1	2	3	
2	4	5	6



map の入力スロットの灰色の部分をリングと言います。右端の右向きの三角をクリックすると、フォーマルパラメータが出てきます。パラメータとは、値を受け取るための変数のようなものです。map には機能が設定された 3 つのフォーマルパラメータがあります。

value には指定されたリストの要素が順番にセットされます。index はリストの何番目の要素を処理しているかを示します。list は指定されたリストを表します。この場合 list は aList を表します。value = item (index) of (list) の関係になっています。



次のようにすると特定の値の要素を入れ替えることができます。

1	1	.
2	tow	.
3	3	.
+	length: 3	▼



map に変数を指定する場合、フォーマルパラメータの list は変数のコピーではなくそのものを指しているので、list の値を変更すると変数の値もそのように変更されます。非推奨ですが、このことを利用するとフィボナッチ数列を求めることができます。

1	1	.
2	1	.
3	1	.
4	1	.
5	1	.
6	1	.
7	1	.
+	length: 7	▼

フィボナッチ数列 ( 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ... ) は、1 項目と 2 項目は 1 で、第  $n$  項は第  $n - 1$  項 + 第  $n - 2$  項の値になります。

まずは簡単にするために aList の各要素を 1 にします。



1	1	-
2	1	-
3	2	-
4	3	-
5	5	-
6	8	-
7	13	-
+	length: 7	▼

次のようにして list の要素に対して値をセットしています。そうしないと、(n - 1) 項と (n - 2) 項の値を得られないからです。

```

map
if <index> > 2 then
  replace item <index> of <list> with
    item <index - 1> of <list> + item <index - 2> of <list>
  report item <index> of <list>
else value
input names: <value> <index> <list>
over <aList>

```

1	1	-
2	1	-
3	2	-
4	3	-
5	5	-
6	8	-
7	13	-
+	length: 7	▼

その結果、aList の要素の値も入れ替わります。  
( 入力したリストが変更されることを示すために変数を使いましたが、`reshape [1 to 7]` を使用することもできます。 )

aList

次に示すようにリスト操作を行うには map ブロックを使うほうが速くできます。

```

reset timer
set [aList v] to [list]
for each [item] in [numbers from 1 to 1000]
  add [item + 10] to [aList]
report [timer v]
0

reset timer
set [aList v] to [map [+ 10] over [numbers from 1 to 1000]]
report [timer v]
0

```

#### 4.5.10 keep items ( ) from ( )



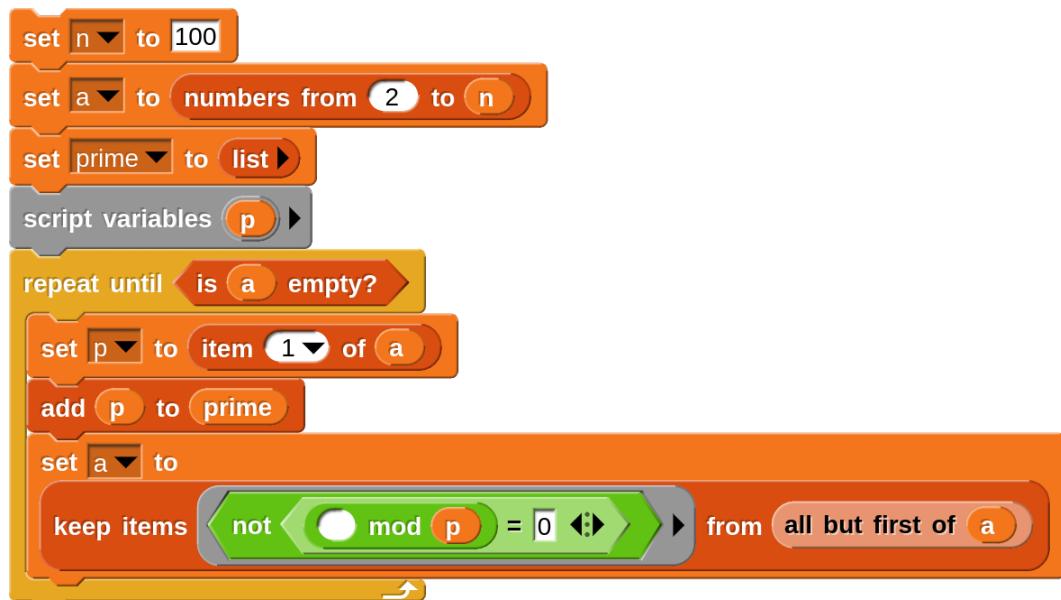
これは、指定されたリストの要素から指定された条件に合った値のリストをリポートします。指定されたリスト自体は変更しません。

**delete [1 v] of [aList]** のブロックでリスト内の位置を指定して要素を削除できますが、次のように **not** を使用すると、指定した条件のものを削除したリストが得られます。



これを応用すると「エラトステネスのふるい」のアルゴリズムを用いて素数列を求めることができます。素数とは 1 より大きな整数で、約数が 1 と自分自身のみである整数です。

a に整数のリストをセットします。先頭の要素は素数なので素数リストに加え、keep でリストから倍数を排除したものを a のリストとする、ということを a のリストが空になるまで繰り返します。(149 ページ参照)



2000 年から 2200 年までのうるう年のリストを求めてみます。

4 で割り切れて   
かつ 100 では割り切れない年   
または、400 で割り切れる年  ということで、2000 年はうるう年だけれども 2100 年と 2200 年は違います。



空の入力スロットが複数ヶ所あります。2000 年の場合、すべてにリストの要素である 2000 が入ります。フォーマルパラメータ value をすべてにセットするのと同じです。

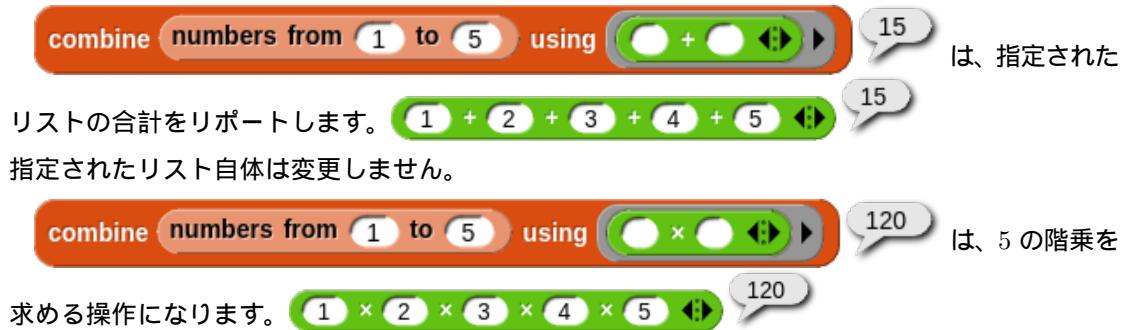
#### 4.5.11 find first item ( ) in ( )



これは、指定されたリストの要素から指定された条件に合った最初の値をリポートします。指定の値がなかったら、 (空) をリポートします。指定されたリスト自体は変更しません。

keep ブロックと同じような操作ですが、keep が条件に合うすべての値のリストをリポートするのに対してこちらは最初の一つの値をリポートするだけです。

#### 4.5.12 combine ( ) using ( )



combine では、以下の演算がおもに使われるようです。

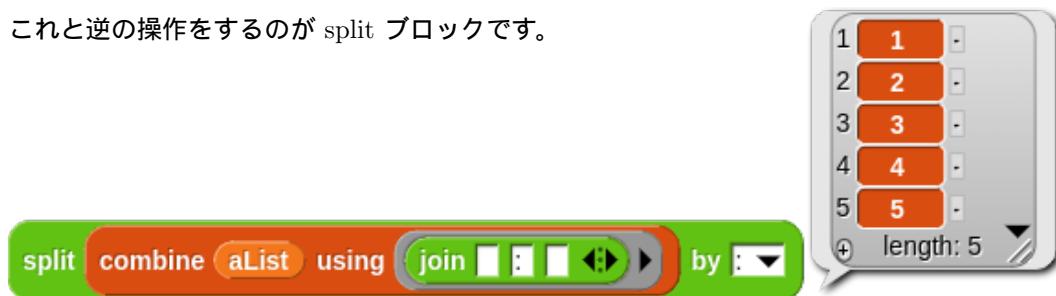


95 ページで述べているように、演算ブロックの右端の三角のところにリストを入れてやることでもリスト全要素に対しての演算の値を得ることができます。

join を指定すると文字列の連結になります。( 入力スロットは空です。 )



これと逆の操作をするのが split ブロックです。



リスト要素の最大値や最小値を求めます。



#### 4.5.13 combinations ( ) ( )

指定されたリスト同士を順番に組み合わせたリストをリポートします。



x 座標、y 座標 それぞれ取り得る値が ( -100, 0, 100 ) の場合、すべての組み合わせからランダムに選んだ座標へ移動するスクリプトです。



なお、go to ブロックは `go to [random position v]` の入力スロットに `item [1 v of ]` ブロックをドロップしたものです。

combinations の動作を for ブロックで行うと次のようにになります。

```

set aList to list
for (x) = 1 to 3
    for (y) = 1 to 3
        add list x y to aList

```

The script uses two nested loops to generate all combinations of elements from two lists. The resulting list 'aList' contains 9 elements:

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

combinations list [1 2 3] list [1 2 3]

組み合わせの順番に意味がある場合もありますが、同じ組み合わせは排除したい時もあります。

```

set aList to list
for (x) = 1 to 3
    for (y) = x to 3
        add list x y to aList

```

This script is similar to the previous one but includes a condition where y starts from x, effectively removing duplicate pairs like (1,1), (2,2), and (3,3). The resulting list 'aList' contains 6 elements:

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

reshape  
map  
map list x y input names: y over numbers from x to 3  
input names: x  
over numbers from 1 to 3  
to 2

map ブロックで作成したリストを reshape で 2 列のリストに整えています。左側の空の入力スロットにより行数はあまかせになります。

リストの要素を整列させる sorted of ブロック (32 ページ参照) や リスト内の同じ要素を排除する uniques of ブロック (32 ページ参照) を使って次のようにすることもできます。

```

sorted of
uniques of
map sorted of over combinations list [1 2 3] list [1 2 3]

```

## 4.6 オプションのリスト操作

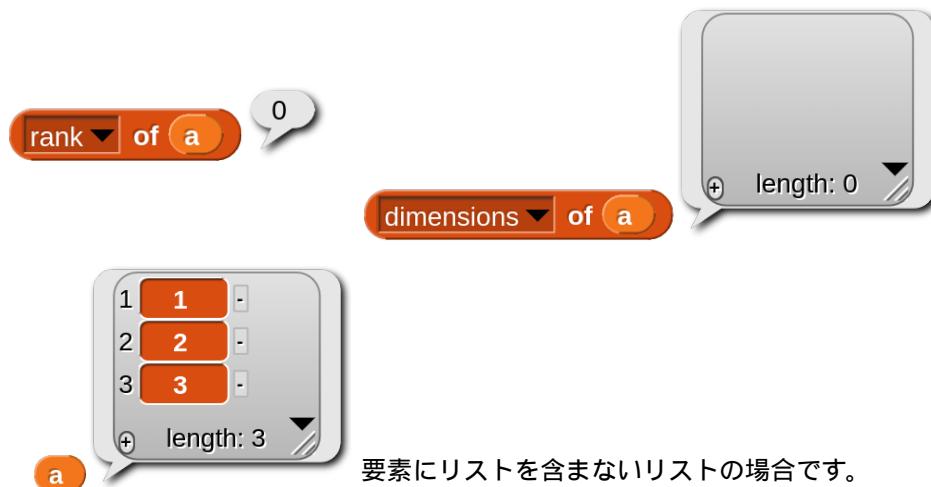
**length ▾ of** ブロックのオプションメニューからいろいろなリスト操作が選べます。

### 4.6.1 rank of ( ), dimensions of ( )

rank, dimensions, reshape は APL 言語の機能を取り入れたものです。

rank ランク (階) は配列の次元を、dimensions は形 (shape) をリポートします。

**a** のようにリストではない場合です。



要素にリストを含まないリストの場合です。



**set [a v] to [reshape numbers from 1 to 6 to 2 3]**

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

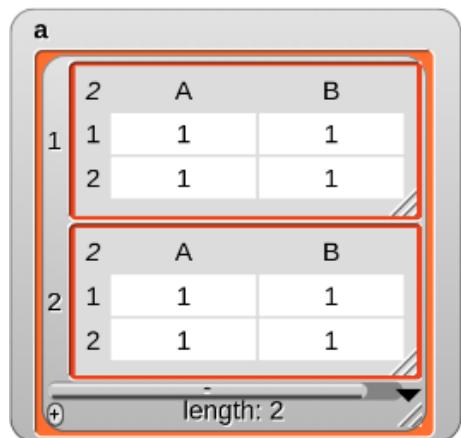
**a** 行列 (マトリックス) の場合です。

reshape で指定した形が dimensions の値になります。



rank は dimensions のリストの要素数を示します。

set **a** to reshape 1 to input list: list 2 2 2



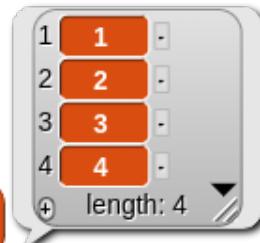
rank of a = 3  
dimensions of a = 2 2 2

リスト操作用のブロックには要素にリストを含むリストには対応していないものもあります。  
その場合、rank が 1 以外は受け付けないというような使い方がありそうです。

#### 4.6.2 flatten of ( )

rank 2 以上のリストを rank 1 に整形してリポートします。

flatten of list 1 list 2 list 3 4



#### 4.6.3 columns of ( ) 転置行列

リストの行と列を入れ替えた転置行列をリポートします。

set **aList** to reshape numbers from 1 to 12 to 4 3

aList

	A	B	C
1	1	2	3
2	4	5	6
3	7	8	9
4	10	11	12

columns ▾ of aList

	A	B	C	D
1	1	4	7	10
2	2	5	8	11
3	3	6	9	12

#### 4.6.4 uniqueness of ( )

リスト中の同じ要素を削除してリポートします。

頻度順になるようです。( 149 ページで別バージョンを作成 )

uniques ▾ of list 1 2 2 3 3 3 ↵

1	3	-
2	2	-
3	1	-
+ length: 3		▼

#### 4.6.5 distribution of ( )

distribution はリストの構成要素の配分をリポートします。次の場合、頻度順に c は 3 個、b は 2 個、a は 1 個という結果になります。

distribution ▾ of list a b b c c c ↵

	A	B
1	c	3
2	b	2
3	a	1

#### 4.6.6 sorted of ( )

リストの要素を整列してリポートします。

数値は後方になるようです。

sorted ▾ of list 2 5 1 e a c ↵

1	a	-
2	c	-
3	e	-
4	1	-
5	2	-
6	5	-
+ length: 6		▼

#### 4.6.7 shuffled of ( )

リストの要素をシャフルしてリポートします。

1	3	.
2	5	.
3	1	.
4	4	.
5	2	.

shuffled ▾ of numbers from 1 to 5

#### 4.6.8 reverse of ( )

リストの要素を逆順にしてリポートします。

1	5	.
2	4	.
3	3	.
4	2	.
5	1	.

reverse ▾ of numbers from 1 to 5

#### 4.6.9 Σ of ( )

リストの要素を合計してリポートします。

Σ ▾ of numbers from 1 to 10 55

要素がリストでも中の値すべて期待したように合計してくれます。

Σ ▾ of list [5 5] list [1 2 3] 16

1	11	.
2	12	.
3	13	.

combine では 1 階層の値の合計値を 2 階層の各要素に加算します。

combine list [5 5] list [1 2 3] using +

#### 4.6.10 text of ( )

リストの要素を空白を間に入れた文字列としてリポートします。

text ▾ of numbers from 1 to 5 1 2 3 4 5

join input list: numbers from 1 to 5 12345

#### 4.6.11 lines of ( )

リストの要素を文字列行としてリポートします。

これが使えるのは rank 1 のリストです。



#### 4.6.12 csv of ( )

リストの要素を csv (comma-separated values) 形式でリポートします。

これをファイルに書き出すと表計算ソフトやスクリプト言語などで利用できます。これが使えるのは rank 1 か 2 のリストです。



#### 4.6.13 json of ( )

リストの要素を json (JavaScript Object Notation) 形式でリポートします。

rank 3 のリストにはこちらを使うことになります。



### 4.7 リストの演算

Snap! では、APL 言語の機能を取り入れているために for ループや map を使わなくてもリストの各要素に演算を施すことができます。( R や Julia などの言語でも可能です。)

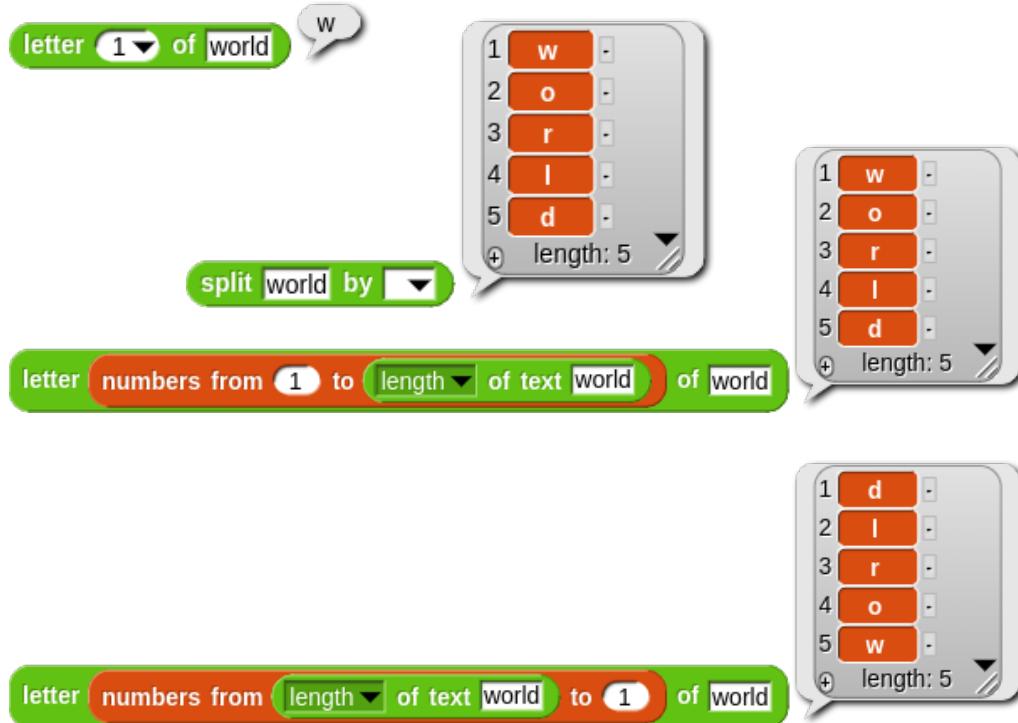




リスト同士で演算をすることもできます。いずれも同じインデックスの要素同士を演算します。  
要素数が合わない場合は対応する部分だけで行います。



リストに文字列の各要素を入れることもできます。



#### 4.8 変数に入れられるもの

変数には、set ブロックで値を入れられますが、変数ウォッチャーの枠の部分を右クリックすると出てくるメニューから値をインポートすることができます。

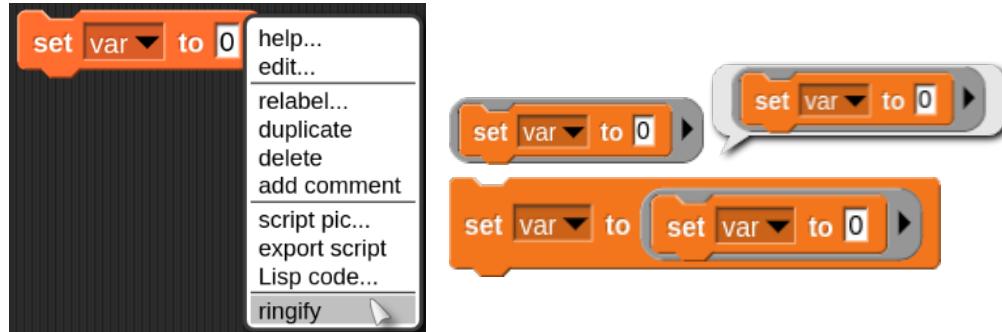


Snap! の変数には、値をリポートするものを入れられます。

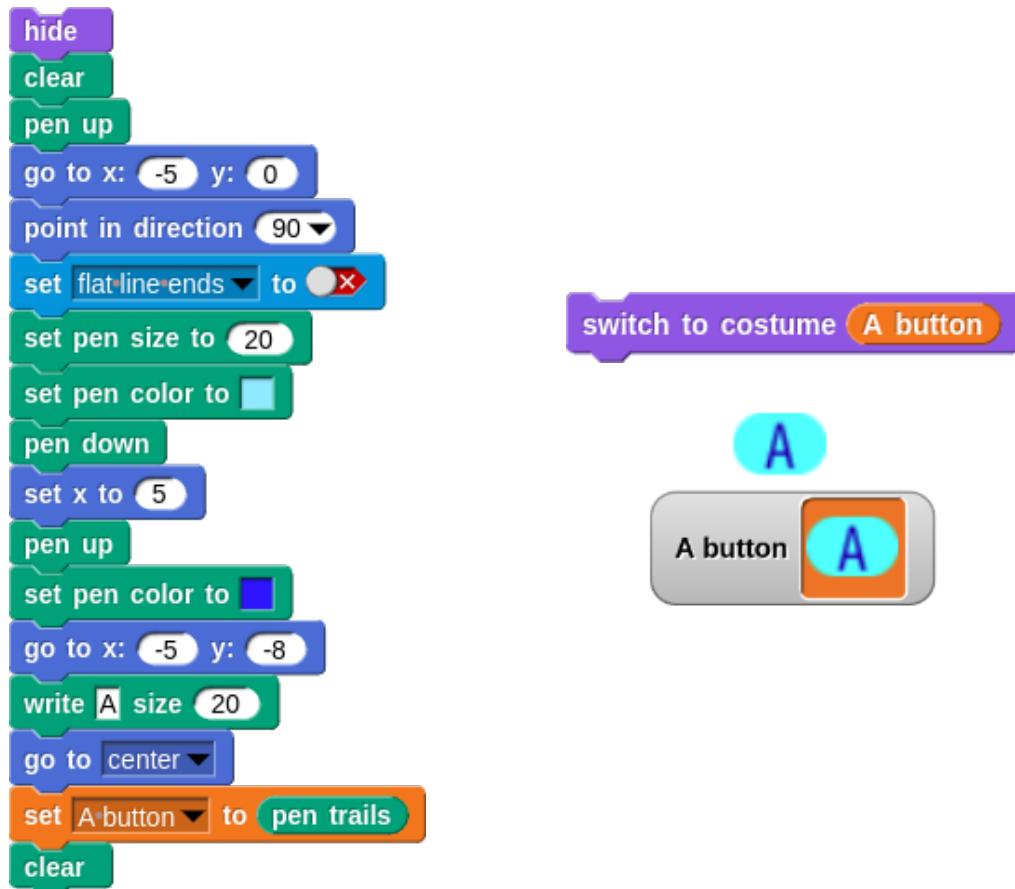




ブロックをリングに入れてやるとリポーターブロックにすることができます。ブロックそのものをリポートするものです。対象のブロックのところで右クリックしてメニューから ringify を選べばリングで囲むことができます。リングを外すには unringify をクリックします。



次のスクリプトは、ステージに pen でボタンを描いてその軌跡を変数にセットします。そして、それを switch to costume でコスチュームにします。これはコスチュームに文字をセットする 1 つの方法です。なお、描き終わりの座標がコスチューム、スプライトの中心になるので go to center を入れて調整しています。

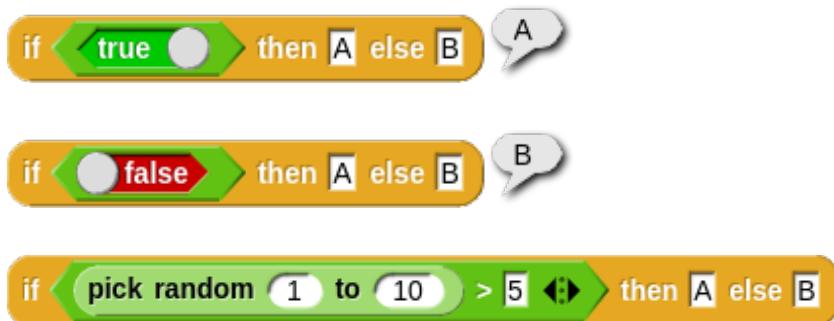


## 5 Control 制御

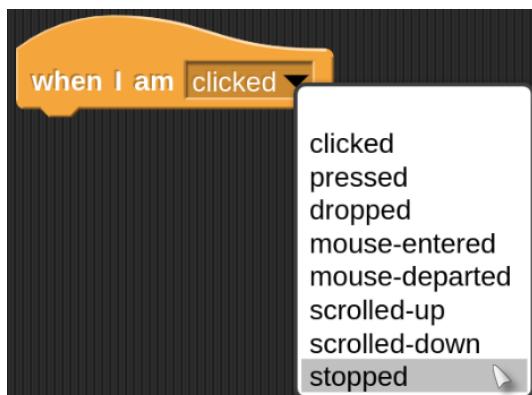
Snap! で使用できる制御ブロックについて。

### 5.1 リポーターの if < > then ( ) else ( )

if < > then ( ) else ( ) 制御ブロックは六角形の条件判断ブロックが true か false かによりどちらかのスクリプトを実行しますが、リポーターブロックは条件判断ブロックが true か false によりどちらかの値をリポートします。



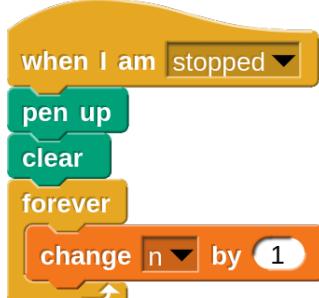
### 5.2 stop ボタンがクリックされた時の終了処理



このブロックで stopped を使用すると のボタンがクリックされた時の処理をすることができます。

終了時のほんの短い時間で機器の終了制御をするためのものらしいのですが、たとえば、接続されたロボットのモーターを止めるとかです。

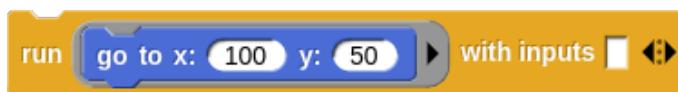
次のスクリプトで動作の確認ができます。変数 n を作成して下さい。けっこうな回数 n のカウントができるようです。



### 5.3 run ブロック

run や call を使うと指定したブロックを実行することができます。これは定義されたブロック内で、引数で渡されたブロックを実行する時などに使用されます。 のように。

たとえば、 で指定の位置に移動します。run ブロックの右端に右向きの三角があります。これをクリックすると、



のようになります。go to x: y: の各入力スロットを空にして、



を実行すると、x と y 両方の入力スロットに 10 が指定されたものとして実行されます。with inputs の入力が 1 個だった場合は、その値がすべての空入力スロットの値になります。右向きの三角をもう一度クリックして入力スロットを追加します。すると、with inputs の入力スロットの値でそれぞれ x y の値を指定できるようになります。



左右の三角のところにリストを持っていくとリストで入力を指定できるようになります。三角のところに近づけると、警告するように赤くなりますが、かまわずにドロップするとセットされます。



with inputs だったのが input list: に変わりました。

#### 5.4 call ブロック

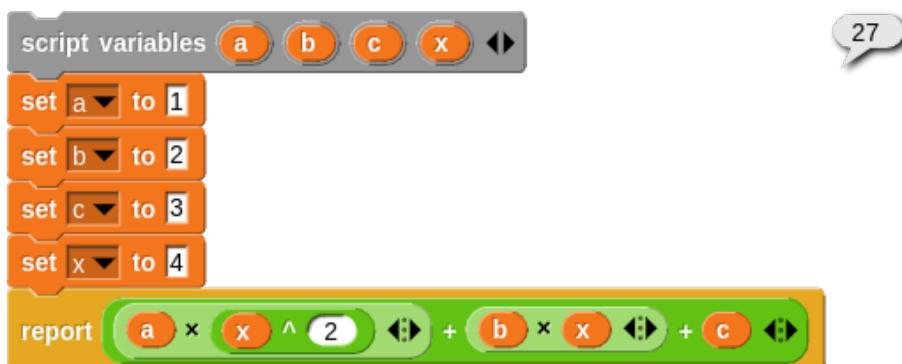
run ブロックに入れられるのが値をリポートしないスクリプトブロックだったのに対して、call ブロックに入れられるのは、実行または評価することによって何らかの値または true か false をリポートするブロックです。call ブロックは、得られた値をリポートします。



call ブロックを使ってちょっとした関数のようなもの（入力に対応する値を返すブロック）を作れます。ブロックを定義するまでもないものなら、これで間に合います。

たとえば、 $ax^2 + bx + c$  の式で、x や a、b、c の値を指定して計算結果を求めてみます。

この式をブロックにすると、 で表され、スクリプトで確かめられるようにするこうなります。



変数の入力スロットを空にして、式をリングで囲みます。



リングの端の三角をクリックしてフォーマルパラメータを出します。

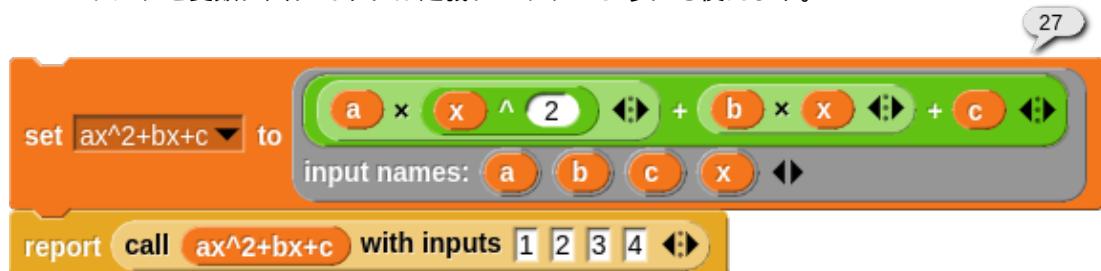


フォーマルパラメータは、クリックして変数名を a、b、c、x に変更します。

それを式の入力スロットに入れれば、call を使った無名関数（ラムダ関数）になります。with inputs で、順にそれぞれの変数の値を指定します。



このリングを変数に入れてやれば定義ブロックのようにも使えます。

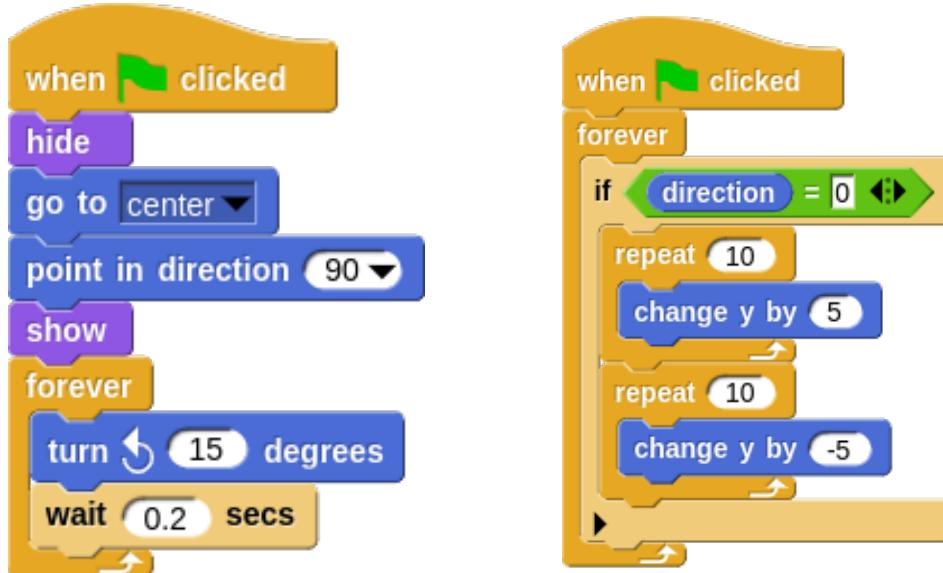


## 5.5 launch ブロック

launch ブロックは指定されたスクリプトを並列で実行するものです。

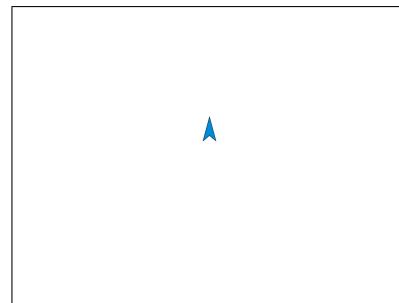
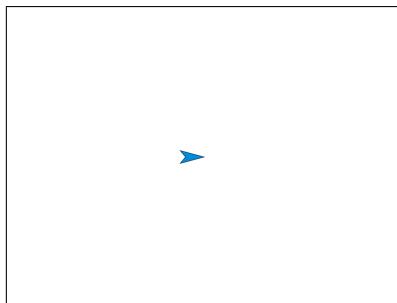
1 個のスプライトに対して、回転させるスクリプトと、角度が上（0 度）の時にジャンプさせるスクリプトを並列で実行してみます。

並列実行は、普通次のようにして 2 つのスクリプトを同時に実行します。

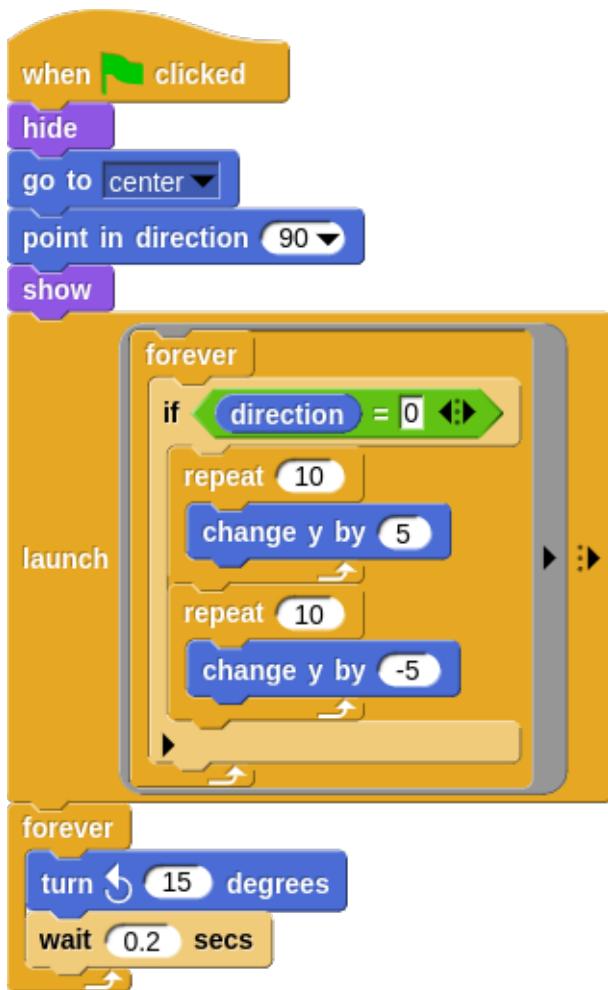


左のスクリプトで、ずっと回転させる動作を行います。

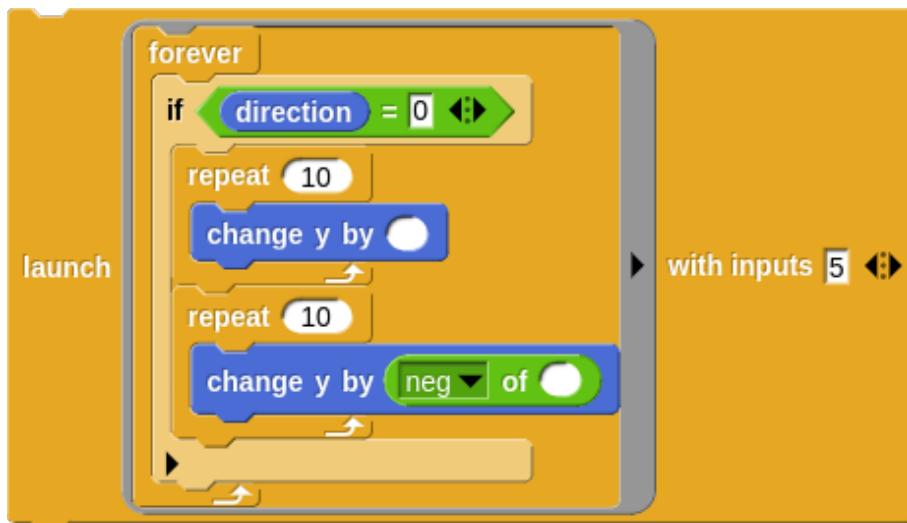
右のスクリプトで、上を向いた時だけジャンプする動作を行います。



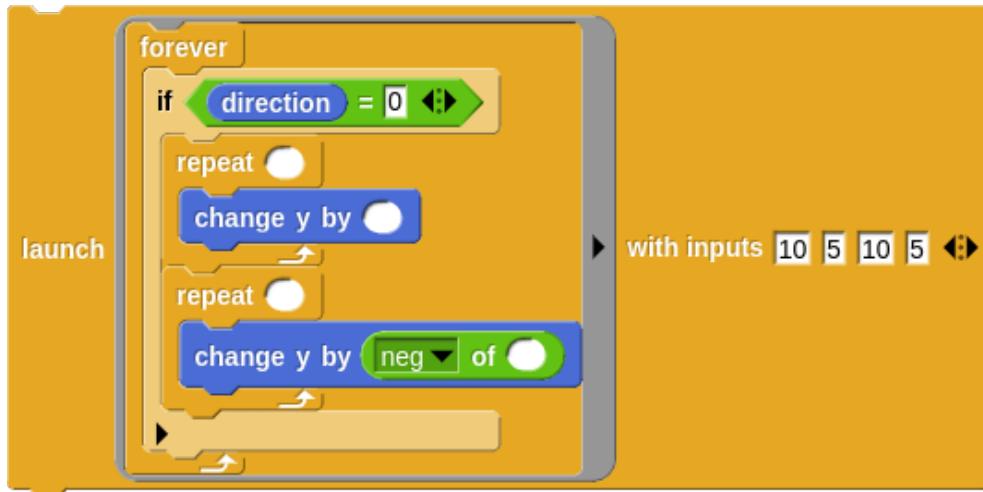
launch を使うことで 1 つのスクリプトで実行することができます。



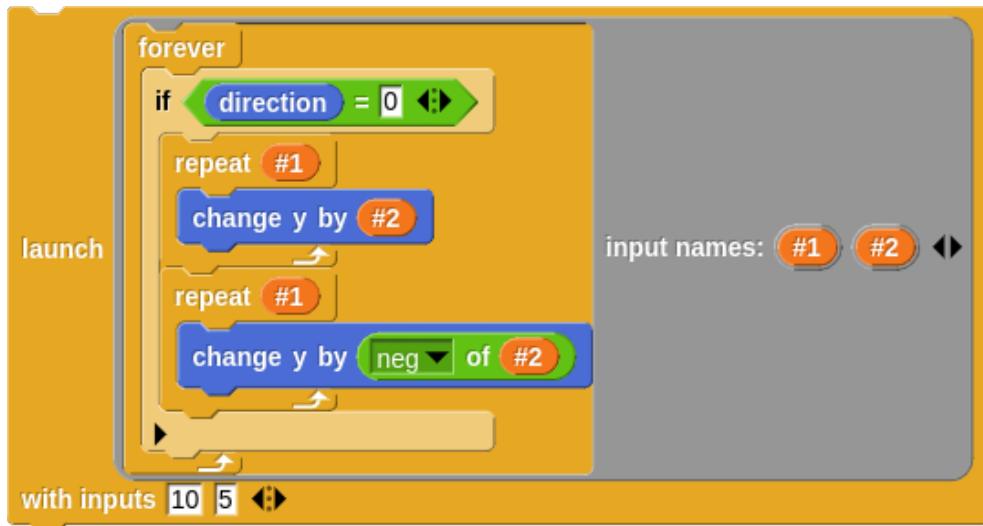
launch の右端の右向き三角をクリックすると、入力値を設定することができます。y の値の設定を空（空白ではなく）にして、入力値をひとつだけ設定すると、その入力値がすべての空の部分の値になります。



2 個以上の入力値を設定する場合、空の入力スロットの個数と合わせる必要があります。

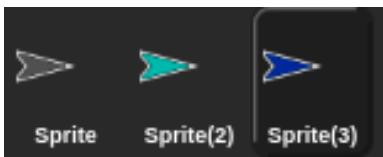


この場合は、リングではフォーマルパラメータが使えるので、次のようにすることができます。リング、灰色の部分の右端の三角をクリックして外側の入力 (10, 5) の個数と同じ個数の変数を用意します。対応する位置に目的の変数を置きます。



フォーマルパラメータの変数名を変更するとスクリプトが分かりやすくなります。  
この with inputs とフォーマルパラメータの利用法は launch だけでなく、他の with inputs を持つプロックで使えます。

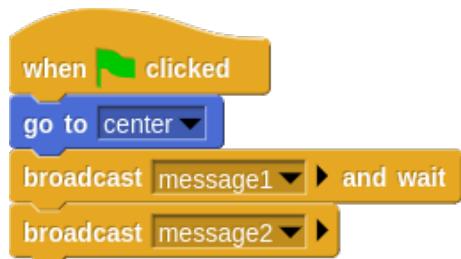
## 5.6 broadcast ブロック, tell to ブロック, request ブロック



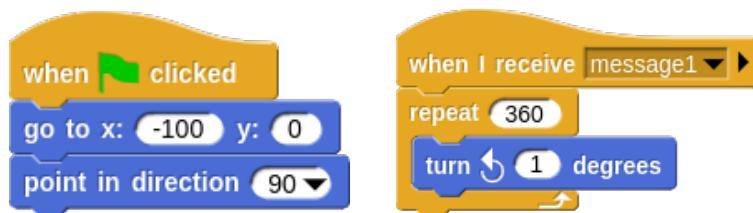
次はスプライトを3個使います。Sprite, Sprite(2), Sprite(3)を用意してください。

Spriteから司令を出してSprite(1)とSprite(2)を順番に動かします。broadcastを使うと次のようになります。

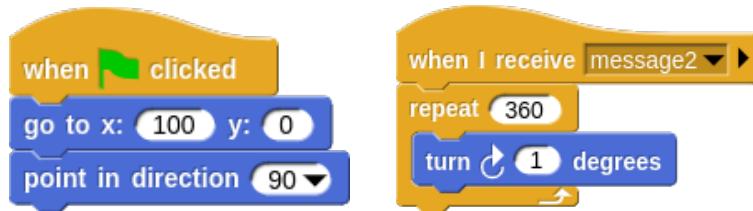
Sprite用



Sprite(1)用



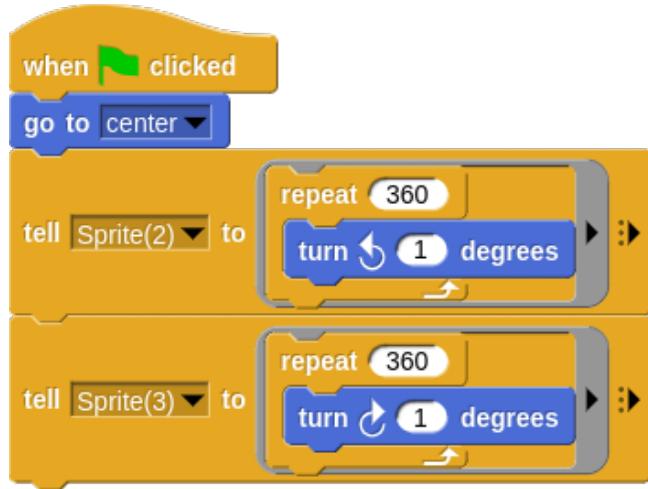
Sprite(2)用



broadcast and wait ブロックは、メッセージを受け取ったがわのwhen I receiveで指定されたスクリプトの実行が終わるまで待ちます。

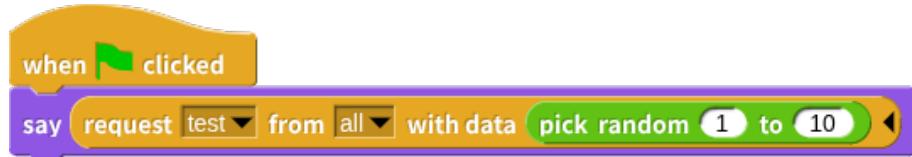
のブロックを使用すると broadcast を使わなくても Spriteから Sprite(1), Sprite(2)を直接操作することができます。

Sprite 用

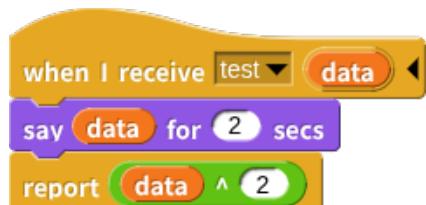


ver. 1.1 から broadcast and wait の機能を持つ request ブロックが追加されました。メッセージを受け取った側で値を返し、送信した側ではそれを受け取ることができます。

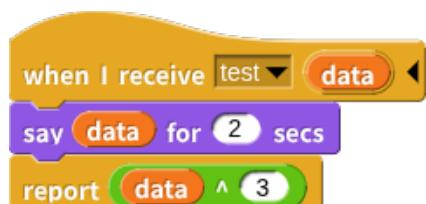
Sprite 用 (すべてのスプライトに対して乱数値をデータとしてメッセージを送ります。)



Sprite(1) 用 (受け取ったデータを 2 乗して返します。)



Sprite(2) 用 (受け取ったデータを 3 乗して返します。)





## 5.7 pipe

Unix 系の OS では、プログラムで処理したデータの出力を別のプログラムが入力として受け取って別な処理をして出力する pipe (パイプ) というデータのリレーのようなことが行われます。

**pipe** で Snap! マニュアル ver.8 冒頭部分の単語数を数えてみます。

We have been extremely lucky in our mentors. Jens cut his teeth in the company of the Smalltalk pioneers: Alan Kay, Dan Ingalls, and the rest of the gang who invented personal computing and object oriented programming in the great days of Xerox PARC. He worked with John Maloney, of the MIT Scratch Team, who developed the Morphic graphics framework that's still at the heart of Snap!.

"We have been ~ at the heart of Snap!." をコピーしてエディターで "SnapText.txt" という名前で保存します。それをスクリプトエリアにドロップすると、SnapText という変数が作成されます。変数 SnapText は文字列をリポートします。文章から単語に分解するには **split** by word です。リストの要素数を求めるには **length** of **目** です。順番に実行してみてください。

**pipe** **SnapText**

**pipe** **SnapText** **split** by word

**pipe** **SnapText** **split** by word **length** of **目**

受け取った値は指定のスロットに入って処理されます。  
67

## 6 プロックを作成する

Scratch のユーザー定義ブロックはある動作をするスクリプトに引数を渡せるようにして汎用的にしたものでした。ですが、定義内のローカル変数は使えず、値をリポートすることもできないので使いづらい面があります。Snap! は、Build Your Own Blocks と銘打っているようにいろいろなカスタムブロック（ユーザー定義ブロック）が作れるようになっています。

### 6.1 定数の作成

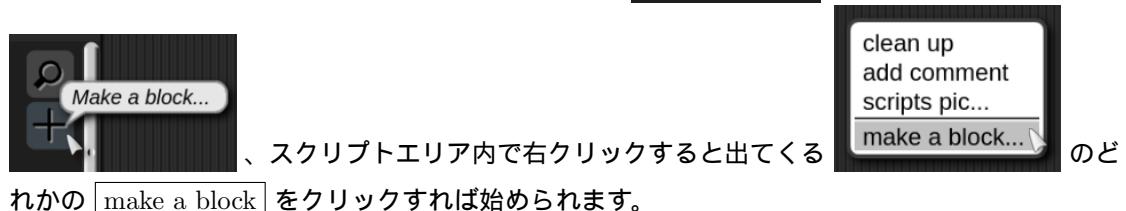
一般的なプログラミング言語では値が変更されない定数を定義することができます。Scratch では、変数で代用するしかありませんでした。



変数は変更されることを前提としているものなので定数として使用するのは好ましくないことはあります。  $\pi$  は良いとしても  $e$  などは普通の変数と思われてしまいそうです。

Snap! ではリポーターブロックを作成できるので、正しく定数を作成することができます。

ブロックを作成するには、パレットエリア内にある  、



Motion のパレットエリアにある作成ボタンをクリックすると Motion カテゴリーのブロックしか作れないというわけではないので、どれを使って始めてかまいません。設定用のウィンドウが現れます。選択できるカテゴリーには Other がありますが、 をクリックしてメニューから New category... を選択すると、独自のカテゴリーを作成してパレットに追加することができます。パレットの色も指定できます。



パレットのカテゴリーを選択するボタンや、新しいブロックの名前を入れる欄、ブロックの機能の種類を選択するボタン、このブロックをこのスプライトだけの機能にするかのボタンがあります。カテゴリーで Other 「その他」というものを選ぶと、置き場所は Variables のところになります。Command は、値をリポートしないブロックです。Reporter は、ブロック内で設定した値をリポートします。Predicate は、述語と訳されます。true か false をリポートします。Event Hat はなんらかのイベント発生用です。(89 ページ参照) それぞれの形が、できたブロックの使われ方を示しています。ブロック名入力欄に  $\pi$  を入れて、上の欄でボタン Operators を、下の段で Reporter を選択してください。Operators を選択することはパレットの種類を決めることであり、置き場所を決めることもあります。

Ok をクリックするとブロックエディターが表示されます。



定義の先頭の  $+ \pi +$  の部分をプロトタイプといいます。ここを左クリックするとブロックのカテゴリーを変更することができます。

Make a block のところで Reporter を選択したのであらかじめ report ブロックが配置されています。

report ブロックに  $\pi$  の値を設定して OK をクリックすると、単に  $\pi$  の値をリポートするブロックができます。パレットエリアに作成したブロックが表示されます。

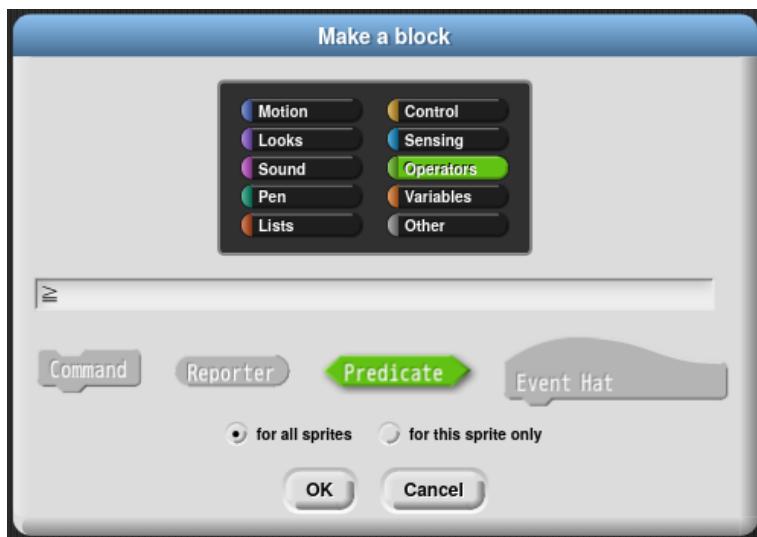


同じように e も作成することができます。



## 6.2 ( ) ( ) ブロック

引数を使用する例として( ) ( )のブロックを作成してみます。12ページで示したように $\geq$ ブロックがあるので必要ないですが、最初の練習としては適切です。

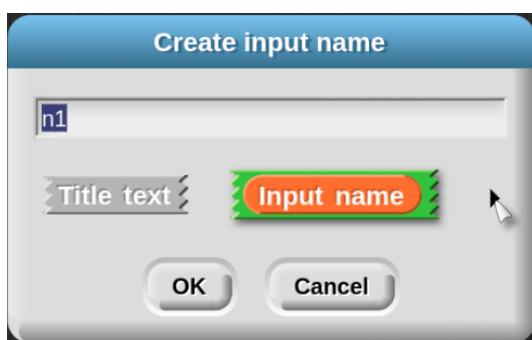


ブロック名入力欄に「」を入れて（全角文字「きごう」とか「だいなりいこーる」で変換できます）、上の欄でボタン Operators を、下の段で Predicate を選択してください。プリミティブの「>」ブロックと同じになります。Predicate は形から分かるように、Control コントロールブロックの条件式のところなどで使用されて true または false を返すためのものです。

Ok をクリックするとブロックエディターが表示されます。



「」の左側の + ボタンをクリックしてください。



すると、入力ウィンドウが出ます。左側に Title text、右側に Input name のボタンがあります。ブロックに表示される文字列を指定する時には Title text をクリックして文字列を設定します。ブロックを使用する時にデータを受け取るための変数を指定する時には Input name で設定します。

今は変数を指定するので Input name を選択します。入力欄に n1 を入れてから右にある小さな三角をクリックしてください。すると、次の窓が表示されます。

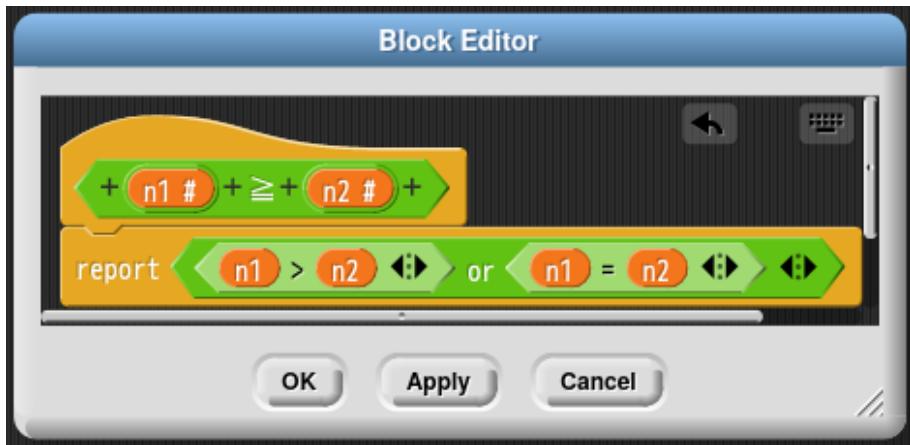


現在は Any type が選択されています。Any つまり数値でも文字でも受け付けるタイプです。このままでいいのですが、あえて Number にしてみます。これは数値しか受け付けないタイプです。Any type を選んだ場合は、変数の表記に「#」が付かないだけで以下の操作は同じです。もう一つ、下の方に Single input. Default Value: [ ] が出ます。ここで入力の初期値が設定できるのですが、この場合は関係ないのでこのままにしておきます。OK をクリックして次に進んでください。

右側の + ボタンをクリックして、同じように n2 の設定をしてください。



パレットエリアからブロックを持ってきて完成させてください。

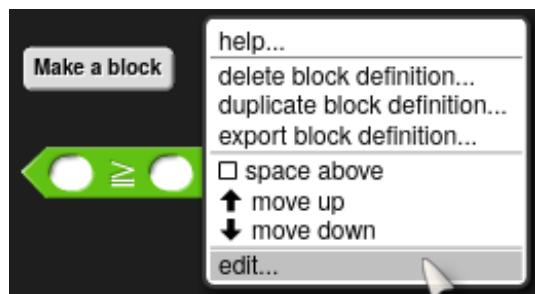


report は値をリポートする（返す）ためのブロックです。この場合は式の結果により真理値、true か false をリポートすることになります。apply をクリックしてください。

パレットエリアに がセットされます。



正しい値をリポートしないならばスクリプトを見直してください。正常ならば OK をクリックして終了です。



ブロックを右クリックすると、edit... で下記のように内容の編集ができます。  
export block definition をクリックすると、このブロック定義だけをファイルに書き出すことができます。



変数 n1# を左クリックすると、変数名や設定の変更をすることができます。

 を使ってステップ実行する場合に、作成したブロックの内部をステップ実行させたければ  
 オンにしてからそのブロックをブロックエディターで表示させておく必要があります。順序が逆だとそのブロック内のステップ実行はされません。( 83 ページ参照 )

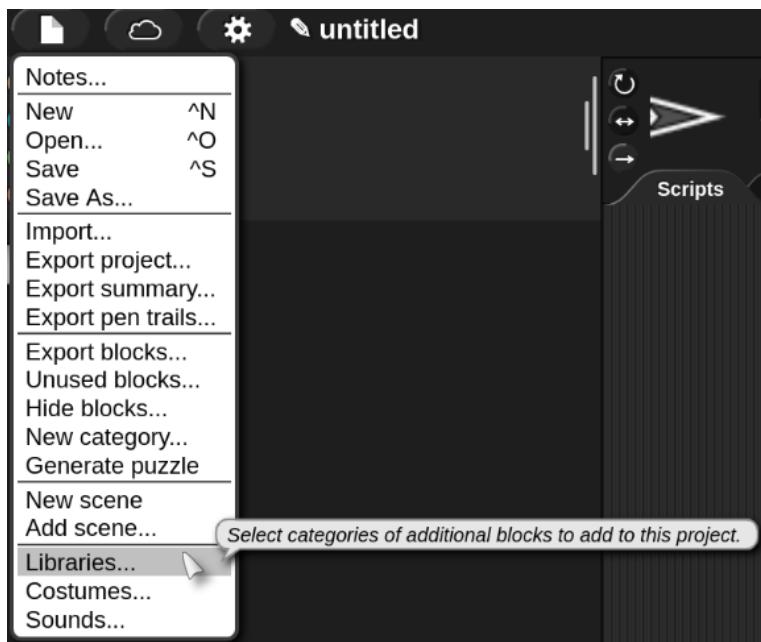
### 6.3 help 説明文の作成

プロトタイプの部分にコメントを付けると、そのコメントの内容が定義ブロックの help で表示されます。普通のブロックの場合はブロックのところで右クリックして add comment をしますが、プロトタイプではブロック定義の背景の部分で右クリックして add comment をします。作成したコメントをプロトタイプの部分に持っていくと、ハロが出るのでドロップします。

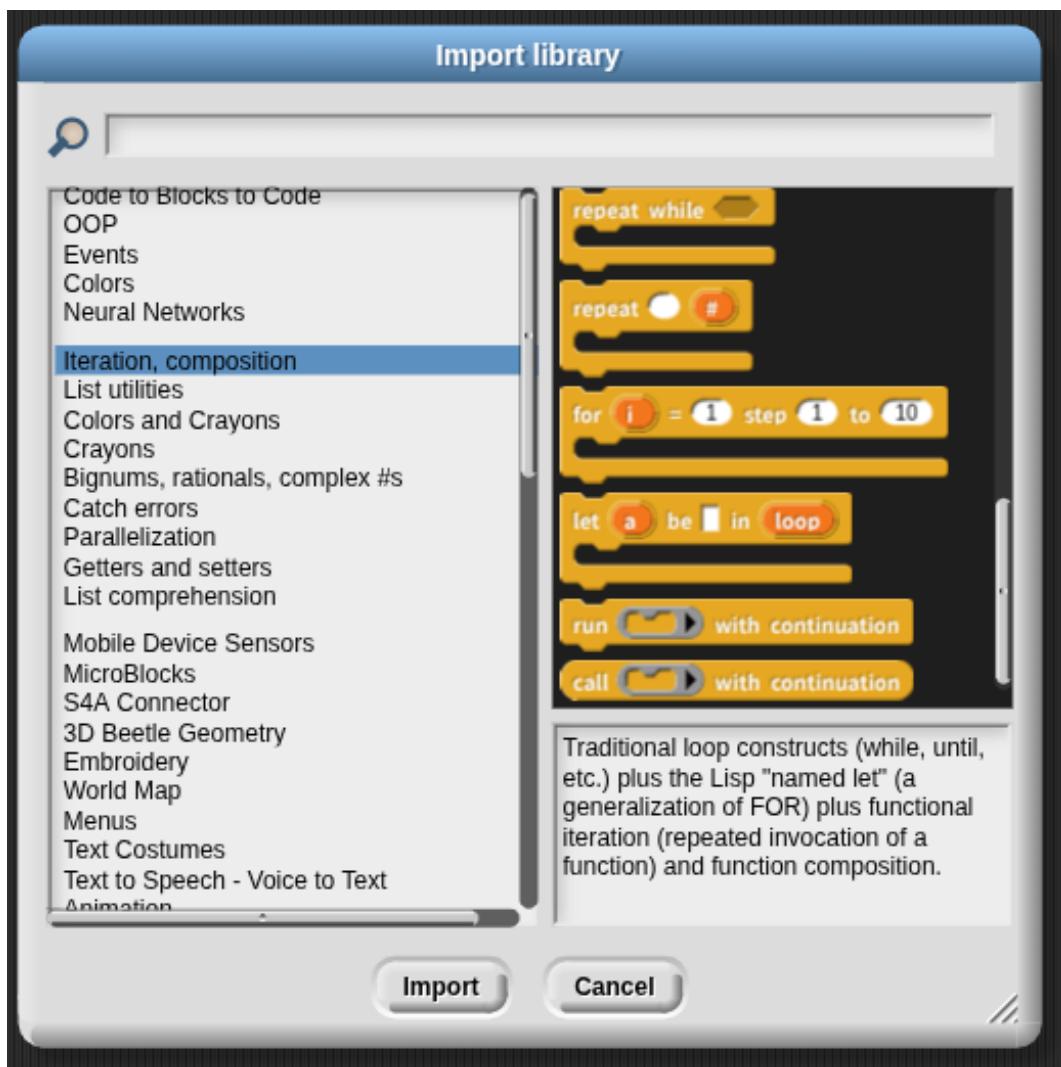


### 6.4 for (i) = (start) to (end) step (step)

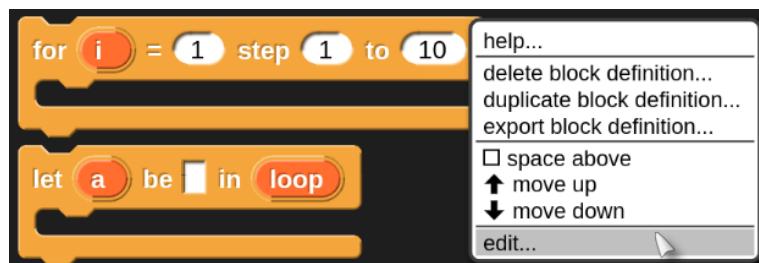
 Snap! には  ブロックがありますが、増減分が指定できるものも Libraries から追加できます。  
 をクリックします。



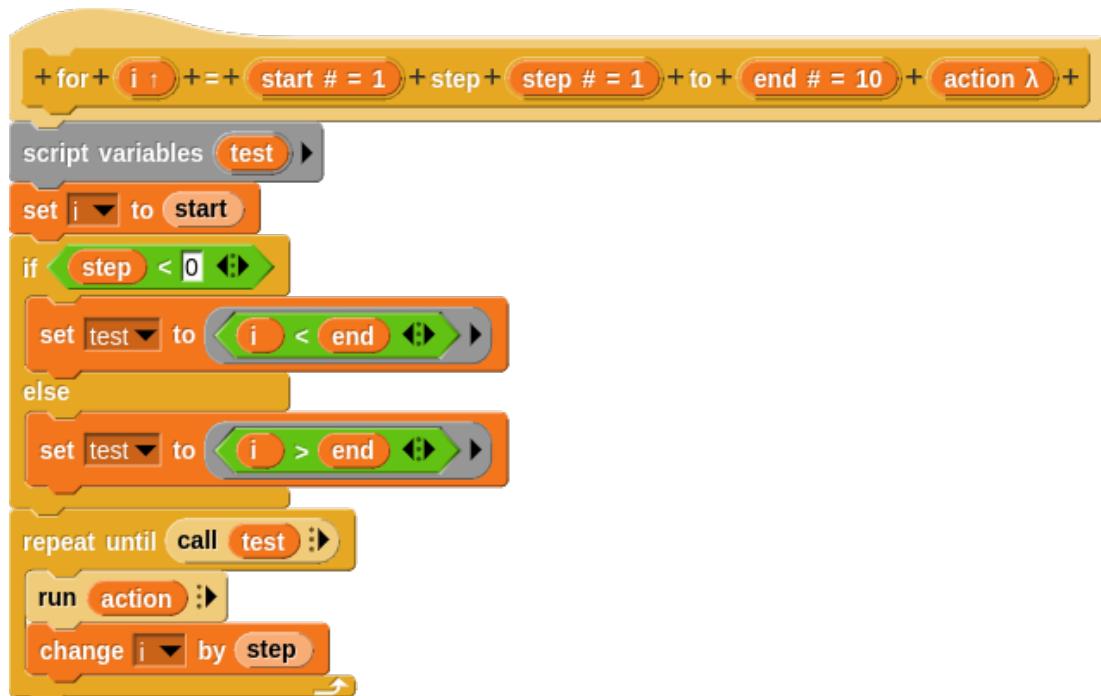
Libraries... をクリックします。Iteration, composition をクリックして内容を確かめてから Import すれば追加できます。



そうすると、パレットエリアの Control コントロール のところの Make a block の下に追加されたブロックが表示されます。



を右クリックして、edit...  
で中身を見てみます。



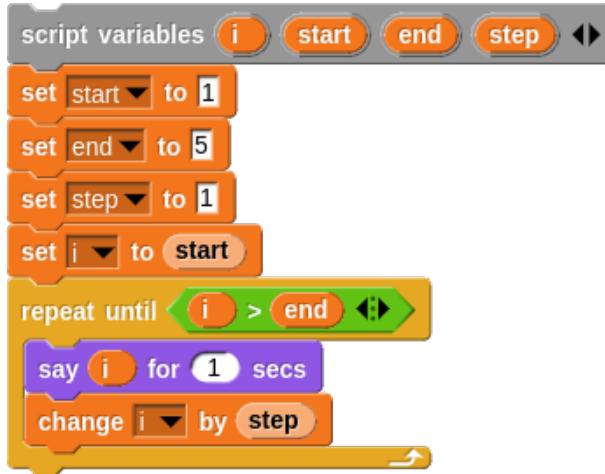
このスクリプトにしたがって、 $\text{my for } (i) = (\text{start}) \text{ to } (\text{end}) \text{ step } (\text{step})$  という増加部分の位置  
が違うだけのブロックを作成してみます。

まずは、スクリプトエリアで for ブロックの動作を確認しながら組み立ててみます。

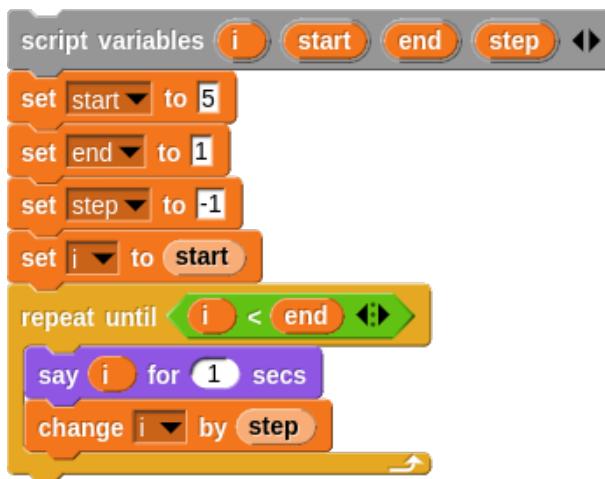
*i* が増加していく場合です。



start, end, step, i の変数をスクリプト変数を使い で作ってみます。



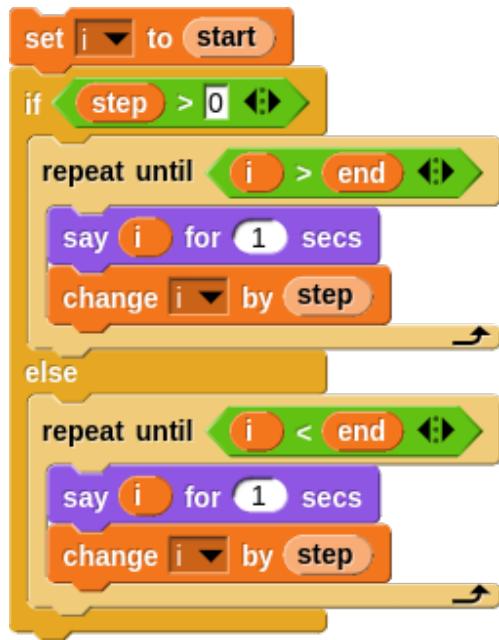
start を 5、end を 1、step を -1 にすると、 $5 > 1$  で、 $i > end$  の終了条件を満たしてしまうので実行されません。減少カウントにする場合は、終了条件を  $i < end$  にして



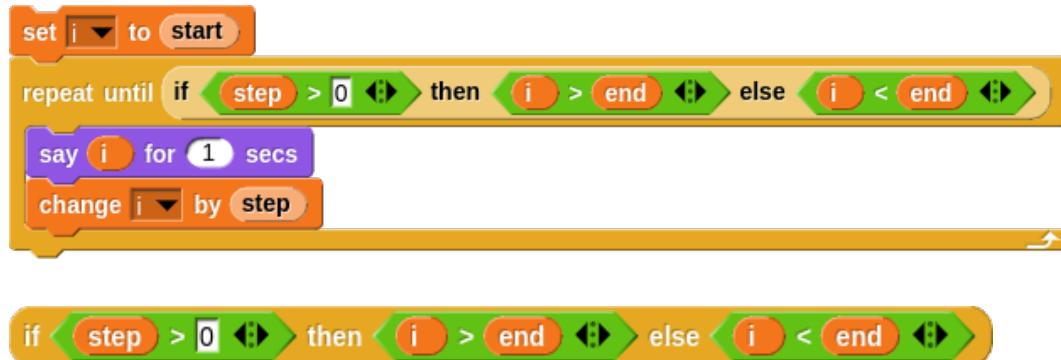
のようにしなければなりません。

増加でも減少でも対応させると、for ループは次のようにになります。

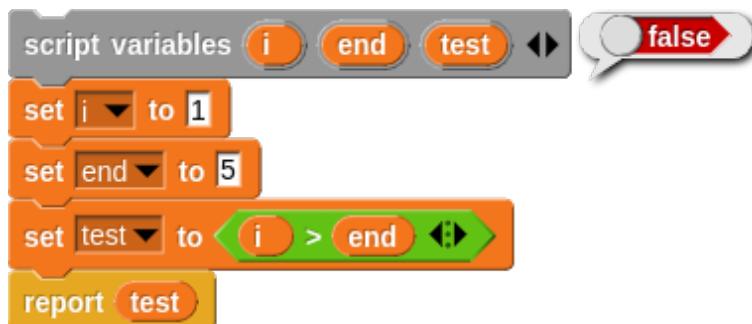
上記スクリプトから **set [i v] to [start]** 以降を表示します。



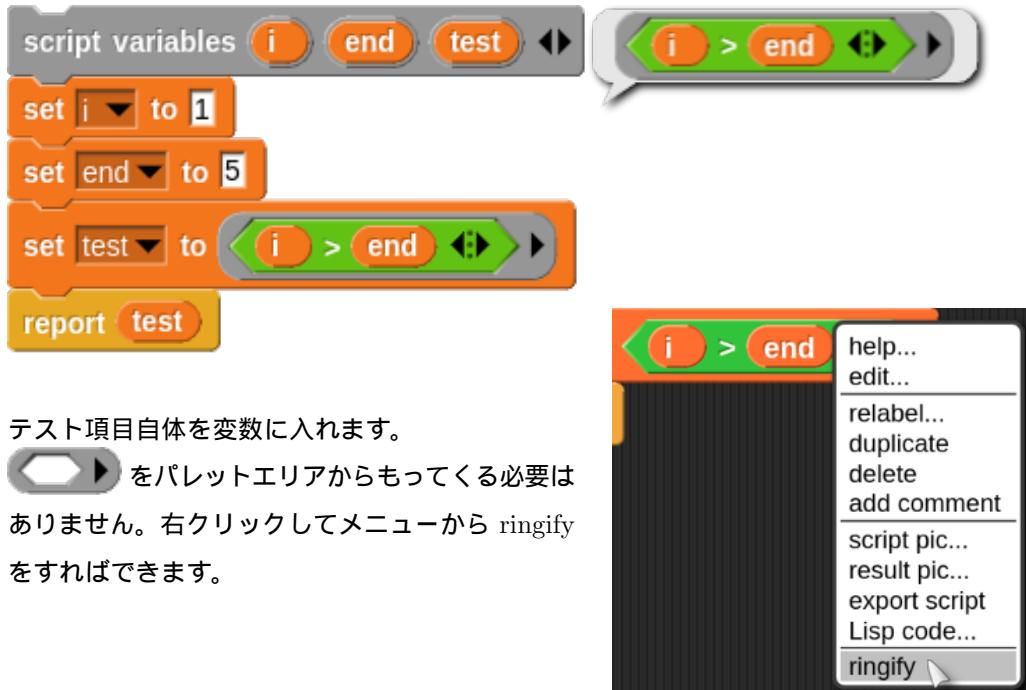
ループのテスト部分をまとめると次のようにすることができます。



の部分で、step の値によって `i > end` か `i < end` がテストされるわけです。ループに入る前にどちらのテストをするべきかは決まっているので変数に設定できればいいのですが、次のようにすると、その時点の i の値でテスト値が設定されてしまいます。つまり、 $1 > 5$  なので、false です。

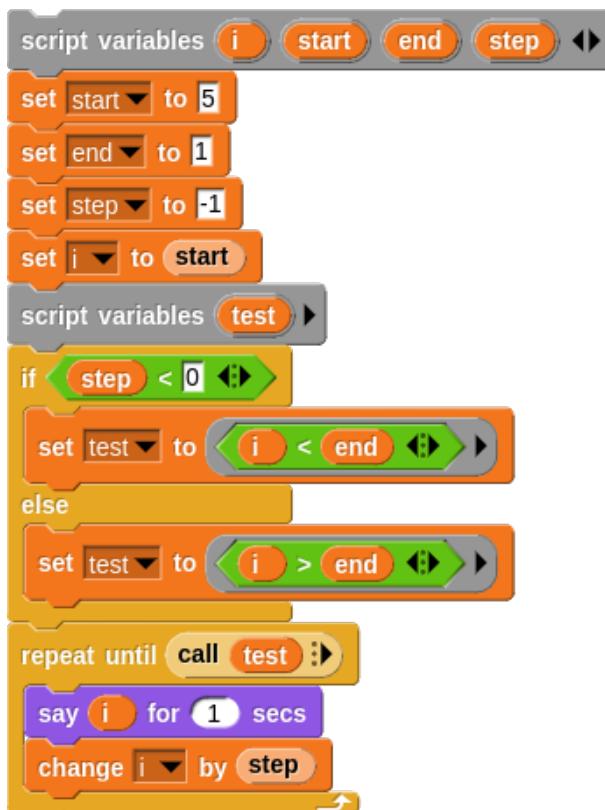


ループしている中で変化する i の値に対してテストする必要があります。そこで使用される機能がリングです。



テスト項目自体を変数に入れます。  
[loop control] をパレットエリアからもってくる必要はありません。右クリックしてメニューから ringify をすればできます。

これを for ループに使用します。



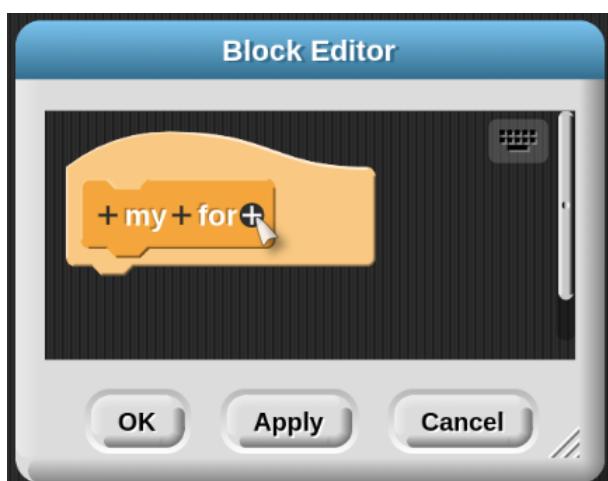
これが Libraries の for ループの内容です。

ところで、このままではちょっと問題があります。増分が 0 の場合に無限ループになってしまう

のです。増分が 0 のだからそれでいいと考えることもできますが、無限ループになるのは嫌です。増分が 0 の場合には何もしないで終わるか、1 回だけ実行するかの選択がありますが、my for では何もしないで終わるようにします。

それでは、いよいよブロックエディターで作成していきます。

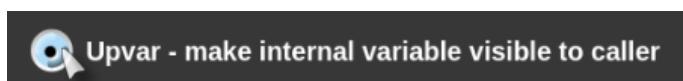
ブロックエディターを開いてから、Control, Command を選択して、my for と入力して OK で次に進んでください。

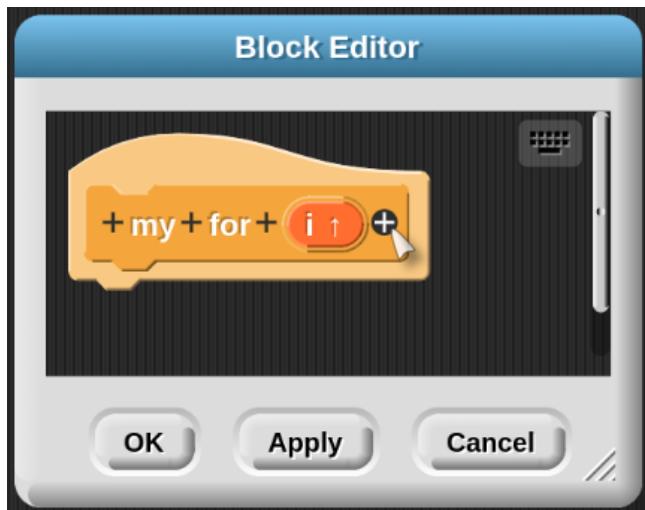


「+」をクリックして、変数 i の設定をします。

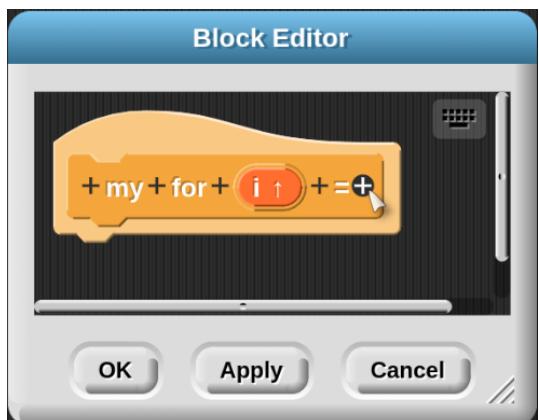


変数 i は、for ループの中にドラッグ&ドロップして使用できる特別な変数です。Upvar オプションを選択します。すると、自動的に Number や Any type のオプションはクリアされます。





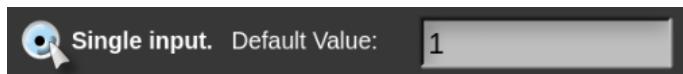
変数 *i* のところに Upvar を示す「↑」が表示されます。続けて、「+」をクリックして、「=」を、Title text として入力してください。



続いて、「+」をクリックして、変数 start を設定します。

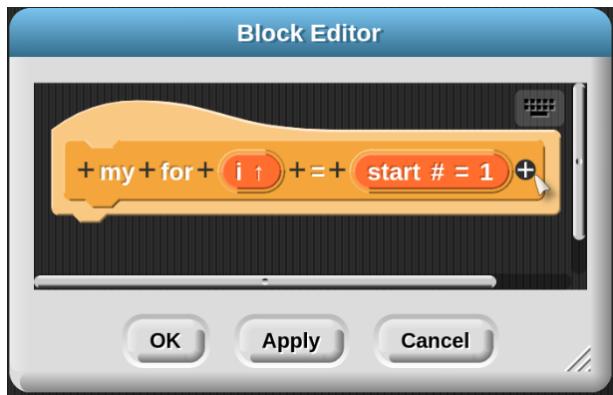


オプションとして、 Number 、数値入力限定で、



規定値を 1 に設定します。

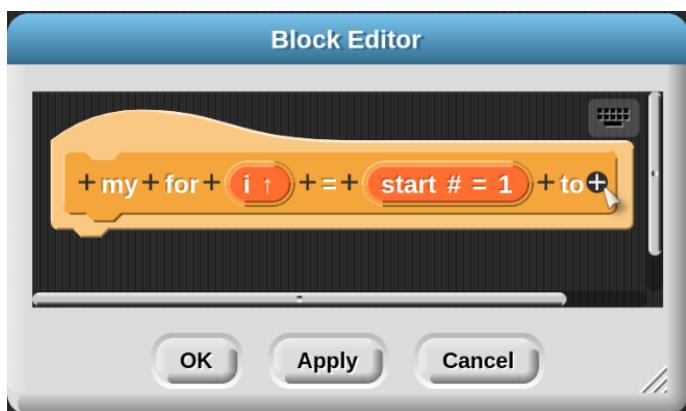
続いて、「+」をクリックして、



「to」を、Title textとして入力してください。



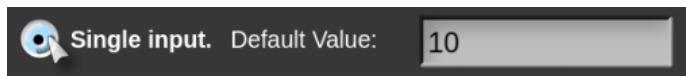
続いて、「+」をクリックして、



変数 end を設定します。

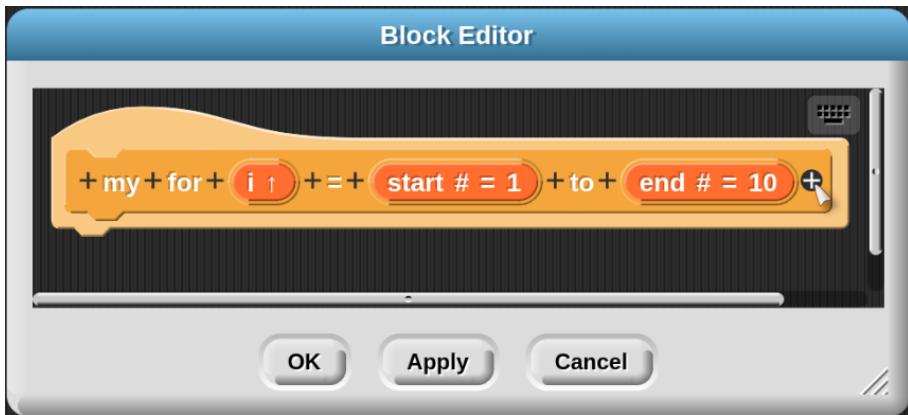


オプションとして、、数値入力限定で、



規定値を 10 に設定します。

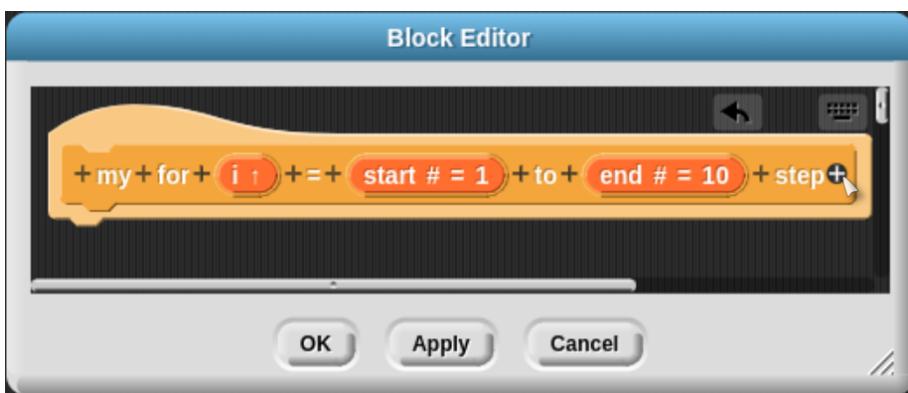
続いて、「+」をクリックして、



「step」を、Title text として入力してください。



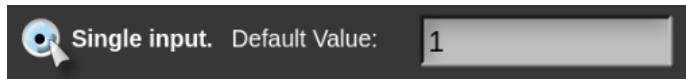
続いて、「+」をクリックして、



変数 step を設定します。

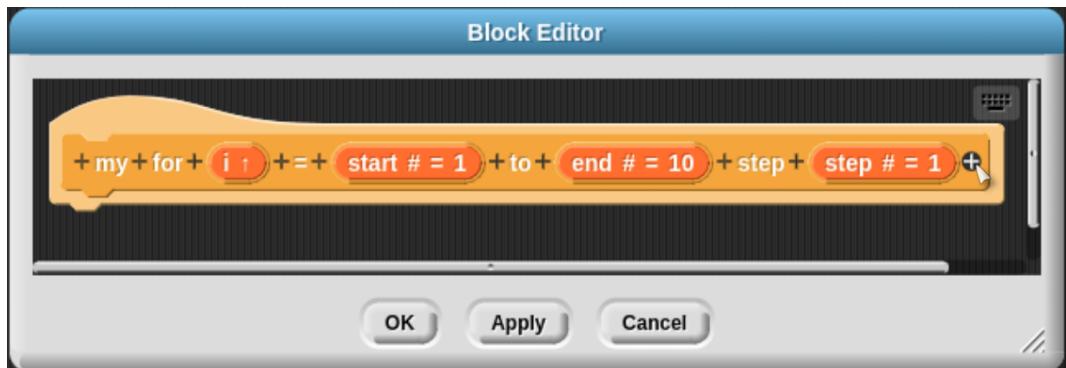


オプションとして、、数値入力限定で、



規定値を 1 に設定します。

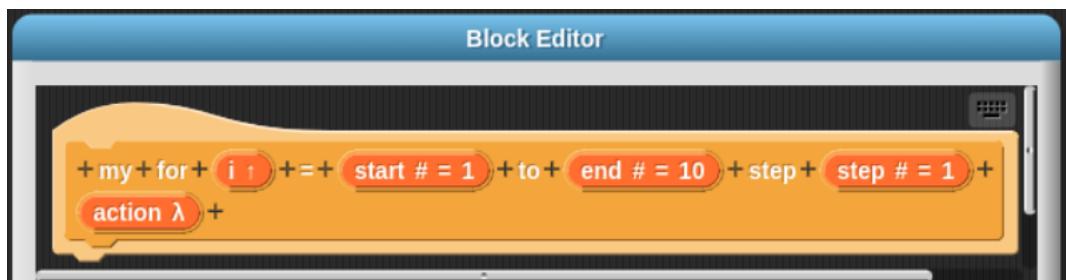
続いて、「+」をクリックして、



変数 action を設定します。



オプションとして、 を選択すると、自動的に がセットされます。(ループのマークのところにチェックを入れるとこれが表示されます。) これによって for ループ内で実行するスクリプトを受け取ります。



変数 action に特別なマークが付きました。

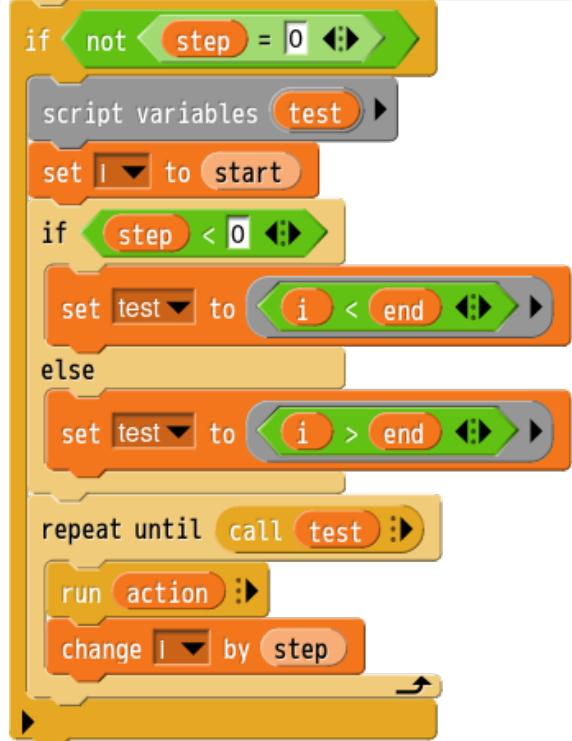
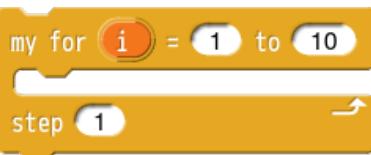
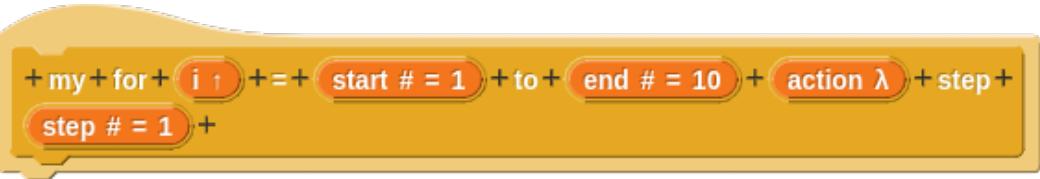
プロック定義の本体として、実験していた for ループのスクリプトを持ってきます。action で指定されたスクリプトを実行するプロックは run です。

で実験用のスクリプト を入れ替えます。

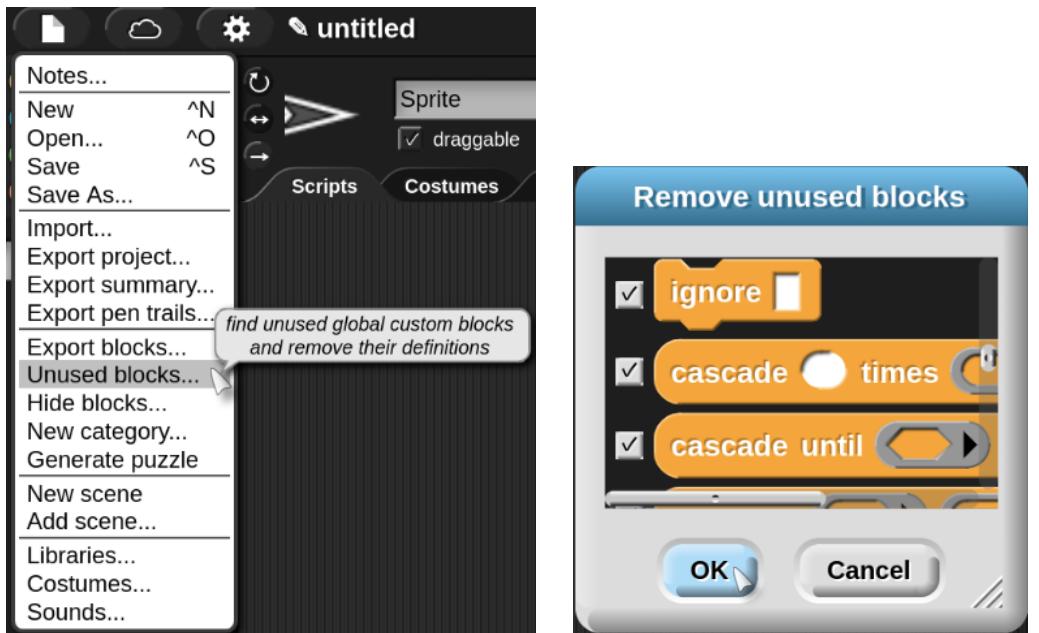
Apply するとパレットエリアに作成したプロックがセットされます。テストをしてみて問題がなければ OK をクリックしてブロックエディターを閉じてください。



プロトタイプで step に関する部分 (step + step # = 1) を action の後ろに持ってくると次のようにすることができます。



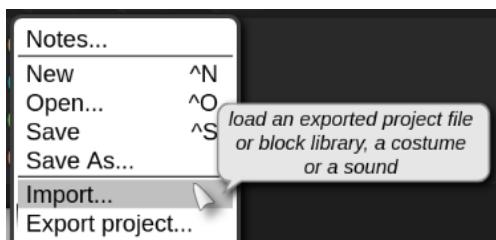
ところで、Libraries... からブロック定義をインポートしてプロジェクトを作成した場合に、普通に保存すると未使用的ブロック定義も含んだファイルになります。プロジェクトを公開する場合は、次のようにして未使用的ブロック定義を削除してファイルの容量を小さくしたほうがいいかもしれません。



自作した定義ブロックを他のプロジェクトで読み込んで利用できると便利です。定義ブロックだけをエクスポートしてやると可能になります。次のようにエクスポートするブロックが選択できます。不要なものはチェックを外します。



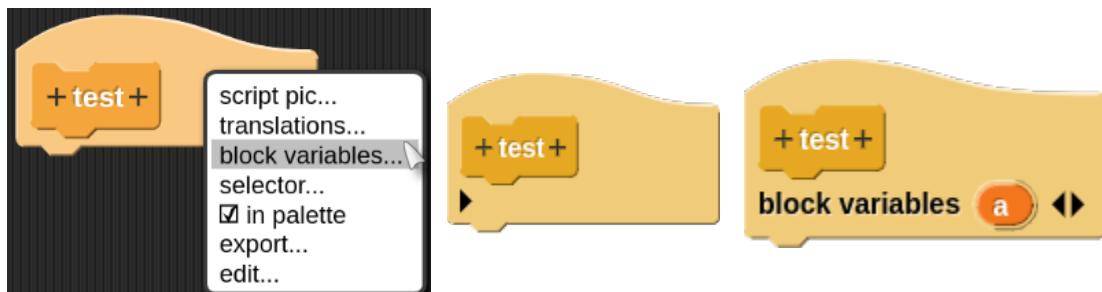
OK をクリックすると、ダウンロードフォルダに「????? blocks.xml」というファイル名で保存されます。????? のところにはプロジェクト名又は untitled が入ります。適宜リネームしてください。これを利用する時は、次のようにしてインポートすれば使用できるようになります。



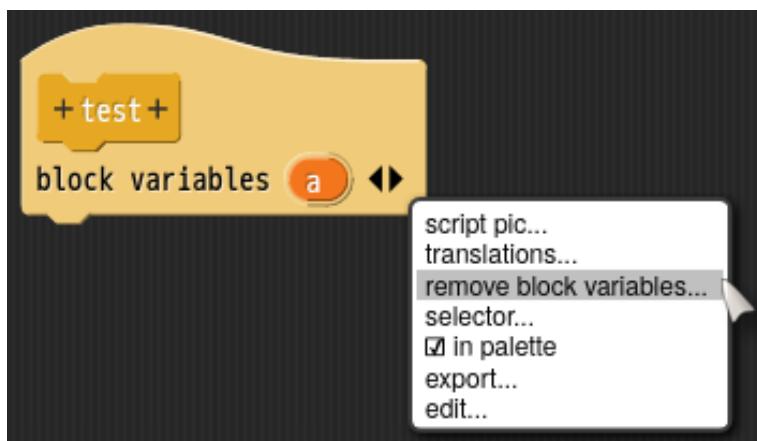
## 6.5 block variables

ブロックのプロトタイプのところで右クリックして、block variables オプションを選択すると、ブロック内変数を使用できるようになります。右向きの三角をクリックすると変数の作成で、左向きの三角をクリックすると削除です。リネームもできます。

スクリプト変数は script variables ブロック以降有効になりますが、block variables はその定義の先頭からブロック内だけで有効な変数になります。



もしも変数が必要なくなった場合は、次のようにブロックのプロトタイプのところで右クリックすると remove block variables... で削除することができます。

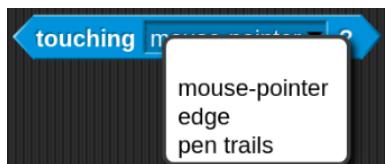


この文書では紙面の都合でブロック内変数を使用していますが、スクリプト変数を使っても同じ結果になります。

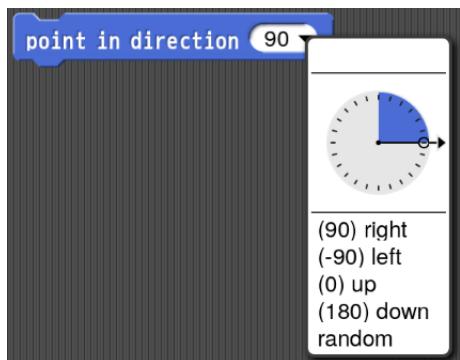
## 7 ブロック定義について

### 7.1 プルダウン入力

touching ブロックなどのように項目指定用のプルダウンメニューが設定されているものがあります。



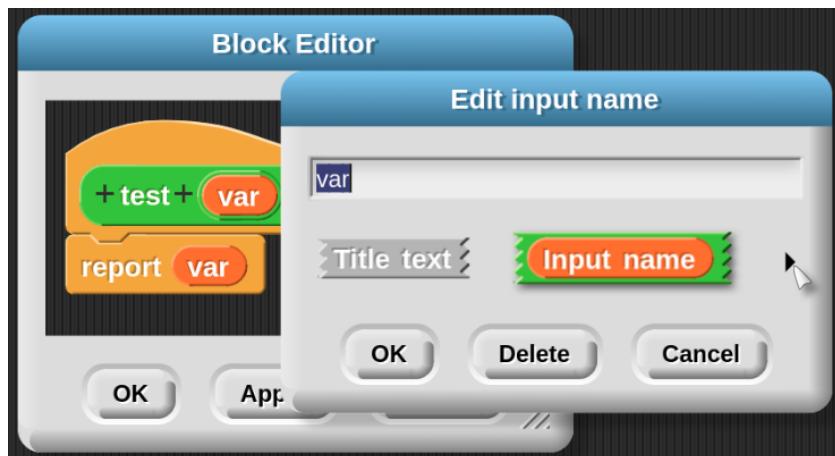
入力スロットに値をキーボードから入力することはできなくなっています。これは、後で出てくる read-only のオプションが指定されているためです。



point in direction ブロックへの入力のように、方向を示す針を動かしての指定や、直接数値を指定できたりするものもあります。touching ブロックと違い、白い入力スロットになっています。これは、ユーザーがプルダウンメニューを使用する代わりに任意の値を入力できることを意味しています。read-only のオプションが指定されていないために、このような仕様になります。

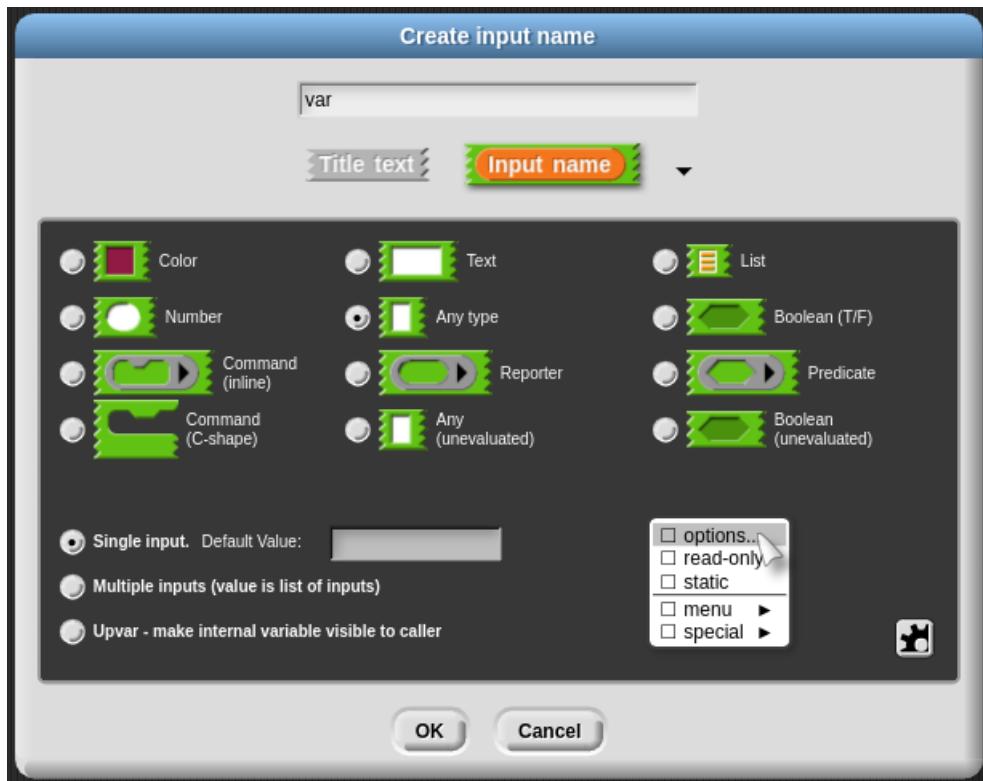
カスタムブロックにもこのようないろいろな入力方法の指定が可能です。ただし、ユーザーインターフェースは今後変更される可能性があります。

説明のために、test というブロックを作成します。

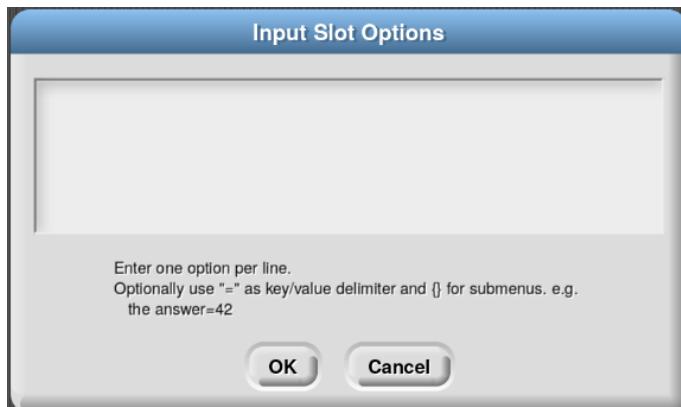


プルダウン入力を行うには、Input name の設定ダイアログを開きます。

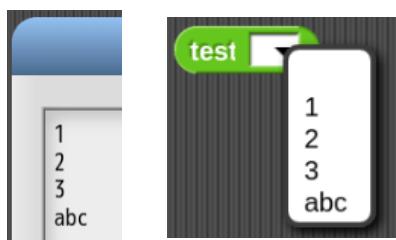
の var をクリックすると Edit input name のダイアログが開きますから、Input name の右側にある三角をクリックします。これで、大きな Edit input name のダイアログが開きます。ここの暗い灰色の領域で右クリックするか、ボタンをクリックします。すると、このようなメニューが表示されます。



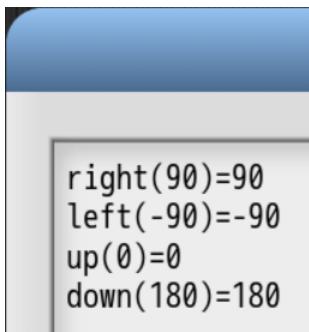
読み取り専用のプルダウン入力にしたい場合は、`read-only` チェックボックスをクリックします。  
メニュー項目を設定するには、`options...` を選択し、このダイアログボックスを表示します。



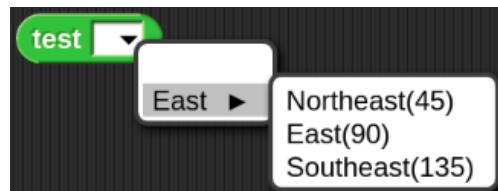
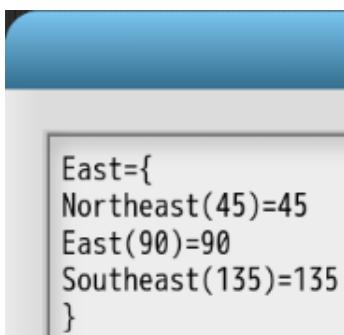
ここに各行にオプションを入れてプルダウンメニューを作っていきます。  
左のように設定して、ブロックエディターを `Apply` すると、右のような結果になります。



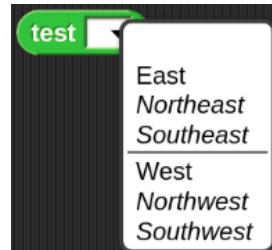
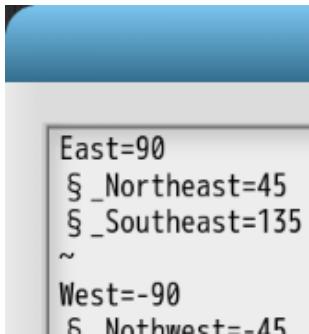
設定したものがそのまま表示され、クリックするとそれが入力値になります。  
次のように、「=」で値を設定すると、メニューには「=」の左側の項目が表示されますが、クリックされると「=」の右側にある項目が入力値になります。



次のように、「 ={ 」で行を終えると、サブメニューを設定することができます。「 ={ 」の左側の項目はサブメニューの名前であり、メニューには「 ► 」を付けて表示されます。ここにマウスポインターを置くとその横にサブメニューが表示されます。「 } 」だけの行はサブメニューを終了させます。サブメニューは任意の深さまで入れ子にすることができます。



次のように、「 ~ 」チルダだけの行はセパレーター（水平線）になります。



項目の前に、「 §\_ 」セクション記号とアンダースコア（アンダーバー）を置くと、シフトキーを押しながらクリックしないとプルダウンメニューに表示されなくなります。因みに、§ 記号は Windows OS の場合だと、Alt キーを押しながらテンキーから（通常の数字キーではなく）0167 と入力、Linux OS の場合だと、[Ctrl]+[Shift]+[u] を押してから、a7 を入力して [Enter] で出すことができます。a7 は 0167 の十六進数コードです。OS を問わず、日本語入力モードで「せくしょん」または「きごう」と入力して変換して出すこともできます。

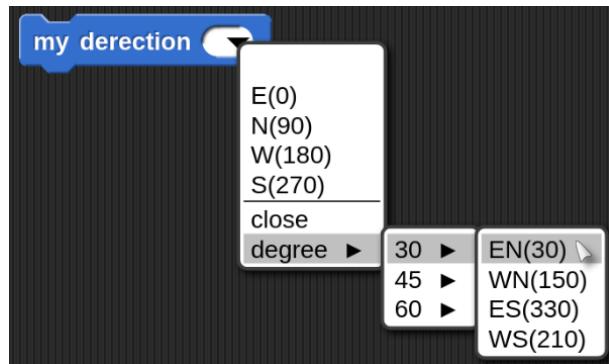
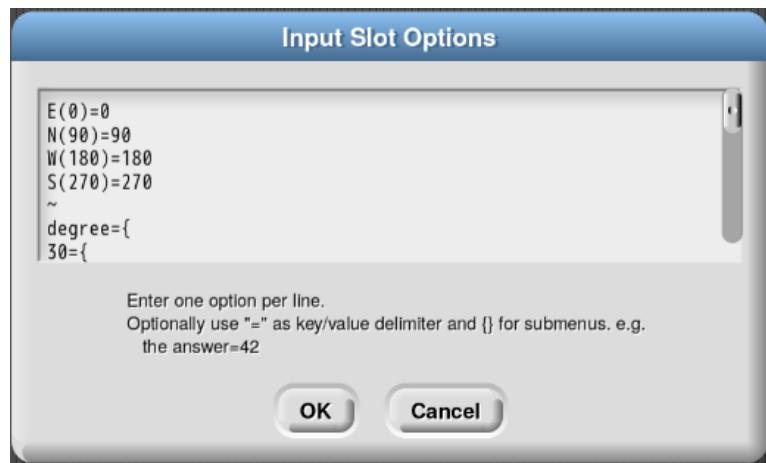
Single input. Default Value: に値を設定すると、選択肢のデフォルトになります。

プルダウンメニューの例として、my direction というブロックを作ってみます。Input name を degree とし、一般的な数学上の角度で向きを指定できるようにします。つまり、右方向が 0 度、上方向が 90 度、左方向が 180 度 という具合です。オプションを以下のように設定します。

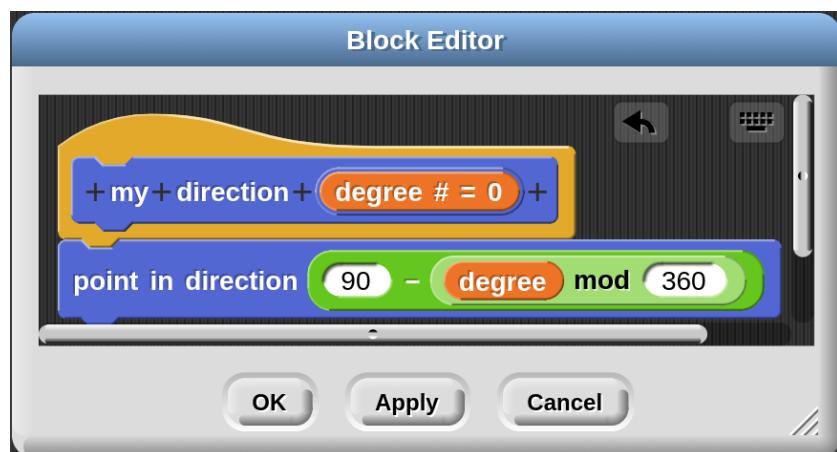
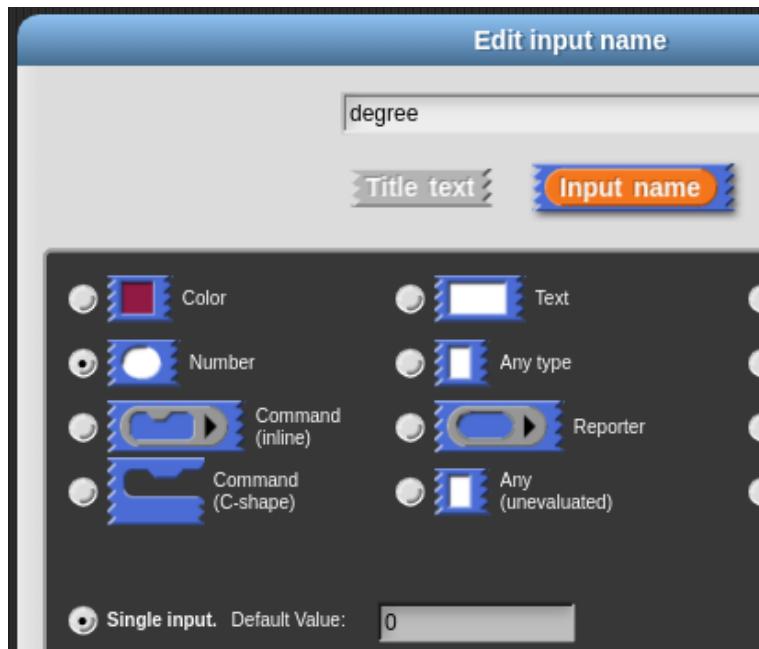
```

E(0)=0
N(90)=90
W(180)=180
S(270)=270
~
degree={}
30={}
EN(30)=30
WN(150)=150
ES(330)=330
WS(210)=210
}
45={}
EN(45)=45
WN(135)=135
ES(315)=315
WS(225)=225
}
60={}
EN(60)=60
WN(120)=120
ES(300)=300
WS(240)=240
}
}

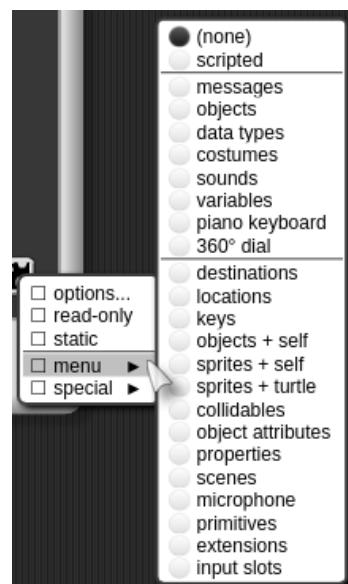
```



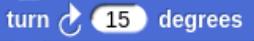
degree から、EN, WN, ES, WS のサブサブメニューを作っています。あくまで機能説明の作例ですので、あまり意味のあるものではありません。

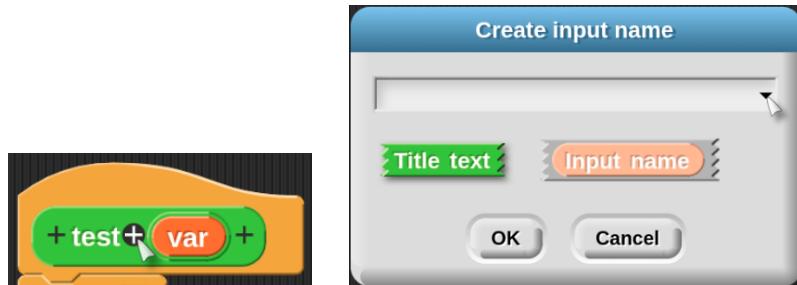


また、menu サブメニューから選択することで、  
入力スロットへの指定の仕方をいろいろ設定す  
ることが可能になります。



## 7.2 Title Text とシンボル

プリミティブブロックの中には、表示に  の回転する矢印のようにシンボルが含まれているものがあります。カスタムブロックでもシンボルを使用できます。ブロックエディターで、プロトタイプのシンボルを挿入したい位置のプラス記号をクリックします。すると、ダイアログが開きますから Title text にしてください。



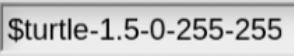
次に、入力待ちのテキストボックスの右端にある  をクリックします。するとシンボルのメニューが表示されます。まとめて表示すると次のようになります。( Title text としてセットされたものを右クリックしてもシンボルメニューが表示されます。)



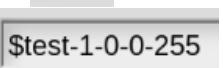
そこからお目当てのシンボルを選択します。 turtle を選んでみます。

すると、入力欄に  \$turtle がセットされます。

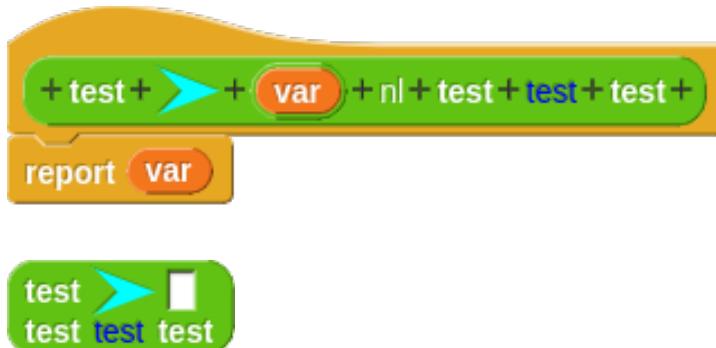
OK して Apply すると、 になります。

定義を  \$Sturtle-1.5-0-255-255 に変更すると、 になります。シンボルの後

の「-1.5-0-255-255」は、表示倍率(1.5)指定、RGB(Red=0, Green=255, Blue=255)によるカラーコード指定です。表示倍率だけの指定もできます。

シンボルメニューの最後に、「new line」があります。Title textにこれ  を設定するとそこで改行されます。また、シンボルじゃなくても、文字列の頭に「\$」  を付けるとシンボルのように倍率と色の指定ができます。反面、シンボルと同じ文字列は使用できないということですが。

定義は、こうなります。



### 7.3 Input name オプションについて

ブロックを作成する時の Input name のオプションについて見ていきます。間違っているかもしれません、こういう考え方をするとオプションを選ぶ目安になるのではないかと思います。

Snap!でブロックを作成する時には受け取る変数のタイプを指定することができます。(指定しなかった場合は、Any type, Single inputになります。)期待する入力のタイプを入力スロットの形で示すためであったり、機能を設定するためです。

ブロックエディターの Input name オプションには 12 個の選択肢がありますが、Command、Reporter、Predicate の 3 つに分類できます。

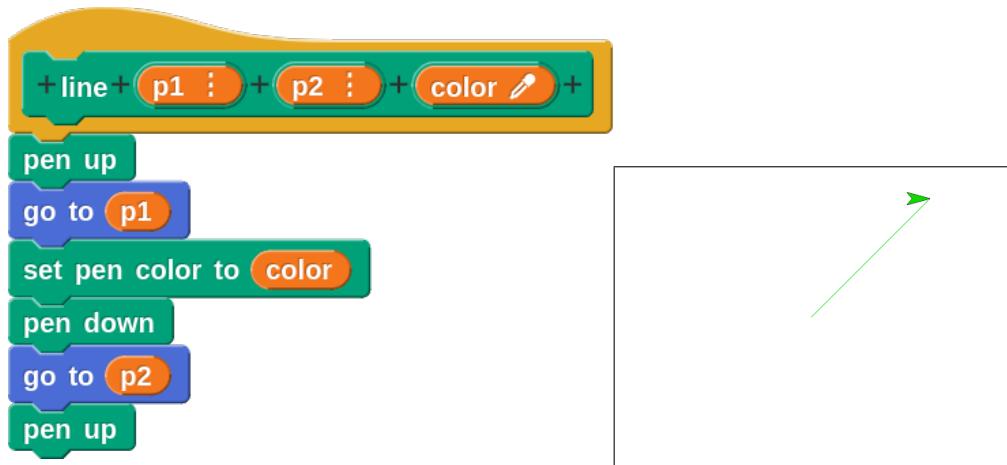
Command 型は、なにかを実行する、上下に凹凸がついたジグソーパズルピースのような形のコマンドブロックを入れられるものです。Reporter 型は、なにかしらの値をリポートする、楕円形のリポーターブロックを入れられるものです。Predicate 型は、真理値 true か false をリポートする、六角形のブロックを入れられるものです。(「真理値」は「真偽値」と表現されることもあります。)

また、それぞれについて下の段の 3 種類のオプション (Single input, Multiple input, Upvar) を設定できる場合があります。設定された内容によってプロトタイプ内では次のように表示されます。

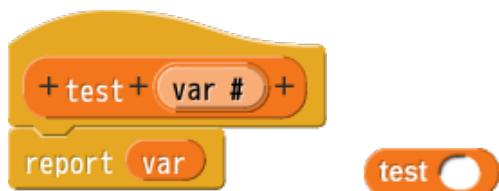
 a = 1	default value	 a...	multiple input	 a ↑	upvar	 a #	number
 a λ	procedure types	 a :	list	 a ?	Boolean	 a ↗	color
 a >>	object	 a ¶	multi-line text				

### 7.3.1 Reporter 型

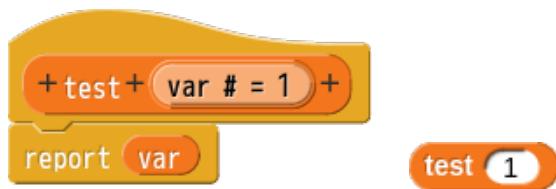
- Color は色用の入力スロットを作成するためのものです。次のスクリプトは二つの点の座標リストと色を指定して線を描くものです。Color を使用すると、プリミティブのブロックと同じように色指定をすることができます。



- 入力スロットへのキーボードからの入力を数値限定にしたのが、Number です。



Default Value を指定すると、



既定値を設定することができます。

- 入力スロットにキーボードからテキストを入力できることをアピールするのが、Text です。しかし、数値をテキストとして扱ってくれるわけではありません。



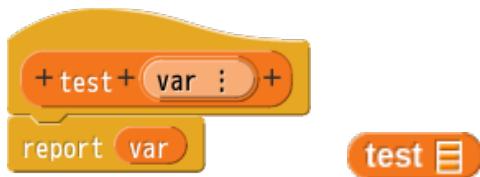
- special のオプションから multi-line を選択して次のような定義にすると、入力スロットにキーボードから改行を含むテキストを入力できる multi-line text になります。



- 入力スロットにキーボードから数値でもテキストでも入力できることをアピールするのが、Any type です。Text とは入力スロットの形がちょっと違います。



- リストの入力を求めていることをアピールするのが、List です。



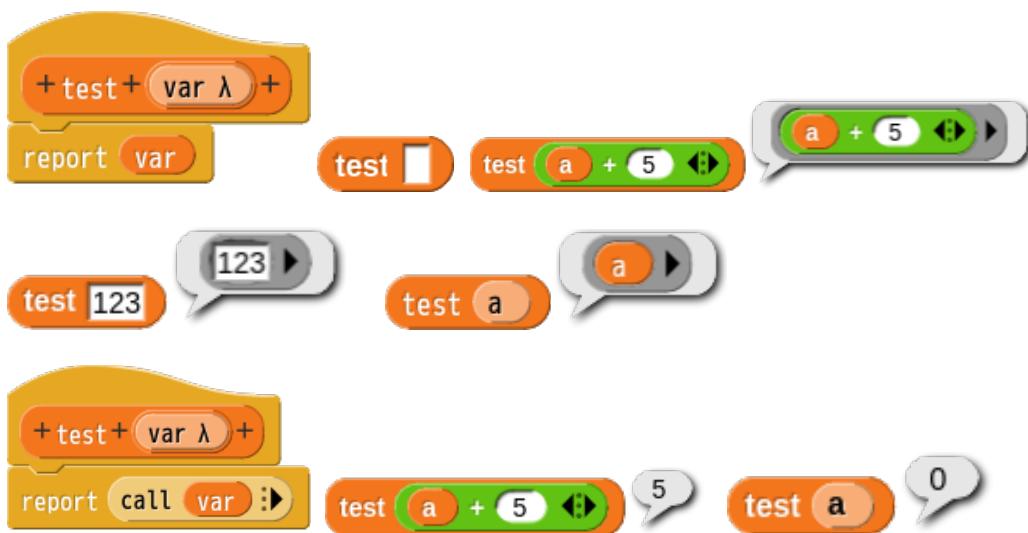
- 入力スロットにリングで囲ったものを扱う必要がある場合があります。使用するごとにリングで囲わせるのではなく、リングを装備したものが Reporter です。ただし、ドロップ入力のみです。



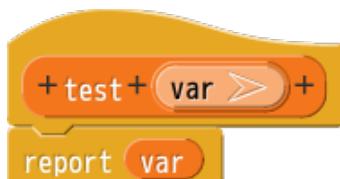
はなくなってしまいます。実際には call ブロックを使ってリングブロックの値を求めます。変数単体だと次のようになるので、場合によっては手動でリングを付ける必要があります。



- Reporter から外観上リングをなくし、キーボードから入力できるようにしたのが Any(unevaluated) です。unevaluated 評価されていない、つまり、値を求めるような操作はされていないということです。評価について… 数字の「1」を評価して 1 という数値を得るというような表現もするので、実行するというのとはちょっとニュアンスが違うようです。

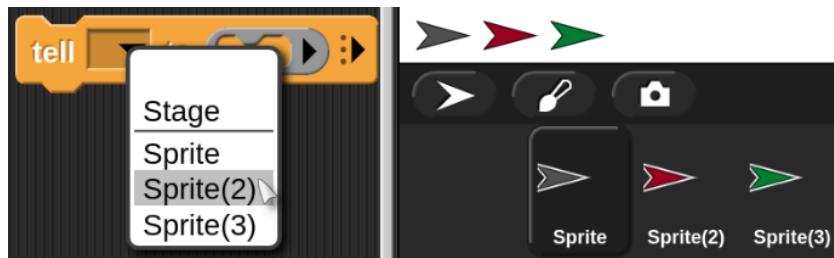


- special のオプションから object を選択して次のような定義にすると、**test >** というブロックができます。



入力スロットには、キーボードからなにかを入力することはできません。変数、スプライト、コスチューム、サウンドなどオブジェクトのドロップ入力を想定するものです。

既存のブロックでも tell ブロックのようにオブジェクトを指定するものがあります。Snap! の初期状態からスプライトを二つ複製して、tell ブロックの入力メニューを開くと次のように指定できるオブジェクトが表示されます。



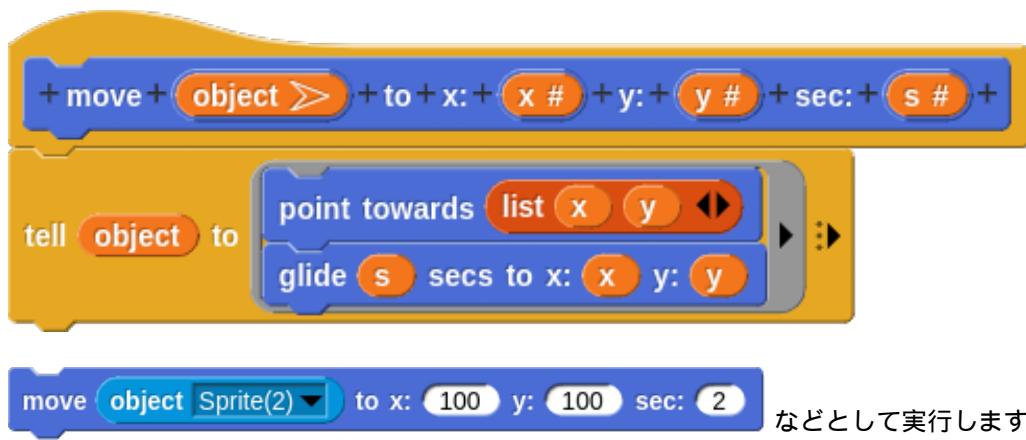
この入力スロットにオブジェクトを示す変数などをドロップすることもできます。Sensingパレットにある Object リポーターブロックの入力メニューを開くと tell ブロックと同様に指定可能なオブジェクトを選択することができます。



このブロックを tell ブロックの入力スロットにドロップすることができます。



Object を指定できるとそのオブジェクトに対して操作することができます。例として、オブジェクトが指定された位置に向かって方向を変え、指定された秒数で移動するブロックを作成してみます。方向を変えるには、移動先の x, y の位置をリストにして指定します。



などとして実行します。

### 7.3.2 Predicate 型

Predicate 型には、Boolean, Predicate, Boolean(unevaluated) があります。Reporter 型での Any type → Reporter → Any(unevaluated) の関係が、Predicate 型にも当てはまります。

- Boolean です。



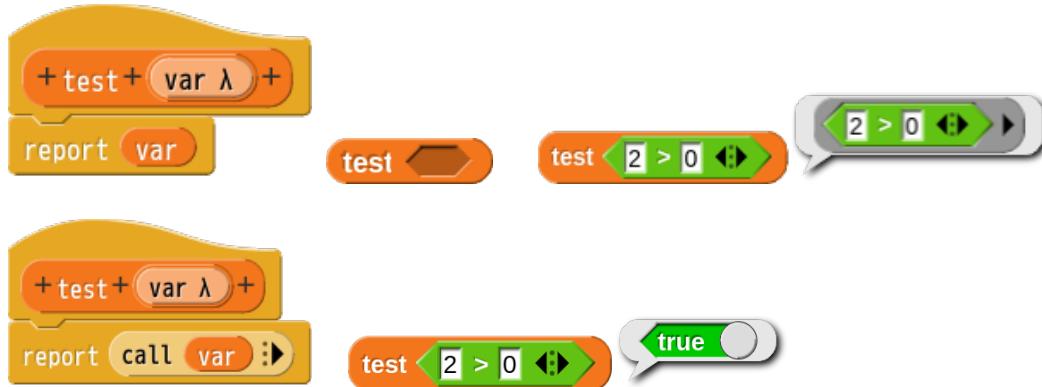
- Predicate です。（変数単体だと Reporter のようにリングが消えてしまうので注意が必要です。）



リングで囲われた Predicate 型変数をテストするには call ブロックを使って、評価し、真理値を求めます。



- Boolean(unevaluated) です。unevaluated つまり、評価されていない、値を求めるような操作はされていないということです。評価には call ブロックが必要です。



### 7.3.3 Command 型

Command(inline) は、ブロックの入力スロットに入れられたコマンドブロックを受け取るためのものです。Command ( C-shape ) は、if や for、repeat などのループで使われる C 型 (C の形をした) ブロックを作成するために使われます。Command ( C-shape ) を複数組み合わせると if else などの E 型 (E の形をした) ブロックが作れます。

Command(inline) 型と Command ( C-shape ) 型を使って、C 言語風の for ループを作つてみます。

c 言語では、

```
for (i = 0; i < 5; i++) {  
    printf("%d\n", i);  
}
```

のようにすると、0、1、2、3、4 と表示するプログラムになります。

(i = 0; i < 5; i++) のようにカッコの中がセミコロンで 3 つの部分に分けられています。

i = 0 の部分が初期設定で、初めに一回だけ実行されます。

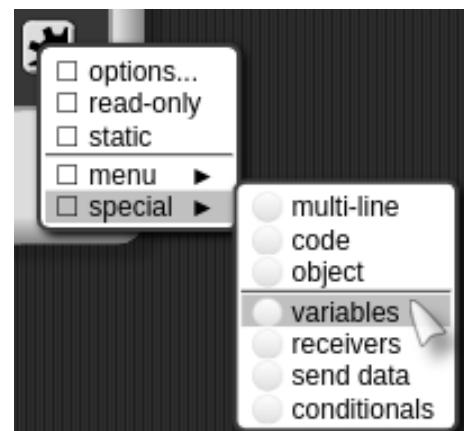
i < 5 の部分のテストが true なら繰り返しを続行します。

i++ の部分は次の繰り返しに進む前に実行されます。 i++ は i に 1 加算する処理をします。

{ } の内部がループの本体です。

以下が定義です。ループ内で使用する変数を定義しています。定義自体は run や call に丸投げするだけなので my for よりもシンプルですね。

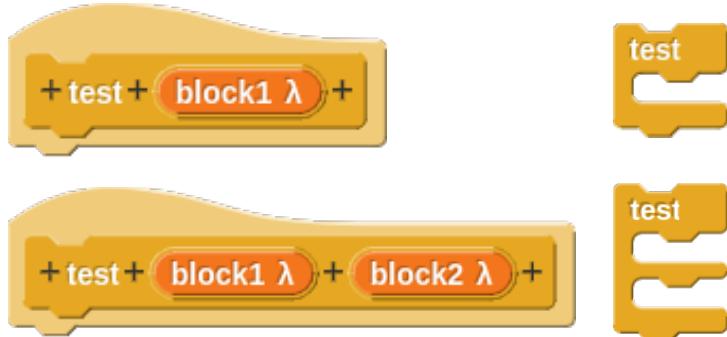
変数 i を作成します。設定で右下の歯車のようなアイコンをクリックすると右図のメニューがあるので、special から variables を指定します。変数名は i にしましたが、スクリプト変数と同じように a からいくつでも増やせますし、名前の変更もできます。



なお、〔初期化〕と〔継続処理〕は Command(inline) 型で、〔継続条件〕は Predicate 型です。〔実行本体〕は Command ( C-shape ) 型 です。ループ表示ボタンをチェックしてください。( 63 ページ参照 )



for ループは C 型が 1 つでしたが、2 つになると E 型になります。 if else の形ですね。 Command ( C-shape ) 型の変数を 並べればいくらでも増やせます。



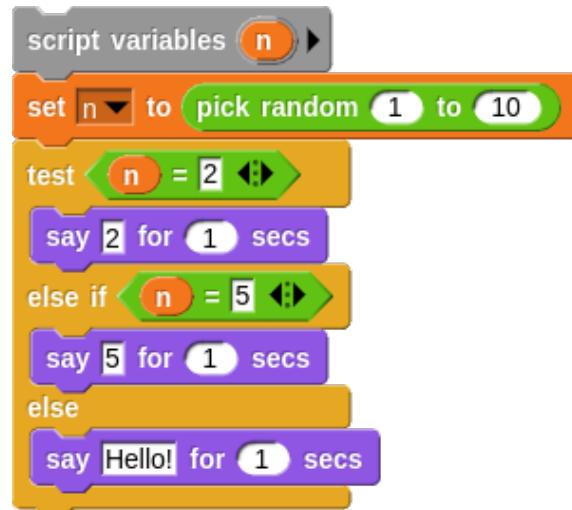
変数 block1 の前に Boolean(unevaluated) の変数を入れると、if else ブロックができます。

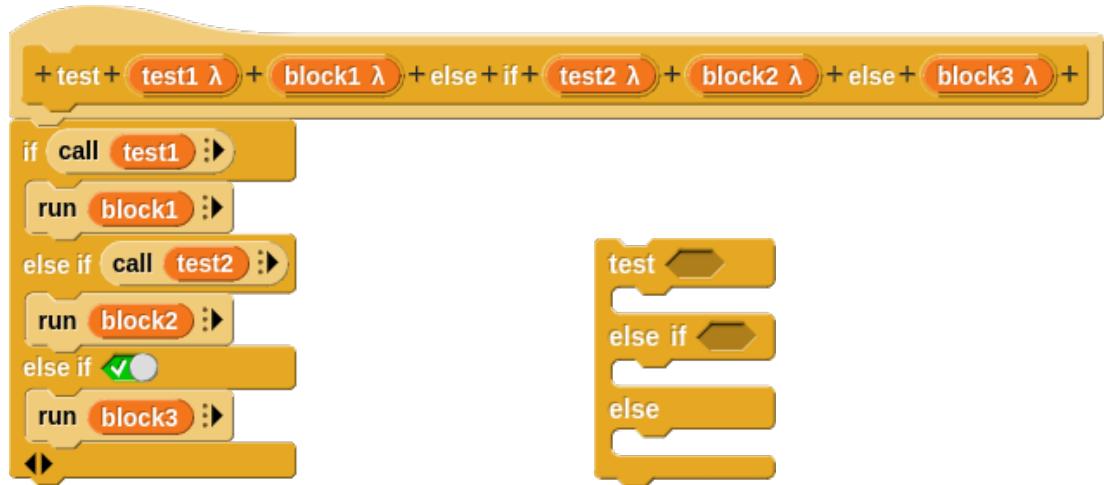


変数 block2 の前にも Boolean(unevaluated) の変数を入れ、block3 を追加すると変わった if else ブロックができます。



if else ブロックとしての体裁を整えてこんなふうに使えるようにしてみます。





C 言語などで利用できる switch case ブロックを作つてみます。

switch の入力スロットで調べる変数を指定して、case ブロックで指定した値と一致した場合にその処理をします。case ブロックは追加できるようにし、default 処理も指定できます。



case ブロックは単に指定された値と実行ブロックの対をリストとしてリポートするだけです。

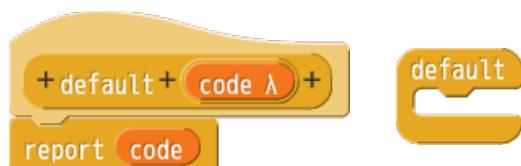


case ブロックは Reporter 型です。  
code は Command(C-shape) 型  
です。

変数「値」は Multiple inputs(value is list of inputs) にし、設定から次のように initial slots... (入力スロット数の初期値) を 1 にします。



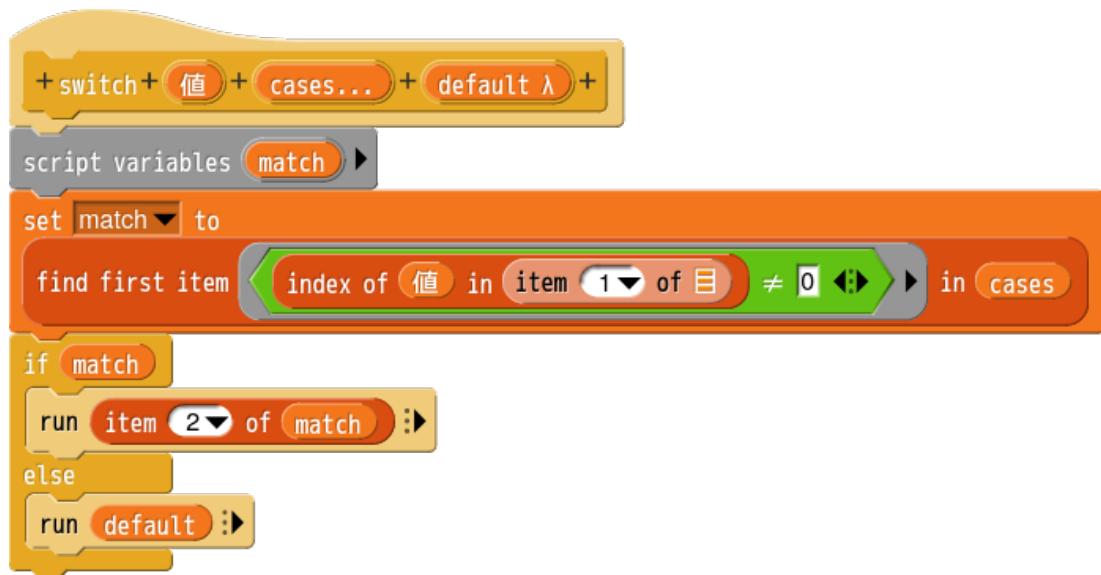
default ブロックは単に実行ブロックをリポートするだけです。



default ブロックは Reporter 型  
code は Command(C-shape) 型  
です。

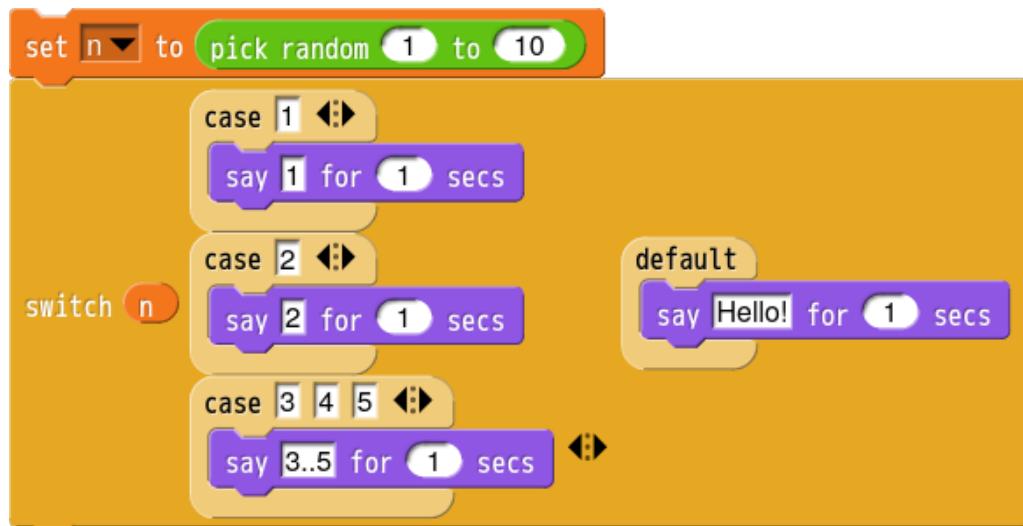
switch の定義ブロックです。cases 変数は Command(inline) かつ Multiple inputs(value is list of inputs) を使って追加できるようにしています。case の変数「値」のように初期入力スロット数を 1 にしてください。これには値と実行ブロックの対のリストが入っているので、switch で指定した値と case の値がマッチしたら指定されたブロックを実行します。マッチするものがなかったら default で指定されたブロックを実行します。default は Command(inline) 型です。

default は無指定、空のままでも大丈夫です。



名古屋文理大学 小橋 一秀 先生が公開されている【05. SNAP!でオブジェクト指向(4/4)】  
「継続を利用した switch case ブロックの作成例」を参考にさせて頂きました。

default ブロックは case ブロックの見た目に合わせるためのもので、これを使わずに実行ブロックを直接入力することもできます。



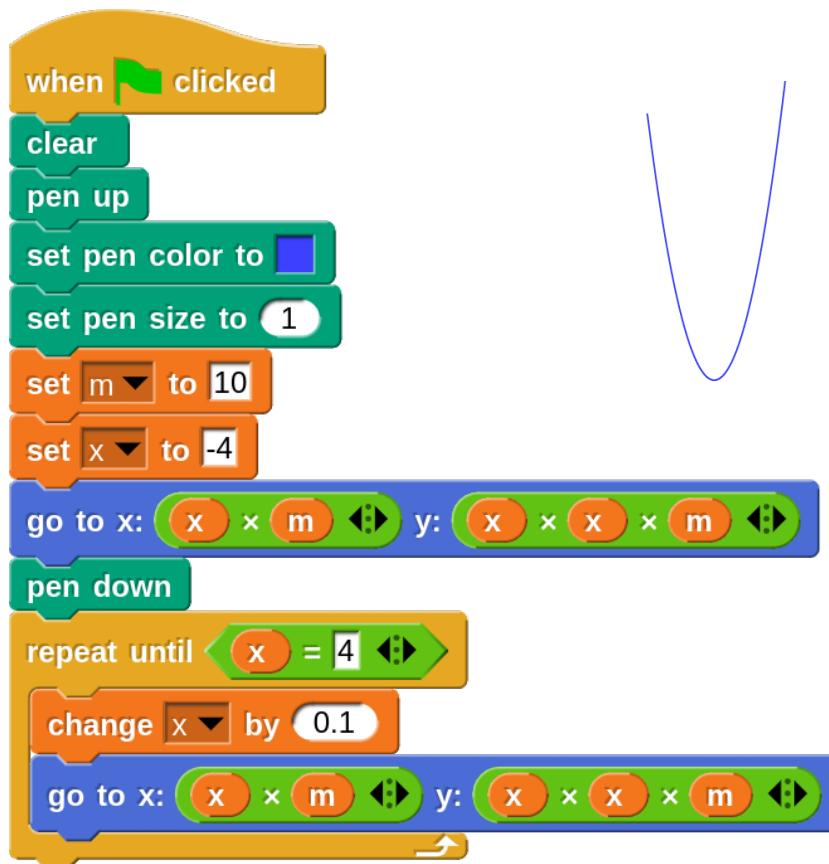
## 8 その他

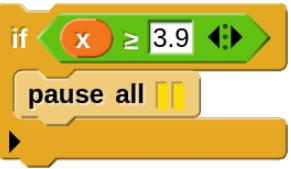
### 8.1 デバッグ

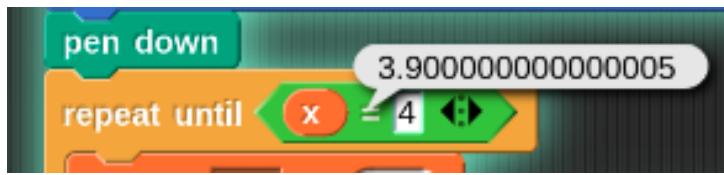
Snap! にはデバッグの機能が用意されています。

 ブロックにより指定の箇所でプログラムの実行を一時停止させることができます。  の操作でプログラムの実行スピードを遅くしたり、ステップ実行にしたりできます。 デバッグ中は実行中のブロックをハイライトしてくれます。 変数などの値もリポートしてくれます。  のクリックで一時停止した実行を再開できます。

次のスクリプトは  $x$  の値が -4 から 4 になるまで 0.1 ずつ増やしながらグラフを描くのですが、終了しません。



 を repeat until 内の最後に追加してから、スクリプトを直接クリックして実行させて下さい。 そうすると、 $x \geq 3.9$  の時点で一時停止します。  のように、スライダーを左端にしてステップ実行モードにします。  をクリックして 1 ブロックずつ実行しながら  $x$  の値を確認して下さい。



このように、 $x$  は 4 にはならないので無限ループになります。

repeat until の終了条件を  $x \geq 4$  にします。

次のように、Julia 言語で 0.1 を表示させてみます。「0.1」というのは十進数で表現された文字列です。コンピューター内部ではこれを二進数に変換して数値として扱います。println はそれを十進数文字列として丸めて 0.1 と表示します。printf を使うと、指定桁表示できます。

0.00000000001 のような小さな数から 10000000000 のような大きな数まで扱える仕組みが浮動小数点数ですが、十進数で表された小数点以下の値は二進数へ正確には変換できません。Snap! も同じ仕組みになっています。

```
julia> println(0.1)
0.1
```

```
julia> @printf("%.30f\n", 0.1)
0.1000000000000005551115123126
```



ボタンをクリックすると、デバッグモード のオンオフができます。

ただし、 のボタンで実行すると、変数などの値をリポートしてくれません。直接スクリプトをクリックして実行してください。

ユーザー定義ブロック内についてもデバッグする場合は、デバッグモードにしてからユーザー定義ブロックを edit で表示させれば見ることができます。

通常は edit で開くと一番下に (Ok) (Apply) (Cancel) が表示されますが、デバッグモードでは (OK) だけ表示されます。

ユーザー定義ブロックのデバッグは必要ないならば、表示させなければユーザー定義ブロック内は通常速度で実行されます。

## 8.2 getc

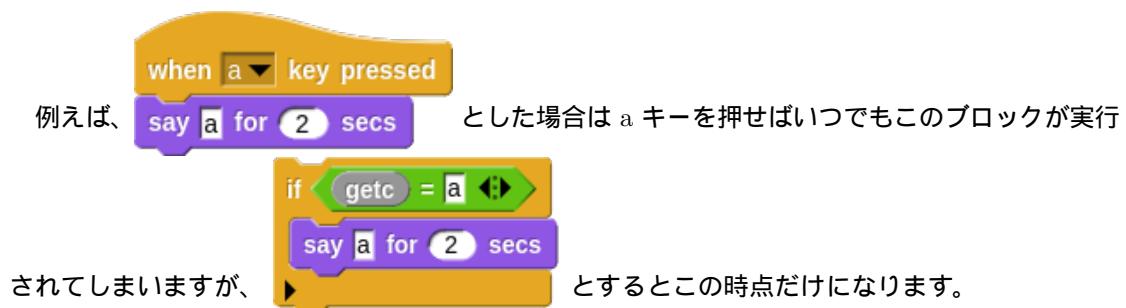
何かのキーが押されたかを調べるために必要な数の when ( ) key pressed ブロックでチェックするのにはたいへんです。



ですが、any key を使うと一つで済みます。any key を選択すると、押されたキーが Upvar 変数の key にセットされます。次のような getc ブロックを作成するとさらに使い勝ってが良くなります。getc は押されたキーを \_@key に記録します。\_@key はどこでも使っていい変数名ということで決めたものです。



getc を使うことの利点として、キー入力は getc を呼び出した時だけ有効になります。



文字列入力の例です。( say ではカーソルとして下線 \_ を表示しています。)



### 8.3 = と identical

二つの値が同じかどうかを調べる ですが、

や

を同値として扱うと不都合な場

合があります。

その場合に使えるのが、 です。

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。

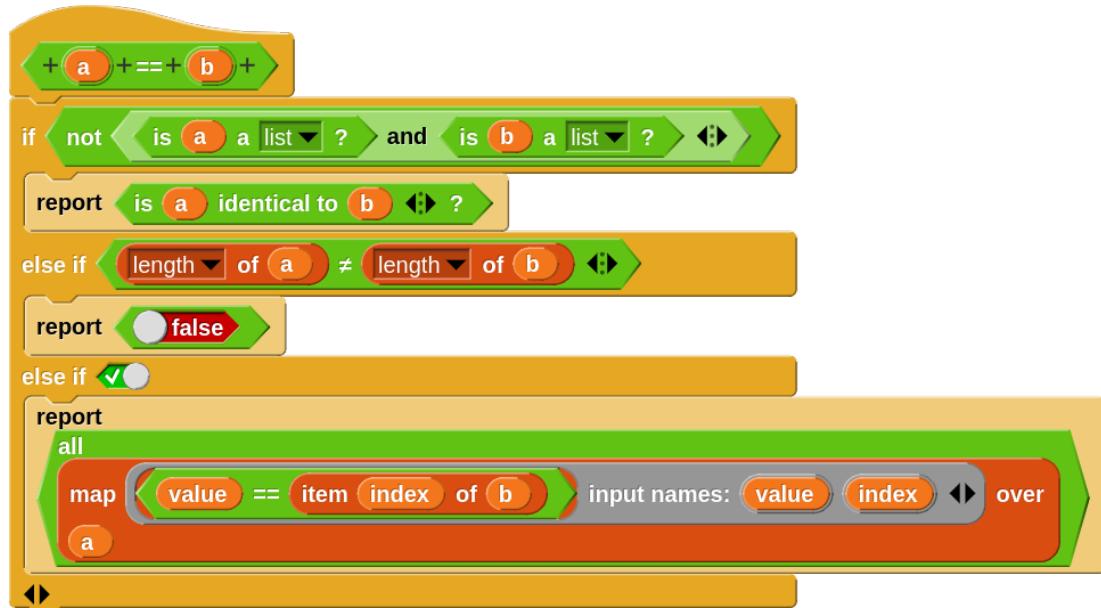
$x$  と  $a$  は同じリストを参照しています。

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。

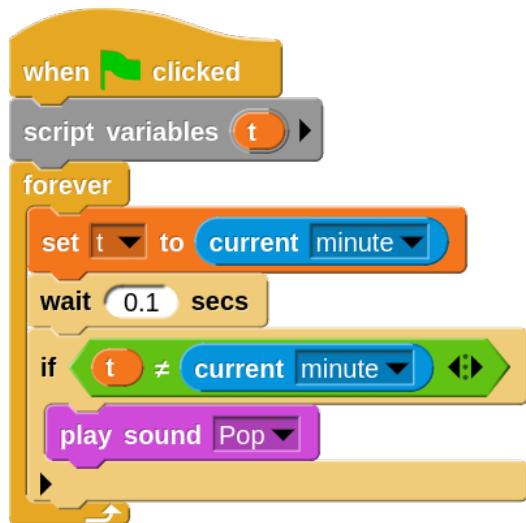
「`identical`」ではなく「`==`」としてみました。

スクリプトです。a と b は Any type 型です。map を使うとすべての要素をテストすることになりますが速いです。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。( 143 ページ参照 ) 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 ( 95 ページ参照 )



## 8.4 Event Hat ブロック

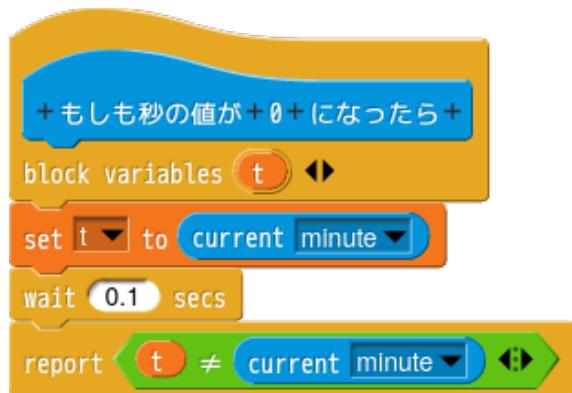
例えば1分ごとに音を鳴らす場合、次のようなやり方があります。



Event Hat ブロックを使うと、1分経過したというイベントを発生させ、指定の処理を行うことができます。まず、make a block から Event Hat ブロックを選択して名前をつけます。



期待する状態になった時に true を返すスクリプトを作成します。

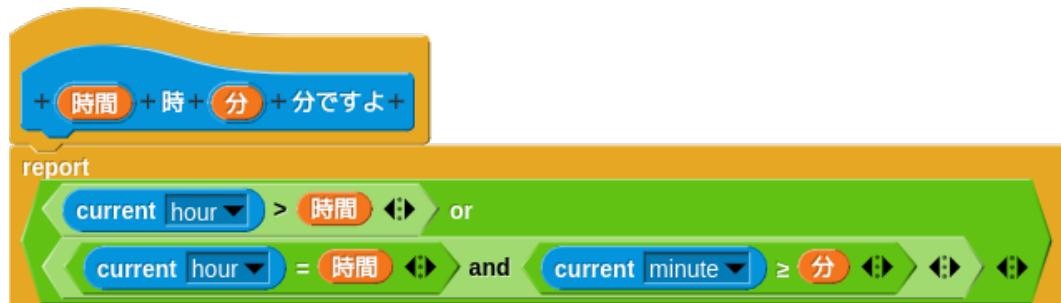


音を鳴らしたいならば次のようにします。これはイベントブロックなので  のボタンをクリックする必要はありません。

もしも秒の値が 0 になったら

play sound [Pop ▾]

指定した時間になつたらイベントを発生させるブロックです。



20 時 0 分ですよ

play sound [computer\_bleeps1 ▾]

いくつかのキー入力に対して同じ処理を行いたい時には、次のようにできます。

ハットブロックの色は選んだパレットと同じになります。



key は Multiple inputs (value is list of inputs) にしてください。or ブロックの右端三角マークに map ブロックを入れると any ブロックになります。(95 ページ参照)

次のようにすると、カーソルキーか エンターキーか スペースキーか バックスペースキーか シフトキーか コントロールキーか a を押した時に表示させることができます。

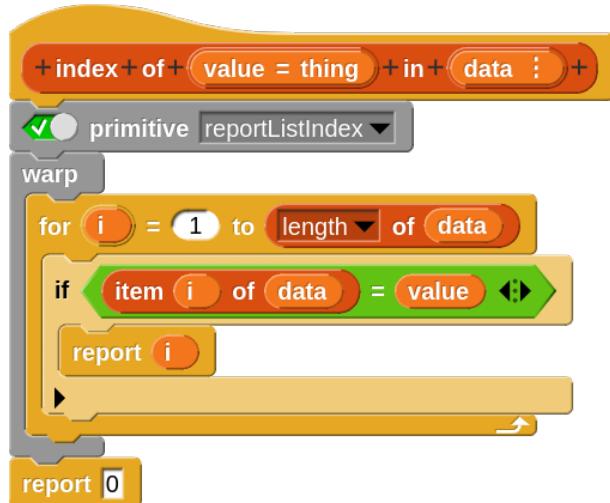
もしも up-arrow down-arrow enter space backspace shift control a のどれかが押されたら

say [Hello! for 2 secs]

## 8.5 プリミティブブロックの編集

ver. 10 から、ハットブロックを除くプリミティブブロックの編集が可能になりました。

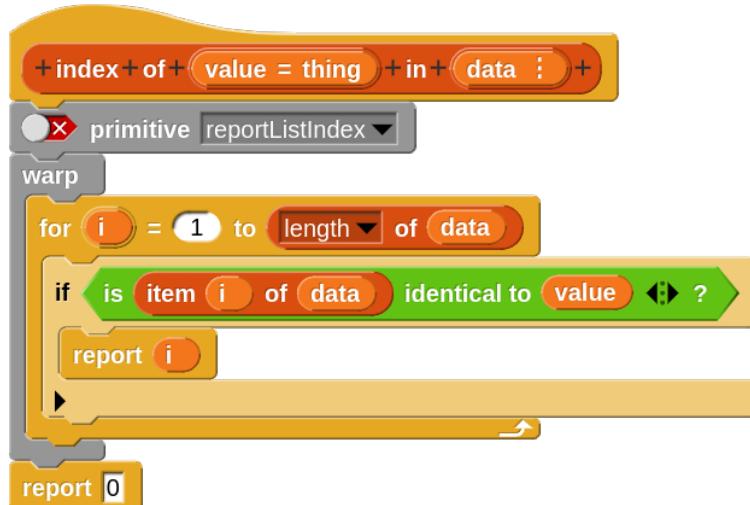
**index of thing in 目** を edit で開くと次のようなスクリプトになっています。primitive のスイッチによってプリミティブのコードか、指定されたスクリプトを実行するかを決定します。スイッチがオンなので、この場合プリミティブのコードが実行され warp 以降のスクリプトは実行されません。



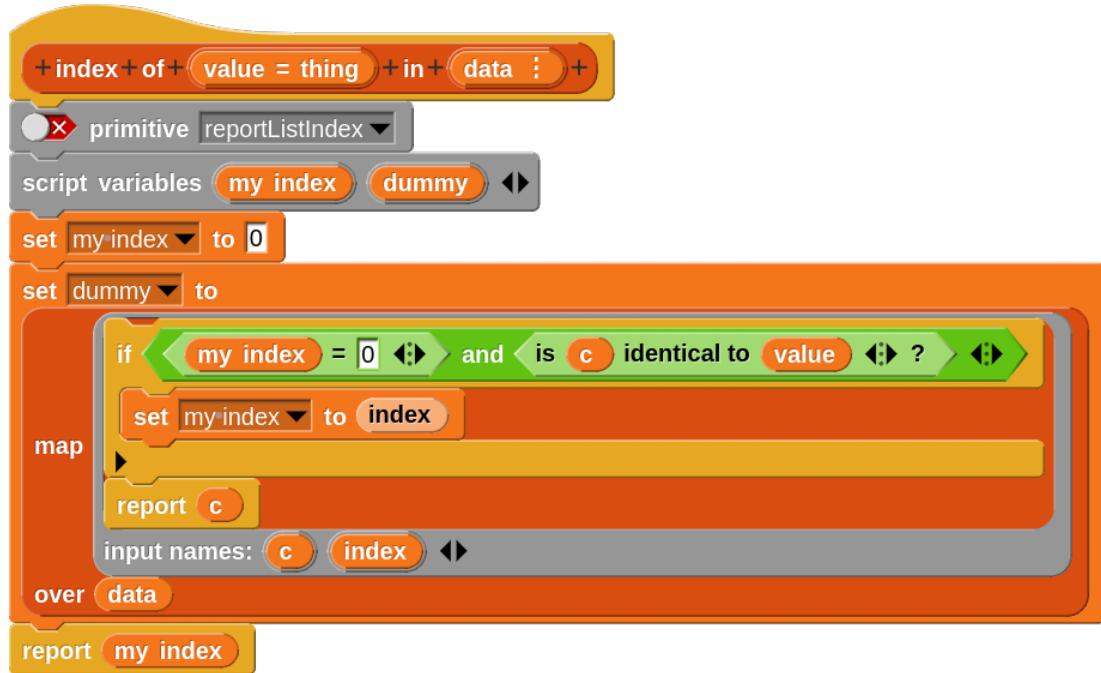
このスクリプトだと、アルファベットの大文字小文字を区別しないため大文字を指定しても小文字にヒットしてそのインデックスを返してしまいます。



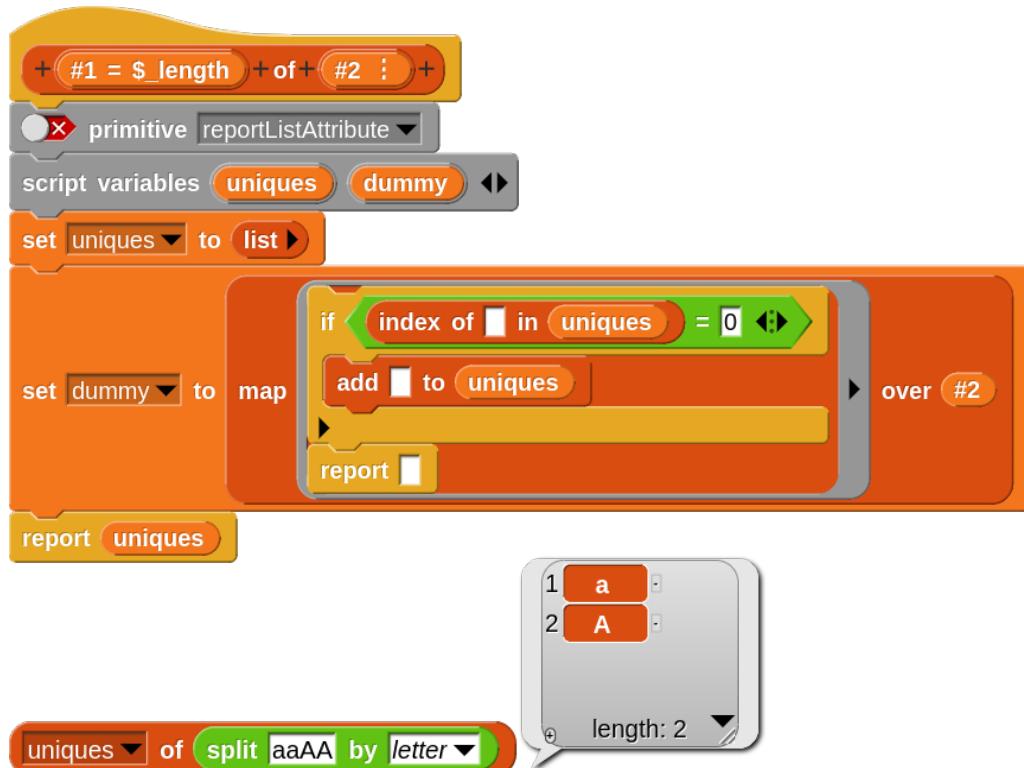
大文字小文字を区別するにはイコールブロックをリラベルして、プリミティブスイッチをオフにします。



map を使用して次のようにすることもできます。



プリミティブの `uniques` ブロックは大文字のアルファベットを小文字に変更してしまいます。上記のように `index` ブロックが変更されると、`uniques` ブロックも大文字小文字を区別するようになります。



ただし、安易に仕様変更するとそのブロックを使用している他のスクリプトで不具合が発生する場合があるかもしれません。

## 8.6 連想配列、辞書

他の言語では連想配列や辞書など、検索用の語であるキーとデータを対にして記録する仕組みがあります。ID と名前のように。ID を指定すれば、そのデータをリポートしてくれます。

find first item ブロックを使えばこの仕組みを真似することができます。キーとデータの対をリストにして、その集まりを一つのリストとします。

The image shows a Scratch script consisting of five blocks:

- set table [list] to [ ]
- insert [list v1] [one] at [1] of [table]
- insert [list v2] [seven] at [1] of [table]
- insert [list v3] [five] at [1] of [table]
- insert [list v4] [three] at [1] of [table]

To the right of the script is a table labeled "table" with four rows:

	A	B
1	3	three
2	5	five
3	7	seven
4	1	one

データをリストに加える順序は任意です。キーとなる値は他のものと同じでないならば、数値でも文字列でもかまいません。

The image shows a Scratch script:

- + look up + [key] + [list : ] +
- report
- pipe find first item [item 1 of ] = [key] in [list]
- if [value = ] [then [ ] else [item 2 of value] input names: [value] ]

Below the script are two speech bubbles:

- look up [3] table three
- look up [6] table

キーの部分にアルファベットを含む場合に、大文字小文字を区別して検索したいならば find first item の条件部分を次のように変更する必要があります。

The image shows a Scratch script:

find first item [is item 1 of ] identical to [key] ? in [list]

find first item ブロックはリストの先頭から順に検索しているようなので他言語のもののように効率的ではありませんが、for ブロックを使って検索するよりも速いです。

## 8.7 read-only 入力スロット

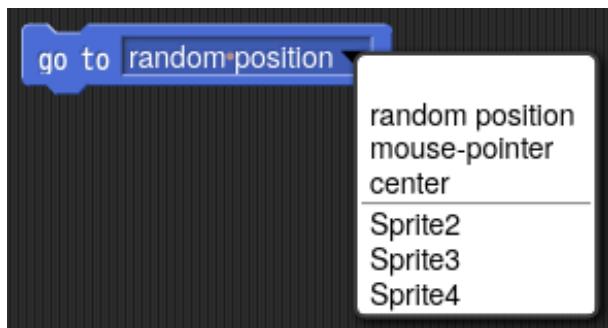
次のように背景が白い入力スロットはキーボードからの入力を受け付けます。



それに対して、次のようなブロックはその時点で利用可能な入力の中から選択することになります。



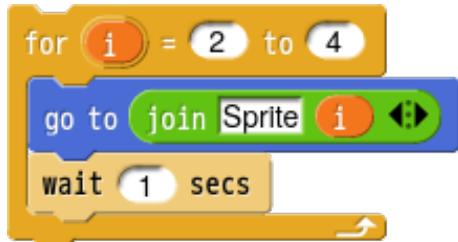
例えば、Sprite, Sprite2, Sprite3, Sprite4 がある場合、次のようにになります。



キーボードからの入力は受け付けませんが、変数などのリポーターブロックはドロップすることができます。変数 a にこれらの引数を入れてやると、同様に動作します。



これを利用すると次のような使い方ができます。



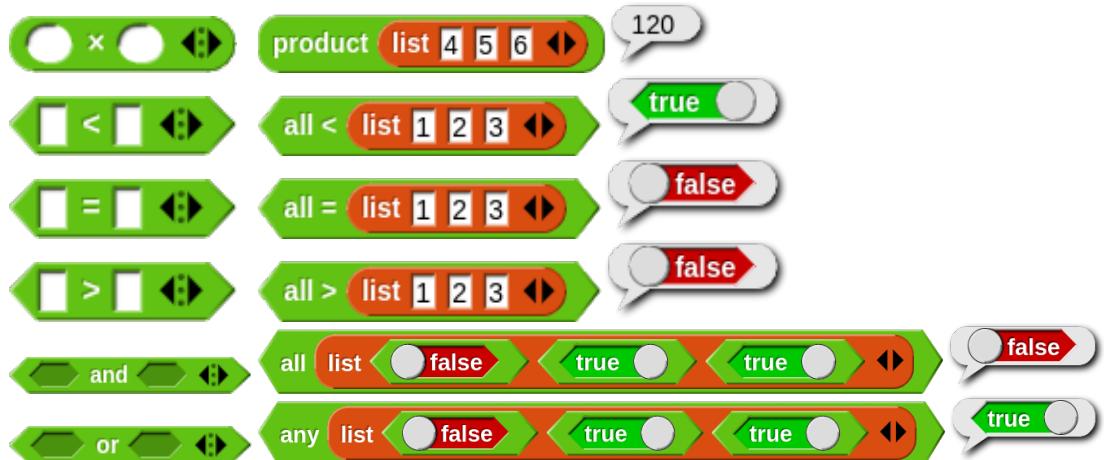
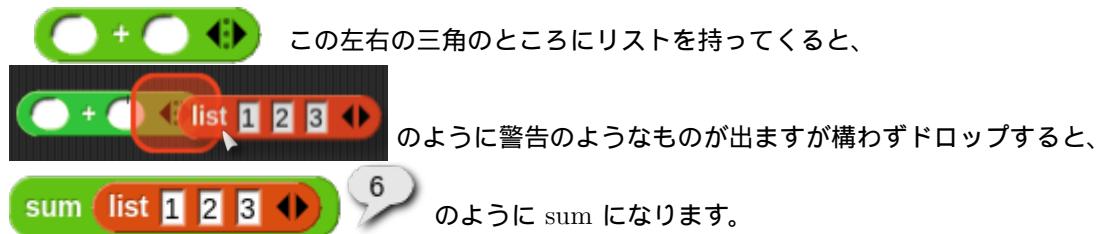
乱数を使って次のようにすることもできます。



このやり方は他の tell や switch to costume などでも有効です。

## 8.8 入力スロットへのリスト指定

次のように、ブロックによっては右端に左右の三角表示があって入力スロットを増減できるものがあります。

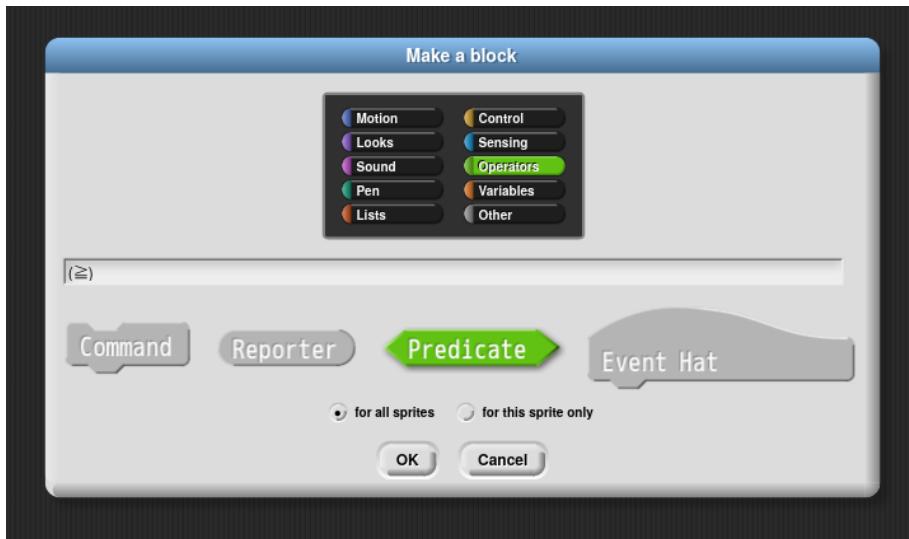


xy 座標を次のようにリストにした場合、

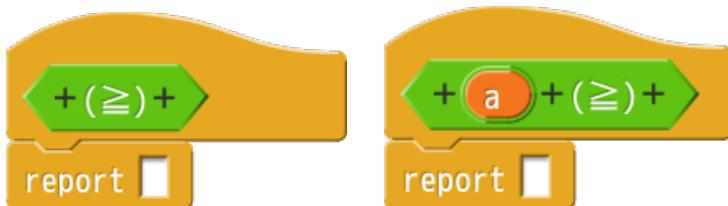


## 8.9 入力スロットの追加

50 ページで ( ) ( ) のブロックを作成しました。  ブロックのように、スロットを追加したりリストを指定できるようにしてみます。次のようにしてプロトタイプ部分を作成して下さい。



a という変数を作成します。



変数 a の設定を変更してスロットを追加できるようにします。まず、 Multiple inputs (value is list of inputs) を指定します。次に、



右下の歯車のようなアイコンをクリックすると右図のメニューがあるので separator, collapse, initial slots の項目をそれぞれ変更します。

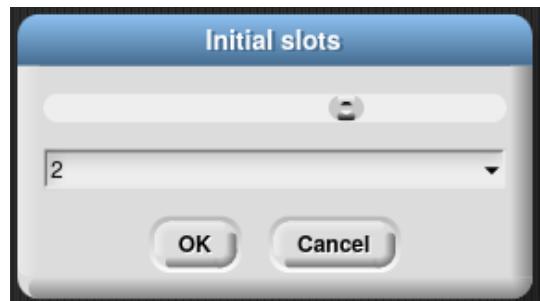
Separator は各スロットの間に入る項目です。この場合は「」になります。全角文字「きごう」とか「だいなりいこーる」で変換すると出できます。



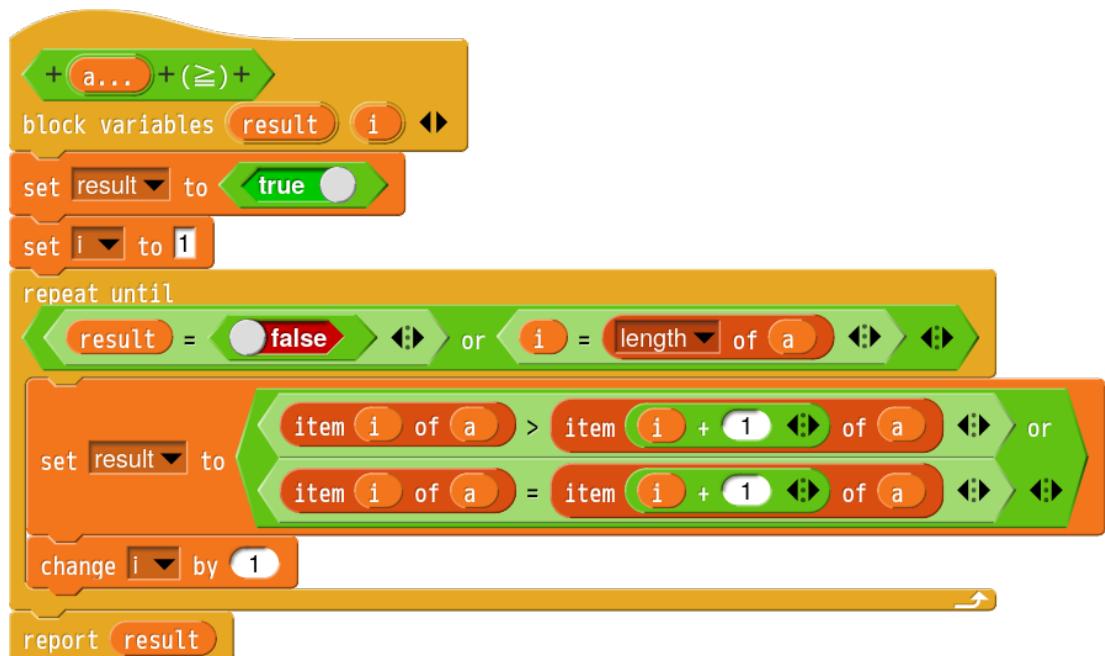
Collapse はリストを入力して、スロットがない場合の表示項目です。この場合は、「all >」になります。



Initial slots はデフォルトのスロット数です。この場合は、2 になります。



変数 a の表示が変更されます。ブロック定義の本体は次のようにします。



( ) を削除するとプリミティブブロックのような表示になりますが、続けて のブロックを作成した場合にプロトタイプが同じになるので (2) が表示されてしまいます。なにか方法があるのかもしれません。

## 8.10 ask

ask what's your name? and wait

リストで指定したメニューからクリックで項目を選択することができます。

ask list メニュー1 メニュー2 メニュー3 ◀◀ and wait

メニュー1  
メニュー2  
メニュー3

リストを組み合わせるとタイトル(説明文)を付けたり、サブメニュー構造にすることもできます。

ask list タイトル list メニュー1 メニュー2 メニュー3 ◀◀ ◀◀ and wait

タイトル

メニュー1  
メニュー2  
メニュー3

ask list list メニュー1 list メニュー1.1 メニュー1.2 ◀◀ ◀◀  
list メニュー2 list メニュー2.1 メニュー2.2 ◀◀ ◀◀ ◀◀ and wait

メニュー1

メニュー2

メニュー1.1

メニュー1.2

script variables レベル 選択肢 ◀◀

set 選択肢 ▼ to list 1.初級 2.中級 3.上級 ◀◀

ask list レベルの選択 (選択肢) ◀▶ and wait

say join item 2 of split answer by . ▶▶ 「が選択されました」 ◀◀ for 2 secs

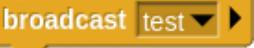
set レベル ▼ to index of answer in 選択肢

say レベル for 2 secs

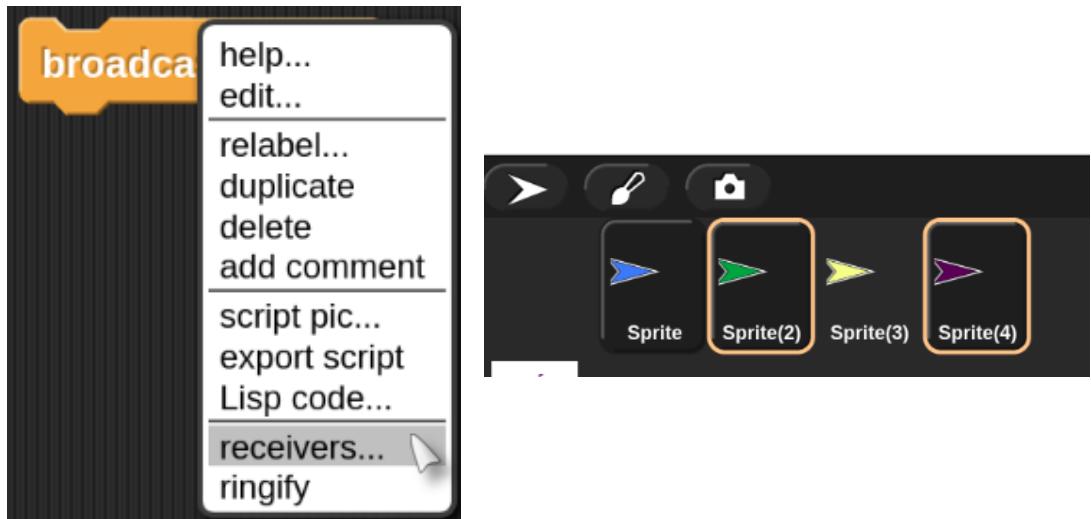
レベルの選択

1.初級  
2.中級  
3.上級

## 8.11 broadcast の検索オプション

例えば、Sprite で  を使用していて、

 を使用しているスプライトを知りたい時には receivers... オプションを使用すると、スプライトコラルの該当スプライトをハロで示してくれます。



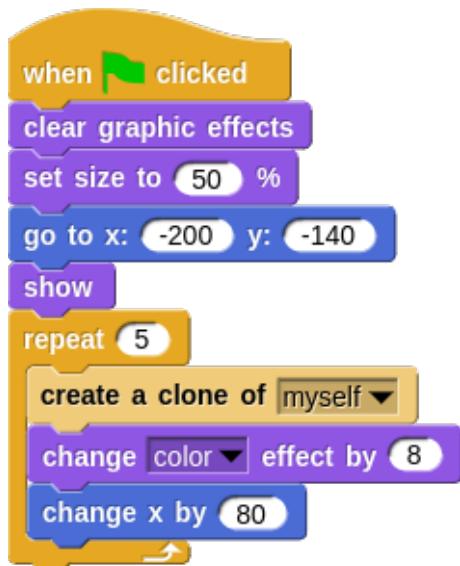
## 8.12 クローン

Snap! には、テンポラリクローンとパーマネントクローンがあります。

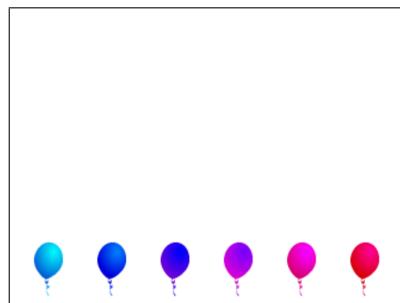
### 8.12.1 テンポラリクローン

テンポラリクローンは Scratch でのクローンと基本的には同じものです。

次のようにして風船のクローンを作成してみます。



このスクリプトでは [ show ] により、作成された 5 個のクローンが左から表示され、右端にクローンではない本体も表示されるので合計 6 個の風船が出現します。

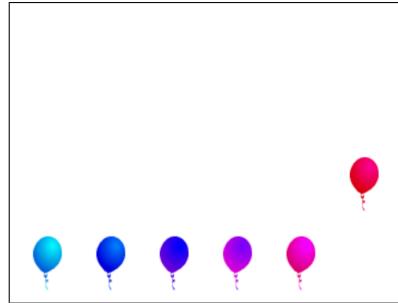


```

when green flag clicked
  clear graphic effects
  set size to 50 %
  go to x: -200 y: -140
  show
repeat (5)
  create a clone of myself
  change color effect by 8
  change x by 80
repeat (10)
  change y by 10

```

このようにして風船を移動させるスクリプトを追加すると、本体だけが移動します。つまり、本体用のスクリプトではクローンの操作はできません。普通は本体を存在させると都合が悪いので、非表示にしてクローンのみを操作します。



```

when green flag clicked
  hide
  clear graphic effects
  set size to 50 %
  go to x: -200 y: -140
repeat (6)
  create a clone of myself
  change color effect by 8
  change x by 80

```

表示は [ when I start as a clone ] 内で行います。

```

when I start as a clone
  show

```

特定のクローンを操作するには、イベントを利用します。タッチイベントの例です。

```

when touching mouse-pointer? 
  repeat (36)
    change y by 10
  delete this clone

```

```

when I start as a clone
  show
forever
  if touching mouse-pointer?
    repeat (36)
      change y by 10
    delete this clone

```

イベント処理スクリプト内ではなく [ when I start as a clone ] 内で行うことできます。

```

when green flag clicked
  hide
  clear graphic effects
  set size to 50 %
  go to x: -200 y: -140
  set balloon to [list v]
  repeat (6)
    add (a new clone of myself) to balloon
    change color effect by 8
    change x by 80
  forever
    for each item in balloon
      tell item to
        for (i = 1 to 180)
          change y by (cos of i)
      wait (5) secs

```

when I start as a clone

show

Snap! では作成したクローンをリストなどの変数に入れることができるので、それを使って対象を指定して操作することができます。

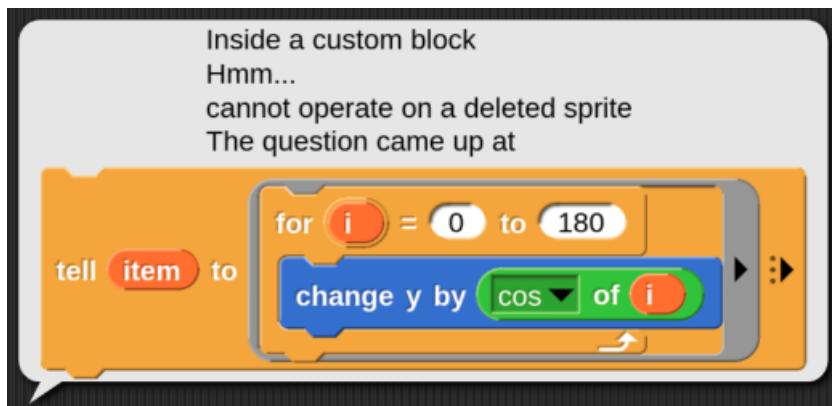
```

when I am clicked
  repeat (36)
    change y by (10)
    delete this clone

```

左のスクリプトを作成し、上記スクリプトの実行中に風船をクリックすると、そのクローンが削除されるので次のようなエラーが起ります。

クローンを格納したリストの要素が削除済みということで、「×」になっています。「削除されたスプライトに対して操作することはできない」というエラーになります。



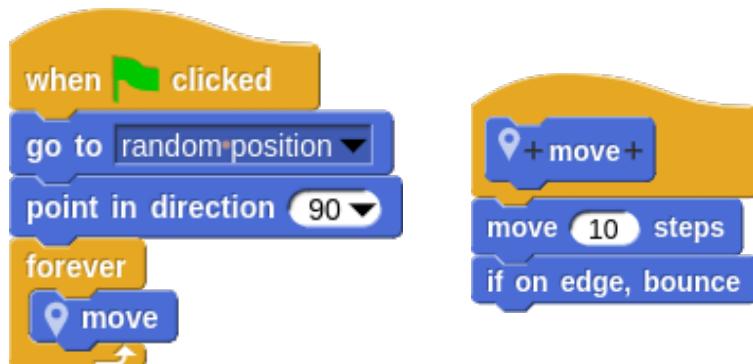
変数やリストに格納されたクローンが delete されているかどうかをテストするためのブロックは定義されていないようです。これに対処する一つの方法として、スプライト変数  にクローン番号を記憶させて、そのインデックスによって削除されたこと false を balloon リストに記録します。( クローン 1 は 1、クローン 2 は 2、... の値の変数 id を持ります。) id 番号をリスト内の位置にしているので、リストから要素を削除することはできません。



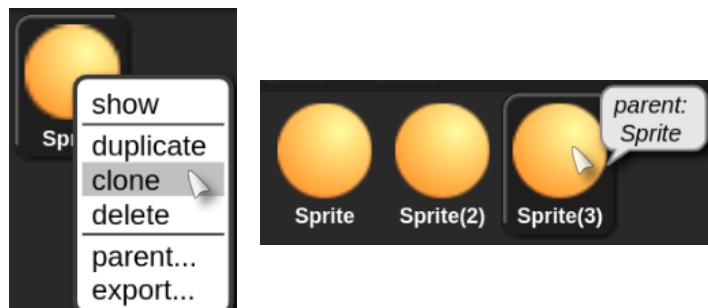
### 8.12.2 パーマネントクローン

パーマネントクローンはテンポラリクローンとは違って、 をクリックしても消滅しません。作成したクローンとは親と子の関係になり、親側のスクリプトの変更が子側に反映されます。ユーザー定義ブロックを `for this sprite only` で作成すると子側で定義を変更できるので、同じブロック名でそれぞれクローンごとに違った動作をさせることができます。

ボールを使います。`ball` のコスチュームをインポートしてください。次のようにスクリプトを作成します。ユーザー定義ブロック `move` を `for this sprite only` でローカルな定義として作成します。変数を作成する場合はローカルな変数を使用します。



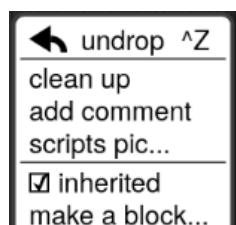
スプライトコラルで `Sprite` を右クリックして、クローンを二つ作成します。作成されたクローンのところにマウスカーソルを置くと、`parent:` 親が `Sprite` であることを表示します。



クローンである `Sprite(2)`, `Sprite(3)` には、`Sprite` と同じスクリプトがコピーされています。親である `Sprite` のスクリプトを変更すると、子である `Sprite(2)`, `Sprite(3)` のスクリプトに反映されます。例えば、サイズを 50 などと変えてみてください。

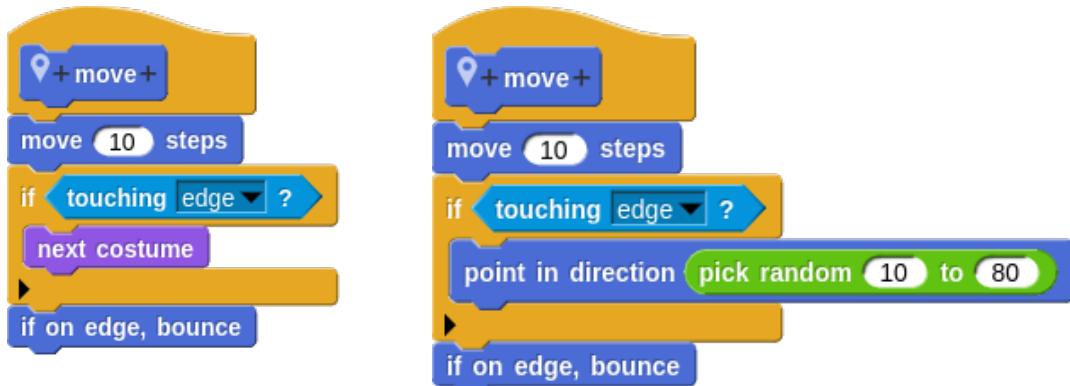
スクリプトエリアで右クリックすると右図のように `inherited` がオンになっています。この状態だと子側も親側のスクリプトと同じになります。

しかし、子側でスクリプトを変更してしまうと `inherited` がオフになり、親側の変更が反映されなくなってしまいます。その場合は `inherited` をオフにしてやれば、親側と同じスクリプトに戻ります。

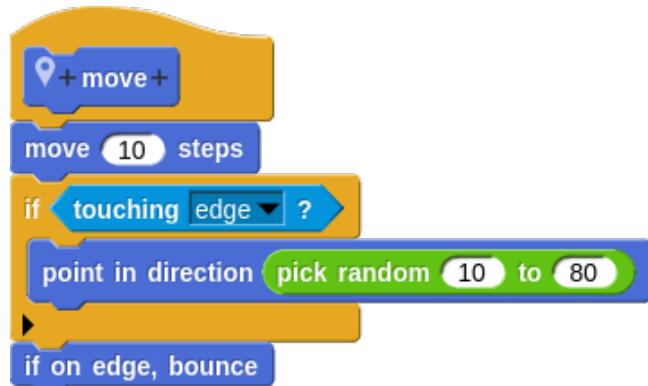


クローンを作成した時に親側のユーザー定義ブロックも子側にコピーされています。しかし、この `move` ブロックは `for this sprite only` で作成されているので、親側で変更しても子側は変わりません。子側で変更しても `inherited` の状態は変わりません。次のように子側の `move` 定義を変更してください。各スプライトで同じ名前の `move` ブロックですが、それぞれ違った動作をさせることができます。

Sprite(2)



Sprite(3)



### 8.13 flat line ends

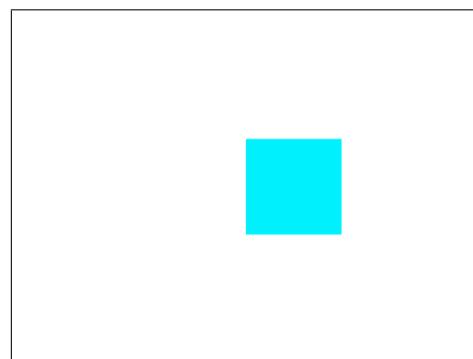
Sensing のパレットに `set [video-capture v] to [ ]` があります。

これを flat line ends にして六角形のスロット部分をクリックして、`set [flat-line-ends v] to [ ]` にすると、ペンで描く線の端を平ら (true) にすることができます。false で丸くする指定になります。

```

clear
hide
pen up
go to x: (0) y: (0)
set [flat-line-ends v] to ( )
set pen color to (blue)
set pen size to (100)
pen down
go to x: (100) y: (0)
pen up

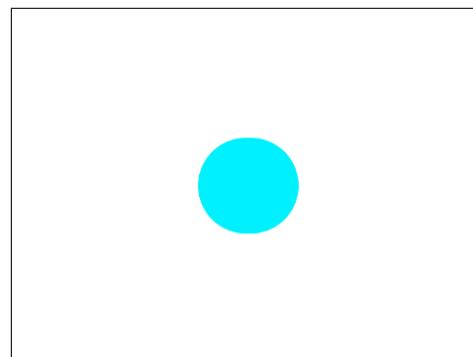
```



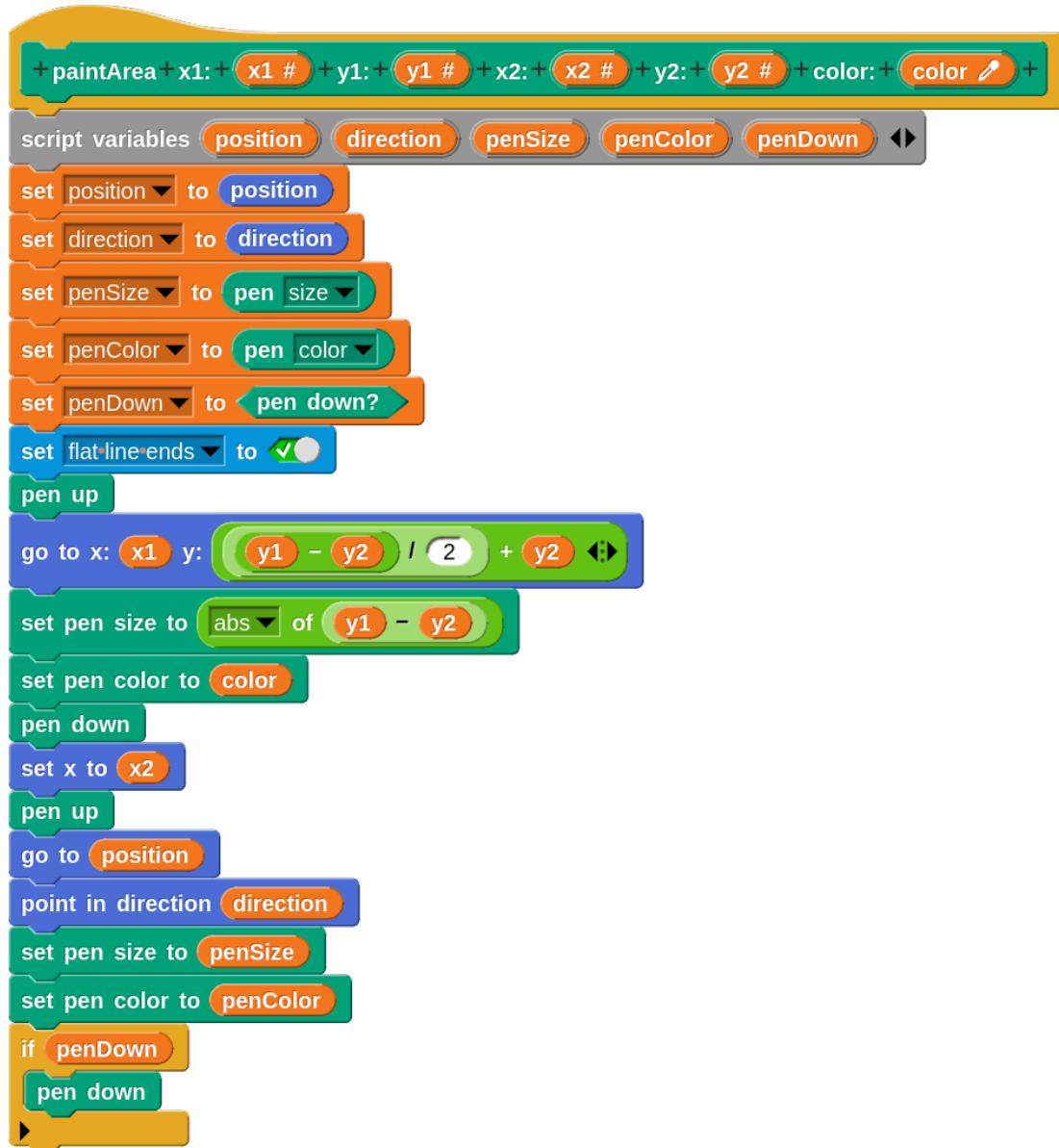
```

clear
hide
pen up
go to x: (0) y: (0)
set [flat-line-ends v] to (false)
set pen color to (blue)
set pen size to (100)
pen down
go to x: (5) y: (0)
pen up

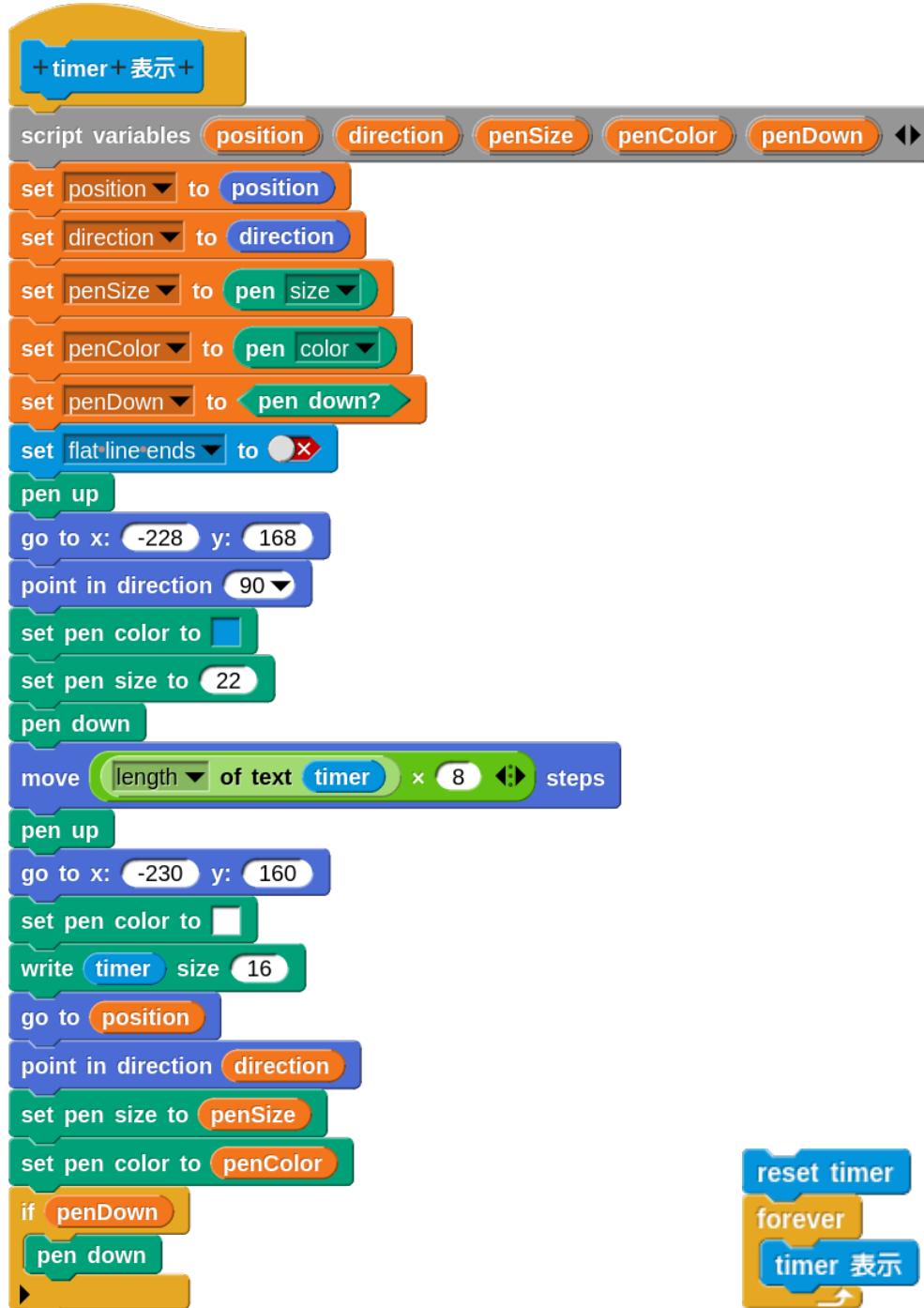
```



対角の座標を指定して、そのエリアを塗りつぶす定義ブロックです。Input name オプションで Color を選択すると、定義したブロック上で選択した色を利用することができます。背景の色を指定するとクリア指定になります。他のスクリプトに影響を与えないようにブロック内変数にスプライトの位置や角度の情報とペンの情報をキープさせています。



プログラムの実行中に何かデータを表示するには変数ウォッチャーを使ってリポートすることができますが、timer などのように更新周期が短いデータなどには表示が追隨できない場合があります。次のように Pen 機能を使用すると、timer 表示の代わりとすることができます。青でエリアを表示していますが、これは更新された値を表示するために表示されていた timer の値を消す働きもしています。



## 8.14 角度

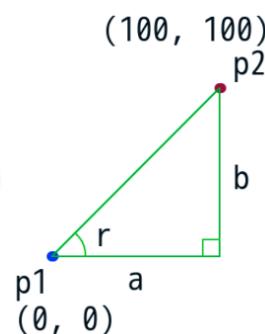
スプライトがある方向に向けたい場合、**point towards** Sprite(2) のようにすれば他のスプライトの方向に向けられます。ある座標に向けるならば **point towards** list Px Py のように指定できます。

直接角度を指定する時は **point in direction** 90 を使いますが、このブロックで指定する角度は右向きが 90 度で上向きが 0 になっていて、一般的な角度の指定方法とは違っています。一方、Snap! に用意されている sin や cos などの三角関数用のブロックでは右向きが 0 度の一般的な指定方法になっています。(ラジアンではありません)

**point in direction** 90  
**turn** r degrees または、  
**point in direction** 90 - r とします。

右図で角度 r は  $\text{atan}(\frac{b \text{の長さ}}{a \text{の長さ}})$  で求めることができます。

**atan** of p2y - p1y / p2x - p1x = 45

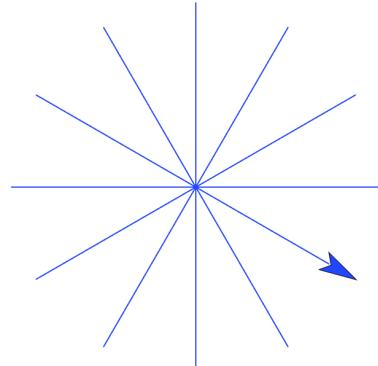


atan を使用すると象限によって向きが逆になるので atan2 を使用します。

```

clear
set pen color to blue
set pen size to 1
set r to 0
for i = 1 to 12
  pen up
  set Px to 150 * cos of r
  set Py to 150 * sin of r
  go to center
  pen down
  point in direction 90 - atan2 Py - 0 / Px - 0
  glide 1 secs to x: Px y: Py
  change r by 30
pen up

```

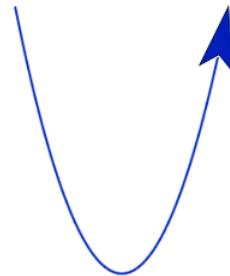


方程式のグラフを描く場合、微分によって接線の傾きが求められるのでスプライトの角度に利用することができます。

```

clear
pen up
set pen color to blue
set pen size to 1
set x to -2.5
set m to 20
go to x: (x * m) y: (x * x * m)
pen down
repeat until (x > 2.5)
  change x by 0.1
  point in direction (90 - atan2 (2 * x) / 1)
  go to x: (x * m) y: (x * x * m)
pen up

```



```

clear
pen up
set pen color to blue
set pen size to 1
set x to -180
set m to 100
go to x: (x) y: (sin of 0 * m)
pen down
for (r) = 0 to 360
  point in direction
    (90 - atan2 (sin of (r + 1) - sin of r) * m / 1)
  go to x: (r) y: (sin of r * m)
  change x by 1
pen up

```

$$\frac{100}{0} \quad \text{Infinity}$$

$$\text{atan of } \frac{100}{0} \quad 90$$

$$\text{atan2 } \frac{100}{0} \quad 90$$

割り算の分母に 0 は使用不可ですが、  
atan や atan2 では有効です。

## 8.15 anchor アンカー

スプライトに関して、ボディーとパーツを別に用意してくっつけるというやり方があります。そうすると、パーツをボディーの一部として付帯しながらパーツ自体の動きをさせることができます。



二つのスプライトを用意し、alonzo と balloon のコスチュームをそれぞれに設定してください。

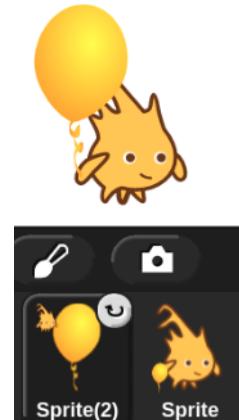
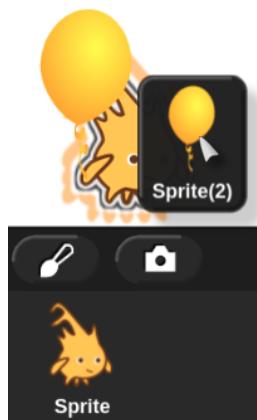
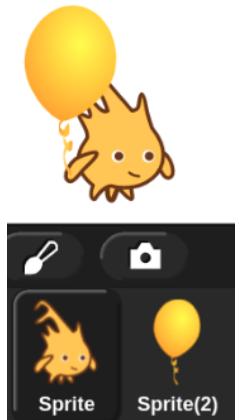
ステージ上で alonzo の手に風船を持たせてください。



balloon をエディターで開いて、動きや回転の中心を糸の端にします。

alonzo 側の中心位置変更の必要はありません。

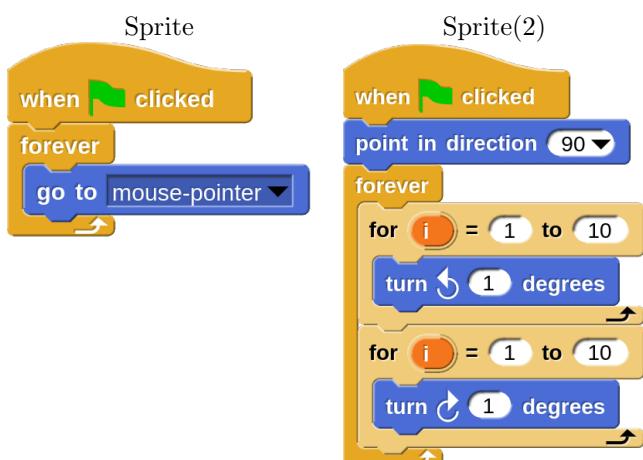
アンカー操作には二つのやり方があります。コラールにある風船のスプライトをステージ上の alonzo のスプライトのところに持っていくとハロがでます。ドロップするとコラールのスプライトの表示が変化します。

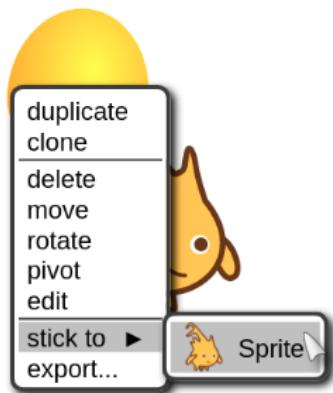


テストスクリプトです。

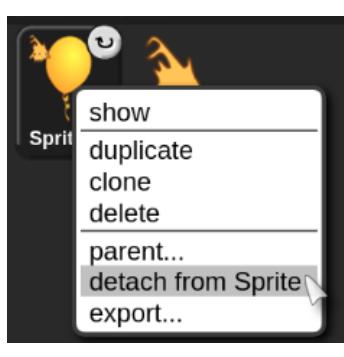
alonzo は動く風船を持ちながらマウスポインターに付いて行きます。

alonzo のサイズを変更すると、連動して風船のサイズも変更されます。





アンカー操作のもう一つのやり方は、ステージ上の風船のスプライトのところで右クリックすると出てくるオプションから stick to で allonzo を選択する方法です。



アンカーを解除するには、ステージ上かコラールにある風船のスプライトを右クリックして、オプションから detach from Sprite を選択します。

アンカーの機能は興味深いものですが、[ if on edge, bounce ] で向きを変えたりすると不具合が出たり、複数のスプライトで利用したりすると重くなります。

## 8.16 JavaScript function (オプション 6 ページ参照)

Snap! には、JavaScript コードを実行させるブロックがあります。次の例は JavaScript 側から数値を返すものです。

```
call [JavaScript function ( ) { return 10; } ] → 10
```

次のようにすると数値をやり取りすることができます。call の入力スロットに入れたものが JavaScript の入力スロットに設定されて JavaScript 側からアクセスできるようになります。

```
call [JavaScript function ( n ) { return n; } with inputs 20 ] → 20
```

```
call [JavaScript function ( n ) { return n * n; } with inputs 20 ] → 400
```

変数を使って演算をしてそれを返すこともできます。

```
call [JavaScript function ( n ) { var a = n; return a * a; } with inputs 20 ] → 400
```

テキスト、文字列を扱うこともできます。

```
call [JavaScript function ( ) { return "abc"; } ] → abc
```

```
call [JavaScript function ( s ) { return s + s; } with inputs abc ] → abcabc
```

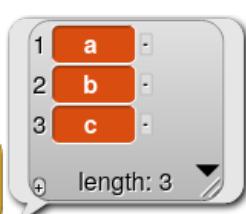
```
call [JavaScript function ( s ) { return s.length; } with inputs abc ] → 3
```

しかし、JavaScript の配列は Snap! のリストとは別物です。次のように配列を返すことはできません。

```
call [JavaScript function ( ) { return [1,2,3]; } ] → «1»
```

受け取ったリストをただ返すだけならば問題ありません。

```
call [JavaScript function ( list ) { return list; } with inputs list a b c ] →
```



Snap! のリストを JavaScript 側で操作するには `asArray()` で配列に変換する必要があります。配列は参照型で、操作の結果はリストに反映されるのでリストを返してやります。

The first example shows a script block:

```
call JavaScript function (list) {
  var l = list.asList();
  l[1] = "x";
  return list;
}
with inputs list [a b c]
```

A message box shows the list after modification:

1	a
2	x
3	c
length:	3

The second example shows a script block:

```
call JavaScript function (list) {
  return list.asList().length
}
with inputs list [1 2 3]
```

A message box shows the length of the list:

length:	3
---------	---

The third example shows a script block:

```
call JavaScript function (list) {
  var l = list.asList();
  l.sort(function(a,b){
    return b-a);
  });
  return list;
}
with inputs list [1 2 3]
```

A message box shows the sorted list:

1	3
2	2
3	1
length:	3

JavaScript 側で作成した配列を Snap! 側に返すには必ずリストとして返さなければならぬので、データとしては必要なくともリスト引数を渡す必要があります。

次の例は、1 から 5 の二乗のリストを返すものです。

The script block contains the following code:

```
script variables [list]
  set list to []
  for [i = 1 to 5]
    add (join (i ^ 2) to list)
  report list
```

A message box shows the resulting list:

1	$1^2 = 1$
2	$2^2 = 4$
3	$3^2 = 9$
4	$4^2 = 16$
5	$5^2 = 25$
length:	5

上記のスクリプトでは call 側からリング内のスクリプトへ引数を渡す必要はありません。JavaScript では、リストを扱うためと `list = []`； としてしまうと `list` への参照がなくなってしまうことから、空のリストを渡すことには重要な意味があります。

The script block contains the following code:

```
call JavaScript function (list)
  var l = list.asList();
  for (var i = 1; i < 6; i++) {
    l.push(i + "² = " + i * i);
  }
  return list;
```

A message box shows the resulting list:

1	$1^2 = 1$
2	$2^2 = 4$
3	$3^2 = 9$
4	$4^2 = 16$
5	$5^2 = 25$
length:	5

Snap! の bitwise library にはビット演算をするブロックがありますが、JavaScript の機能を使って作ってみます。

十進数では一桁を 0 ~ 9 の数値だけを使用して、一桁で表せない時は桁上がりして表記します。上位の桁はその桁の  $\times 10$  であり、十進数の 123 は  $1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$  という意味になります。二進数では一桁を 0 と 1 の数値だけを使用します。上位の桁はその桁の  $\times 2$  であり、二進数の 111 は  $1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$  という意味になります。

二進数では一つの桁をビットと言い、4 ビットをニブル、8 ビットをバイトと言います。

主要なビット演算として、ビット反転 (NOT)、論理積 (AND)、論理和 (OR)、排他的論理和 (XOR) があります。2 つの 1 ビットの数 n1, n2 が取り得る値 0, 1 に対してのそれぞれのビット演算の結果を示します。

n1	n2	AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

論理積 (AND) は両方の値が 1 の時だけ 1、  
論理和 (OR) は片方だけでも 1 ならば 1、  
排他的論理和 (XOR) は双方が違う値ならば 1 になります。

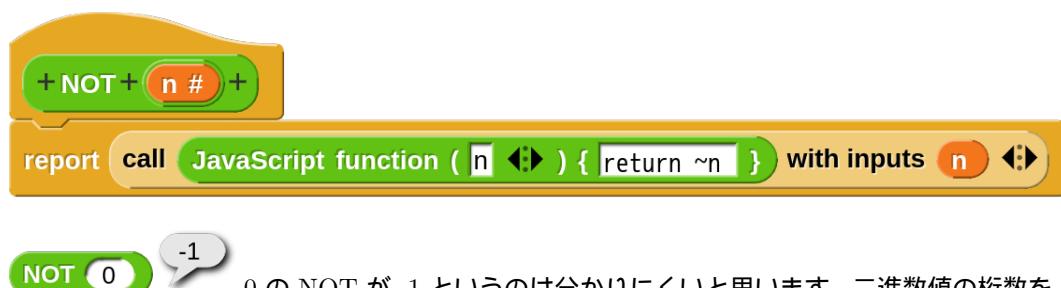
JavaScript には論理積 (AND) 演算子として "&"、論理和 (OR) 演算子として "|"、排他的論理和 (XOR) 演算子として "^"、ビット反転演算子 (NOT) として "~" があります。

AND ブロックの定義です。n1 と n2 が整数であることを前提にしています。



"&" の部分を "|" にすれば OR 演算、"^" にすれば XOR 演算になります。

NOT ブロックです。NOT はビット反転です。つまり、そのビットが 0 ならば 1、1 ならば 0 に反転させます。



0 の NOT が -1 というのは分かりにくいと思います。二進数値の桁数を 4 桁、4 ビット (ニブル) に限定してみます。すると表せる数は 0000 ~ 1111 (十進数では 0 ~ 15) になります。0000 の NOT は 1111 になります。二進数では、最上位ビットをサインビット (符号ビット) として使用することで負数も扱えるようにしています。その場合、1111 は最上位ビットが 1 なので負数です。1111 + 0001 を行うと桁上がりをして 4 ビットの範囲では 0000 にな

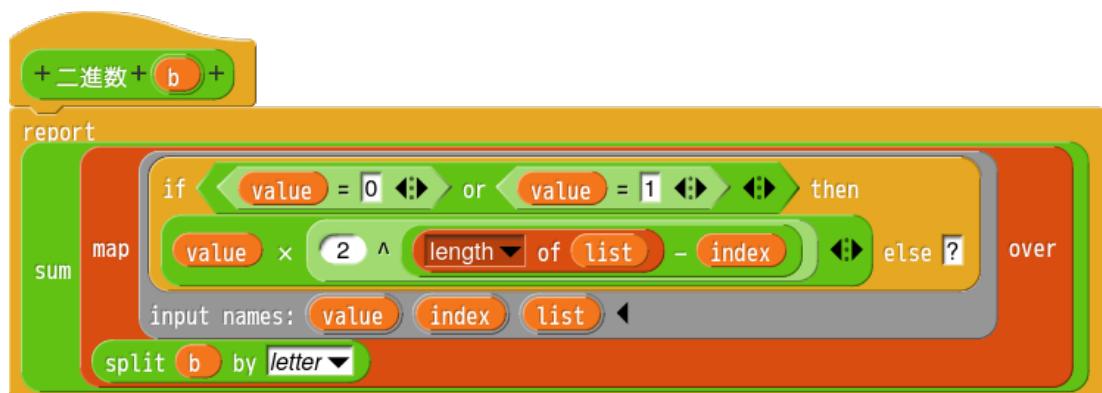
ります。つまり、1111 は 0001 に加えると 0000 になる数である十進数の -1 ということになります。二ブルで表せる符号付き数は 1000 ~ 0111、十進数の -8 ~ 7 になります。十進数の 0 と -1 を 64 ビットの二進数で表すとそれぞれ 0 と 1 が 64 個並んだものになります。

整数 n に対して NOT を取り 1 を加えると、 $n \times (-1)$  の値を得ることができます。



二進数で数値を指定するリポーターブロックを作成してみます。b の入力のタイプは text です。split で指定するのは letter です。

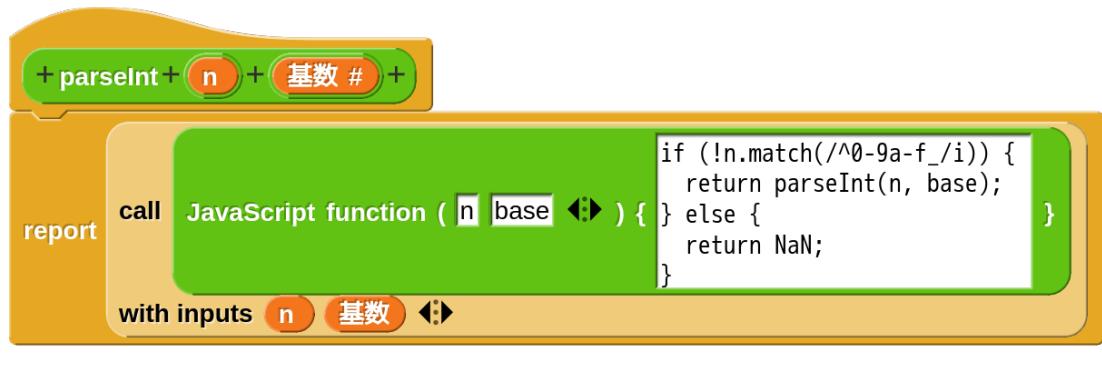
二進数 1100 の場合、 $1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$  なので、それぞれの桁ごとにエラーチェックをしながら map で十進数に変換して、得られたリストを sum で合計します。



二進数では 0 と 1 しか使ないので、それ以外は「?」に置き換えてエラーになるようにしています。



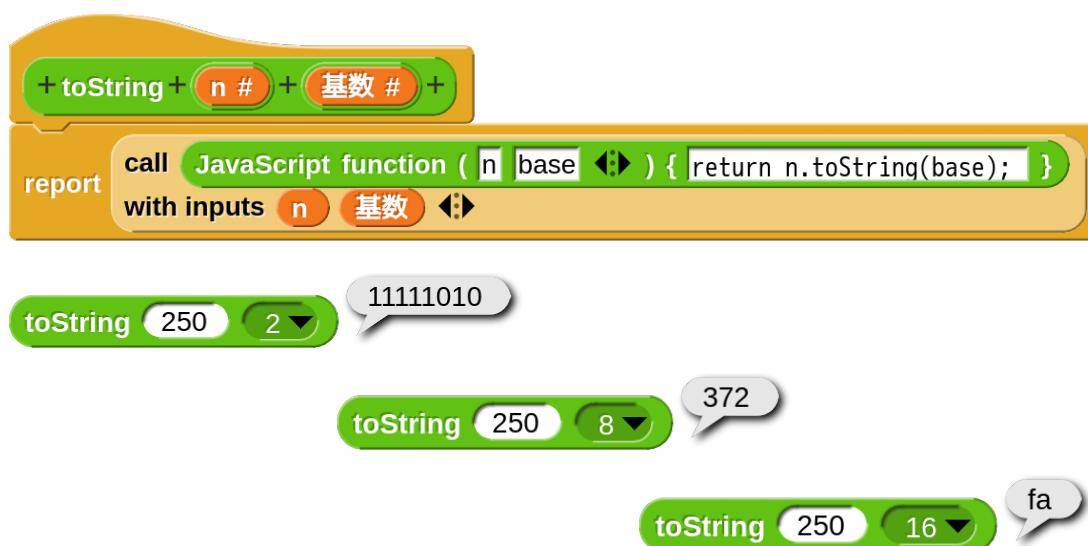
JavaScript には文字列を整数に変換する parseInt 関数があります。2 番目の引数で基底を指定できます。2 で二進数、16 で 16 進数の文字列からの変換になります。なお、「base」と「基底」が合っていませんが、JavaScript の引数は名前ではなく inputs のスロット位置に対応した値が設定されます。



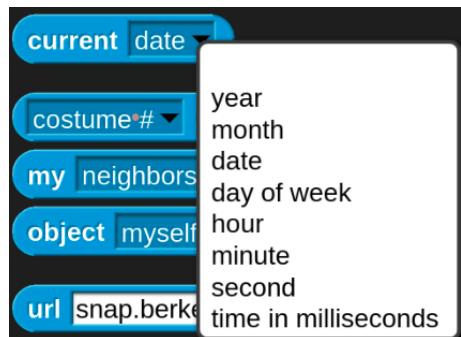
こうなると、数値を二進数表示するブロックも必要です。どうせなら基底を指定して文字列にする定義ブロックにしてみます。ただし、対応しているのは 2, 8, 10, 16 進数などです。



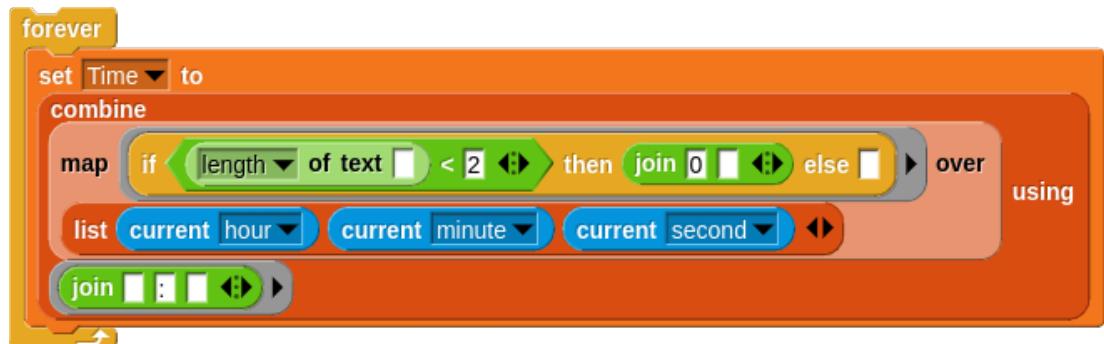
JavaScript では `toString` がそれをしてくれます。



## 8.17 時計



Sensing のところには日時を取得できるブロックがあります。これを使って、時、分、秒のデータを表示すればデジタル時計ができます。各データを 2 衔表示にして、区切り文字を挿入するには次のようなスクリプトになります。これを変数にセットして、forever でループします。



時間のデータを各針の角度に変換すればアナログ時計ができます。針はスプライトで表現してもいいですし、その都度ペンで clear 描画を繰り返してもいいです。

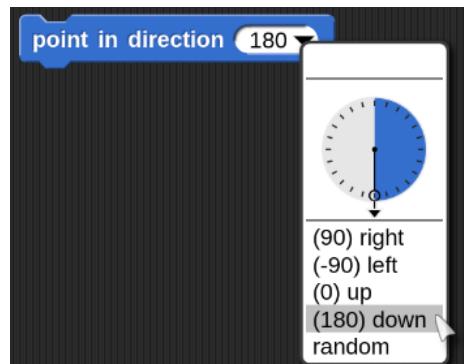
秒数は、**current second** で求められます。

秒針をスムーズに動かしたければ、**current time in milliseconds** を使用すればできます。このブロックがリポートする値は、1970 年 1 月 1 日からの経過秒数です。( UTC 協定世界時 ) milliseconds とあるように、1/1000 秒単位の値になります。この値を 1000 で割って秒単位の値にします。それを 60 で割った余りを求めれば秒数が得られます。



秒数をアナログ時計の秒針の角度に変換するには  $\text{秒数} \times 360(\text{度}) \div 60(\text{秒})$  になります。

30 秒の場合、 $30 \text{ 秒} \times 360(\text{度}) \div 60(\text{秒}) \Rightarrow 180 \text{ 度}$  です。



秒針の角度です。( 60 秒で 1 回転 )



秒単位の時間を 60 で割って分単位の時間にしてから、それを 60 で割った余りを求めれば分数が求められます。

set minute ▾ to UTC / 60 mod 60

分針の角度です。( 60 分で 1 回転 )

point in direction minute × 360 / 60 ◀▶

秒単位の時間を  $60 \times 60$  で割って時単位の時間にしてから、それを 24 で割った余りを求めれば時数が求められます。ただし、これは協定世界時なので日本とは 9 時間の時差があります。この値で補正してしまうと日本でしか通用しないプログラムになってしまふので、時数は current hour ▾ から、分秒のデータは hour の小数部分を使用します。また、アナログ時計なので mod で 12 までの値にします。

set hour ▾ to UTC / 60 × 60 ◀▶ mod 24

set hour ▾ to current hour ▾ + hour - floor ▾ of hour ◀▶ mod 12

時針の角度です。( 12 時間で 1 回転 )

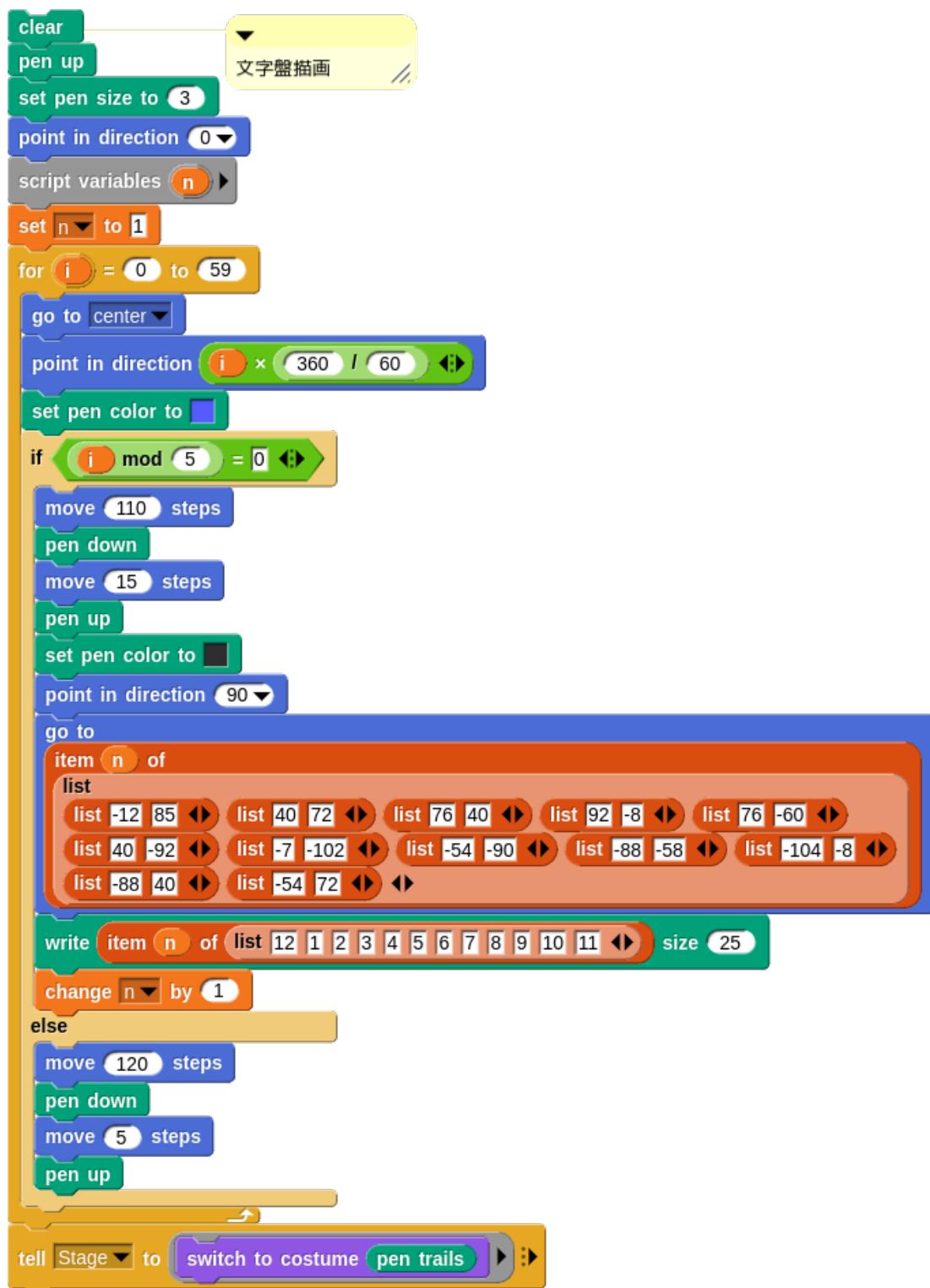
point in direction hour × 360 / 12 ◀▶

ペンで描く場合はこの逆の順序で描けば実際の時計と同じ重なりになります。

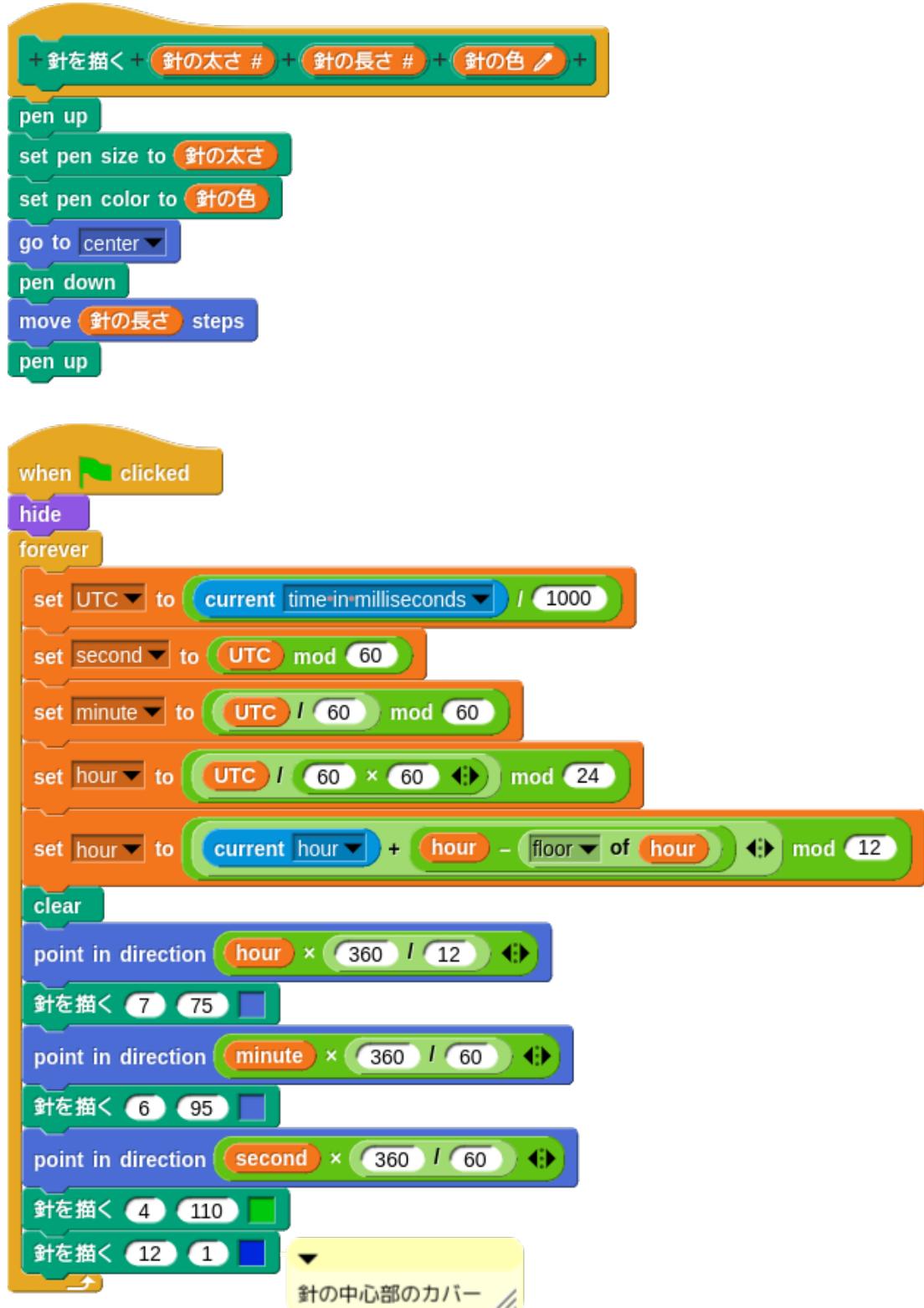
Scratch や Snap! の角度指定で画面上方向が 0 なのは時計やメーターの針の角度を指定しやすくするためかもしれません。

次のページから文字盤を描画し、ペン描画による時計のスクリプトを示します。

時計のスクリプトを実行する前にバックグラウンドを文字盤にしてください。これは [clear] ブロックで消えません。

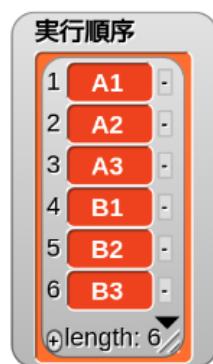
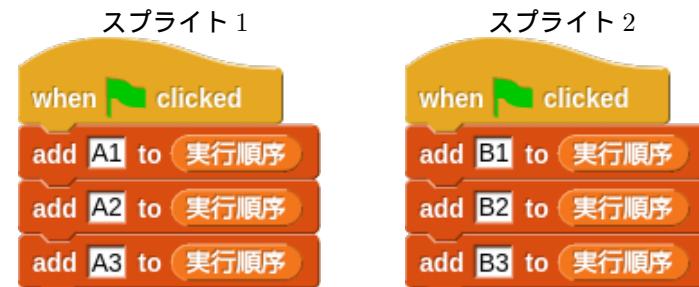


針を描くための定義ブロックです。



## 8.18 並列実行について

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。

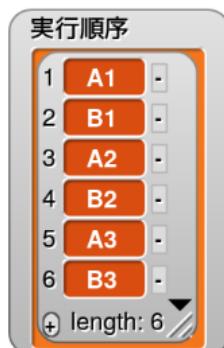


**set [実行順序 ▾] to [list ▶]** を実行してから  をクリックすると、左のようにリストに実行順序が表示されます。1 ブロックずつ交互に実行されるのではなく、一方はもう一方のスクリプトの実行が終わるまで待たれます。実行の順番はスプライト、スクリプトが作成された順番になるようです。

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。



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。）



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 1 ブロックずつ交互に行なうことができるようです。



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。

プログラムを統括するメインの役割を Stage に持たせるやり方があります。プログラムによっていろいろなスプライトが用いられるますが、Stage はどんなプログラムにも必ず存在するためと思われます。しかし、次のようにスクリプトの実行順序も最下位になっているので、ここでプログラムの初期設定をするには Stage のスクリプトを最初に実行させる工夫（他のスクリプトを broadcast で起動させるとか）が必要になります。



## 8.19 行列の積

### 8.19.1 inner product

配列要素同士の掛け算とは別に、代数学には行列と行列の積というものがあります。

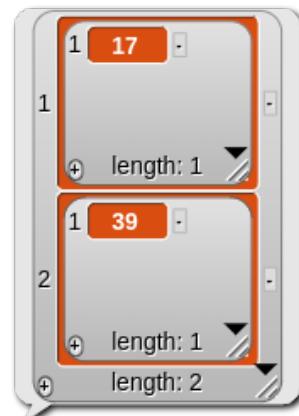
2行2列の行列同士では次のような計算方法になります。

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11} \times b_{11} + a_{12} \times b_{21} & a_{11} \times b_{12} + a_{12} \times b_{22} \\ a_{21} \times b_{11} + a_{22} \times b_{21} & a_{21} \times b_{12} + a_{22} \times b_{22} \end{pmatrix}$$

a側の行列の列数とb側の行列の行数は同じ必要があります。

定義ブロックを作成することは可能ですが、APL primitives ライブラリーにinner product ブロックがあるので利用できます。

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



$$\begin{aligned} & \begin{pmatrix} 1 & 2 & 3 \end{pmatrix} \begin{pmatrix} 4 & 5 \\ 6 & 7 \\ 8 & 9 \end{pmatrix} \\ &= \begin{pmatrix} 1 \times 4 + 2 \times 6 + 3 \times 8 & 1 \times 5 + 2 \times 7 + 3 \times 9 \end{pmatrix} \\ &= \begin{pmatrix} 40 & 46 \end{pmatrix} \end{aligned}$$

1	A	B
1	40	46



$\begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$ は1行3列の行列です。**list [1 2 3]**としてしまうとリストになってしまふので、



にする必要があります。

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  $\theta$  度回転させた座標  $(x', y')$  を求めるものです。

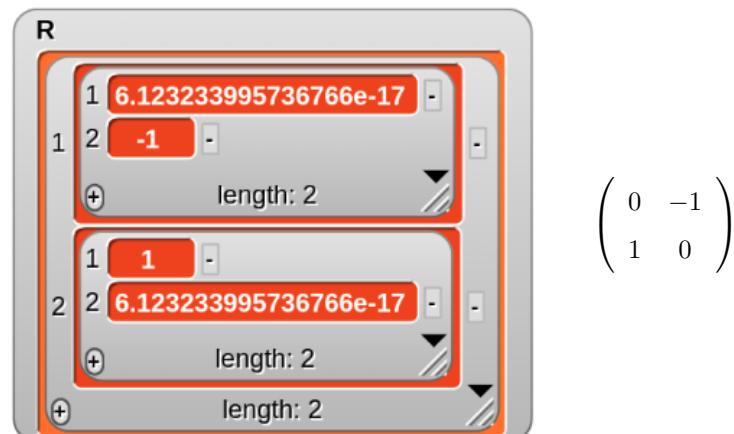
$$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

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 度反時計回りに回転させると  $(0, 1)$  の点になります。  
左側の行列、回転させるための変換行列  $R$  を作成するブロックです。



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



この行列  $\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$  と座標  $(x, y)$   $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$  の積

$$\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} \Rightarrow \begin{pmatrix} 0 \times 1 + (-1) \times 0 \\ 1 \times 1 + 0 \times 0 \end{pmatrix} \Rightarrow \begin{pmatrix} 0 \\ 1 \end{pmatrix} \text{を求めます。}$$



V2	
1	6.123233995736766e-17
1	length: 1
1	1
2	length: 1
1	length: 2

結果として、座標  $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$  を 90 度回転させた座標  $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$  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。

(x, y) 座標の対を並べたものを指定すればそのまま変換された座標の対が得られますが、普通の (x, y) 座標の指定の仕方ではありません。



The Scratch script consists of two parts. On the left, there is a list of lists block: `list [list [x1 x2] [list [y1 y2]]]`. To its right is a speech bubble containing a 2x2 matrix with columns labeled A and B:

2	A	B
1	x1	x2
2	y1	y2

転置行列を使うと、普通の (x, y) 座標の指定の仕方ができます。

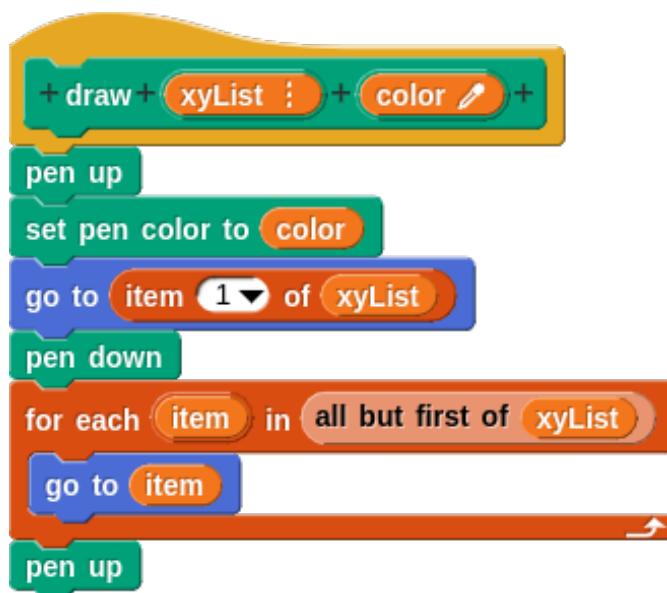


The Scratch script consists of two parts. On the left, there is a transpose block: `columns of [list [list [x1 y1] [list [x2 y2]]]]`. To its right is a speech bubble containing a 2x2 matrix with columns labeled A and B:

2	A	B
1	x1	x2
2	y1	y2

(x, y) 座標のリストで与えられた図形を一筆書きで描くブロックです。

go to ブロックは `go to [random position]` にリストをドロップしたものです。





三角形 を描くスクリプトです。

```

hide
set [V0 v] to [list [list [0] [0] <--> [list [20] [100] <--> [list [40] [0] <--> [list [0] [0] <--> [ ]]]]]]
draw [V0] [blue]

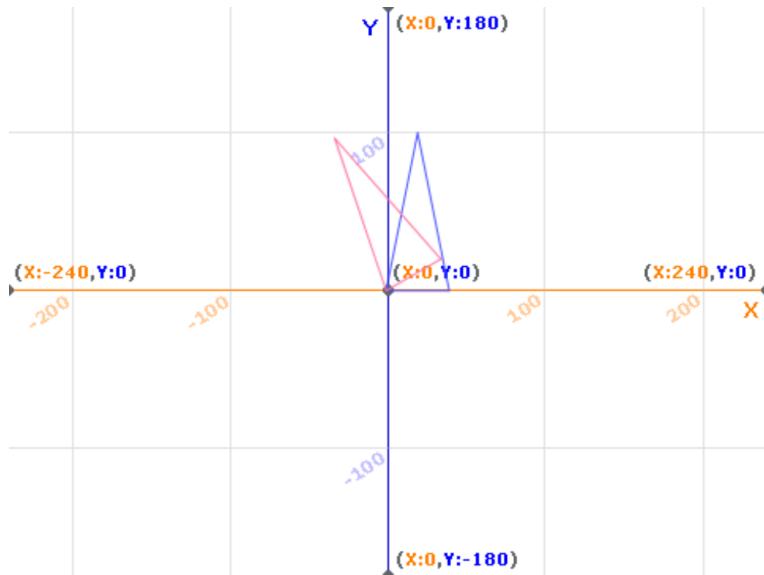
```

それを半時計回りに 30 度回転させるスクリプトです。

```

set [θ v] to [30]
set [R v] to [list [list [cos [v] of [θ]] [neg [v] of [sin [v] of [θ]]] <--> [list [sin [v] of [θ]] [cos [v] of [θ]]]]]
set [V2 v] to [inner product [R] [[+ [v] [v]] <--> [× [v] [v]]] <--> [columns [v] of [V0]]]
draw [columns [v] of [V2]] [pink]

```



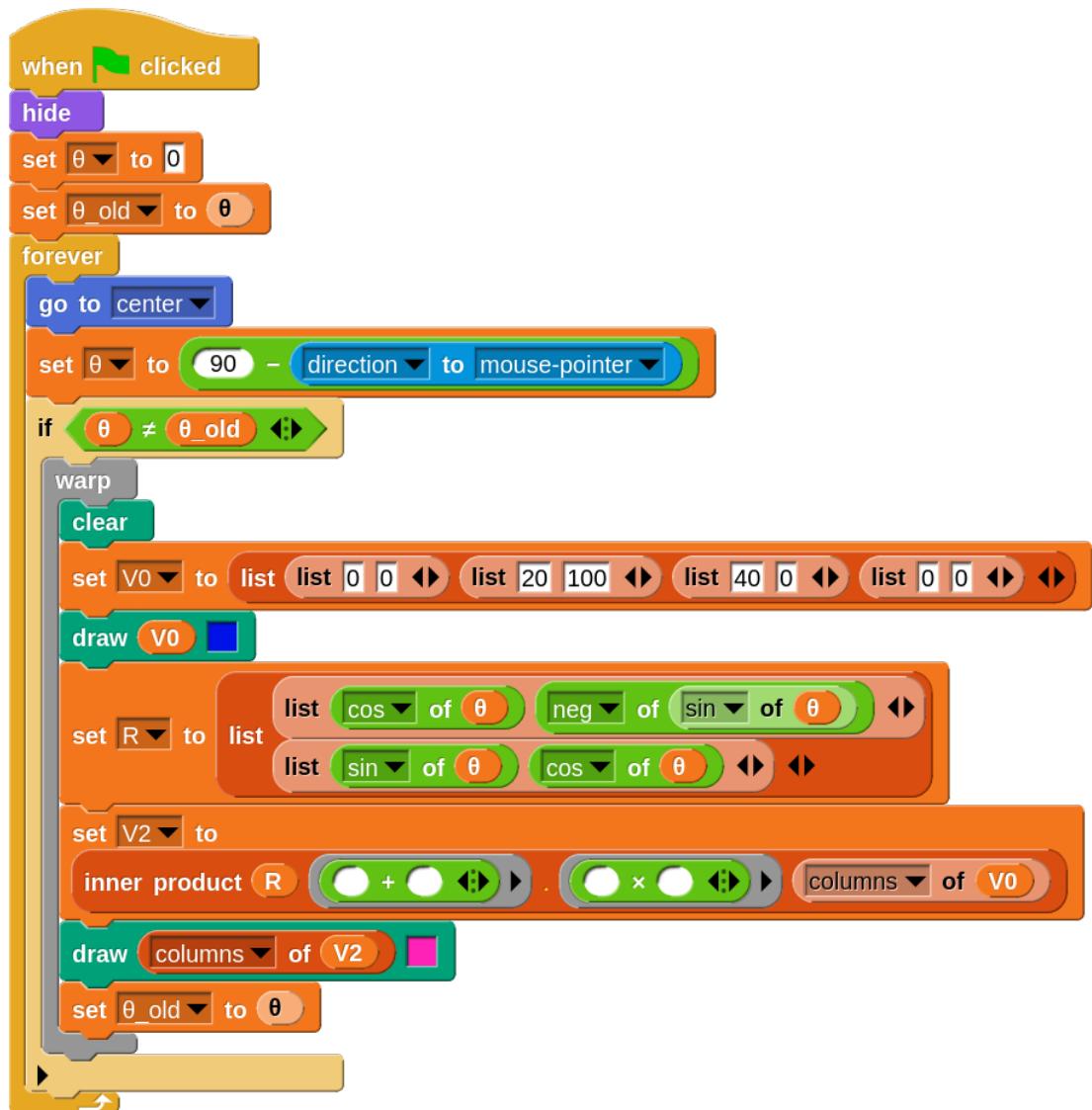
### [ 参考文献 ]

『Python ではじめる数学の冒険

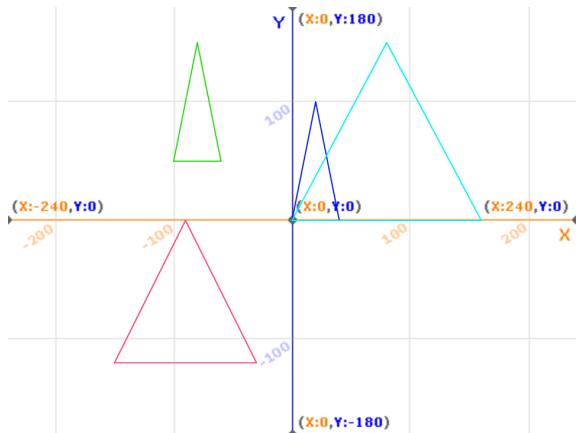
プログラミングで図解する代数、幾何学、三角関数』

Peter Farrell 著 鈴木幸敏 訳 オライリー・ジャパン 刊

原点座標から見たマウス位置の角度を として、マウスの動きに合わせて回転させてみます。なお、角度は水平方向を 0 度とする必要があるので補正します。角度を記録して変化があった時だけ warp で描画しています。



行列の積の演算には加算と乗算が含まれるので、加算部分だけが有効になるようにすれば図形の平行移動が可能ですし、乗算部分だけが有効になるようにすれば図形の拡大縮小が可能です。



x 軸方向に 4 倍、y 軸方向に 1.5 倍します。

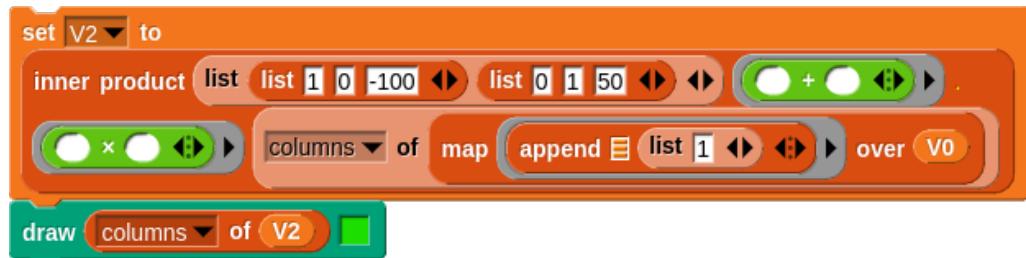
$$\begin{pmatrix} 4 & 0 \\ 0 & 1.5 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \Rightarrow \begin{pmatrix} 4 \times x + 0 \times y \\ 0 \times x + 1.5 \times y \end{pmatrix} \Rightarrow \begin{pmatrix} 4x \\ 1.5y \end{pmatrix}$$



平行移動の場合は 1 (有効化) と 0 (無効化) の要素を追加 (指定) して、移動座標を求めます。

x 軸方向に -100、y 軸方向に 50 平行移動します。

$$\begin{pmatrix} 1 & 0 & -100 \\ 0 & 1 & 50 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 \times x + 0 \times y + (-100) \times 1 \\ 0 \times x + 1 \times y + 50 \times 1 \end{pmatrix} \Rightarrow \begin{pmatrix} x - 100 \\ y + 50 \end{pmatrix}$$



Snap! ではリスト全体に対する演算ができるので、inner product を使わなくても次のように拡大縮小 (x 軸方向に 3 倍、y 軸方向に 1.2 倍) と平行移動 (-150, -120) が一度にできます。

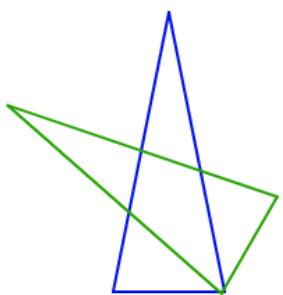


平行移動を利用すれば図形の任意の位置を中心として回転させることができます。三角形 V0 の重心の座標を中心にして 60 度回転してみます。

(Gx, Gy) を V0 の重心にし、原点に平行移動し、回転させ、元の位置に戻します。  
座標値を変換しているだけで図形をあちこち移動させているわけではありません。

The Scratch script consists of the following blocks:

- hide
- set [V0 v] to [list   
 list [0 0] <--> list [20 100] <--> list [40 0] <--> list [0 0] <--> <-->]
- draw [V0 v]
- set [Gx v] to  
item [1 v] of item [1 v] of [V0 v] + item [1 v] of item [2 v] of [V0 v] +  
item [1 v] of item [3 v] of [V0 v] / 3
- set [Gy v] to  
item [2 v] of item [1 v] of [V0 v] + item [2 v] of item [2 v] of [V0 v] +  
item [2 v] of item [3 v] of [V0 v] / 3
- set [θ v] to [60]
- set [R v] to  
list [cos v of θ neg v of sin v of θ] <-->  
list [sin v of θ cos v of θ] <-->
- pipe [V0 v] - list [Gx v Gy v] <-->  
columns v of [ ]  
inner product [R v]  
columns v of [ ]  
[ ] + list [Gx v Gy v] <--> <--> <-->
- draw [V2 v]



### 8.19.2 outer product

inner product に対して、 outer product 目 o. というものもあります。

APL には、 numbers from 1 to X の機能を持つ l x があります。

この記号のようなものはギリシャ文字の ノイオタです。

次のようにすると、九九の表の一部ができます。

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

outer product l 3 o. x l 3

計算方法を表示させてみます。

3	A	B	C
1	1x1	1x2	1x3
2	2x1	2x2	2x3
3	3x1	3x2	3x3

outer product l 3 o. join x l 3

単位行列なども作成できます。

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. if = then 1 else 0 l 4

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

outer product l 4 o. if ≥ then 1 else 0 l 4

#### [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

『基礎からの APL 解説と例題例解』では、素数の求め方が解説されています。

set N ▾ to L 5

1 ~ 5 の整数のリスト N を作り、その中の素数を求めてみます。

まず、そのリストの各要素に対して、そのリストの各要素で割った余りが 0 のものを調べます。

(129 ページ、outer product の計算方法で × の代わりに mode になります。)

	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

pipe N → outer product 目 mod = 0 目

A の列の意味は、1 ~ 5 の値を 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 ~ 5 の値を 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 の時だけ true です。他の C, D, E ではどちらも同じ値の時だけ true です。

素数は 1 と自分自身でしか割り切れないものなので、その行の true の合計が 2 のものということになります。

配列で、各行の要素の合計を求めるのが combine in rows ブロックです。

combine in rows (reduce by column vectors) + / 目

true と false についても、1 と 0 として合計してくれるようです。

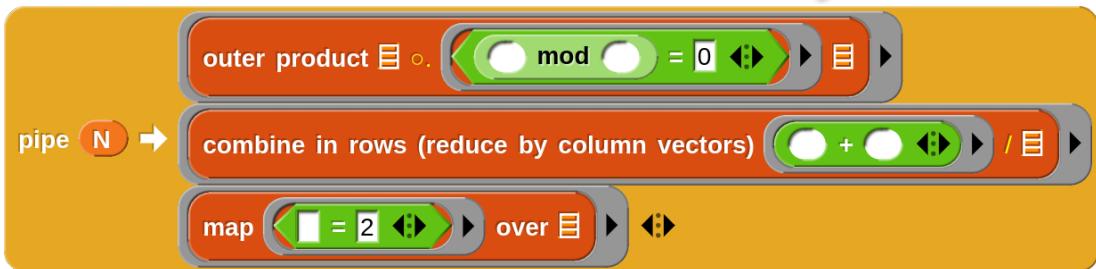
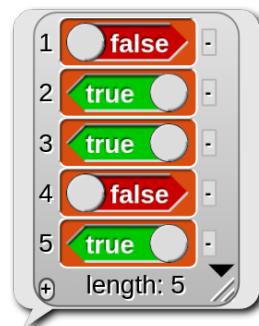
1	1	-
2	2	-
3	2	-
4	3	-
5	2	-
+	length: 5	▼

pipe N →

outer product 目 mod = 0 目

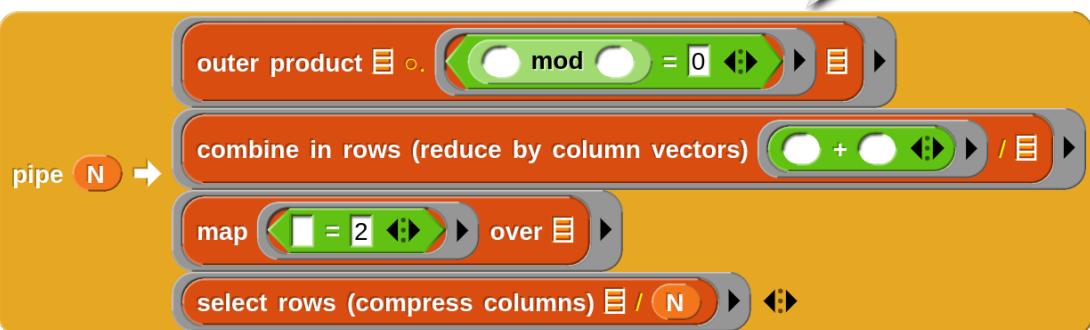
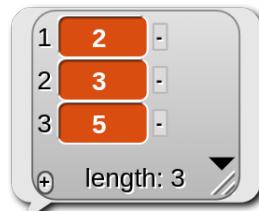
combine in rows (reduce by column vectors) + / 目

行中の true の要素の合計が 2 のものを map を使って求めます。



2, 3, 5 の位置が true になりました。その位置に対応する N の要素のリストを求めます。  
select rows ブロックはリストの中から true で指定した位置の要素を集めてリストにします。  
これにより、素数のリストが求められます。

select rows (compress columns) [N]



次のようにまとめることもできます。

select rows (compress columns)

map [= 2] over

combine in rows (reduce by column vectors) / [N]

outer product [N] [mod = 0] / [N]

## 9 Continuation 繼続

「Continuation 繼続」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。

### 9.1 run と call の with continuation バージョン

Snap! の Continuations と Iteration, composition のライブリーには、run with continuation と call with continuation があります。

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

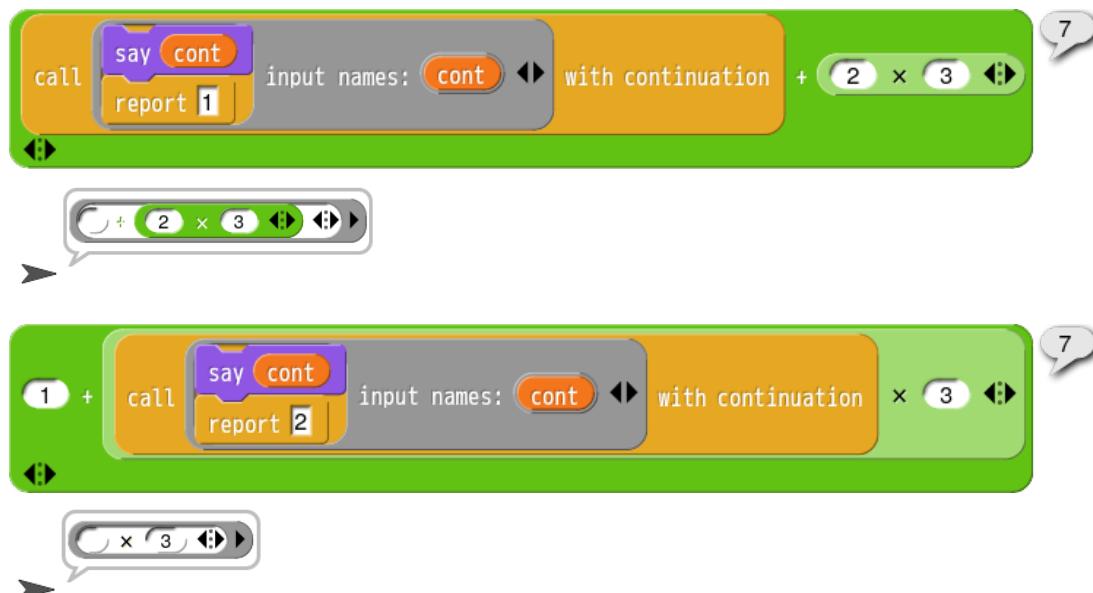


しかし、リング端の三角をクリックしてフォーマルパラメーターを使用してみると、call with continuation の方は空のリングが表示されます。



call with continuation のフォーマルパラメーターにはこのブロックに継続するスクリプトがセットされます。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。

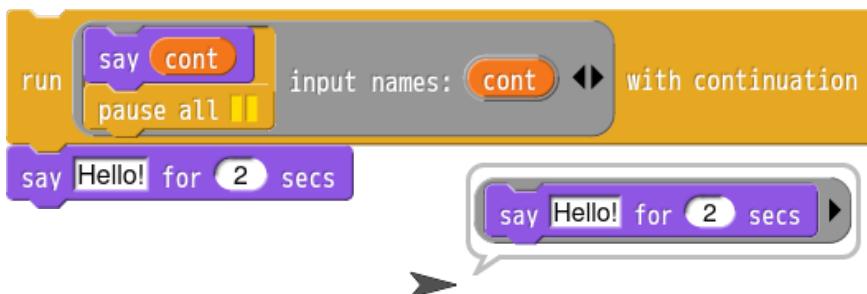
 の各スロットにこれをセットして、その位置での継続を表示させてみます。それぞれその時点での継続、その後に処理すべき内容がセットされます。（フォーマルパラメーターを cont にリネームしています。）



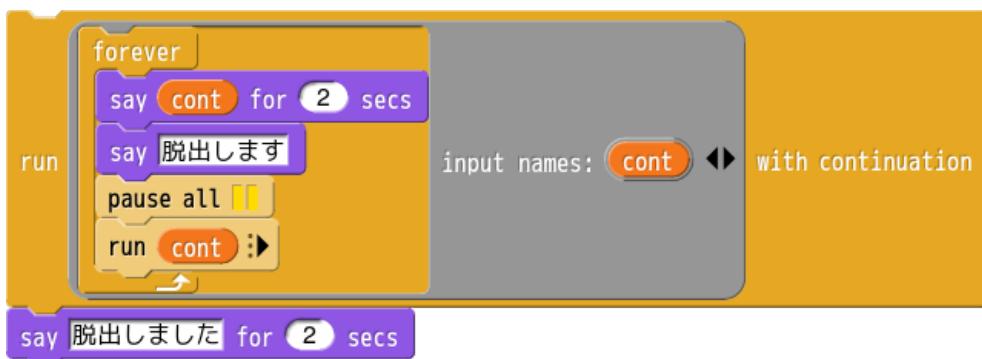


用途としては、call with continuation や run with continuation のリング内部にあるスクリプトで、ある条件の時にその外（継続）への脱出に使われます。

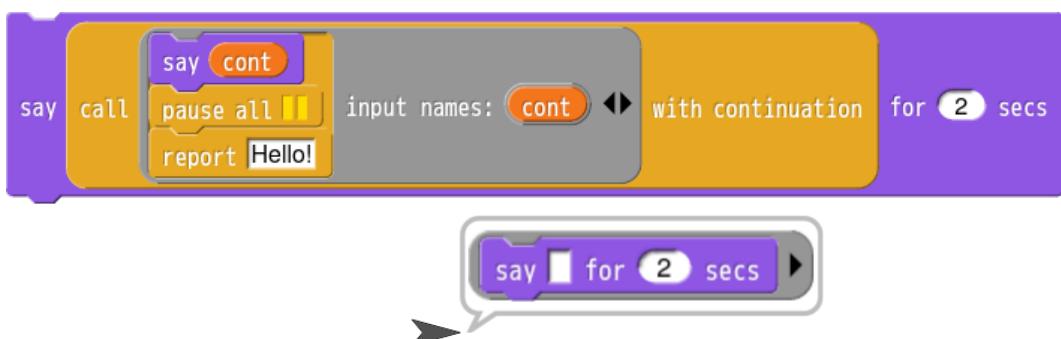
このスクリプトを実行すると、run with continuation の継続が表示されます。



say や pause all が置かれているところにループを置き、その内で run cont とすると、run with continuation の中から外へジャンプすることができます。



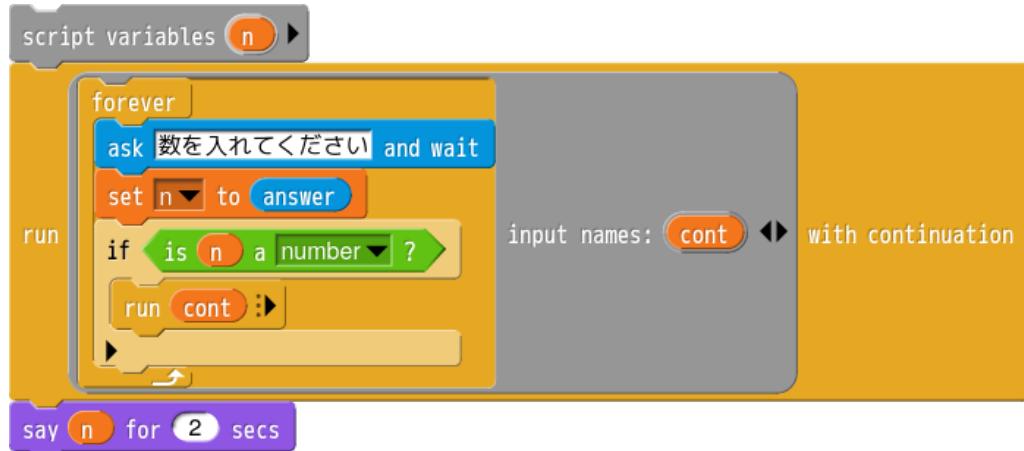
このスクリプトを実行すると、call with continuation の継続が表示されます。自分自身が入っているので、say の入力が空のブロックが表示されます。



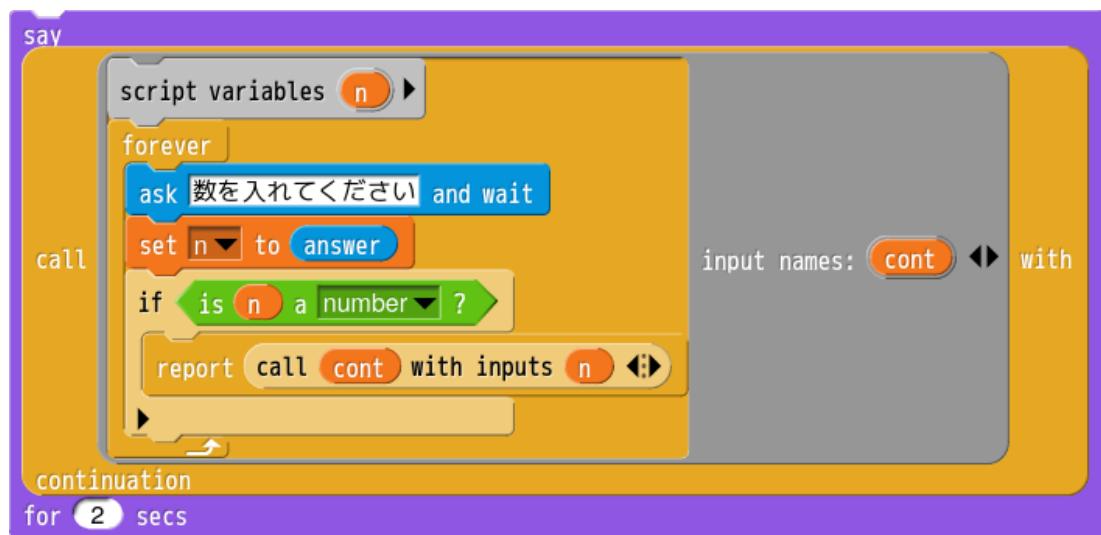
say や pause all が置かれているところにループを置き、その中で call cont とすると、call with continuation の中から外へジャンプすることができます。call with continuation はリポーターブロックなので値を返さなければなりません。call cont とすることで脱出になりますが、with inputs で値を指定することでその値「脱出しました」を返すことができます。



ask を使って数値のみを受け付けるスクリプトを作ってみます。run with continuation 版です。

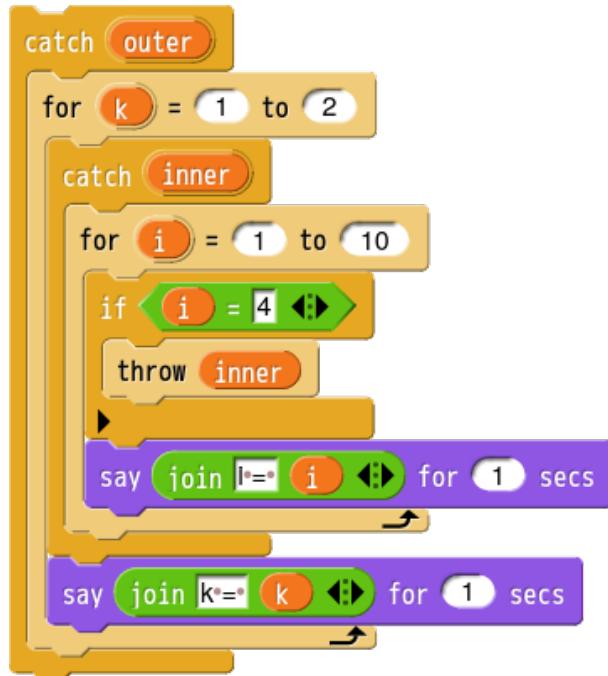


call with continuation 版です。



## 9.2 catch と throw

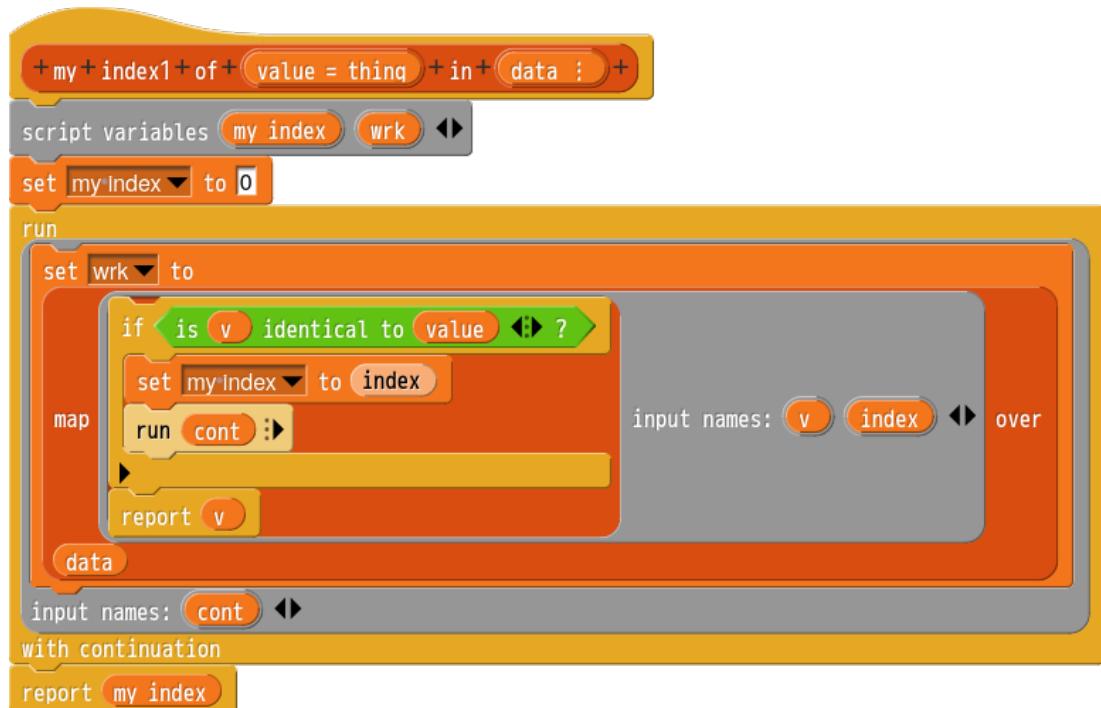
継続を意識することなくループなどからの脱出に使用できるものが、Libraries の Iteration, composition にある catch と throw です。



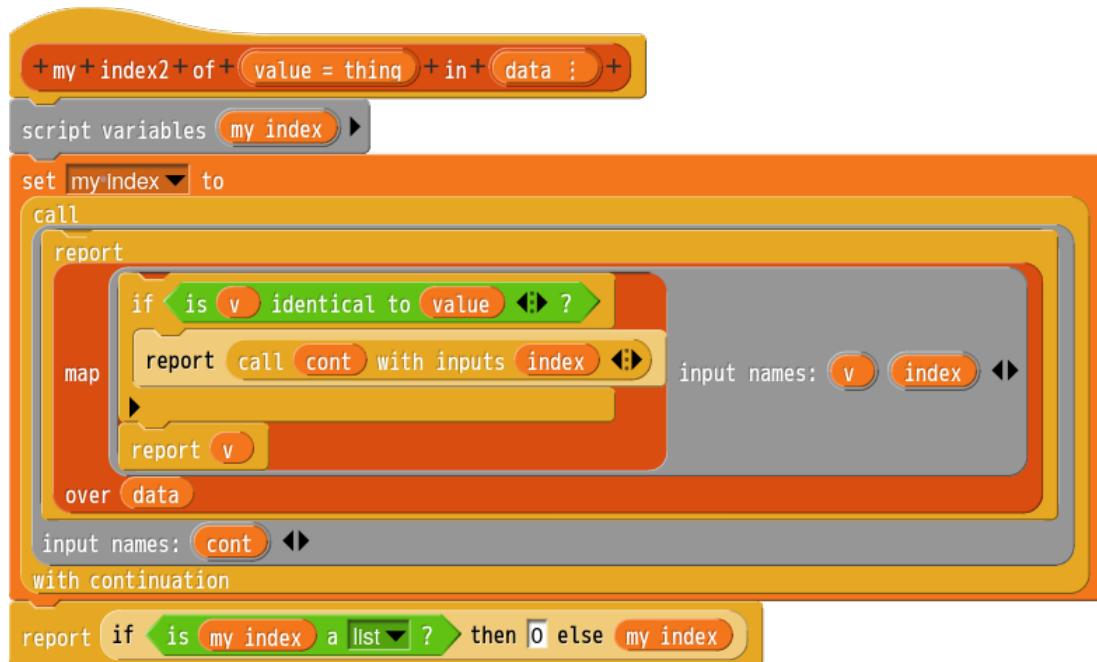
これを実行すると、`i` の値が 4 になつた時に内側のループから脱出し、`catch inner` の次のスクリプトに進みます。`throw outer` になると、外側のループから脱出し、`catch outer` の次のスクリプトに進みます。ある条件の時に、`throw` で指定したラベルの付いた `catch` ブロックの外側に脱出するという仕組です。

## 9.3 map からの脱出

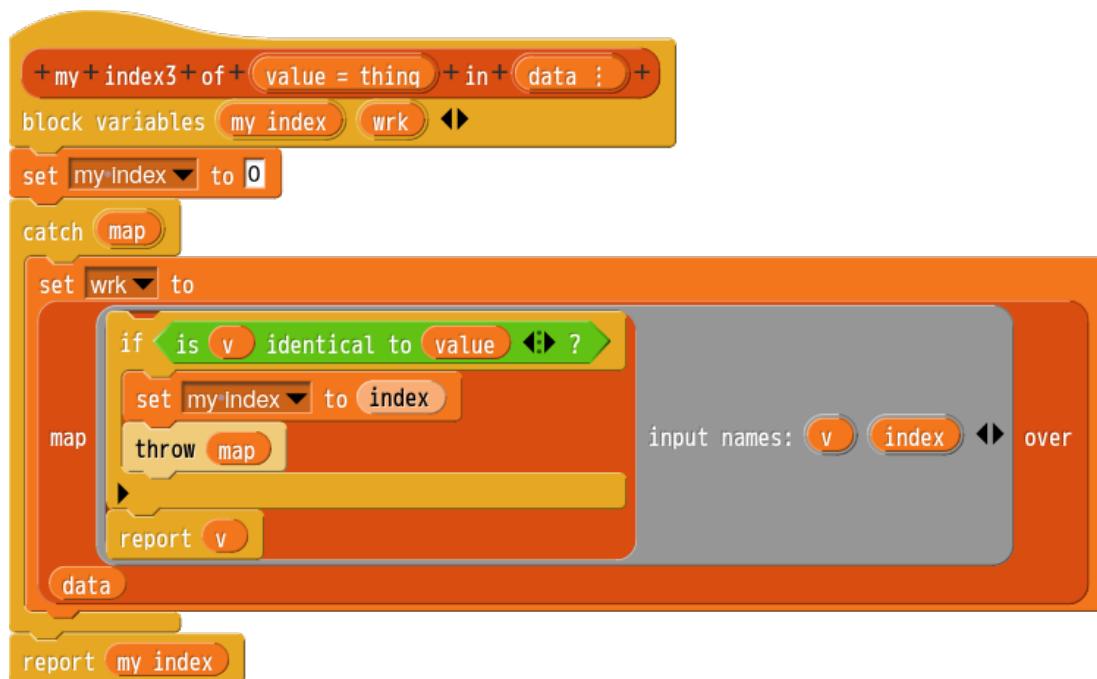
92 ページで `index of [thing] in [list]` を扱いました。この方法だとリストのすべての要素を走査しています。次のようにすると、要素が見つかった時点で脱出することができます。run with continuation 版では `map` を使用するにはダミーの変数 `wrk` が必要です。



call with continuation 版です。wrk は使用していません。map 操作の結果は、見つかった場合はインデックス (number) 、そうでない場合はリストのコピー (list) になります。結果が list ならば 0 を返します。

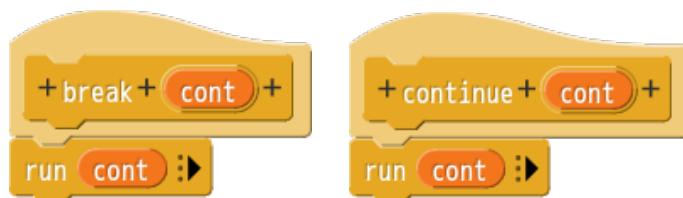


catch throw 版です。

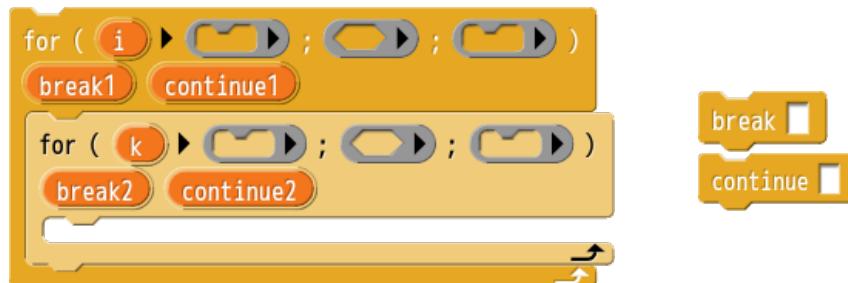


## 9.4 break と continue

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「継続処理」へジャンプして次の繰り返しを開始する機能です。紛らわしいですが、ここで入力変数名に使われている「継続」はループ処理を続けるという意味で、continuation に関するものではありません。変数 i は special から variables に設定します。( 79 ページ参照) break と continue 定義中の cont は Any type 変数です。



break と continue は、upvar オプションの変数です。for ( ; ; ) ブロックから break, continue 変数をそれぞれ break や continue ブロックにドロップして使用します。二重三重のループの場合は、変数名に数字を追加するなどして区別してください。



次のスクリプトは、「0, Hmm..., 1, Hmm..., 2, Hmm..., 3, 6, Hmm..., 7」と順に表示するものです。3で値を5にしてcontinue、7でbreakです。

```

for ( [ i ] set [ i ] to [ 0 ] ; [ i < 10 ] ; [ change [ i ] by [ 1 ] ] )
  break
  continue
  say [ i ] for [ 1 ] secs
  if [ i = 3 ]
    set [ i ] to [ 5 ]
    continue
  end
  if [ i = 7 ]
    break
  end
  think [ Hmm... ] for [ 1 ] secs
end

```

同じことをする、プリミティブのforとrepeat untilを使ったcatchとthrow版です。

```

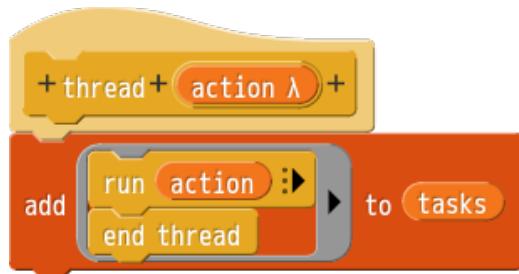
script variables [ i ] [ start ] [ end ]
set [ start ] to [ 0 ]
set [ end ] to [ 10 ]
set [ i ] to [ start ]
repeat until [ i > end ]
  catch [ break-for ]
    for [ i ] = [ 0 ] to [ 10 ]
      catch [ continue-for ]
        say [ i ] for [ 1 ] secs
        if [ i = 3 ]
          set [ i ] to [ 5 ]
          throw [ continue-for ]
        end
        if [ i = 7 ]
          throw [ break-for ]
        end
        think [ Hmm... ] for [ 1 ] secs
      end
    end
  end
  change [ i ] by [ 1 ]
end

```

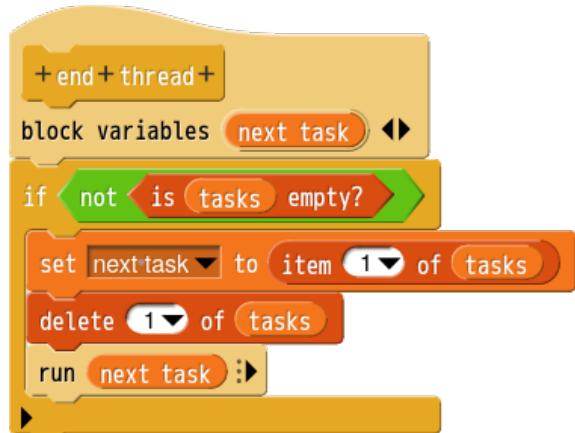
## 9.5 thread

Snap! では複数の **when green flag clicked** が存在する場合、それぞれのスクリプトを並列に実行しているように見せています。継続を使用すると、一つのスクリプト内にある複数の処理ブロックを並列に実行の順番や処理の移行のタイミングをプログラミングすることができます。並列処理されるそれぞれのスクリプトを **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** の変数を作成しておきます。以下に示す 3 個の定義ブロックはスレッド処理を行うための 3 点セットです。

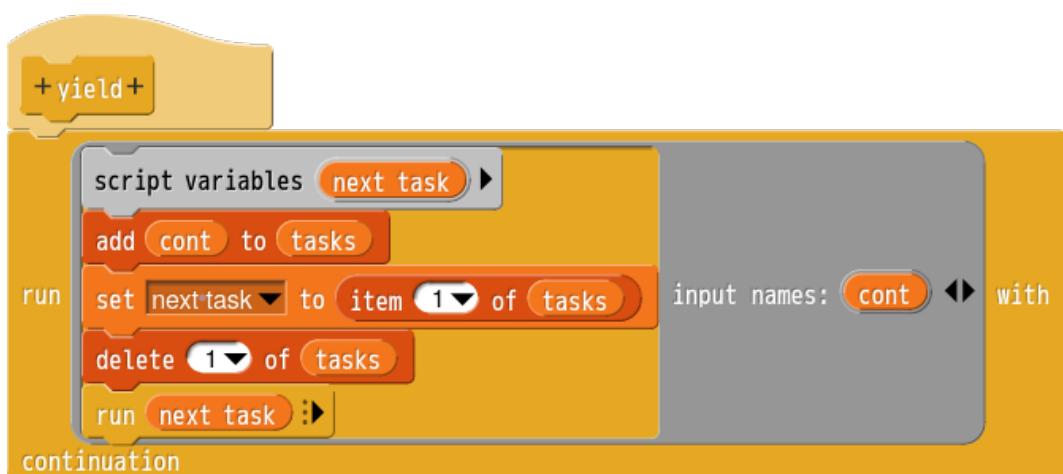


**thread** スレッドを設定するブロックです。( 設定されたスレッドを **tasks** リストに加えていきます。) **action** は Command(C-shape) 型です。



**end thread** **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。

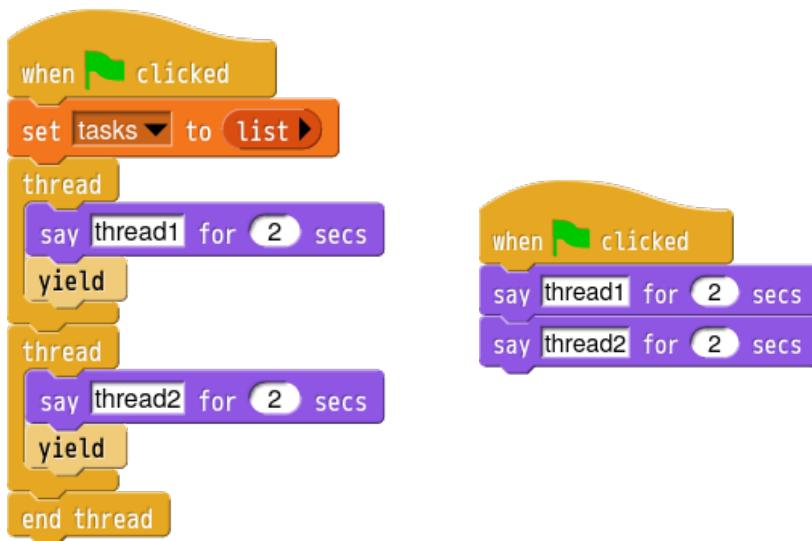
**yield** ブロックは、継続を **tasks** リストに加え、**tasks** リストの先頭のスクリプト(スレッド)を取り出して実行します。つまり、このブロックがこの **thread** の実行と次の **thread** への切り替えるタイミングを示します。



**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ます。

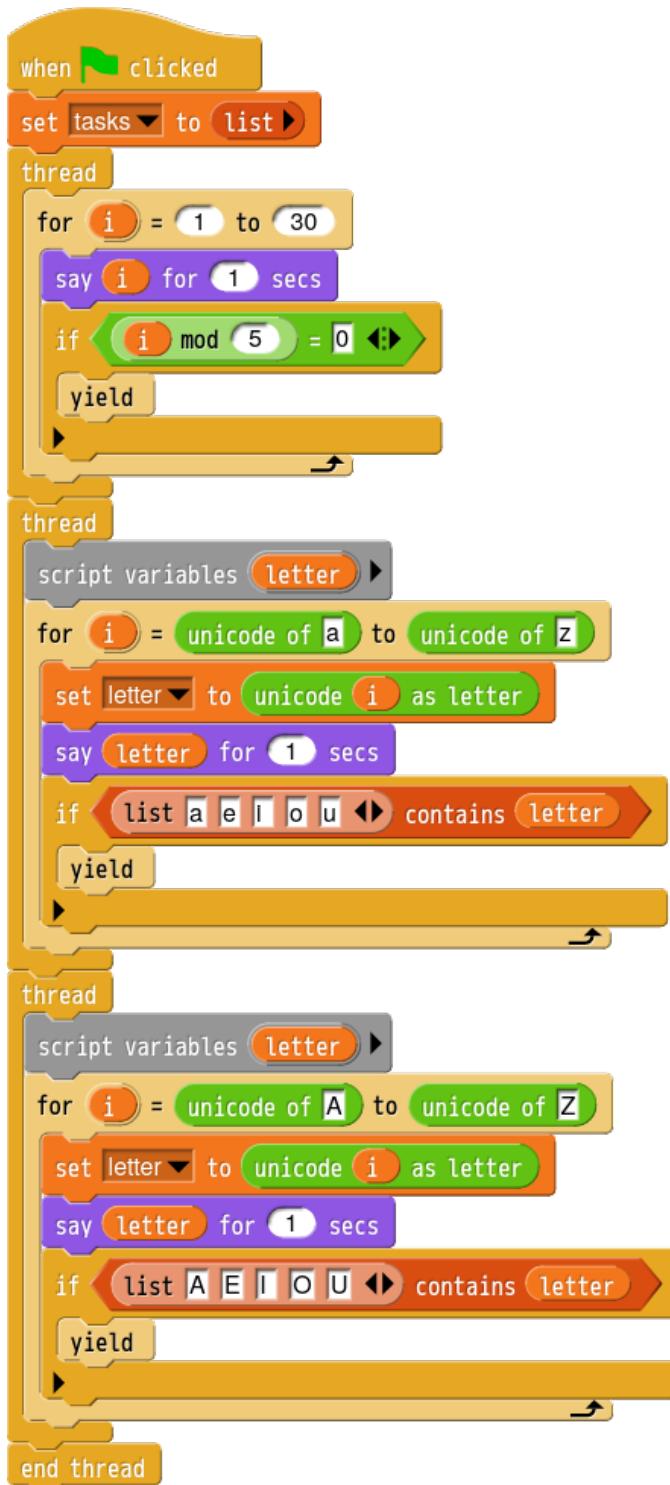
次のスクリプトは、右のスクリプトと同じことなのであまり意味はありませんが、使い方の例です。



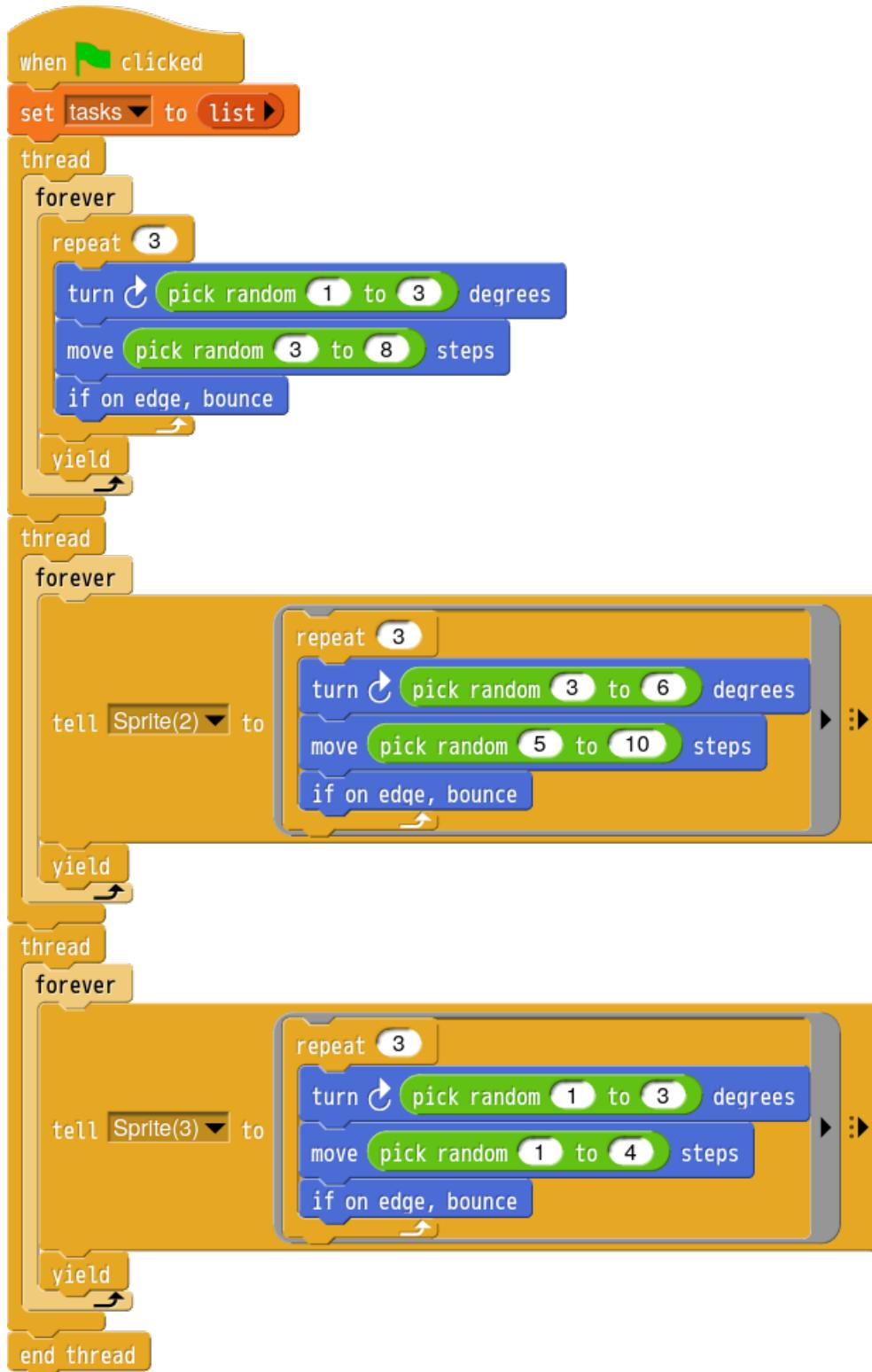
番号を表示するだけの 3 個のスレッドを実行してみます。



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。指定の条件の時に次のスレッドに移行します。最初のスレッドは 1 から 30 までカウントしますが、5 で割り切れる値の時に次のスレッドに移行します。次のスレッドでは a から z までを表示しますが、a, e, i, o, u の時に次のスレッドに移行します。次のスレッドでは A から Z までを表示しますが、A, E, I, O, U の時に次のスレッド(最初のスレッド)に移行します。この操作がそれぞれのスレッドが終了するまで続けられます。



次のスクリプトはスプライトを 3 個使います。それぞれ Sprite, Sprite(2), Sprite(3) という名前になっているとします。スプライトをすこし回転させ前進させることを 3 回行います。thread を使わなくともそれぞれのスプライトに同じような動きをさせることはできますが、thread を使うと各 thread の実行の順番や処理が切り替わるタイミングが制御できます。このスクリプトは Sprite 用で、他のスプライトにスクリプトは必要ありません。



## 10 再帰呼び出し

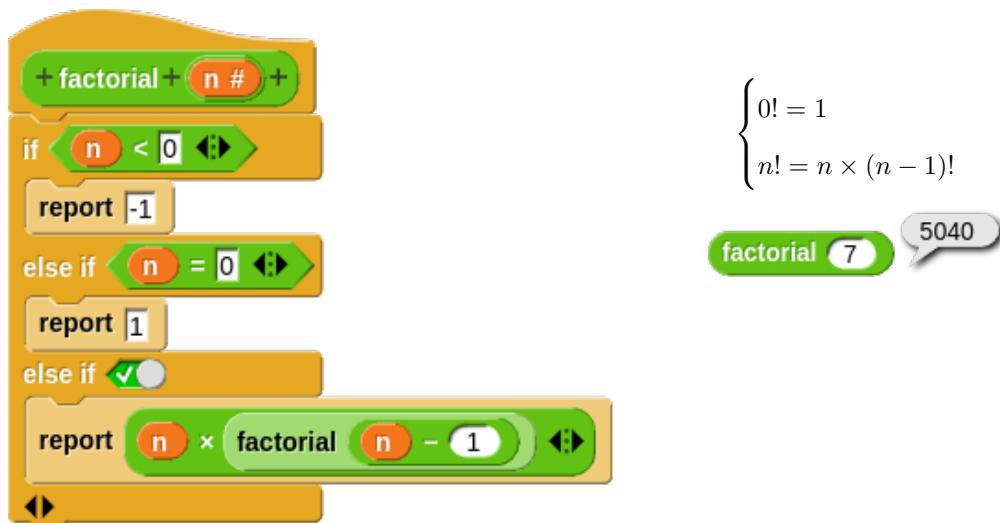
再帰、再帰呼び出し（recursive call）は作成したブロックの中で自分自身を呼び出す（実行する）ものです。関数型プログラミングでは再帰呼び出しが繰り返し処理の手法になっています。

### 10.1 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。

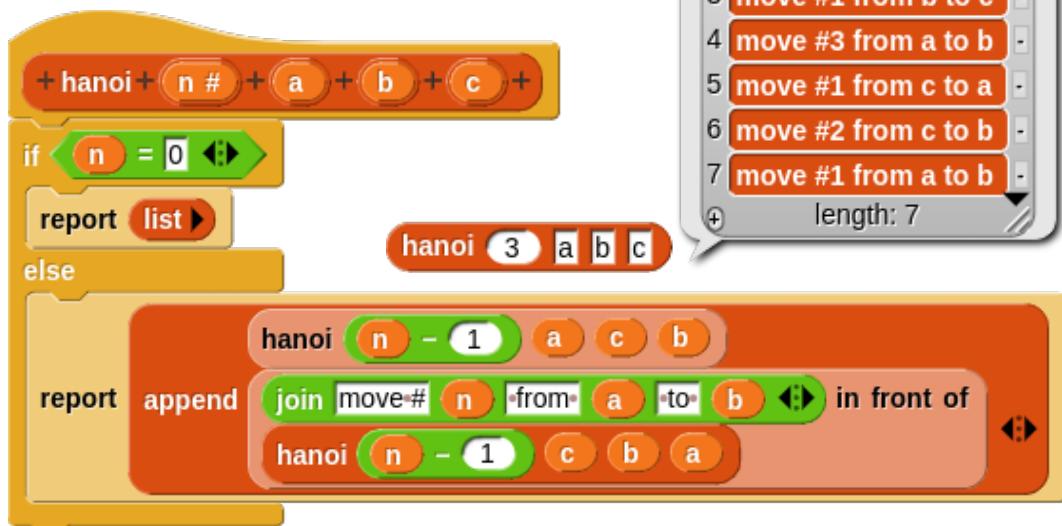
#### 10.1.1 階乗

factorial 階乗は再帰呼び出しの例としてよく使用されます。



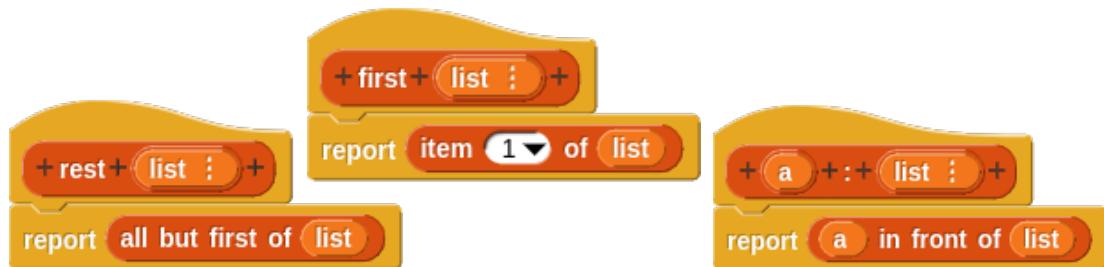
#### 10.1.2 ハノイの塔

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。



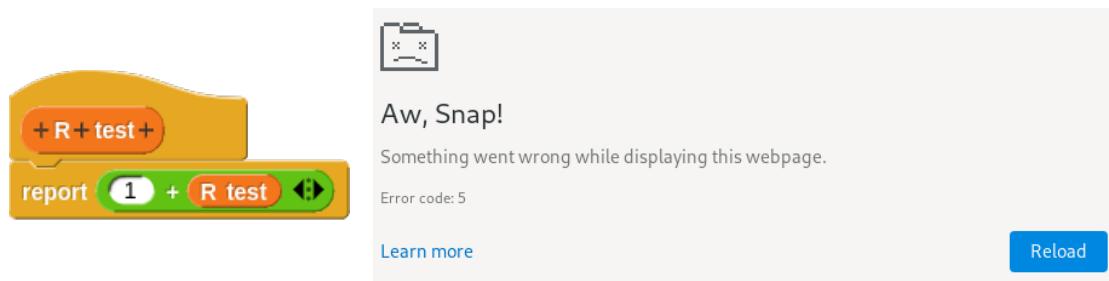
## 10.2 再帰呼び出しの使用

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。



### 10.2.1 繰り返し

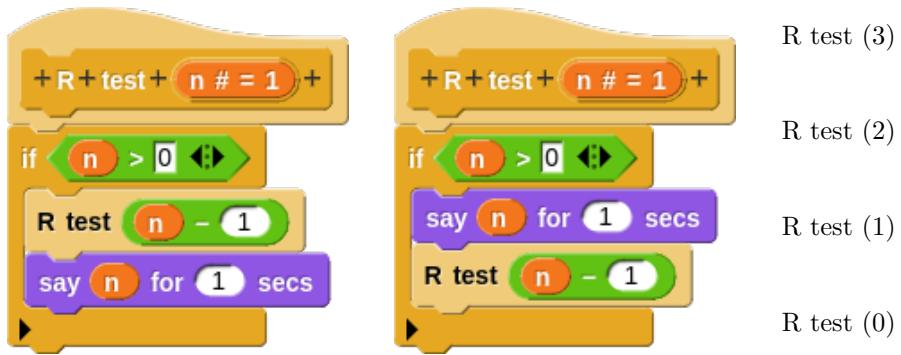
これはただ自分自身を呼び出すものです。(エラーになるので実行しないでください)



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。

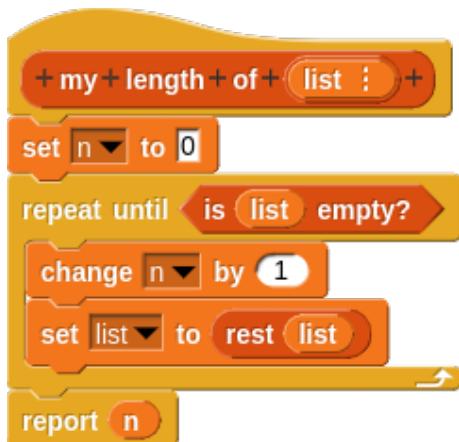


次は指定した回数だけ何かを行う定義ブロックです。n = 0 が終了条件になります。n の値を減らしながら以下の矢印(→)のように自分自身を呼び出していきます。この定義ブロックは 0 以下だと何もしないで呼び出し元に戻ります。呼び出し元に戻るを繰り返し、一番最初の呼び出し元に戻ったら終了です。再帰呼び出しブロックの位置により n の値はカウントアップにもカウントダウンにもなります。



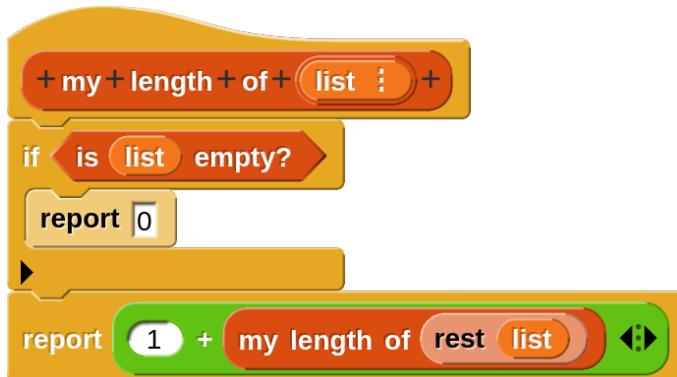
### 10.2.2 my length

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。

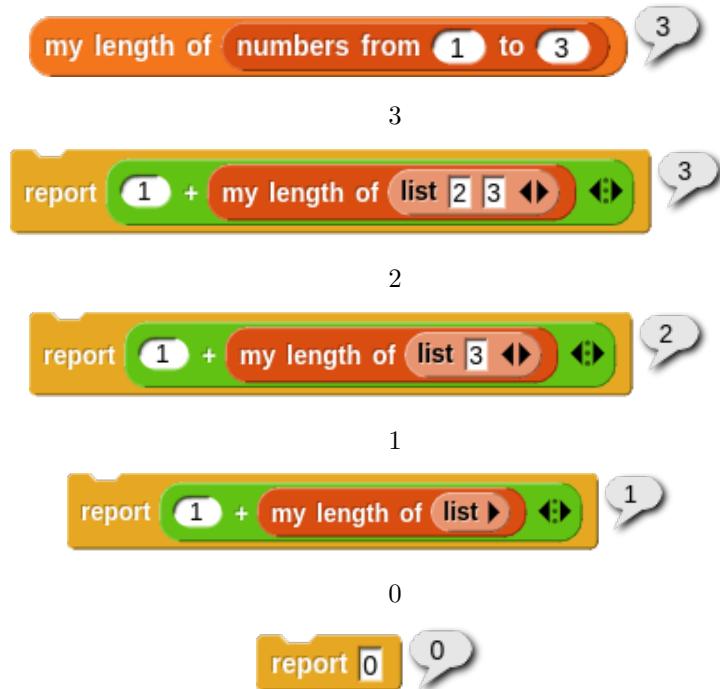


再帰版です。カウント用の変数が無いので理解しにくいですが、report が返す値がカウント用変数の役割を果たしています。**my length of rest list** でリストが空になるまで再帰呼び出しされて、0, 1, 2, ... と、report が返す値 +1 を積み重ねて、結果的に 0 からのカウントアップで要素数を求めることができます。自分自身を呼び出すことを除けば repeat until 版と同じやり方になります。

[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。] ということを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、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。



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。

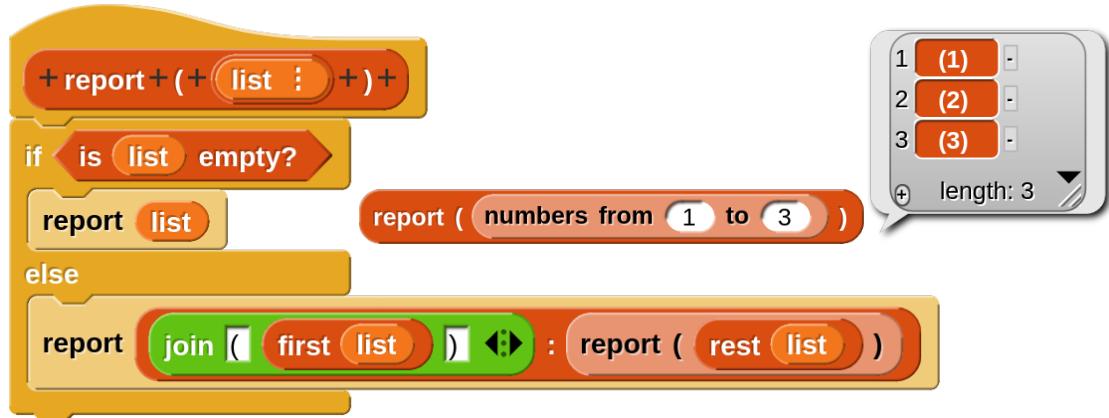


因みに、map で要素と同数の 1 のリストを作成し、それを合計するやり方もあります。



### 10.2.3 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  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。)



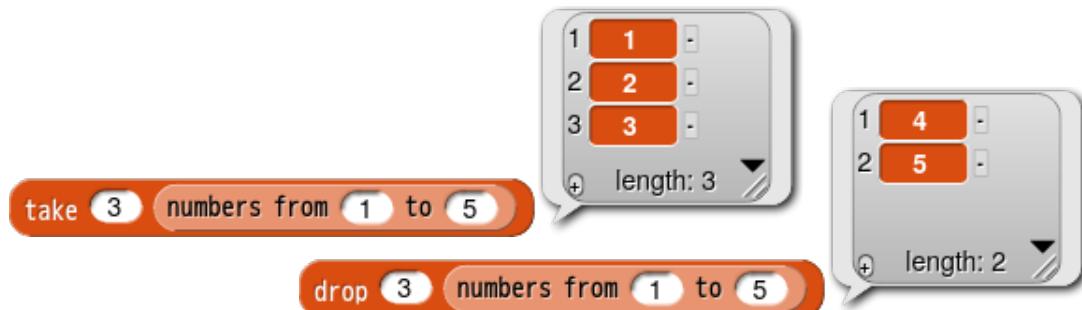
```

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 = ()

```

#### 10.2.4 take と drop

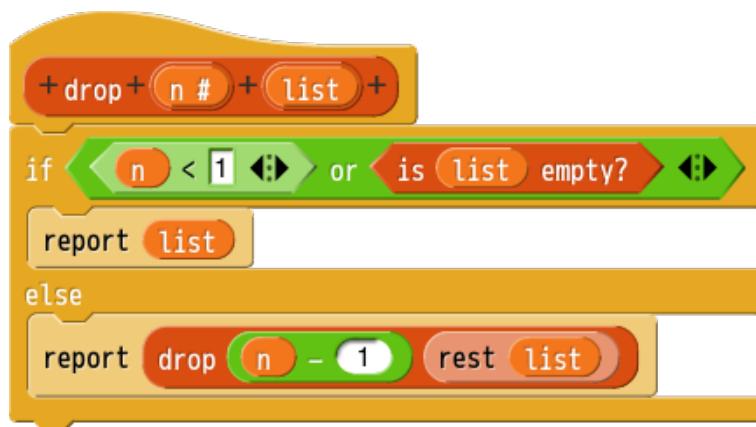
Haskell 言語にはリストの先頭から  $n$  個の要素を取り出す `take` と、リストから先頭の  $n$  個の要素を取り除いたリストを返す `drop` があります。以下のような動作になります。



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。



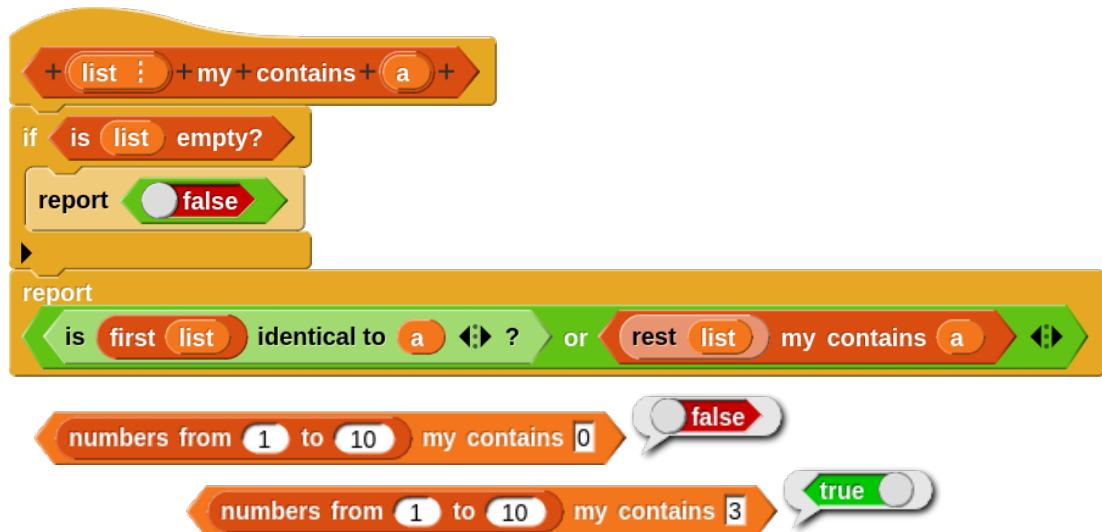
`drop` では  $n$  の値が 0 になるまで  $n$  の値を  $-1$  しながら先頭以降のリストに対して再帰を繰り返します。 $n$  が 0 の時に返すのは引数であるリスト、つまり求めるリストです。引数であるリストが空の場合は空リストを返すことになります。



### 10.2.5 my contains

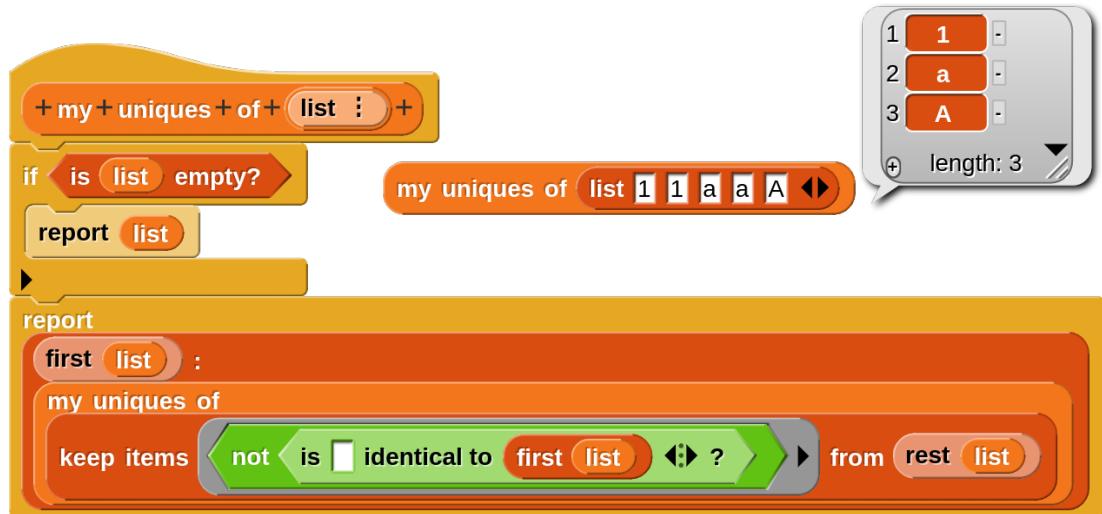
リストの中に指定の要素が存在するかを求める contains ブロックを作ります。

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。



### 10.2.6 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。

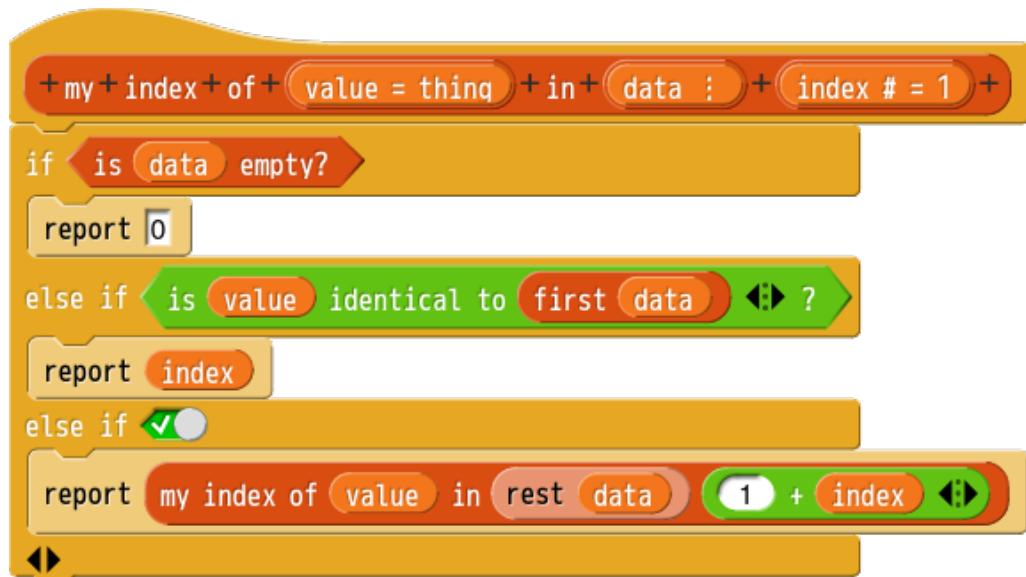


keep を使って先頭の要素と同じでないものを集めることを再帰的に行っています。

因みに、**not [is [list] 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。(26 ページ参照)

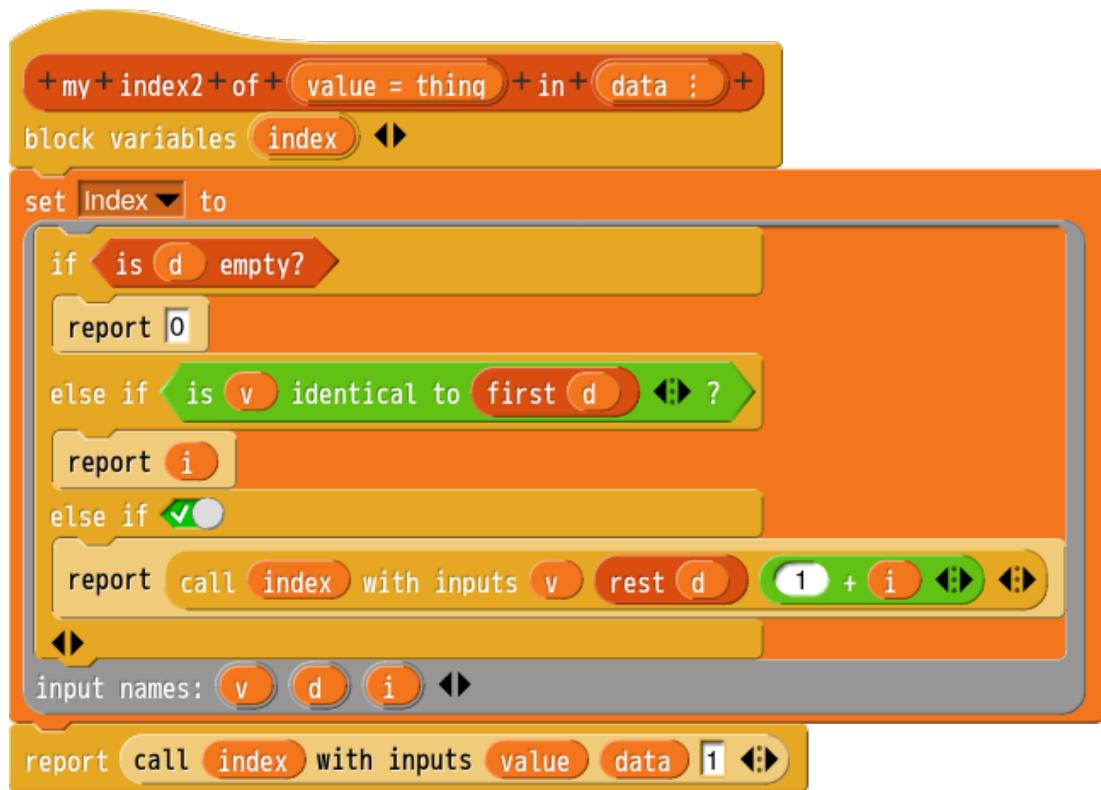
### 10.2.7 my index of ( ) in ( )

`index of [thing] in [list]` 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で値を渡していきます。



`my index of [A] in [split aazz11000AA by letter ▾ 1] [10]`

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。



`my index2 of [A] in [split aazz11000AA by letter ▾ 1] [10]`

### 10.2.8 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。



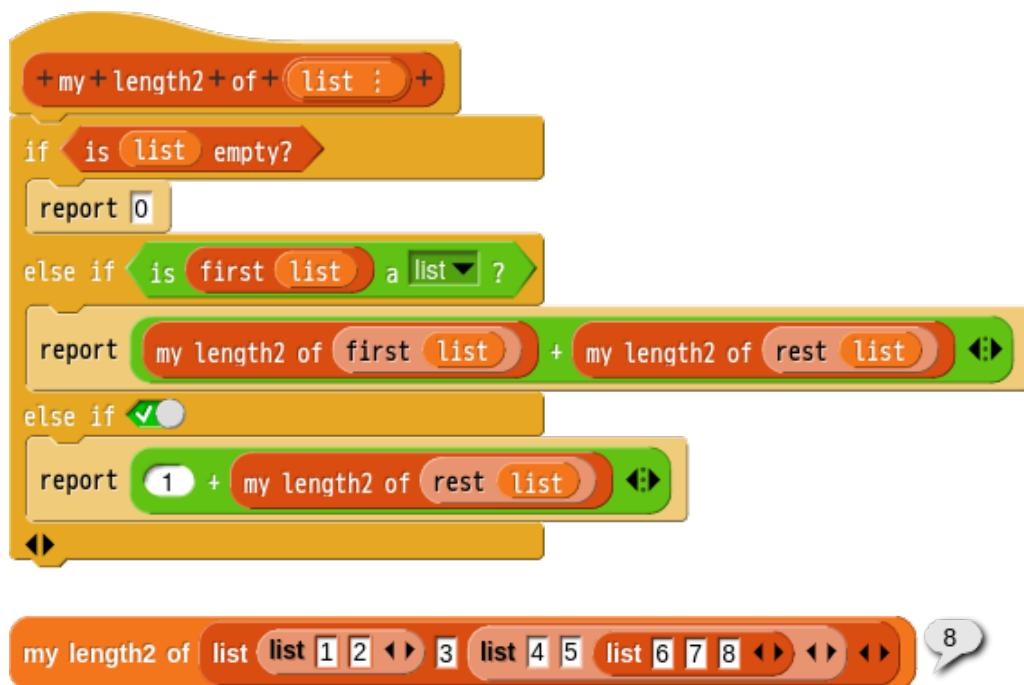
これは my length も同様です。



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。

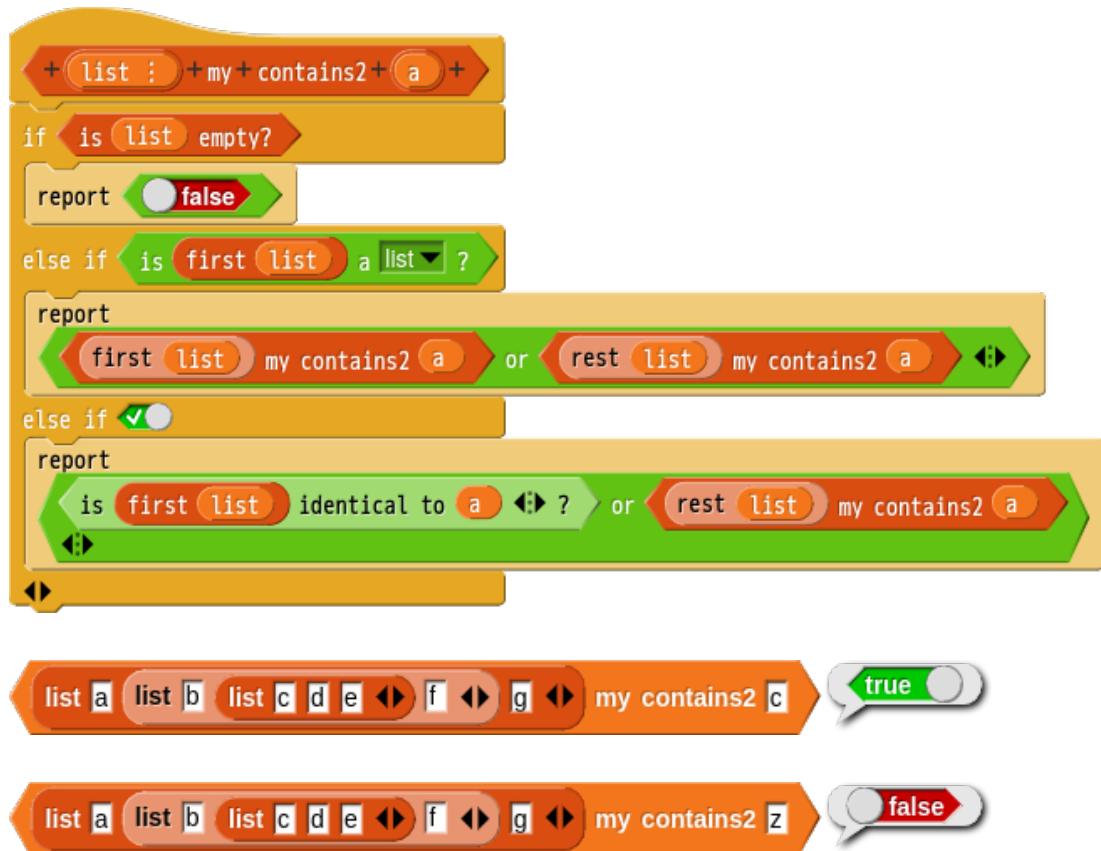
- もしリストが空ならば 0 をリポートする。
- もし先頭の要素がリストならば、そのリストに my length2 をしたものと残りに対して my length2 をしたものを加える。
- そうじゃなかったら、残りに対して my length2 をしたものに 1 を加える。



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くって、指定の要素が見つか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。



### 10.2.9 指定の要素に対する delete と replace

リストからある要素を削除することは 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。

```

keep items
not [a = 2]
from list [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]]]
end

```

```

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、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。

	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。つまり、次の下から二番目の report ブロックで new をくっつけるかどうかで replace か delete になります。

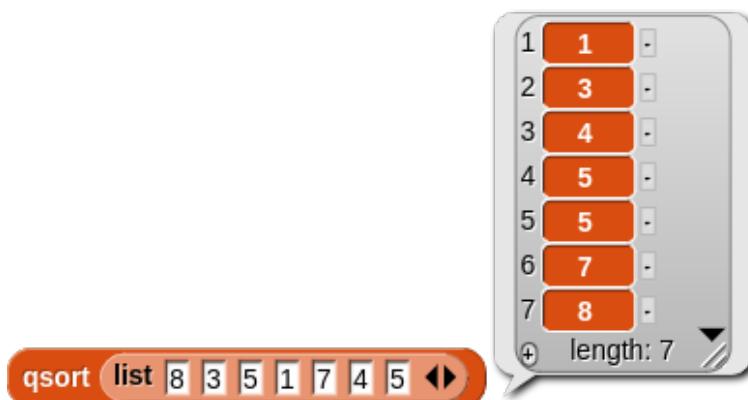
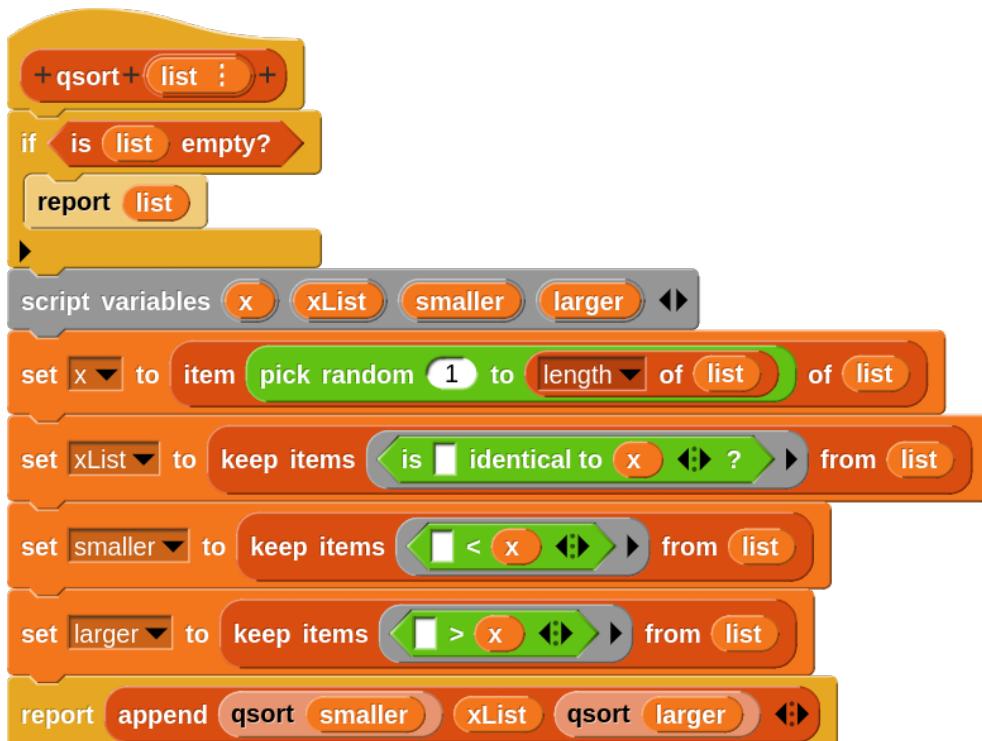
```
+ replace + old + with + new + of + list : +
if is list empty?
report list
else if is first list a list? :
    report replace old with new of first list :
        replace old with new of rest list
else if is first list identical to old? :
    report new : replace old with new of rest list
else if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

	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 ]
```

### 10.2.10 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、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。

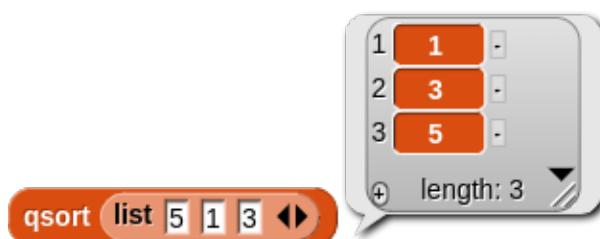


このように自分自身を複数参照することを多重再帰と言います。

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。



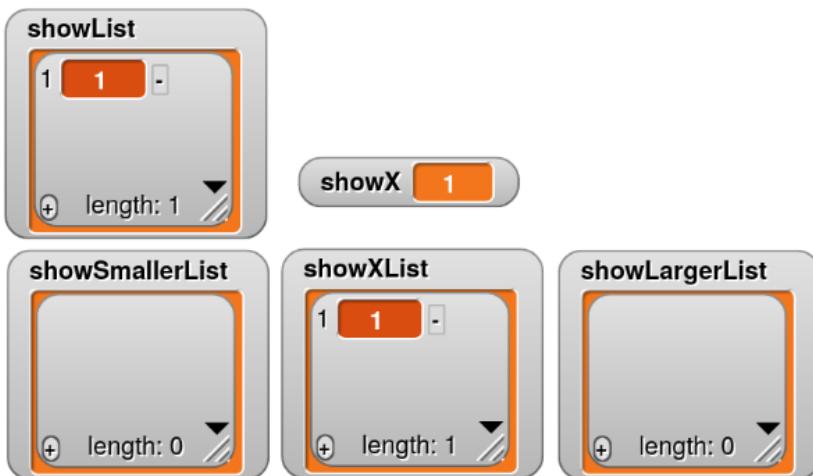
リストの値表示のたびに pause all になります。 をクリックして続行してください。



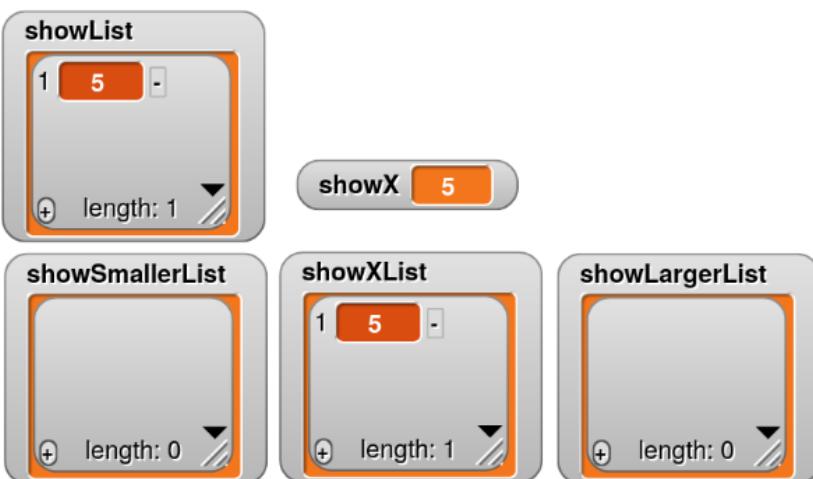
最初に選択された値が 3 だった場合です。



3 より小さい値のグループ処理



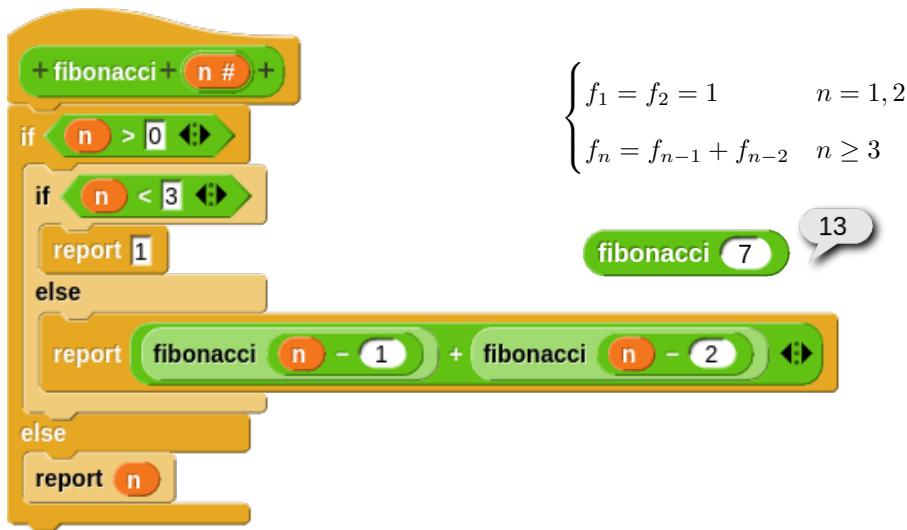
3 より大きい値のグループ処理



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。

### 10.2.11 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。

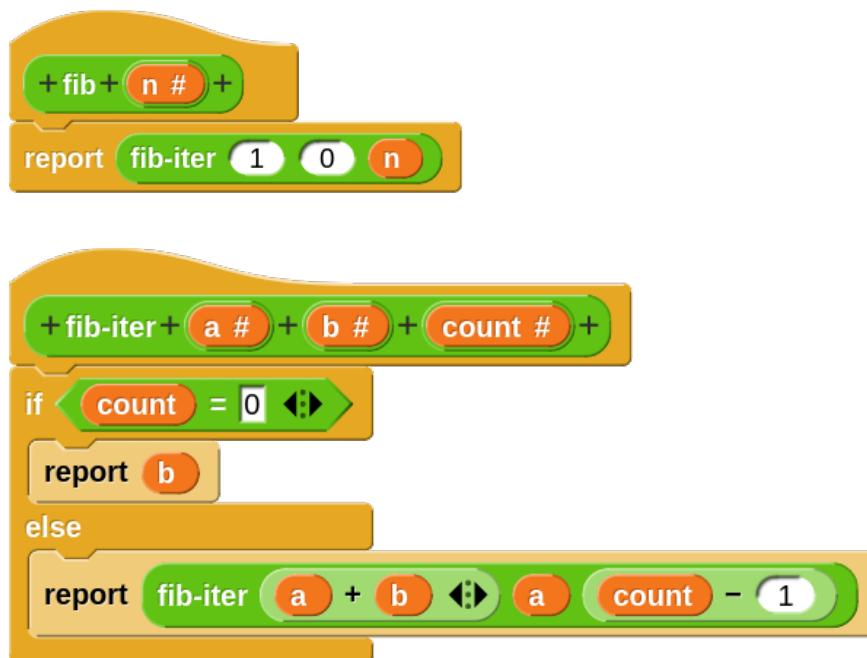


これは  $(n - 1)$  と  $(n - 2)$  を引数にして 2 度再帰呼び出しています（多重再帰）。そのため、 $n$  が大きくなると時間がかかるてしまいます。

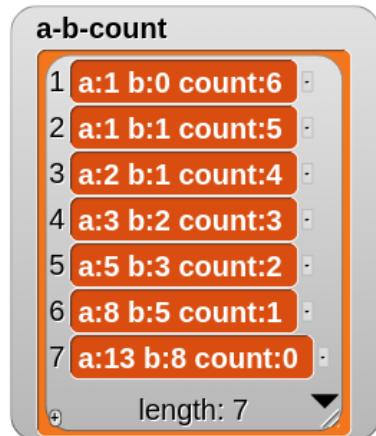
例えば  $\text{fib}(6)$  を求める場合は、右のようになりますが、 $\text{fib}(6)$  で必要とする  $\text{fib}(4)$  は  $\text{fib}(5)$  で処理済みで、 $\text{fib}(5)$  で必要とする  $\text{fib}(3)$  は  $\text{fib}(4)$  で処理済み … ということで、処理済みのものはその値を渡してやれば済むのでその分の再帰呼び出しの必要がなくなります。

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

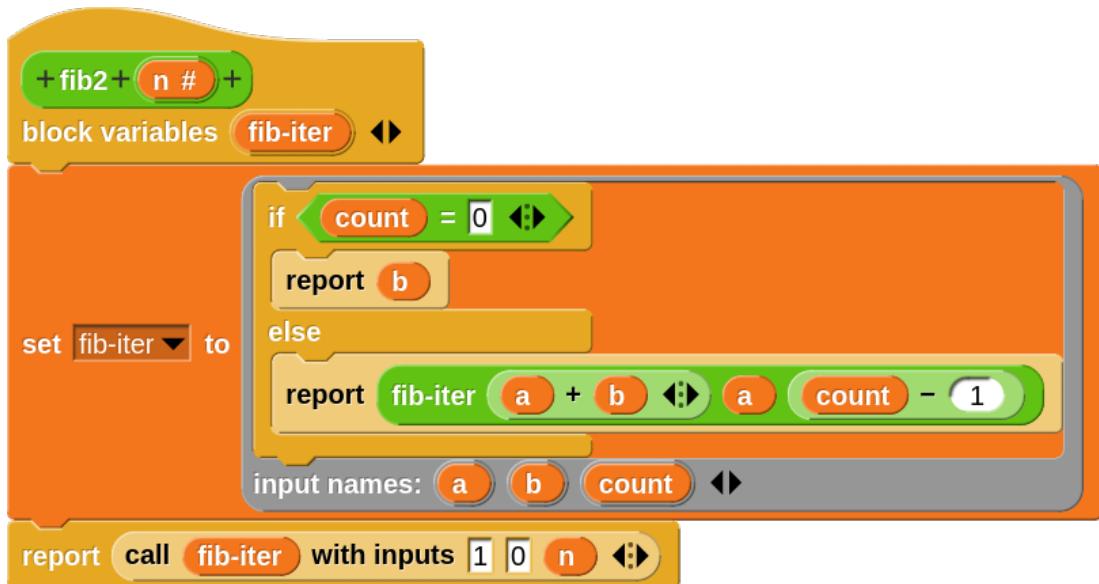
SICP の非公式日本語版翻訳改訂版 真鍋宏史氏訳 の 39 ページに反復プロセス版が載っています。前二項の値を渡すことで余分な再帰呼び出しを避けています。



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。

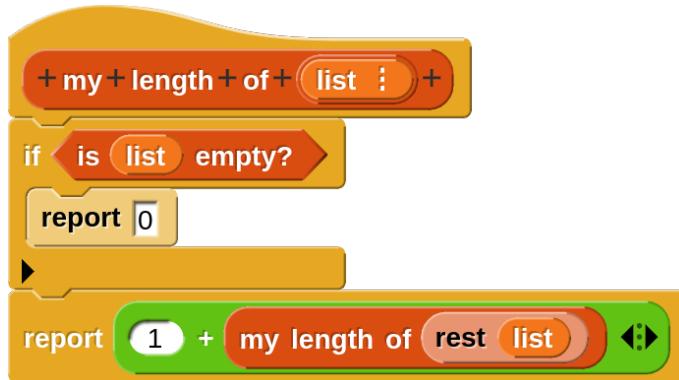


fib-iter はここでしか使わないので、本体内部で変数にセットすると局所定義ブロックにすることができます。



### 10.2.12 末尾再帰

145 ページで my length を扱いました。



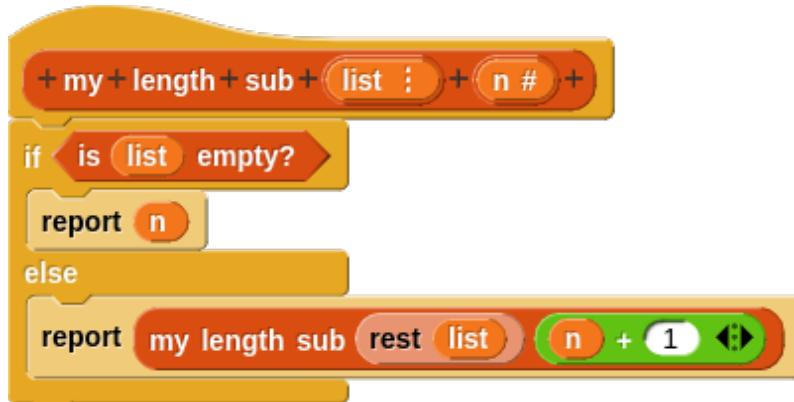
この定義ブロックでは `report [1 + my length of rest list]` で、計算式に再帰呼出しが含まれていて、再帰呼出しによる値が確定しないと計算することができません。リストが空になると確定した 0 の値を使って順々に計算した値を戻しながら一番最初のところまで戻って、ようやく値 3 を得ることになります。my length of (1 2 3) を L(1 2 3) と表してみます。

$L(1 \ 2 \ 3)$

$$\begin{aligned}
& 1 + L(2 \ 3) \\
& \quad 1 + L(3) \\
& \quad \quad 1 + L() \\
& \quad \quad \quad 1 + 0 \\
& \quad \quad 1 + 1 \\
& \quad 1 + 2 \\
& \quad \quad 3
\end{aligned}$$

これに対して、次のようにすると再帰呼出した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。

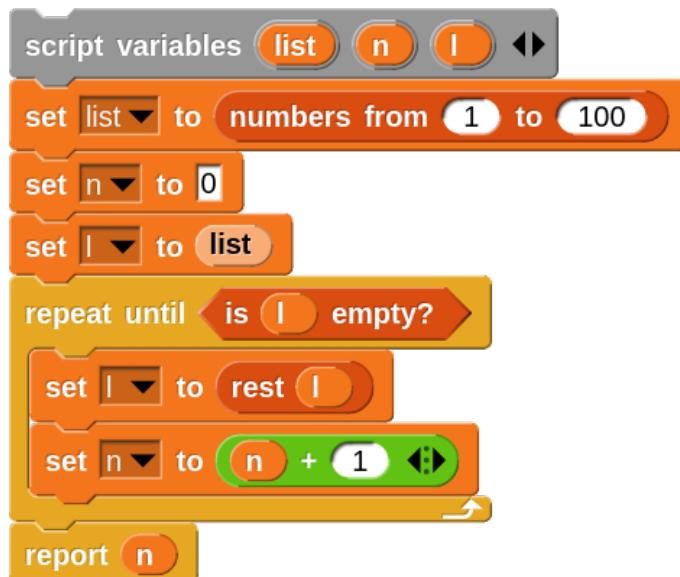


これを呼び出す本体定義は次のようにになります。引数  $n$  の値を 0 にすることで、カウンターの値を初期化しています。



```
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
```

このように、一番最後の再帰呼出しの返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。末尾再帰のスクリプトは再帰を使わない反復のスクリプ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。

## 11 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。

### 11.1 高階関数型ブロックの基本

半径を入力して、円の面積をリ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。

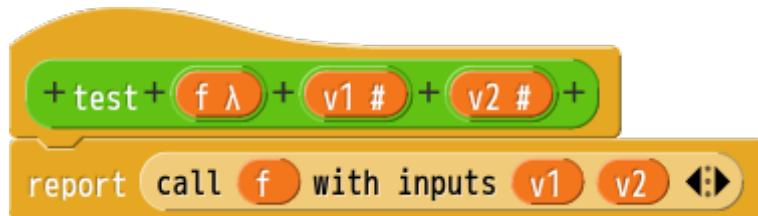


map を二重にするとこんな使い方もできません。



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 リングで囲つ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。

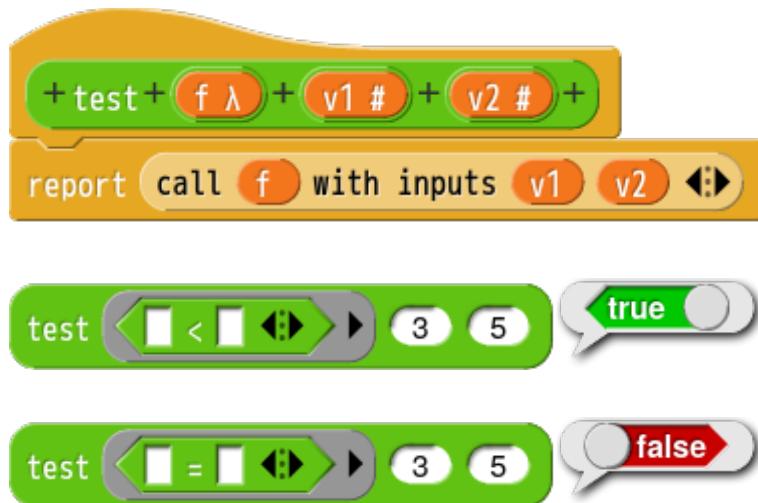
次のブロック定義 test は二つの引数を必要とする関数型ブロックを実行して値を返すものです。入力 f は Reporter 型です。v1 と v2 の入力は Number 型です。関数型ブロックの実行は call ブロックを使用します。これが高階関数型の定義ブロックの基本となります。



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。



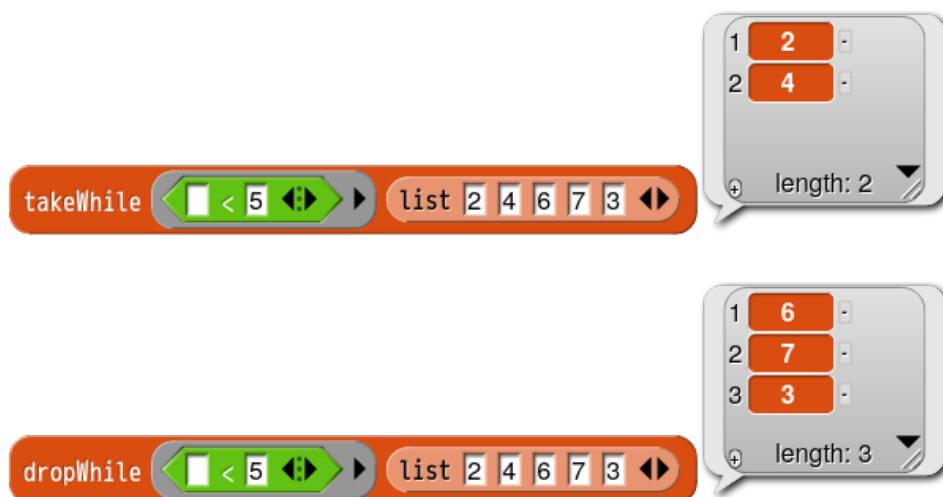
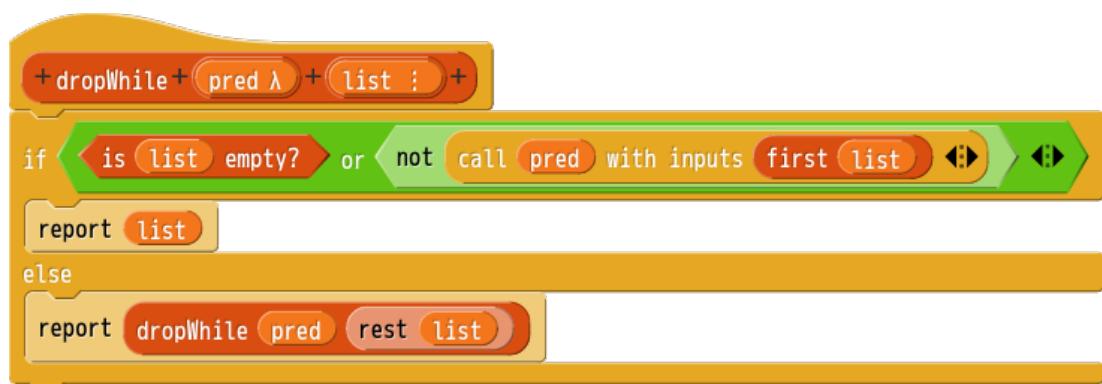
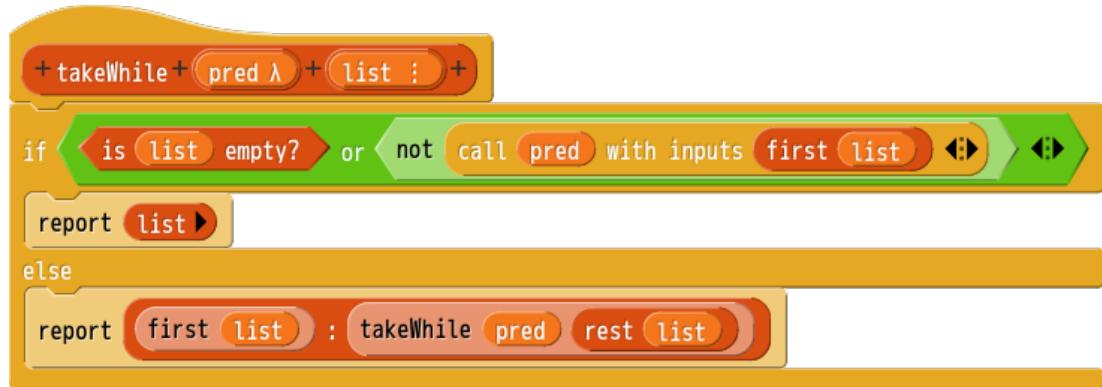
入力 f を Predicate 型にすると、次のように変更されます。( ブロック定義自体の見た目は変わりませんが )



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。

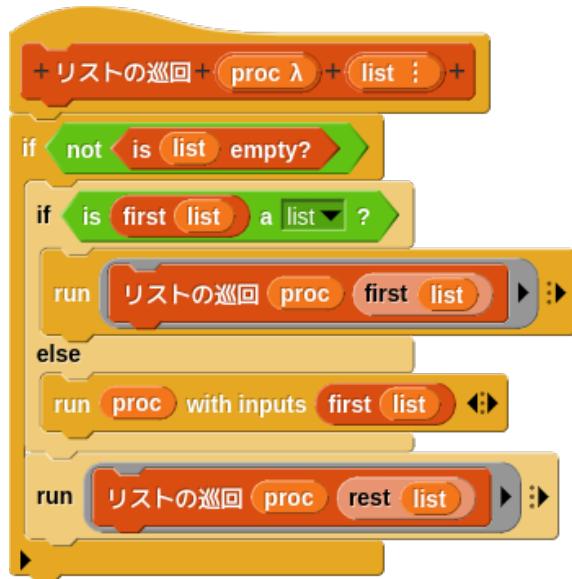
## 11.2 高階関数型の take と drop

148 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。



### 11.3 操作を指定するリストの巡回

151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。

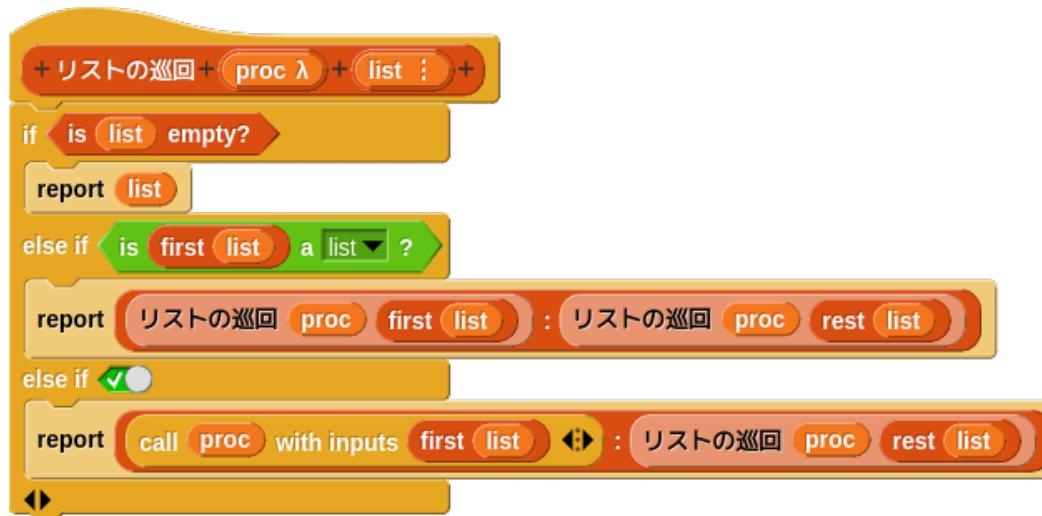


リストの要素の値の合計値を求めます。上とは違い[change (n) by ()]と引数のスロットにリストの要素が入り加算されます。

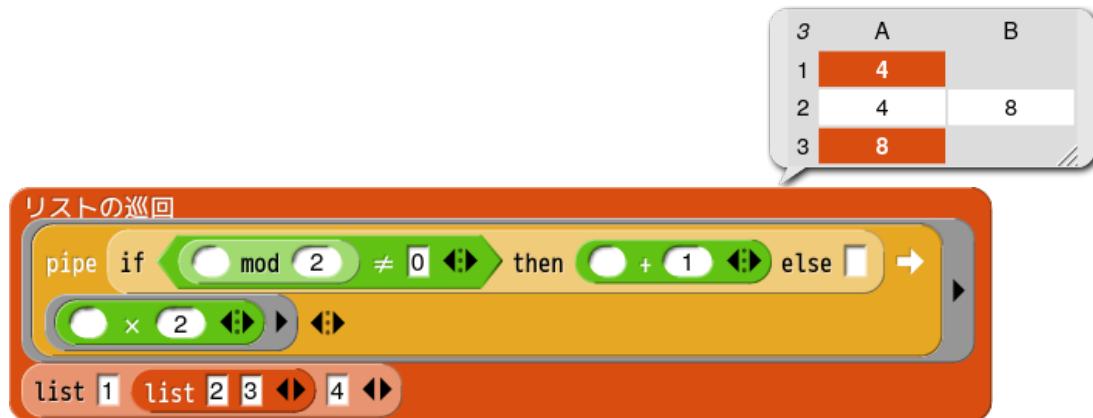


proc には引数として各要素を使用するスクリプトブロック入れることができます。

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。



## 11.4 foldl, foldr

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

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なります。 (#1, #2 から名前の変更をしています)

foldl

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

The script uses an if-else structure. If the list is empty, it reports the current value (init). Otherwise, it calls the proc block with inputs init, first list, and rest list.

非再帰版だとこうなります。

The script uses a for each item in list loop. Inside, it sets init to the result of calling proc with inputs init and item, and then reports init.

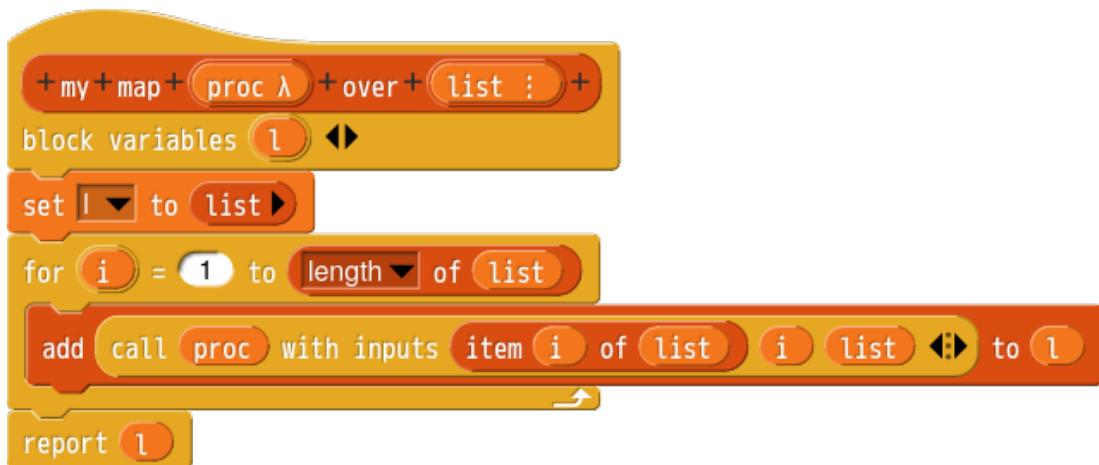
次のようにすると、リストの要素の値は使用しないで要素数 length を求めることになります。



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。



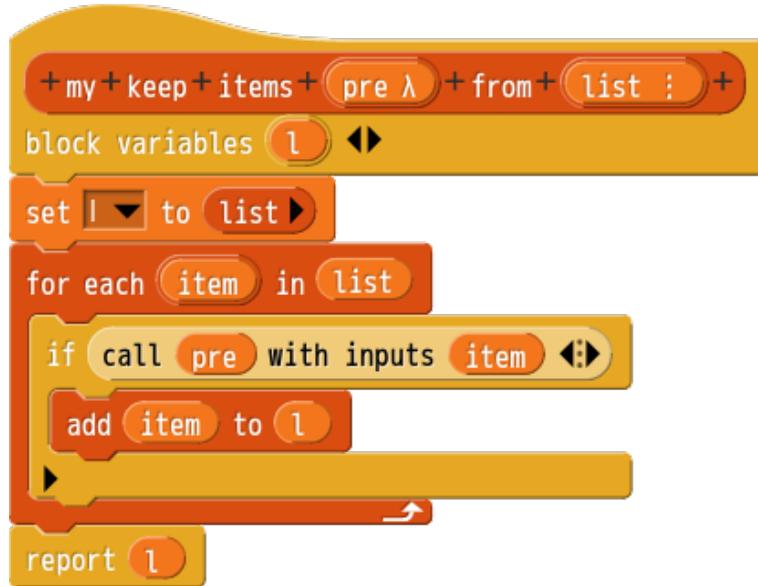
再帰呼び出しを使用して作成してみます。 value, index, list 値は利用できません。



foldl 版です。



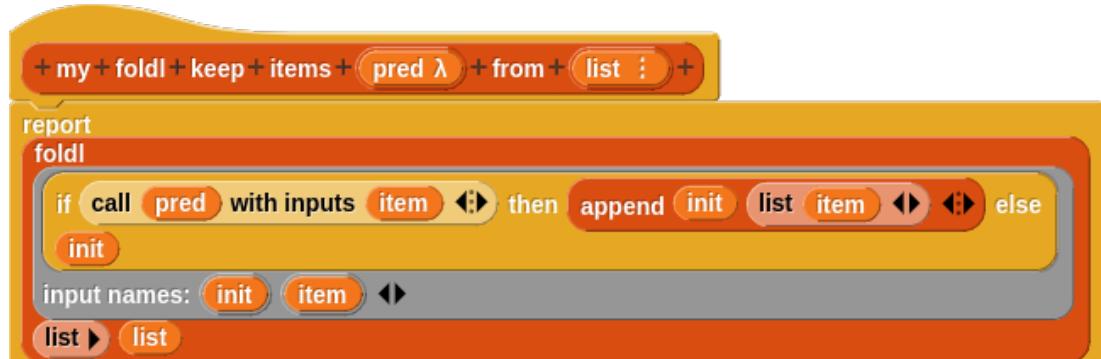
keep ブロックを再帰呼び出しを使用しないで作成してみます。引数 pred は、Predicate 型です。



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。



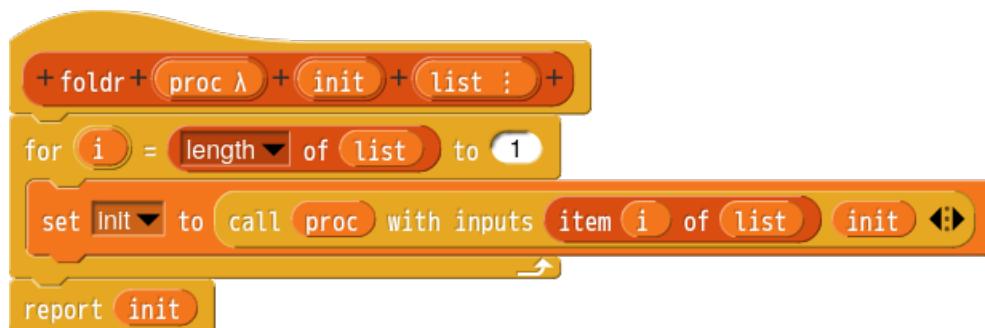
フォーマルパラメータの順序は右下の表のようになります。foldl とは違う item, init の順になります。

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。

12



foldr を使っても foldl と同じようなことができます。

The image shows two Scratch scripts demonstrating the implementation of foldr using my functions.

The first script uses the following blocks:

- report
- foldr
- call proc with inputs [item : init] input names: [item init]
- [list ▶ list]

The second script uses the following blocks:

- report
- foldr
- if [call pred with inputs [item : init] then [item : init] else [init]]
- [input names: [item init] list ▶ list]
- [list]

foldr がリストを右側から操作することを利用した append です。

The image shows a Scratch script demonstrating how foldr can be used to implement append.

The script uses the following blocks:

- report
- foldr
- item : init
- [input names: [item init] list2 list1]

To the right of the script is a data stack containing the following values:

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

Below the data stack is a list operator block:

my foldr append [numbers from 1 to 3] [numbers from 4 to 6]

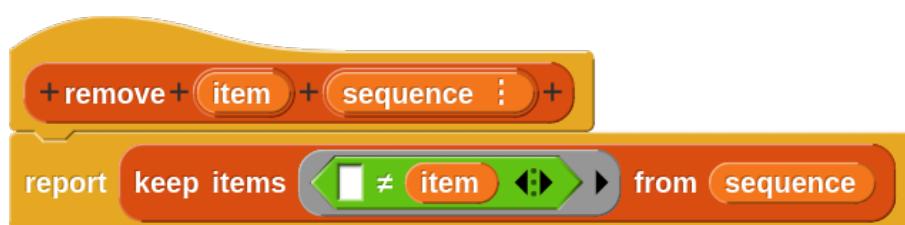
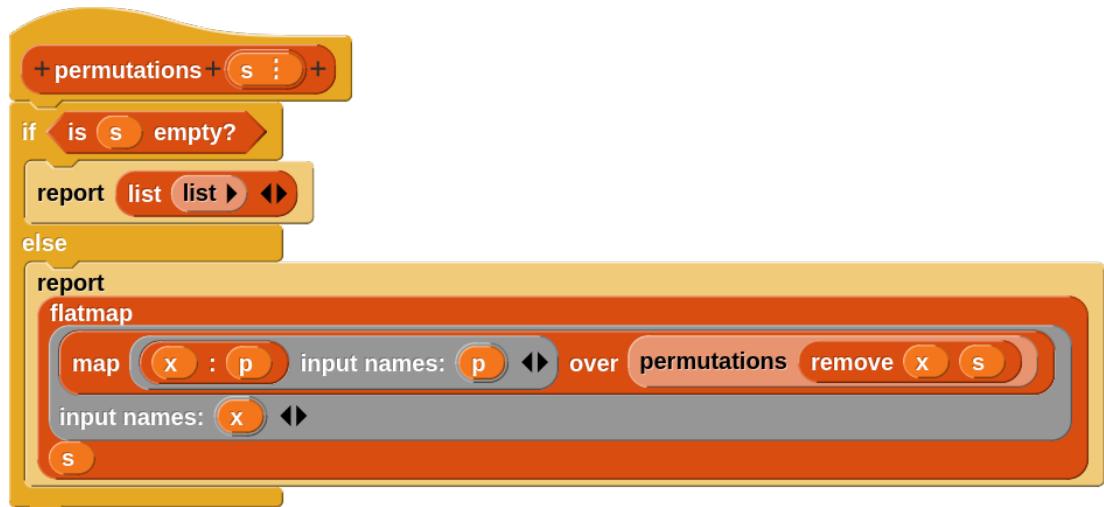
foldl で作成する場合には、リストを逆順にする操作が必要になります。

The image shows a Scratch script demonstrating how foldl needs to reverse the list for append.

The script uses the following blocks:

- report
- foldl
- item : init
- [input names: [init item] list2 reverse of list1]

SICP の非公式日本語版翻訳改訂版 真鍋宏史氏訳 の 133 ページに permutations という順列を作成するスクリプトが載っています。この中の flatmap という関数には accumulate という名前で foldr が使われています。

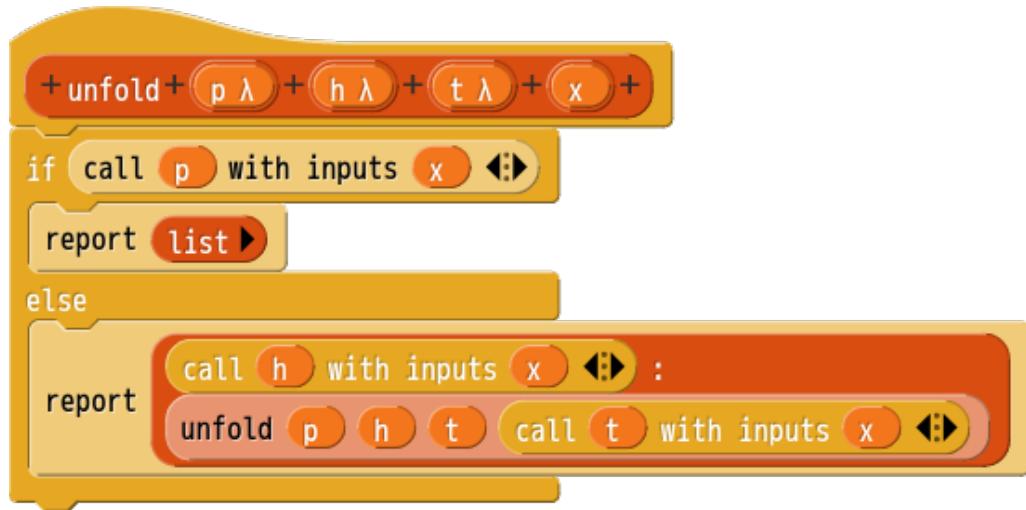


	A	B	C
1	1	2	3
2	1	3	2
3	2	1	3
4	2	3	1
5	3	1	2
6	3	2	1

permutations numbers from 1 to 3

## 11.5 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。

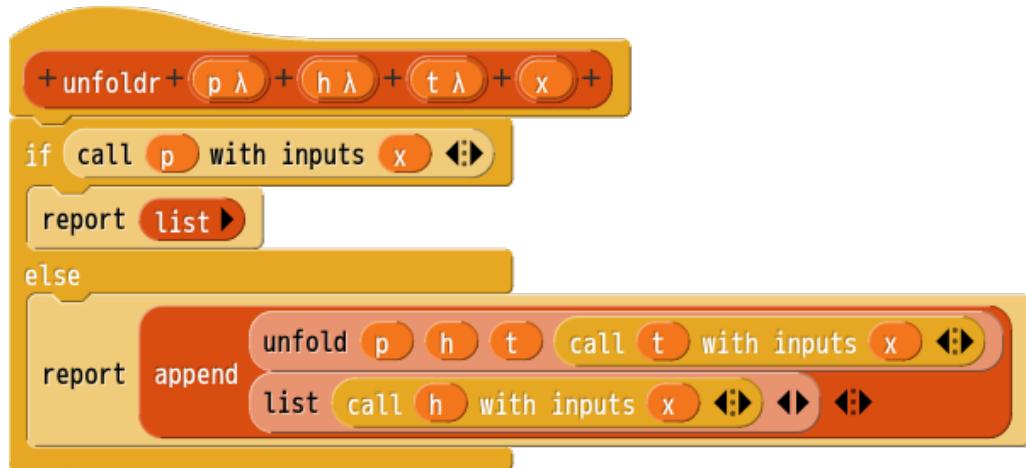


逆の操作は `unfold` を使用すると、次のようにできます。

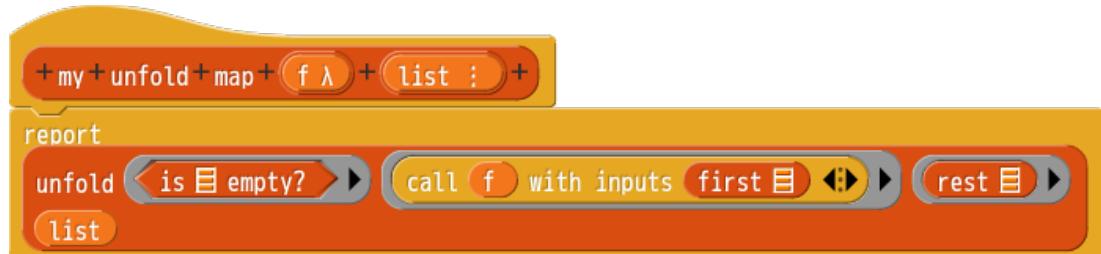


値を 2 で割り切ると商と余りが求められます。余りが最下位桁の値になります。商は次の桁を求めるための値になります。この操作を商が 0 になるまで行います。リストは逆順で作られるので `reverse` する必要があります。（115 ページ参照）

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



unfold を利用して map を作成することもできます。



## 11.6 concat

Haskell 言語には、リストの要素がリストである

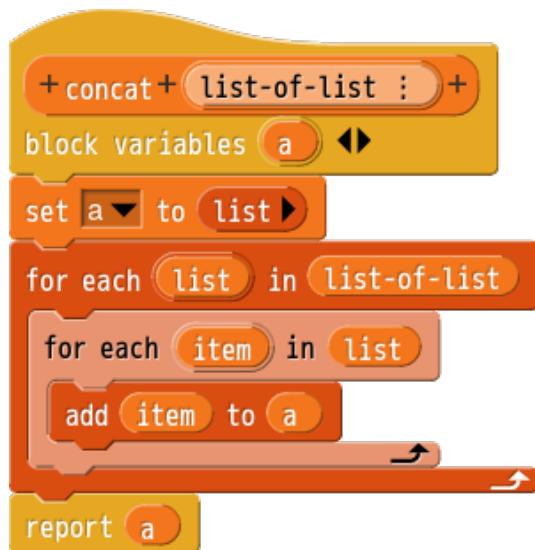
list [list 1 ◆◆] list 2 ◆◆ list 3 ◆◆ の要素を結合した

list [1 2 3 ◆◆] を返す関数 concat があります。

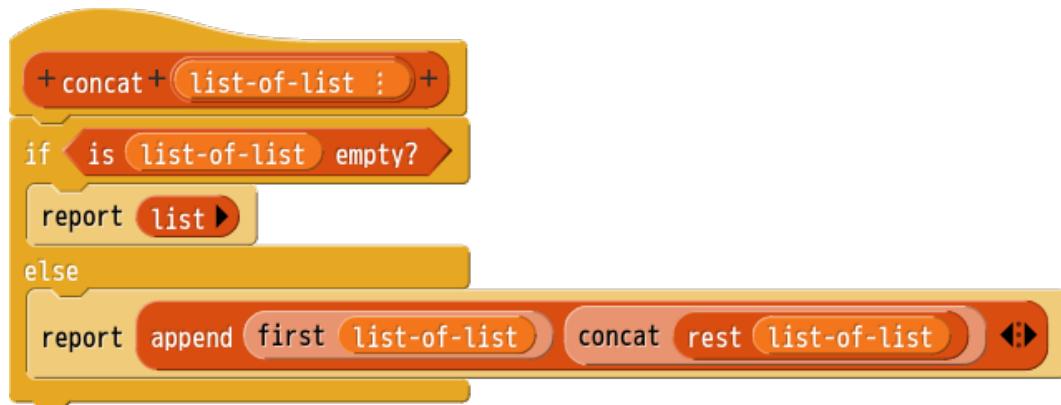
これによりランクが一つ下がります。



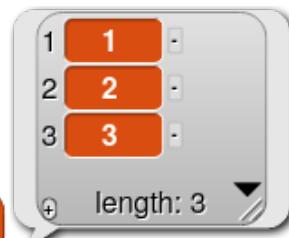
for each ブロックを使った定義は次のようになります。



再帰呼び出しを使うと次のようになります。



foldr ブロックを使うと次のようにになります。



この処理内容は次のようになります。



なお、append と ( ) in front of ( ) は似ていますが、append がリストを結合するのに対して、( ) in front of ( ) はリストの先頭に要素を挿入するものなので違う結果になります。



実は、append の右端の左右向きの三角形部分にリストを入力することができるので、append ブロック自体で concat ができます。



concat を考えることで append や ( ) in front of ( ) の理解が深まりました。

## 11.7 カリー化

複数個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します。関数を返す関数ということで高階関数の例として示されることがあります。Snap! ではスロットを増設して複数の引数をブロックに渡すことができるので必要なことですが、シミュレートしてみるとリングの理解が深まります。

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。join-c は次のようにになります。



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。



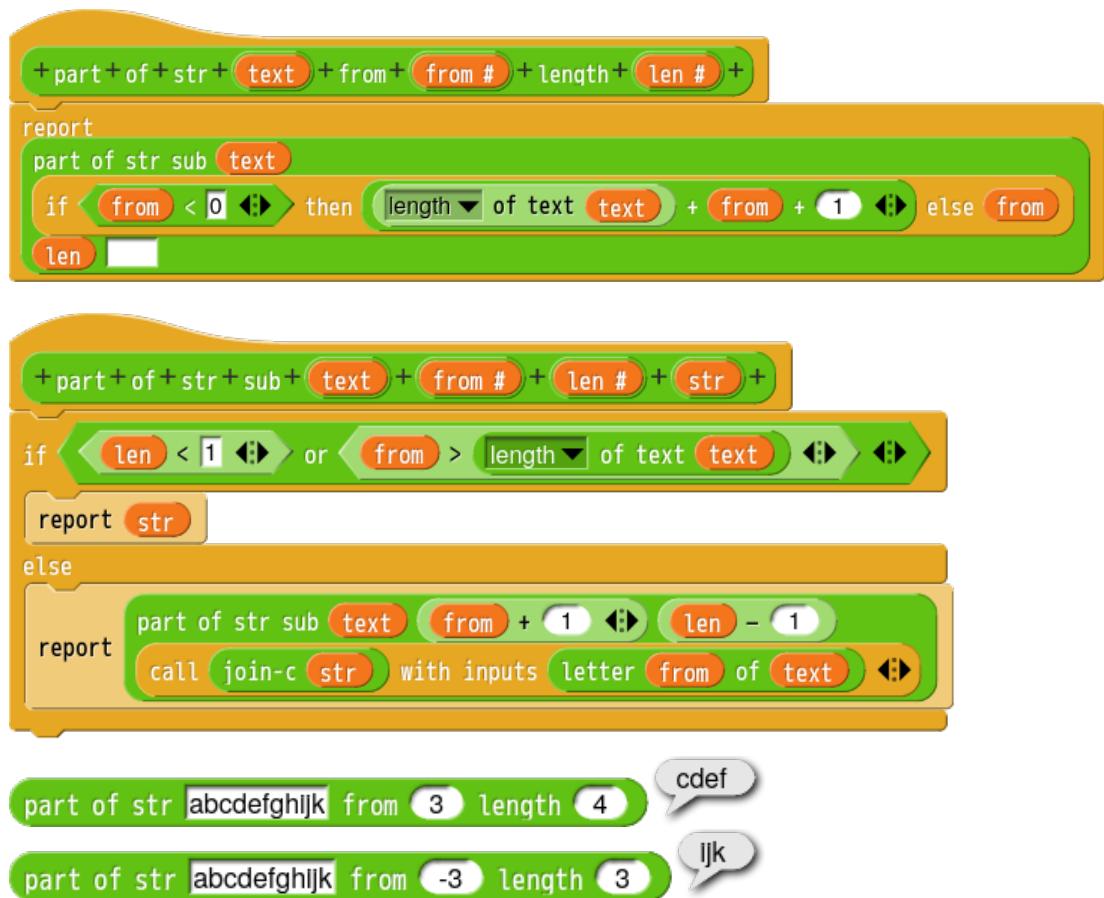
join-c ブロック部分を右クリックして unringify する必要があります。



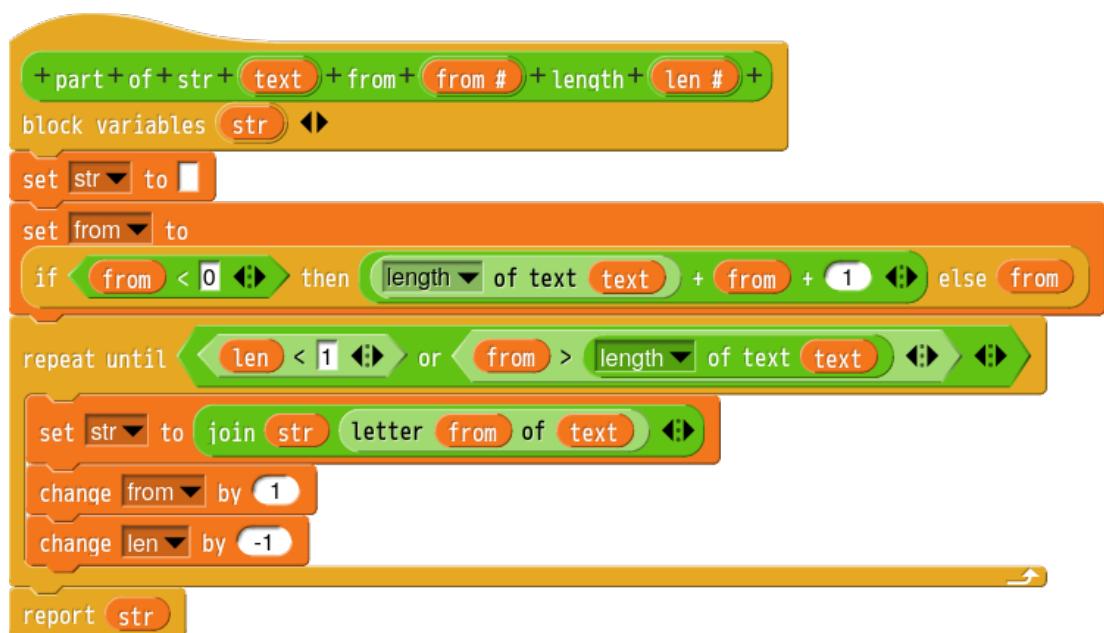
この文字を追加するブロックを連ねることで複数の文字を追加することができます。



join-c ブロックを使って、文字列から部分文字列を作成するブロックを作ってみます。引数として、「元になる文字列」「文字列の何番目から」「部分文字列の長さ」を指定します。「何番目から」を負数にすると末尾から何番目になります。part of str sub ブロックに渡される最後の文字列は空文字列です。空白ではありません



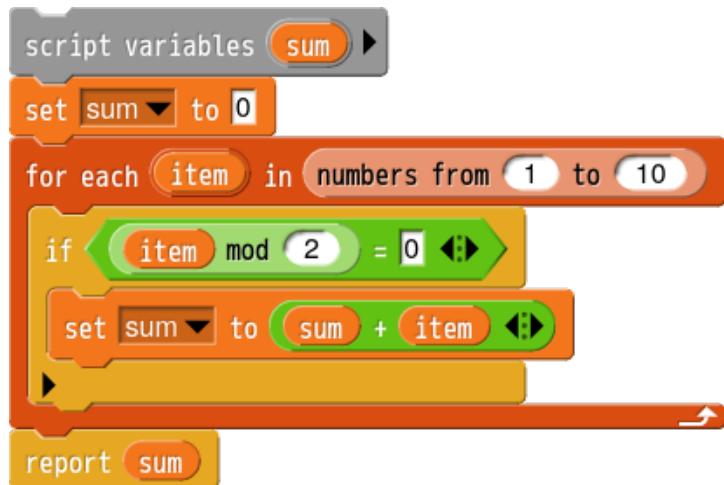
カリー化ブロックを使わない非再帰版です。



## 11.8 ブロックの合成

高階関数型ブロックの入力スロットに高階関数型ブロックを入れてやると、より複雑な機能のブロックになります。

リストの偶数要素を合計することを命令型プログラミングで行うと次のようにになります。



関数型プログラミングだと次のようにになります。



この場合は二つのブロックの合成なので読み解くのはそれほど難しくはありませんが、もっと複雑になると分かりにくくなります。pipe ブロックを使用すると、データの流れが操作順で表示されるので分かりやすくなります。



操作をリストにして、foldl で実行することもできます。



## 索引

$\Sigma$  of ( ), 33  
: プロック, 144  
  
all, 88, 95  
all but first of, 21  
all<, 95  
all=, 95  
all>, 95  
AND, 113  
any, 95  
append, 22, 176  
atan, 107  
atan2, 107  
  
block variables, 66  
break, 137  
  
call, 40  
case, 81  
catch, 135  
collapse, 97  
columns, 124  
columns of ( ), 31  
combine, 27, 116  
composition, 54  
concat, 175  
continuation, 132  
continue, 137  
Costumes, 10  
csv, 7, 14, 36  
csv of ( ), 34  
  
deep contains, 152  
delete, 153  
dimensions of ( ), 30  
distribution of ( ), 32  
draggable?, 10  
drop, 148  
dropWhile, 164  
  
end thread, 139  
Event Hat プロック, 89  
  
factorial, 143  
find first item, 27, 93  
first プロック, 144  
flatten of ( ), 31  
for プロック, 53, 79  
  
getc, 85  
identical, 87  
import, 10, 36, 54  
( ) in front of ( ), 21, 176  
initial slots, 97  
inner product, 122  
input list:, 40  
Iteration, 54  
iteration-composition, 135  
  
JavaScript, 111  
JavaScript extensions, 6  
join, 28, 86  
json, 7, 14, 36  
json of ( ), 34  
  
keep, 26, 149, 153, 155, 169  
  
Language, 6  
letter, 115  
Libraries, 10  
lines of ( ), 34  
list view, 19  
  
map, 24, 25, 88, 154, 168  
multi-line, 75  
multiple inputs, 82  
  
New, 10  
NOT, 113

object, 76  
Open, 10  
OR, 113  
outer product, 129  
pen trails, 37  
permutations, 172  
pipe, 47, 128, 130, 131, 179  
product, 95  
qsort, 155  
rank of ( ), 30  
read-only, 67, 94  
relabel, 12  
replace, 154  
reshape, 23  
rest プロック, 144  
reverse of ( ), 33  
ringify, 37, 58  
rotation style, 9  
run, 39  
Save, 10  
Save As, 10  
scene, 10  
script pic, 12  
separator, 97  
shuffled of ( ), 33  
sorted of ( ), 32  
split, 28  
sum, 95  
switch, 81  
table view, 19  
take, 148  
takeWhile, 164  
text of ( ), 33  
thread, 139  
throw, 135  
transient, 15  
turbo mode, 11  
unfold, 173  
unique, 149  
uniques of ( ), 32  
unringify, 37  
Upvar 変数, 18  
UTC, 116  
with continuation, 132  
with inputs, 40  
XOR, 113  
yield, 139  
Zoom, 6  
イベント, 89, 100  
エラトステネスのふるい, 26  
オフライン版, 6  
階乗, 27, 143  
回転行列, 123  
角度, 107  
カスタムプロック, 8, 48  
カリー化, 177  
協定世界時, 116  
行列の積, 122  
クイックソート, 155  
グローバル変数, 14, 15  
継続, 132  
高階関数, 162  
再帰呼び出し, 143  
座標変換, 123  
辞書, 93  
ジューケボックス, 8  
順列, 172  
スクリプトエリア, 8

スクリプト変数, 17  
ステージエリア, 7  
ステップ実行, 9, 53, 83  
スプライトコラル, 8  
スプライト変数, 16  
スレッド, 139  
ゼブラカラーリング, 12  
素数, 26, 130, 149  
ターボモード, 11  
大域変数, 15  
多重再帰, 155, 158  
単位行列, 129  
定義ブロックのインポート, 65  
定義ブロックのエクスポート, 65  
定数, 48  
デバッグ, 9, 83  
転置行列, 31, 124  
時計, 116  
二進数, 113  
日本語化, 6  
排他的論理和 (XOR), 113  
配列, 23  
バックグラウンド, 8  
ハノイの塔, 143  
パレットエリア, 8  
八口, 13  
ビット演算, 113  
ビット反転 (NOT), 113  
評価, 76, 78  
フィボナッチ数列, 24, 158  
フォーマルパラメータ, 24, 41  
プリミティブブロック, 8  
ブロック内変数, 66  
プロトタイプ, 49, 53, 72, 73  
並列実行, 120  
変数ウォッチャー, 7, 15, 19, 20, 36  
末尾再帰, 160  
無名関数, 41  
ラムダ関数, 41  
リスト要素の巡回, 151, 165  
リング, 24, 37, 40, 44, 57, 75, 76, 78  
連想配列, 93  
ローカル変数, 16  
ロケーションピンアイコン, 17  
論理積 (AND), 113  
論理和 (OR), 113  
ワードローブエリア, 8  
ワープ, 11