

# Snap! のこと

齋藤文康

2023 年 7 月 18 日

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

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

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

# 目 次

1 始め方	5
2 画面まわり	5
2.1 エリア . . . . .	6
2.1.1 ステージエリア . . . . .	6
2.1.2 スプライトコラル . . . . .	7
2.1.3 スクリプトエリア . . . . .	7
2.1.4 パレットエリア . . . . .	7
2.2 エリアの大きさ . . . . .	8
2.3 実行に関するボタン . . . . .	8
2.4 ブロック表示 . . . . .	11
3 キーボード入力	12
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 for ループ変数 . . . . .	18
4.5 リスト処理用ブロック . . . . .	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 ( ) . . . . .	21
4.5.5 append ( ) ( ) . . . . .	22
4.5.6 for each ( ) in ( ) . . . . .	22
4.6 reshape ( ) to ( ) ( ) . . . . .	22
4.6.1 map ( ) over ( ) . . . . .	24
4.6.2 keep items ( ) from ( ) . . . . .	26
4.6.3 find first item ( ) in ( ) . . . . .	27
4.6.4 combine ( ) using ( ) . . . . .	27
4.6.5 combinations ( ) ( ) . . . . .	28
4.7 リストの演算 . . . . .	29
4.8 変数に入れられるもの . . . . .	30
5 Control 制御	33
5.1 リポーターの if then else . . . . .	33
5.2 when ( ) . . . . .	33
5.3 stop ボタンがクリックされた時の終了処理 . . . . .	34

5.4 run ブロック . . . . .	35
5.5 call ブロック . . . . .	36
5.6 launch ブロック . . . . .	38
5.7 broadcast ブロック, tell to ブロック . . . . .	41
<b>6 ブロックを作成する</b>	<b>43</b>
6.1 >= ブロック . . . . .	43
6.2 help 説明文の作成 . . . . .	47
6.3 for i = start to end step add . . . . .	48
<b>7 ブロック定義について</b>	<b>61</b>
7.1 プルダウン入力 . . . . .	61
7.2 Title Text とシンボル . . . . .	65
7.3 Input name オプションについて . . . . .	67
7.3.1 Reporter 型 . . . . .	67
7.3.2 Predicate 型 . . . . .	70
7.3.3 Command 型 . . . . .	71
<b>8 Continuation 繙続</b>	<b>77</b>
8.1 w/continuation . . . . .	77
<b>9 その他</b>	<b>82</b>
9.1 デバッグ . . . . .	82
9.2 ask . . . . .	84
9.3 broadcast の検索オプション . . . . .	84
9.4 クローン . . . . .	85
9.4.1 テンポラリクローン . . . . .	85
9.4.2 パーマネントクローン . . . . .	87
9.5 flat line ends . . . . .	89
9.6 anchor アンカー . . . . .	90
9.7 JavaScript function (オプション 5 ページ参照) . . . . .	93
9.8 時計 . . . . .	99
9.9 並列処理について . . . . .	102
<b>10 再帰</b>	<b>111</b>
10.1 再帰の例 . . . . .	111
10.1.1 階乗 . . . . .	111
10.1.2 ハノイの塔 . . . . .	111
10.2 再帰の使用 . . . . .	112
10.2.1 繰り返し . . . . .	112
10.2.2 カウントダウンとカウントアップ . . . . .	113

10.2.3 my length . . . . .	113
10.2.4 my contains . . . . .	115
10.2.5 リスト要素の巡回 . . . . .	116
10.2.6 reverse 逆順リスト . . . . .	118
10.2.7 クイックソート（整列 / 並べ替え） . . . . .	119
10.2.8 再帰呼び出しをする局所的な定義プロック . . . . .	122
10.2.9 末尾再帰 . . . . .	125
<b>11 高階関数</b>	<b>128</b>
11.1 カリー化 . . . . .	133
11.2 OOP オブジェクト指向プログラミング . . . . .	134
<b>12 APL ライブラリー</b>	<b>137</b>
12.1 形 . . . . .	137
12.2 配列の連結 . . . . .	140
12.3 配列要素の配置転換 . . . . .	140
12.4 ベクトル、配列の範囲指定、選択 . . . . .	141
12.5 配列要素に対する演算 . . . . .	143
12.6 outer product . . . . .	145
12.7 inner product . . . . .	147
12.8 ソート、順位付け . . . . .	154

## 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。

画面構成は Scratch と似ています。日本語化もできますが、英語のままのほうが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、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。ただし、これはブロックのドロップ（どこに移動させたか）に関する限りで、入力スロットに設定した値を元に戻すことはできないようです。

 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ちます。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。

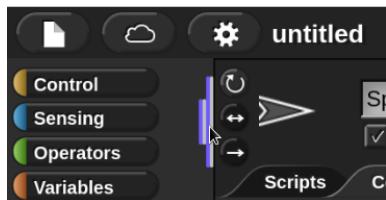


のボタンはカスタムブロックを作成する時に使います。その上のボタンはブロック検索用です。カスタムブロック（ユーザー定義ブロック）とは、ブロックエディターで作成、修正可能なブロックです。それに対して、プリミティブブロックは、Snap! に備わっているブロックです。

## 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。一番左だと一動作ごとの  
ステップ実行になります。デバッグの時に使えます。(82 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。



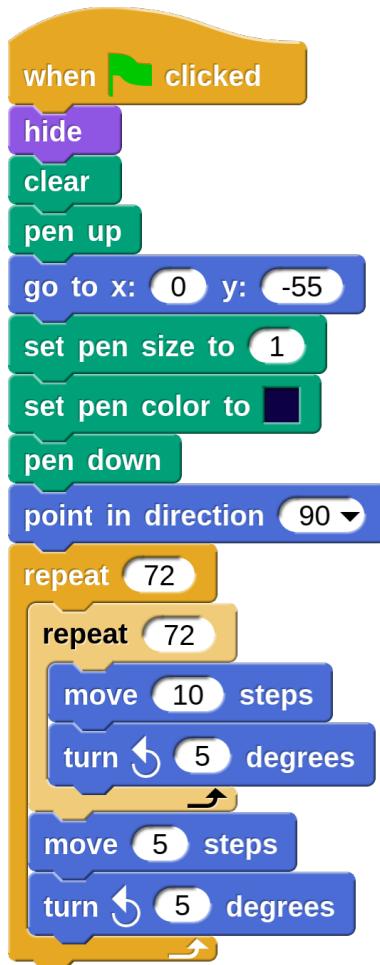
スプライトをマウスでドラッグできるかを設定するのが、 **draggable** です。スクリプトで設定するのが、 です。こちらは、true か false で設定します。

 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。

次を実行すると、かなり時間がかかります。



[Shift] を押しながら [N] 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。この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。



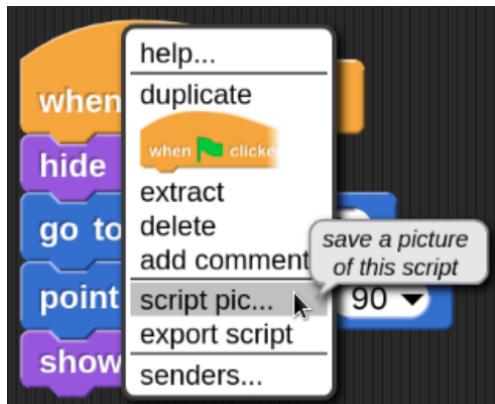
## 2.4 ブロック表示

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

このような長い式を Snap! でスクリプトにすると、

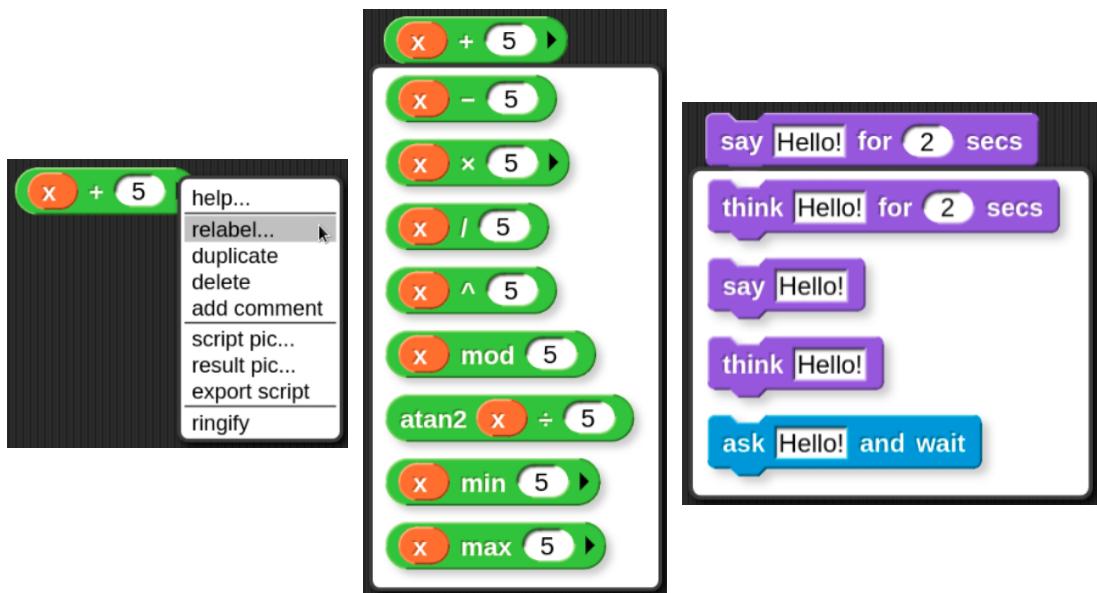


となります。長くなった場合は、自動的に折り畳まれます。また、同じパレットのブロックが重なった場合は、色合いが交互に変化して表示されます。ゼブラカラーリングだそうです。



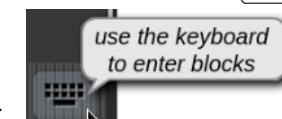
スクリプトをプリンターで印刷したい場合は、スクリプトを右クリックすると、script pic ...で、画像ファイルとしてダウンロードフォルダへエクスポートされます。ファイル名はプロジェクト名 + script pic + (番号) + 画像ファイルを表す拡張子になります。

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。Operators の場合はパレットに無いブロックも使えたりします。



### 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。

**when green flag clicked** を出すために、「when」の最初の文字 **W** 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。

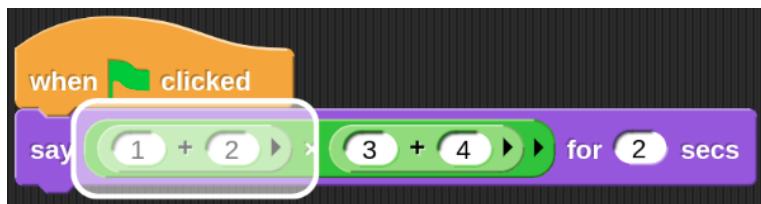
次に、**say Hello! for 2 secs** です。「say」から **S** **a** と打つと欲しいものが得られます。スクリプトエリアにセットされると、入力スロットの部分が白く（ハロ）囲われています。



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、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 变数

变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** **a** を利用する。
- for ブロックなどの变数を利用する。
- .txt .csv .json などのファイルを Snap! 画面にドロップする。

英語版のままでも变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变数)

变数は、有効になる (变数が見える) 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 varAll が表示されます。こ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、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、var 0 です。

**set var to url snap.berkeley.edu** をスクリプトエリアに置きます。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 8.6KB でした。これを実行すると、変数 var に Snap! のホームページの html ファイルが入ります。(オンラインの Snap! を使用のこと)

```

var <!DOCTYPE html>
<html>
  <head>
    <meta name="snap-cloud-domain" location="https://snap.berkeley.edu:443">
      <meta charset="UTF-8">
      <title>Snap! Build Your Own Blocks</title>
      <meta name="description" content="The Snap! Community. Snap! is a blocks-based programming language built by UC Berk</meta>
      <meta name="author" content="Bernat Romagosa, Michael Ball, Jens Mönig, Brian Harvey, Judge Hügle">
...

```

この状態で save します。すると、ファイルの容量は 91.0KB 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の容量を大きくするということです。transient をチェックして save すると、ファイルの容量は 12.9KB 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。

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団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。



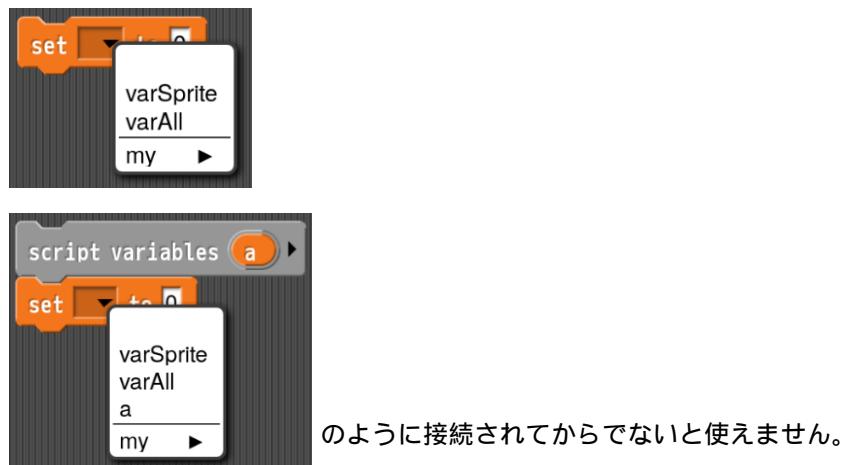
#### 4.3 script variables スクリプト変数

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。



a の隣りにある右向きの三角をクリックすると変数を追加できます。  
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。





のようにその前でも有効なりません。

#### 4.4 for ループ変数

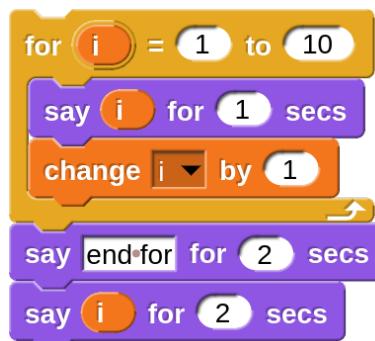
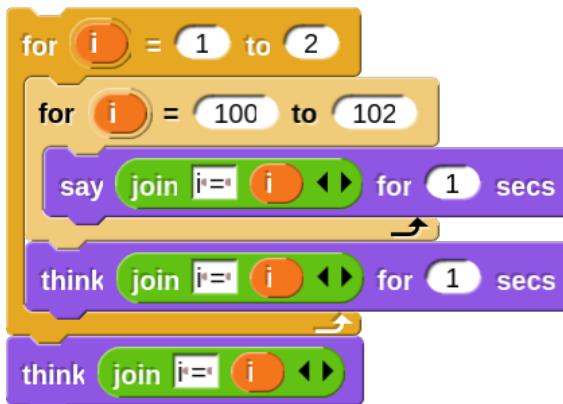


このようにあらかじめ用意されている変数があります。

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。この変数は、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。



このようにしても、内側のループと外側のループは混乱せずに動くようですが、こんなことはしないほうがいいです。適切な名前を使用しましょう。

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 リスト処理用ブロック

Snap! には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。



で aList という変数を作成すると、ステージエリアに が表示されます。 の左向きの三角をクリックして とすると、空のリストができます。 の右向きの三角をクリックして で

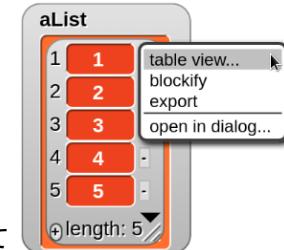
値を入れてやると(左端の入力スロットに1を入れて、**Tab**キーを押すと次の入力スロットに移

れます)、aListは要素を持つリストになり、ステージエリアに



が表示されます。

**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  
に変更することができます。また右ク  
リックしてlist view...を選択すると元に戻ります。



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。



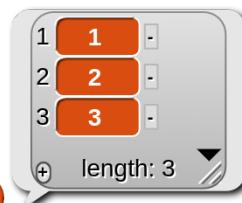
変数ウォッチャーの内部を右クリックして、blockify をクリックすると、リストブロックとして取り出すことができます。

list 1 2 3 4 5 ◀▶

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、変数の内容がテキスト表示できてリスト以外ならば .txt 形式で、リストの場合は .csv 形式、複雑なリストの場合は .json 形式の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。



たとえば、aList や list 1 2 3 ◀▶ のように。

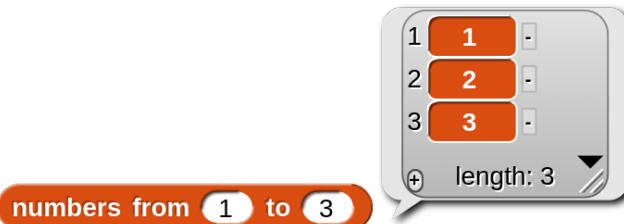
なお list view では、一度に扱える要素数は 100 までになっています。次の 101 から 200 の要素



に移るには、下向きの三角をクリックすると出てくる範囲から選びます。

#### 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。



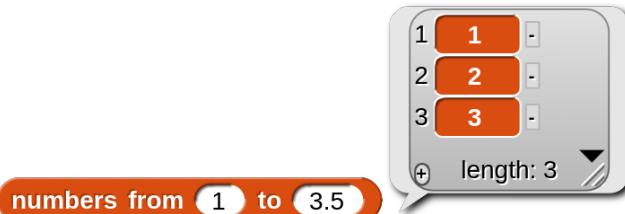
指定された範囲内で 1 ずつ増加するリストをリポート



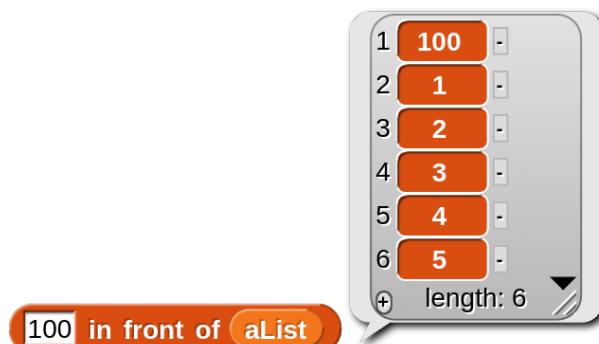
指定された範囲内で 1 ずつ減少するリストをリポート



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。

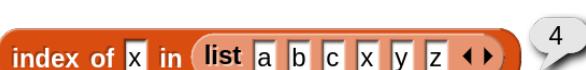
#### 4.5.3 all but first of ( )



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。Lisp 系言語の cdr にあたり、再帰処理で重要な働きをします。

#### 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 append ( ) ( )

これは、指定されたリストの要素を繋げたリストをリポートします。

The image shows a Scratch script and its resulting data visualization. The script consists of two main parts: an 'append' block followed by a 'list' block containing the numbers 1, 2, 3, 10, and 11. This is followed by another 'list' block containing 10 and 11, and finally a 'length' block. To the right, there are two data visualizations. The top one shows a list with items 1, 2, 3, 10, and 11, with a note 'length: 5'. The bottom one shows a 2D grid with columns labeled A and B. Column A contains items 1, 2, 3, 10, 20, and 12. Column B contains item 21. The grid has 6 rows and 2 columns.

#### 4.5.6 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。

The image shows two Scratch scripts side-by-side. The left script uses a 'for each' loop with 'item' as the variable, 'in aList' as the range, and 'say item for 2 secs' as the action. The right script uses a standard 'for' loop with 'i' as the variable, '1 to length of aList' as the range, and 'say item i of aList for 2 secs' as the action. Both scripts have a single green flag icon at the start.

#### 4.6 reshape ( ) to ( ) ( )

リストの要素にはリストを含ませることができますので、表のような形にすることができます。このように縦横二次元的なものを配列と言います。

The image shows a Scratch script and its resulting data visualization. The script consists of four 'list' blocks: one with 1, 2, 3, another with 4, 5, 6, a third with 7, 8, 9, and a fourth with 10, 11, 12. To the right, there is a data visualization of a 4x3 grid. The columns are labeled A, B, and C. Row 1 contains 1, 2, 3. Row 2 contains 4, 5, 6. Row 3 contains 7, 8, 9. Row 4 contains 10, 11, 12. The grid has 4 rows and 3 columns.

`reshape` ブロックを使って、一列の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部分は無視されます。

	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。

	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。

```

script variables [a]
set [a] to [reshape [list 0 < > to 4 3 < >]]
replace item [3] of [item [1] of [a]] with [99]
show variable [a]
say [item [3] of [item [1] of [a]]] for [2] secs
hide variable [a]

```

#### 4.6.1 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演算を行ったリストをリポートします。

「+」の左側の入力スロットが空になっています。ここに aList 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。空の場所は **+ 100** でもかまいません。

緑の演算子ブロックの外側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ーが出てきます。パラメーターとは、値を受け取るための変数のようなものです。map のフォーマルパラメーターには機能が設定されています。

1番目のフォーマルパラメーターは value で、指定されたリストの要素が順番にセットされます。これを使うと上と同じことができます。

リストの要素の値を複数箇所で参照する処理の場合は value を使う必要があります。

2番目のフォーマルパラメーターは index 、3番目が list です。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。この場合 list は aList を表します。ただし、list は aList 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 aList の値もそのように変更されます。value = item (index) of (list) の関係になっています。

これを使うと逆順のリストが作れたりします。

文字列に対する逆順操作です。

dlrow olleh

join は のように、左右の三角のところにリストをドロップするとリストからの入力とすることができます。

join input list: list a b

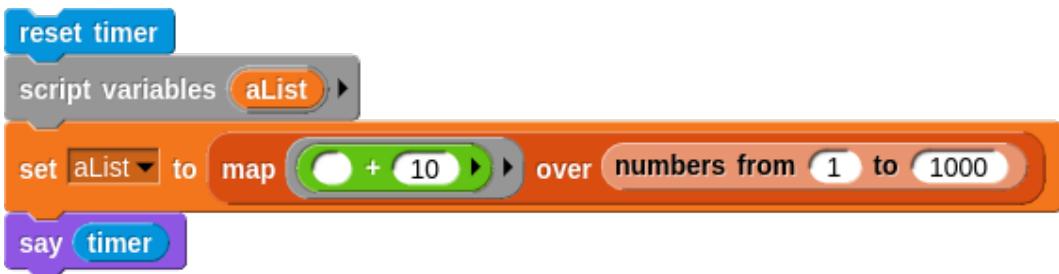
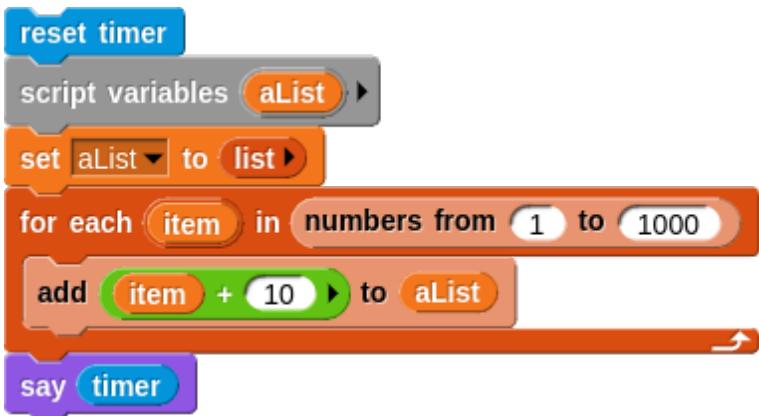
ab

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

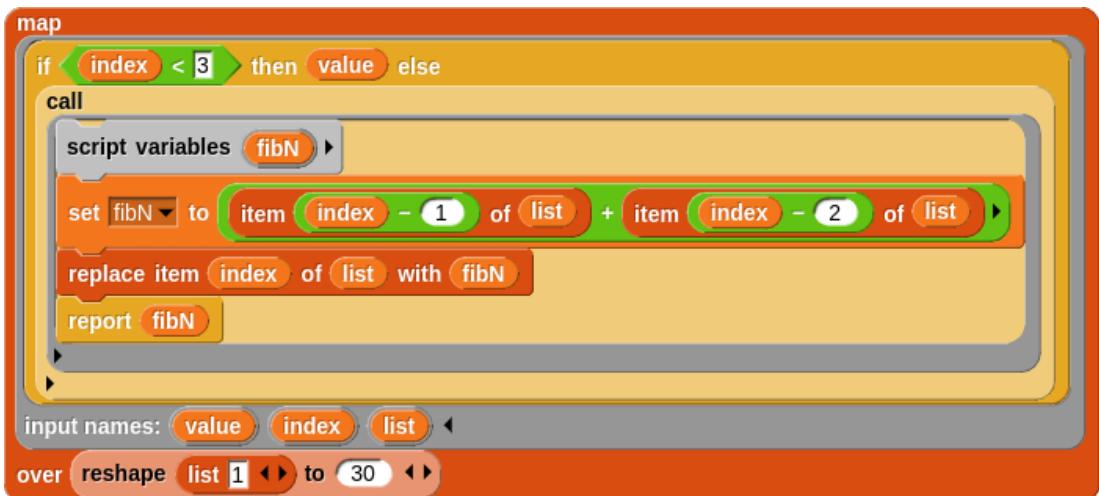
こんなふうにすると、次の要素とのペアのリストができます。  
 (リポーターブロックの if then else については、33 ページ参照)



フォーマルパラメーターの使用例を示すためにひねり出したもので、あまり意味はありません。複雑になって分かりづらくなるようならば、素直に for ループなどで作ったほうがいいでしょう。ただ、次に示すように繰り返し処理を行うには map ブロックを使うほうが速くできます。私の環境では、for each 版が 16.6 秒、map 版が 0 秒でした。



これは指定したリストを変更するといいいけないことをするのでおすすめできないやり方ですが、フィボナッチ数列（122 ページ参照）を map ブロックで作ることができます。



リストの値を使用しないで、必要な要素のリストを作成するためだけの使い方もあります。



( 146 ページに使用例あり )

#### 4.6.2 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。

2000 年から 2100 年までのうるう年のリストを求めてみます。



(119 ページに使用例あり)

#### 4.6.3 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処理ですが、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。

#### 4.6.4 combine ( ) using ( )

**combine aList using** [ + ] 15 は、指定されたリストの要素に対して指定された合算のような演算を行った結果をリポートします。指定されたリスト自体は変更しません。この場合は、[ 1 + 2 + 3 + 4 + 5 ] を行うのと同じ内容になります。combine では、[ + ] [ × ] [ join ] [ and ] [ or ] の演算がおもに使われるようです。

**combine aList using** [ × ] は、[ 1 × 2 × 3 × 4 × 5 ] つまり、5 の階乗を求める操作になります。

$n$  個のものから  $r$  個取り出して並べる順列の総数は、 ${}_n P_r = n(n-1)(n-2)\cdots(n-r+1)$  になります。 $(0 < r \leq n)$

set [n] to [4]  
set [r] to [3]

4 個のものから 3 個取り出して並べる順列の数はとして、

**combine numbers from [n] to [n - r + 1] using** [ × ] 24

また、 $n$  個のものから  $r$  個取り出して並べる組み合わせの数は、 ${}_n C_r = \frac{{}_n P_r}{r!}$  で求められます。4 個のものから 3 個取り出して並べる組み合わせの数は、

**combine numbers from [n] to [n - r + 1] using** [ × ] /  
**combine numbers from [r] to [1] using** [ × ]

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。)

**combine aList using** [ join ] 12345

**combine aList using** [ join ] 1:2:3:4:5

これと逆の操作をするのが split ブロックです。

`split combine aList using join [ ] by letter`

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

`split combine aList using join [ ] [ ] by [ ]`

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

and と or を使った例です。

`combine list [true true true] using [true] and`

`combine list [true false true] using [false] and`

`combine list [true true true] using [true] or`

`combine list [true false true] using [true] or`

#### 4.6.5 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。

`combinations list [A B] list [1 2]`

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

## 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。)

The image shows four examples of list operations in Snap!:

- Example 1:** A green script block "numbers from 1 to 3 + 10 ▶" is followed by a gray data block containing three orange boxes labeled 11, 12, and 13. Below the data block is the text "+ length: 3".
- Example 2:** A green script block "numbers from 1 to 3 × 10 ▶" is followed by a gray data block containing three orange boxes labeled 10, 20, and 30. Below the data block is the text "+ length: 3".
- Example 3:** A green script block "numbers from 1 to 3 × 0 ▶" is followed by a gray data block containing three orange boxes labeled 0, 0, and 0. Below the data block is the text "+ length: 3".
- Example 4:** A green script block "numbers from 1 to 3 × 0 ▶ + 1 ▶" is followed by a gray data block containing three orange boxes labeled 1, 1, and 1. Below the data block is the text "+ length: 3".

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。

The image shows two examples of element-wise list operations in Snap!:

- Example 1:** A green script block "list [1 2 3] ▶ + list [1 2 3] ▶" is followed by a gray data block containing three orange boxes labeled 2, 4, and 6. Below the data block is the text "+ length: 3".
- Example 2:** A green script block "list [1 2 3] ▶ × list [1 2 3] ▶" is followed by a gray data block containing three orange boxes labeled 1, 4, and 9. Below the data block is the text "+ length: 3".



リストに文字列の各要素を入れることもできます。

letter [1 of world] w

letter [numbers from 1 to length of world] of world

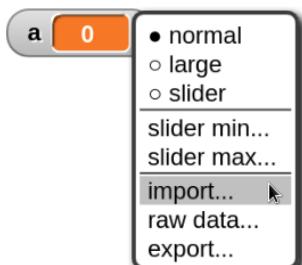


letter [numbers from length of world to 1 of world]



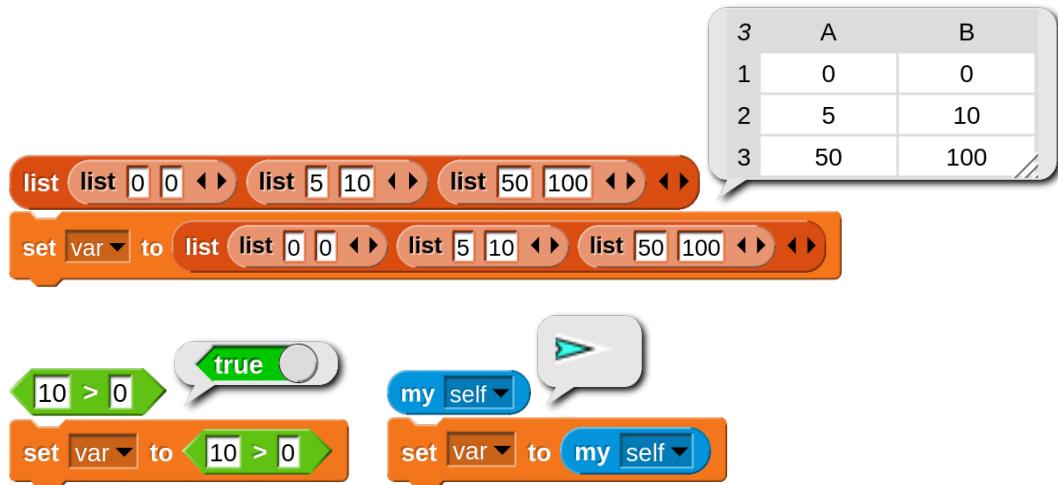
## 4.8 変数に入れられるもの

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。

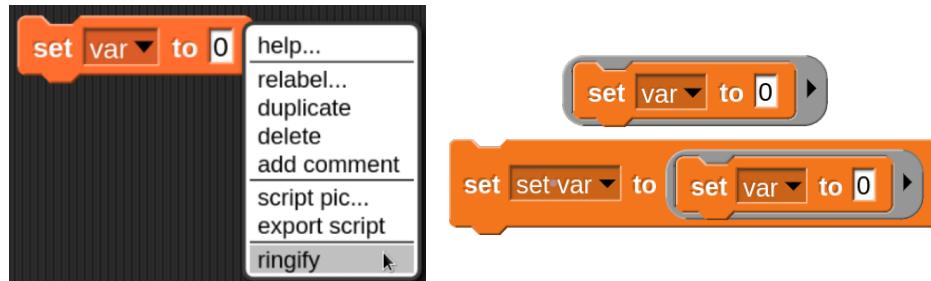


「import...」で、指定したファイルの内容を取り込むことができます。  
.csv や .json のファイルの場合は、そのデータ形式に従ってリストを作成します。  
.csv ファイルではカンマと改行をセパレーターとした要素をリストにします。単なるテキストデータとして取り込みたい場合は、「raw data...」を使います。

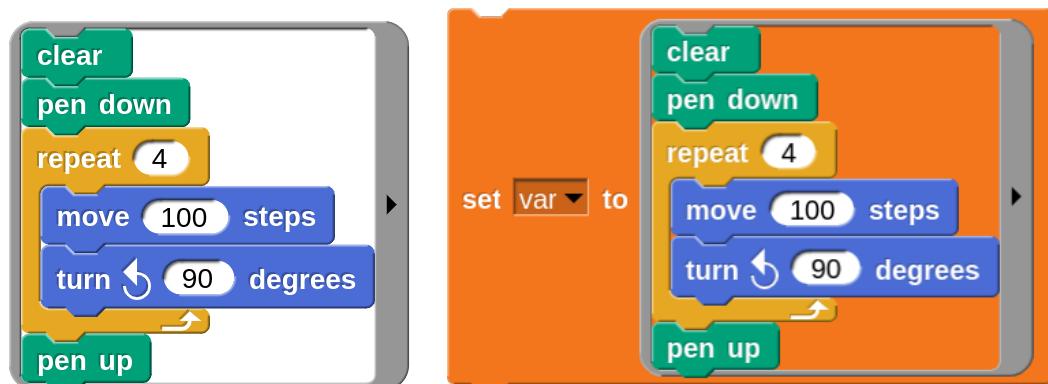
Snap! の変数には、値をリポートするものは入れられるようです。



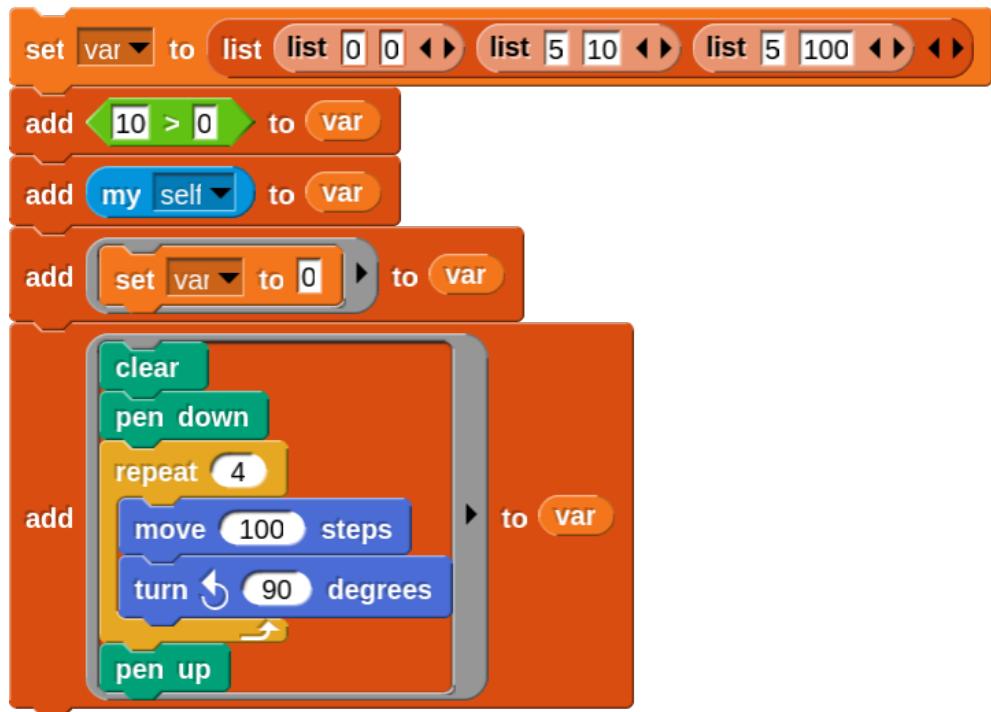
ブロックをリングに入れてやるとリポーターブロックにすることができます。ブロックそのものをリポートするものです。対象のブロックのところで右クリックしてメニューから `ringify` を選べばリングで囲むことができます。リングを外すには `unringify` をクリックします。



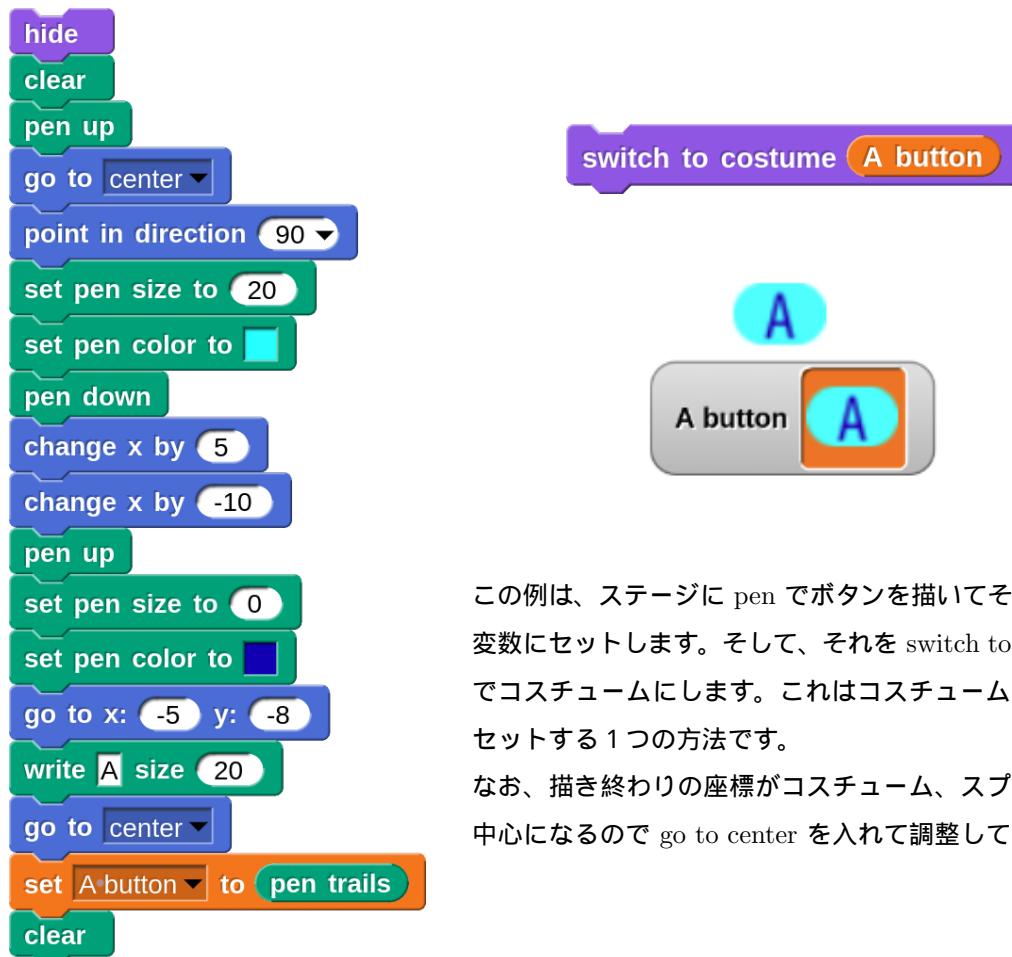
1つのブロックだけではなく、スクリプトもまとめてリングで囲むことができます。それを変数に入れることができます。



リストに入れることもできます。



入れることができることを示すためのものでスクリプトとしての意味はありません。



この例は、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

```
set n1 ▾ to pick random 1 to 100  
set n2 ▾ to pick random 1 to 100
```

二つの数のうち大きいほうを max という変数にセットするスクリプトは、if else 制御ブロックでも、if then else リポーター ブロックでも作ることができます。

```
if n1 > n2  
  set max ▾ to n1  
else  
  set max ▾ to n2
```

```
set max ▾ to if n1 > n2 then n1 else n2
```

n1 のほうが大きかったら n1 をリポート、そうでなかったら n2 をリポートするので、それが max にセットされます。

### 5.2 when ( )

```
when green flag clicked
```

ずっとチェックを続け、指定された条件 ブロックが True 真になると実行します。

```
when touching [ ] ?
```

```
when touching [Sprite(2)] ?
```

```
when green flag clicked  
forever  
  go to random position  
  wait 1 secs
```

条件を True 真にする  
と、forever ループを使  
うようなスクリプトを  
when ブロックで行うこ  
とができます。

```
when green flag checked  
  go to random position  
  wait 1 secs
```

when ブロック版の場合は、 をクリックすると のように形が四角に  
なり、それをクリックして再開することができます。

when ブロックを使って、人が歩く動作をしてみます。



```

when green flag clicked
set size to 30 %
set [my rotation style v] to 2
set [歩行中 v] to [false]
switch to costume [avery walking b]
wait [3] secs
set [歩行中 v] to [true]
wait [20] secs
set [歩行中 v] to [false]

```

```

when [歩行中 v]
wait [0.2] secs
next costume
move [10] steps
if on edge, bounce

```

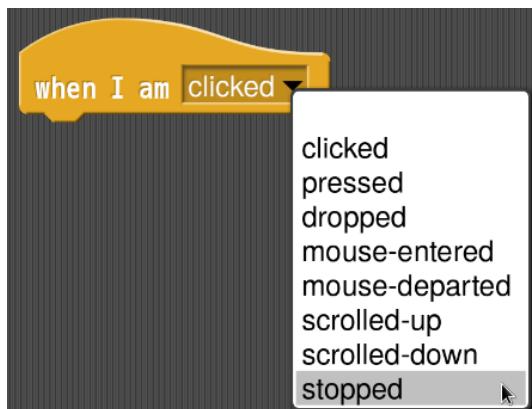
```

when [not 歩行中 v]
if [costume # mod 2 > 0]
next costume

```

足を開いていたならば閉じる

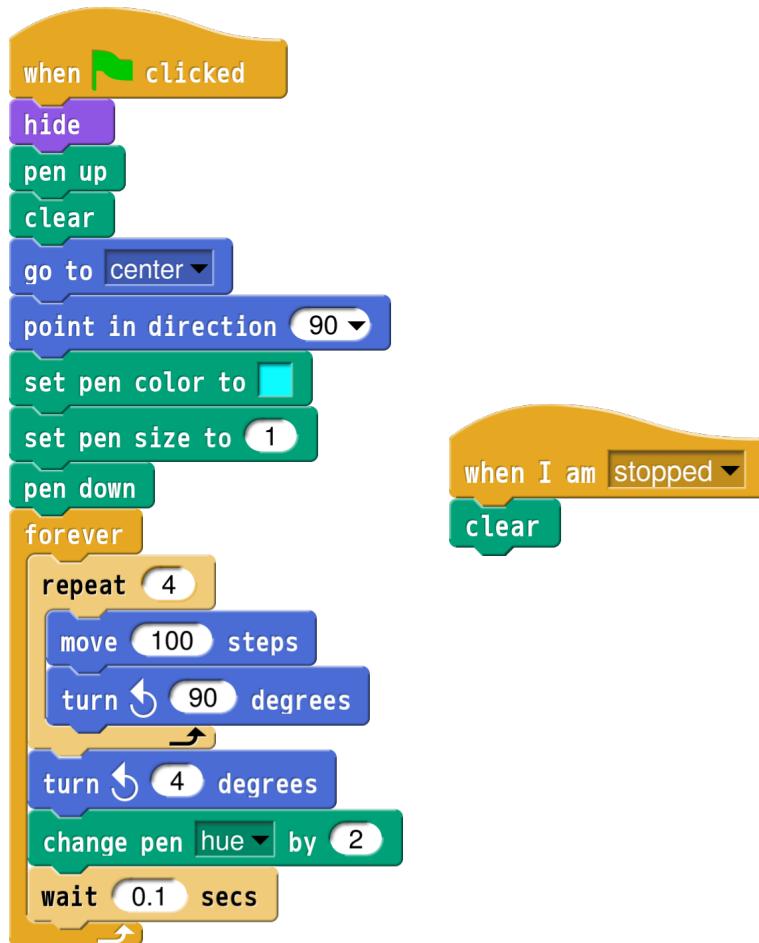
### 5.3 stop ボタンがクリックされた時の終了処理



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。

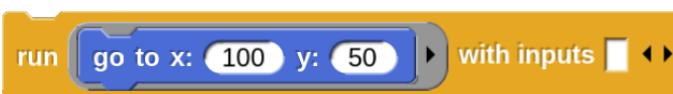
次のスクリプトで動作の確認ができます。時間的にも限られたことしかできないようです。状況によってはきれいに clear できない場合があります。



## 5.4 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。run func 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。

## 5.5 call ブロック

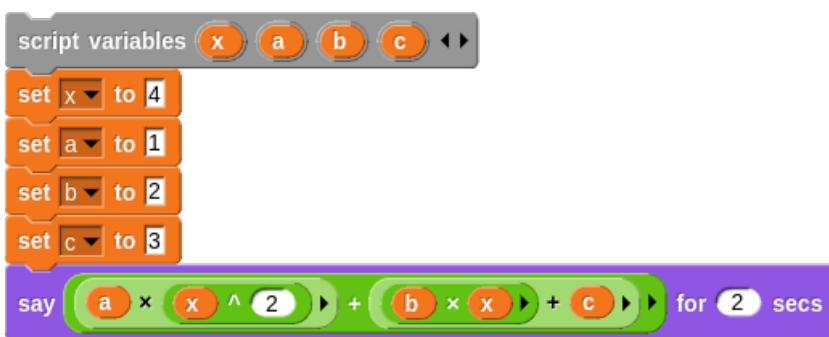
run ブロックに入れられるのが値をリポートしない実行 (Command) ブロックだったのに対して、call ブロックに入れられるのは、実行または評価することによって何らかの値または true か false をリポートするブロックです。call ブロックは、得られた値をリポートします。



call ブロックを使ってちょっとした関数のようなものが作れます。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ところになります。



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ーを出します。



フォーマルパラメーターは、クリックして変数名を `x`、`a`、`b`、`c` 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。名前がないので使用する場所ごとにこのブロック自体を使っての使用になります。このやり方はまさに Scheme の `lambda` 式そのものです。



このリングを変数に入れてやれば一度きりではなく定義ブロックのようにも使えます。



次の例は入力スロットで使用する演算ブロックを指定できることを示すためのもので、あまり意味はありません。その演算ブロックはリングで囲う必要があります。

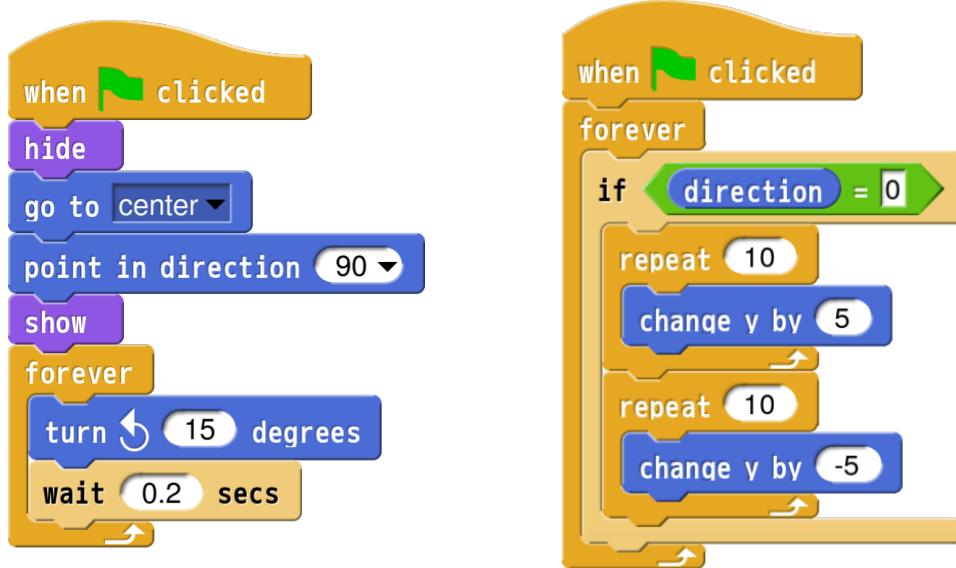


## 5.6 launch ブロック

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。（102 ページ参照）

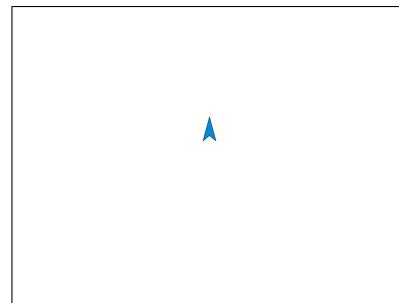
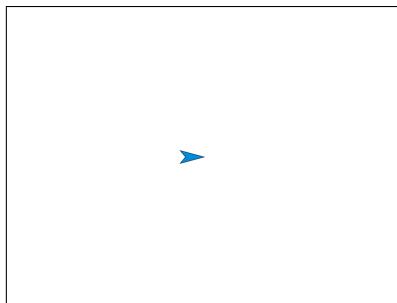
1 個のスプライトに対して、回転させるスクリプトと、角度が上（0 度）の時にジャンプさせるスクリプトを並列で実行してみます。

並列処理は、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。

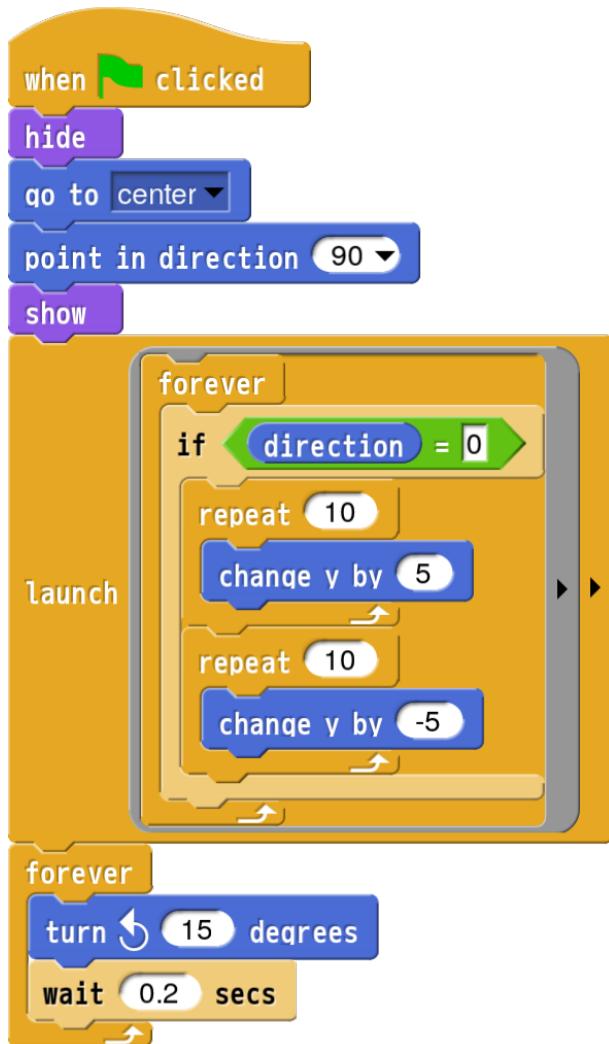


左のスクリプトで、ずっと回転させる動作を行います。

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。



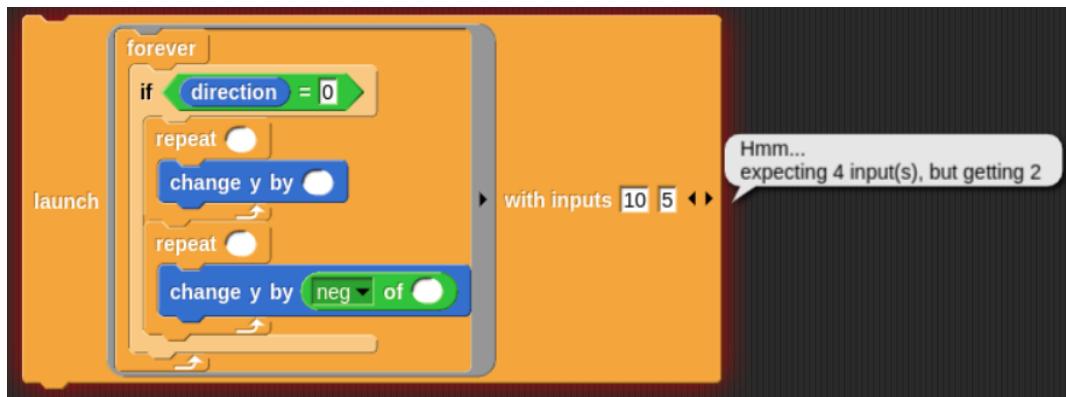
launch を使うことで 1 つのスクリプトで実行することができます。



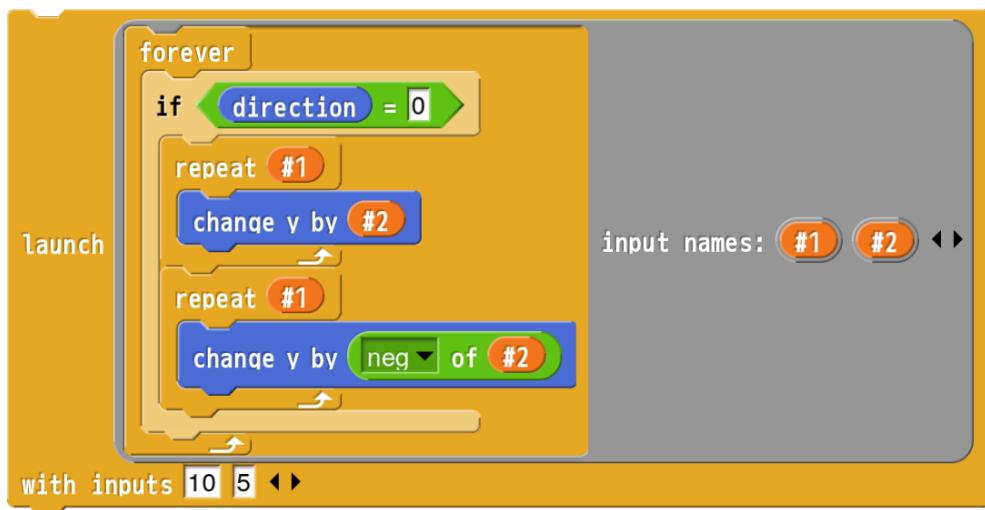
launch の右端の右向き三角をクリックすると、入力値を設定することができます。y の値の設定を空（空白ではなく）にして、入力値をひとつだけ設定すると、その入力値がすべての空の部分の値になります。



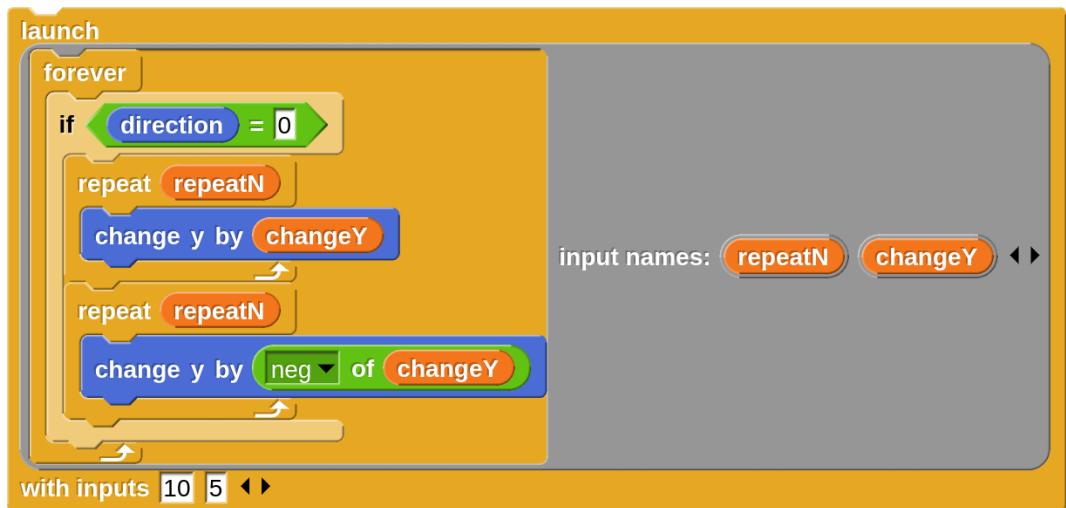
2 個以上の入力値を設定した場合、空の入力スロットの個数が合わないとエラーになります。



with inputs [10] [5] [10] [5] とすればいいのですが、リングではフォーマルパラメーターが使えるので、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。

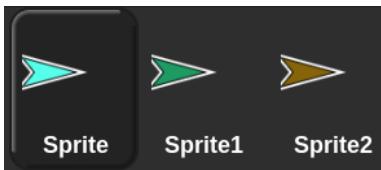


フォーマルパラメーターをクリックして変数名を変更するとスクリプトが分かりやすくなります。



この with inputs とフォーマルパラメーターの利用法は launch だけでなく、他の with inputs を持つブロックで使えます。

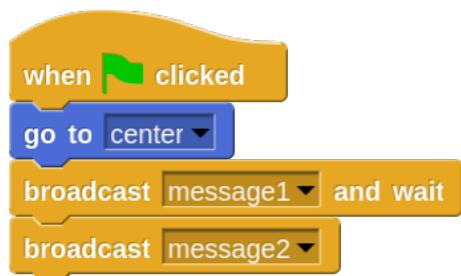
## 5.7 broadcast ブロック, tell to ブロック



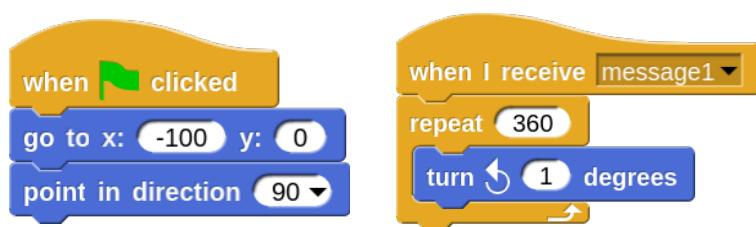
次はスプライトを3個使います。Sprite, Sprite1, Sprite2 を用意してください。

2つ目と3つ目の名前をそれぞれ Sprite1, Sprite2 にしてください。Sprite から命令を出して Sprite1 と Sprite2 を順番に動かします。broadcast を使うと次のようになります。

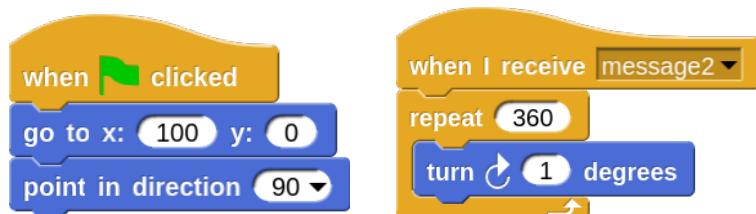
Sprite 用



Sprite1 用

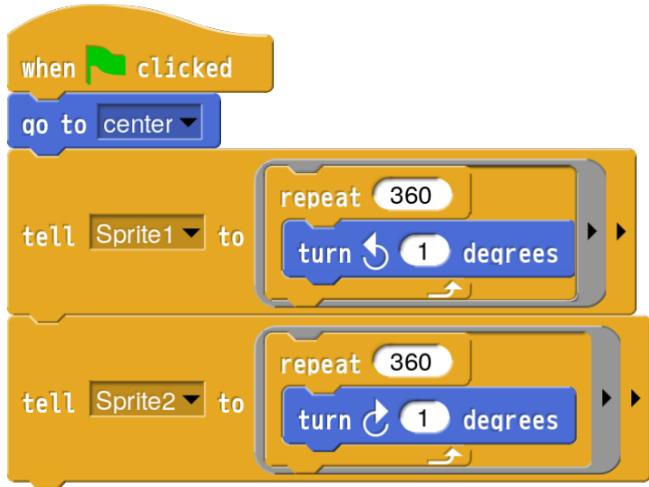


Sprite2 用



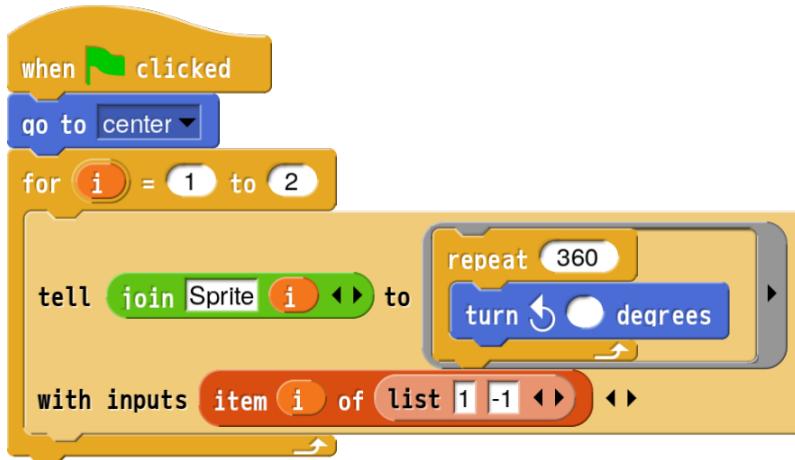
**tell [ ] to [ ]** のブロックを使用すると broadcast を使わなくても Sprite から Sprite1, Sprite2 を直接操作することができます。

Sprite 用



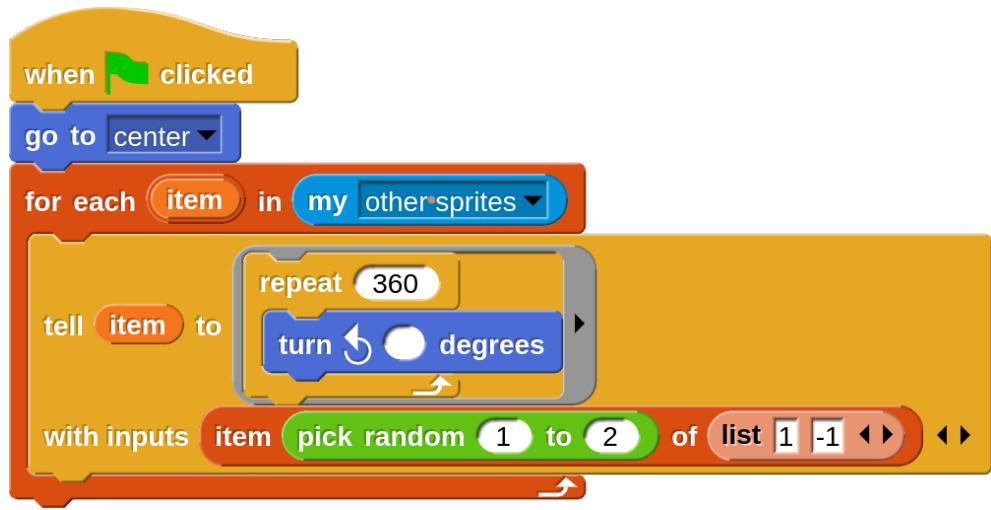
もしも Sprite1 と Sprite2 を同時に動かすのならば、tell Sprit1 のブロックを launch で囲みます。

tell の宛先は文字列も使えるみたいなので、こんなふうにすることもできます。



with inputs で指定する値は i が 1 の時は 1 で、2 の時は -1 になります。これが turn の入力スロットに入り、左回りか右回りに 1 度回転ということになります。

自分以外の全部のスプライトを操作するなら、こんなふうにすることもできます。tell の宛先には item つまり、my other sprites が順に入れります。with inputs で指定する値は乱数により 1 か -1 になります。これが turn の入力スロットに入り、左回りか右回りに 1 度回転ということになります。



## 6 プロックを作成する

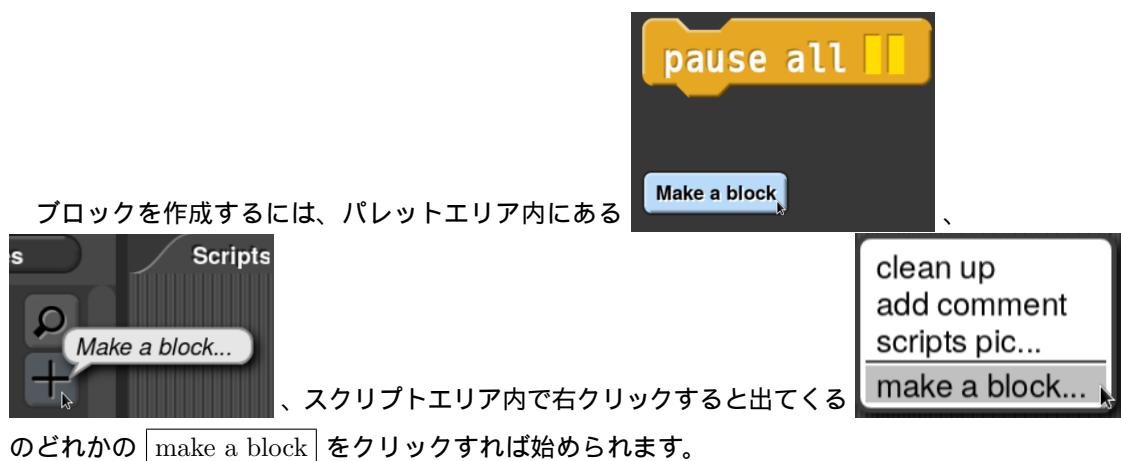
Snap! では、ローカル変数が使え、値をリポートすることもできるので、カスタムブロック（ユーザー定義ブロック）が作りやすくなりました。

### 6.1 $\geq$ ブロック

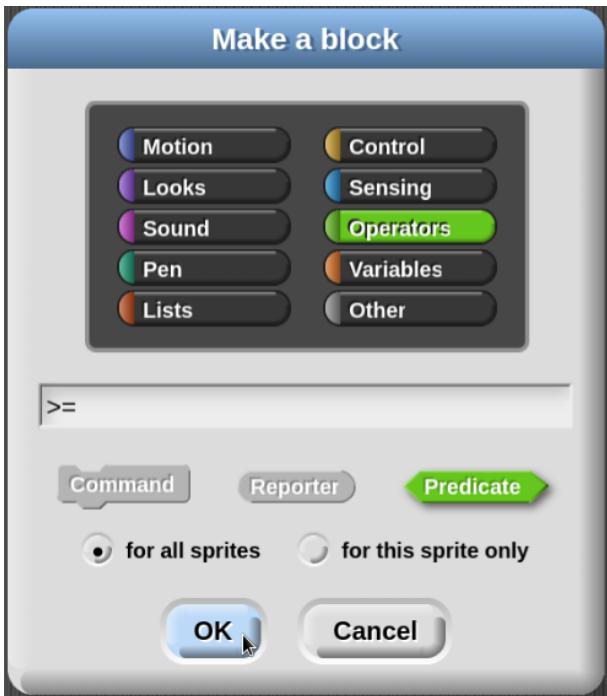
最初の例として、 $\geq$  のブロックを作ってみます。

これがあると、`var > 0 or var = 0` ではなく、`var  $\geq$  0` とできるのですっきりします。

ブロック内部では `var > 0 or var = 0` を行っているので見た目だけのことですが。



Motion のパレットエリアにある作成用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、なんらかの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。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  $>=$  を入れて、上の欄でボタン Operators を、下の段で Predicate 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事であり、置き場所を決めることでもあります。プリミティブの「 $>$ 」ブロックが Operators に置いてあるのでそこにします。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。



定義の先頭の  $+>=+$  の部分をプロトタイプといいます。ここを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。

「 $>=$ 」の左側の + ボタンをクリックしてください。



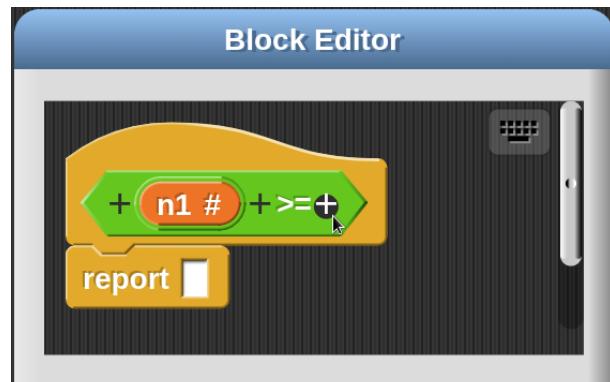
すると、入力ウインドウが出ます。左側に Title text, 右側に Input name のボタンがあります。ブロックに表示される文字列を指定する時には Title text をクリックして文字列を設定します。ブロックを使用する時にデータを受け取るための変数を指定する時には Input name で設定します。

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、

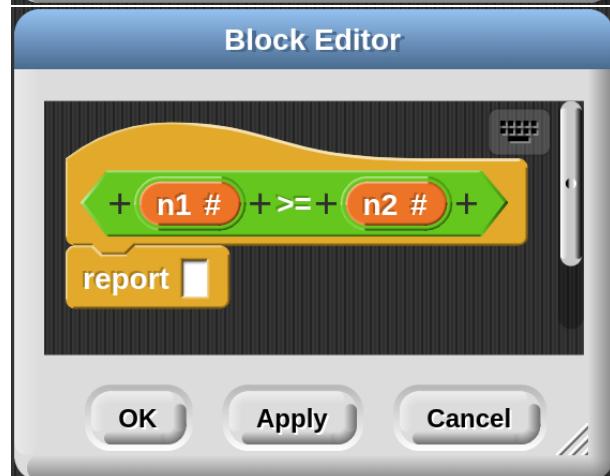


が出ます。現在は Any type が選択されています。 Any type つまり数値でも文字でも受け付けるタイプです。このままでいいのですが、あえて Number にしてみます。これは数値しか受け付けないタイプです。 Any type を選んだ場合は、変数の表記に「#」が付かないだけで以下の操作は同じです。

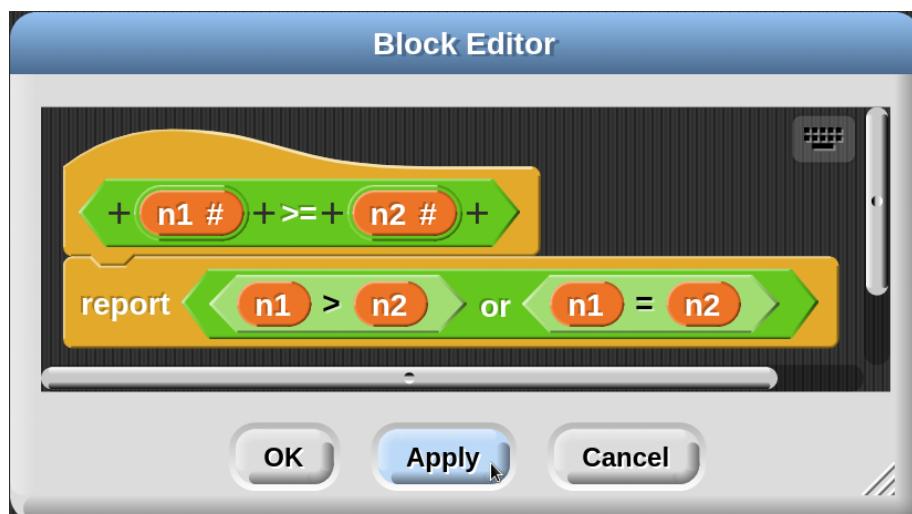
もう一つ、下の方に Single input. Default Value:  が出ます。ここで入力の初期値が設定できるのですが、この場合は関係ないのでこのままにしておきます。 OK をクリックして次に進んでください。



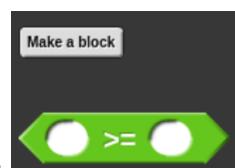
右側の + ボタンをクリックして、同じように n2 の設定をしてください。



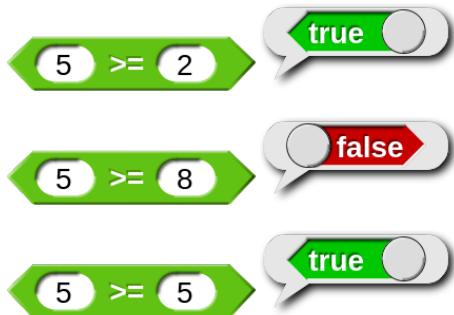
パレットエリアからブロックを持ってきて完成させてください。



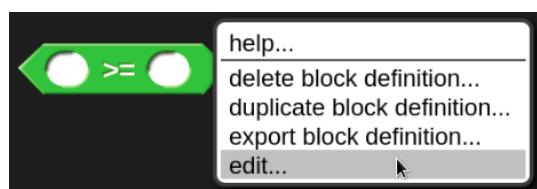
report は値をリポートする(返す)ためのブロックです。この場合は式の結果により真理値、true か false をリポートすることになります。apply をクリックしてください。



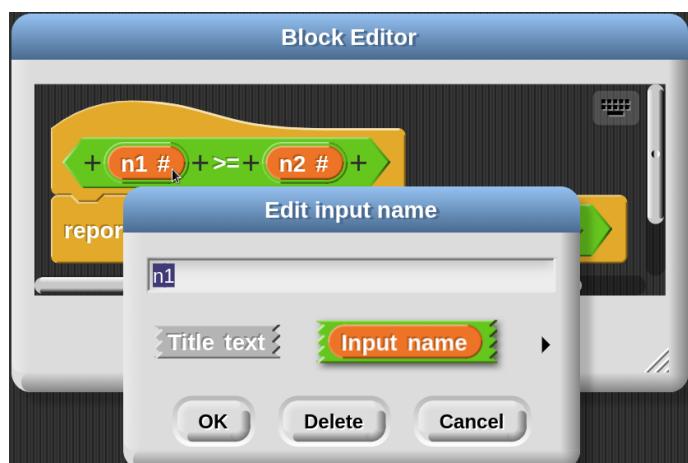
パレットエリアに  がセットされます。



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。



プロックを右クリックすると、edit... で内容の編集ができます。  
export block definition をクリックすると、このプロック定義だけをファイルに書き出すことができます。



から変更ができます。変数を Any type に設定し直すこともできます。

を使ってステップ実行する場合に、作成したプ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プ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。順序が逆だとそのプロック内のステップ実行はされません。

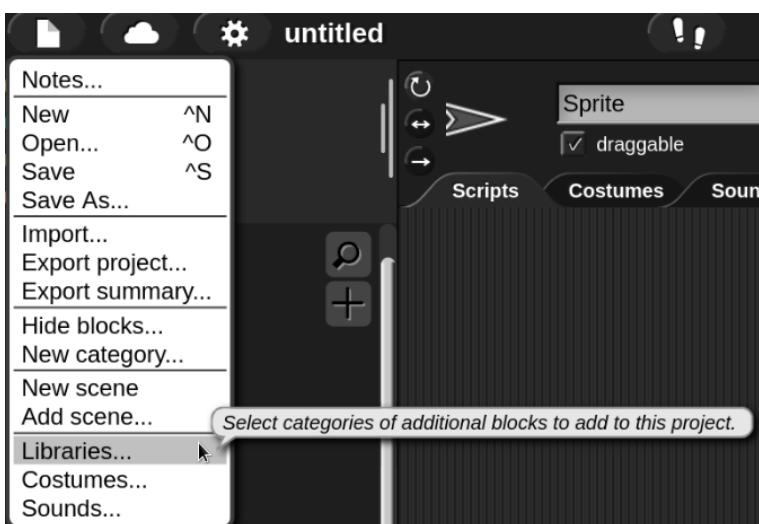
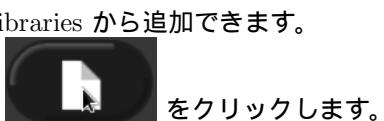
## 6.2 help 説明文の作成

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プロックの help で表示されます。普通のプロックの場合はプロックのところで右クリックして add comment をしますが、プロトタイプではプ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ドロップします。

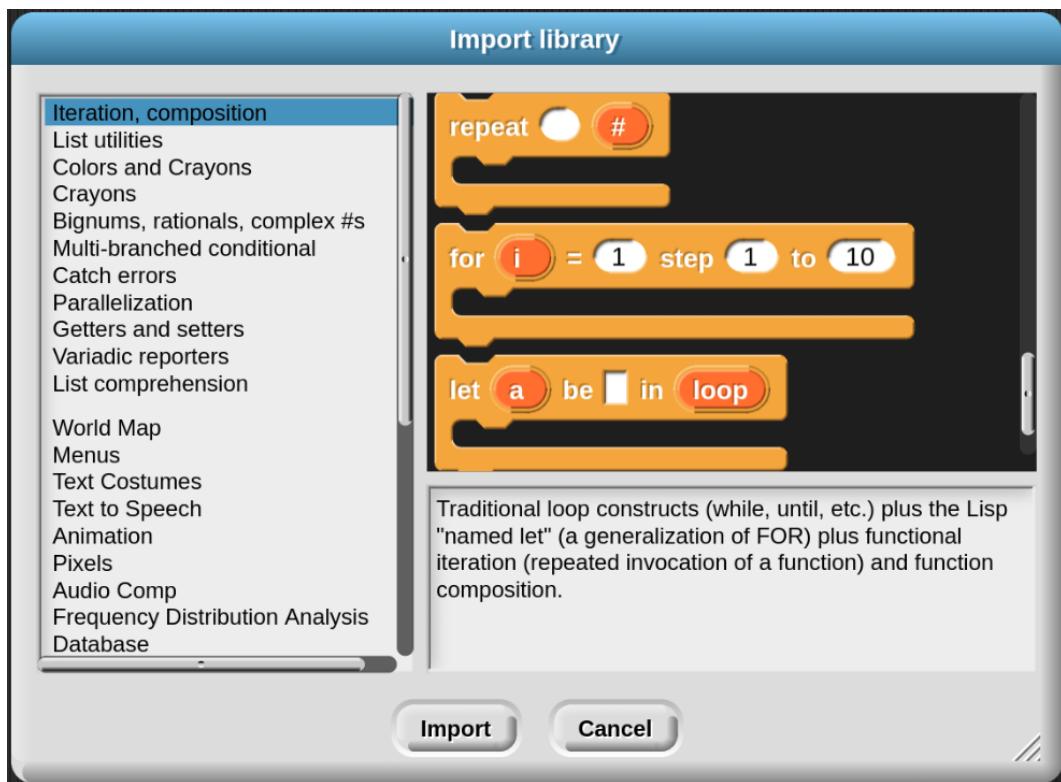


### 6.3 for i = start to end step add

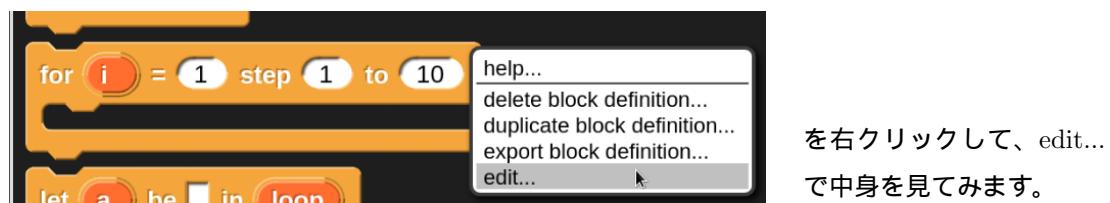
Snap! には ブロックがありますが、増減分が指定できるものも Libraries から追加できます。

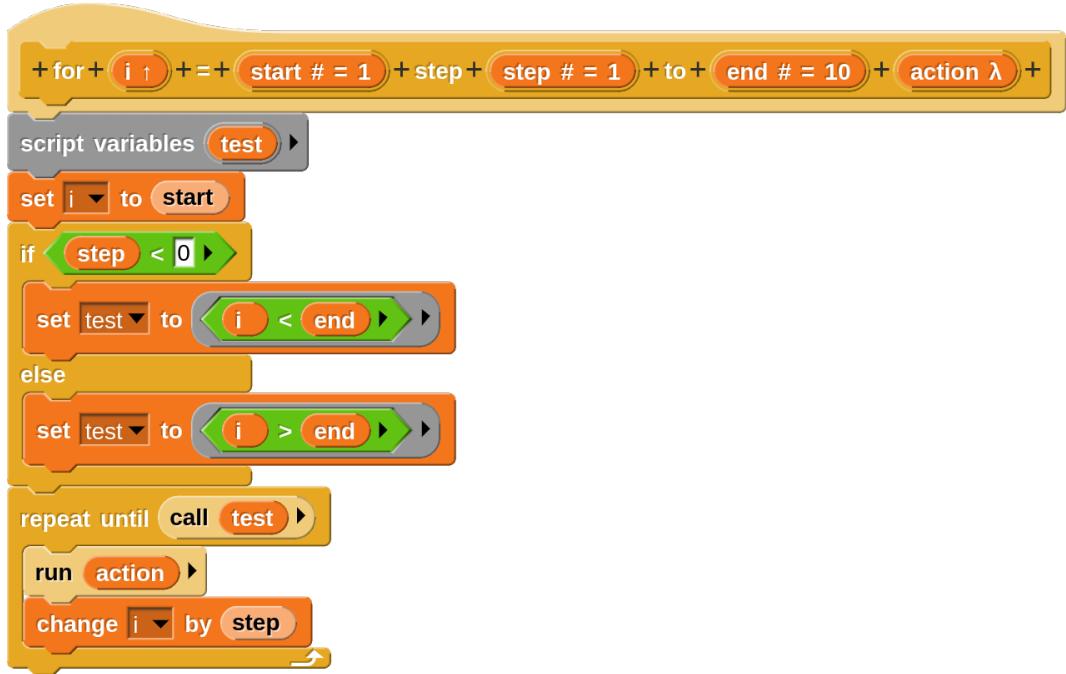


Libraries... をクリックします。Iteration, composition をクリックして内容を確かめてから Import すれば追加できます。



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。





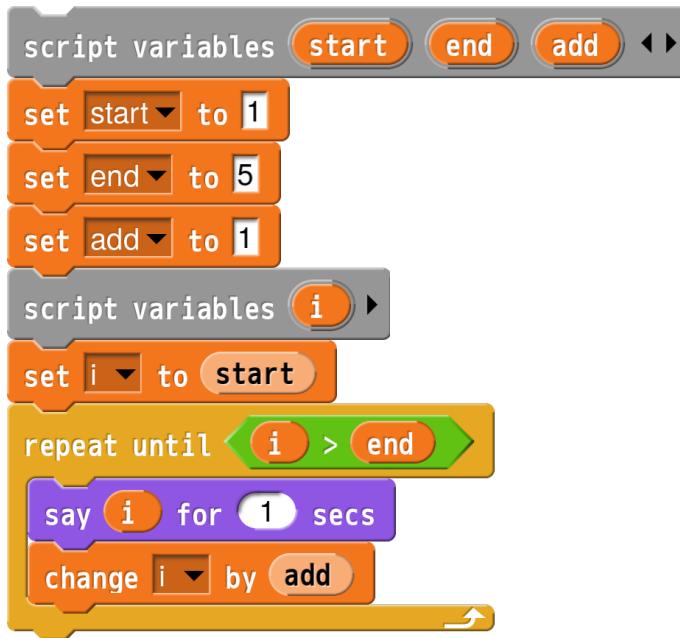
このスクリプトにしたがって、`my for i = start to end step add` という増加部分の位置が違うだけのプロックを作成してみます。

まずは、スクリプトエリアで for プ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, add, *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、add 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

```

script variables [start] [end] [add]
set start to 5
set end to 1
set add to -1
script variables [i]
set i to start
repeat until (i < end)
  say (i) for (1) secs
  change i by add
end

```

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。

```

set i to start
if (add > 0)
  repeat until (i > end)
    say (i) for (1) secs
    change i by add
  end
else
  repeat until (i < end)
    say (i) for (1) secs
    change i by add
  end
end

```

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。

```

set i to start
repeat until (if (add > 0) then (i > end) else (i < end))
  say (i) for (1) secs
  change i by add
end

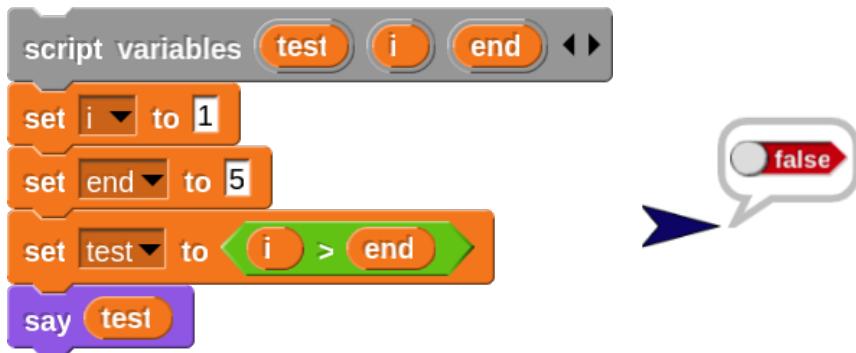
```

```

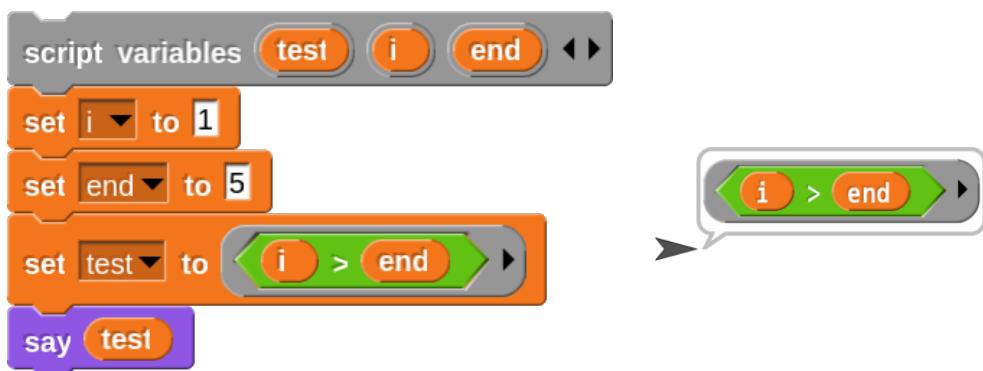
if [add > 0] then
  [i > end]
else
  [i < end]
end

```

の部分で、add 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。

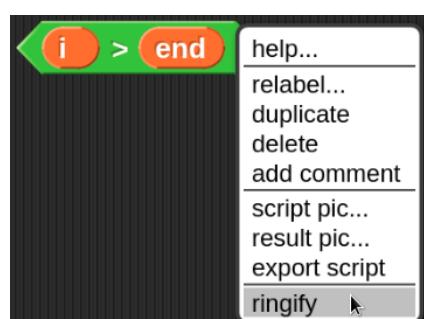


ループしている中で変化する i の値に対してテストする必要があります。そこで使用される機能がリングです。



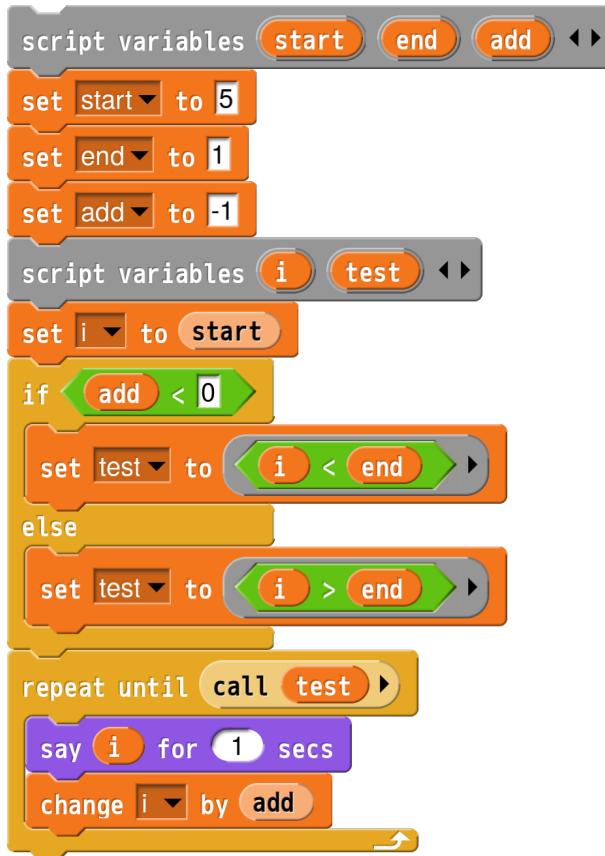
テスト項目自体を変数に入れます。

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。



`call [test v]` は形から分かるようにリポーターブロックで、`test` つまり、`[i > end v]` のブロックをテストした結果をリポートしてくれます。

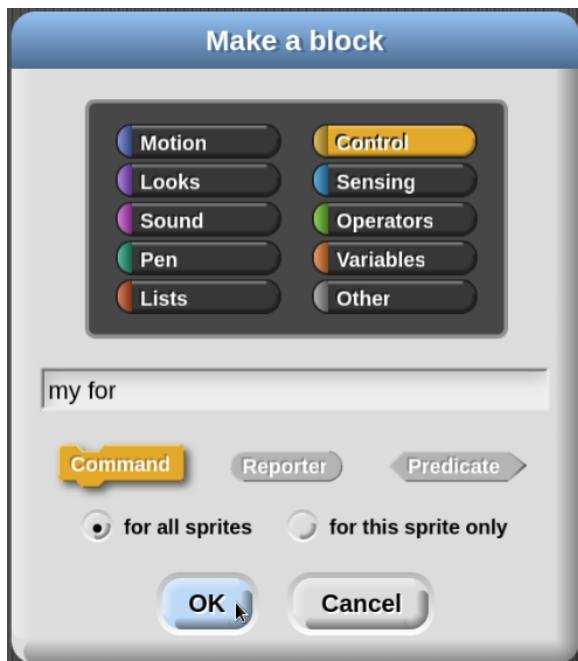
これを for ループに使用します。



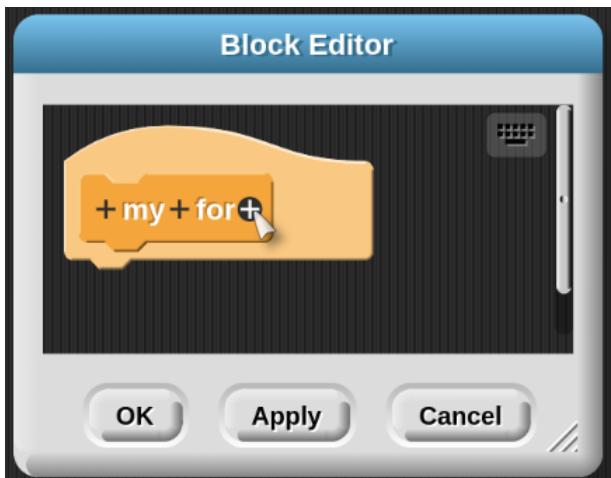
これが Libraries の for ループの内容です。

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ふのです。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。



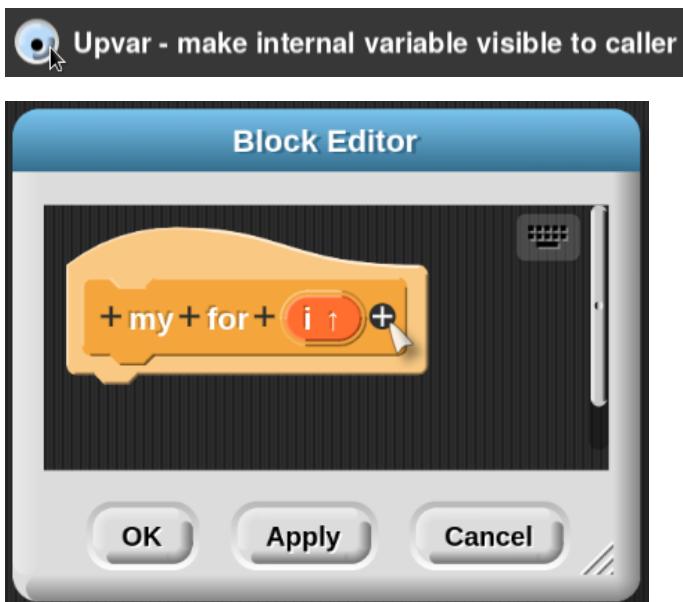
ブロックエディターを開いてから、  
Control, Command を選択して  
my for と入力して OK で次に進んで  
ください。



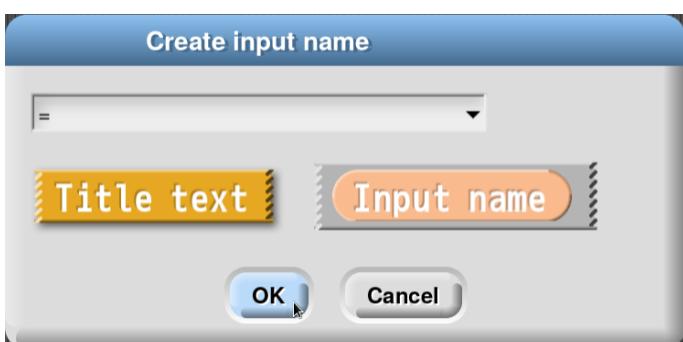
「+」をクリックして、変数 i の設定をします。

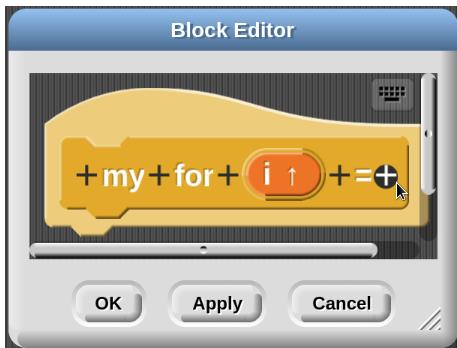


変数 i は、for ループの中にドラッグ&ドロップして使用できる特別な変数です。Upvar オプションを選択します。すると、自動的に Number や Any type のオプションはクリアされます。

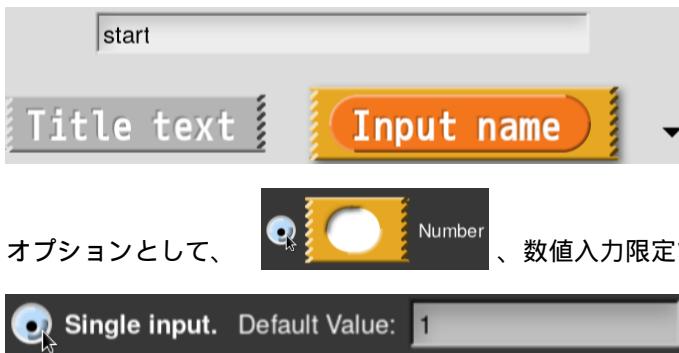


変数 i のところに Upvar を示す「↑」が表示されます。続いて、「+」をクリックして、「=」を、Title text として入力してください。



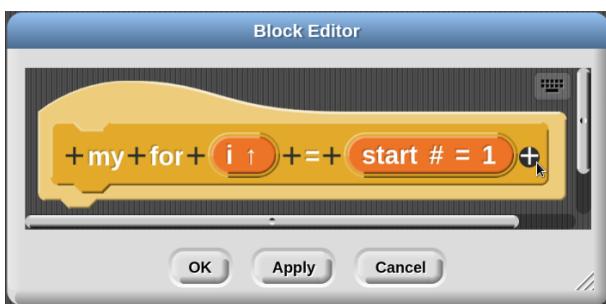


続いて、「+」をクリックして、変数 start を設定します。

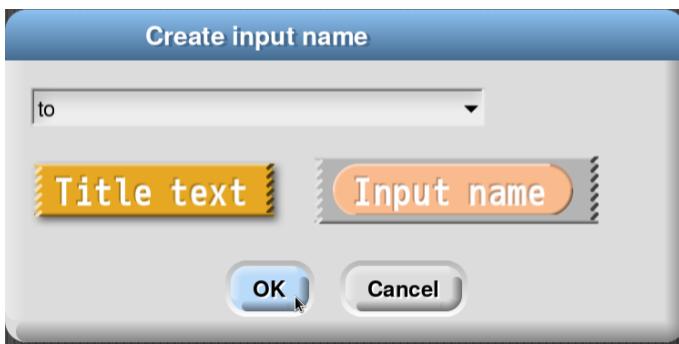


規定値を 1 に設定します。

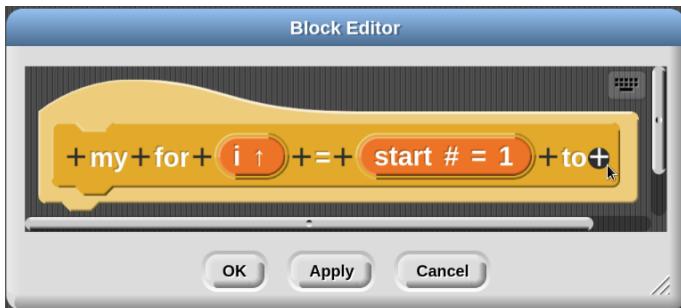
続いて、「+」をクリックして、



「to」を、Title text として入力してください。



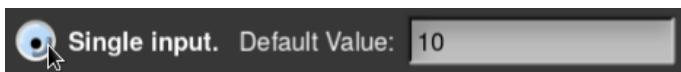
続いて、「+」をクリックして、



変数 end を設定します。

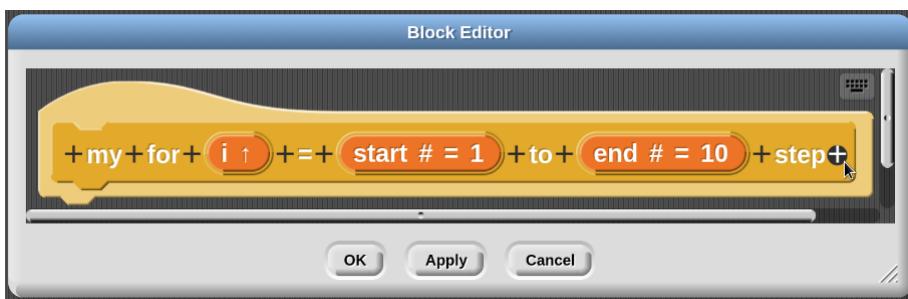


オプションとして、 Number 、数値入力限定で、

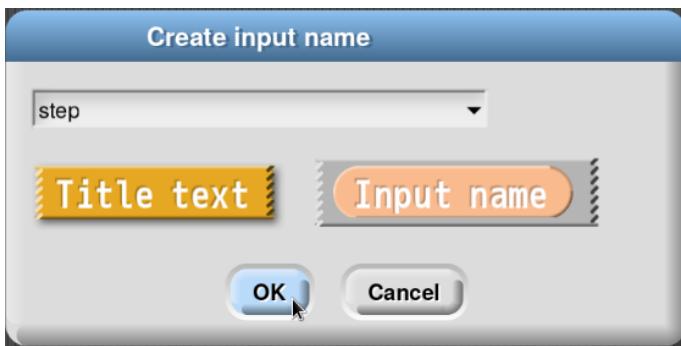


規定値を 10 に設定します。

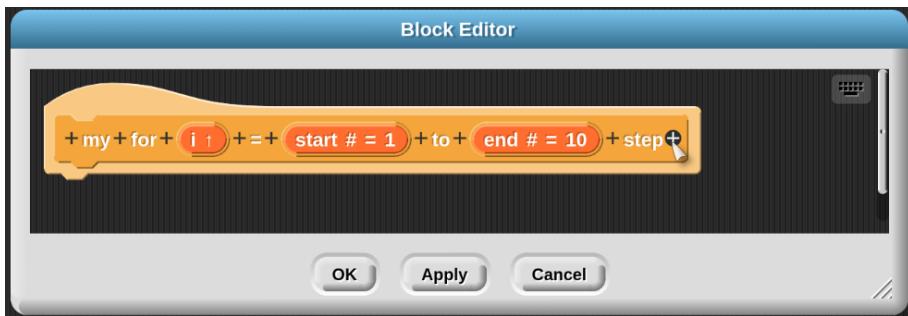
続いて、「+」をクリックして、



「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 add を設定します。

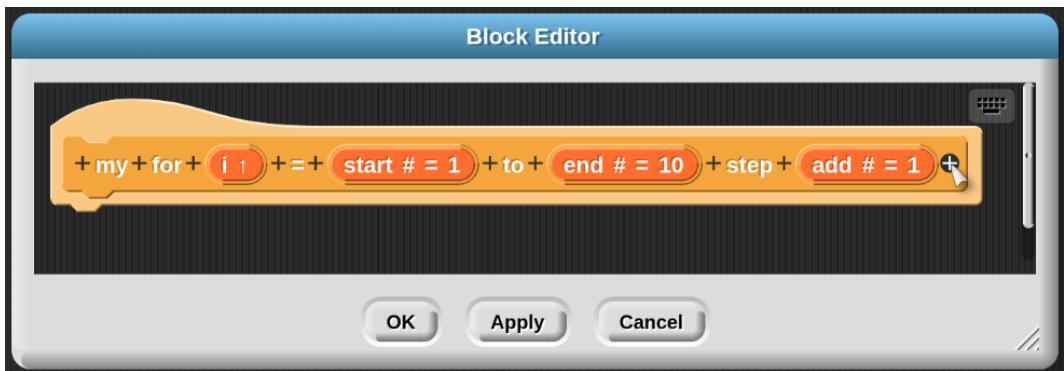


オプションとして、、数値入力限定で、



規定値を 1 に設定します。

続いて、「+」をクリックして、



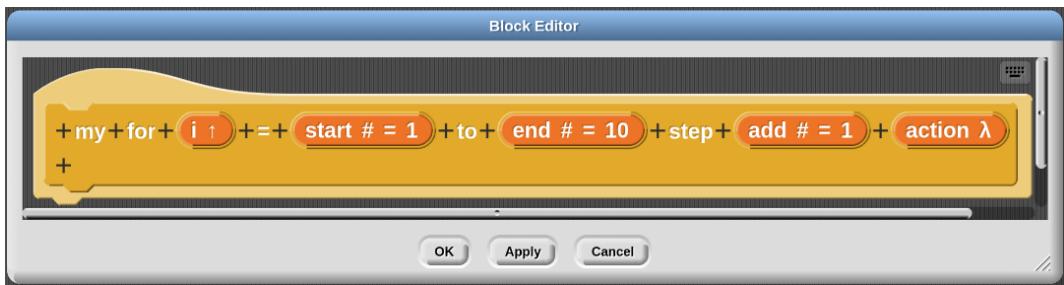
変数 action を設定します。



オプションとして、 を選択すると、自動的に

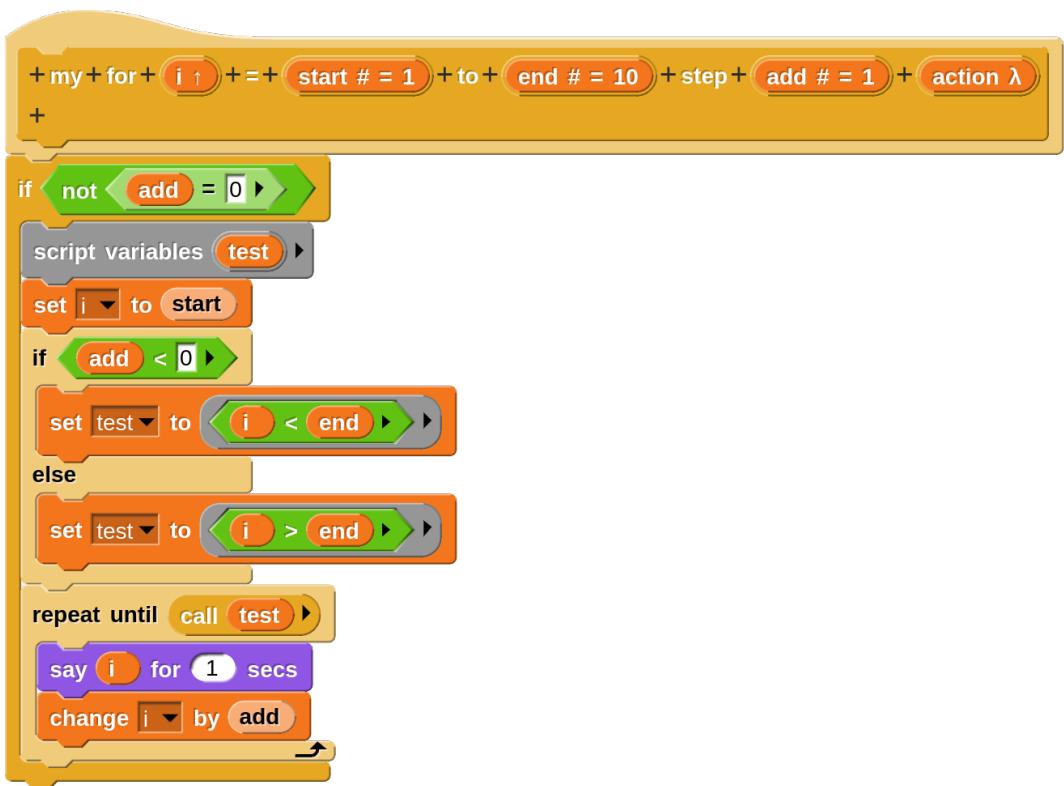


がセットされます。これによって for ループ内で実行するスクリプトを受け取ります。



変数 action に特別なマークが付きました。

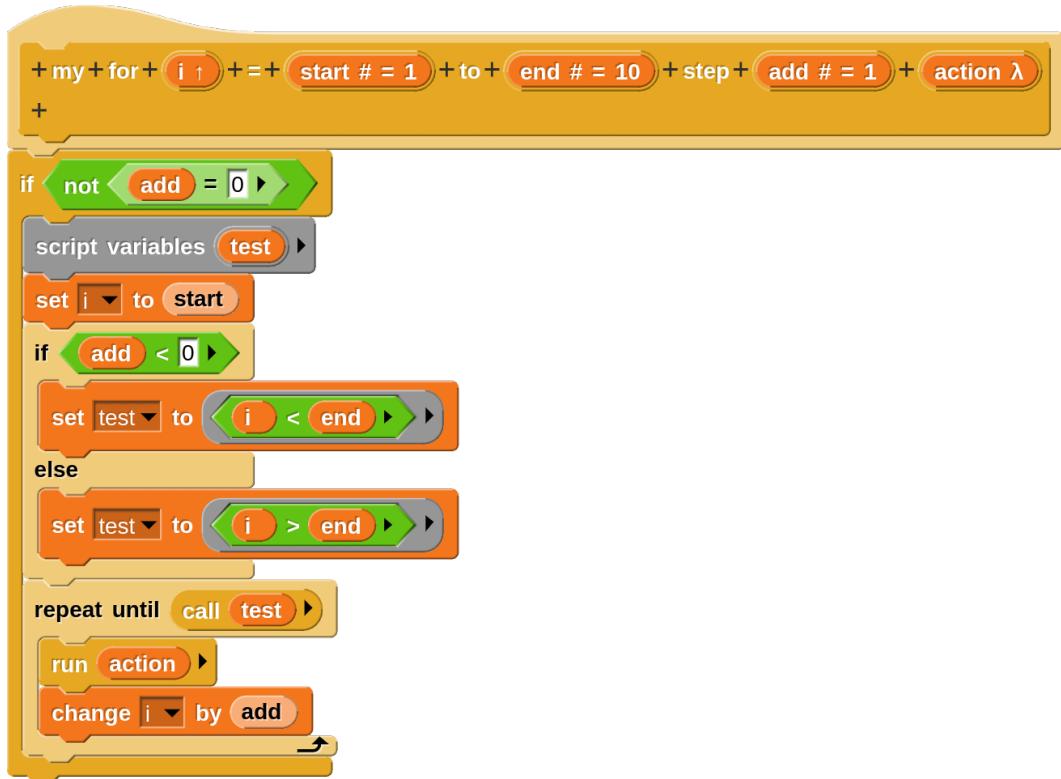
ブロック定義の本体として、実験していた for ループのスクリプトを持ってきます。



action で指定されたスクリプトを実行するブロックは run です。パレットエリアから持ってきて action をはめ込みます。



これで、実験用のスクリプトを入れ替えます。

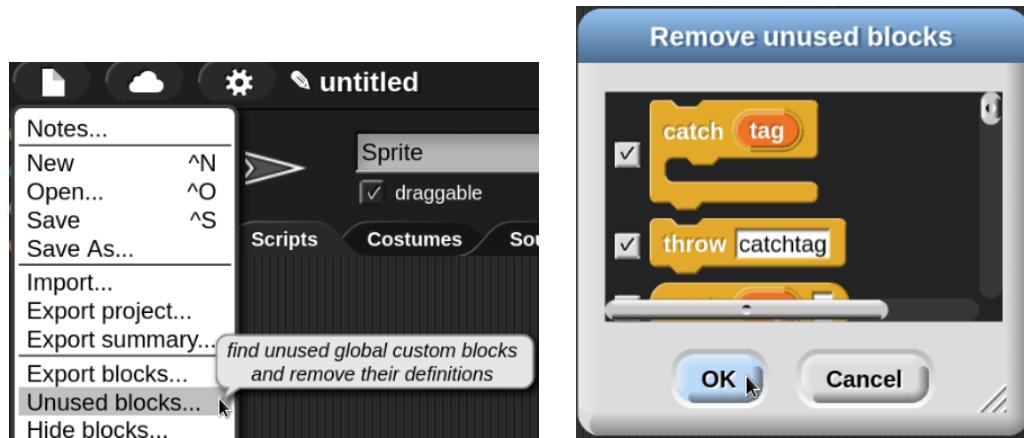


Apply するとパレットエリアに作成したブロックがセットされます。

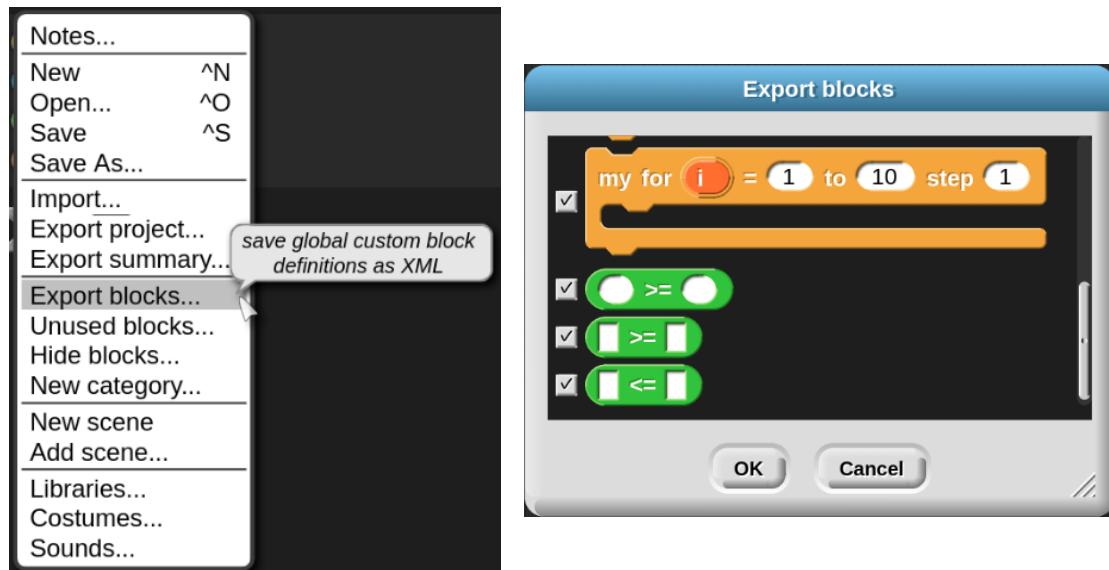


テストをしてみて問題がなければ OK をクリックしてブロックエディターを閉じてください。

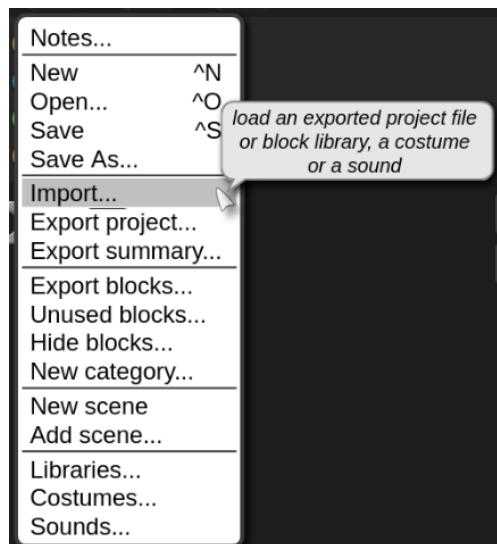
ところで、Libraries... からブロック定義をインポートしてプロジェクトを作成した場合に、普通に保存すると未使用のブロック定義も含んだファイルになります。プロジェクトを公開する場合は、次のようにして未使用のブロック定義を削除してファイルの容量を小さくしたほうがいいかもしれません。



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。



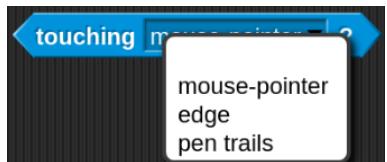
OK をクリックすると、ダウンロードフォルダに「????? blocks.xml」というファイル名で保存されます。????? のところにはプロジェクト名又は untitled が入ります。適宜リネームしてください。これを利用する時は、次のようにしてインポートすれば使用できるようになります。



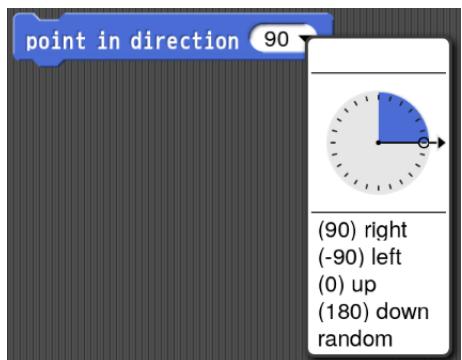
## 7 ブロック定義について

### 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。



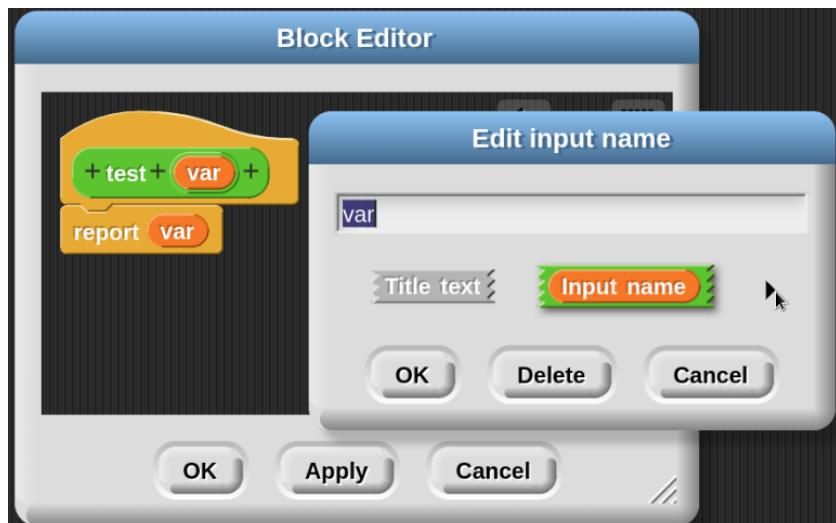
入力スロットに値をキーボードから入力することはできなくなっています。これは、後で出てくる read-only のオプションが指定されているためです。



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。

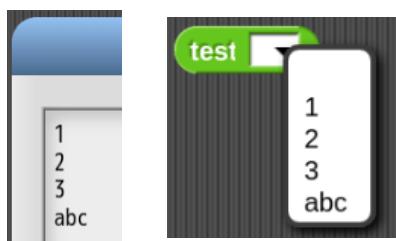
図の var をクリックすると Edit input name のダイアログが開きますから、Input name の右側にある三角をクリックします。これで、大きな Edit input name のダイアログが開きます。この暗い灰色の領域で右クリックします。すると、このようなメニューが表示されます。



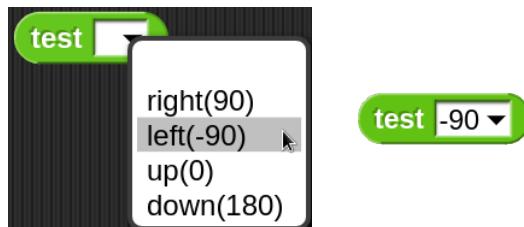
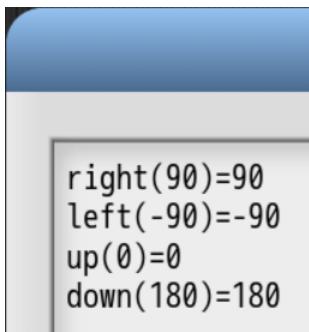
読み取り専用のプルダウン入力にしたい場合は、`read-only` チェックボックスをクリックします。  
メニュー項目を設定するには、`options...` を選択し、このダイアログボックスを表示します。



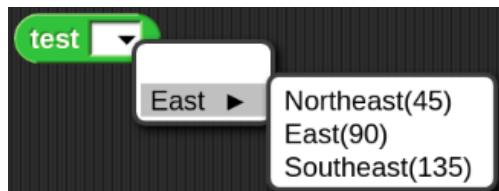
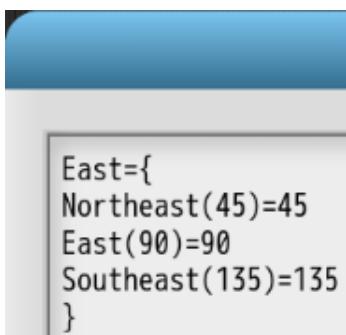
ここに各行にオプションを入れてプルダウンメニューを作っていきます。  
左のように設定して、ブロックエディターを `Apply` すると、右のような結果になります。



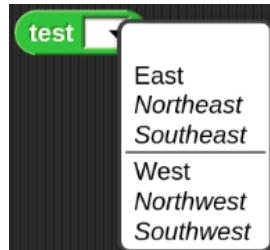
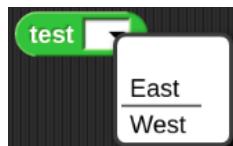
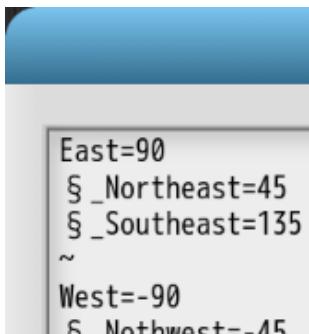
設定したものがそのまま表示され、クリックするとそれが入力値になります。  
次のように、「=」で値を設定すると、メニューには「=」の左側の項目が表示されますが、クリックされると「=」の右側にある項目が入力値になります。



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。



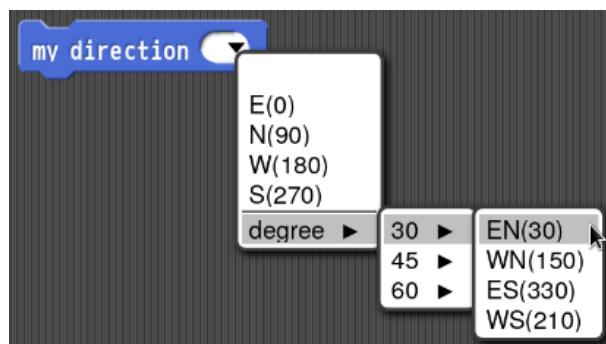
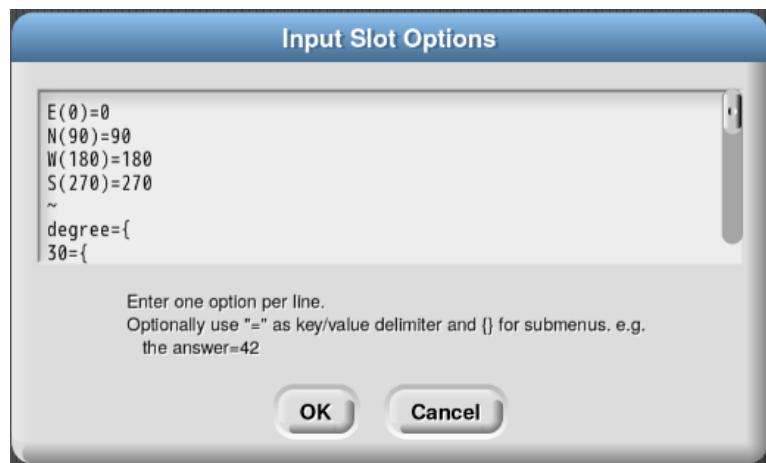
項目の前に、「 §\_ 」セクション記号とアンダースコア（アンダーバー）を置くと、シフトキーを押しながらクリックしないとプルダウンメニューに表示されなくなります。因みに、§ 記号は Windows OS の場合だと、Alt キーを押しながらテンキーから（通常の数字キーではなく）0167 と入力、Linux OS の場合だと、[Ctrl]+[Shift]+[u] を押してから、a7 を入力して [Shift+Space] で出すことができます。a7 は 0167 の十六進数コードです。OS を問わず、日本語入力モードで「せくしょん」または「きごう」と入力して変換して出すこともできます。

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成します。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。

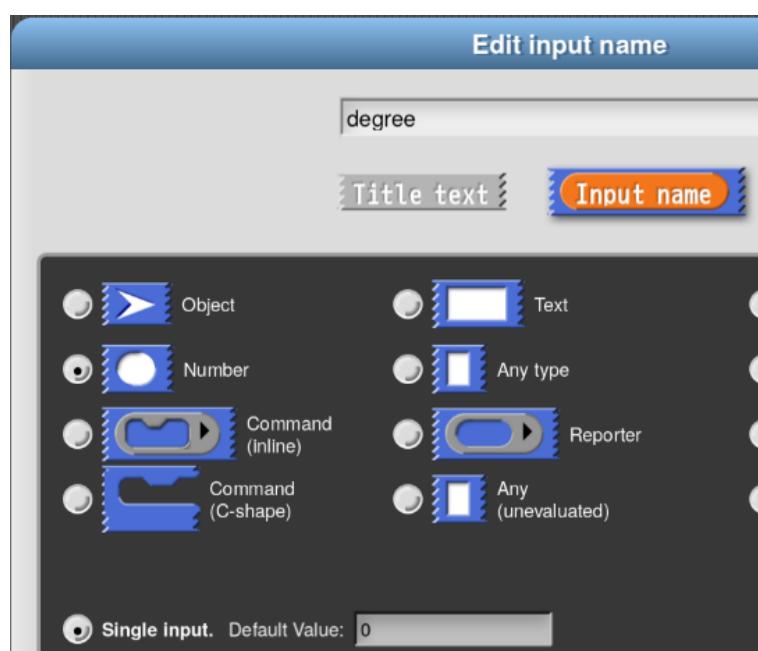
```

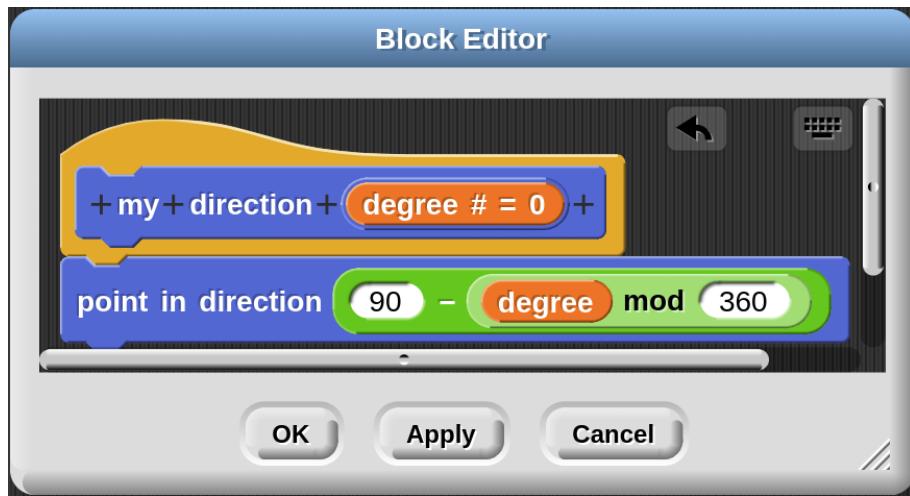
E(0)=0
N(90)=90
W(180)=180
S(270)=270
~
degree={
30={
EN(30)=30
WN(150)=150
ES(330)=330
WS(210)=210
}
45={
EN(45)=45
WN(135)=135
ES(315)=315
WS(225)=225
}
60={
EN(60)=60
WN(120)=120
ES(300)=300
WS(240)=240
}

```

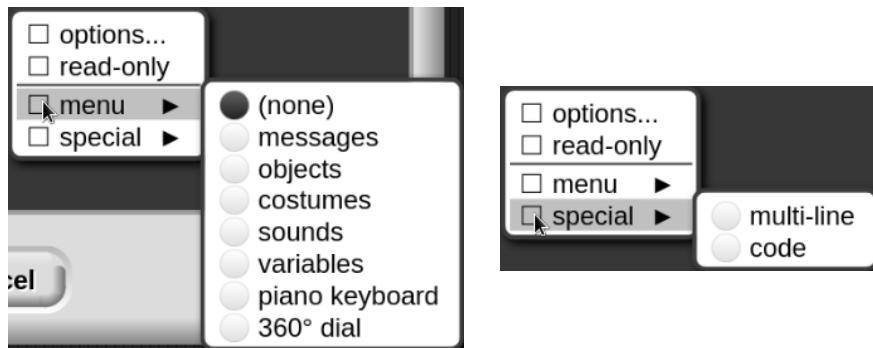


degree から、EN, WN, ES, WS のサブサブメニューを作っています。あくまで機能説明の作例ですので、あまり意味のあるものではありません。





また、menu サブメニューから選択することで、いくつかのプリミティブブロックで使用されている特別なメニューを取得することも可能です。いろいろと試してみてください。



## 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。)

square	flash	← arrowLeftThin
▶ pointRight	brush	↔ arrowLeftRightThin
▶ stepForward	✓ tick	↓ arrowDown
⚙ gears	checkbox	↳ arrowDownOutline
⬆ gearPartial	rectangle	↓ arrowDownThin
⚙ gearBig	rectangleSolid	▶ arrowRight
📄 file	circle	↳ arrowRightOutline
↗ fullScreen	circleSolid	→ arrowRightThin
↗ grow	ellipse	robot
↗ normalScreen	line	🔍 magnifyingGlass
↘ shrink	+	🔍 magnifierOutline
█████ smallStage	cross	▣ selection
█████ normalStage	crosshairs	○ polygon
▶ turtle	paintbucket	○ closedBrush
▶ turtleOutline	eraser	♫ notes
█████ stage	pipette	📷 camera
█████ pause	speechBubble	📍 location
🚩 flag	speechBubbleOutline	❗ footprints
● octagon	↗ loop	⌨ keyboard
☁ cloud	⬅ turnBack	⌨ keyboardFilled
☁ cloudGradient	➡ turnForward	🌐 globe
☁ cloudOutline	↑ arrowUp	🌐 globeBig
↻ turnRight	↑ arrowUpOutline	☰ list
↶ turnLeft	↑ arrowUpThin	⤒ flipVertical
⟳ turnAround	↑ arrowUpDownThin	⤓ flipHorizontal
🗄 storage	⬅ arrowLeft	🗑 trash
poster	⬅ arrowLeftOutline	🗑 trashFull
		↳ new line

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、 **test ➤ [ ]** になります。

定義を **\$Sturtle-1.5-0-255-255** に変更すると、 **test ➤ [ ]** 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 **\$nl** 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「\$」 **\$test-1-0-0-255** 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。

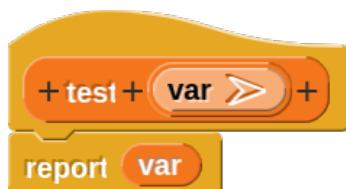
Command 型は、なにかを実行する、上下に凹凸がついたジグソーパズルピースのような形のコマンドブロックを入れられるものです。Reporter 型は、なにかしらの値をリポートする、楕円形のリポーターブロックを入れられるものです。Predicate 型は、真理値 true か false をリポートする、六角形のブロックを入れられるものです。（「真理値」は「真偽値」と表現されることもあります。）

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 ≫	object

#### 7.3.1 Reporter 型

Object を選択して次のような定義にすると、 というブロックができます。

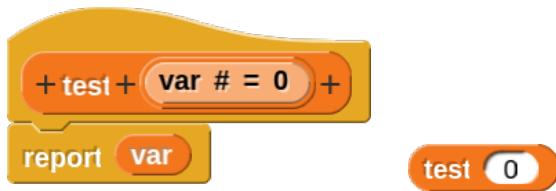


入力スロットには、キーボードからなにかを入力することはできません。スプライト、コスチューム、サウンドなどオブジェクトのドロップ入力を想定するものです。

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。

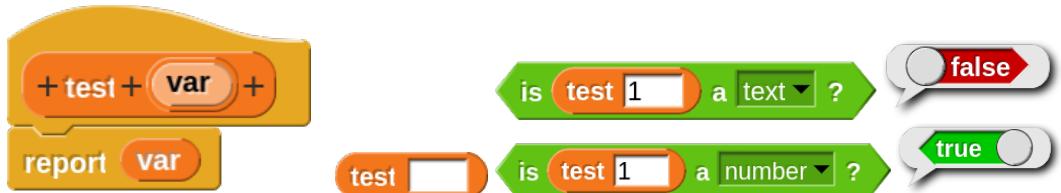


Default Value を指定すると、

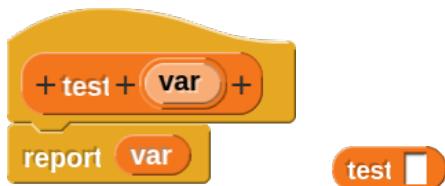


既定値を設定することができます。

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。



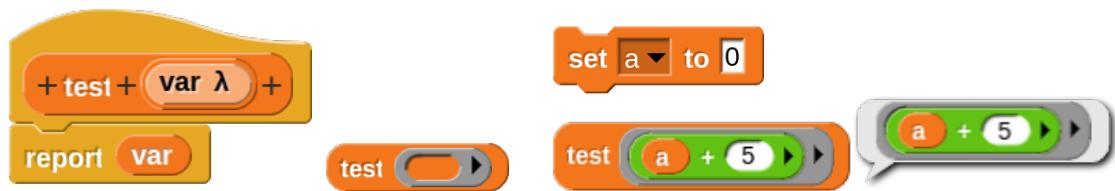
入力スロットにキーボードから数値でもテキストでも入力できることをアピールするのが、Any type です。Text とは入力スロットの形がちょっと違います。



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。



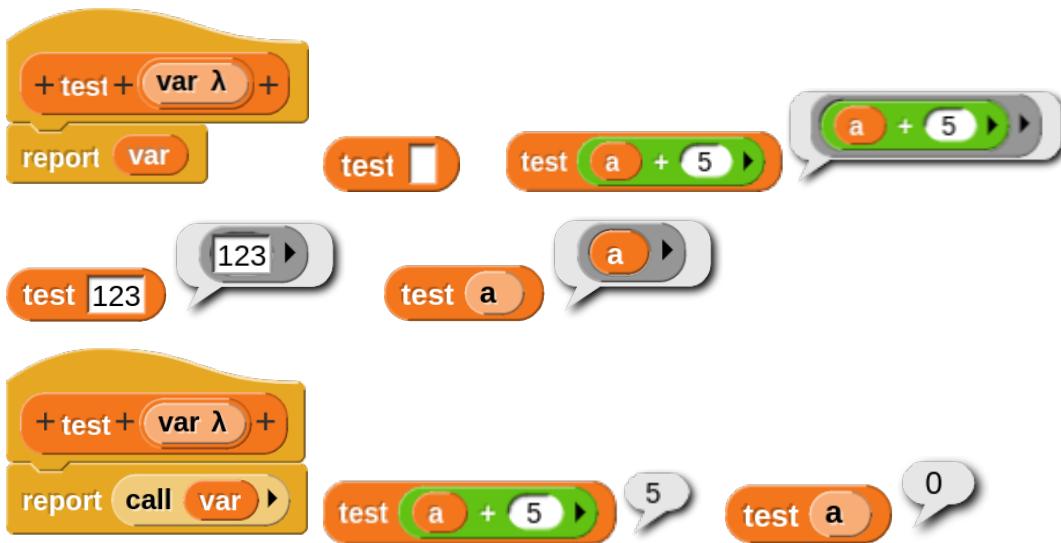
ただし、変数単体だと次の Any(unevaluated) とは違い **test [a]** とリング付きではなくなってしまいます。（ $a = 0$  とセットされているとします。）その場合は手動でリングを付けてから入力スロットにドロップする必要があります。

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。



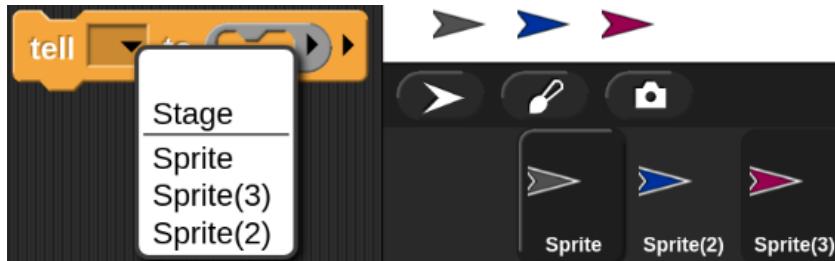
a にリングが付いていない状態なので以前はエラーになりましたが、無表示に変更されたようです。

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、みたいな使い方もするので、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。

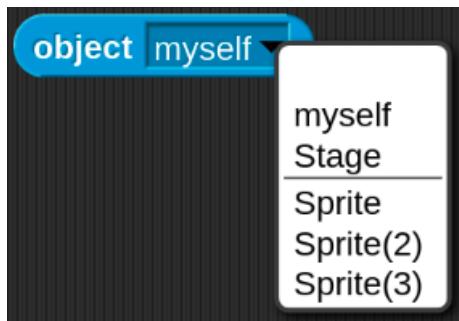


### 【 Object についての補足 】

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。



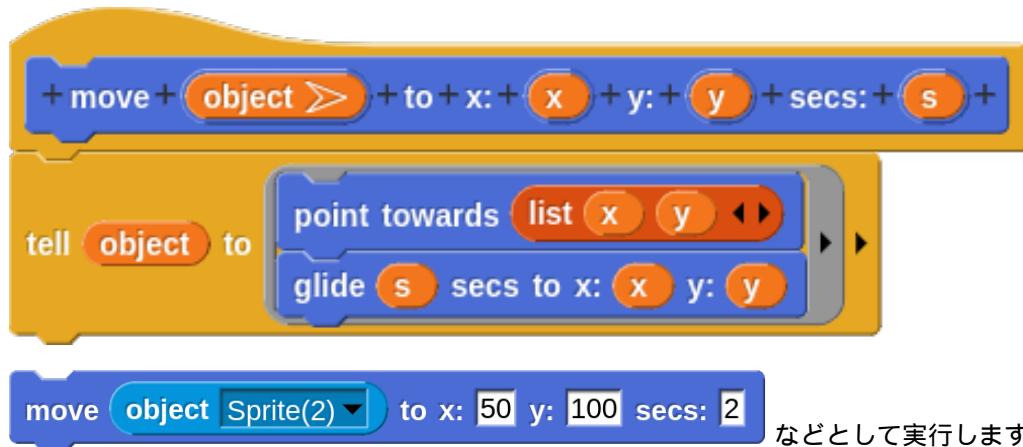
この入力スロットにオブジェクトを示す変数などをドロップすることもできます。 Sensing パレットにある Object リポーターブロックの入力メニューを開くと tell ブロックと同様に指定可能なオブジェクトを選択することができます。



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。



### 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。

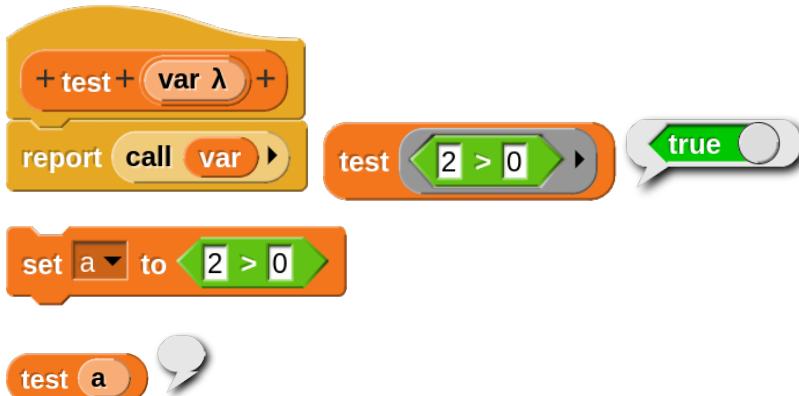
Boolean です。



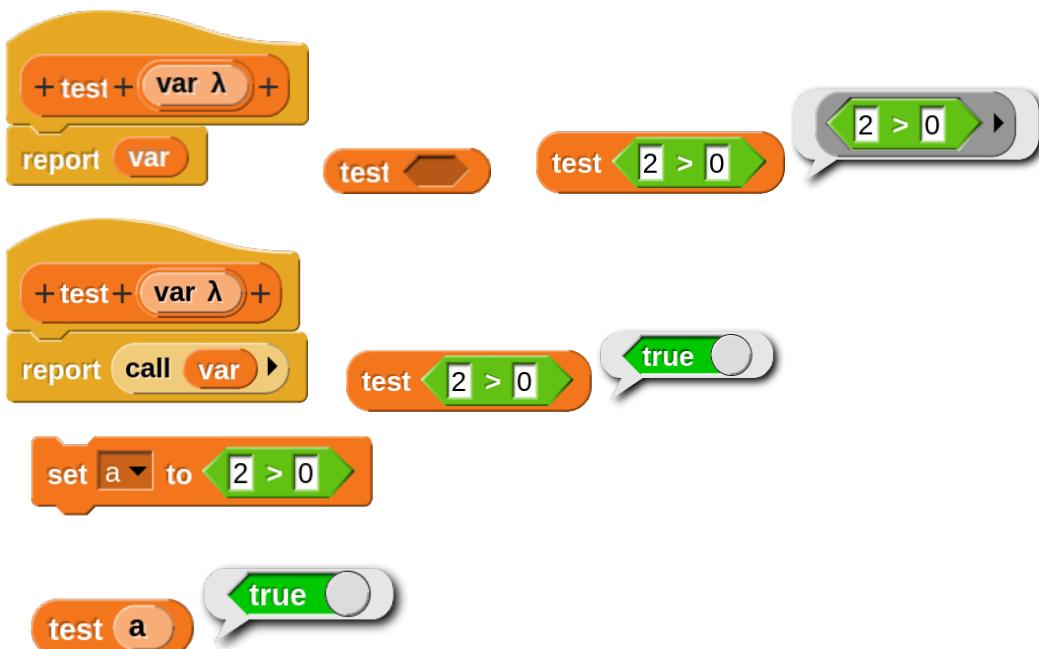
Predicate です。



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。変数単体に対して、Reporter と同じことがここでも起こります。



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。



### 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);  
}
```

のようになると、0、1、2、3、4 と表示するプログラムになります。 ( ; ; ) 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。

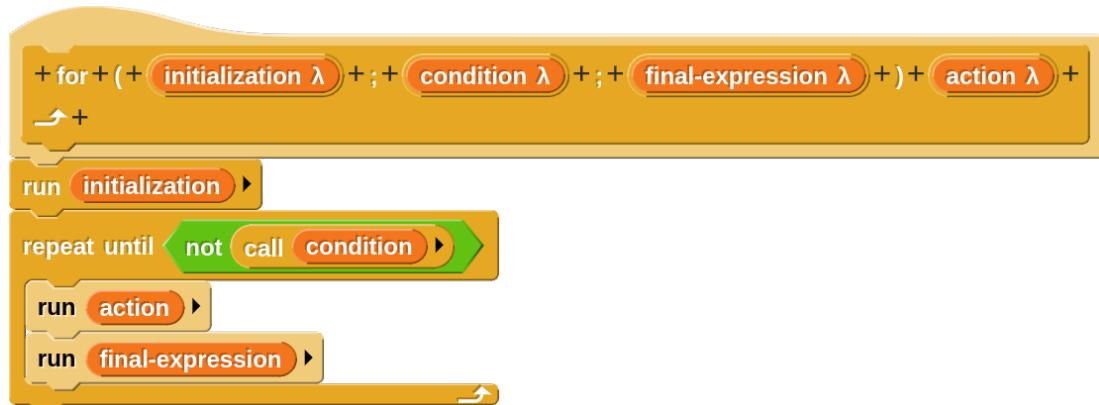
i = 0 の部分が初期設定で、初めに一回だけ実行されます。

i < 5 の部分が実行を続けるかどうかのテストをします。

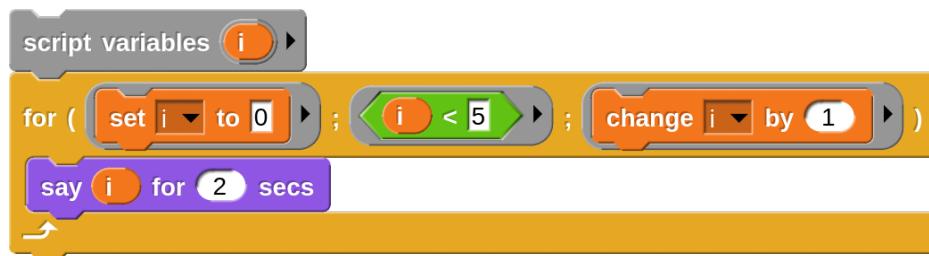
i++ の部分は次の繰り返しに進む前に実行されます。 i++ は i に 1 加算する処理をします。

{ } の内部がループの本体です。

以下が定義です。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。



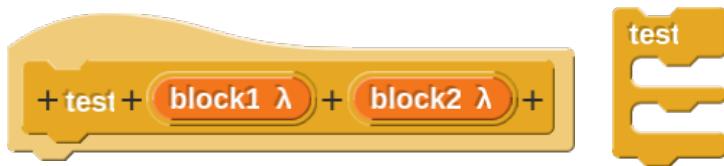
なお、initialization と final-expression は Command(inline) 型で、condition は Predicate 型です。action は Command ( C-shape ) 型 です。



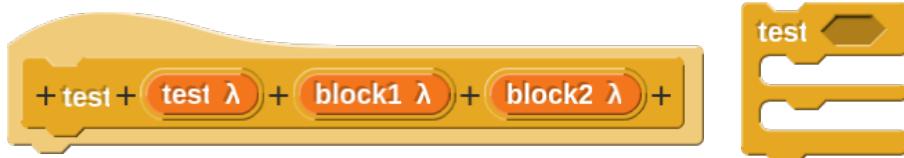
for ループは C 型が 1 つでしたが、2 つになると E 型になります。 if else の形ですね。

Command ( C-shape ) 型の変数を くっつけてやればいくらでも増やせます。

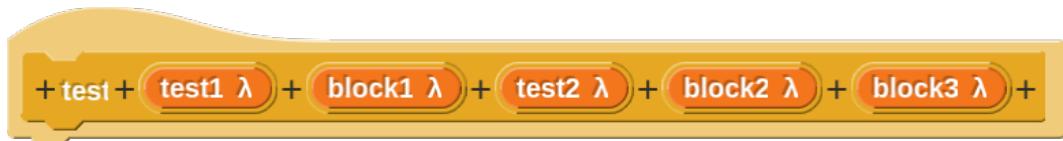




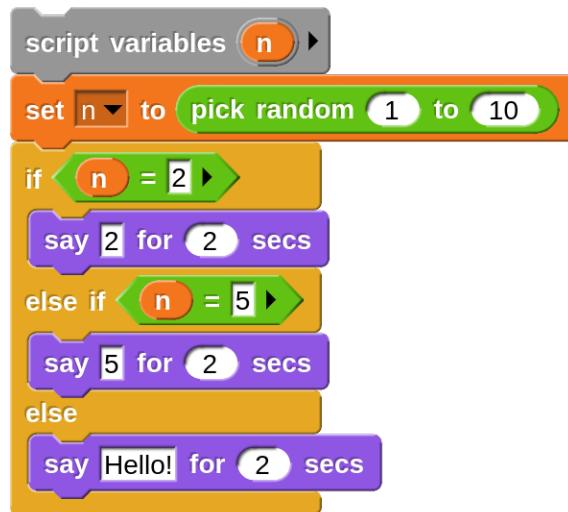
変数 block1 の前に Boolean(unevaluated) の変数を入れると、if else ブロックができます。



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。



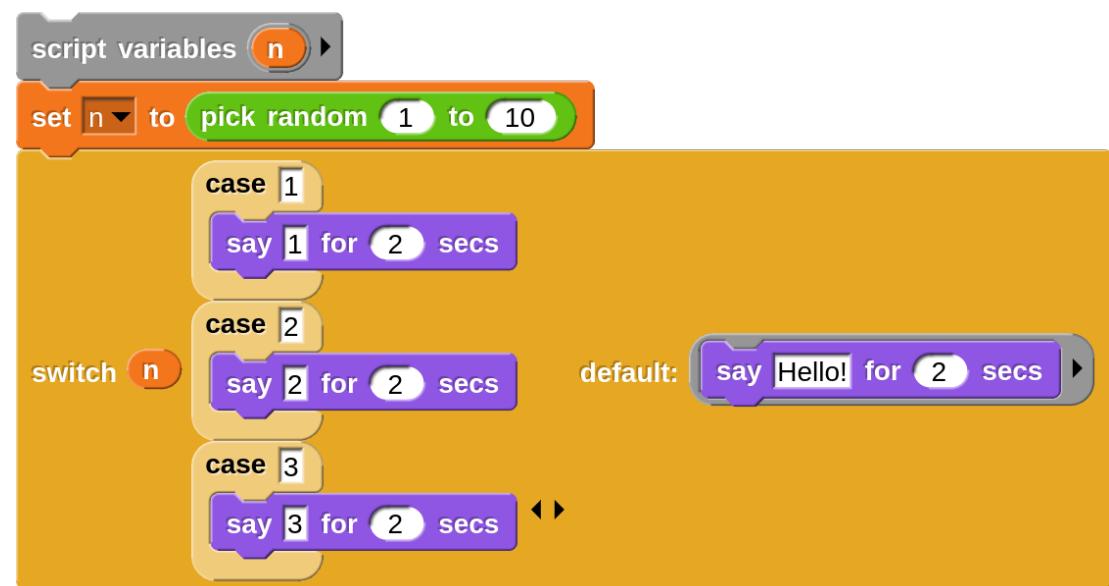
C 言語などで利用できる switch case ブロックを作ってみます。

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置をします。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。



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。これには上記のように、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。

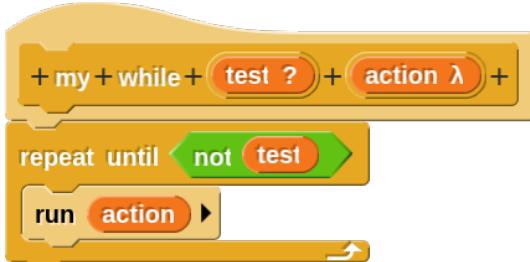


while ループを作成しながら、Predicate 型の補足説明をします。



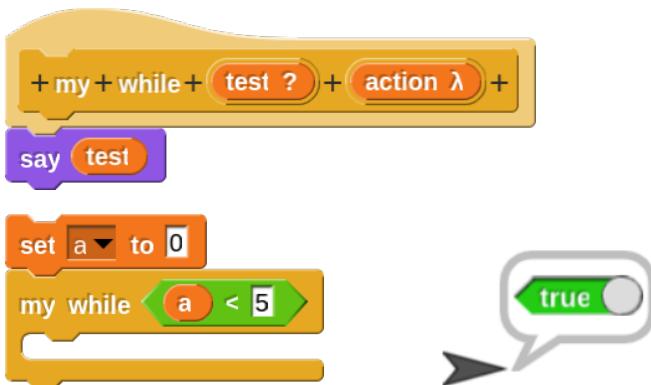
これは、 $a < 5$  のテストが true ならば、my while 内のループを繰り返します。

test を Boolean で、action を Command(C-shape) で次のように作成します。



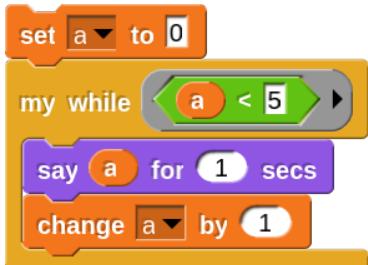
しかし、これを実行するとテストがうまくいかないようで、終わらなくなります。

定義を変更してテストしてみます。



これは、test が受け取ったのは  $a < 5$  という式ではなくて true という値だということです。

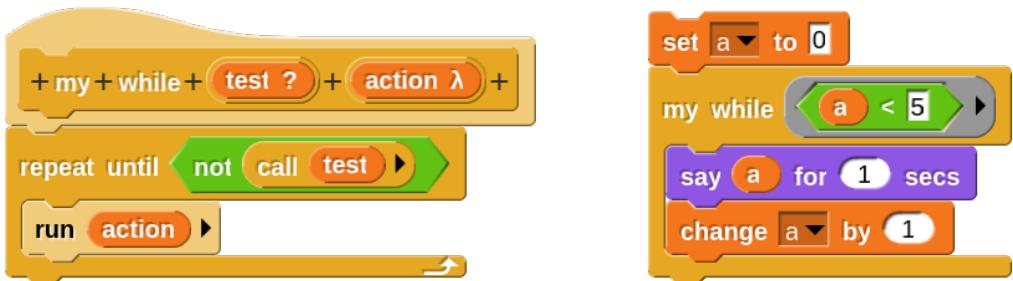
定義をもとに戻して、 $a < 5$  をリングで囲ってやってみます。



しかし、今度も終わりません。原因是、リングはリポーターですがこれは真理値ではないためです。

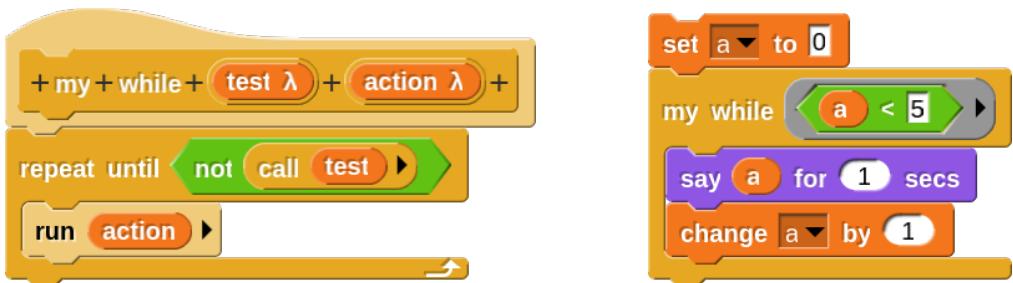


受け取った式から真理値を得るために call を使う必要があります。



今度はうまくいきました。

`test` の型を Boolean 型から Predicate 型に変更すると、  
リングで囲ってからドロップする必要がなくなります。



さらに、Predicate 型から Boolean(unevaluated) 型に変更すると、  
リングが消えてすっきりします。



## 8 Continuation 繰続

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

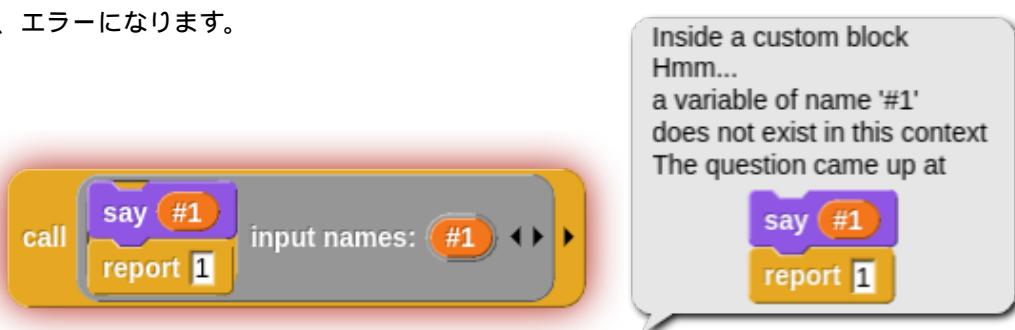
### 8.1 w/continuation

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

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



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



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



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



フォーマルパラメーターを表示させてみます。



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



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

call  
say `cont`  
report 1

input names: `cont` ↔ w/continuation

+ 



1 + call  
say `cont`  
report 2

input names: `cont` ↔ w/continuation

× 



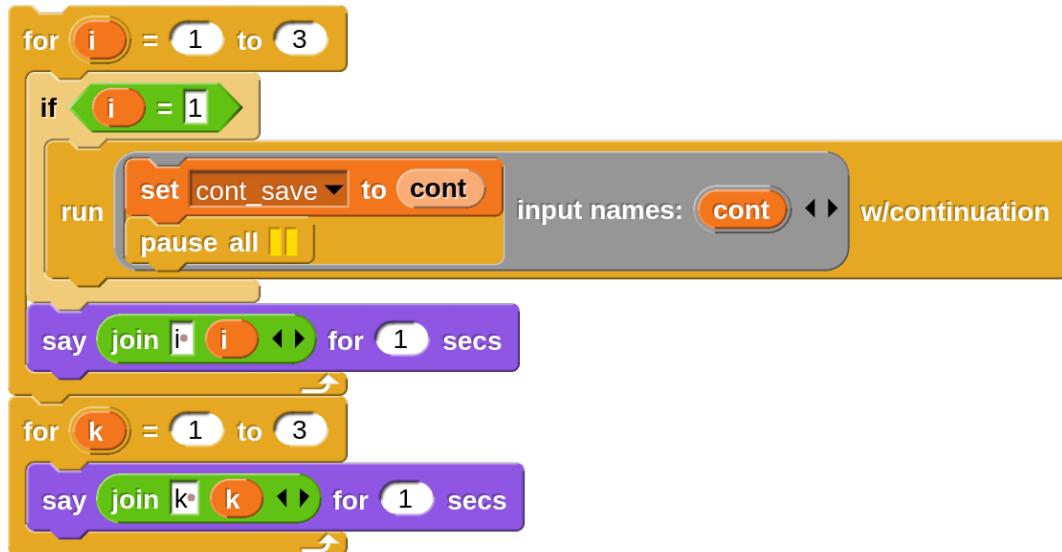
1 + 2 \* call  
say `cont`  
report 3

input names: `cont` ↔ w/continuation

▶ 



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

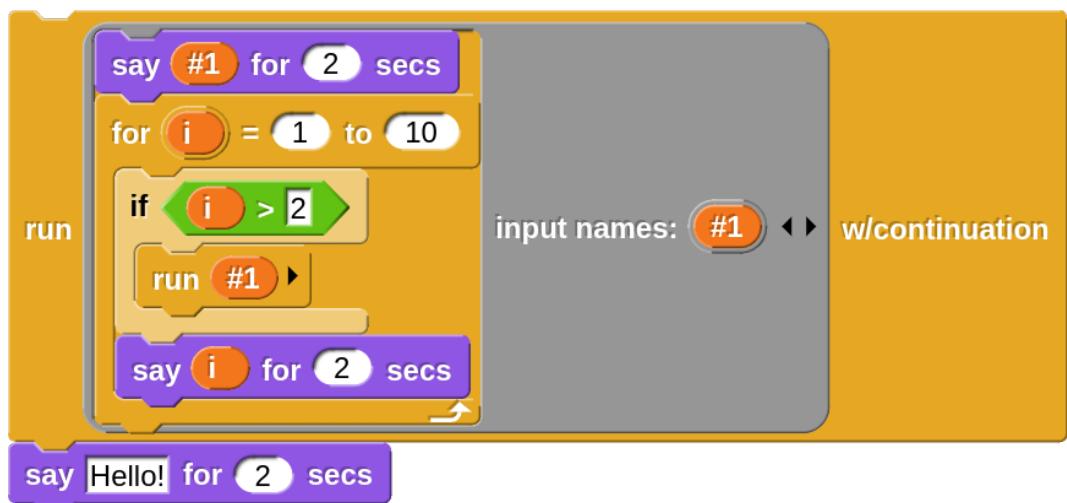


を実行してください。if ブロックの部分がなければ、i 1, i 2 ,i 3, k 1, k 2, k 3 と表示するスクリプトです。変数が表示されるようになっていると、

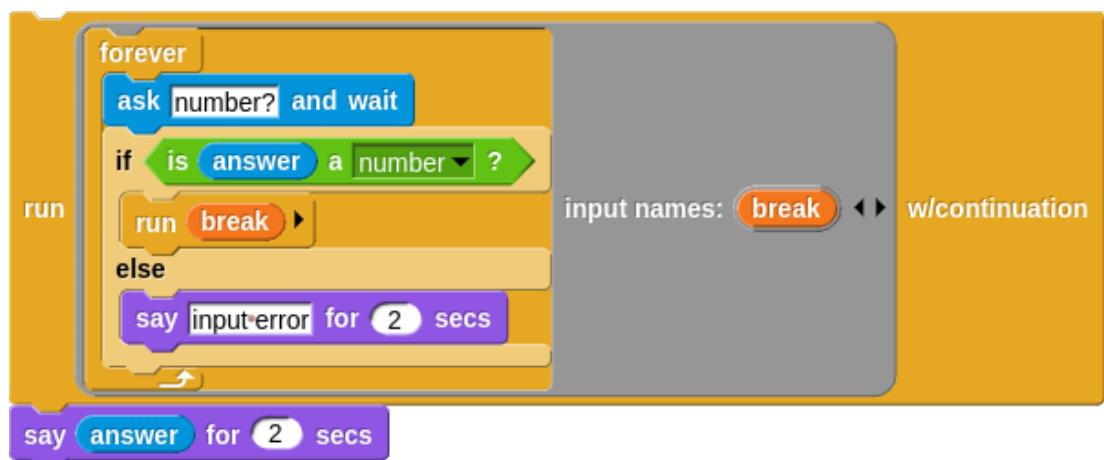
 のようにセットされた継続が表示されます。pause の状態になっているので、ここで  で終了させてください。

 をクリックすると、継続部分、つまりスクリプトの最後まで実行されます。ここではわかりやすい用途として、繰り返しなどのブロックからの脱出について説明します。

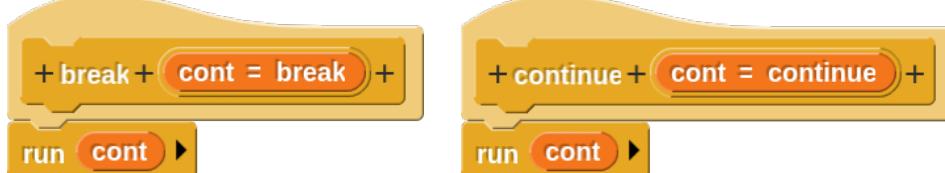
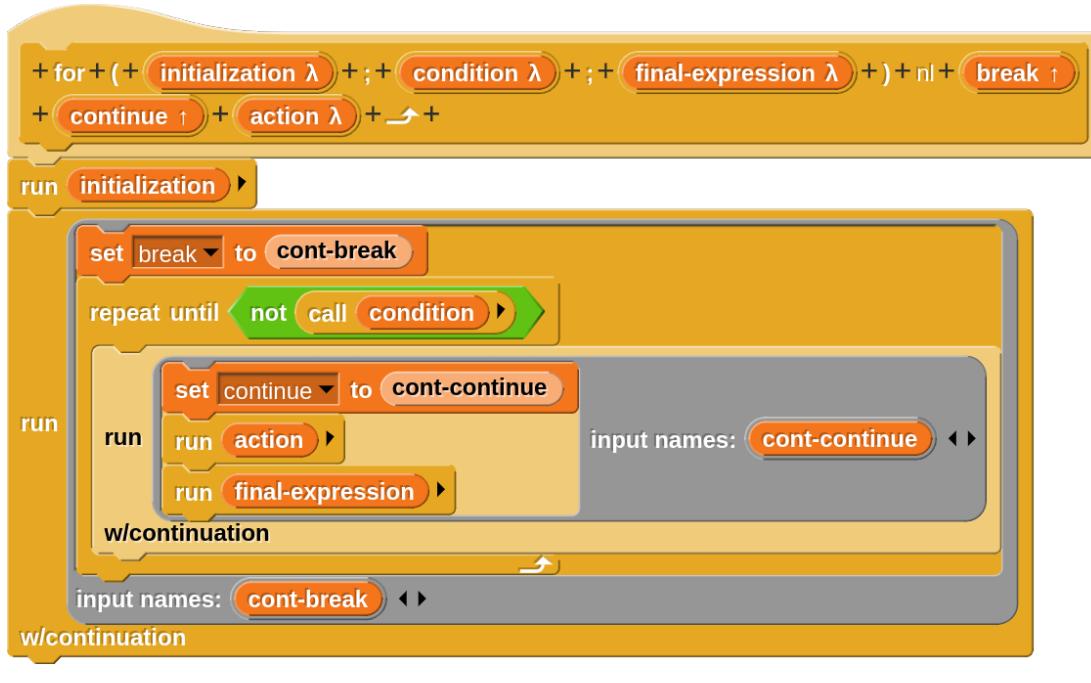
 の中に繰り返しのブロックを入れてやります。その内でフォーマルパラメーター #1 にセットされる継続（このブロックに続くスクリプト）を run すると、繰り返しブロックから脱出して継続するスクリプト動作に移行します。このスクリプトでは、i の値が 3 以上になったら継続スクリプトの実行に移行、つまり繰り返しブロックから脱出します。（最初にフォーマルパラメーター #1 にセットされる継続部分を表示します。）



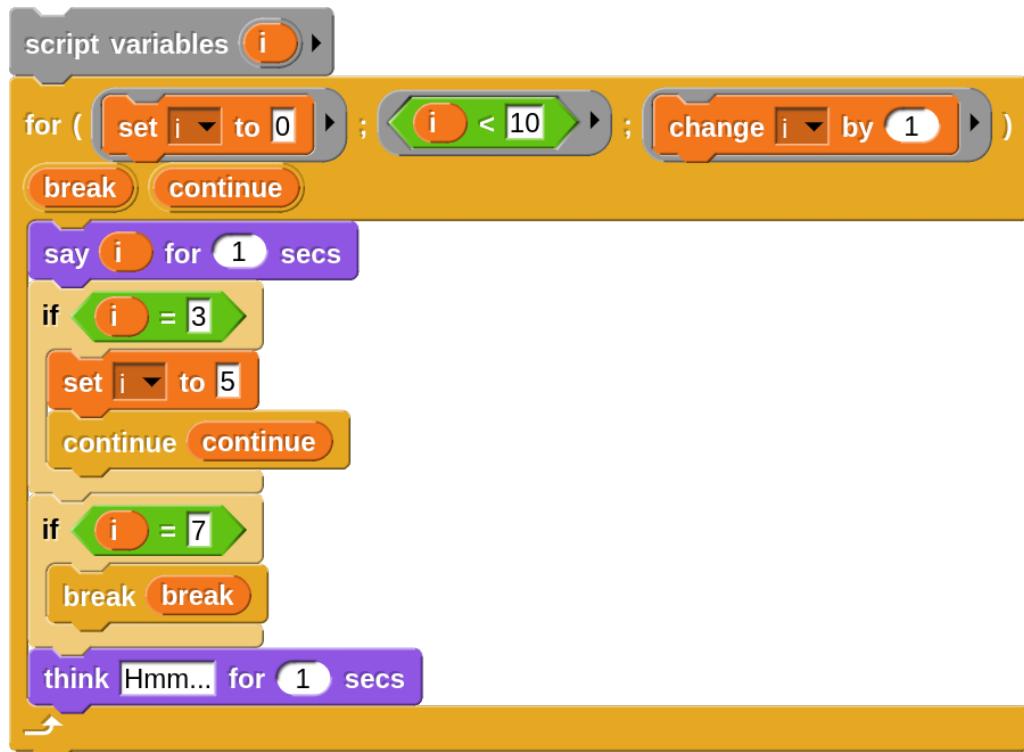
同じような例です。これを使わなくても repeat until でできますが。



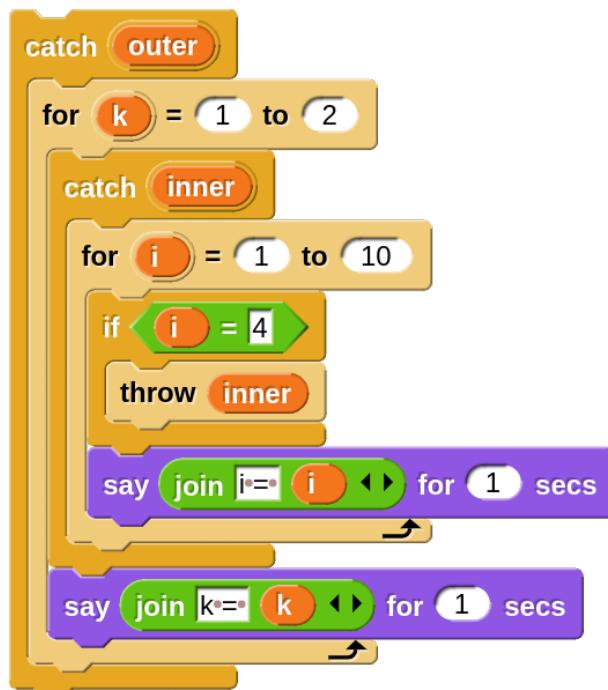
継続の機能を利用すると、C 言語風 for ( ; ; ) ループに break や continue の機能を持たせることができます。break はループから脱出する機能で、continue はループの先頭に戻る機能です。



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



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



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

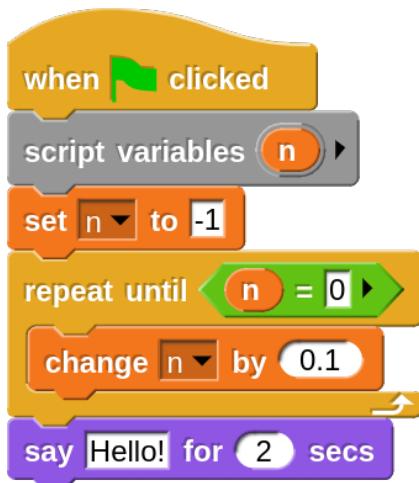
## 9 その他

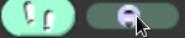
### 9.1 デバッグ

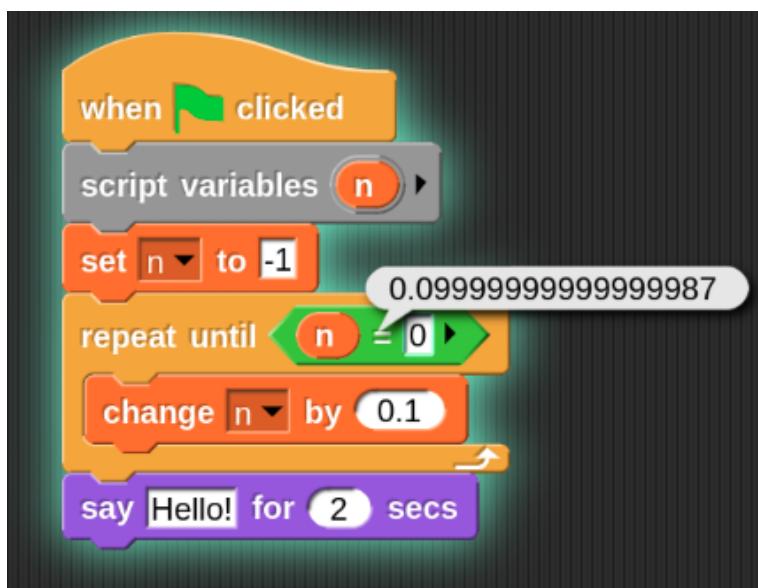
Snap! にはデバッグの機能が用意されています。

 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をステップ実行にしたり、実行スピードを遅くしたりできます。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。

次のスクリプトはバグがあって終了しません。



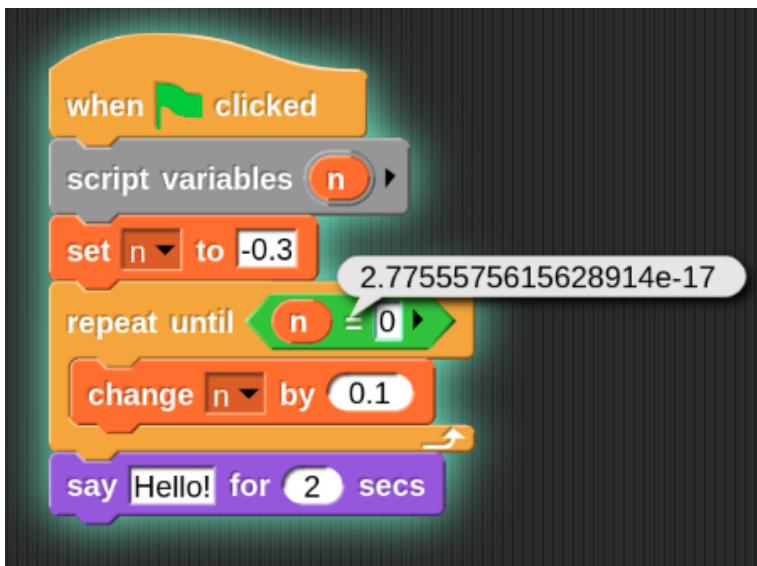
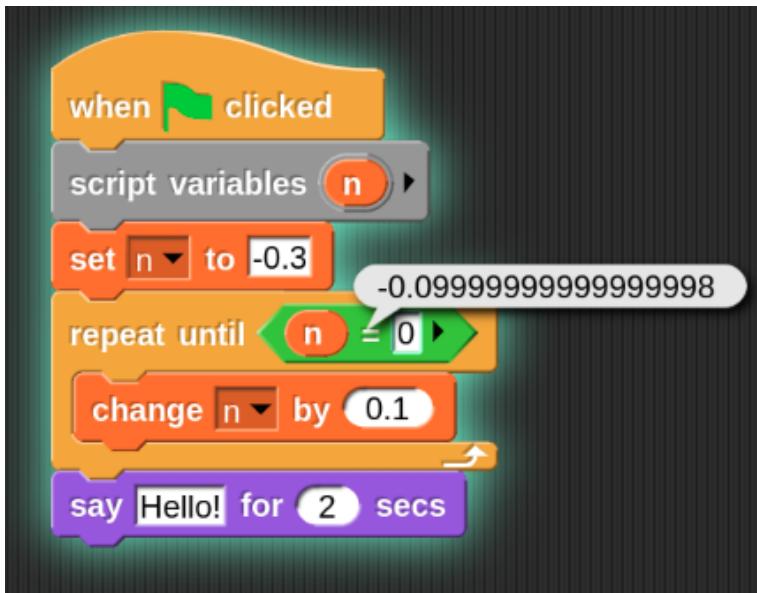
 ボタンをクリックしてデバッグモード  にして実行してみます。ただし、 のボタンで実行すると、変数などの値をリポートしてくれません。直接スクリプトをクリックして実行してください。



n の値が 0 を超えてしまいます。

n の初期値を -0.2 にしてみると、n = 0 が true になり正常終了します。

n の初期値を -0.3 にしてみると、0 を通り越してしまいます。



これは数値を浮動小数点数で扱っているために起こることなのですが、気が付きにくいことです。終了判定を  $n = 0$  or  $n > 0$  などにしなければなりません。

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。逆に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。

## 9.2 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

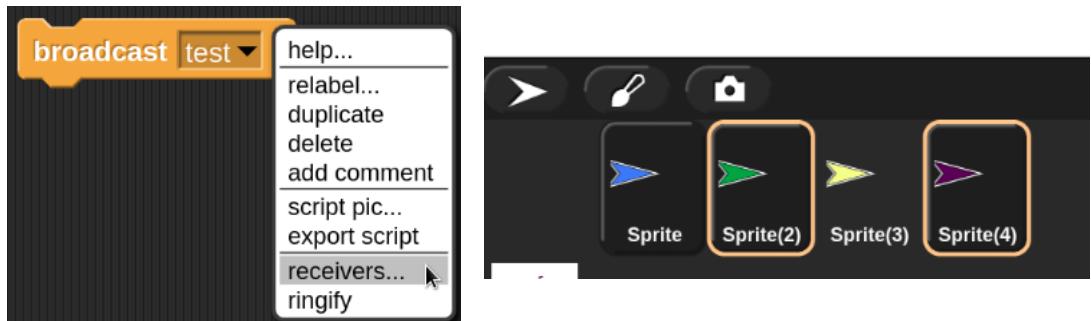
メニュー1 ▶  
メニュー2 ▶  
メニュー1.1  
メニュー1.2

script variables レベル menue ◀▶  
set menue ▾ to list 1.初級 2.中級 3.上級 ◀▶  
ask list レベルの選択 menue ◀▶ and wait  
say join answer が選択されました ◀▶ for 2 secs  
set レベル ▾ to index of answer in menue  
say レベル for 2 secs

## 9.3 broadcast の検索オプション

例えば、Sprite で broadcast test ▾ ▶ を使用していて、

**when I receive test** 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。



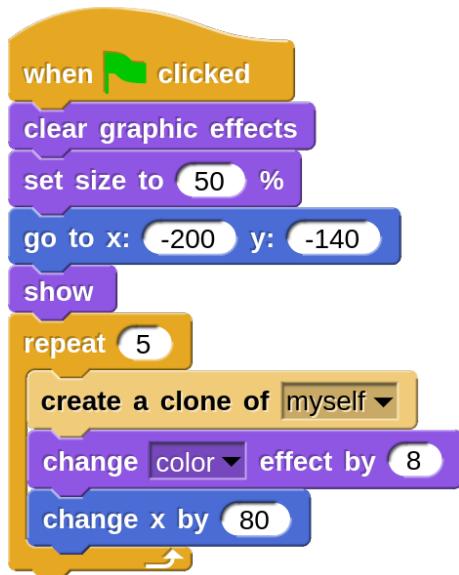
## 9.4 クローン

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。

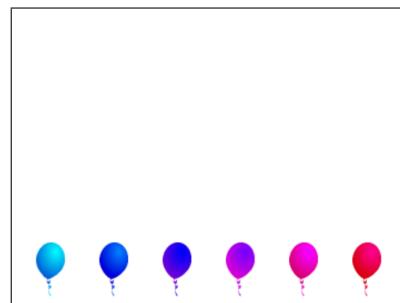
### 9.4.1 テンポラリクローン

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、一番右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。

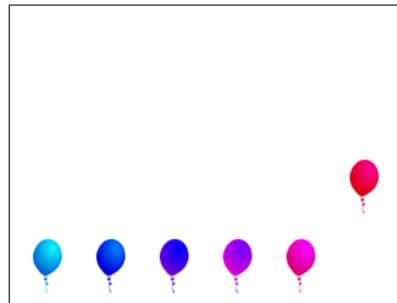


```

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
  end
  repeat (10)
    change y by 10
  end

```

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
  end

```

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
  end
  delete this clone

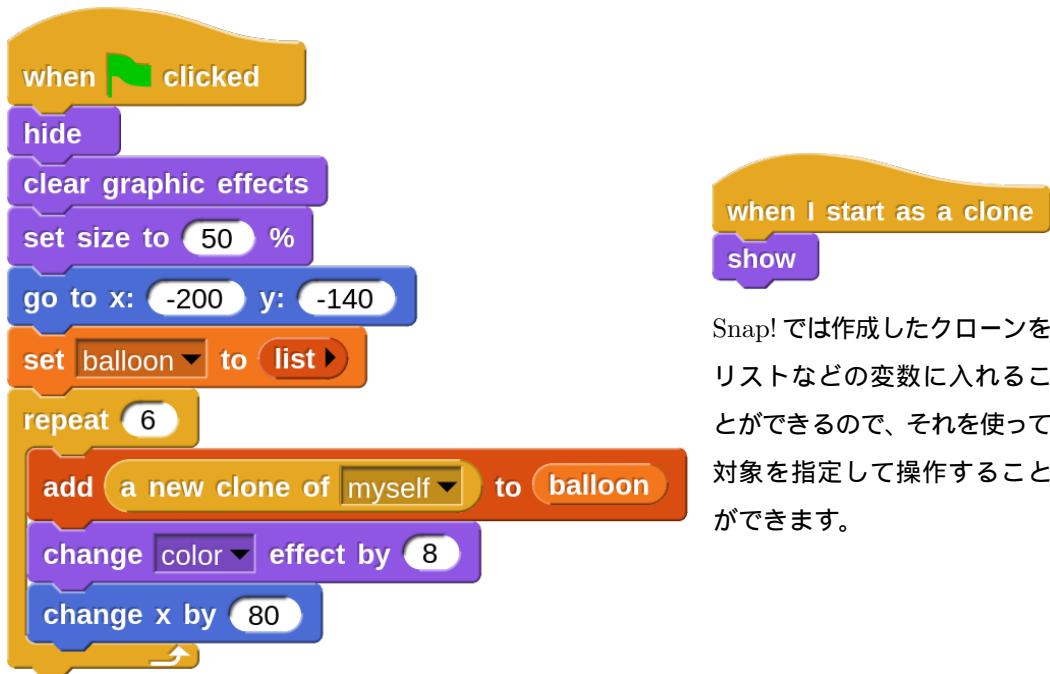
```

```

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

```

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

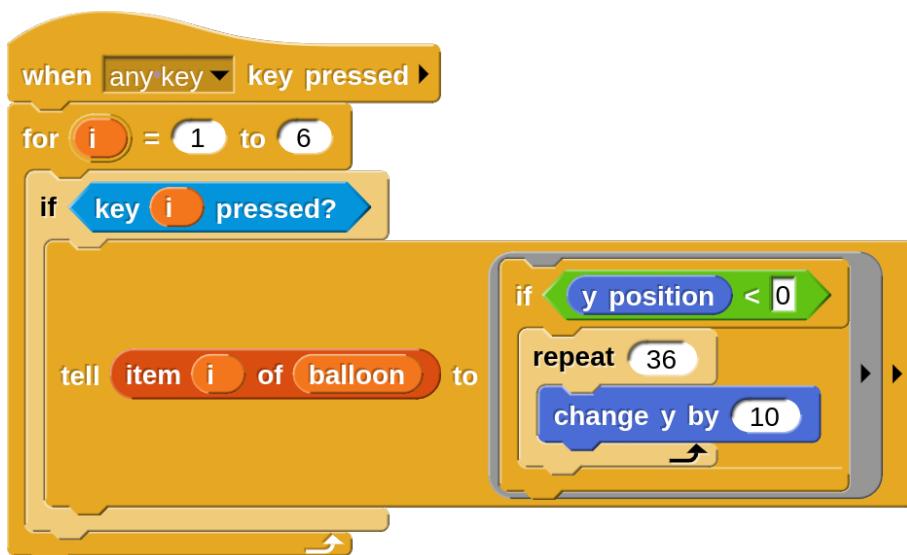


### when I start as a clone

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。

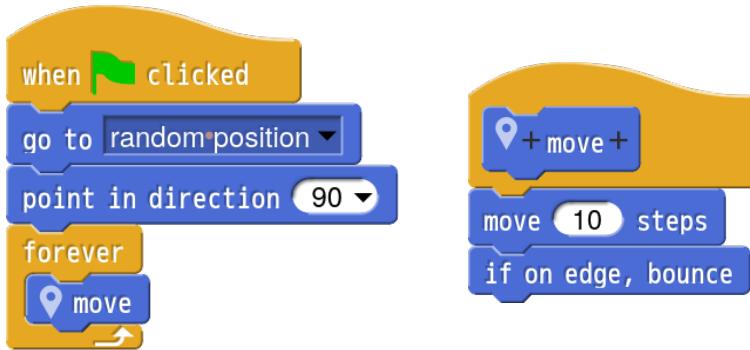
このスクリプトではクローンを入れる順番がリストの1番目 2番目ということを利用して、1のキーが押されたら1番目を、2のキーが押されたら2番目のクローンを対象として操作します。



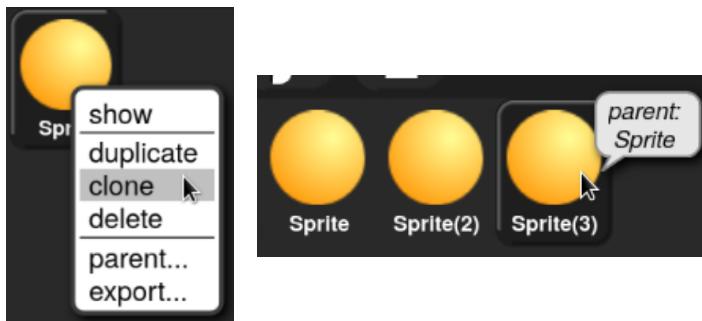
#### 9.4.2 パーマネントクローン

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。何種類かの 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。



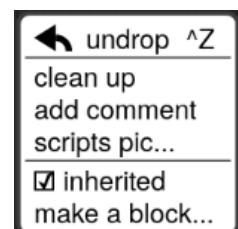
スプライトコラルで Sprite を右クリックして、クローンを二つ作成します。作成されたクローンのところにマウスカーソルを置くと、parent: 親が Sprite であることを表示します。



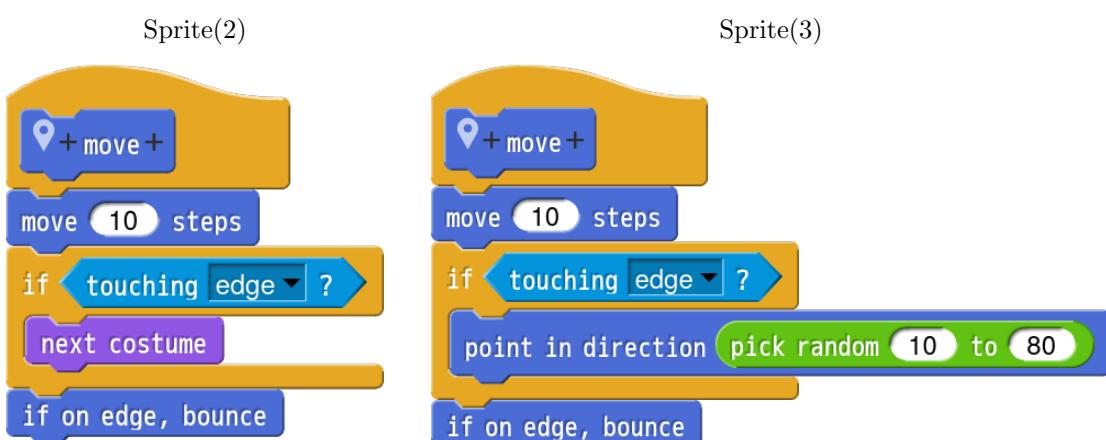
クローンである Sprite(2), Sprite(3) には、Sprite と同じスクリプトがコピーされています。親である Sprite のスクリプトを変更すると、子である Sprite(2), Sprite(3) のスクリプトに反映されます。例えば、サイズを 50 などと変えてみてください。

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 をオンにしてやれば、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。本体は親側と同じスクリプトですが、違った動作をさせることができます。



## 9.5 flat line ends

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

これを `set flat-line-ends ▾ to [true]` 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。

(六角形のスロット部分をクリックして、`set flat-line-ends ▾ to [ ]` や

`set flat-line-ends ▾ to [ ]` で指定することもできます。)

`pen up`

`point in direction 90 ▾`

`go to x: 0 y: 0`

`set flat-line-ends ▾ to [true]`

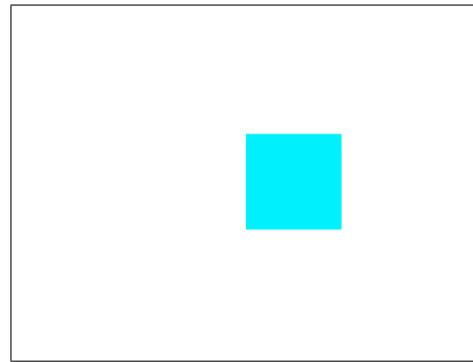
`set pen color to [ ]`

`set pen size to 100`

`pen down`

`go to x: 100 y: 0`

`pen up`



`pen up`

`point in direction 90 ▾`

`go to x: 0 y: 0`

`set flat-line-ends ▾ to [false]`

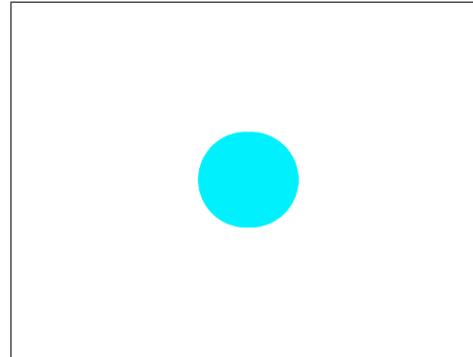
`set pen color to [red]`

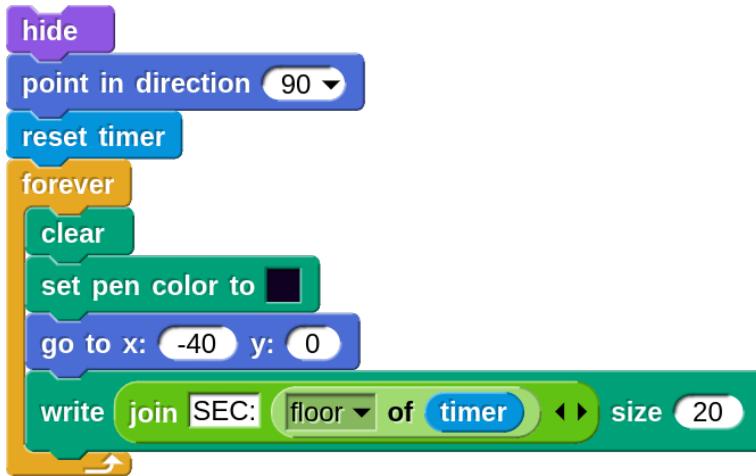
`set pen size to 100`

`pen down`

`go to x: 5 y: 0`

`pen up`

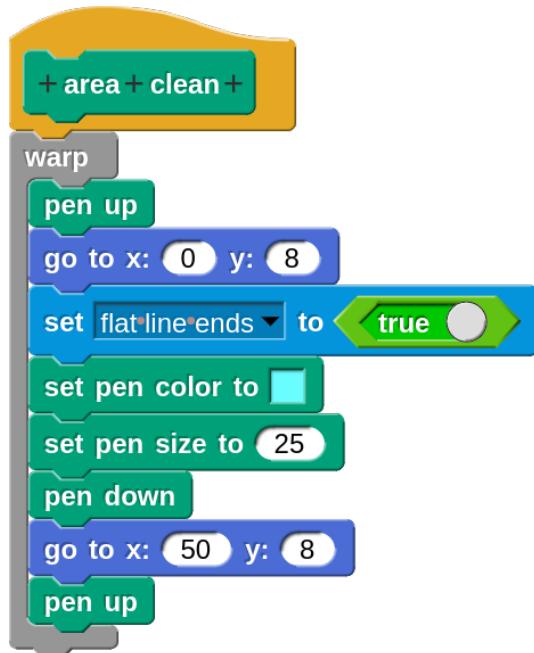
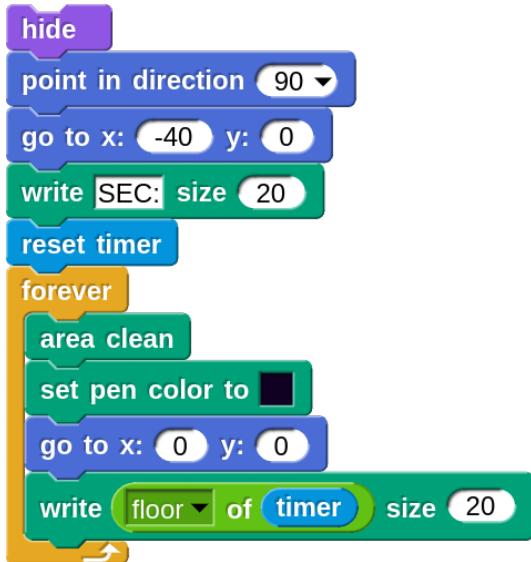




ステージにデータを表示する場合に普通は変数ウォッチャーを使いますが、Penを使って左のようになります。

ステージ全体を消しているので他の部分で Pen 描画をしている場合は、それらも消してしまいます。

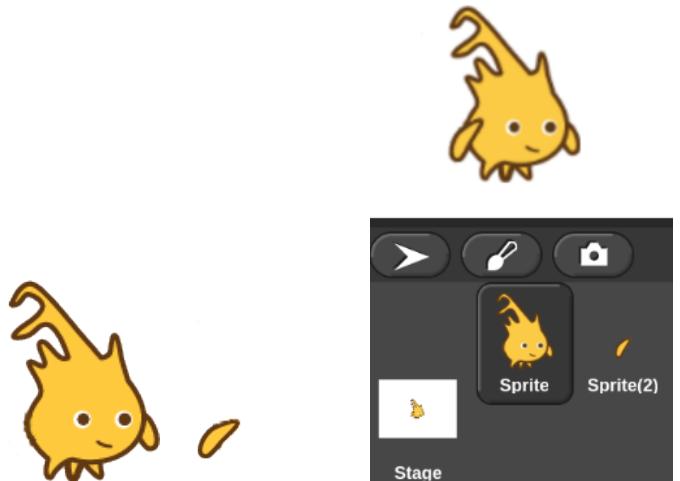
データの部分だけを消すやり方です。



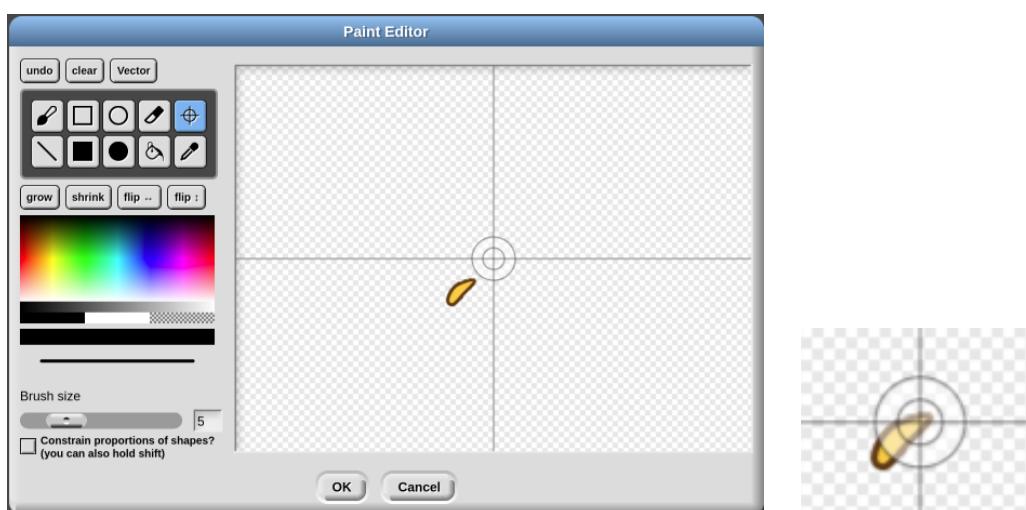
**set pen color to** で色を指定している部分をクリックしてから、ステージ上のデータを表示させたいところでクリックするとそこの色を指定することができます。

## 9.6 anchor アンカー

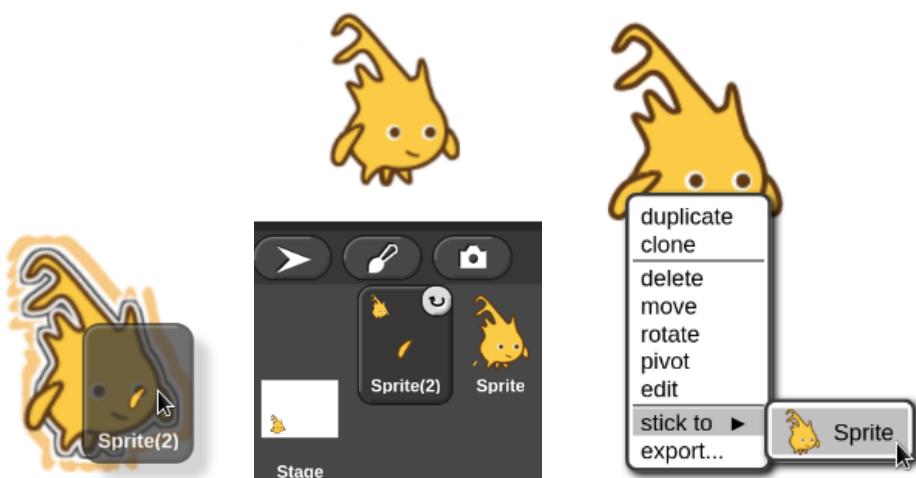
スプライトに関して、ボディーとパーツを別に用意してくっつけるというやり方があります。そうすると、パーツをボディーの一部として付随しながらパーツ自体の動きをさせることができます。alonzo というコスチュームを利用して、ボディーと腕のパーツを作ります。そして、腕のパーツをボディーの取り付けたいところに持っていきます。



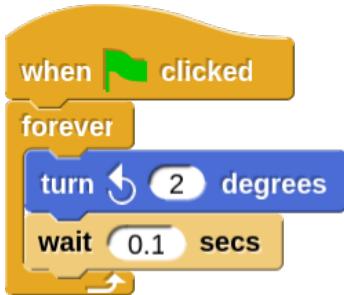
腕のパートですが、コスチュームエディターで以下のように回転の中心を設定します。



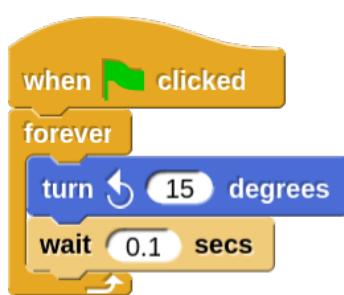
スプライトコラルにある腕のパートのスプライトをステージにあるボディーのところへドラッグ&ドロップします。すると、スプライトコラルにある腕のパートのスプライトの表示が変化します。ステージ上の腕のパートを右クリックして、stick to ですることもできます。



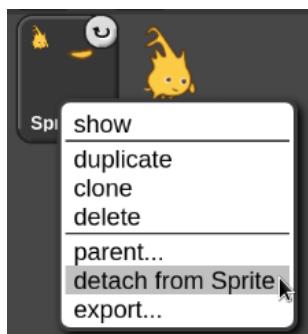
ボディー用のスクリプト



腕用のスクリプト



こんなふうに腕を回しながらボディーも回転します。



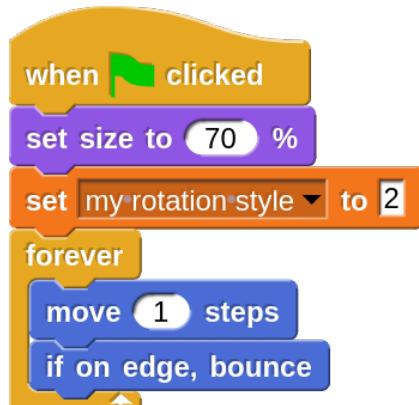
anchor の解除はスプライトコラルにある腕のパーツのスクリプトを右クリックして detach を選択します。

その場での回転は問題ないのですが、左右の動きで、端にあたって方向転換すると具合が悪いです。

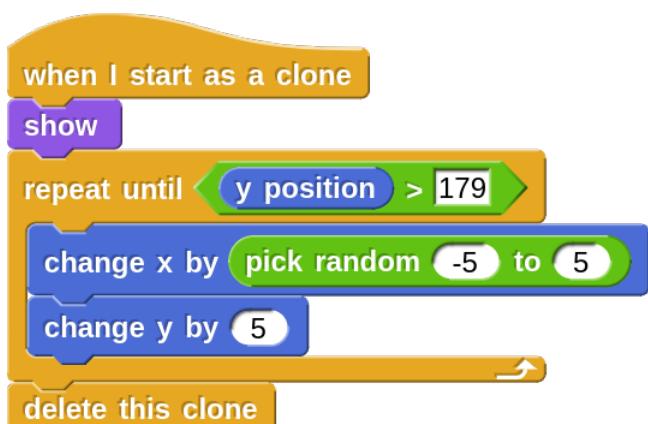
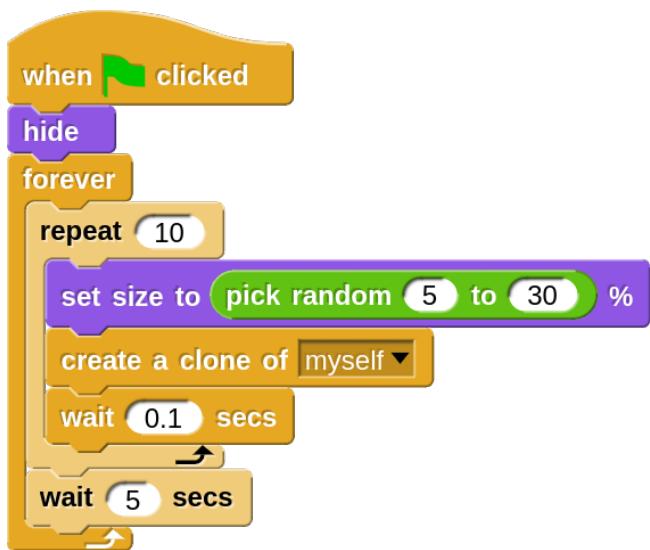
青いボールを anchor を使ってダイバーに付帯させて、吐いた息のように見せてみます。このように両者の y 座標を合わせてセットしてからボールをダイバーに stick させます。



ダイバーのスクリプトです。



ボールのスクリプトです。ボール本体はダイバーに付帯していますが、クローンはその座標を起点にして浮上していきます。[go to ダイバー](#) を使ってもできることではあります。



## 9.7 JavaScript function (オプション 5 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]
```

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

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]
```

Snap! のリストはそのまま JavaScript 側で配列として操作することはできません。ただ返すだけならば問題ありません。

A screenshot of a Scratch-like programming environment. On the left, a yellow 'call' block is shown with the code: 'call JavaScript function (list) { return list; }'. Below it is another 'call' block with the code: 'call JavaScript function (list) { list.sort(function(a,b){ return a-b}); return list; }'. Both blocks have the input 'with inputs numbers from 3 to 1'. To the right, a grey box labeled 'length: 3' contains three orange boxes labeled '1', '2', and '3' respectively.

JavaScript の配列ソートを利用しようとするとエラーになります。

A screenshot of a Scratch-like programming environment. A green 'call' block has the code: 'var l = list.toArray(); l.sort(function(a,b){ return a-b}); return l;'. It has the input 'with inputs numbers from 3 to 1'. To the right, a grey box labeled 'length: 3' contains three orange boxes labeled '1', '2', and '3' respectively. A speech bubble says 'TypeError list.sort is not a function'.

`var l = list.toArray()` で変換して配列にすると、配列として操作ができるようになります。配列は参照型ということでなのか、結果的に list 自体がソートされるため、list を返すことができるようです。

A screenshot of a Scratch-like programming environment. A green 'call' block has the code: 'var l = list.toArray(); l.sort(function(a,b){ return a-b}); return l;'. It has the input 'with inputs numbers from 3 to 1'. To the right, a grey box labeled 'length: 3' contains three orange boxes labeled '1', '2', and '3' respectively.

なお、`return l;` とすると、この場合「1,2,3」というものが返されます。これは、テストしてみると、number, list, text のチェックで false になります。つまり、数値でもリストでもテキストでもないということで使用できません。

変数 l を使わなくても可能です。こちらは比較関数を変更して降順にしてみました。

A screenshot of a Scratch-like programming environment. A green 'call' block has the code: 'list.toArray().sort(function(a,b){ return b-a}); return list;'. It has the input 'with inputs numbers from 1 to 3'. To the right, a grey box labeled 'length: 3' contains three orange boxes labeled '3', '2', and '1' respectively.

JavaScript function の使用例としてソートをやってみましたが、APL ライブライバーにソートブロックが用意されています。

Snap! にはビット演算をするブロックがありませんが、JavaScript の機能を使って自作することができます。

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、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。

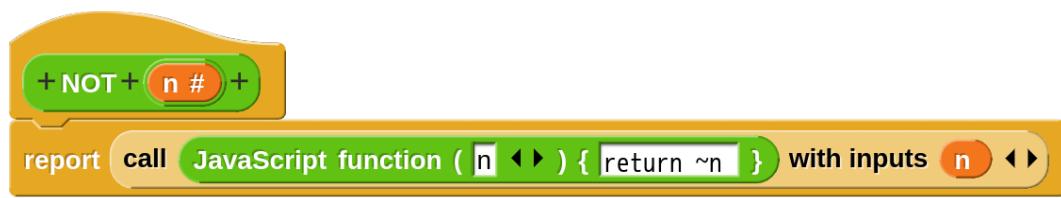
JavaScript には論理積 (AND) 演算子として `&`、論理和 (OR) 演算子として `|`、排他的論理和 (XOR) 演算子として `^` があります。

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。



”`&`” にすれば OR ブロック、  
”`^`” にすれば XOR ブロックになります。

NOT ブロックです。NOT はビット反転です。つまり、0 ならば 1、1 ならば 0 にします。



NOT 0 の NOT が -1 というのは分かりづらいかもしれません。数値の桁数を 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 なので 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いくて 4 ビットの範囲では 0000 になります。つまり、1111 は 1 に加えると 0 になる数である -1 ということになります。ニ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。JavaScript のビット演算は 32 ビット整数としてなされるようなので、0 と -1 はそれぞれ 0 と 1 が 32 個並んだものになります。

数値に対して NOT を取り 1 を加えると、その数値の負数 (負数の場合は正数) を得ることができます。



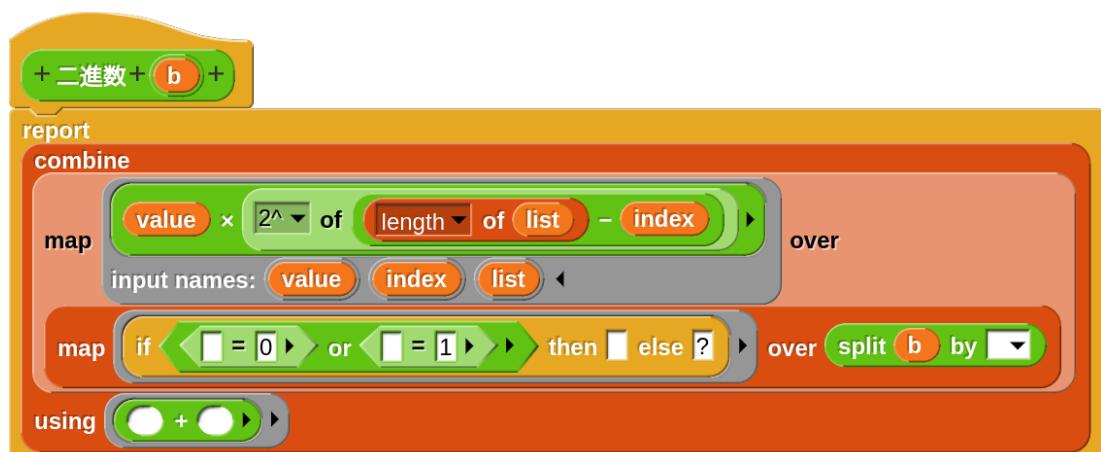
二進数で数値を指定するリポーターブロックを作成してみます。b の入力のタイプは text です。  
split で指定するのは空文字です。( デフォルトの空白を削除します。)



二進数以外を入れると「二進数エラー」がリポートされます。  
それを数値演算に使用すると NaN ( 非数 ) と表示されます。



map と combine を使うと次のようにすることができます。

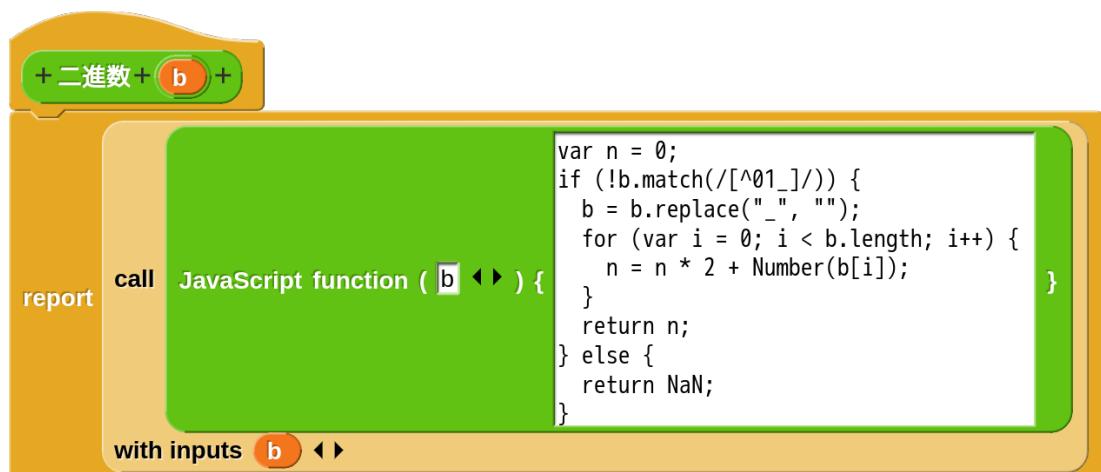


この場合、エラー表示は、二進数 123 + 5 = NaN になります。

一番内側の map で作成された list には二進数文字列の各桁が入ります。もしも「0」「1」以外ならば「?」を入れます。エラーを起こさせるためです。

「 $2^i$ 」で二進数の桁ごとの位数を求めます。1, 2, 4, 8, ... という具合です。十進数での一の位、十の位、百の位 ... にあたります。この場合は、入力された値の最上位ビットからの位数になります。各桁に位数をかけたものの合計が求める値になります。

JavaScript だと次のようにすることができます。

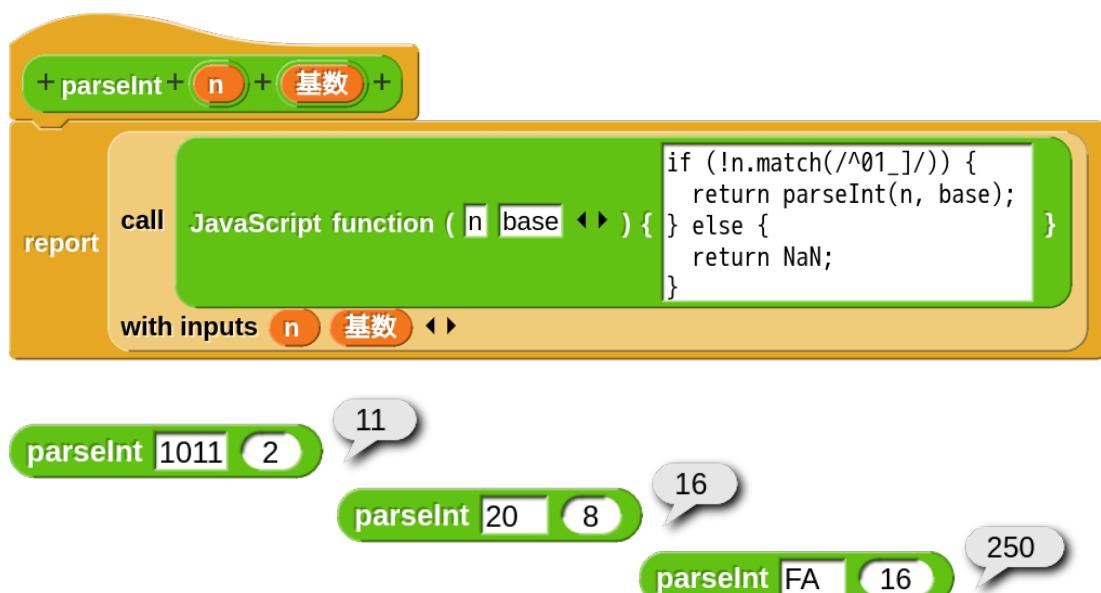


入力は文字列として扱い、二進数の文字チェックを行って不正な文字の場合 NaN を返すようにしています。「1111\_1111」のように「\_」を挿入すると分かりやすいので可能にしています。

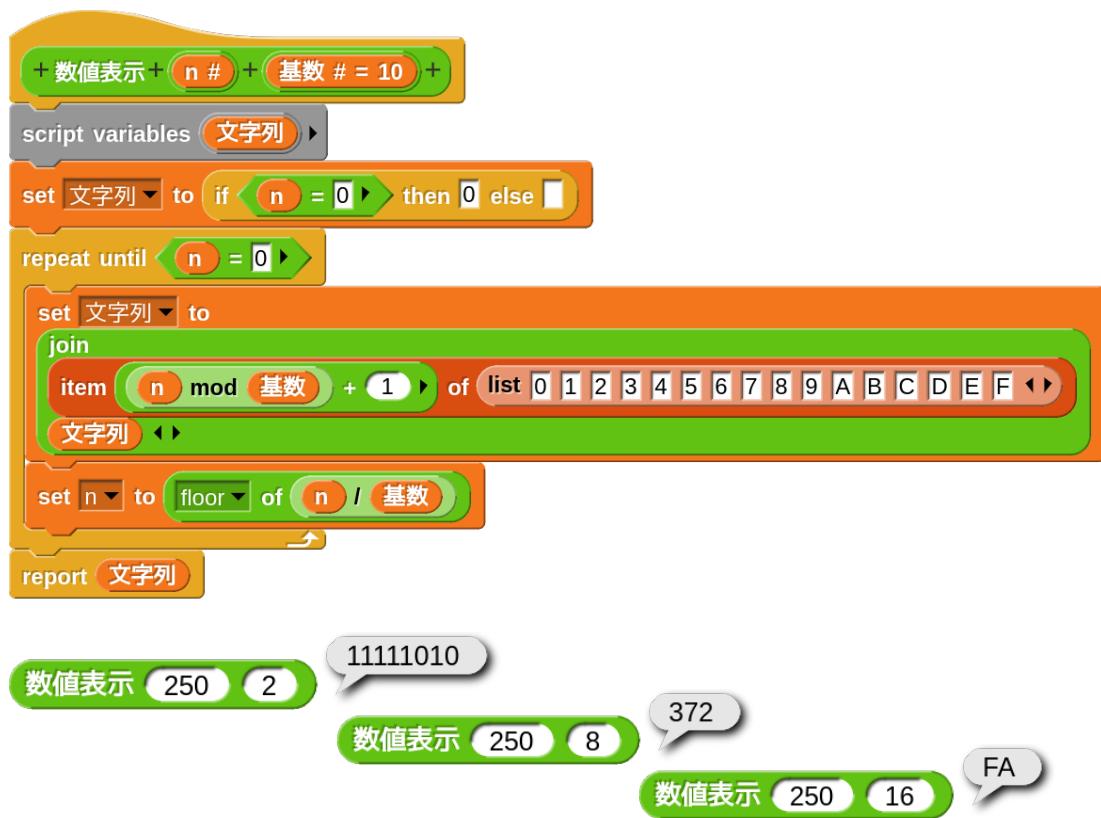
`b.match(/^01_*/)` は、0, 1, \_ 以外の文字にマッチすることを意味します。それの !(否定) なので、0, 1, \_ だけの文字列ならばということになります。

変換プログラムでは文字「\_」は不要なので `b.replace("_", "")` で削除します。

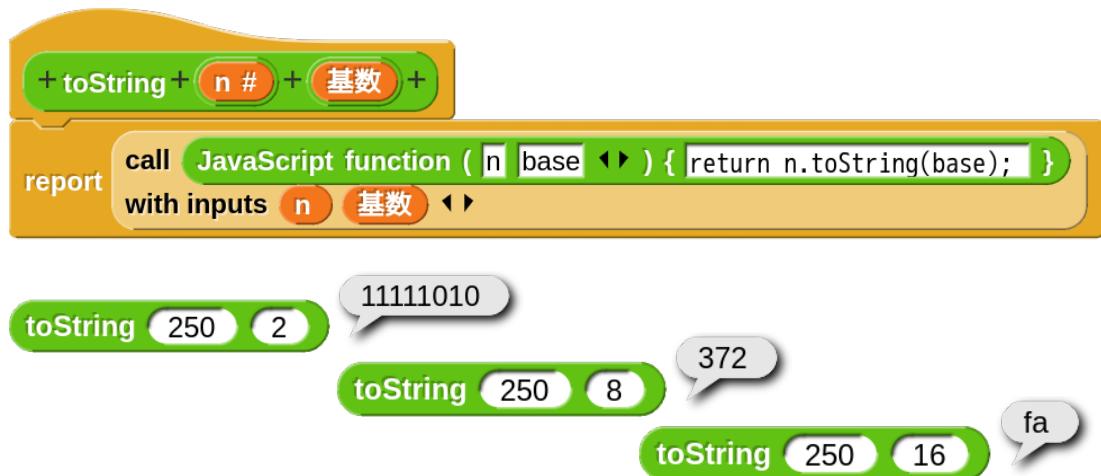
JavaScript には文字列を数値に変換する `parseInt` 関数があります。2 番目の引数で基底を指定できます。2 で二進数、16 で 16 進数の文字列からの変換になります。`parseInt` 関数は文字「\_」を無視してくれるので、`replace` 関数は不要です。なお、「base」と「基底」が合っていないが、JavaScript の引数は名前ではなく inputs のスロット位置に対応した値が設定されます。



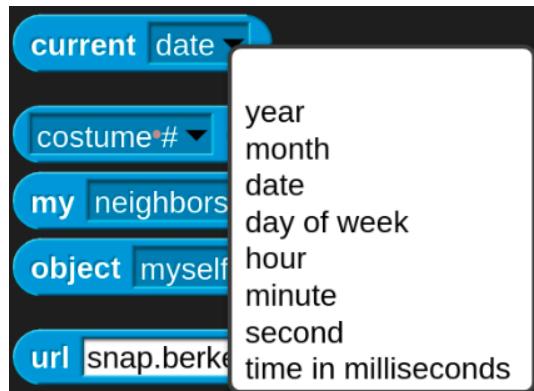
こうなると、数値を二進数表示するブロックも必要です。どうせなら基底を指定して文字列にする定義ブロックにしてみます。ただし、対応しているのは 2, 8, 10, 16 進数などです。



JavaScript では `toString` がそれをしてくれます。



## 9.8 時計



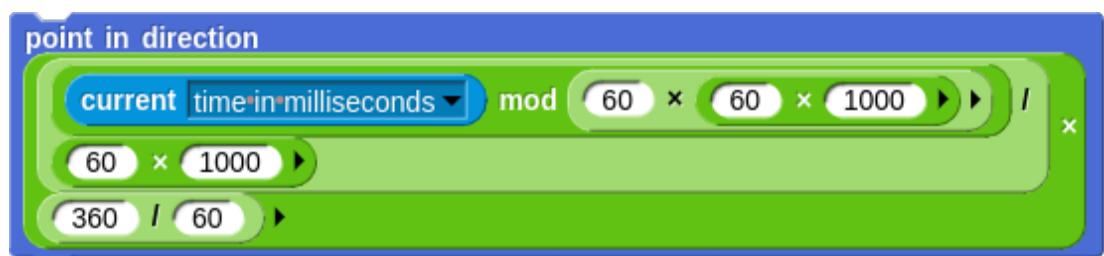
Sensing のところには日時を取得できるブロックがあります。これを使って、時、分、秒のデータを表示すればデジタル時計ができます。各データを 2 桁表示にして、区切り文字を挿入するには次のようなスクリプトになります。



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。この値から秒のデータを取り出すには 1 分 ( 60 秒 × 1000 ミリ秒 )、つまり 60000 で割った余りを求め、それを 1000 で割れば . 秒が得られます。



分針の角度です。

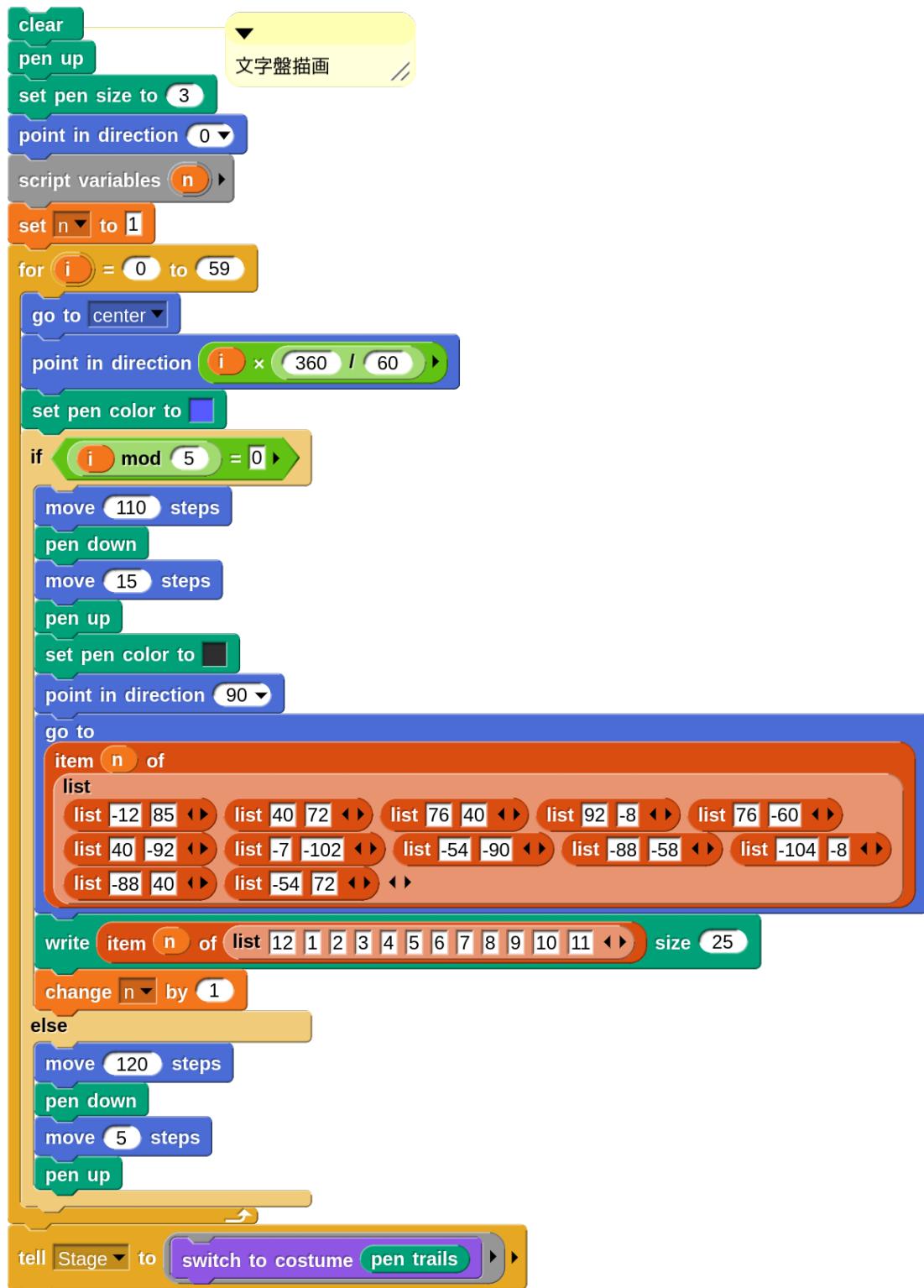


時針の角度です。 UTC 協定世界時を使用すると時差調整が必要になるので用いません。

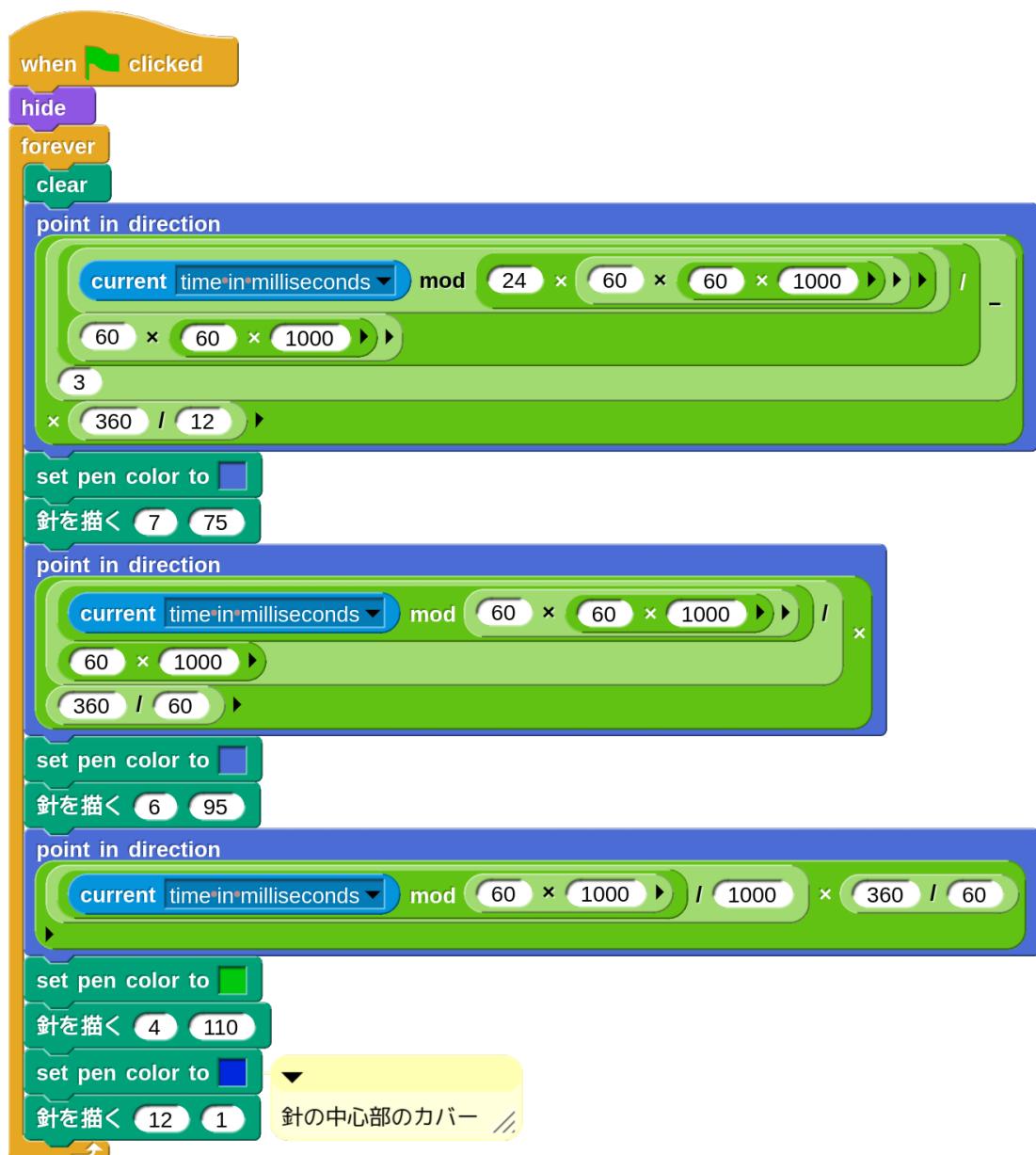
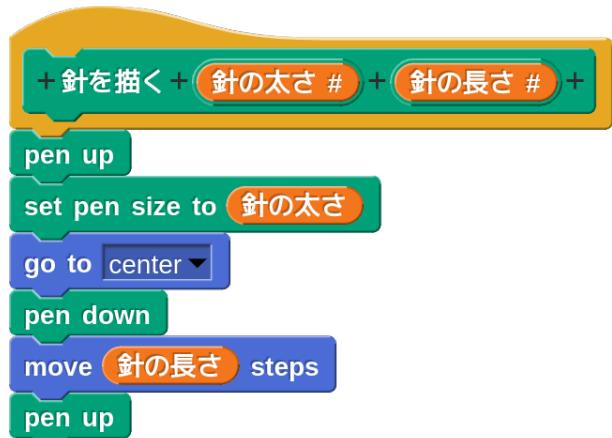


ペンで描く場合はこの逆の順序で描けば実際の時計と同じになります。

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。



## 9.9 並列処理について

Snap! では、各スプライトに **when green flag clicked** のスクリプトを設定すれば  をクリックすることでそれらの各スクリプトが同時に実行されるように見えます。しかし、実際にはそれ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 v] to [list >]** を実行してから  をクリックすると、左のようにリストに実行順序が表示されます。一ブロックずつ交互に実行されるのではなく、一方はもう一方のスクリプトの実行が終わるまで待たれます。実行の順番はスプライトが作成された順番になるようです。確実にあるスプライトのスクリプトを先に実行させたいならば、他のスプライトのスクリプトの先頭に **wait [0 secs]** を入れるなどして実行させる必要があります。

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



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

スプライト 1

```

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

```

スプライト 2

```

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

```

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

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

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

スプライト 1

```

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

```

スプライト 2

```

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

```

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

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

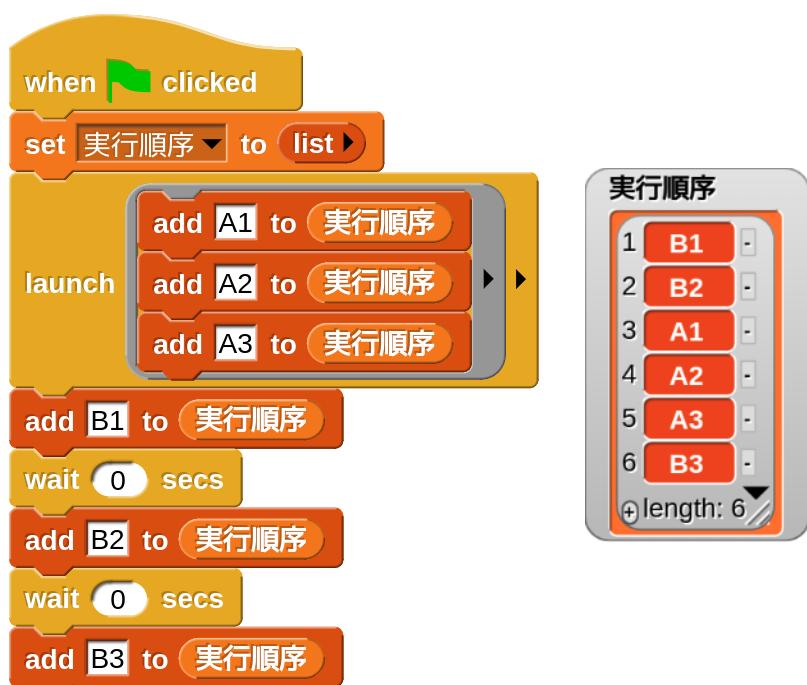
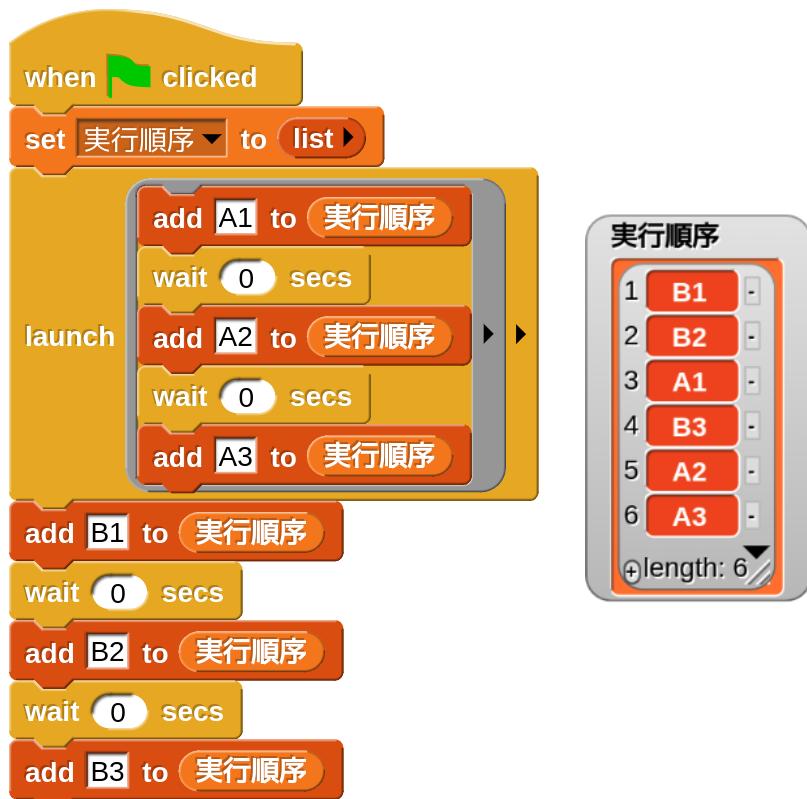
launch での動作です。



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

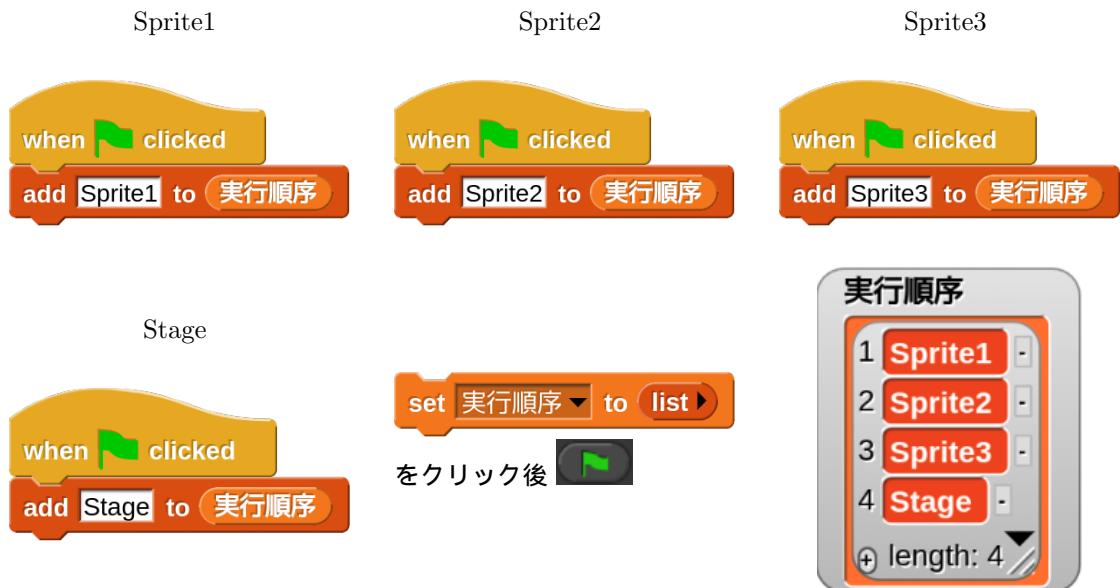


実行順序	
1	B1
2	B2
3	A1
4	B3
5	A2
6	A3
+ length: 6	



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

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



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



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

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

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

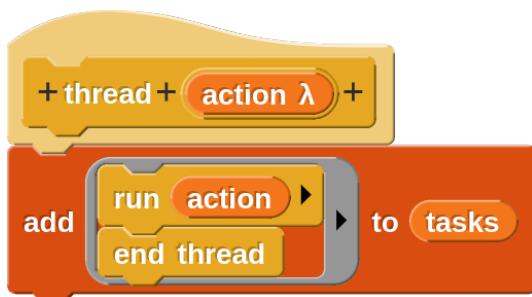
処理するスクリプトの記憶場所として **tasks** の変数を作成します。以下に示す 3 個の定義ブロックは後で例示するスクリプトで共通に使用します。

### thread

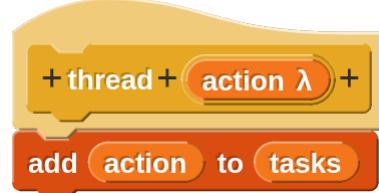
プロックです。スレッドを設定するプロックです。( 設定されたスレッドを tasks リストに加えていきます。)

このプロック自体はスレッドを記憶させるだけで、実行はしません。

action は Command(C-shape) 型です。



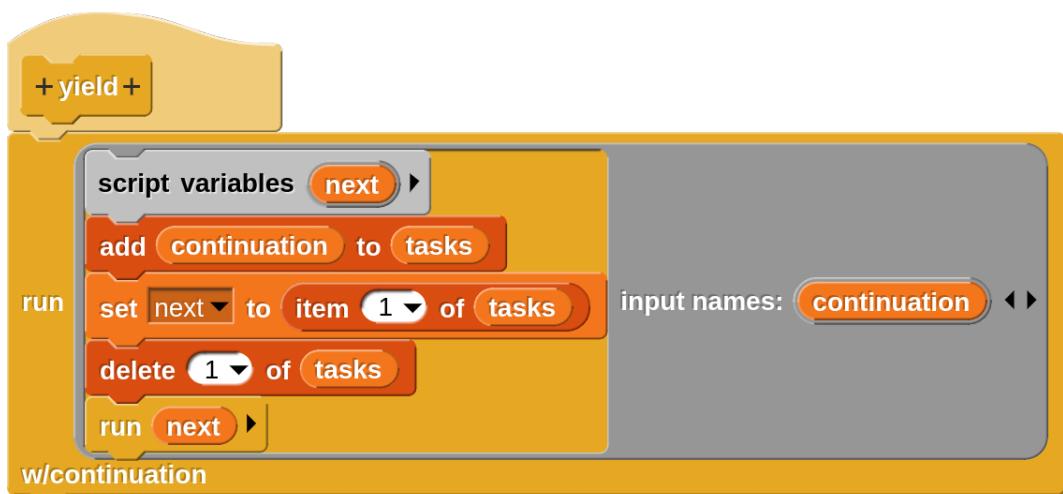
左はマニュアルに掲載されている定義ですが、後で例示するスクリプトの動作においては下の定義でも動くようです。( 非推奨 )



### yield

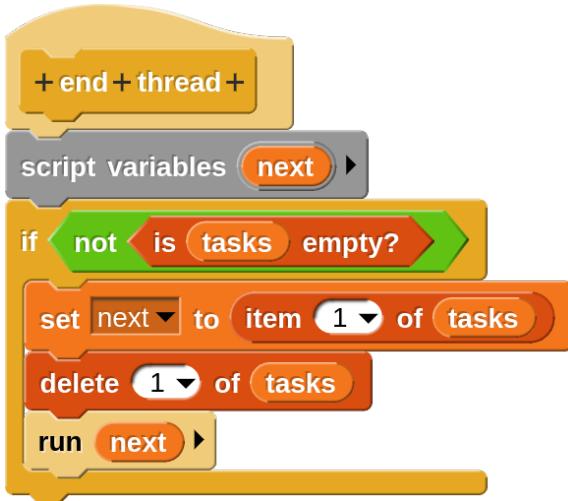
プロックです。( 継続を tasks リストに加え、tasks リストの先頭のスクリプトを取り出して実行します。)

次のスレッドの実行へ移行させます。



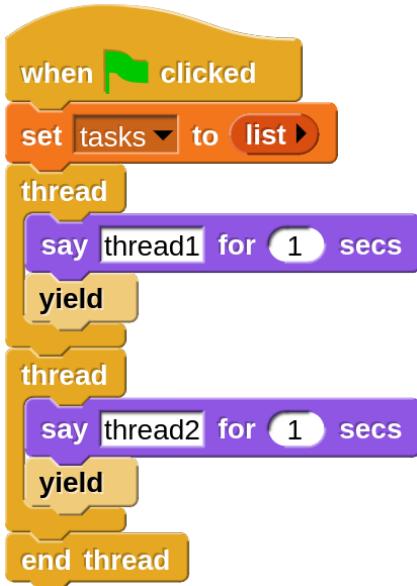
**end thread** ブロックです。( tasks リストが空でなかったら tasks リストの先頭のスクリプトを取り出して実行します。)

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



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

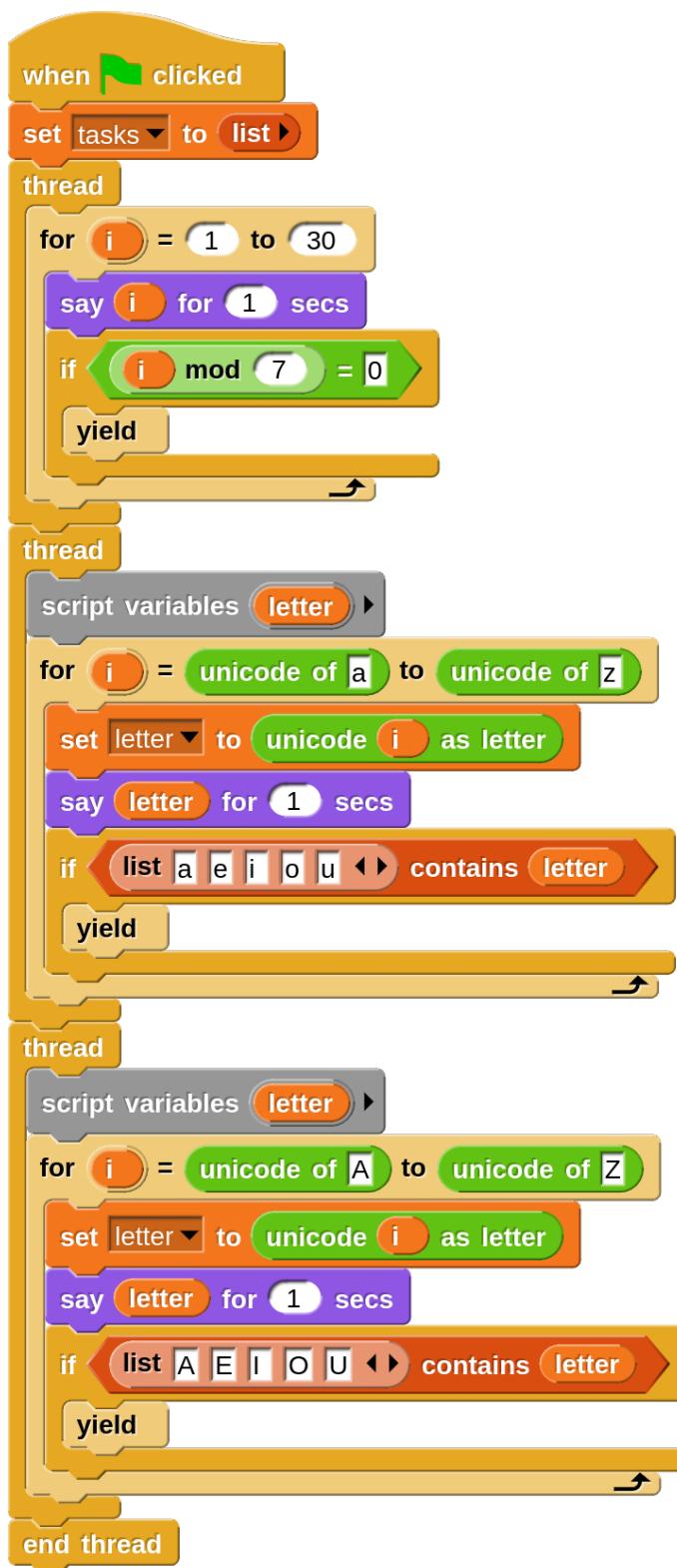
次のスクリプトはスレッド内に繰り返しがないのであまり意味はありませんが、使い方の例です。



番号を表示するだけの 3 個のスレッドを実行してみます。



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



## 10 再帰

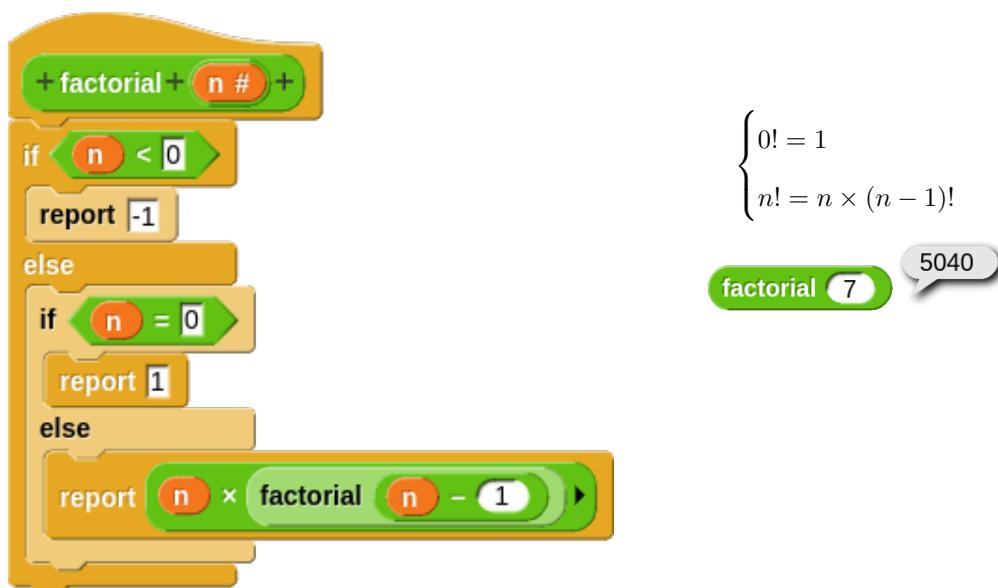
再帰、再帰呼出し（recursive call）は作成したブロックの中で自分自身を呼び出す（実行する）ものです。関数型プログラミングでは繰り返し処理の手法として再帰を使うことは一般的で、効率的だったりします。

### 10.1 再帰の例

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

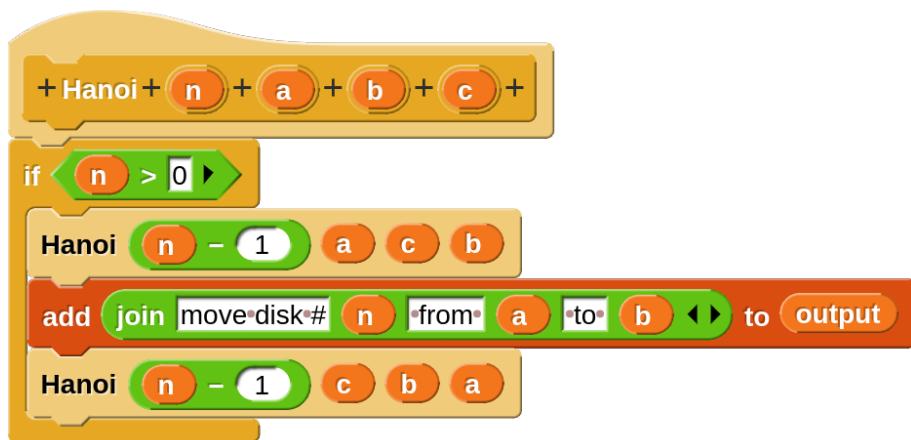
#### 10.1.1 階乗

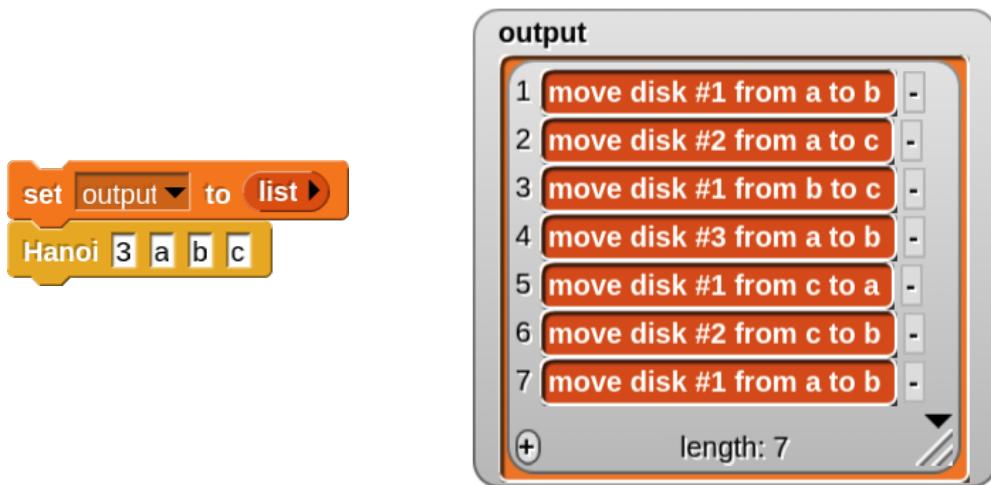
factorial 階乗は再帰の例としてよく使用されます。



#### 10.1.2 ハノイの塔

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

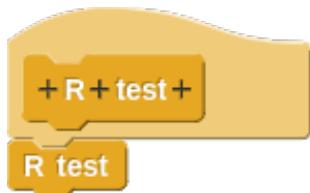




## 10.2 再帰の使用

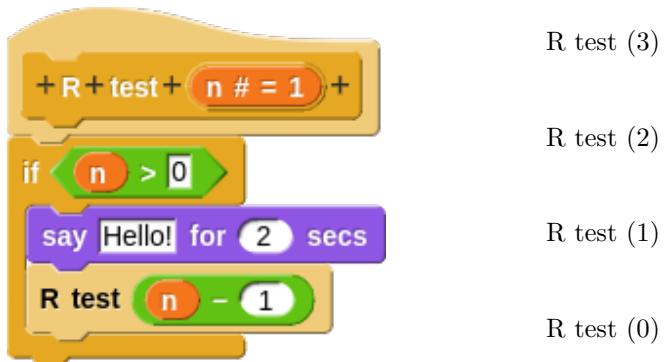
### 10.2.1 繰り返し

まずはただ自分自身を呼び出してみます。(実行しないでください)

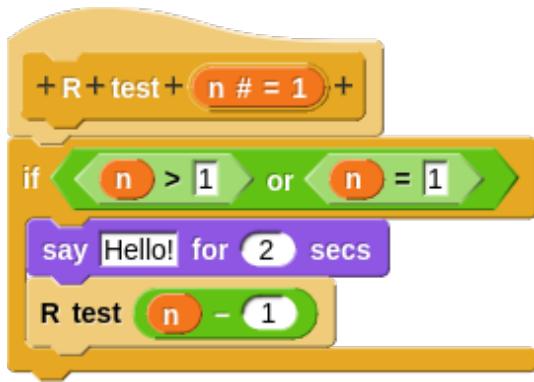


再帰呼び出しが終了するコードブロックがないので無限ループになります。普通の処理系ではスタックオーバーフローでエラー終了かフリーズします。

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



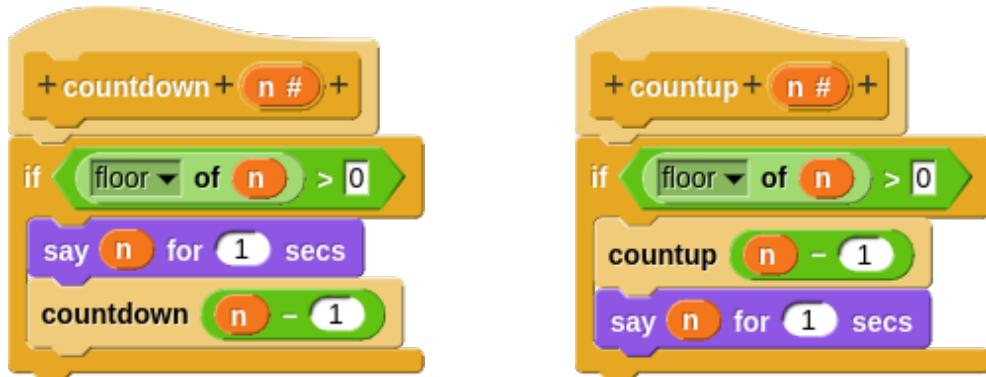
回数に 0.5 を指定しても実行されるので終了条件を変更します。



### 10.2.2 カウントダウンとカウントアップ

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

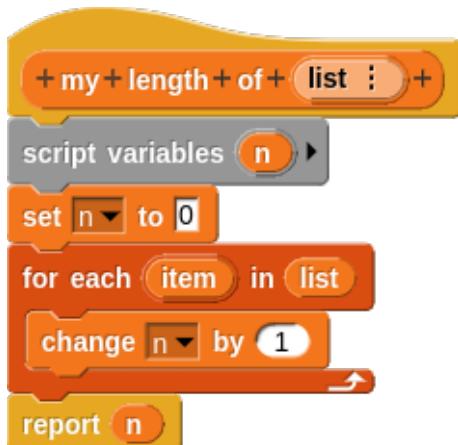
`say` ブロックのところに処理したいコードブロックを持ってくれば繰り返しの処理ができます。



### 10.2.3 my length

リストの要素数を求める `length` ブロックを作ってみます。

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



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

処理にかかる時間を表示します。

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

16.8

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

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

処理にかかる時間を表示します。

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

16.7

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



処理にかかる時間を表示します。



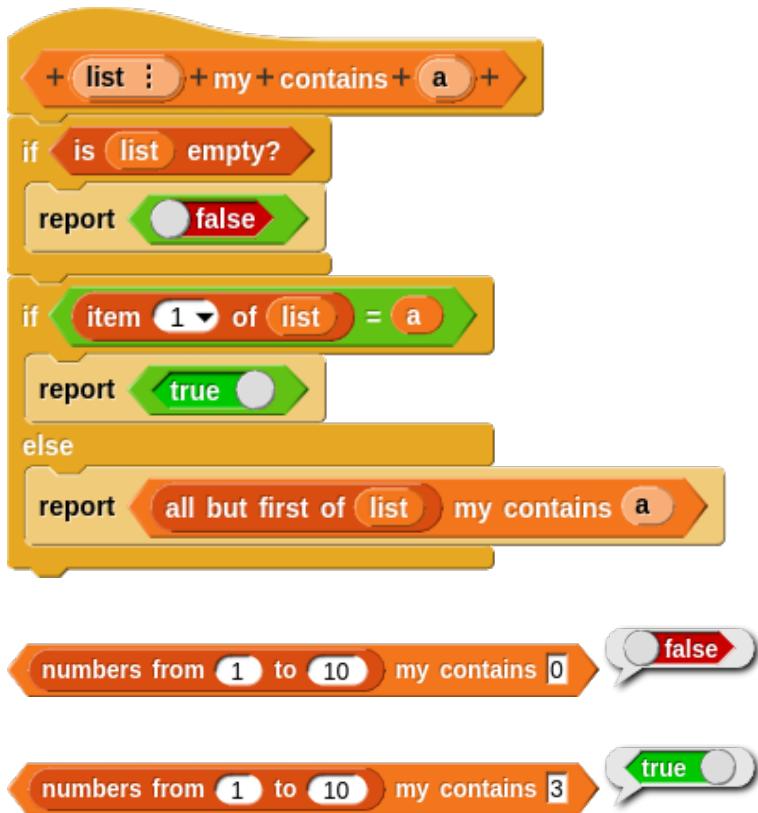
再帰呼び出しを使わないものに比べてとても効率が良いことがわかります。

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



#### 10.2.4 my contains

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



### 10.2.5 リスト要素の巡回

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



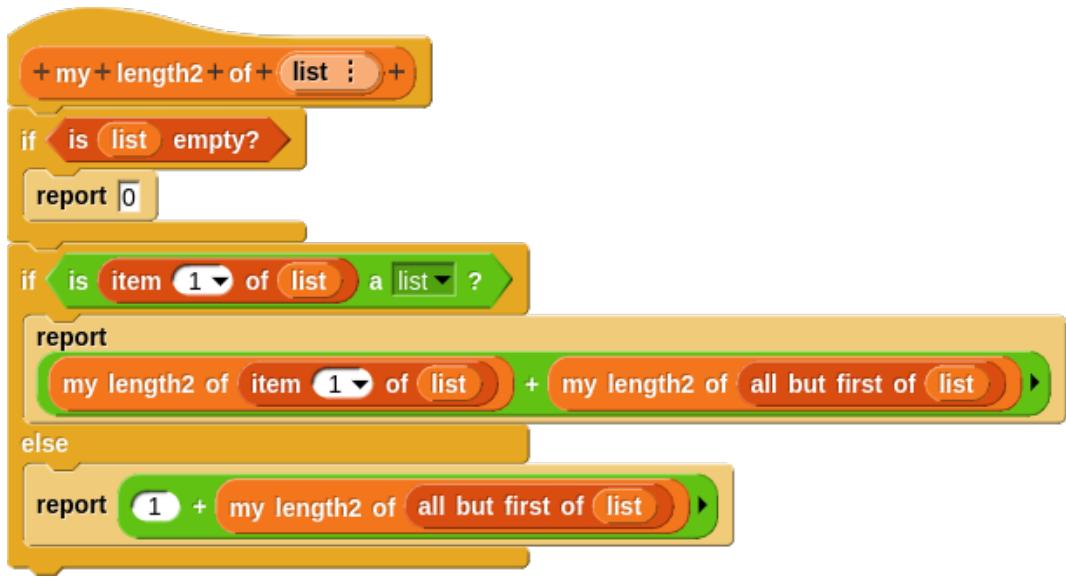
これは my length も同様です。



再帰を使って内部の要素に対してもアクセスしてみます。

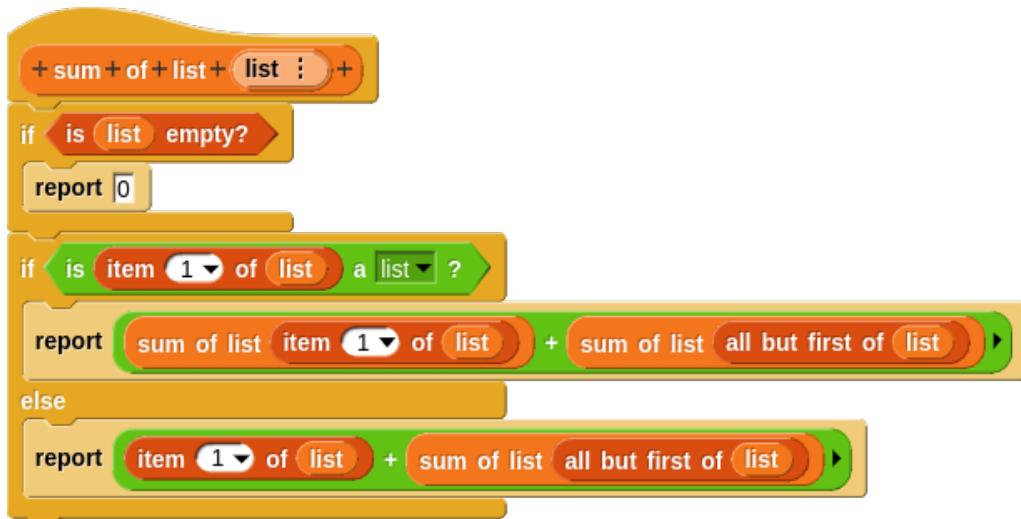
処理の内容は次のようにになります。

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



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

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



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

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

### 10.2.6 reverse 逆順リスト

リスト要素の逆順リストリポートを再帰呼出しで行うことができます。

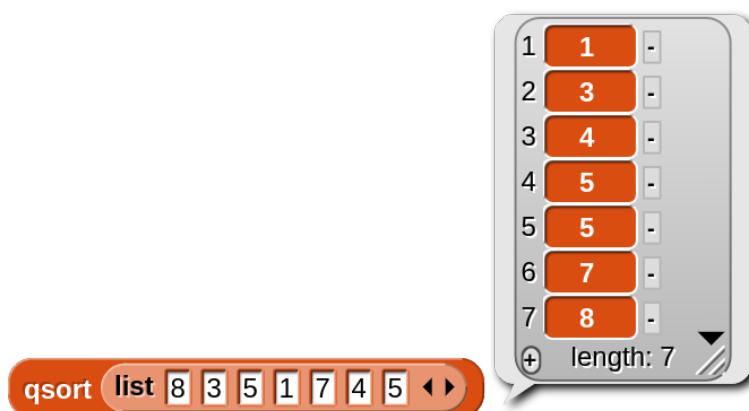
リストから先頭の要素を取り出して〔残りのリスト〕の最後尾に加えます。その〔残りのリスト〕に対して同じ操作をしていけば逆順のリストが得られます。

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



### 10.2.7 クイックソート（整列 / 並べ替え）

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



#### [ 参考文献 ]

『Programming in Haskell, 2nd edition』

Graham Hutton 著 山本和彦 訳 ラムダノート株式会社 刊

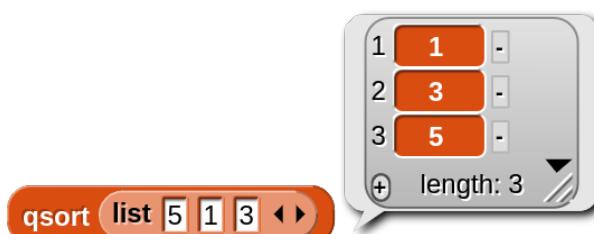
次のように作成したリストを通して値を表示すると操作の様子が見られます。



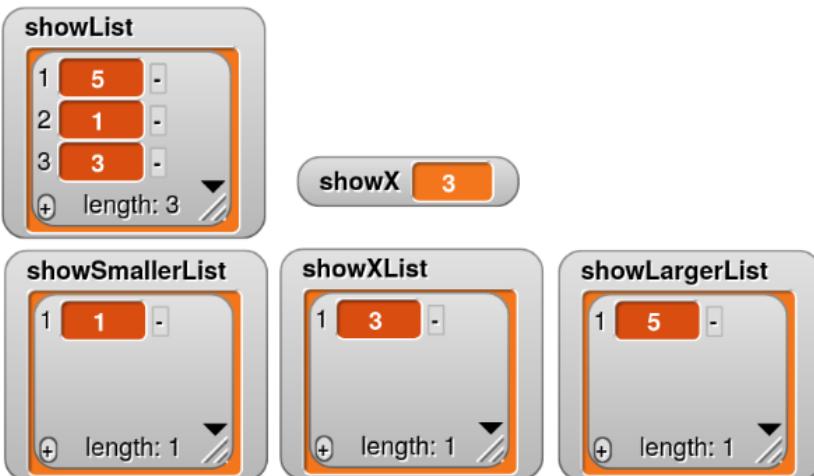
The Scratch script visualizes the quicksort algorithm. It starts by setting up variables: `x`, `xList`, `smaller`, and `larger`. It then picks a random item `x` from the list. It creates two new lists: `xList` containing items less than `x`, and `larger` containing items greater than `x`. Finally, it reports the results: `append qsort smaller xList qsort larger`.

```
+ qsort + [list : ] +  
set showList to [list]  
if is [list] empty?  
report [list]  
  
script variables  
x  
xList  
smaller  
larger  
  
set [x] to [item pick random 1 to [length] of [list] of [list]]  
set [xList] to [keep items < [x] from [list]]  
set [smaller] to [keep items < [x] from [list]]  
set [larger] to [keep items > [x] from [list]]  
set [showX] to [x]  
set [showXList] to [xList]  
set [showSmallerList] to [smaller]  
set [showLargerList] to [larger]  
pause all  
report [append qsort [smaller] xList qsort [larger]]
```

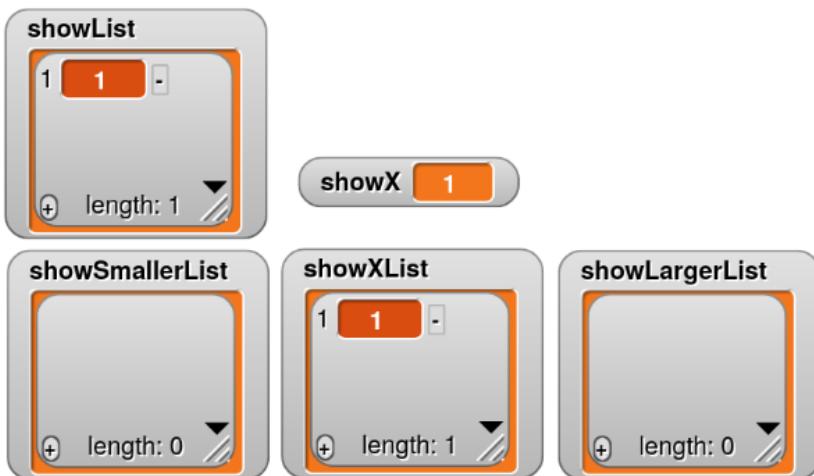
リストの値表示のたびに pause all になります。  をクリックして続行してください。



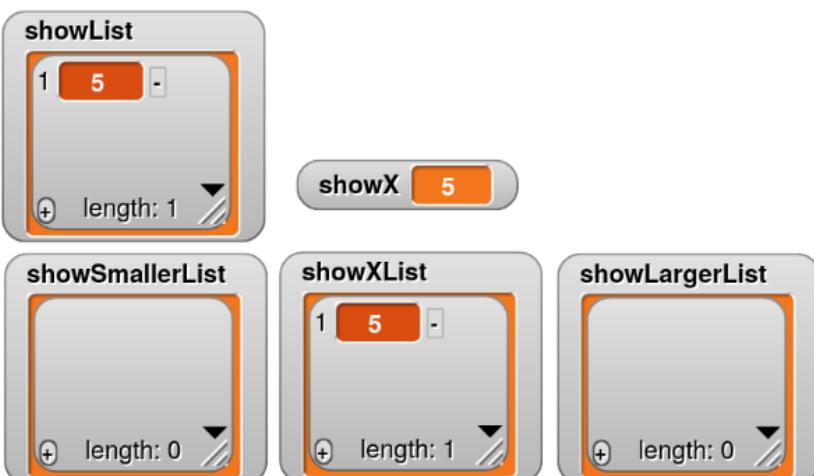
最初に選択された値が 3 だった場合です。



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

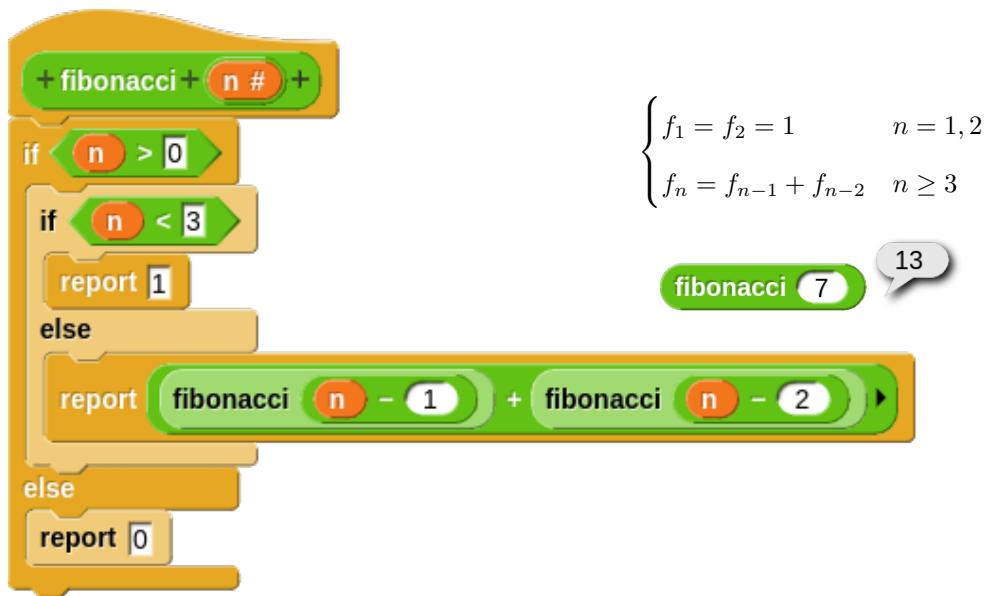


この場合要素数は 1 でしたが、それぞれのグループに対して再帰的に処理されます。

### 10.2.8 再帰呼び出しをする局所的な定義ブロック

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

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

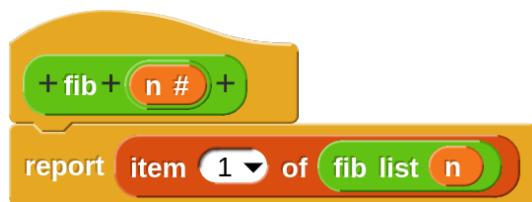


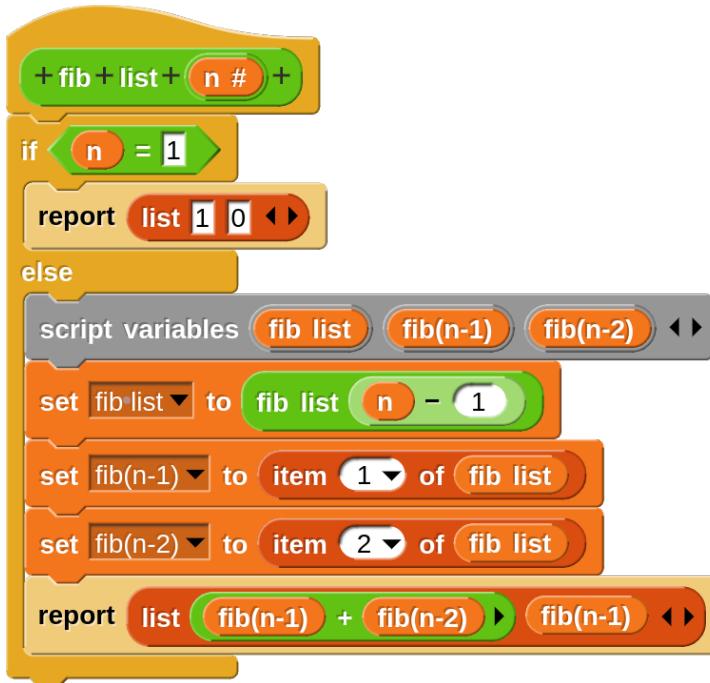
これは  $(n - 1)$  と  $(n - 2)$  を引数にして 2 度再帰呼び出しています。そのため、 $n$  が大きくなると時間がかかるてしまいます。

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

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

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

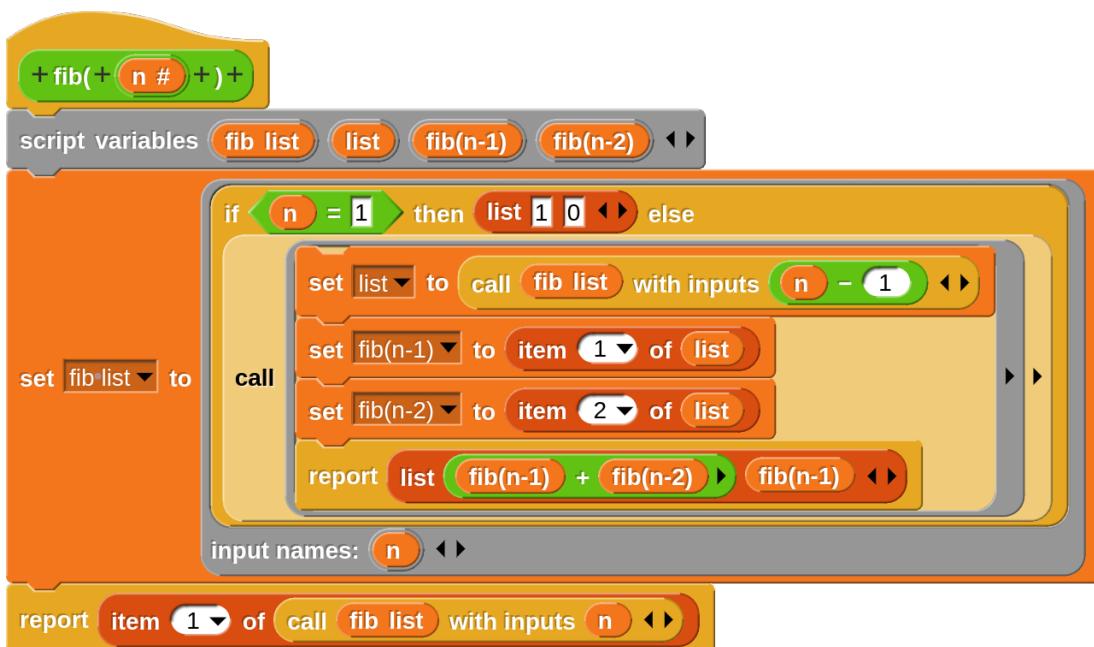




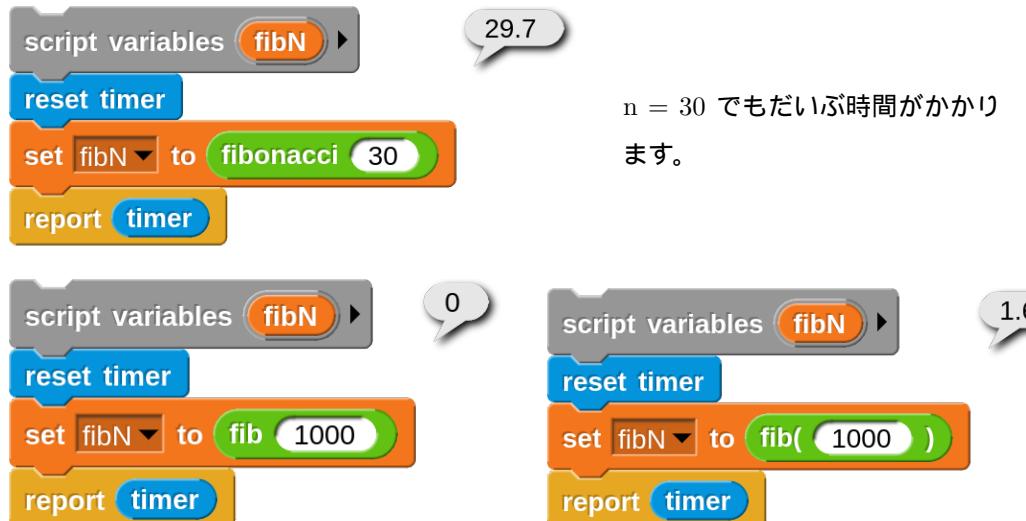
右の表は  $n$  の値に対する  $\text{fib}$  と  $\text{fib list}$  の値です。( は前の項のという意味です。)  $\text{fib}$  の値は、 $\text{fib list}$  がリポートする値であるリストの 1 番目の値になります。その 2 番目の値は  $\text{fib}(n - 1)$  の値です。これによって処理済みの  $\text{fib}$  値を渡していきます。

$n$	$\text{fib}$	$\text{fib list}$
1	1	list(1, 0)
2	1	list(1, 1) ( 1 + 0, 1)
3	2	list(2, 1) ( 1 + 1, 1)
4	3	list(3, 2) ( 2 + 1, 2)
5	5	list(5, 3) ( 3 + 2, 3)
6	8	list(8, 5) ( 5 + 3, 5)

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



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

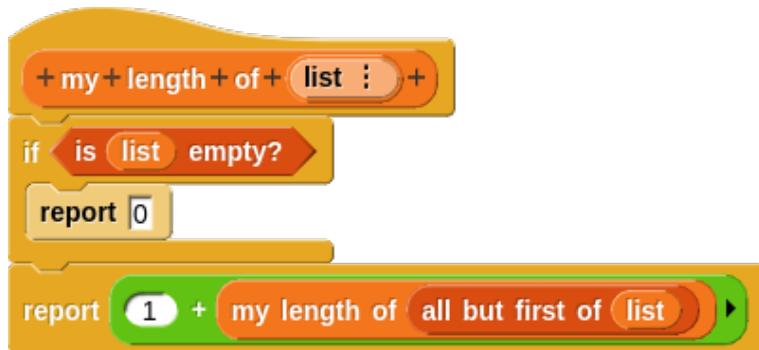


#### [ 参考文献 ]

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

### 10.2.9 末尾再帰

113 ページで my length を扱いました。



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

```

L(1, 2, 3)
[
  1 + L(2, 3)
    1 + L(3)
      1 + L()
        1 + 0
          1 + 1
            1 + 2
              3
]

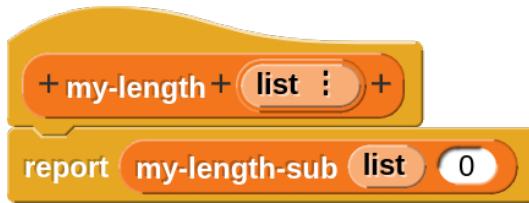
```

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



引数  $(n + 1)$  を渡していくことでカウントしています。

これを呼び出す本体定義は次のようにになります。



引数 n の値を 0 にすることで、カウンターの値を初期化します。



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

[

`L(2, 3)(1)`

`L(3)(2)`

`L()(3)`

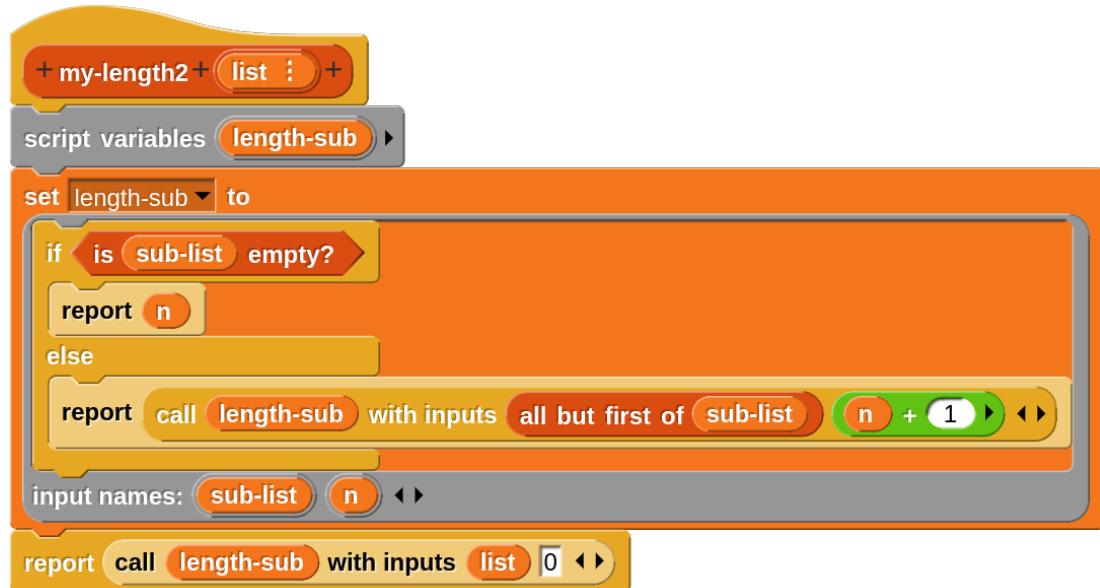
`3`

]

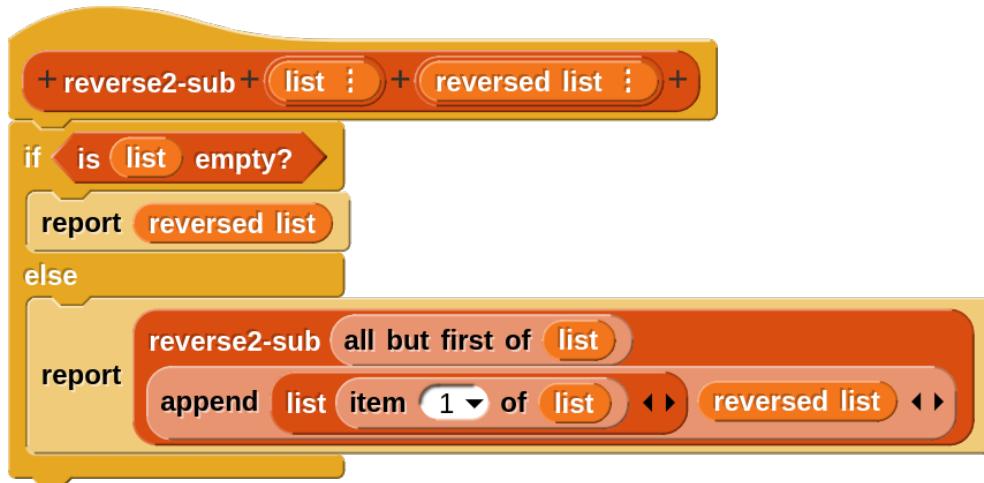
このように、再帰呼出しの結果がそのまま定義ブロックの値になる形を末尾再帰といいます。

Scheme では、末尾再帰はループに最適化されるので効率がよくなります。

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



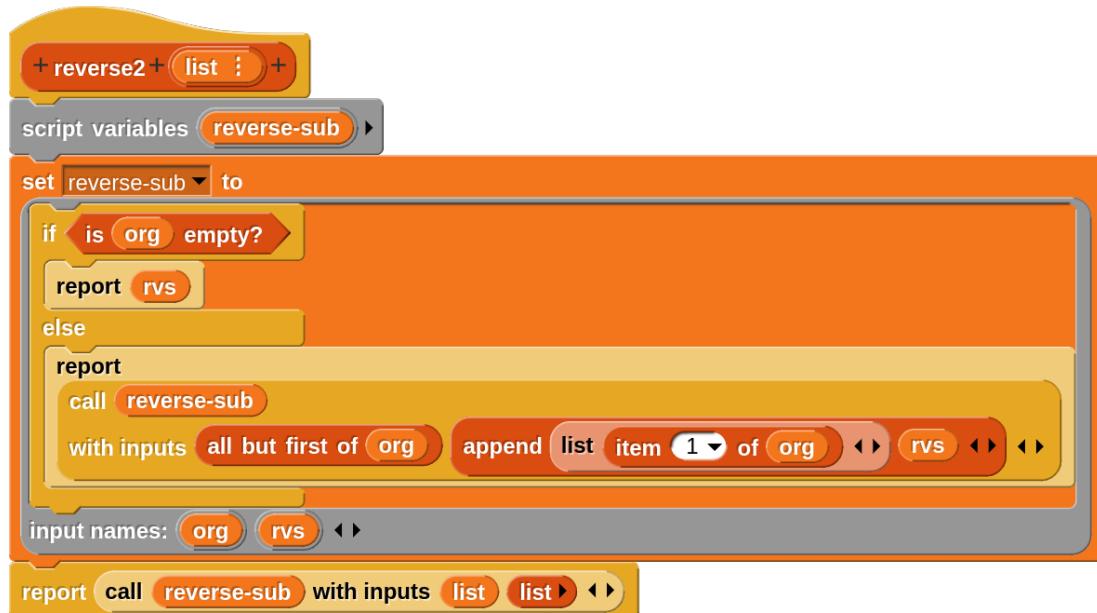
reverse の末尾再帰版です。



これを呼び出す本体定義は次のようにになります。



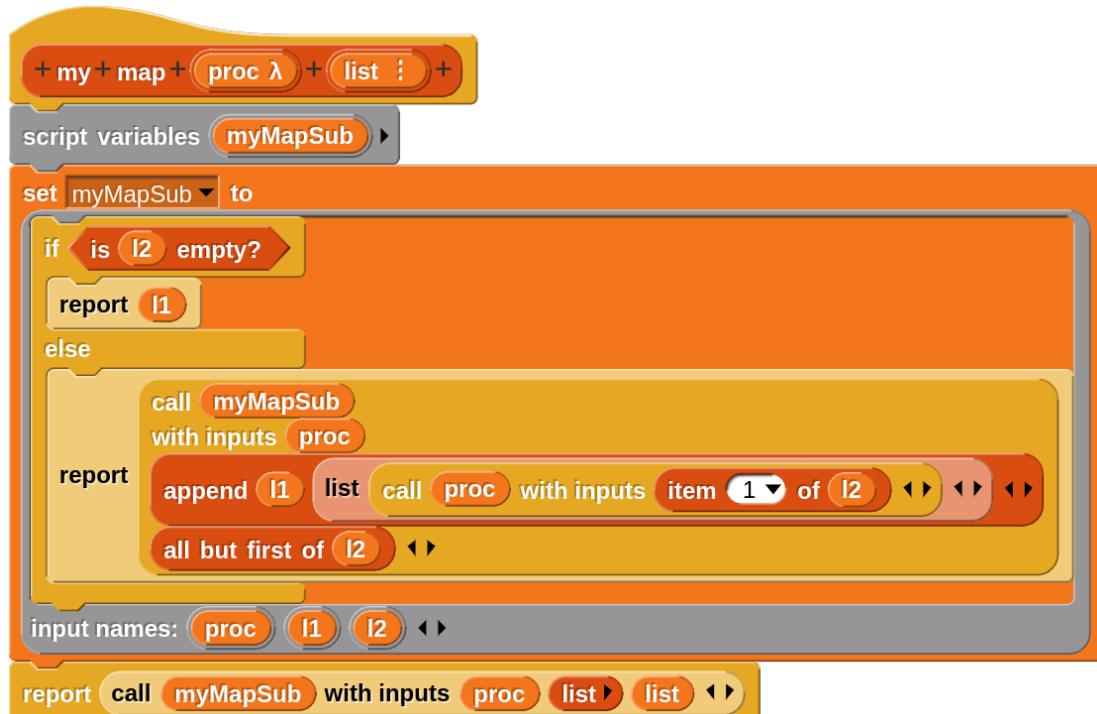
局所定義ブロック版です。



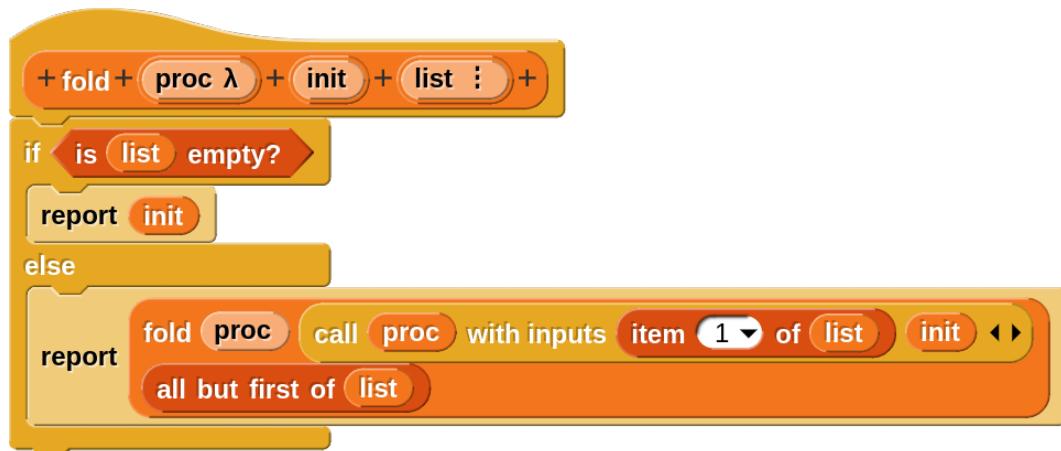
## 11 高階関数

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

map ブロックを作成してみます。



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



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

 の場合、

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

 となります。

fold は有益なもので、これで map や reverse などを作成することができます。



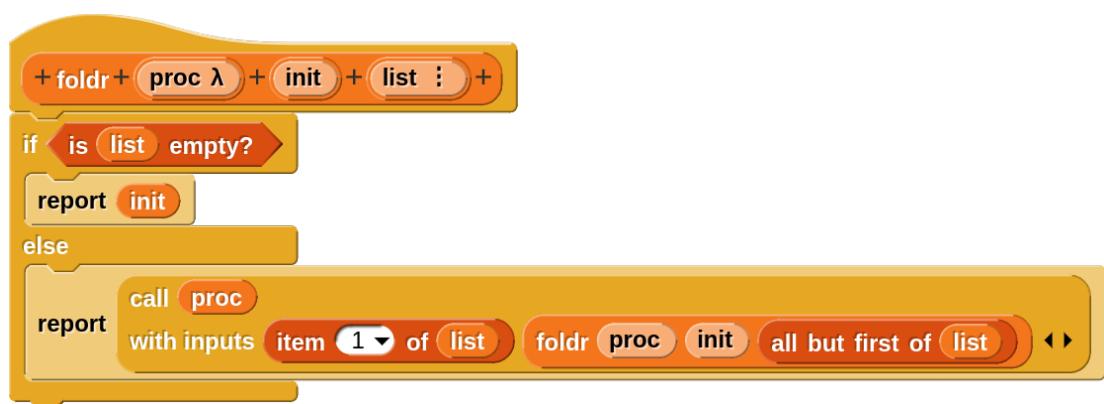
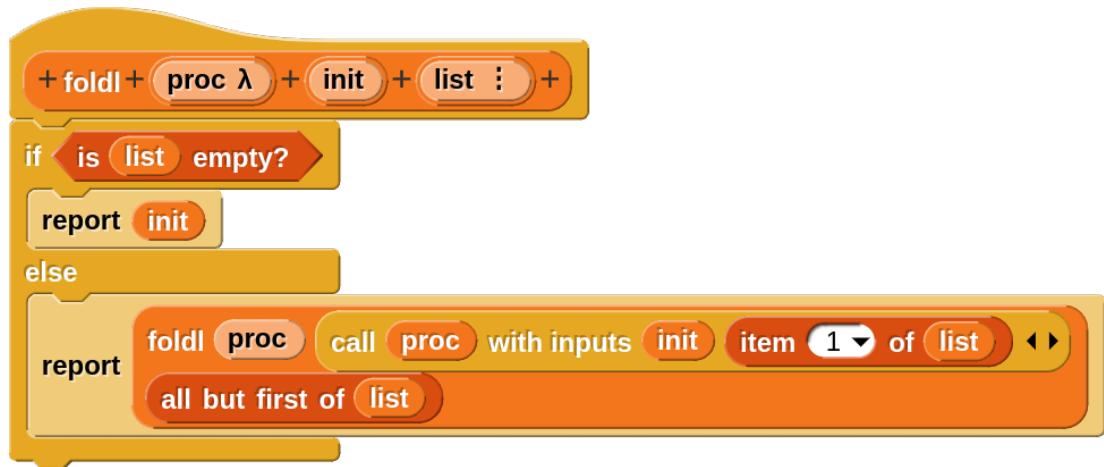
fold を使わないものに比べてスッキリしています。また、fold を利用すれば自動的に再帰呼出しを使った処理になるという利点があります。

参考までに、Haskell の foldl と foldr も示しておきます。Gauche の fold とは操作ブロックの入力スロットに対して init と要素の入る位置が違っているため結果が違う場合があります。

$n_1 + n_2$  と  $n_2 + n_1$  は結果が同じですが、 $n_1 - n_2$  と  $n_2 - n_1$  は違います。

append l1 l2 と append l2 l1 は違います。

fold	foldl	foldr																																																																											
																																																																													
<table border="1" style="border-collapse: collapse; width: 100%;"> <thead> <tr> <th>init</th> <th>n1</th> <th>-</th> <th>n2</th> <th>値</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>1</td> <td>-</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>2</td> <td>-</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>3</td> <td>-</td> <td>1</td> <td>2</td> </tr> <tr> <td>2</td> <td>4</td> <td>-</td> <td>2</td> <td>2</td> </tr> </tbody> </table>	init	n1	-	n2	値	0	1	-	0	1	1	2	-	1	1	1	3	-	1	2	2	4	-	2	2	<table border="1" style="border-collapse: collapse; width: 100%;"> <thead> <tr> <th>init</th> <th>n1</th> <th>-</th> <th>n2</th> <th>値</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>-</td> <td>1</td> <td>-1</td> </tr> <tr> <td>-1</td> <td>2</td> <td>-</td> <td>2</td> <td>-3</td> </tr> <tr> <td>-3</td> <td>3</td> <td>-</td> <td>3</td> <td>-6</td> </tr> <tr> <td>-6</td> <td>4</td> <td>-</td> <td>4</td> <td>-10</td> </tr> </tbody> </table>	init	n1	-	n2	値	0	0	-	1	-1	-1	2	-	2	-3	-3	3	-	3	-6	-6	4	-	4	-10	<table border="1" style="border-collapse: collapse; width: 100%;"> <thead> <tr> <th>init</th> <th>n1</th> <th>-</th> <th>n2</th> <th>値</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>4</td> <td>-</td> <td>0</td> <td>4</td> </tr> <tr> <td>4</td> <td>3</td> <td>-</td> <td>4</td> <td>-1</td> </tr> <tr> <td>-1</td> <td>2</td> <td>-</td> <td>-1</td> <td>3</td> </tr> <tr> <td>3</td> <td>1</td> <td>-</td> <td>3</td> <td>-2</td> </tr> </tbody> </table>	init	n1	-	n2	値	0	4	-	0	4	4	3	-	4	-1	-1	2	-	-1	3	3	1	-	3	-2
init	n1	-	n2	値																																																																									
0	1	-	0	1																																																																									
1	2	-	1	1																																																																									
1	3	-	1	2																																																																									
2	4	-	2	2																																																																									
init	n1	-	n2	値																																																																									
0	0	-	1	-1																																																																									
-1	2	-	2	-3																																																																									
-3	3	-	3	-6																																																																									
-6	4	-	4	-10																																																																									
init	n1	-	n2	値																																																																									
0	4	-	0	4																																																																									
4	3	-	4	-1																																																																									
-1	2	-	-1	3																																																																									
3	1	-	3	-2																																																																									



1	3
2	2
3	1
+	length: 3

```
foldr append subList list member < > < > input names: member subList < >
list > list 1 2 3 < >
```

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

まずは基本となる 2 つの引数を取り、足し算の式として表示するものです。

```
+ [n1 #] + 足し算 [n2 #] +
report join [n1 + n2 =] [3 足し算 5]
```

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

```
+ [n1 #] + 引き算 [n2 #] +
if [n1 > n2] then
  report join [n1 - n2 =] [8 引き算 4]
else
  report join [n2 - n1 =] [8 引き算 4]

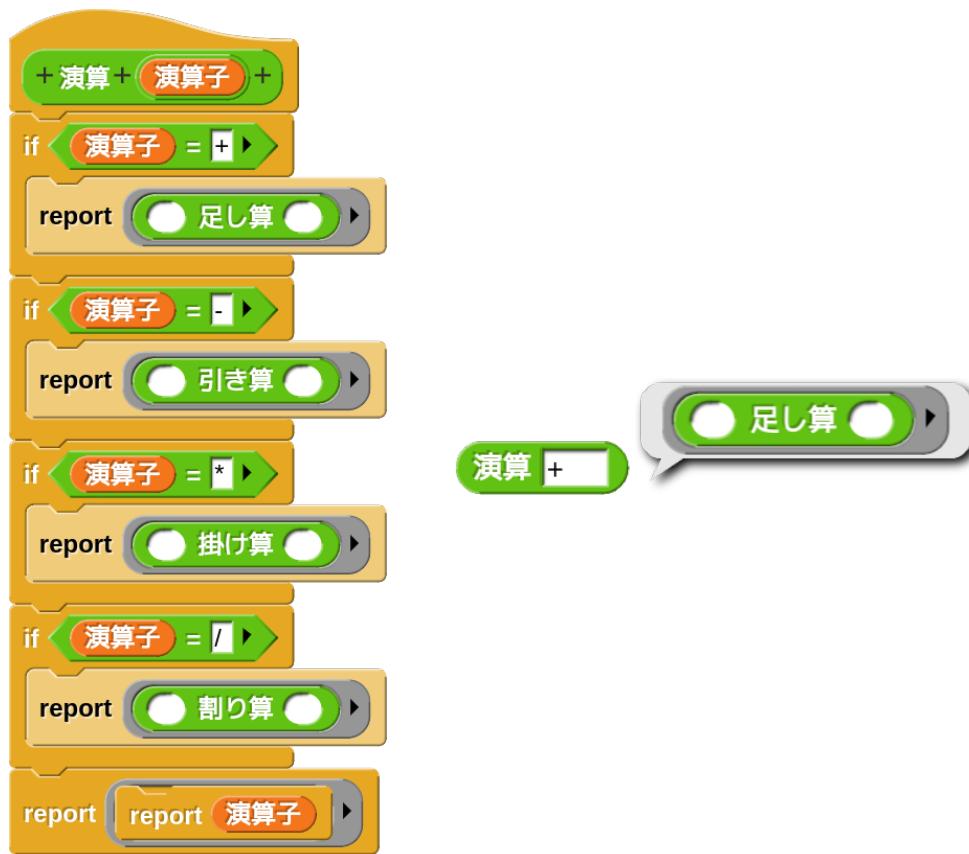
+ [n1 #] + 掛け算 [n2 #] +
report join [n1 × n2 =] [3 掛け算 6]

+ [n1 #] + 割り算 [n2 #] +
report join [n1 ÷ n2 =] [8 割り算 2]
```

```
足し算 [ ] を ringify してリポートすると、
report 足し算 [ ]
```

のようにスクリプトブロックをリポートすることができます。

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



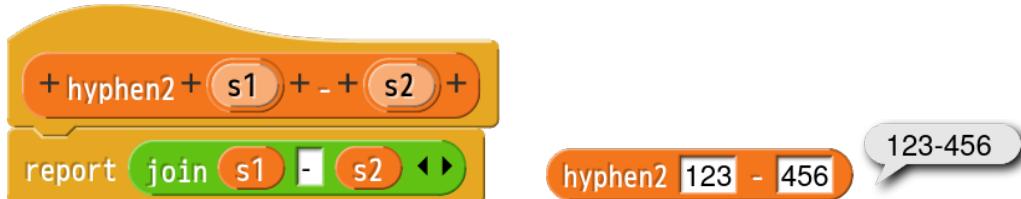
それを呼び出す本体のブロックです。call の入力スロットに **演算 演算子** を入れると ringify **演算 演算子** の状態になります。**演算 演算子** のブロックは ringify されたものを返すものなので、それをまた ringify してしまうと call で実行しても ringify されたものを表示するだけになります。演算のところを右クリックして unringify してください。



## 11.1 カリー化

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

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



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



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



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



カリー化したものをユーザー定義ブロックで使用してみます。

例えば、東京都の電話番号に市外局番 03 を付ける定義ブロックです。



## 11.2 OOP オブジェクト指向プログラミング

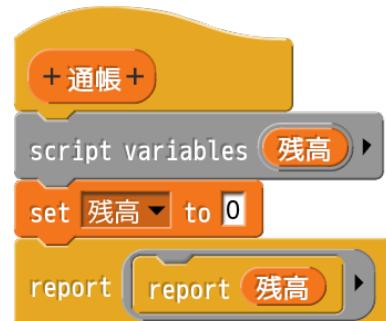
オブジェクト指向プログラミングでは、Class という型を作り、変数や操作（メソッド）を設定します。それを基に作られた変数などの実態にメッセージという形で指令を送って結果を得ます。Snap! には Class のような表立ってオブジェクト指向プログラミングの仕組みはありませんが、高階関数を使うことで同じようなことができます。

預貯金通帳を OOP 的に作ってみます。

預貯金額の変数や通帳で行う操作のスクリプトを通帳自身に持たせます。

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

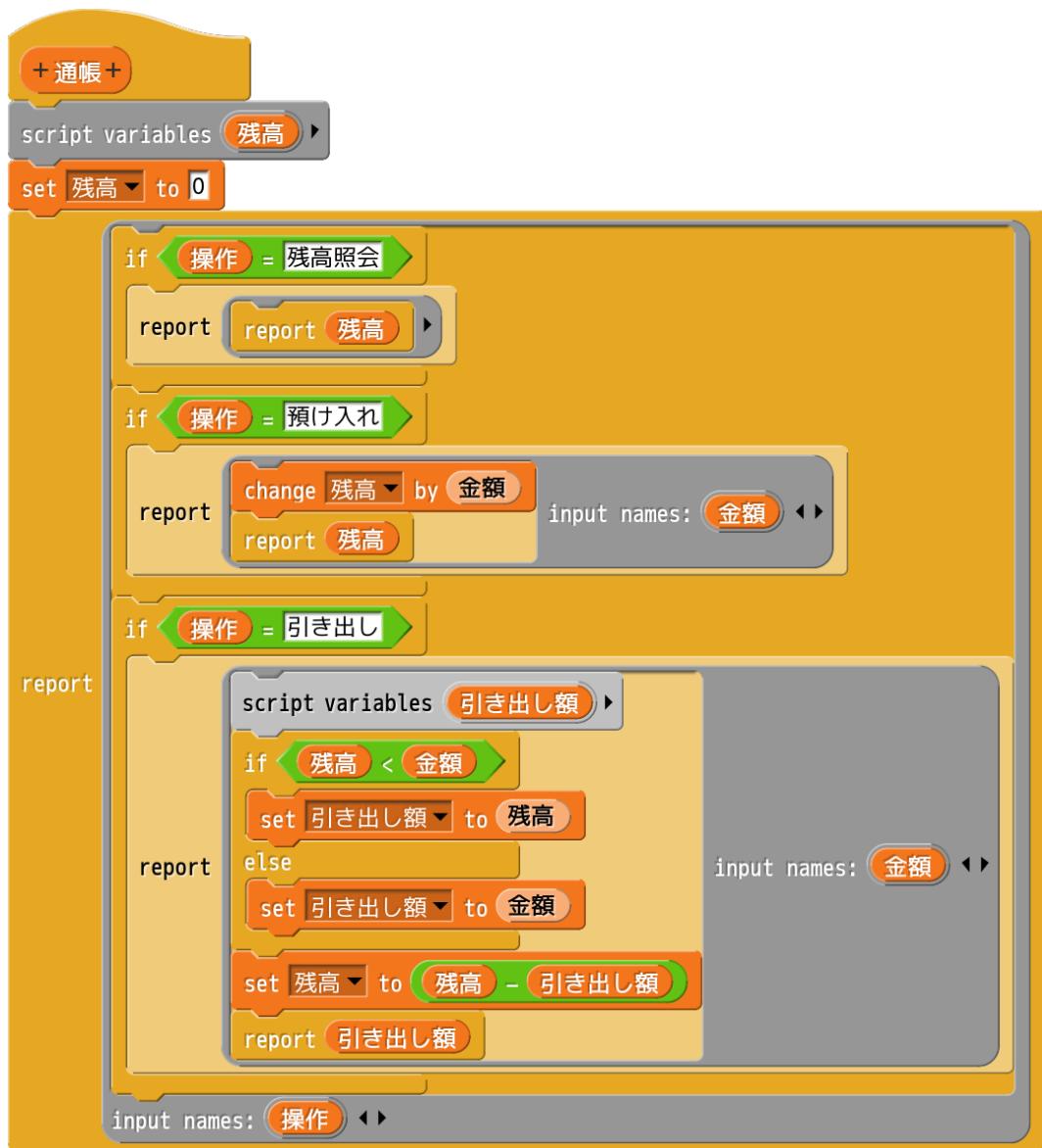
この型を基に以下のようにして通帳を作ります。



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



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



定義を変更したので set ブロックの実行が必要です。

set 太郎の通帳 to 通帳

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

call call 太郎の通帳 with inputs 残高照会 0

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

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

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

使いやすいように ATM ブロックを定義します。「操作」はマウスで選択できるように設定します。(61 ページ参照) 「通帳名」は Command(Inline)、「操作」は Text にしてあります。実は、「通帳名」は「通帳」型の変数を受け取れればいいので Any type でも大丈夫みたいです。

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

ATM 太郎の通帳 残高照会 ▾ 0

ATM 太郎の通帳 預け入れ ▾ 10000

ATM 太郎の通帳 引き出し ▾ 3000

ATM 太郎の通帳 残高照会 ▾ 7000

set 花子の通帳 ▾ to 通帳

ATM 花子の通帳 残高照会 ▾ 0

ATM 花子の通帳 預け入れ ▾ 10000

ATM 花子の通帳 引き出し ▾ 50000

ATM 花子の通帳 残高照会 ▾ 0

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

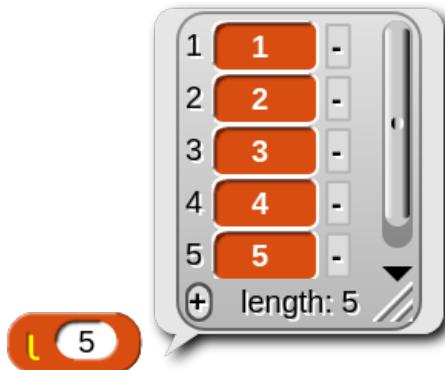
## 12 APL ライブライバー

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

### 12.1 形

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

ベクトルを生成するのに、



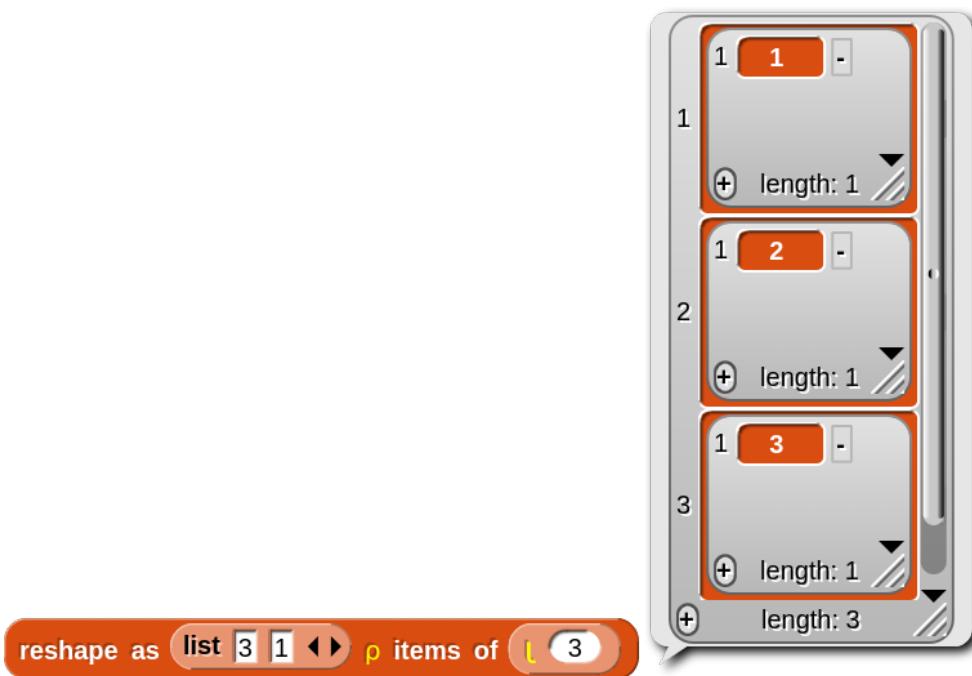
とすれば、1 から指定の数値までのベクトルが得られます。ちなみにこの記号のようなものはギリシャ文字の  $\iota$  イオタです。

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



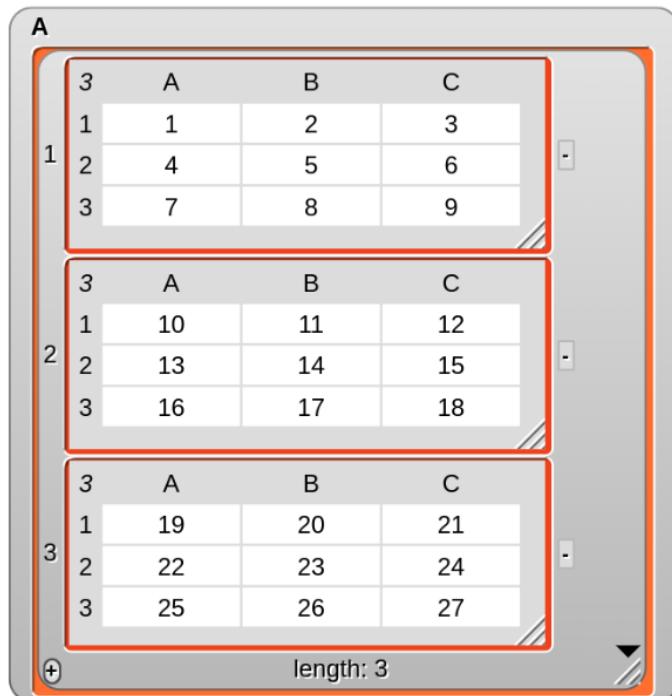
形はリストを使って指定します。次のように 1 行 3 列や、





のように3行1列にすることもできます。

`reshape`のリストを次のようにすると、



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

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

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

shape of p 目 は指定されたものの形をリストで返します。

スカラーの形 shape は空です。

shape of p [1]

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

shape of p [l 5]

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

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

1	3	-	
2	3	-	
[+]	length: 2		

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

1	3	-	
2	3	-	
3	3	-	
[+]	length: 3		

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

rank of pp [1] 0

rank of pp [l 5] 1

rank of pp reshape as list [3 3 ⏪ p items of l 3] 2

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

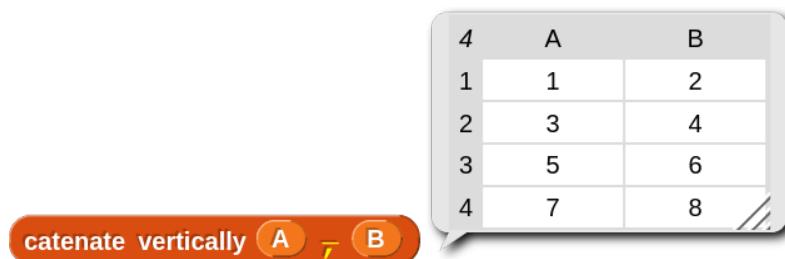
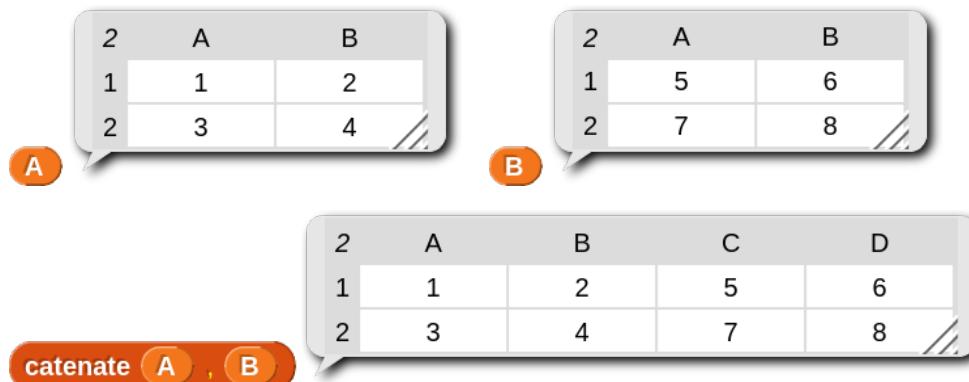
rank を 1 のベクトルにするブロックもあります。

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

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

## 12.2 配列の連結

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



連結される箇所の双方の要素数が同じである必要があります。

## 12.3 配列要素の配置転換

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

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

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

A

- 記号が示しているように上下方向、垂直方向に入れ替えます。

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

reverse row order (column contents) ⊖ A

- 記号が示しているように左右方向、水平方向に入れ替えます。

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

reverse column order (row contents) ⚡ A

- 記号が示しているように対角線を軸にして入れ替えます。( 転置 )

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

transpose ◎ A

## 12.4 ベクトル、配列の範囲指定、選択

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

1	1	-	▼
2	2	-	
3	3	-	
⊕	length: 3		

take 3 ↑ from [ 5 ]

1	3	-	▼
2	4	-	
3	5	-	
⊕	length: 3		

take -3 ↑ from [ 5 ]

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

drop 3 from [5 v]

drop -3 from [5 v]

配列が指定された場合は、行単位の指定になるようです。

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

take 2 from [A v]

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

list [1 2 3 4 5 v]

select rows (compress columns)

list [1 0 0 1 0 v]

map mod 2 over [5 v]

これを使用すると奇数番の要素を選択できます。

select rows (compress columns)

map mod 2 over [5 v]

list 5

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

1	A	B
1	2	4

**select columns (compress rows) map**

map

mod

1	A	B
1	2	4

4

reshape as list 1 5 ← → p items of

ems of

2 = 0 ➤ ov

**ver** l

5

1

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

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

A

	A	B	C	D
1	1	2	3	4
2	9	10	11	12

### select rows (compress columns)

list 1 0 1 ◀ ▶ / A

3	A	B
1	2	4
2	6	8
3	10	12

### select columns (compress rows)

## 12.5 配列要素に対する演算

APLでは、配列に対してスカラーの演算することができます。

reshape as list [3 3] ⏪ p items of l [3] + 10 ⏪

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

**combine**  **using**  の配列版が用意されています。

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

A

combine in rows (reduce by column vectors)  / 目

配列の 1 行内の各列の要素に対して指定された演算をします。

1	6	.
2	15	.
3	24	.
+	length: 3	▼

combine in rows (reduce by column vectors)  / A

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

この場合の演算子は + なので、1 行内の各列要素の合計になります。

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

combine in columns (reduce by row vectors)  ✖ 目

配列の 1 列内の各行の要素に対して指定された演算をします。

1	A	B	C
1	12	15	18

combine in columns (reduce by row vectors)  ✖ A

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

## 12.6 outer product

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

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

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

計算方法を表示させてみます。

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

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

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

単位行列なども作成できます。

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

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

0, 1 で表すには、

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

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

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

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

```

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

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

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

```

set [N] to [5]
    
```

	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

```

outer product [N] o. [0 = mod] [N]
    
```

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

次の処理で N の値に対する各行の true の合計を求めます。

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

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

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

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

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

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

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

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

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

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

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

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

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

1	2
2	3
3	5
+	length: 3

### [ 参考文献 ]

『基礎からの APL 解説と例題例解』

西川利男/日本アイビー・エム 共著 サイエンスハウス 刊

## 12.7 inner product

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

The screenshot shows APL code for setting up two 2x2 matrices  $A$  and  $B$ :

```
set A to reshape as list 2 2 ⌈ p items of list a b c d
set B to reshape as list 2 2 ⌈ p items of list e f g h
```

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

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

```
inner product A join ? join ?? B
```

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

## inner product A

join  +  <  >

join  x  B

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

## inner product

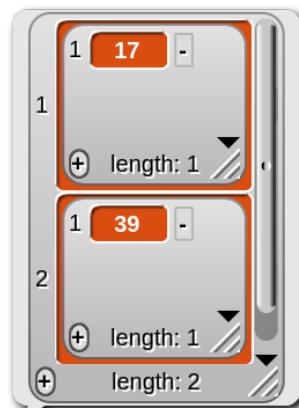
プロセスで求めることによって本末

2行2列配列同士では次のような計算方法になります

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

左側の配列の列数と右側の配列の行数は同じ必要があります。

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



inner product

list list 1 2 ◀ ▶

list 3 4 < > << >>

10

list 5 list 6

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

The Scratch script shows an 'inner product' operation. It has two lists: list A [1, 2, 3, 4] and list B [90, 100]. The script uses a green '+' block followed by a grey 'x' block to calculate the sum of the products of corresponding elements from both lists.

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  $\theta$  度回転させた座標を求めるものです。

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

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

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

The Scratch script sets  $\theta$  to 90 and creates a list R containing the elements of the rotation matrix. The matrix is defined as:

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} = \begin{bmatrix} \cos 90 & -\sin 90 \\ \sin 90 & \cos 90 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

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

The variable R contains the following list of lists:

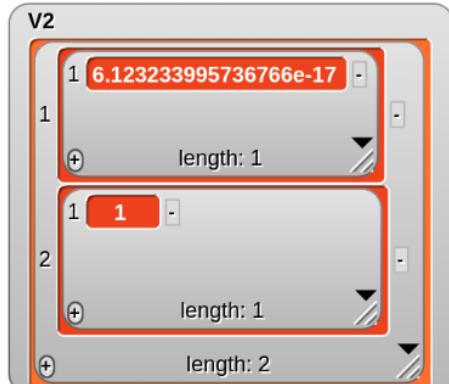
$$R = \begin{bmatrix} \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \end{bmatrix}$$

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

```

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

```



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

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

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

```

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

```

```

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

```



これを実行すると、 が表示されます。

```

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

```



これを実行すると、 回転後の図形が追加表示されます。

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

```

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

```

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

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

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

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

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

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

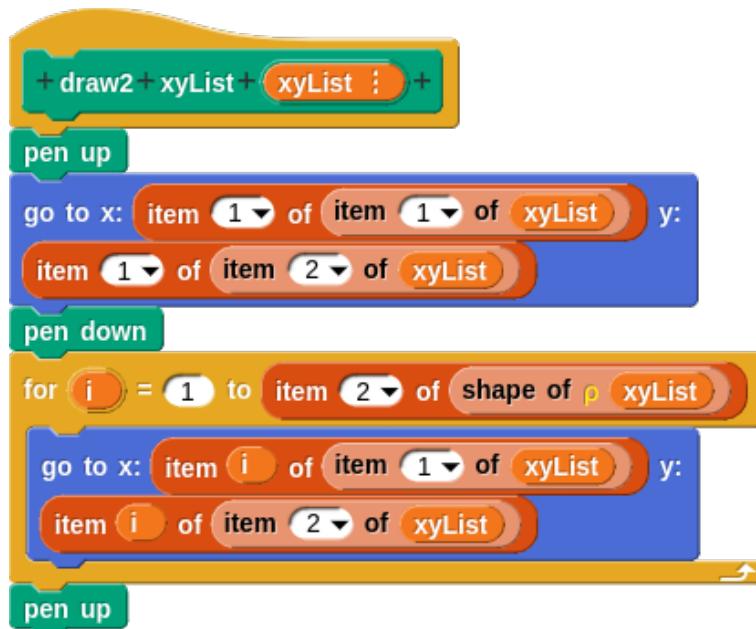
  


transpose を使用すると、前と同じ座標リスト指定から変換することができます。

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


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



変換前の図形描画スクリプトです。

```

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

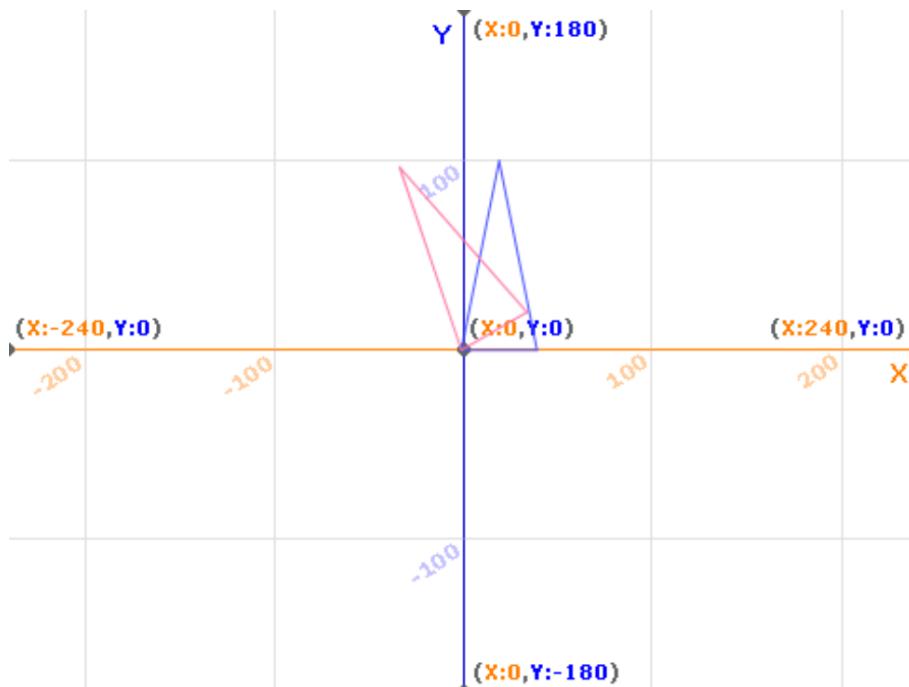
```

30 度反時計回りに回転するよう変換して図形描画するスクリプトです。

```

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

```



### [ 参考文献 ]

『Python ではじめる数学の冒険

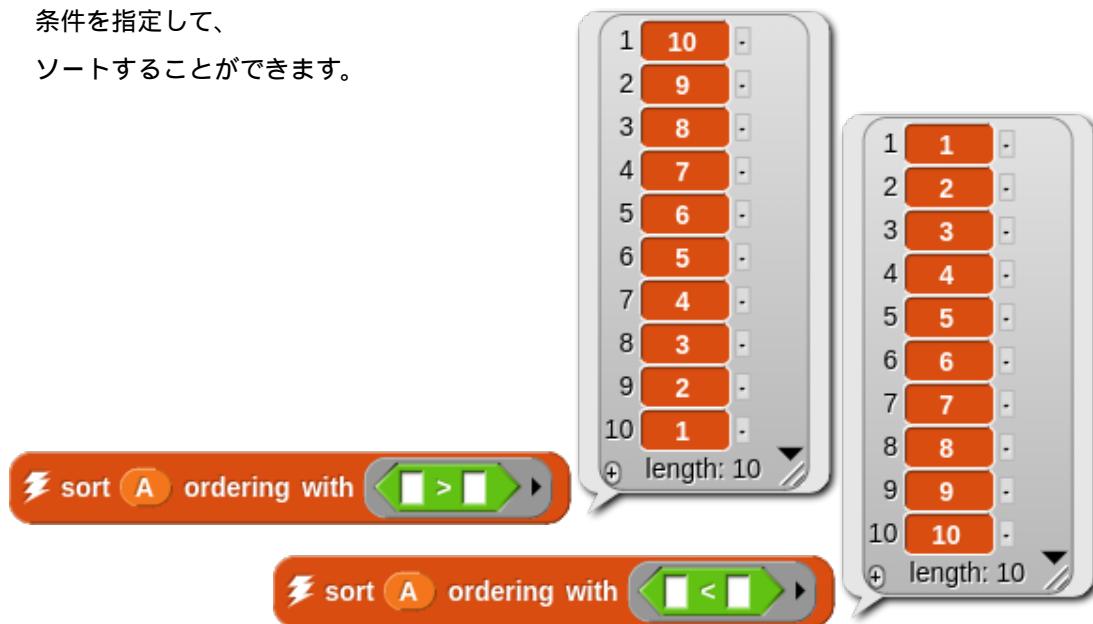
プログラミングで図解する代数、幾何学、三角関数』

Peter Farrell 著 鈴木幸敏 訳 オライリー・ジャパン 刊

## 12.8 ソート、順位付け

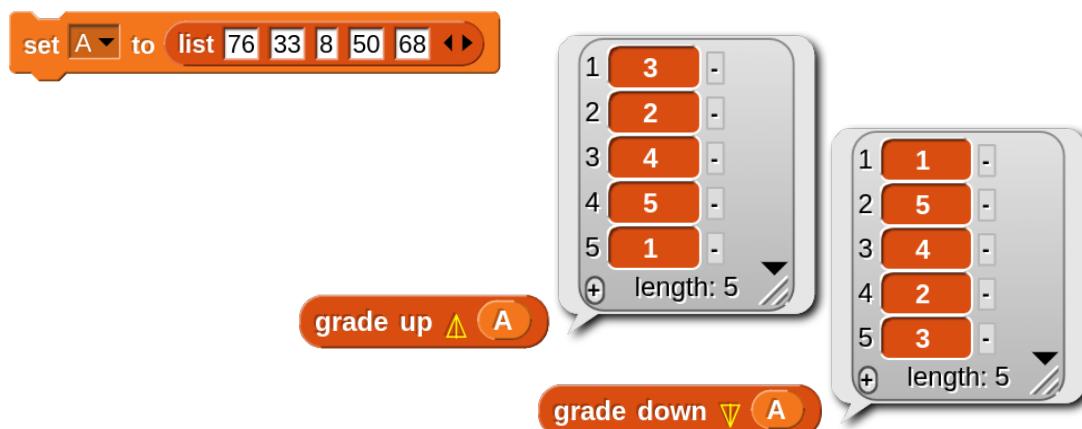
set A ▾ to list 1 8 2 6 3 5 4 7 10 9 ⏪とした場合、

条件を指定して、  
ソートすることができます。



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

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



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

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

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

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

grade up ▲ grade up ▲ A

grade up ▲ grade down ▼ A

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

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

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

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

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

## 索引

- all but first of, 21, 114–117, 125
- APL, 137
- break, 80
- call, 36
- case, 74
- catch, 81
- combine, 27, 96
- composition, 49
- continuation, 77
- continue, 80
- Costumes, 9
- csv, 6, 14, 30
- draggable?, 9
- factorial, 111
- fold, 128
- foldl, 129
- foldr, 129
- import, 9, 30, 49
- input list:, 36
- Iteration, 49
- iteration-composition, 81
- JavaScript, 93
- JavaScript extensions, 5
- join, 24, 27
- json, 6, 14, 30
- keep, 26, 119
- Language, 5
- letter, 30
- Libraries, 9
- list view, 19
- map, 24, 25, 96, 128, 129
- New, 9
- object, 67, 69
- Open, 9
- pen trails, 32
- rank, 139
- relabel, 12
- reshape, 22, 23, 137
- reverse, 118, 129
- ringify, 31, 52
- rotation style, 8
- run, 35
- Save, 9
- Save As, 9
- scene, 9
- script pic, 12
- split, 24, 28
- switch, 74
- table view, 19
- thread, 107
- throw, 81
- transient, 15
- turbo mode, 10
- unringify, 31
- UTC, 99
- w/continuation, 77
- while, 75
- with inputs, 35
- Zoom, 5
- オフライン版, 5
- 階乗, 27, 111
- 回転行列, 149
- カスタムブロック, 7, 43
- 形, 137

カリー化, 133  
協定世界時, 99  
行列の積, 148  
クイックソート, 119  
組み合わせ, 27  
グローバル変数, 14, 15  
継続, 77  
高階関数, 128  
再帰, 111  
座標変換, 149  
ジュークボックス, 7  
順列, 27  
スカラー, 137  
スクリプトエリア, 7  
スクリプト変数, 17  
ステージエリア, 6  
ステップ実行, 8, 47, 82  
スプライトコラル, 7  
スプライト変数, 16  
スレッド, 107  
ゼブラカラーリング, 12  
ソート, 94  
素数, 146  
ターボモード, 10  
大域変数, 15  
単位行列, 145  
定義ブロックのインポート, 60  
定義ブロックのエクスポート, 60  
デバッグ, 8, 82  
転置, 141  
時計, 99  
二進数, 95  
日本語化, 5  
排他的論理和, 95  
配置転換, 140  
配列, 22, 94, 137, 140, 141, 143  
バックグラウンド, 7  
ハノイの塔, 111  
パレットエリア, 7  
八口, 13  
ピット演算, 94  
評価, 69, 71  
フィボナッチ数列, 122  
フォーマルパラメーター, 24, 37  
プリミティブブロック, 7  
並列処理, 102  
ベクター, 137  
ベクトル, 137  
変数ウォッチャー, 6, 15, 19, 20, 30  
末尾再帰, 125  
無名関数, 37  
ラムダ関数, 37  
リング, 24, 31, 36, 40, 52, 68, 69, 71, 75  
ループ変数, 18  
連結, 140  
ローカル変数, 16  
ロケーションピンアイコン, 16  
論理積, 95  
論理和, 95  
ワードローブエリア, 7  
ワープ, 10