

Snap! のこと

齋藤文康

2024 年 3 月 1 日

Snap! Build Your Own Blocks(ver. 9) 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。

見た目は Scratch に似ていますが、関数型プログラミングの要素を取り入れた Snap! には Snap! Build Your Own Blocks が示すように、Scratch とは異次元のブロックが作れるという楽しみがあります。プログラミング、アルゴリズムの実験にも有益と思われます。

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

Snap! は短い周期で更新されていますので記述が合っていない箇所があるかもしれません。私の理解不足で間違っているところがある可能性もあります。正しい内容になるよう努めましたが、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	13
4.1 for all sprites (全部のスプライトで使えるグローバル変数)	13
4.2 for this sprite only スプライト変数	15
4.3 script variables スクリプト変数	16
4.4 for ループ変数	17
4.5 リスト処理用ブロック	18
4.5.1 numbers form () to ()	19
4.5.2 () in front of ()	20
4.5.3 all but first of ()	20
4.5.4 index of () in ()	21
4.5.5 append () ()	21
4.5.6 for each () in ()	21
4.6 reshape () to () ()	22
4.6.1 map () over ()	23
4.6.2 keep items () from ()	24
4.6.3 find first item () in ()	25
4.6.4 combine () using ()	25
4.6.5 combinations () ()	26
4.7 リストの演算	27
4.8 変数に入れられるもの	28
5 Control 制御	30
5.1 リポーターの if then else	31
5.2 when ()	31
5.3 stop ボタンがクリックされた時の終了処理	32

5.4	run ブロック	33
5.5	call ブロック	34
5.6	launch ブロック	36
5.7	broadcast ブロック, tell to ブロック	39
5.8	pipe	41
6	ブロックを作成する	42
6.1	() >= () ブロック	42
6.2	help 説明文の作成	45
6.3	for (i) = (start) to (end) step (add)	46
7	ブロック定義について	59
7.1	プルダウン入力	59
7.2	Title Text とシンボル	63
7.3	Input name オプションについて	65
7.3.1	Reporter 型	65
7.3.2	Predicate 型	68
7.3.3	Command 型	69
8	その他	75
8.1	デバッグ	75
8.2	入力スロットへのリスト指定	77
8.3	ask	78
8.4	broadcast の検索オプション	78
8.5	クローン	79
8.5.1	テンポラリクローン	79
8.5.2	パーマネントクローン	83
8.6	flat line ends	84
8.7	anchor アンカー	86
8.8	JavaScript function (オプション 5 ページ参照)	87
8.9	時計	92
8.10	並列処理について	95
9	再帰呼び出し	100
9.1	再帰の例	100
9.1.1	階乗	100
9.1.2	ハノイの塔	100
9.2	再帰呼び出しの使用	101
9.2.1	繰り返し	101
9.2.2	カウントダウンとカウントアップ	102
9.2.3	my length	102

9.2.4	my contains	105
9.2.5	unique	105
9.2.6	リスト要素の巡回	106
9.2.7	リストからある要素を削除したリストを返す	108
9.2.8	クイックソート（整列 / 並べ替え）	109
9.2.9	末尾再帰	112
10	高階関数	116
10.1	カリー化	125
10.2	Class 的な応用	126

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 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、ブロックの大きさを変更することができます。
JavaScript extensions がチェックされていると JavaScript ブロックが使用可能になります。

2 画面まわり

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

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

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

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

英語版でも変数名やデータの内容など日本語が使えます。

2.1 エリア

各エリアには次のような名前がついています。



2.1.1 ステージエリア

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



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

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

2.1.2 スプライトコラル



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

2.1.3 スクリプトエリア

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

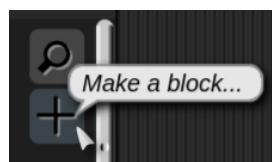
スクリプトエリアの右上には、 と があります。

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

は、キーボードを使ってスクリプトを作成するモードへのスイッチです。

2.1.4 パレットエリア

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



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

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

2.2 エリアの大きさ



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



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

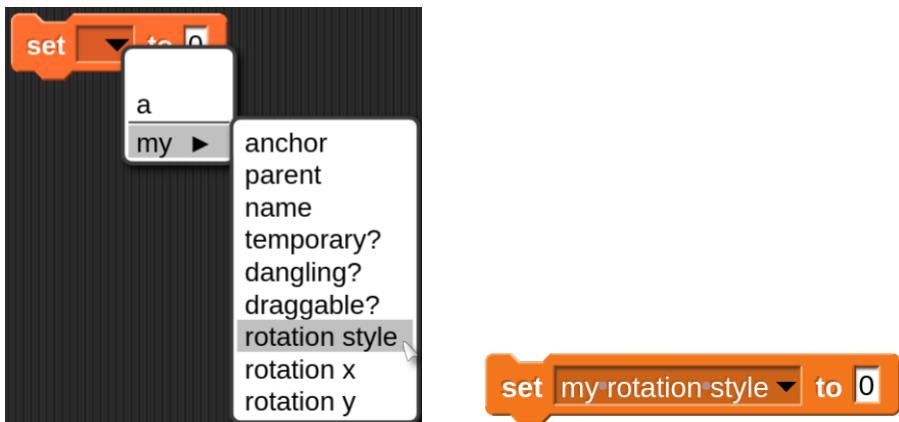
2.3 実行に関するボタン

ステージエリアの上には のボタンがあります。これは に接続された
スクリプトを実行するボタンです。 はスクリプトの実行を終了させるボタンです。 はスクリプトの実行を一時停止させるボタンです。 のボタンをクリックすると実行中のブ
ロックをハイライトさせてゆっくり実行させたりできます。 のマウスポインター
が置かれているところをドラッグすると実行のスピードが調整できます。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。(75 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。

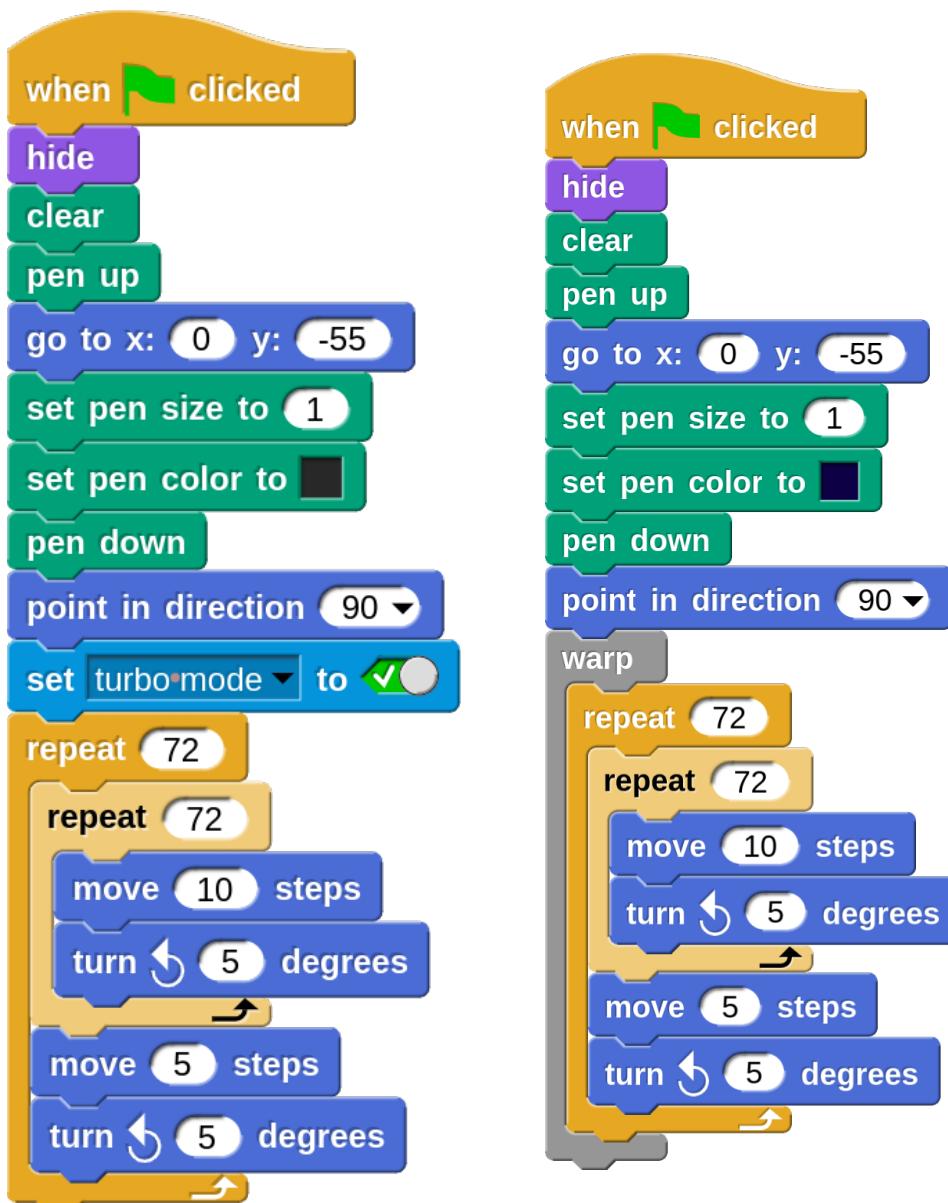
Scratch で高速に実行するためのターボモードが Snap! にもあります。

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

Sensing パレットに  ブロックがあります。こ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、

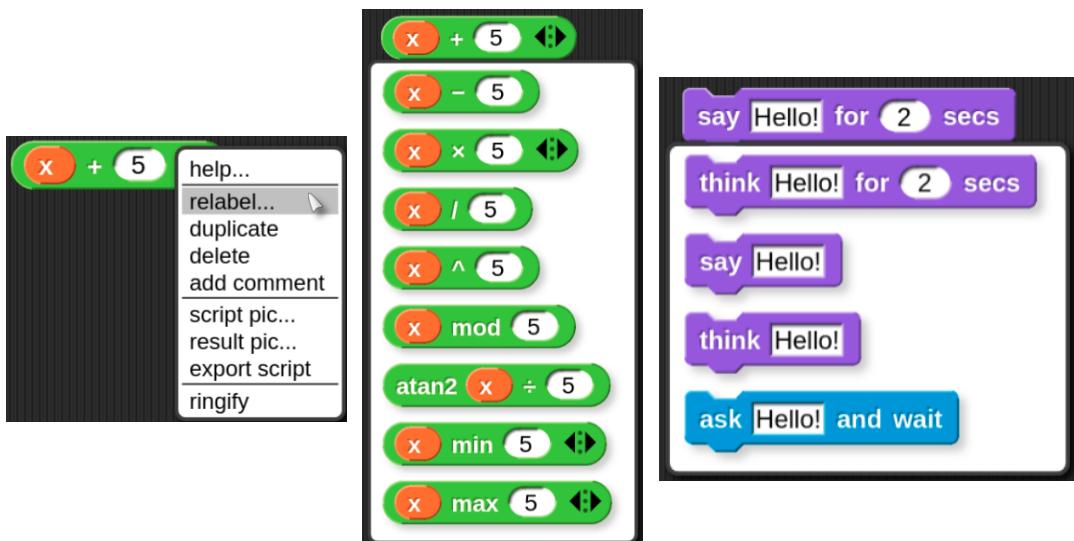


となります。長くなった場合は、自動的に折り畳まれます。また、同じパレットのブロックが重なった場合は、色合いが交互に変化して表示されます。ゼブラカラーリングだそうです。



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ーへエクスポートされます。ファイル名はプロジェクト名 + script pic + (番号) + 画像ファイルを表す拡張子になります。

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

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

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



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

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



16/4/2
16 / 4 / 2 2

16/(4/2)
16 / 4 / 2 8

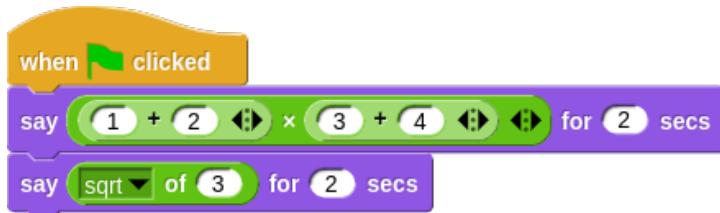


  で入力スロットを移動して変更できます。 で次のブロックに移ります。

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、o (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。

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



4.1 for all sprites (全部のスプライトで使えるグローバル变数)

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

名前を変更する場合はその変数のところを右クリックして、



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



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

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

 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 8.1KB 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)

```
<!DOCTYPE html>
<html>
  <head>
    <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.">
    <meta name="author" content="Bernat Romagosa, Michael Ball, Jens Mönig, Brian Harvey, Judge Parker, and many others.">
    <meta name="snap-cloud-domain" location="https://snap.berkeley.edu">
  ...

```

この状態で save します。すると、ファイルの容量は 1.6MB 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 13.2KB 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囲によって種類があります。

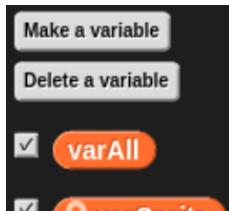
4.2 for this sprite only スプライト変数



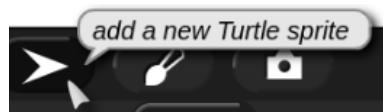
(Make a variable) をクリックして、

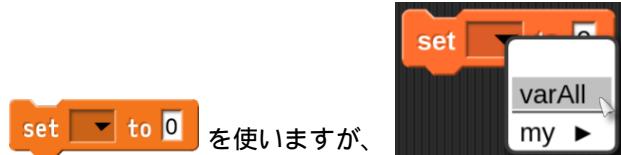


for this sprite only を選んで varSprite を作



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

 をクリックして、 新しいスプライトを追加します (Sprite(2))。するとスクリプトエリアやパレットエリアには追加したスプライトのためのものが表示されます。パレットエリアには varAll は表示されますが varSprite は表示されません。Sprite(2) からは varSprite が見えない、つまり使えないということです。変数に値を入れるには

 を使いますが、 varSprite は変数名リストに出てきません。

4.3 script variables スクリプト変数

 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  をスクリプトエリアに持ってくるだけでは使えません。



のように接続されてからでないと使えません。



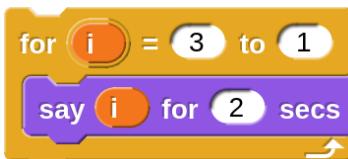
のようにその前でも有効になりません。

4.4 for ループ変数

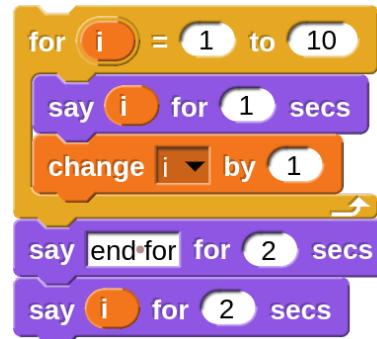
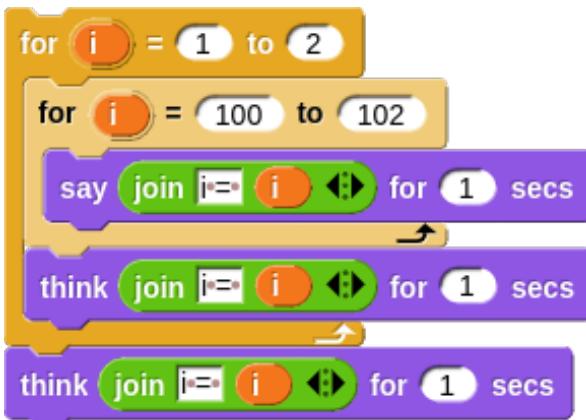


このようにあらかじめ用意されている変数があります。

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

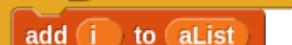
Snap! には Scratch のようなリスト専用の変数はありません。変数に数値や文字列を入れるよう に、リストを入れればリストを記憶する変数になります。



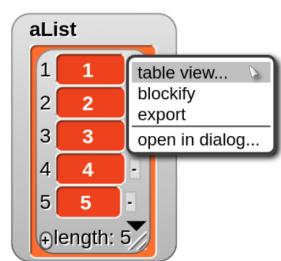
で `aList` という変数を作成すると、ステージエリアに  が表示さ れます。 の左向きの三角をクリックして  とすると、空のリ ストができます。 の右向きの三角をクリックして  で 値を入れてやると(左端の入力スロットに 1 を入れて、`Tab` キーを押すと次の入力スロットに移 れます)、`aList` は要素を持つリストになり、ステージエリアに  が表示されます。

 を使うと、 にすることができます





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  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  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。



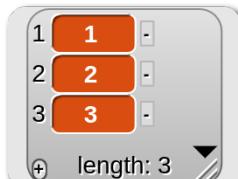
変数ウォッチャーの内部を右クリックして、blockify をクリックすると、リストブロックとして取り出すことができます。

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、 や のように。

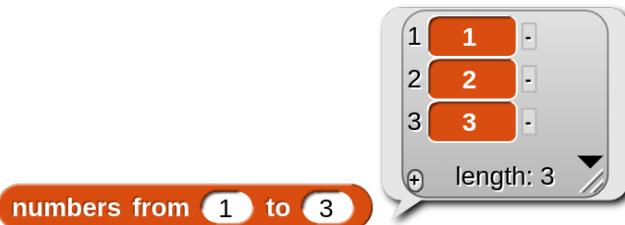
なお list view では、一度に扱える要素数は 100 までになっています。次の 101 から 200 の要素



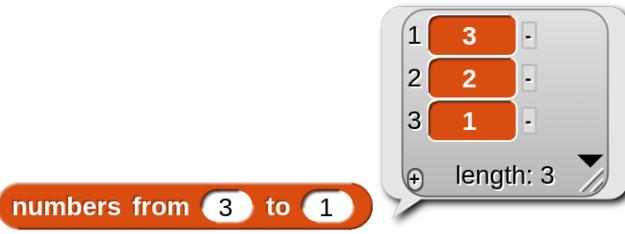
に移るには、下向きの三角をクリックすると出てくる範囲から選びます。

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。



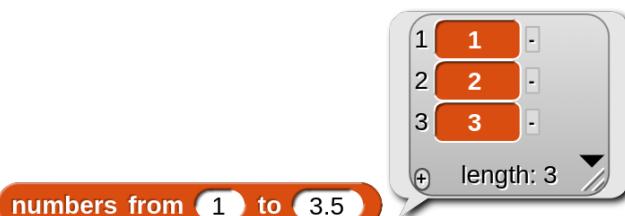
指定された範囲内で 1 ずつ増加するリストをリポート



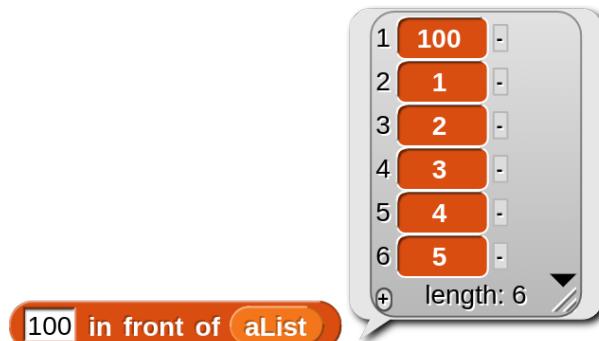
指定された範囲内で 1 ずつ減少するリストをリポート



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。

index of 5 in numbers from 1 to 10 5

index of 12 in numbers from 1 to 10 0

index of x in list a b c x y z 4

4.5.5 append () ()

これは、指定されたリストを結合したリストをリポートします。

append list 1 2 3 list 10 11 12 length: 5

append list 1 2 3 list 10 list 20 21 12 13 14

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。

for each item in aList
say item for 2 secs

for i = 1 to length of aList
say item i of aList for 2 secs

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。

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

list list 1 2 3 ⏪ list 4 5 6 ⏪ list 7 8 9 ⏪ list 10 11 12 ⏪ ⏪

reshape ブロックを使って、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。

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

reshape numbers from 1 to 20 to 4 3 ⏪

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

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

reshape list 0 ⏪ to 4 3 ⏪

1	0	-
2	0	-
3	0	-
4	0	-
5	0	-

+ length: 5

reshape list 0 ⏪ to 5 ⏪

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

set aList ▾ to reshape list 0 ⏪ to 4 3 ⏪

replace item 3 ▾ of item 1 ▾ of aList with 99

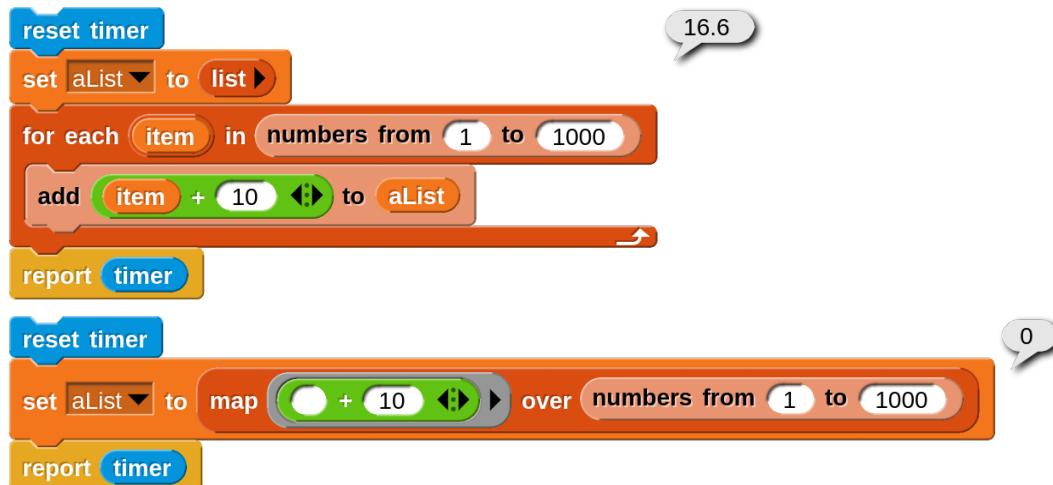
4.6.1 map () over ()



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。+ 演算なので **100 + []** でも同じです。

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



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が出てきます。パラメータとは、値を受け取るための変数のようなものです。map のフォーマルパラメータ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。



map のフォーマルパラメータは次のようにになります。index はリストの何番目の要素を処理しているかを示します。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。



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。



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。

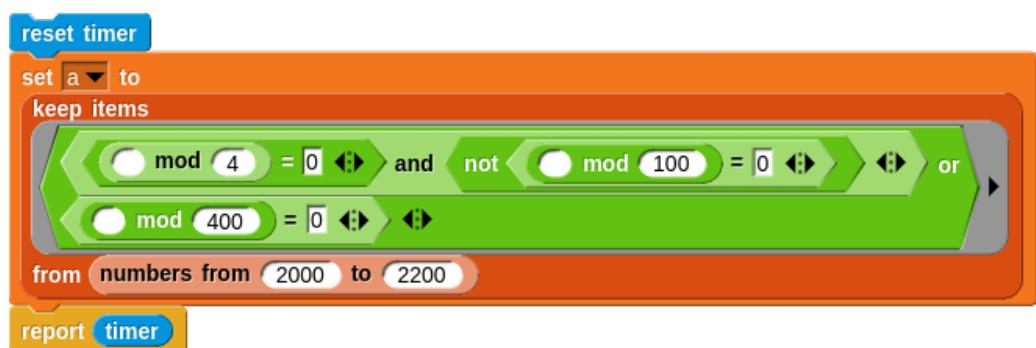
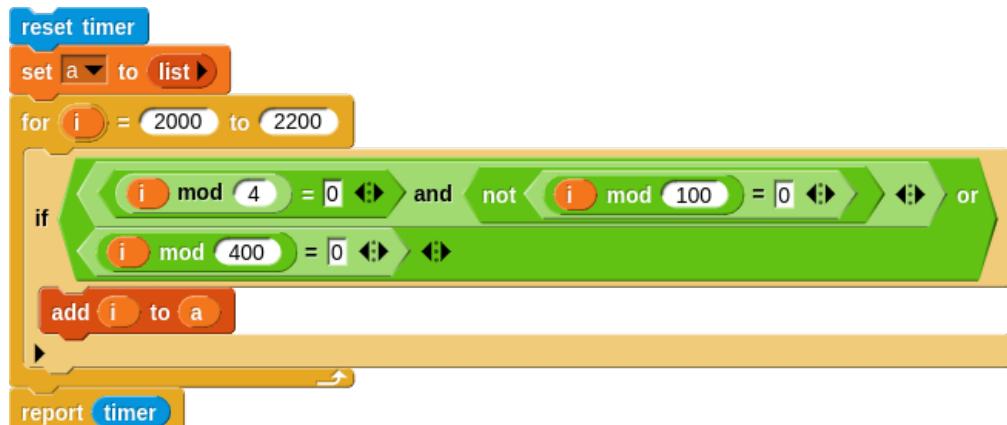
2000 年から 2200 年までのうるう年のリストを求めてみます。

4 で割り切れて 

かつ 100 では割り切れない年 

または、400 で割り切れる年  ということで、2000 年はうるう年だけれども 2100 年と 2200 年は違います。

map でもそうでしたが、for や for each ループを使うよりも高速になります。



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

4.6.3 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。

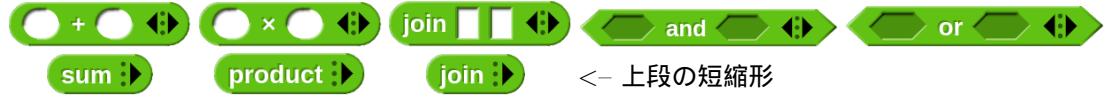
4.6.4 combine () using ()



リストの合計をリポートします。

指定されたリスト自体は変更しません。

combine では、以下の演算がおもに使われるようです。



n 個のものから r 個取り出して並べる順列の総数は、 $P_r = n(n-1)(n-2)\cdots(n-r+1)$ 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、



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、 $C_r = \frac{P_r}{r!}$ 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、



join を指定すると文字列の連結になります。(入力スロットは空です。)



これと逆の操作をするのが split ブロックです。

split combine aList using join by

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
and

combine list true false true using
and

combine list true true true using
or

combine list true false true using
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

4	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。)

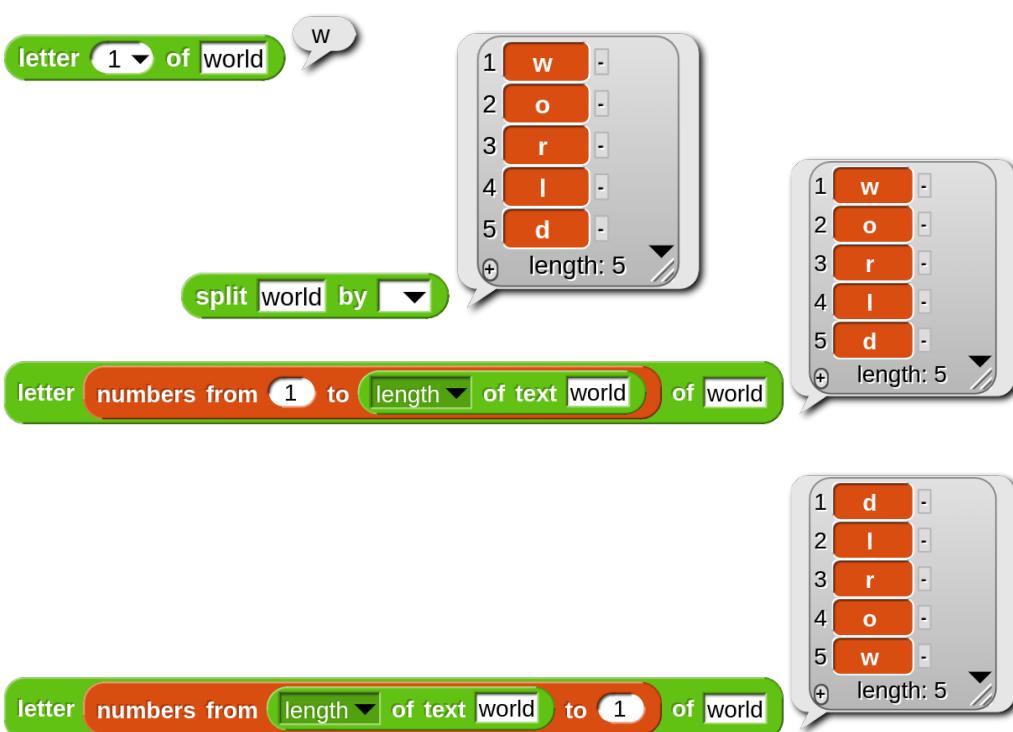


リスト同士で演算をすることもできます。いずれも同じインデックスの要素同士を演算します。要素数が合わない場合は対応する部分だけで行います。



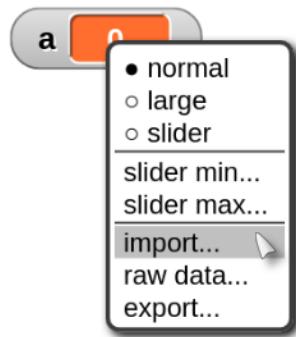


リストに文字列の各要素を入れることもできます。



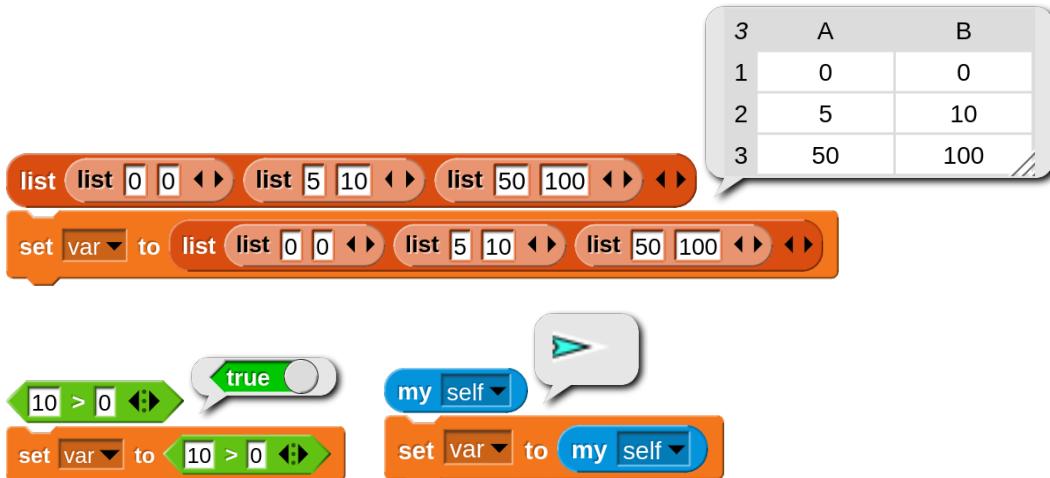
4.8 変数に入れられるもの

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

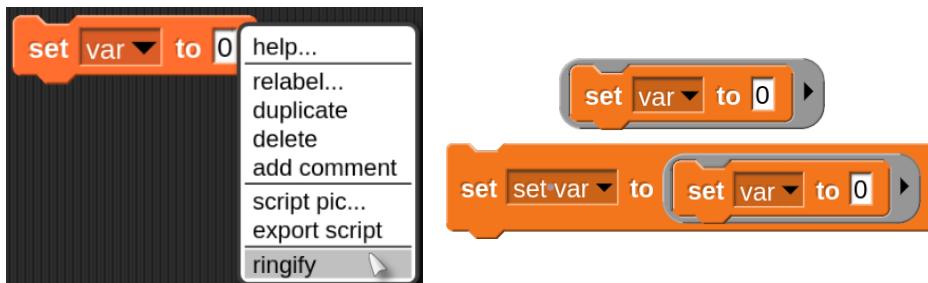


「import...」で、指定したファイルの内容を取り込むことができます。.csv や .json のファイルの場合は、そのデータ形式に従ってリストを作成します。.csv ファイルではカンマと改行をセパレーターとした要素をリストにします。単なるテキストデータとして取り込みたい場合は、「raw data...」を使います。

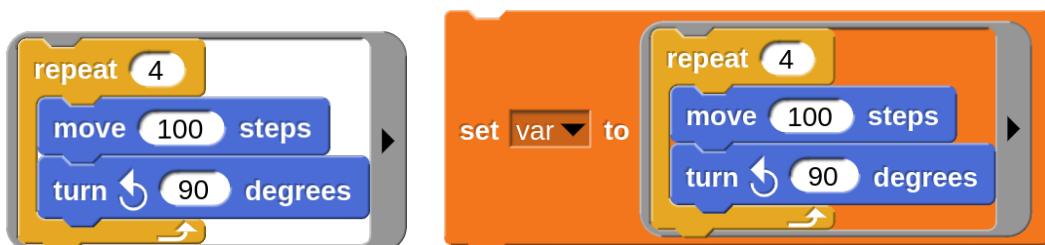
Snap! の変数には、値をリポートするものは入れられるようです。



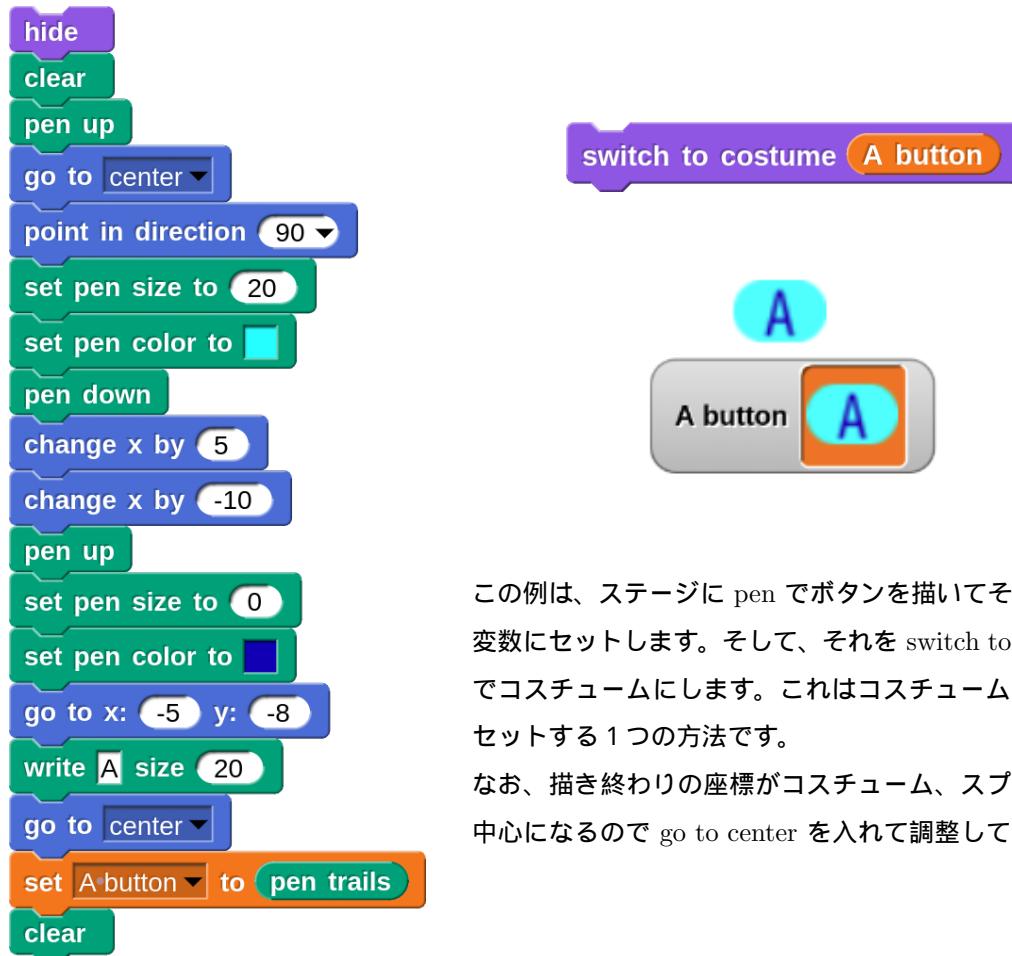
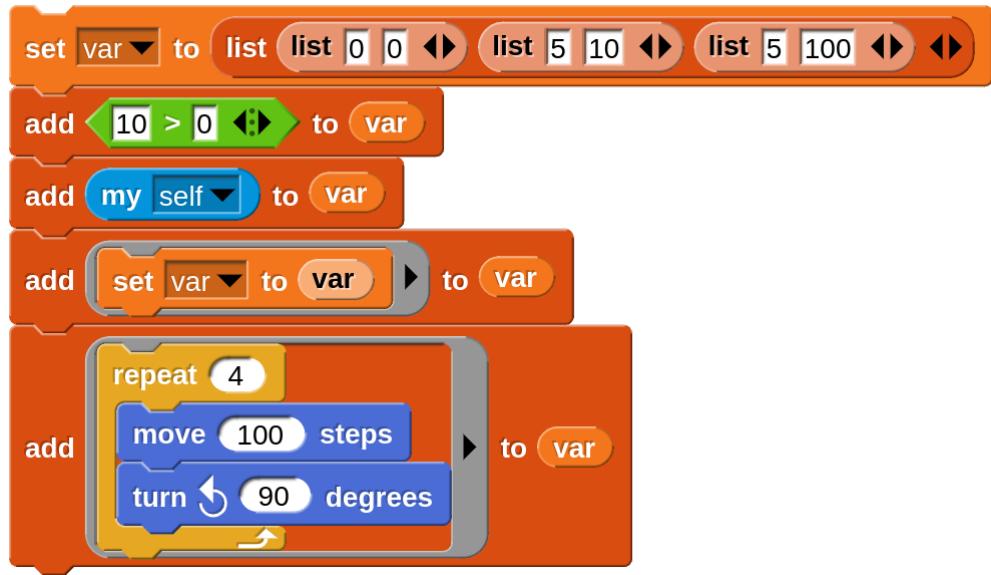
ブロックをリングに入れてやるとリポーターブロックにすることができます。ブロックそのものをリポートするものです。対象のブロックのところで右クリックしてメニューから ringify を選べばリングで囲むことができます。リングを外すには unringify をクリックします。



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 then  
  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 []
 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]
wait [0.2] secs
next costume
move [10] steps
if on edge, bounce

```

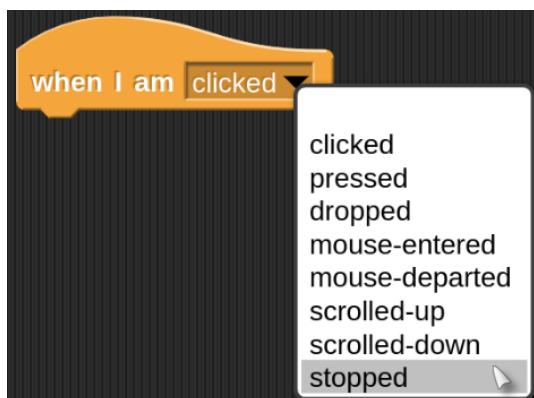
```

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

```

足を開いていたならば閉じる

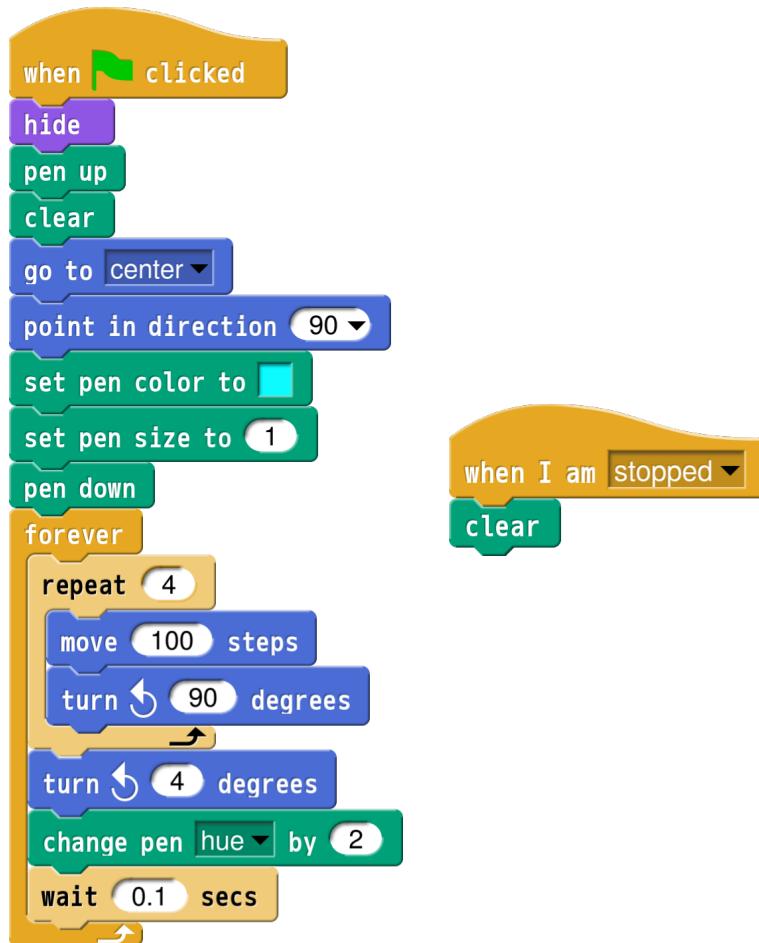
5.3 stop ボタンがクリックされた時の終了処理



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。

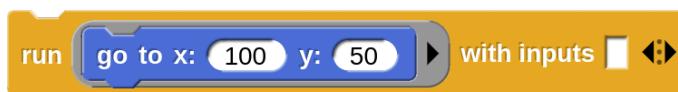
次のスクリプトで動作の確認ができます。時間的にも限られたことしかできないようです。状況によってはきれいに clear できない場合があります。



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

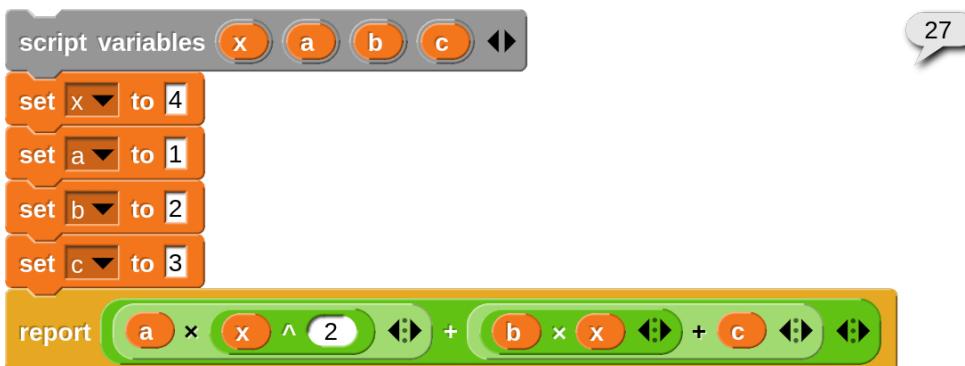
run ブロックに入れられるのが値をリポートしない実行 (Command) ブロックだったのに対して、call ブロックに入れられるのは、実行または評価することによって何らかの値または true か false をリポートするブロックです。call ブロックは、得られた値をリポートします。



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こうなります。



27

変数の入力スロットを空にして、式をリングで囲みます。



リングの端の三角をクリックしてフォーマルパラメータを出します。



フォーマルパラメータは、クリックして変数名を **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。

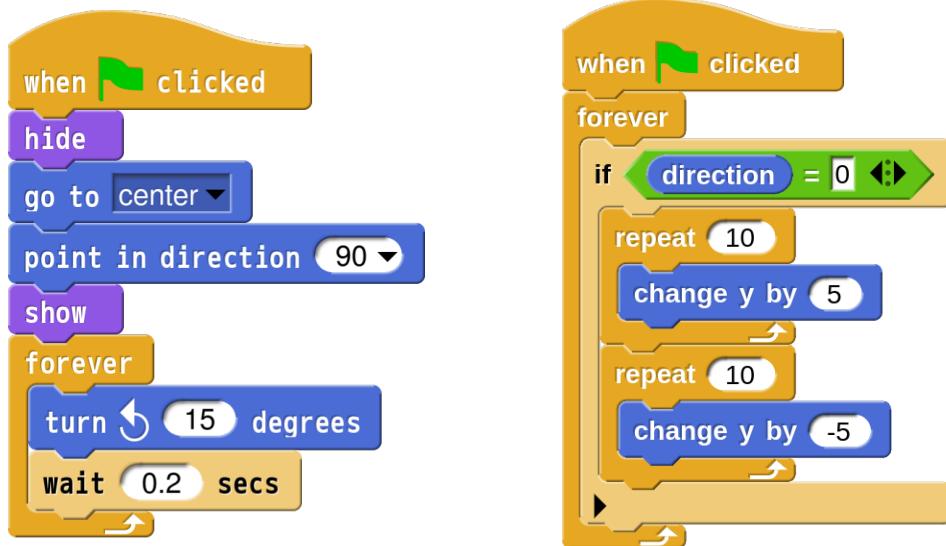


5.6 launch ブロック

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。

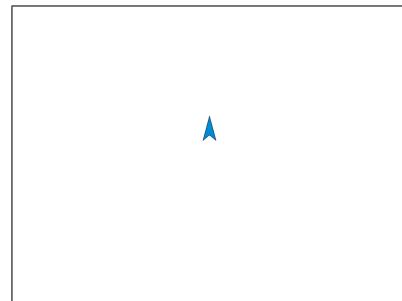
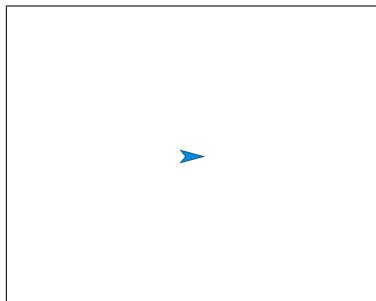
1 個のスプライトに対して、回転させるスクリプトと、角度が上(0度)の時にジャンプさせるスクリプトを並列で実行してみます。

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。

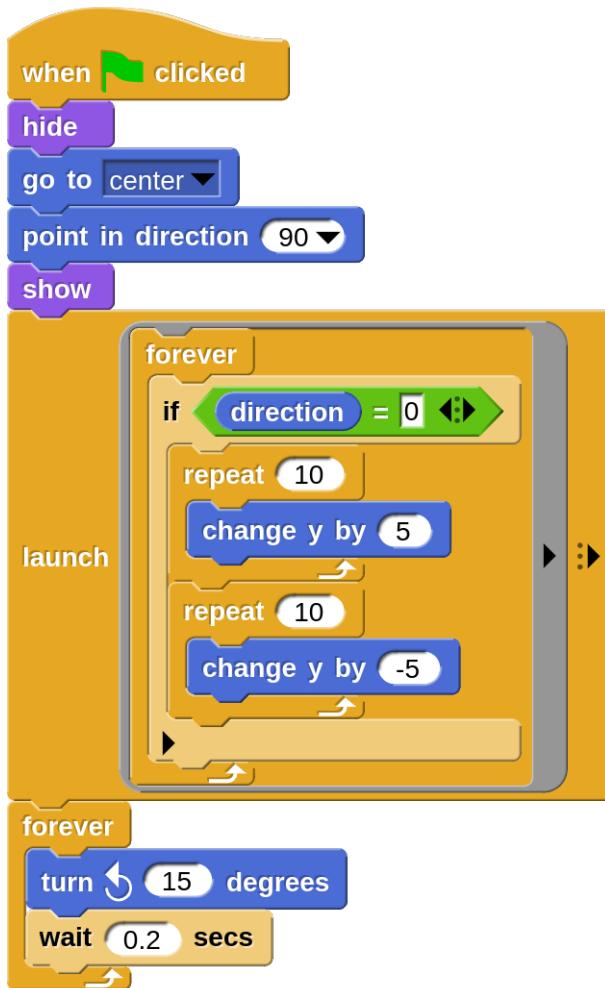


左のスクリプトで、ずっと回転させる動作を行います。

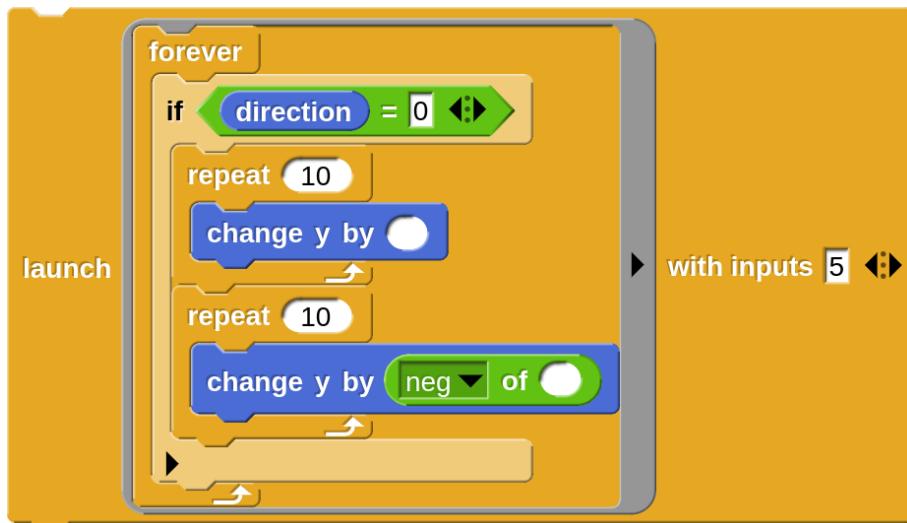
右のスクリプトで、上を向いた時だけジャンプする動作を行います。



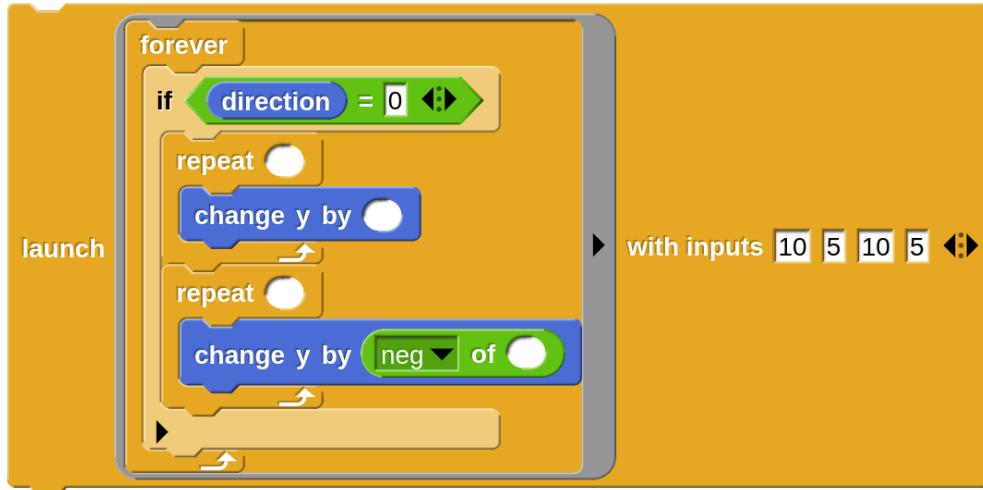
launch を使うことで1つのスクリプトで実行することができます。



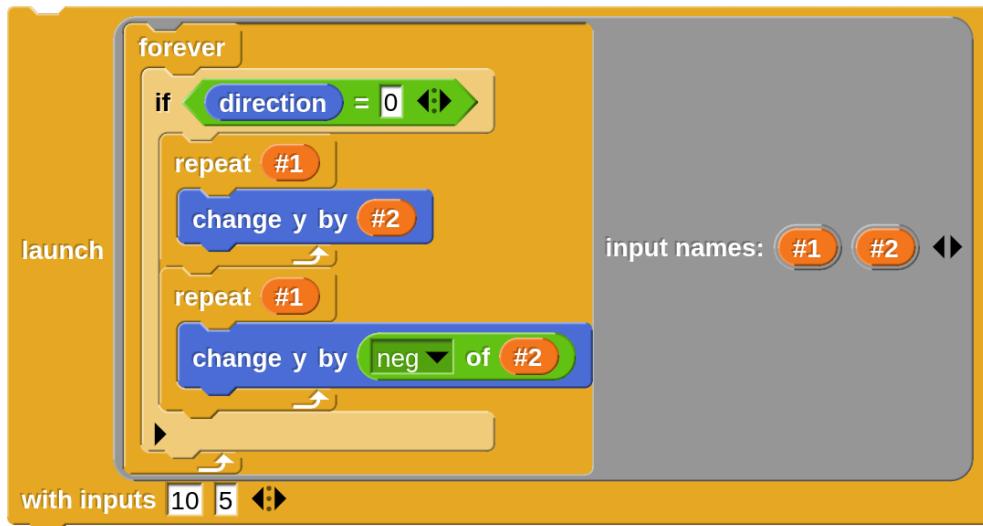
launch の右端の右向き三角をクリックすると、入力値を設定することができます。y の値の設定を空（空白ではなく）にして、入力値をひとつだけ設定すると、その入力値がすべての空の部分の値になります。



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

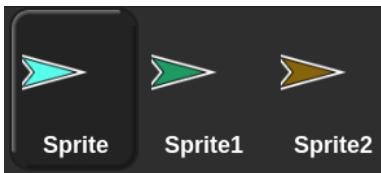


この場合は、リングではフォーマルパラメータが使えるので、次のようにすることができます。リング、灰色の部分の右端の三角をクリックして外側の入力 (10, 5) の個数と同じ個数の変数を用意します。対応する位置に目的の変数を置きます。



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

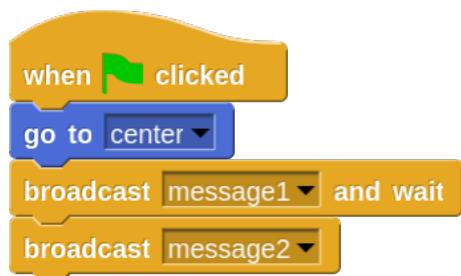
5.7 broadcast ブロック, tell to ブロック



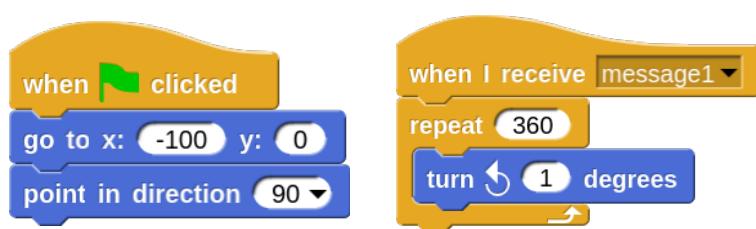
次はスプライトを3個使います。Sprite, Sprite1, Sprite2 を用意してください。

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。

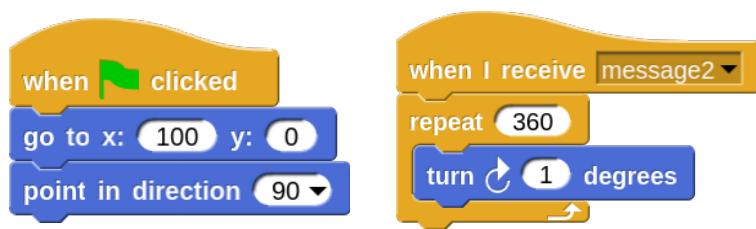
Sprite 用



Sprite1 用

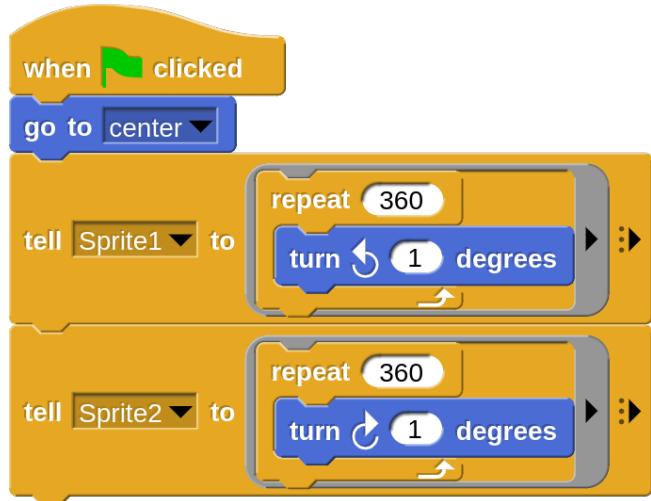


Sprite2 用



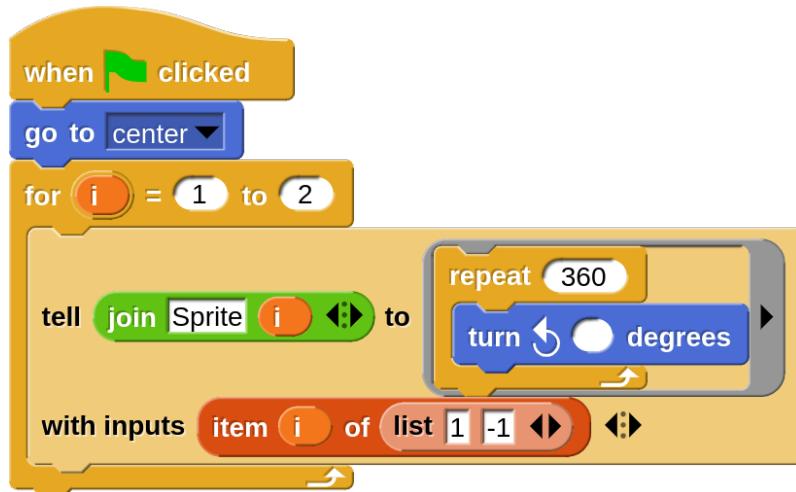
[tell [] to []] のブロックを使用すると broadcast を使わなくても Sprite から Sprite1, Sprite2 を直接操作することができます。

Sprite 用



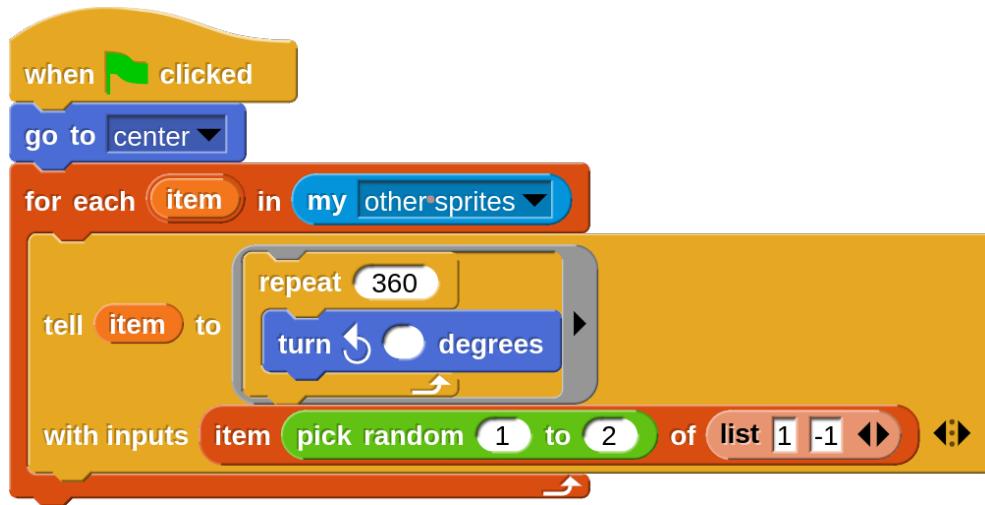
もしも Sprite1 と Sprite2 を同時に動かすのならば、tell Sprit1 のブロックを launch で囲みます。

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。

自分で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。



5.8 pipe

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

で Snap! マニュアル ver.8 冒頭部分の単語数を数えてみます。

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

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

→

→ →

→ → →

67

受け取ったリポートは指定のスロットに入って処理されます。

Snap! では次のようにできてしまうのですが、pipe を使うとデータの流れが分かりやすいかもしません。

of by

6 プロックを作成する

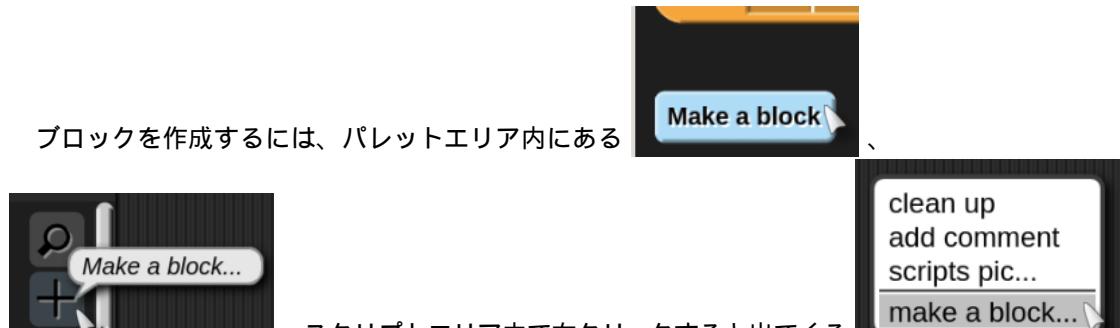
Snap! では、ローカル変数が使え、値をリポートすることもできるので、カスタムブロック（ユーザー定義ブロック）が作りやすくなりました。

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。

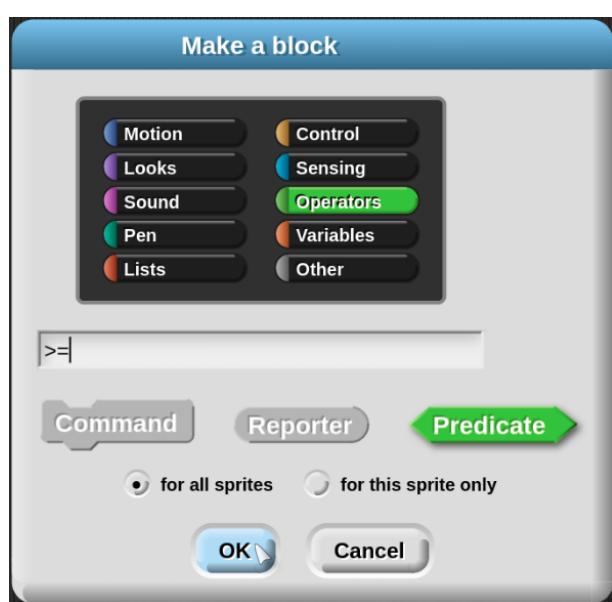
これがあると、`var > 0 or var = 0` ではなく、`var \geq 0` とできるのですっきりします。

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。



のどれかの [make a block] をクリックすれば始められます。

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。



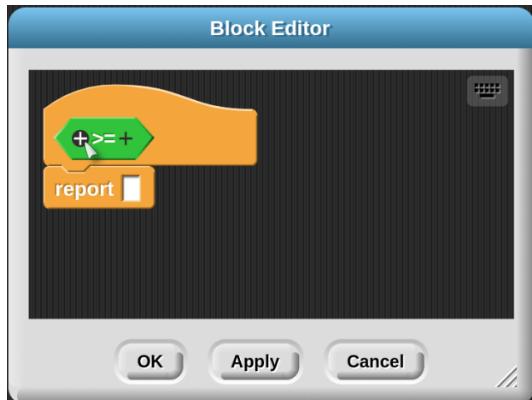
パレットのカテゴリーを選択するボタンや、新しいブロックの名前を入れる欄、ブロックの機能の種類を選択するボタン、このブロックをこのスプライトだけの機能にするかのボタンがあります。

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。

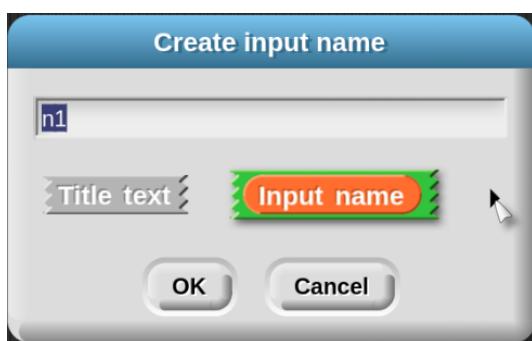
ブロック名入力欄に \geq を入れて、上の欄でボタン Operators を、下の段で Predicate を選択してください。 Operators を選択することはパレットの種類を決める事であり、置き場所を決め

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。



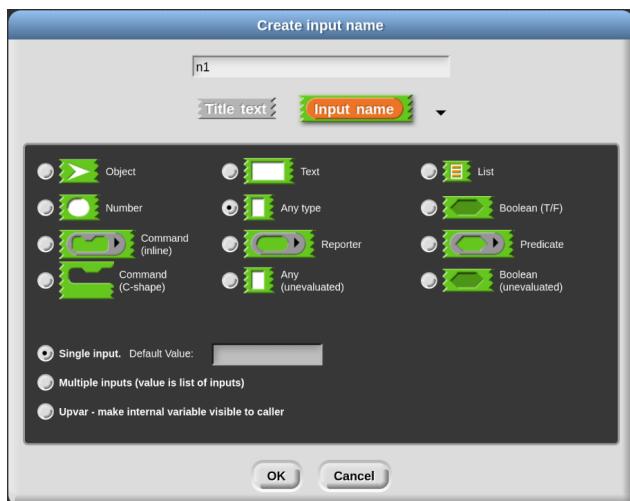
定義の先頭の の部分をプロトタイプといいます。ここをクリックするとブロックのカテゴリーを変更することができます。

「>=」の左側の + ボタンをクリックしてください。



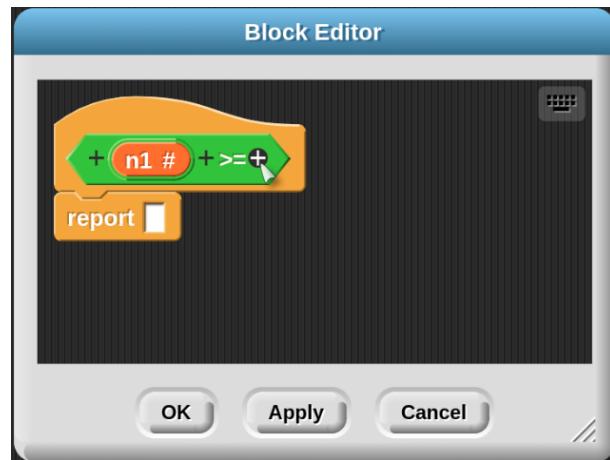
すると、入力ウィンドウが出ます。左側に Title text、右側に Input name のボタンがあります。ブロックに表示される文字列を指定する時には Title text をクリックして文字列を設定します。ブロックを使用する時にデータを受け取るための変数を指定する時には Input name で設定します。

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

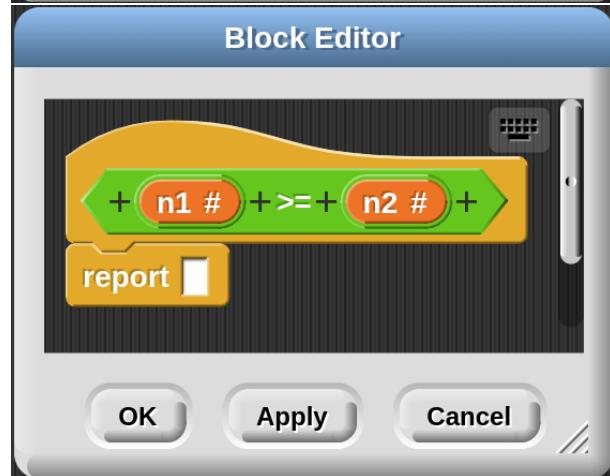


が出ます。現在は Any type が選択されています。Any type つまり数値でも文字でも受け付けるタイプです。このままでもいいのですが、あと Number にしてみます。これは数値しか受け付けないタイプです。Any type を選んだ場合は、変数の表記に「#」が付かないだけで以下の操作は同じです。

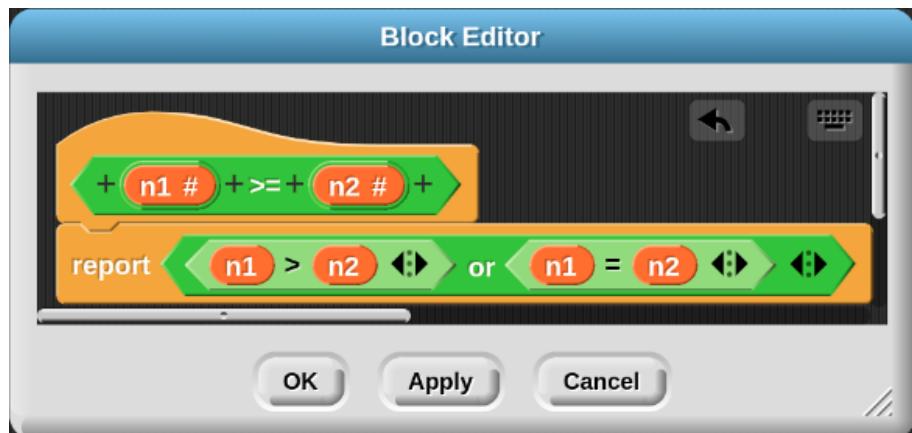
もう一つ、下の方に Single input. Default Value: が出ます。ここで入力の初期値が設定できるのですが、この場合は関係ないのでこのままにしておきます。OK をクリックして次に進んでください。



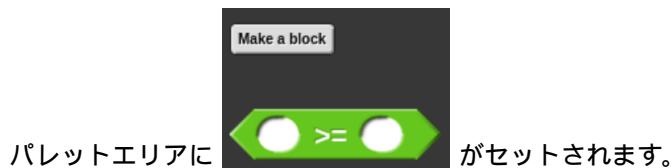
右側の + ボタンをクリックして、同じように n2 の設定をしてください。



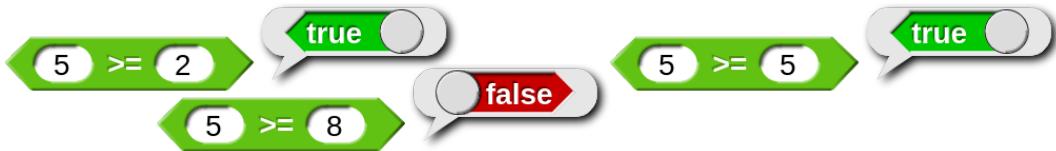
パレットエリアからブロックを持ってきて完成させてください。



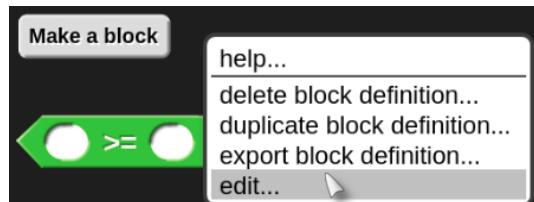
report は値をリポートする(返す)ためのブロックです。この場合は式の結果により真理値、true か false をリポートすることになります。apply をクリックしてください。



パレットエリアに がセットされます。

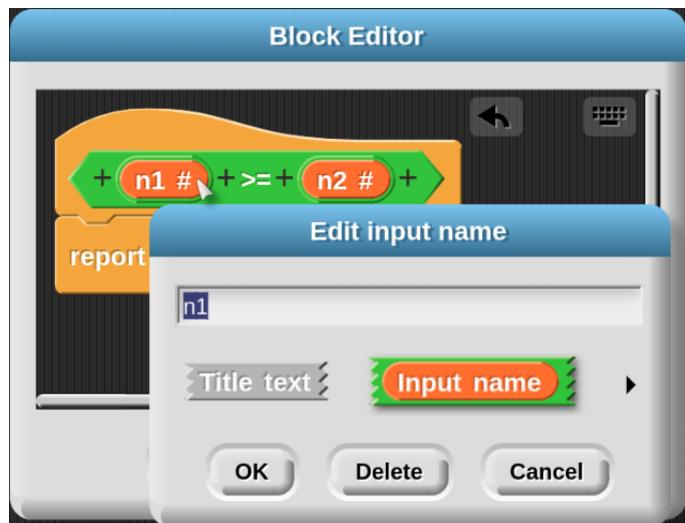


正しい値をリポートしないならばスクリプトを見直してください。正常ならば OK をクリックして終了です。



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。



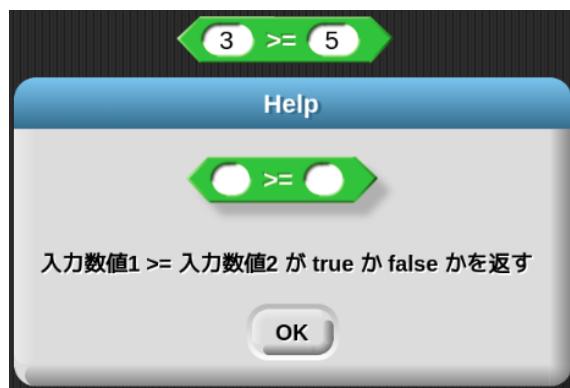
から変更ができます。変数を Any type に設定し直すこともできます。

! 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。(75 ページ参照)

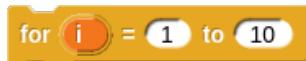
6.2 help 説明文の作成

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持っていくと、ハロが出るのでドロップします。





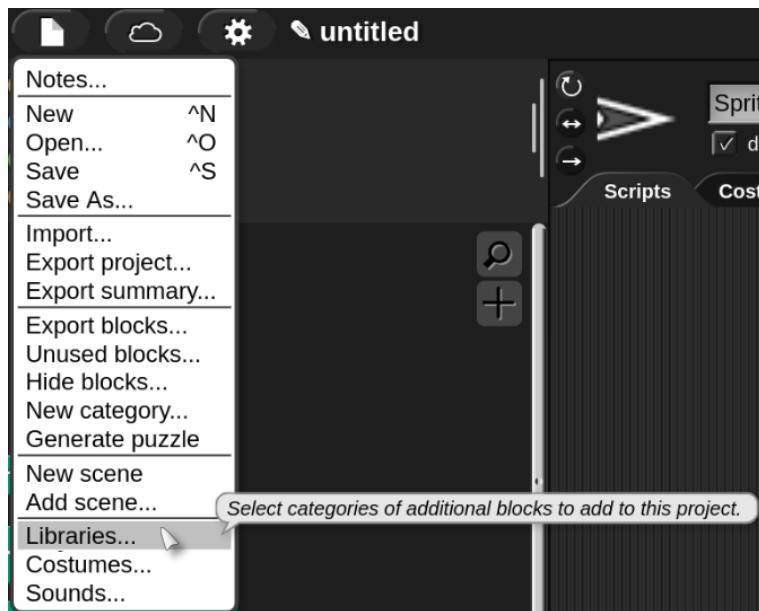
6.3 for (i) = (start) to (end) step (add)



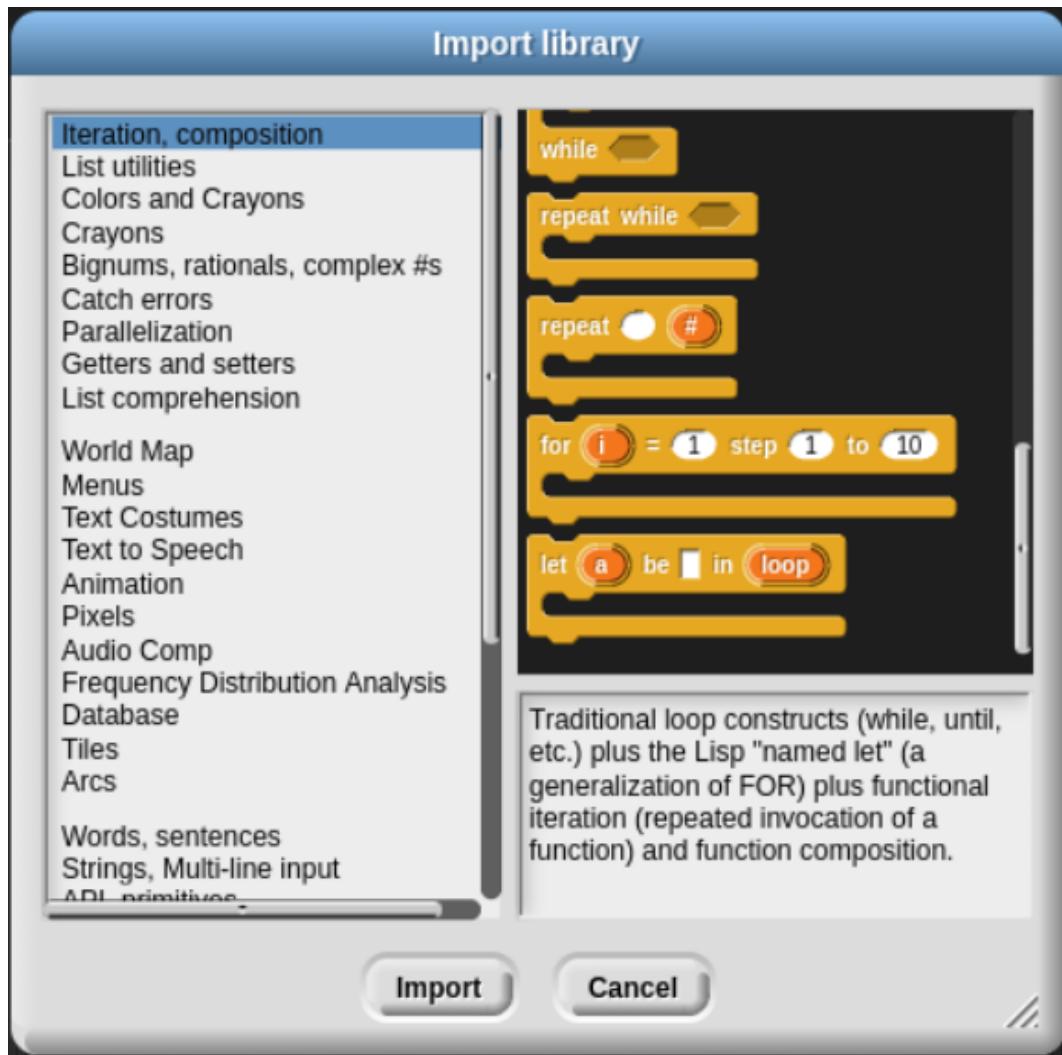
Snap! には ブロックがありますが、増減分が指定できるものも Libraries から追加できます。



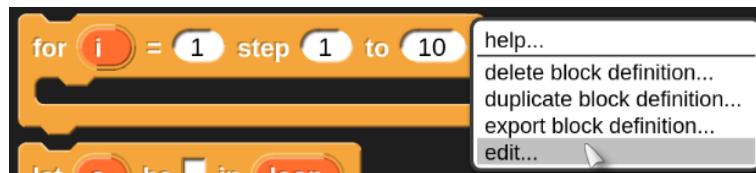
をクリックします。



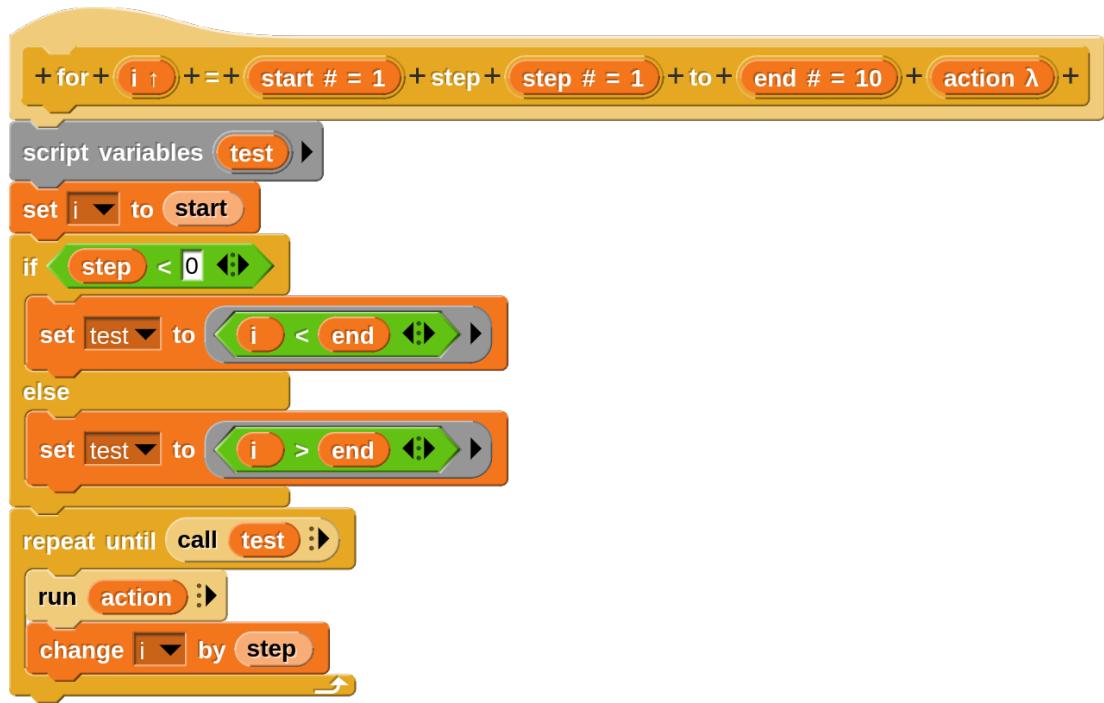
Libraries... をクリックします。Iteration, composition をクリックして内容を確かめてから Import すれば追加できます。



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



を右クリックして、edit...
で中身を見てみます。



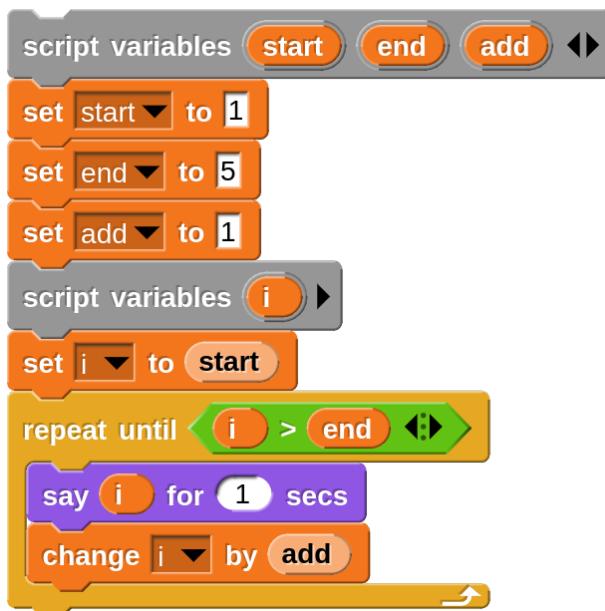
このスクリプトにしたがって、`my for (i) = (start) to (end) step (add)` という増加部分の位置が違うだけのブロックを作成してみます。

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

`i` が増加していく場合です。



`start`, `end`, `add`, `i` の変数をスクリプト変数を使い `repeat (10)` 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。

上記スクリプトから `set i to start` 以降を表示します。

```

set i to start
if > add > 0
  repeat until > i > end
    say i for 1 secs
    change i by add
else
  repeat until < i < end
    say i for 1 secs
    change i by add
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 か 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、 $i > 5$ なので、false です。

```

script variables
  [test : false]
  [set [i] to (1)]
  [set [end] to (5)]
  [set [test] to (i > end)]
report [test]

```

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。

```

script variables
  [test : false]
  [set [i] to (1)]
  [set [end] to (5)]
  [set [test] to (i > end)]
report [test]

```

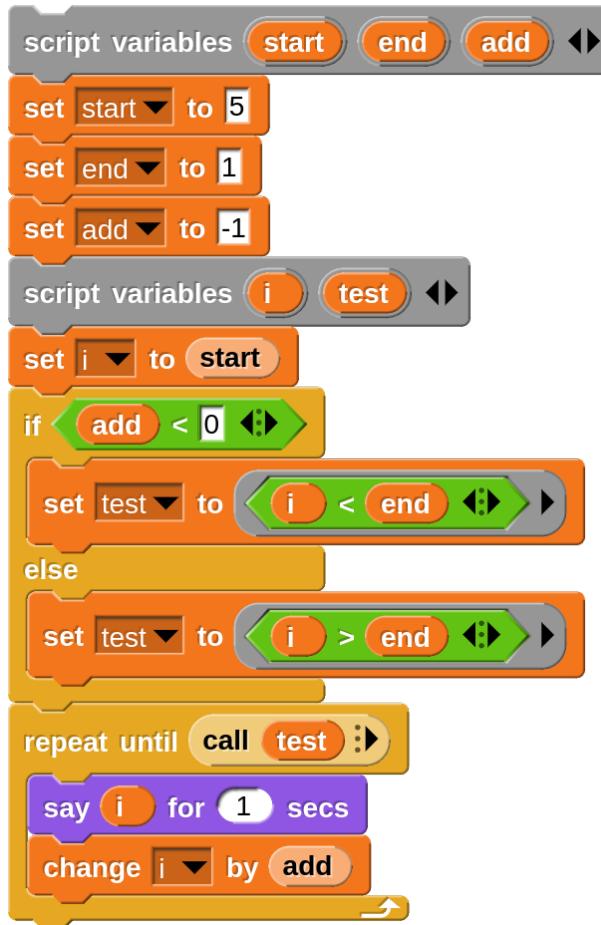


テスト項目自体を変数に入れます。

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。

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

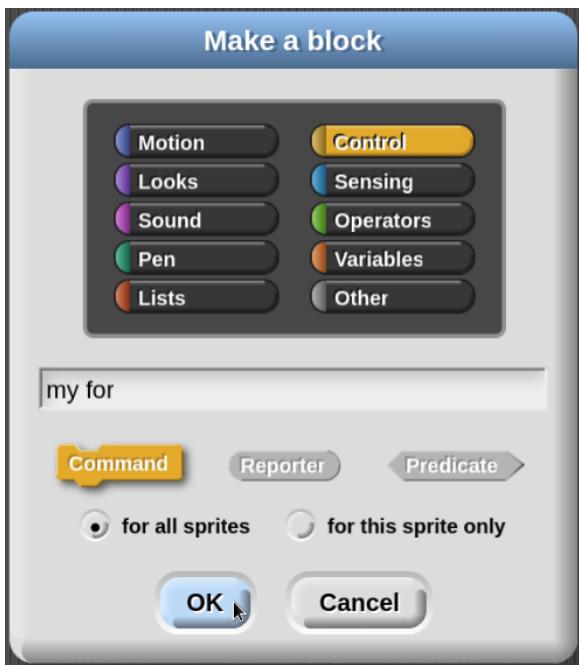
これを for ループに使用します。



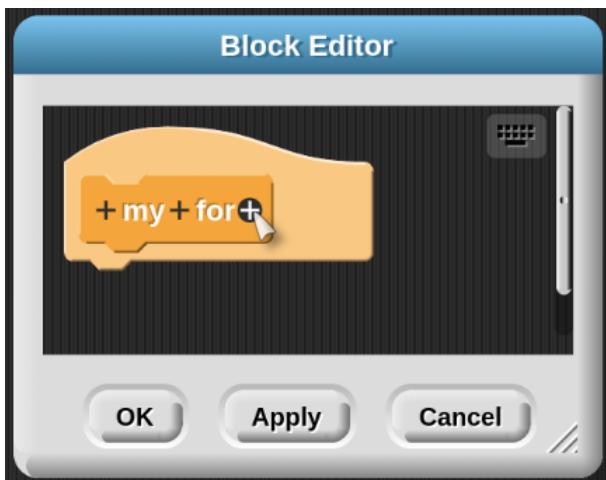
これが Libraries の for ループの内容です。

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

それでは、いよいよブロックエディターで作成していきます。



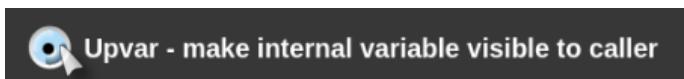
ブロックエディターを開いてから、
Control, Command を選択して
my for と入力して OK で次に進んで
ください。

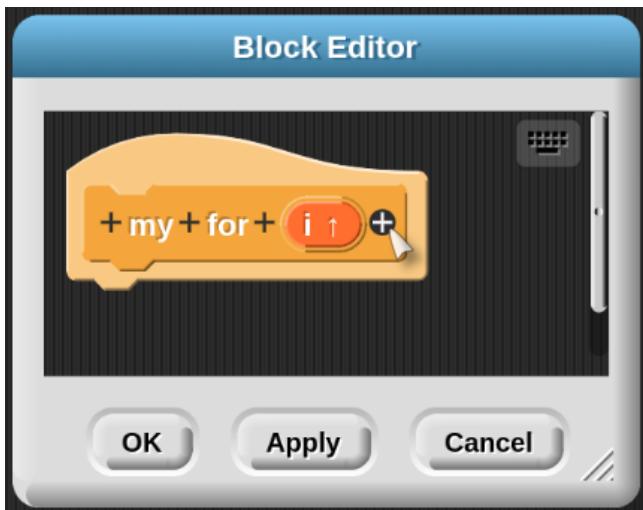


「+」をクリックして、変数 i の設
定をします。

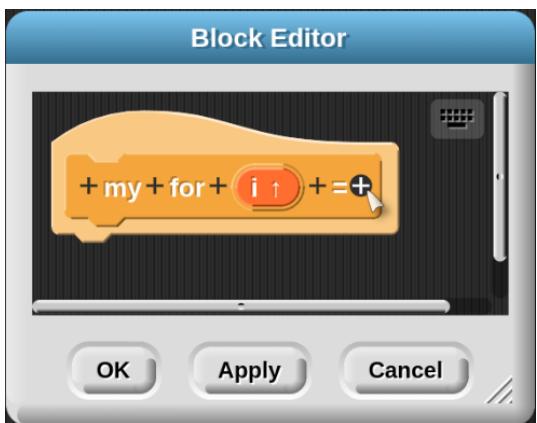


変数 i は、for ループの中にドラッグ&ドロップして使用できる特別な変数です。Upvar オプショ
ンを選択します。すると、自動的に Number や Any type のオプションはクリアされます。





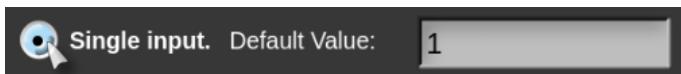
変数 i のところに Upvar を示す「↑」が表示されます。続けて、「+」をクリックして、「=」を、Title text として入力してください。



続いて、「+」をクリックして、変数 start を設定します。

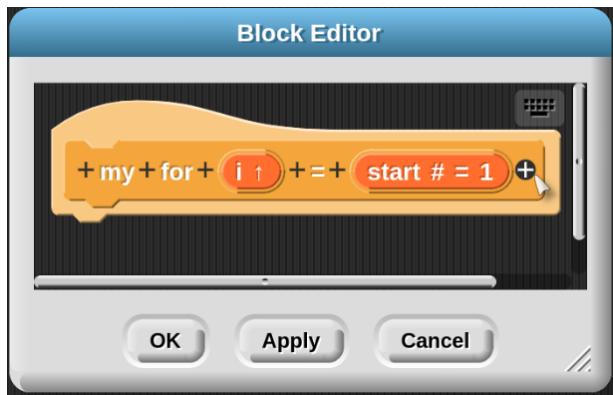


オプションとして、 Number 、数値入力限定で、



規定値を 1 に設定します。

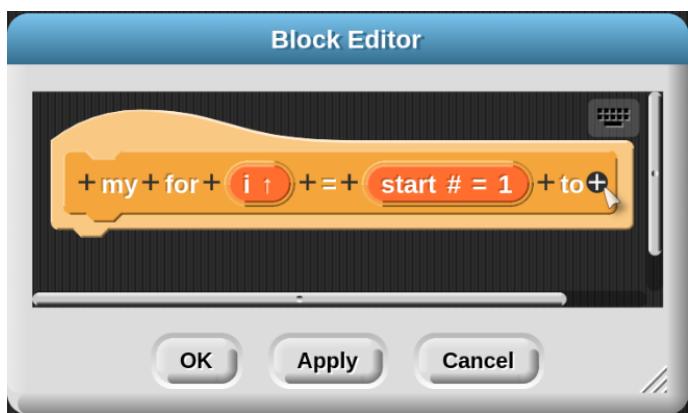
続いて、「+」をクリックして、



「to」を、Title textとして入力してください。



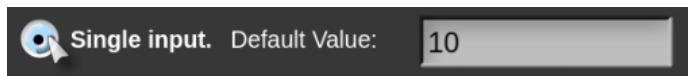
続いて、「+」をクリックして、



変数 end を設定します。

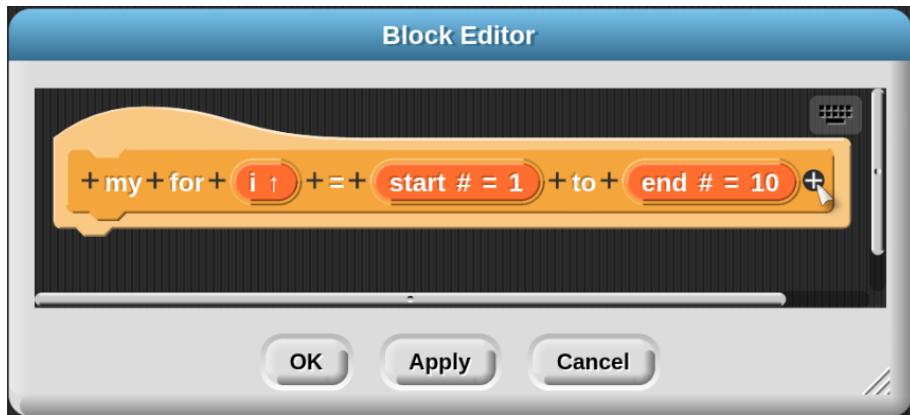


オプションとして、、数値入力限定で、



規定値を 10 に設定します。

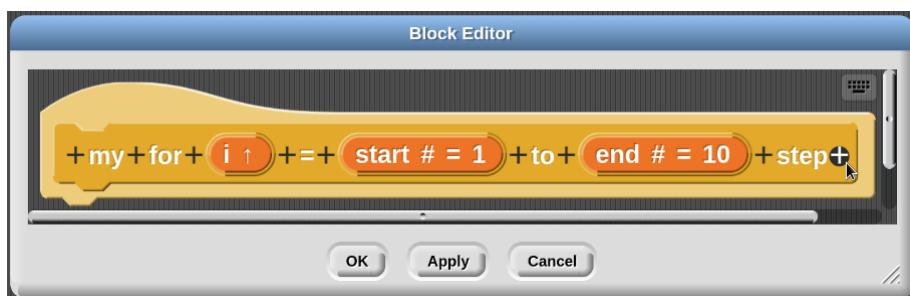
続いて、「+」をクリックして、



「step」を、Title textとして入力してください。



続いて、「+」をクリックして、



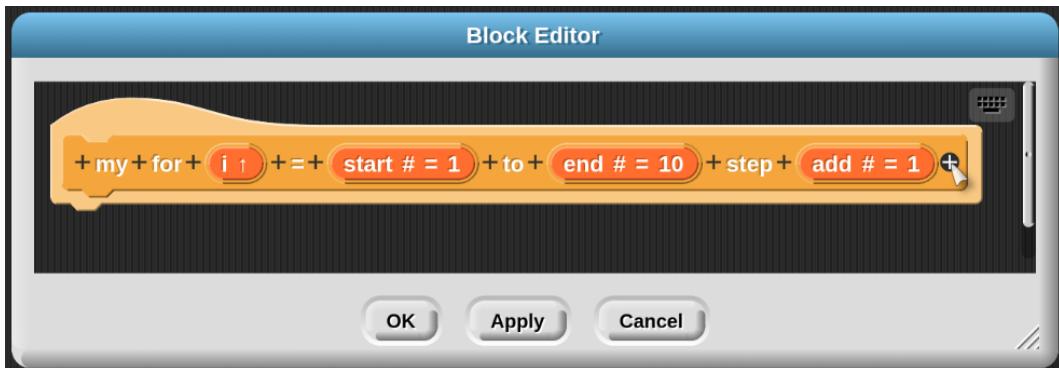
変数 add を設定します。



オプションとして、 Number 、数値入力限定で、

規定値を 1 に設定します。

続いて、「+」をクリックして、



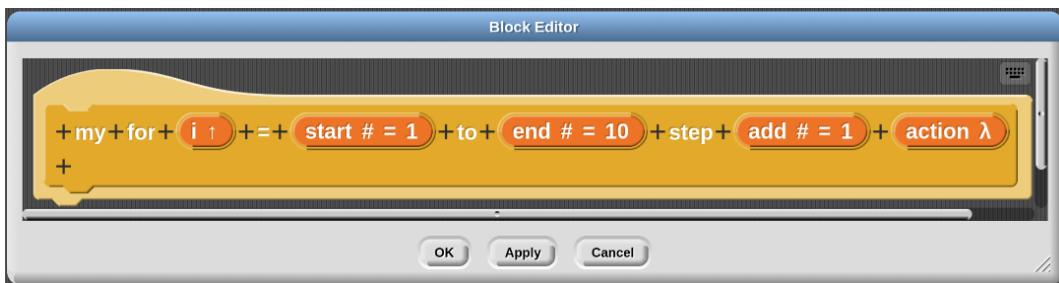
変数 action を設定します。



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。



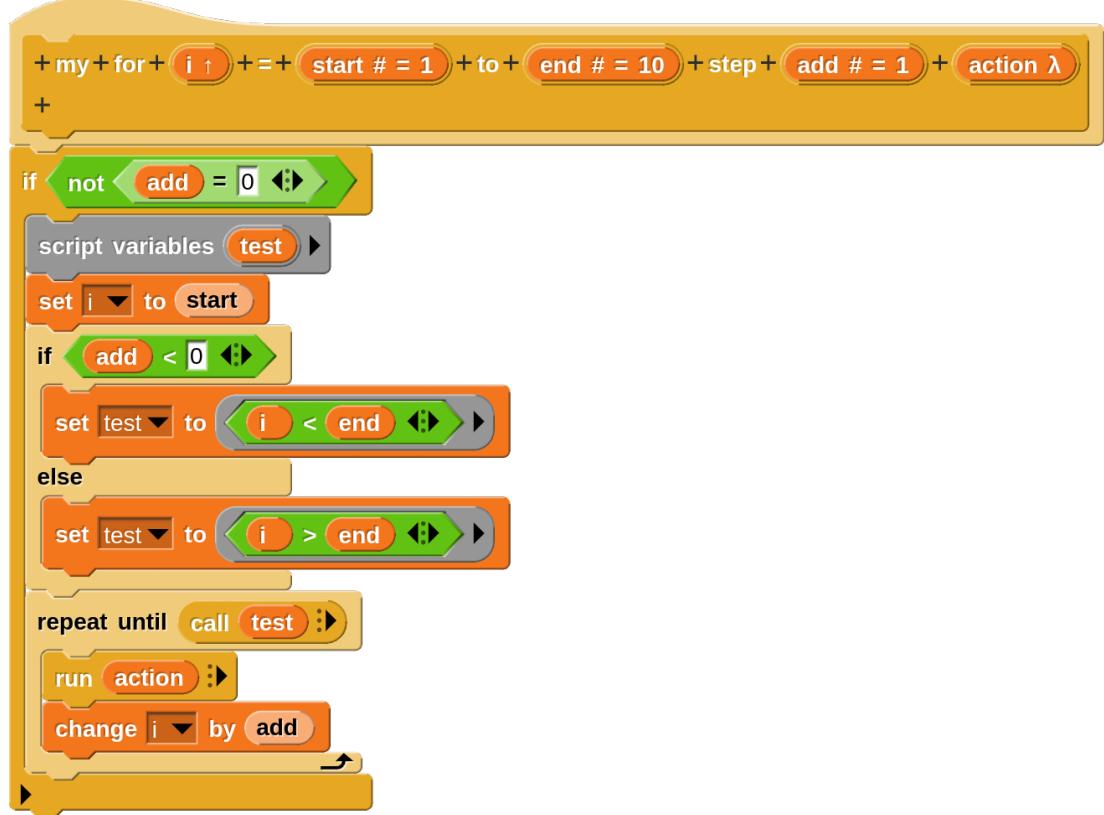
変数 action に特別なマークが付きました。

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。

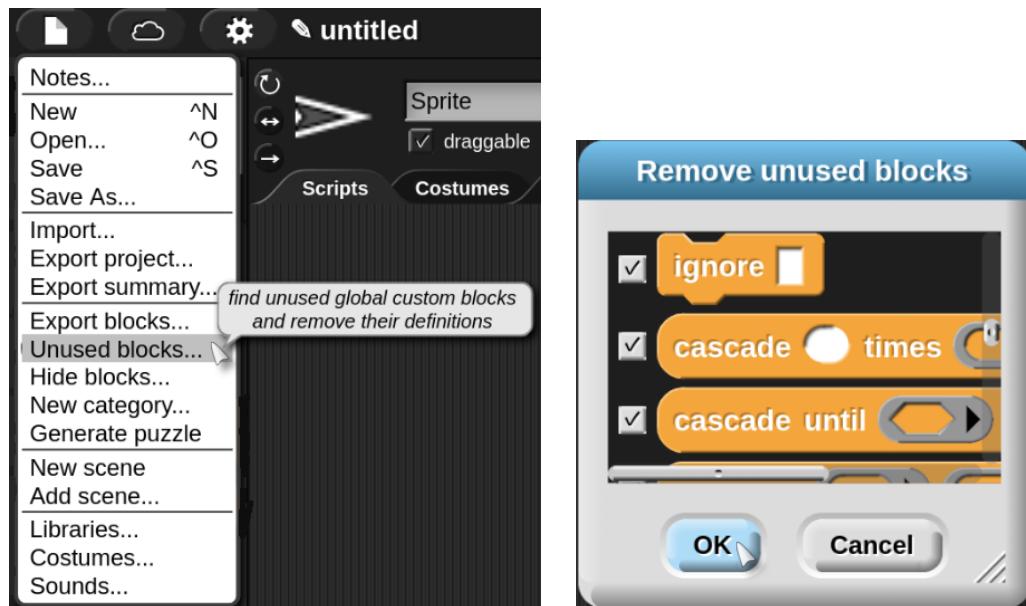
 で実験用のスクリプト  を入れ替えます。

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。

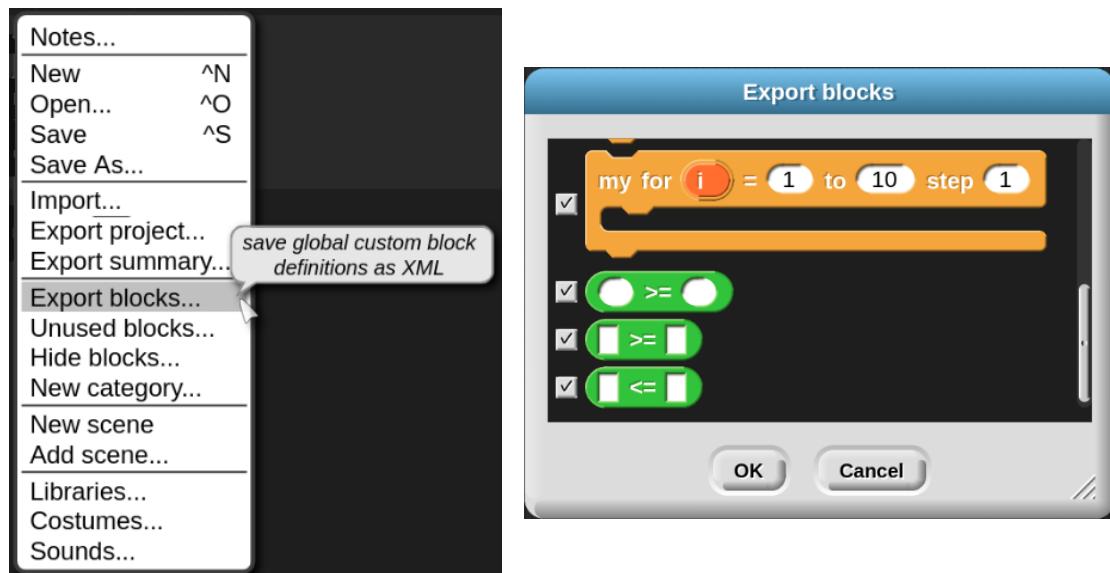




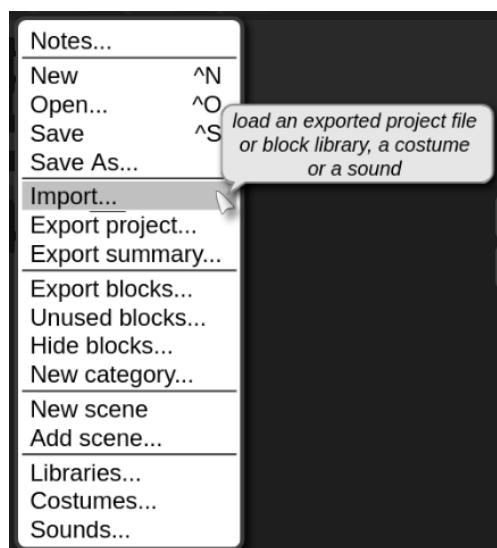
ところで、Libraries... からブロック定義をインポートしてプロジェクトを作成した場合に、普通に保存すると未使用のブロック定義も含んだファイルになります。プロジェクトを公開する場合は、次のようにして未使用のブロック定義を削除してファイルの容量を小さくしたほうがいいかもしれません。



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。



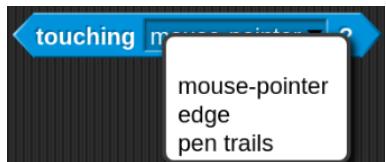
OK をクリックすると、ダウンロードフォルダに「????? blocks.xml」というファイル名で保存されます。????? のところにはプロジェクト名又は untitled が入ります。適宜リネームしてください。これを利用する時は、次のようにしてインポートすれば使用できるようになります。



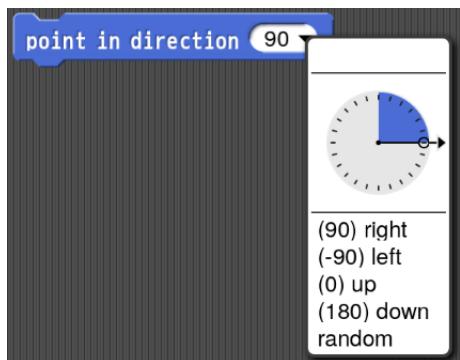
7 ブロック定義について

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。



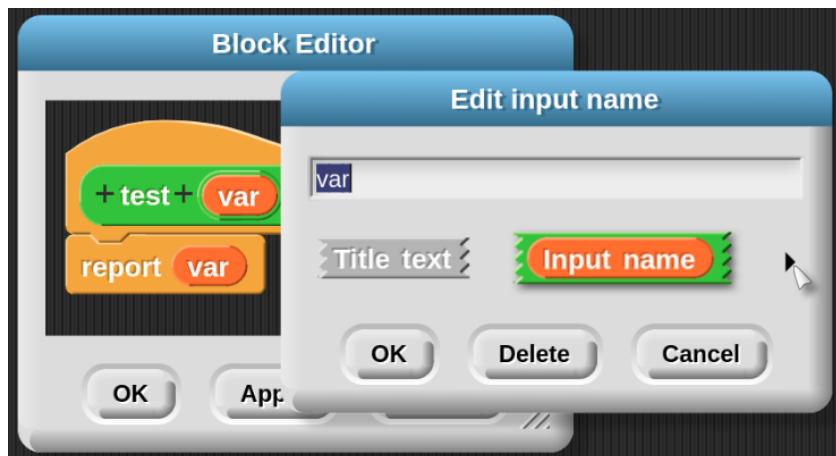
入力スロットに値をキーボードから入力することはできなくなっています。これは、後で出てくる read-only のオプションが指定されているためです。



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。

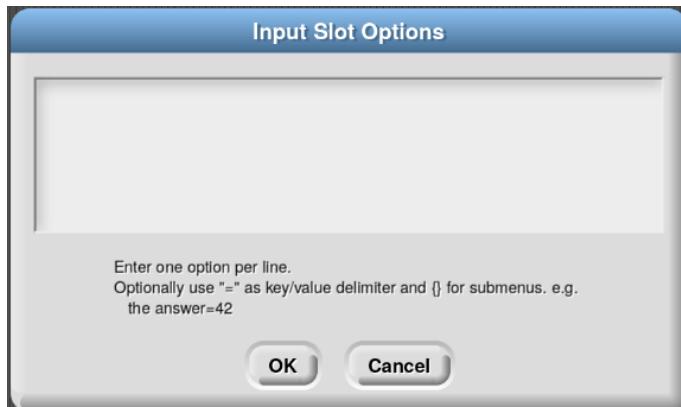


プルダウン入力を行うには、Input name の設定ダイアログを開きます。

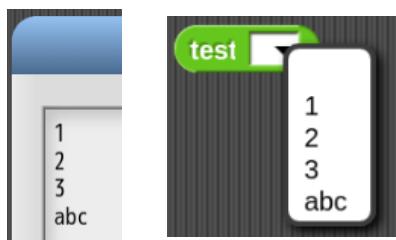
の var をクリックすると Edit input name のダイアログが開きますから、Input name の右側にある三角をクリックします。これで、大きな Edit input name のダイアログが開きます。ここの暗い灰色の領域で右クリックします。すると、このようなメニューが表示されます。



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。



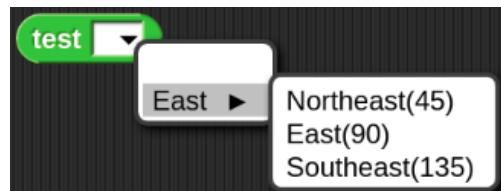
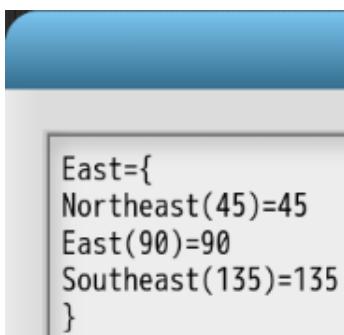
ここに各行にオプションを入れてプルダウンメニューを作っていきます。
左のように設定して、ブロックエディターを `Apply` すると、右のような結果になります。



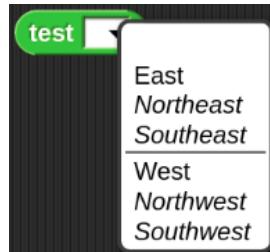
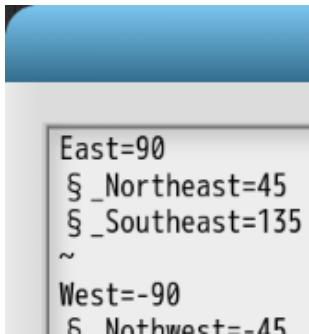
設定したものがそのまま表示され、クリックするとそれが入力値になります。
次のように、「=」で値を設定すると、メニューには「=」の左側の項目が表示されますが、クリックされると「=」の右側にある項目が入力値になります。



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。



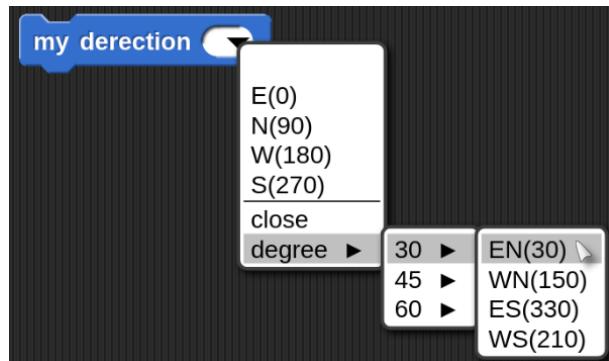
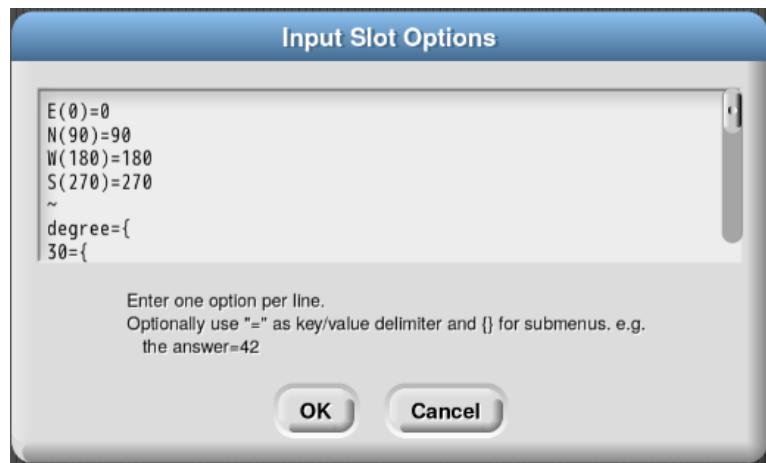
項目の前に、「 §_ 」セクション記号とアンダースコア（アンダーバー）を置くと、シフトキーを押しながらクリックしないとプルダウンメニューに表示されなくなります。因みに、§ 記号は Windows OS の場合だと、Alt キーを押しながらテンキーから（通常の数字キーではなく）0167 と入力、Linux OS の場合だと、[Ctrl]+[Shift]+[u] を押してから、a7 を入力して [Shift+Space] で出すことができます。a7 は 0167 の十六進数コードです。OS を問わず、日本語入力モードで「せくしょん」または「きごう」と入力して変換して出すこともできます。

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。

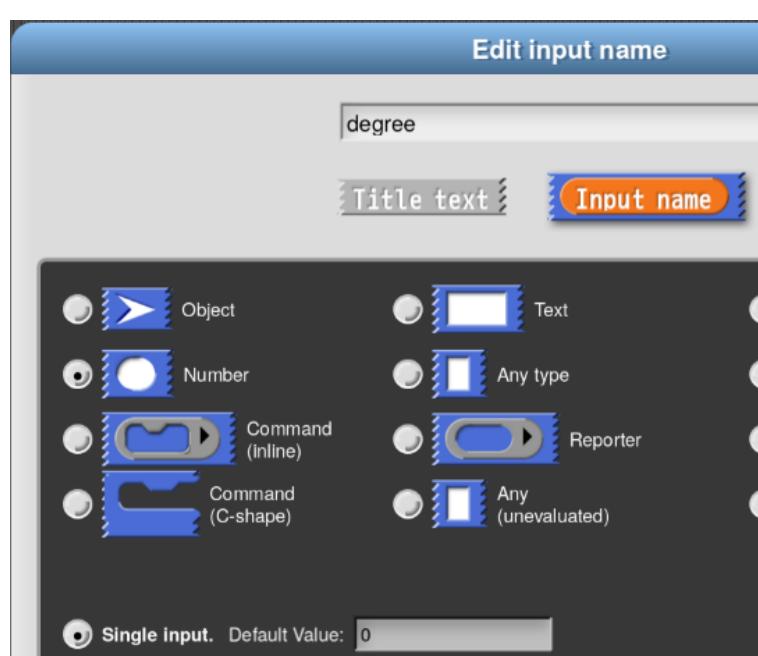
```

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
}
}

```

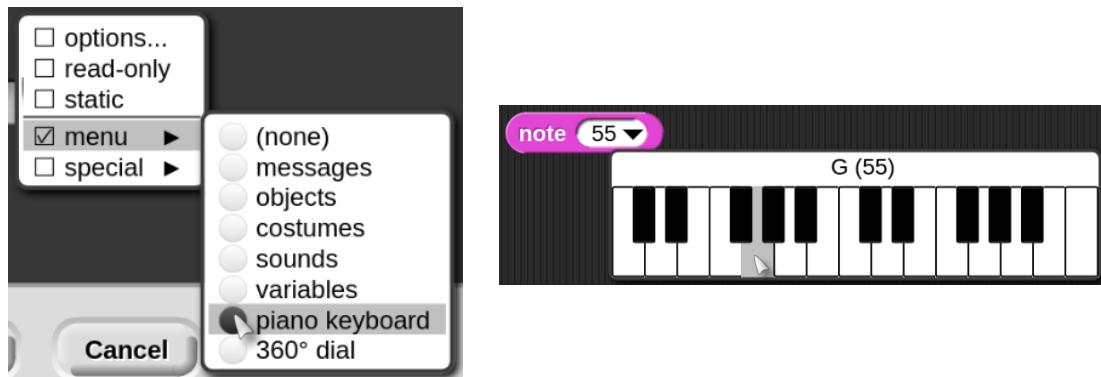


degree から、EN, WN, ES, WS のサブサブメニューを作っています。あくまで機能説明の作例ですので、あまり意味のあるものではありません。





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	↔ arrowLeftRightThin
▶ pointRight	brush	↓ arrowDown
▶ stepForward	✓ tick	▷ arrowDownOutline
⚙ gears	checkbox	↓ arrowDownThin
▶ gearPartial	rectangle	→ arrowRight
⚙ gearBig	rectangleSolid	▷ arrowRightOutline
▶ file	circle	→ arrowRightThin
▶ fullScreen	circleSolid	robot
▶ grow	ellipse	🔍 magnifyingGlass
▶ normalScreen	line	🔍 magnifierOutline
▶ shrink	+	selection
▶ smallStage	cross	○ polygon
▶ normalStage	crosshairs	○ closedBrush
▶ turtle	paintbucket	♫ notes
▶ turtleOutline	eraser	📷 camera
▶ stage	pipette	📍 location
▶ pause	speechBubble	❗ footprints
▶ flag	speechBubbleOutline	⌨ keyboard
● octagon	↗ loop	⌨ keyboardFilled
cloud	⬅ turnBack	🌐 globe
cloudGradient	➡ turnForward	🌐 globeBig
cloudOutline	↑ arrowUp	☰ list
turnRight	↑ arrowUpOutline	☰ listNarrow
turnLeft	↑ arrowUpThin	⋮ verticalEllipsis
turnAround	↓ arrowUpDownThin	◀ flipVertical
storage	← arrowLeft	▲ flipHorizontal
poster	⬅ arrowLeftOutline	🗑 trash
	← arrowLeftThin	🗑 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 **\$turtle-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「\$」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

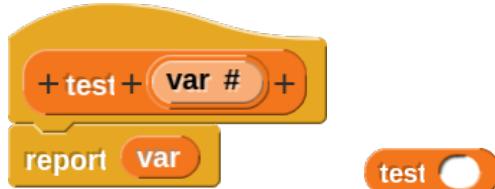
7.3.1 Reporter 型

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。



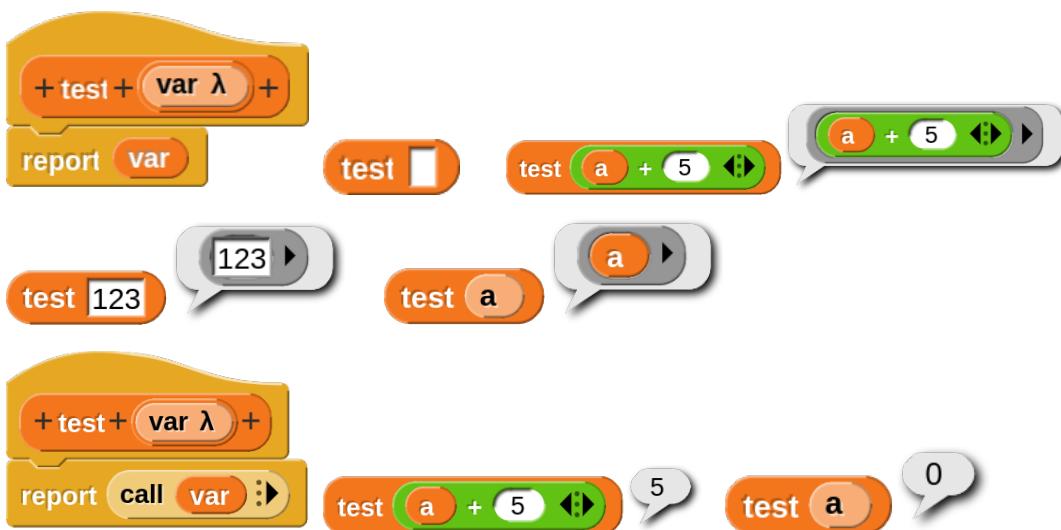
ただし、変数単体だと次の Any(unevaluated) とは違い `test a` とリング付きではなくなってしまいます。 $(a = 0)$ とセットされているとします。) その場合は手動でリングを付けてから入力スロットにドロップする必要があります。

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。



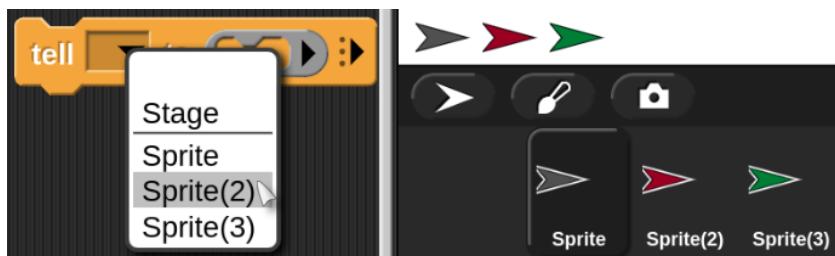
a にリングが付いていない状態なのでエラーになりました。

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。



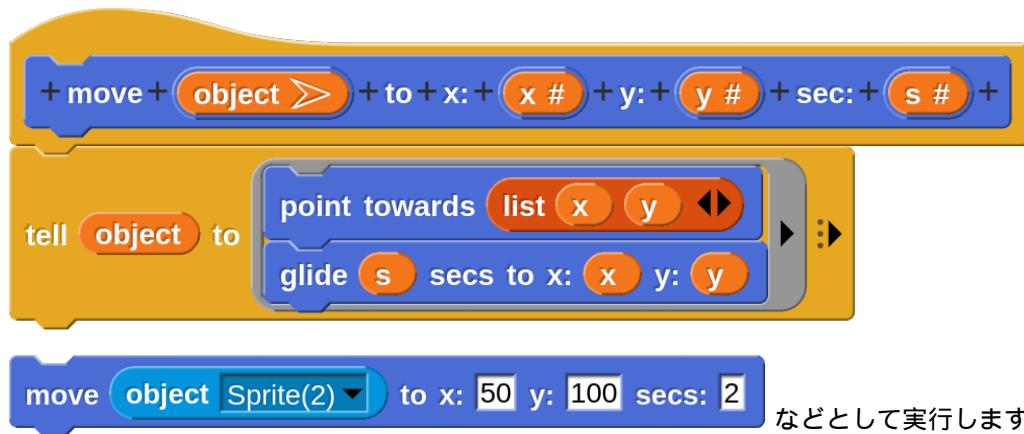
この入力スロットにオブジェクトを示す変数などをドロップすることもできます。 Sensing パレットにある Object リポーターブロックの入力メニューを開くと tell ブロックと同様に指定可能なオブジェクトを選択することができます。



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

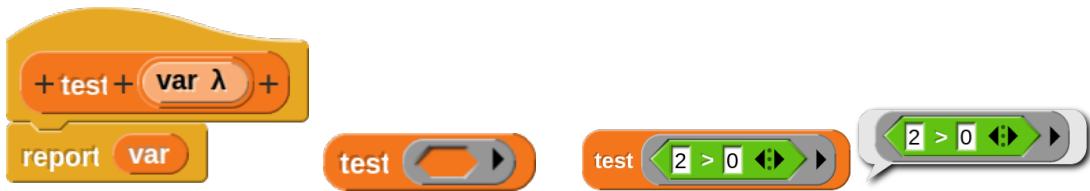
Predicate 型には、Boolean, Predicate, Boolean(unevaluated) があります。

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

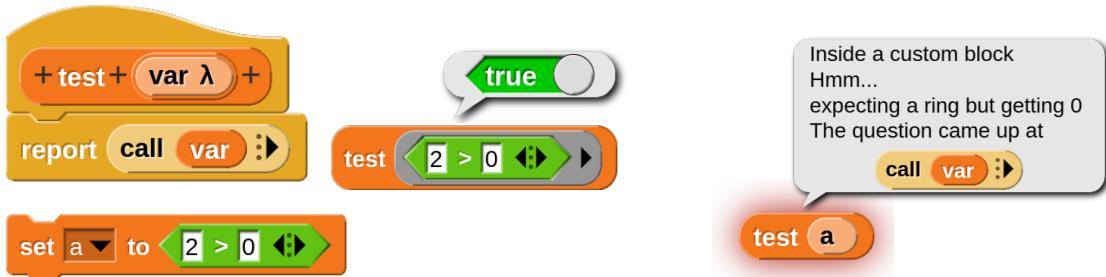
Boolean です。



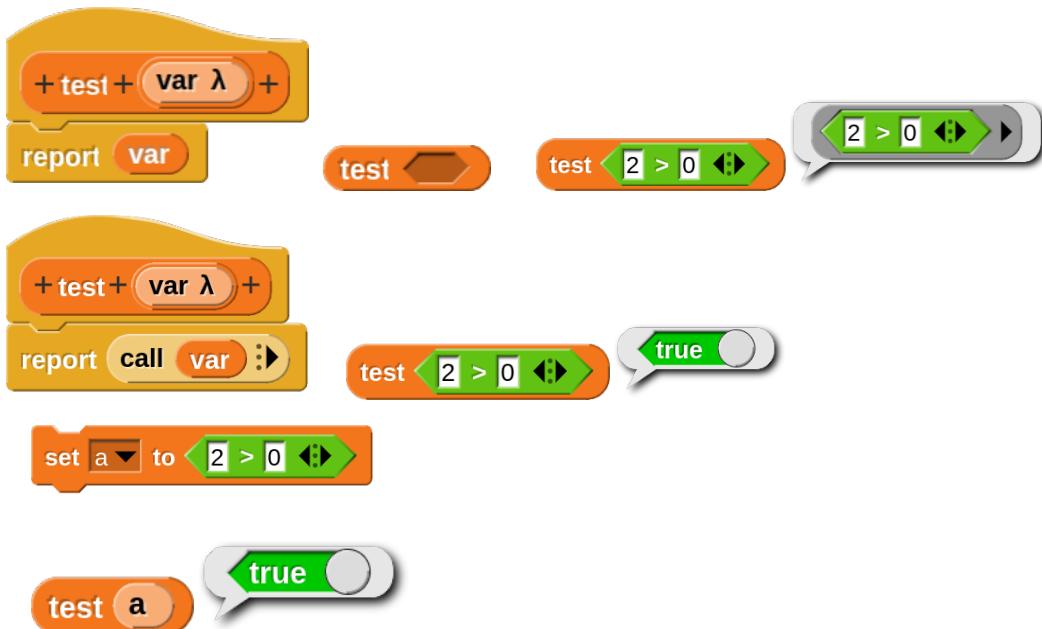
Predicate です。



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。評価には call ブロックが必要です。



7.3.3 Command 型

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

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

c 言語では、

```

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

```

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

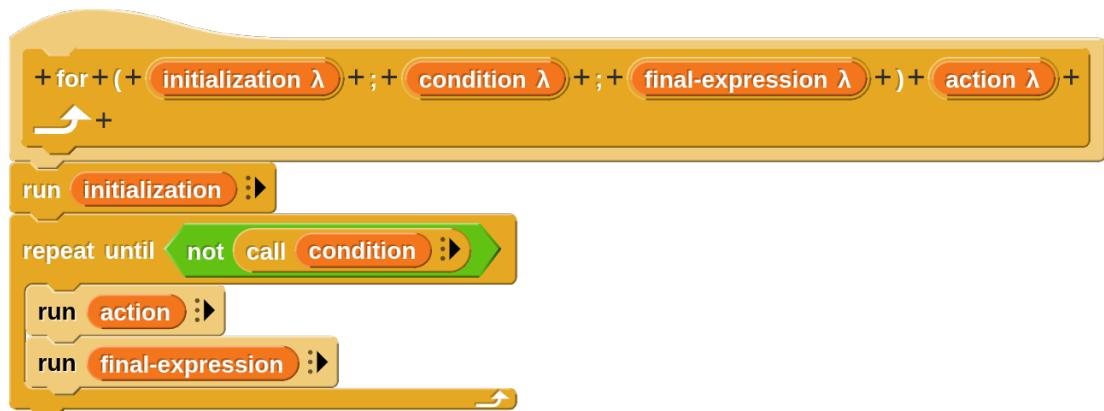
i = 0 の部分が初期設定で、初めに一回だけ実行されます。

i < 5 の部分が実行を続けるかどうかのテストをします。

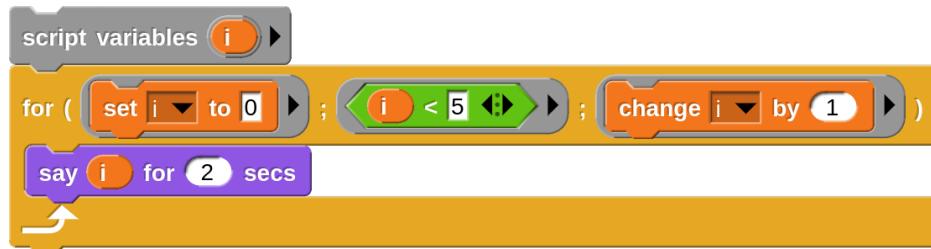
i++ の部分は次の繰り返しに進む前に実行されます。 i++ は i に 1 加算する処理をします。

{ } の内部がループの本体です。

以下が定義です。プロトタイプの末尾に text で Loop を入れて矢印を表記しています。定義自体は run や call に丸投げするだけなので my for よりもシンプルですね。

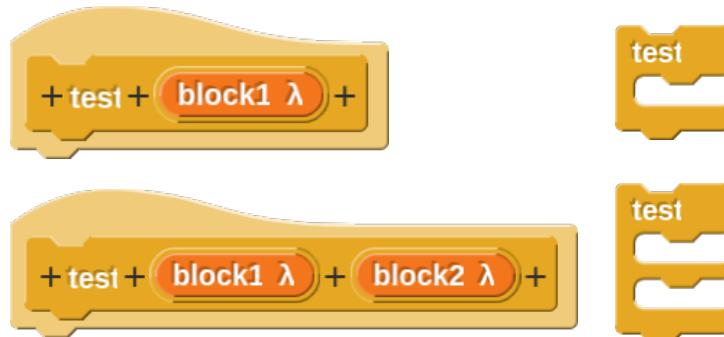


なお、initialization と final-expression は Command(inline) 型で、condition は Predicate 型です。action は Command (C-shape) 型 です。



for ループは C 型が 1 つでしたが、2 つにすると 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。



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理をします。case ブロックは追加できるようにし、default 処理も指定できます。
case ブロックは単に指定された値と実行ブロックの対をリストとしてリポートするだけです。



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。



default は Command(inline) 型です。

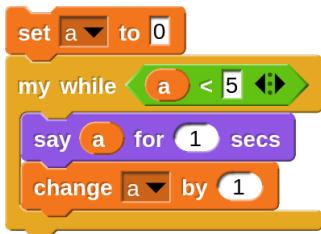
名古屋文理大学 小橋 一秀 先生が公開されている

【 05. SNAP!でオブジェクト指向 (4/4) 】

「継続を利用した switch case ブロックの作成例」
を参考にさせて頂きました。

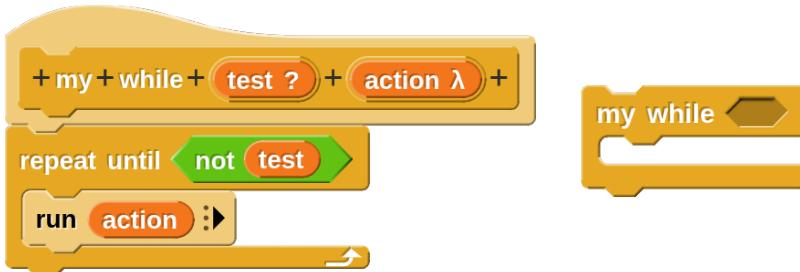


while ループを作成しながら、Predicate 型の補足説明をします。



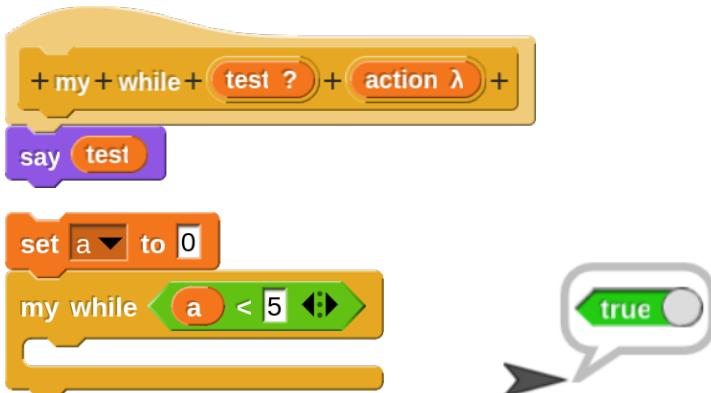
これは、 $a < 5$ のテストが true ならば、my while 内のループを繰り返します。

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



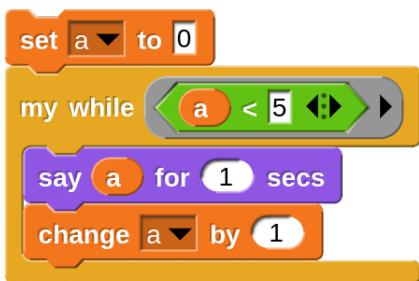
しかし、これを実行するとテストがうまくいかないようで、終わらなくなります。

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 my while が実行された時点の $a < 5$ の値 true ということです。したがって、常に true となり終わりません。

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。



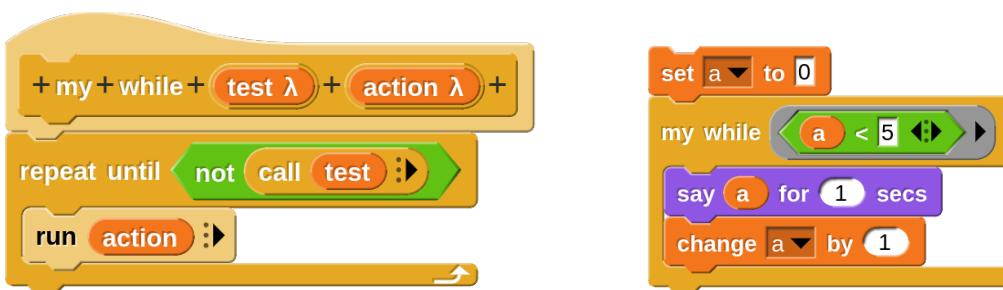
受け取った式から真理値を得るために call を使う必要があります。



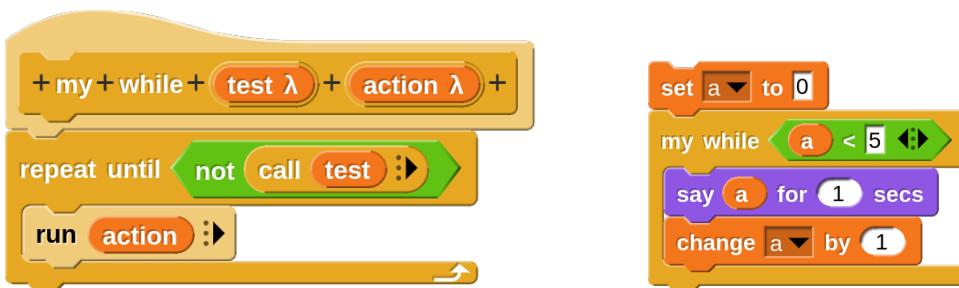
今度はうまくいきました。

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

になるため、式



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。



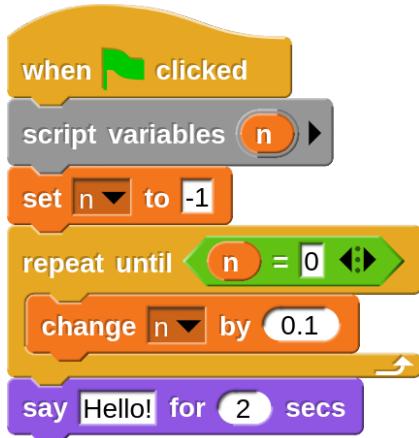
8 その他

8.1 デバッグ

Snap! にはデバッグの機能が用意されています。

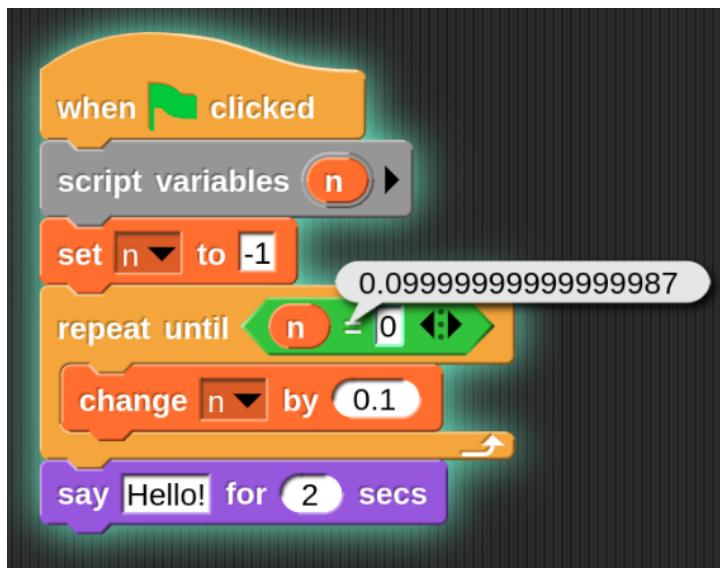
 ブロックにより指定の箇所でプログラムの実行を一時停止させることができます。  の操作でプログラムの実行をステップ実行にしたり、実行スピードを遅くしたりできます。デバッグ中は実行中のブロックをハイライトしてくれます。変数などの値もリポートしてくれます。  のクリックで一時停止した実行を再開できます。

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。
 のように、スライダーを左端にするとステップ実行になります。  のクリックで一時停止した実行を再開できます。

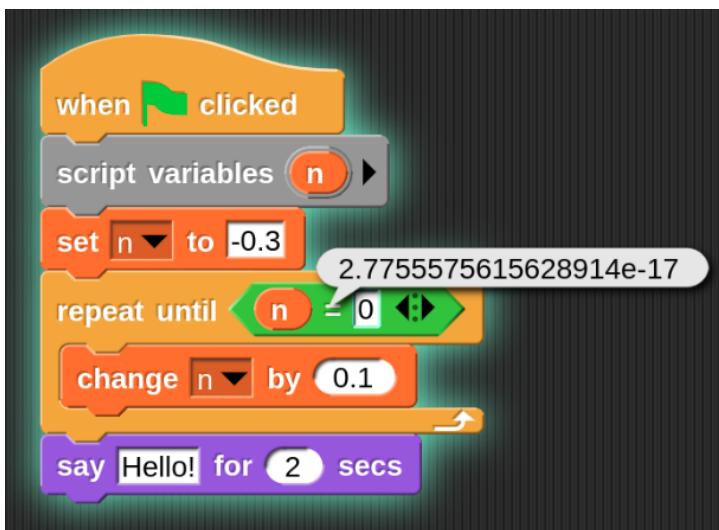
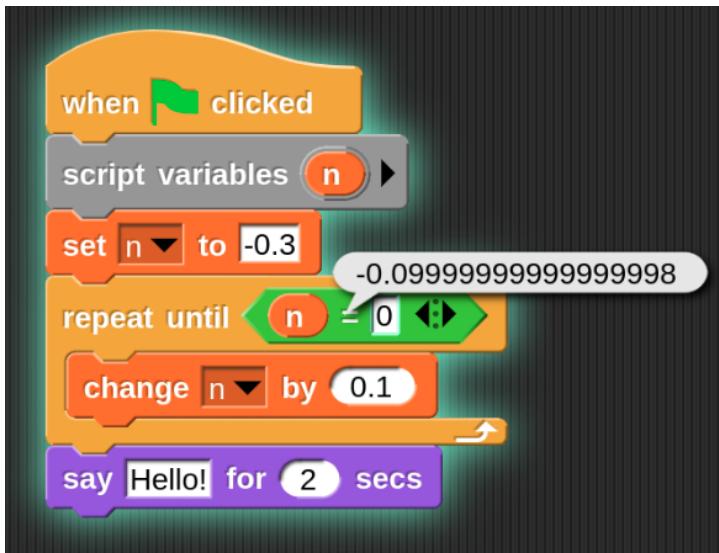
ただし、 のボタンで実行すると、変数などの値をリポートしてくれません。直接スクリプトをクリックして実行してください。



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。

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

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

8.2 入力スロットへのリスト指定

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

 この左右の三角のところにリストを持ってくると、
 のように警告のようなものがでますが構わずドロップすると、
 のように sum になります。

リスト操作として実行すると次のようになります。

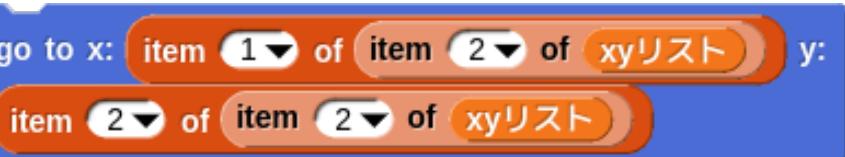
 6

 120

xy 座標を次のようにリストにした場合、



 で x と y の値を指定するとすこし面倒です。

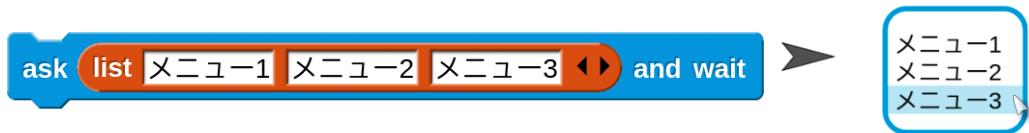


 で x と y の値をリストで指定することができます。

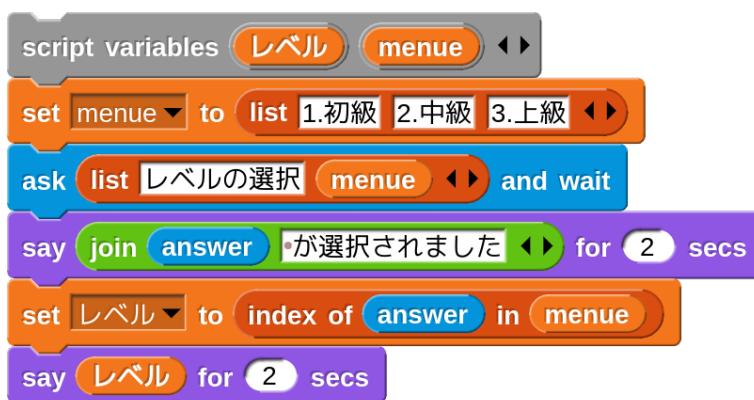
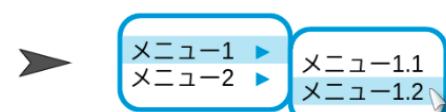
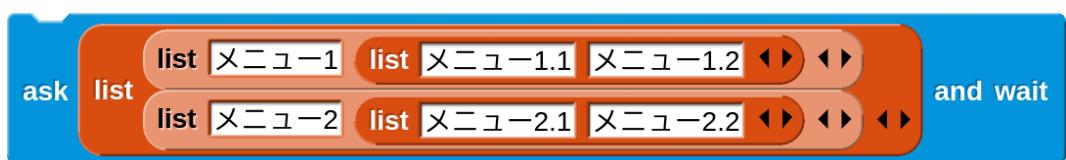
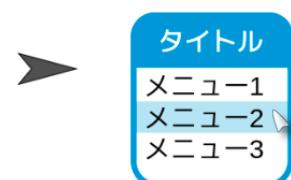


8.3 ask

`ask what's your name? and wait` ブロックの入力スロットにリストを入れてやると、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。



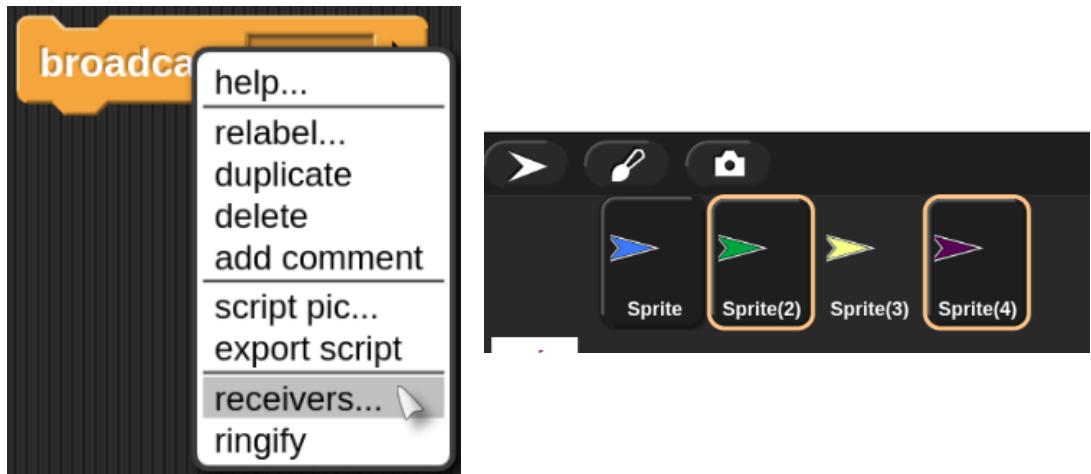
リストを組み合わせるとタイトル(説明文)を付けたり、サブメニュー構造にすることもできます。



8.4 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。



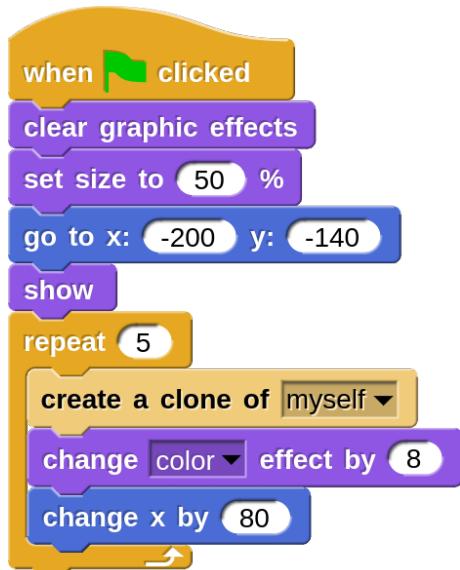
8.5 クローン

Snap! には、テンポラリクローンとパーマネントクローンがあります。

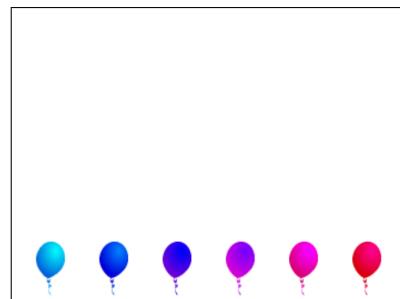
8.5.1 テンポラリクローン

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。

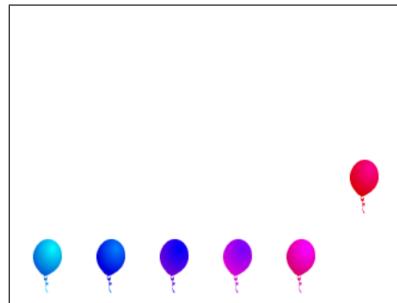


```

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?
      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] 内で行うことでもできます。

```

when green flag clicked
  hide
  clear graphic effects
  set size to 50 %
  go to x: -200 y: -140
  set [balloon v] to [list]
repeat (6)
  add a new clone of [myself] to [balloon]
  change [color v] effect by 8
  change x by 80
forever
  for each [item] in [balloon]
    tell [item] to
      for [i] = 0 to 180
        change y by [cos of i]
    wait (5) secs

```

when I start as a clone

show

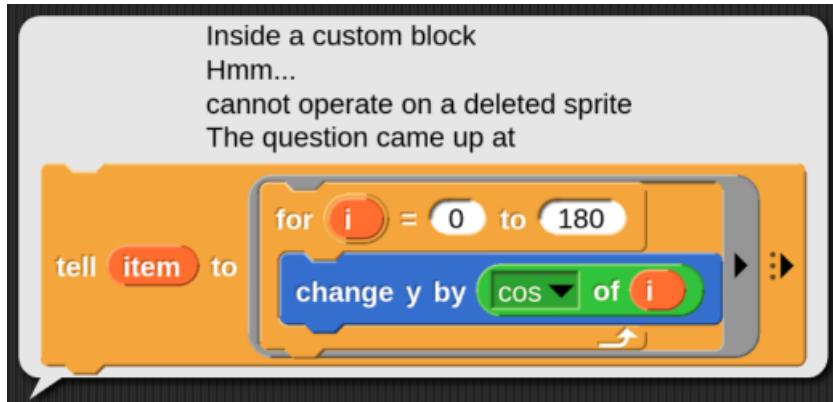
Snap! では作成したクローンをリストなどの変数に入れることができるので、それを使って対象を指定して操作することができます。

```

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

```

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



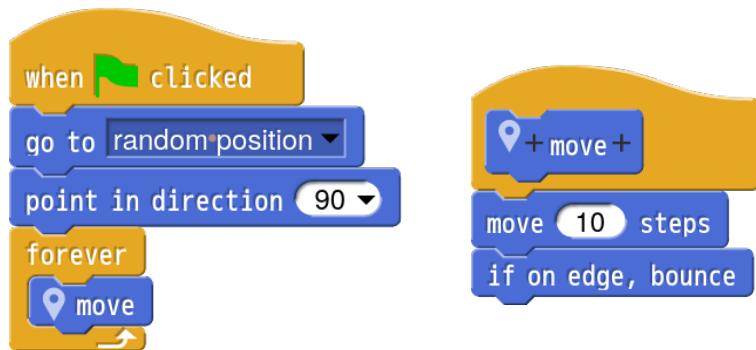
変数やリストに格納されたクローンが delete されているかどうかをテストするためのブロックは定義されていないようです。これに対処する一つの方法として、スプライト変数  にクローン番号を記憶させて、そのインデックスによって削除されたこと false を balloon リストに記録します。(クローン 1 は 1、クローン 2 は 2、... の値の変数 id を持ります。) id 番号をリスト内の位置にしているので、リストから要素を削除することはできません。



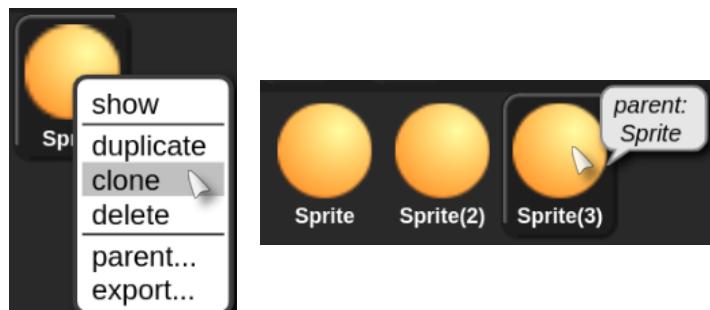
8.5.2 パーマネントクローン

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

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



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



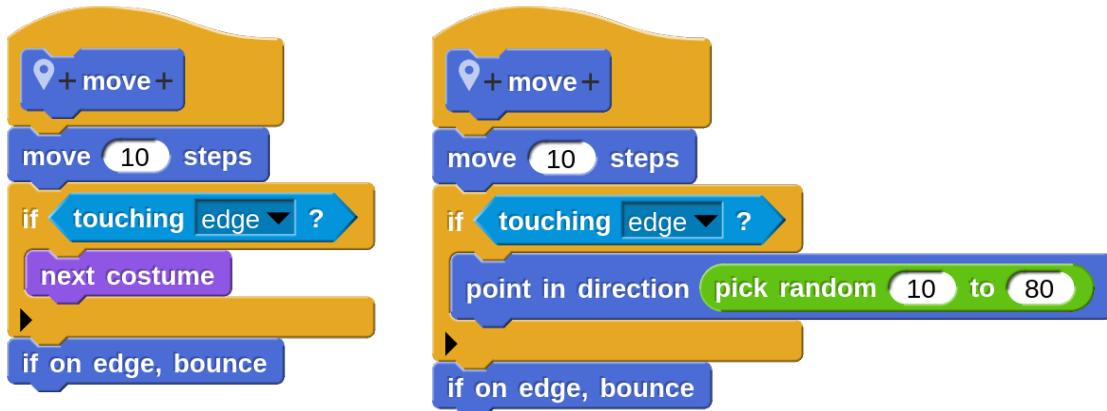
クローンである `Sprite(2)`, `Sprite(3)` には、`Sprite` と同じスクリプトがコピーされています。親である `Sprite` のスクリプトを変更すると、子である `Sprite(2)`, `Sprite(3)` のスクリプトに反映されます。例えば、サイズを 50 などと変えてみてください。

スクリプトエリアで右クリックすると右図のように `inherited` がオンになっています。この状態だと子側も親側のスクリプトと同じになります。しかし、子側でスクリプトを変更してしまうと `inherited` がオフになり、親側の変更が反映されなくなってしまいます。その場合は `inherited` をオフにしてやれば、親側と同じスクリプトに戻ります。

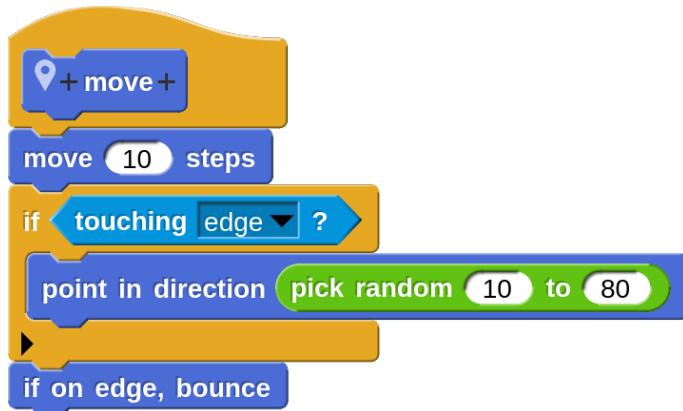


クローンを作成した時に親側のユーザー定義ブロックも子側にコピーされています。しかし、この `move` ブロックは `for this sprite only` で作成されているので、親側で変更しても子側は変わりません。子側で変更しても `inherited` の状態は変わりません。次のように子側の `move` 定義を変更してください。各スプライトで同じ名前の `move` ブロックですが、それぞれ違った動作をさせることができます。

Sprite(2)



Sprite(3)

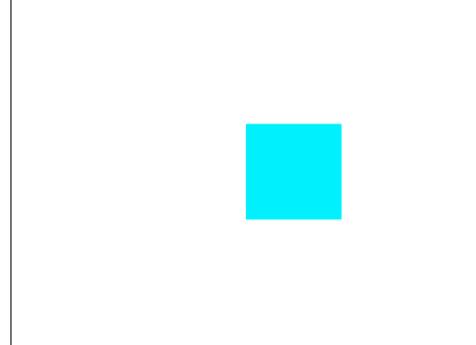
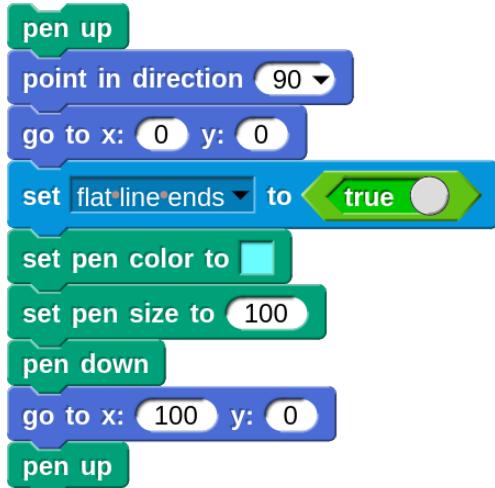


8.6 flat line ends

Sensing のパレットに `set video capture to [video camera icon]` 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。

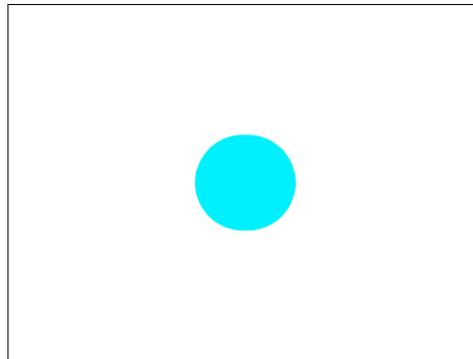
(六角形のスロット部分をクリックして、`set flat-line-ends to [checkmark]` や
`set flat-line-ends to [x]` で指定することもできます。)



```

pen up
point in direction 90 ▾
go to x: 0 y: 0
set flatline•ends to false
set pen color to light blue
set pen size to 100
pen down
go to x: 5 y: 0
pen up

```



```

hide
point in direction 90 ▾
reset timer
forever
clear
set pen color to black
go to x: -40 y: 0
write join SEC: floor of timer size 20

```

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することができます。
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。

```

hide
point in direction 90 ▾
go to x: -40 y: 0
write SEC: size 20
reset timer
forever
area clean
set pen color to black
go to x: 0 y: 0
write floor of timer size 20

```

```

+ area + clean +
warp
pen up
go to x: 0 y: 8
set flatline•ends to true
set pen color to light blue
set pen size to 25
pen down
go to x: 50 y: 8
pen up

```

set pen color to [light blue] 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。

8.7 anchor アンカー

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

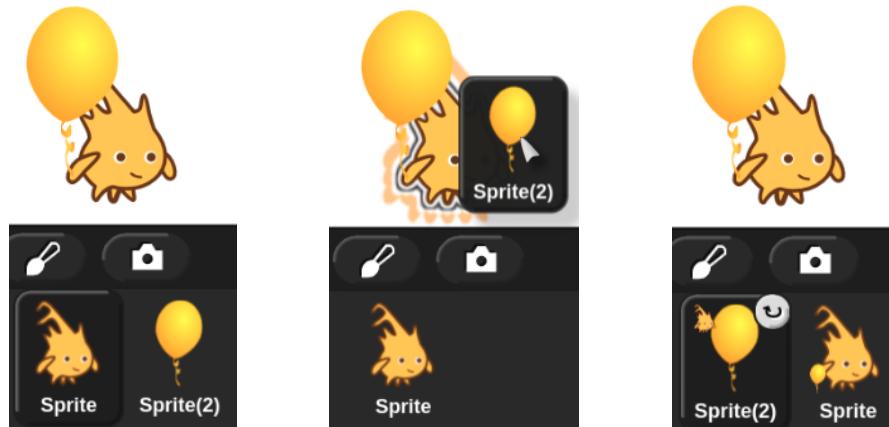


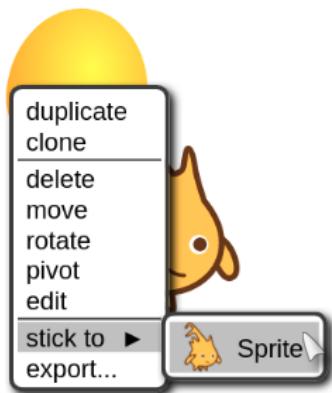
alonzo というコスチュームの手に風船を持たせてみます。風船は alonzo に持たれてはいますが、独自の動きをします。
alonzo と balloon のコスチュームをそれぞれのスプライトに設定してください。



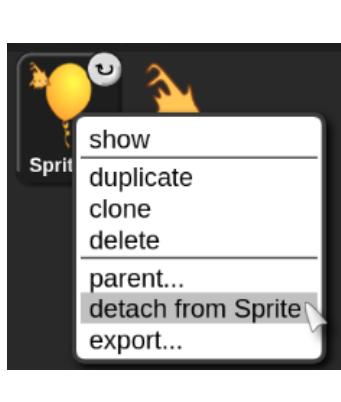
balloon をエディターで開いて、動きや回転の中心を糸の端にします。

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





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



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

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

8.8 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` 20

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

変数を使って演算をしてそれを返すこともできます。

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

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。

call **JavaScript function** ([list $\blacktriangleleft \triangleright$] { return list; })
with inputs numbers from 3 to 1 $\blacktriangleleft \triangleright$

1 3
2 2
3 1
+ length: 3

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。

call **JavaScript function** ([list $\blacktriangleleft \triangleright$] { list.sort(function(a,b){ return a-b}); return list; })
with inputs numbers from 3 to 1 $\blacktriangleleft \triangleright$

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。

call **JavaScript function** ([list $\blacktriangleleft \triangleright$] { var l = list.toArray(); l.sort(function(a,b){ return a-b}); return l; })
with inputs numbers from 3 to 1 $\blacktriangleleft \triangleright$

1 1
2 2
3 3
+ length: 3

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。

call **JavaScript function** ([list $\blacktriangleleft \triangleright$] { list.toArray().sort(function(a,b){ return b-a}); return list; })
with inputs numbers from 3 to 1 $\blacktriangleleft \triangleright$

1 3
2 2
3 1
+ length: 3

JavaScript function の使用例としてソートをやってみましたが、APL ライブラリ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 n_1, n_2 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。

n_1	n_2	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。 n_1 と n_2 が整数であることを前提にしています。



`1 AND 0` → 0
`1 AND 1` → 1
”`&`” の部分を ”`|`” 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。

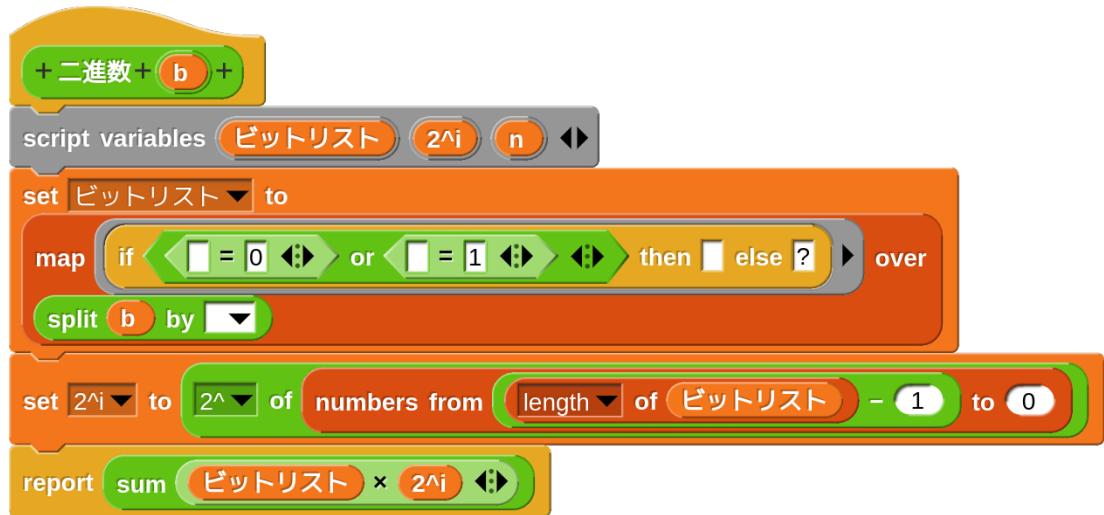


`NOT 0` → -1
0 の NOT が -1 というのは分かりづらいかもしれません。数値の桁数を 4 ビット (ニブル) に限定してみます。すると表せる数は 0000 ~ 1111 (十進数では 0 ~ 15) になります。0000 の NOT なので 1111 になります。二進数では、最上位ビットをサインビット (符号ビット) として使用することで負数も扱えるようにしています。その場合、1111 は最上位ビットが 1 なので負数です。1111 + 0001 を行うと桁上がりをしていて 4 ビットの範囲では 0000 になります。つまり、1111 は 1 に加えると 0 になる数である -1 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。整数を 64 ビットで表すならば、0 と -1 はそれぞれ 0 と 1 が 64 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、数値 $x(-1)$ の値を得ることができます。



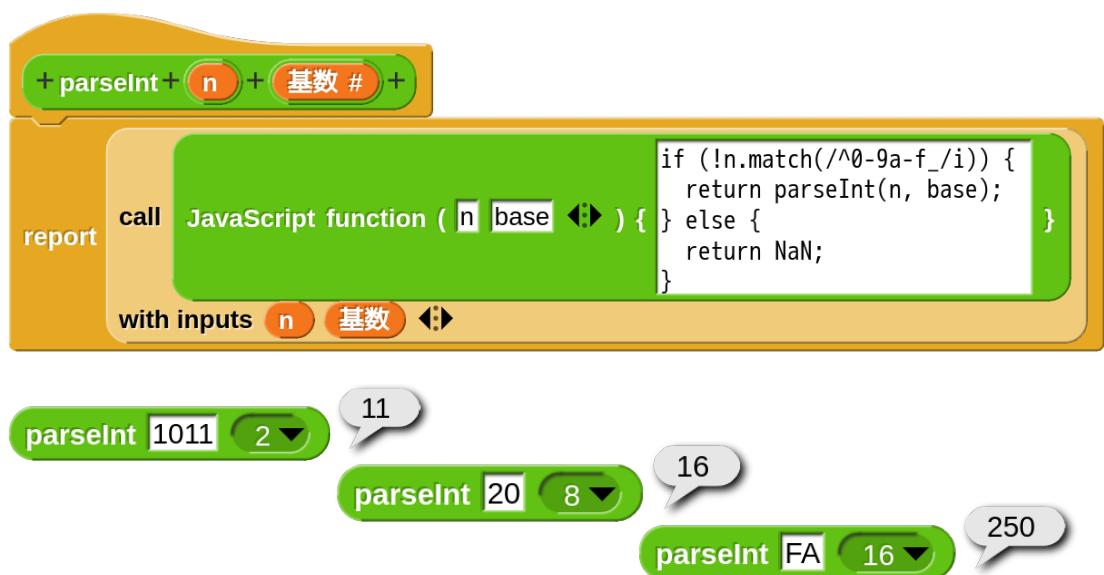
二進数で数値を指定するリポーターブロックを作成してみます。b の入力のタイプは text です。
split で指定するのは空文字です。(デフォルトの空白を削除します。)



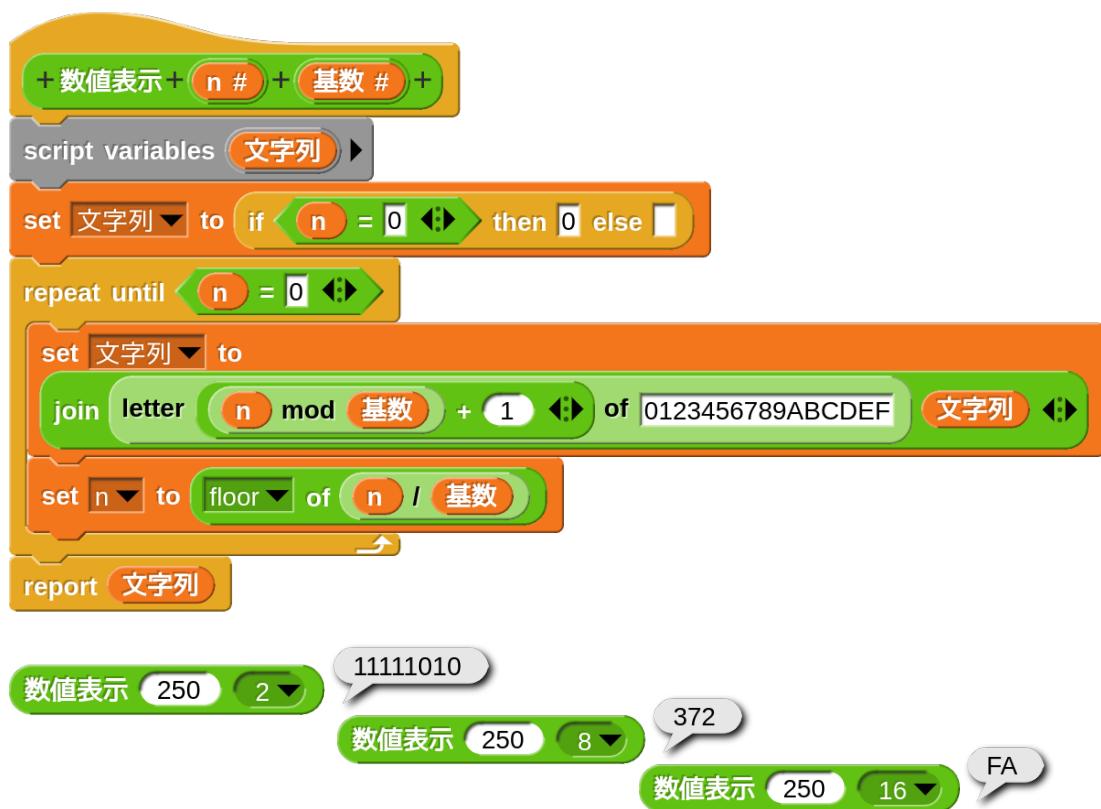
二進数では 0 と 1 しか使わないので、それ以外は「?」に置き換えてエラーになるようにしています。



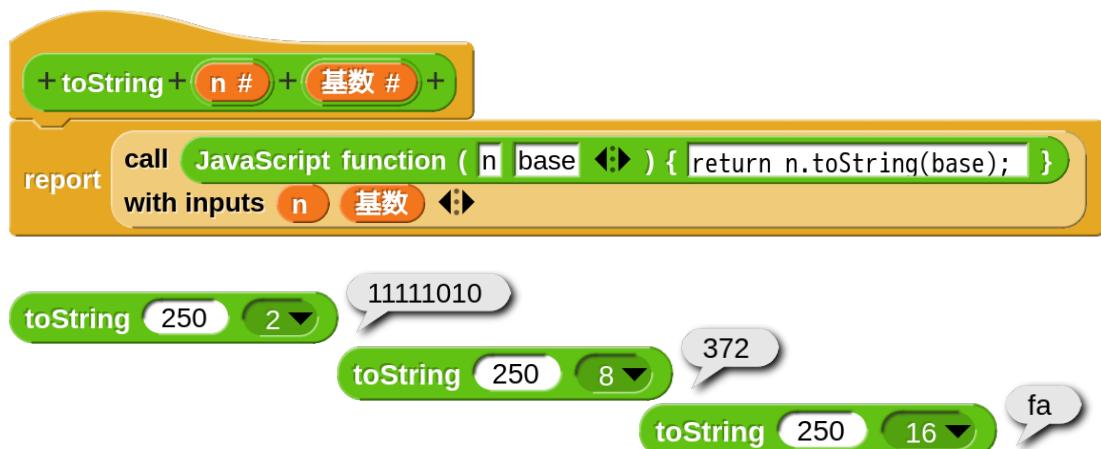
JavaScript には文字列を整数に変換する parseInt 関数があります。2 番目の引数で基数を指定できます。2 で二進数、16 で 16 進数の文字列からの変換になります。なお、「base」と「基数」が合っていませんが、JavaScript の引数は名前ではなく inputs のスロット位置に対応した値が設定されます。



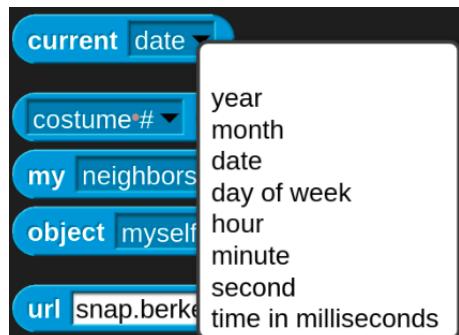
こうなると、数値を二進数表示するブロックも必要です。どうせなら基底を指定して文字列にする定義ブロックにしてみます。ただし、対応しているのは 2, 8, 10, 16 進数などです。



JavaScript では `toString` がそれをしてくれます。



8.9 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 衡表示にして、区切り文字を挿入するには次のようなスクリプトになります。これを変数にセットして、forever でループします。



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

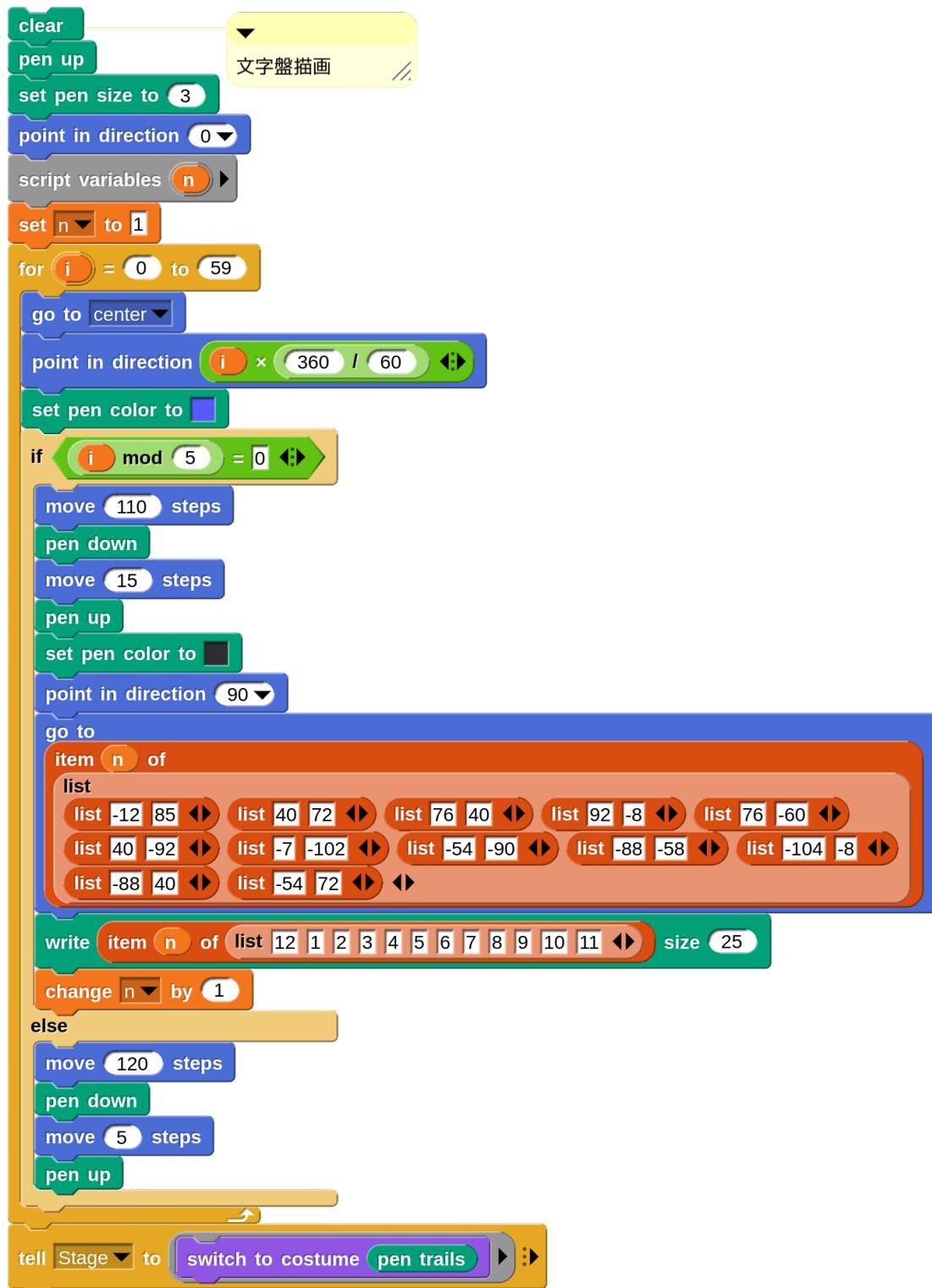


時針の角度です。 UTC 協定世界時を使用すると時差調整が必要になるので用いません。

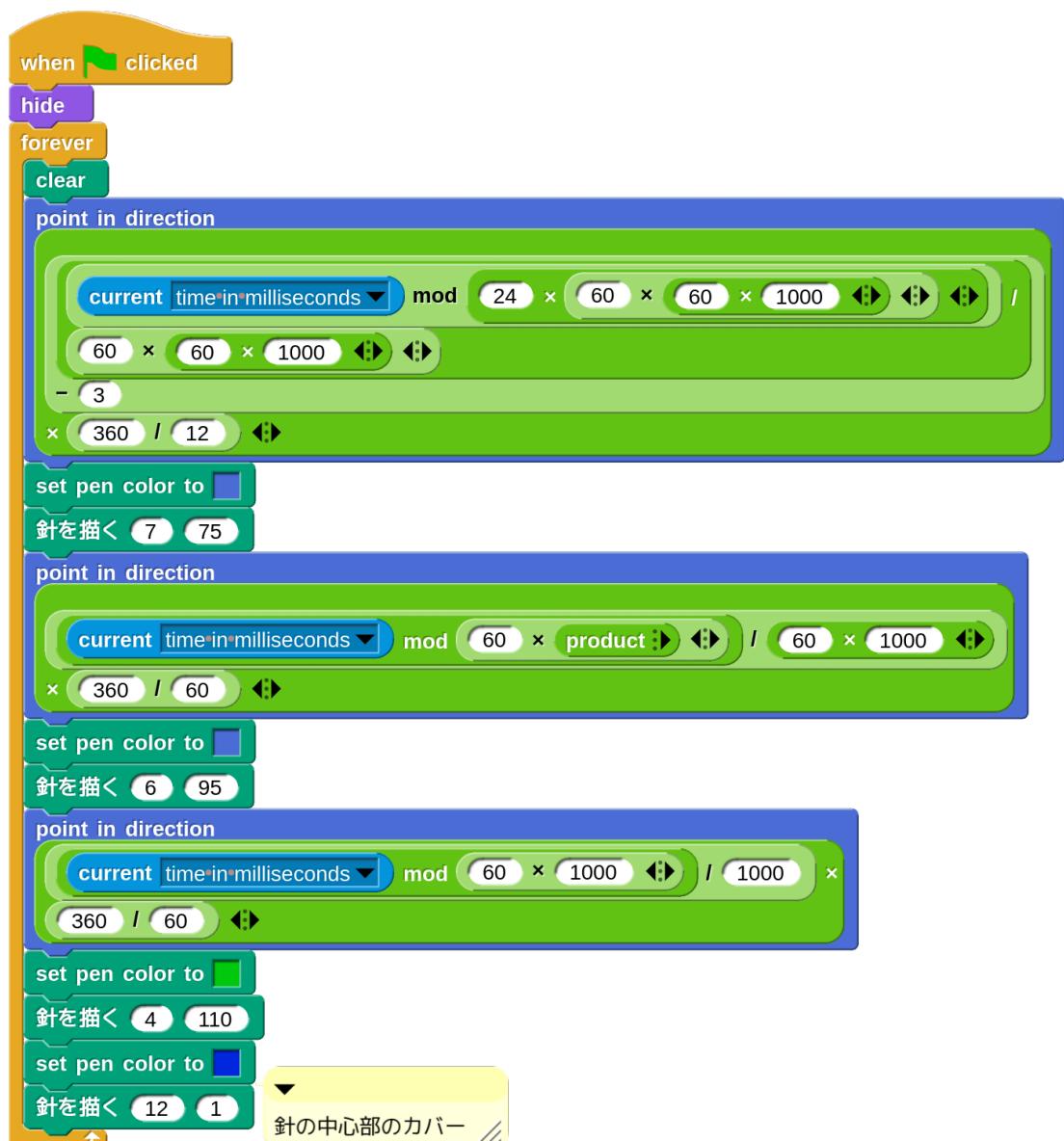
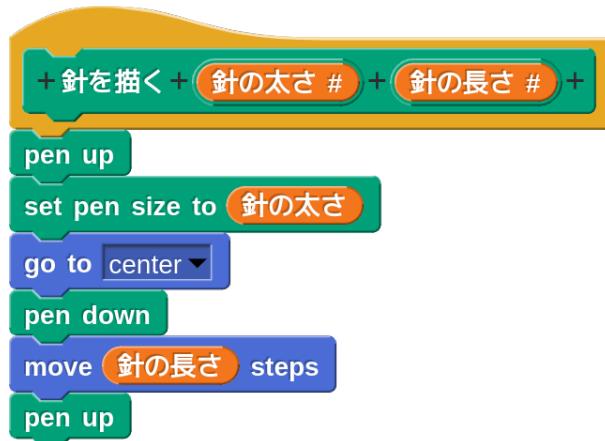


ペンで描く場合はこの逆の順序で描けば実際の時計と同じになります。

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。

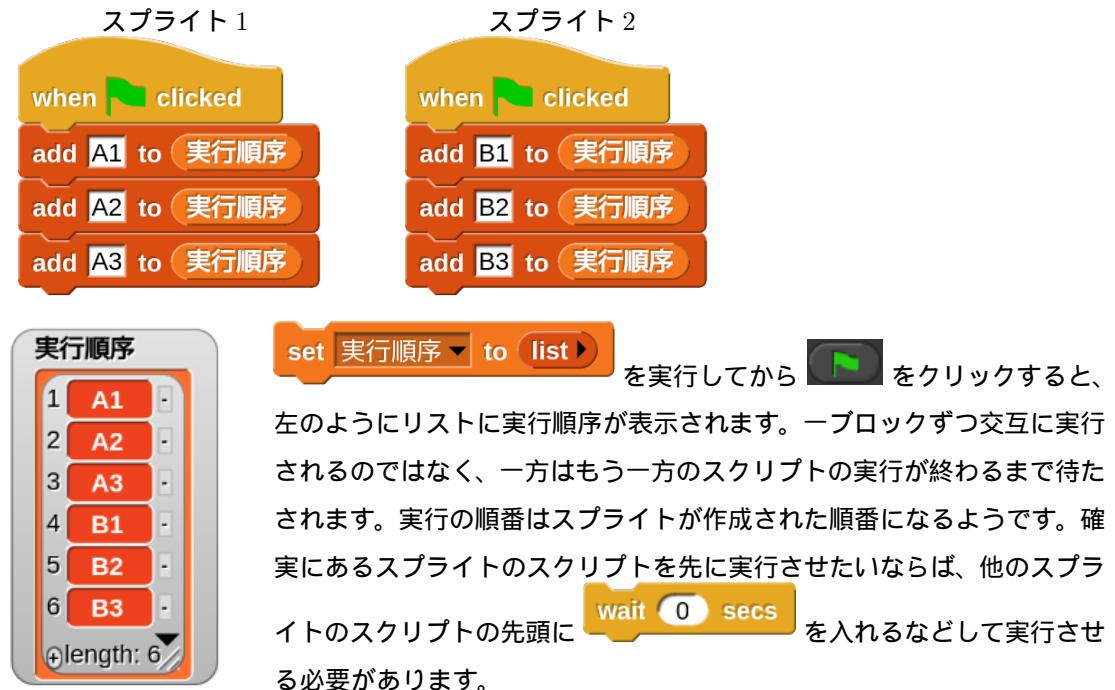


針を描くための定義ブロックです。



8.10 並列処理について

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



スプライト内に複数の **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 v i]) to [実行順序]
  add (join [A2 v i]) to [実行順序]
end

```

スプライト 2

```

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

```

実行順序

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

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

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。

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 v i]) to [実行順序]
    add (join [A2 v i]) to [実行順序]
  end
end

```

スプライト 2

```

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

```

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。

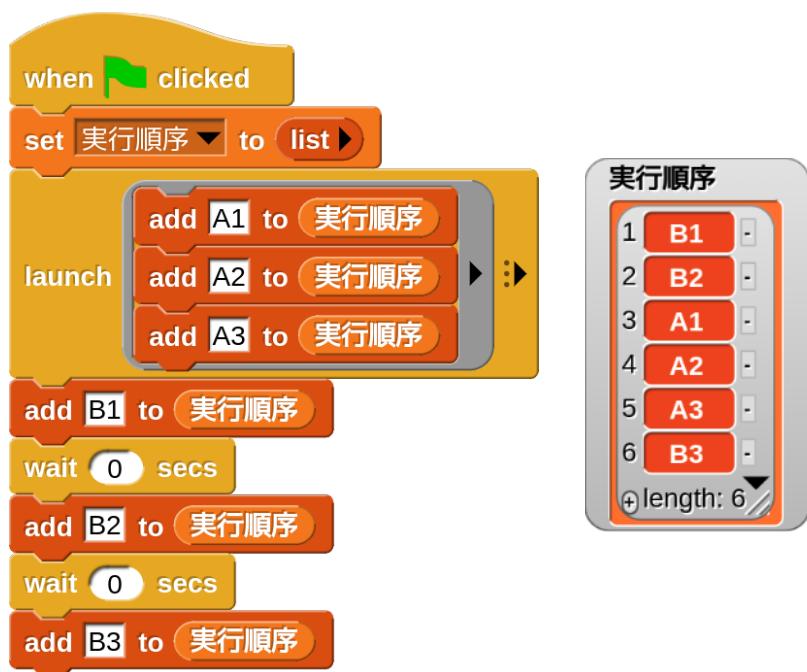
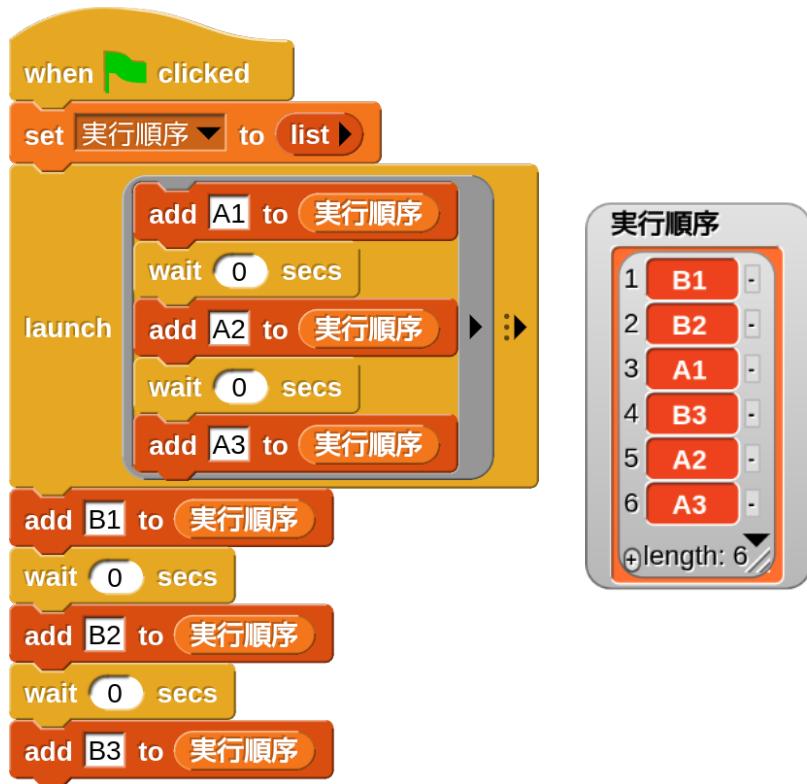
launch での動作です。



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

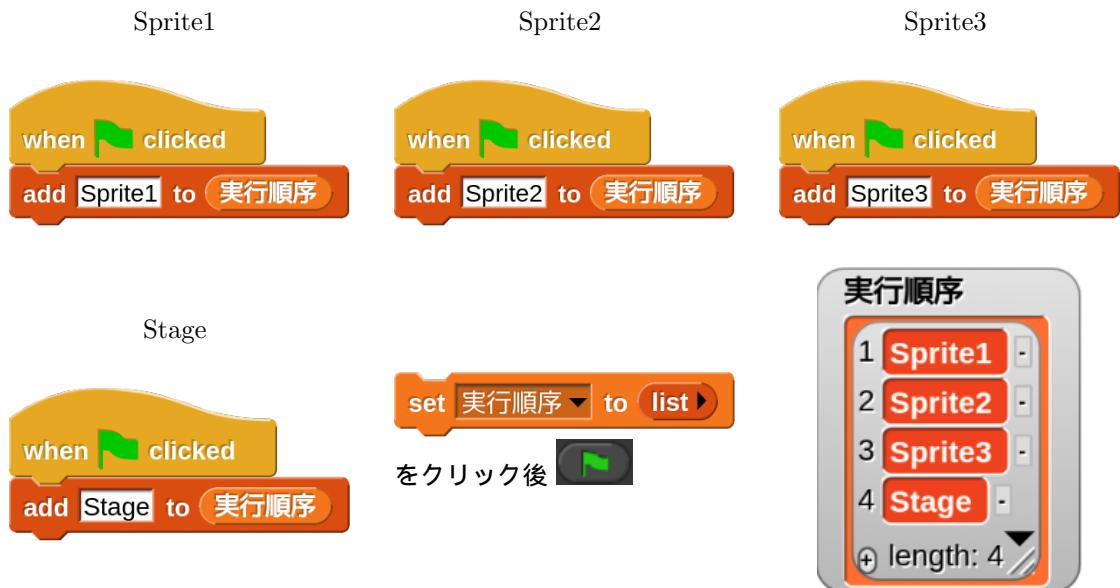


実行順序	
1	B1
2	B2
3	A1
4	B3
5	A2
6	A3
+ length: 6	



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。

9 再帰呼び出し

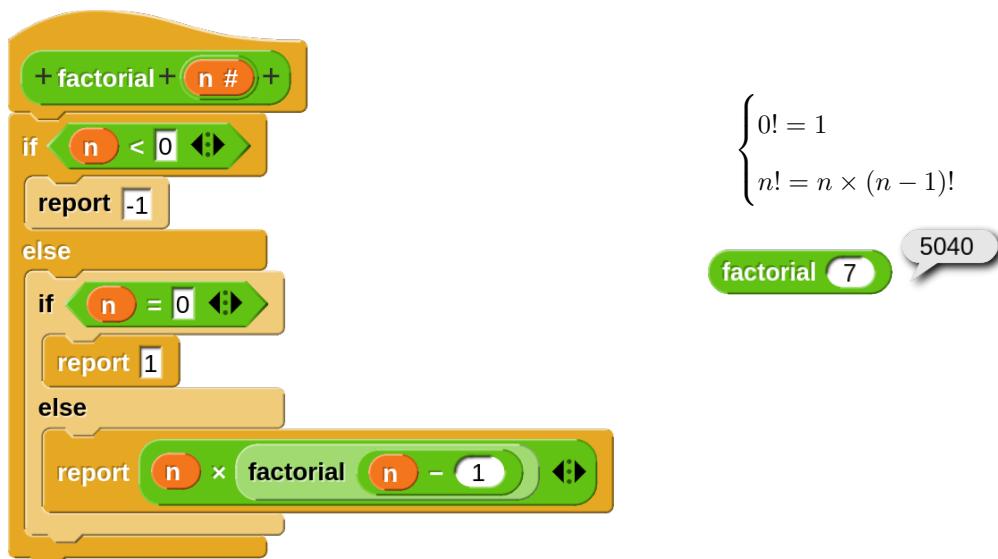
再帰、再帰呼び出し（recursive call）は作成したブロックの中で自分自身を呼び出す（実行する）ものです。関数型プログラミングでは繰り返し処理の手法として再帰を使うことは一般的で、効率的だったりします。

9.1 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。

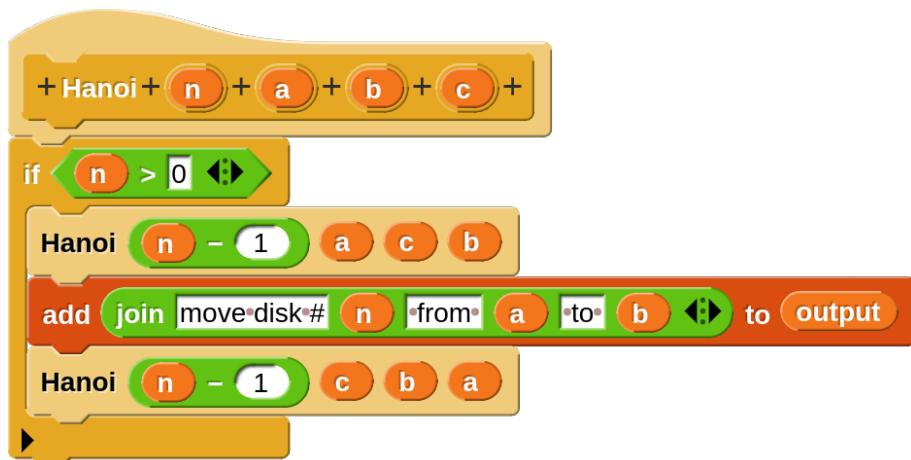
9.1.1 階乗

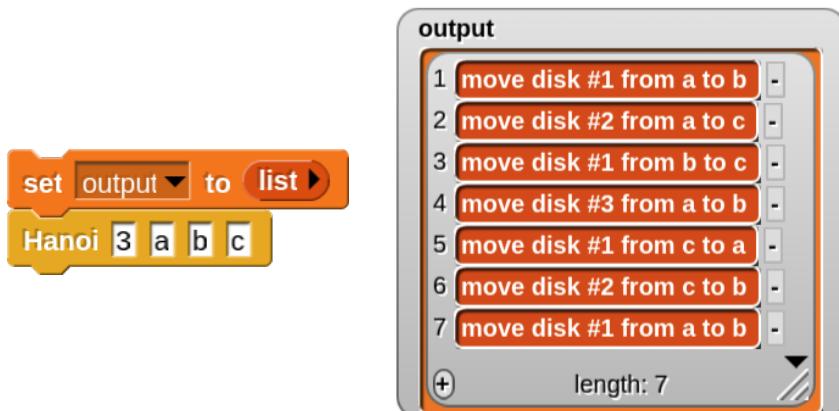
factorial 階乗は再帰の例としてよく使用されます。



9.1.2 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。

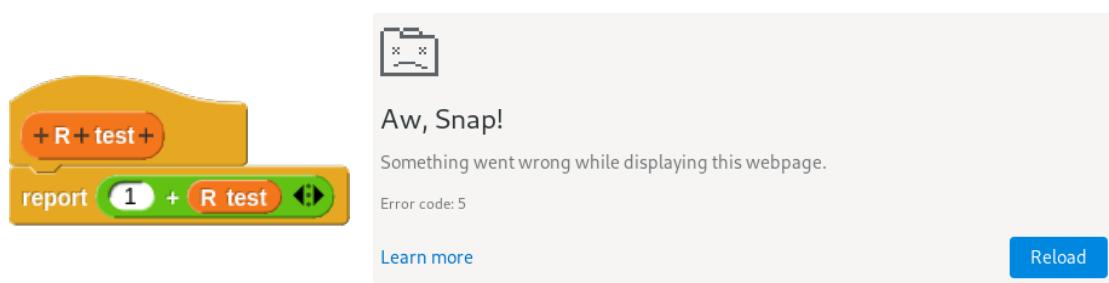




9.2 再帰呼び出しの使用

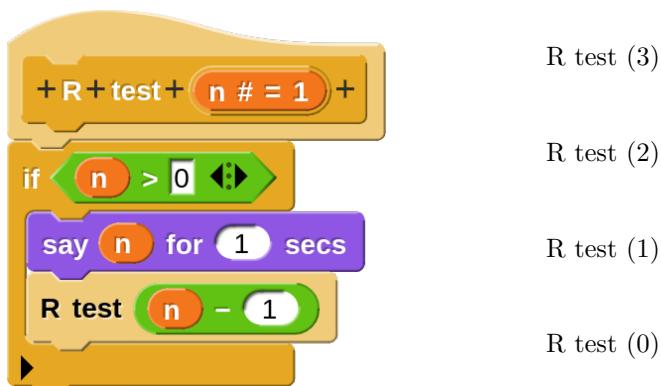
9.2.1 繰り返し

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

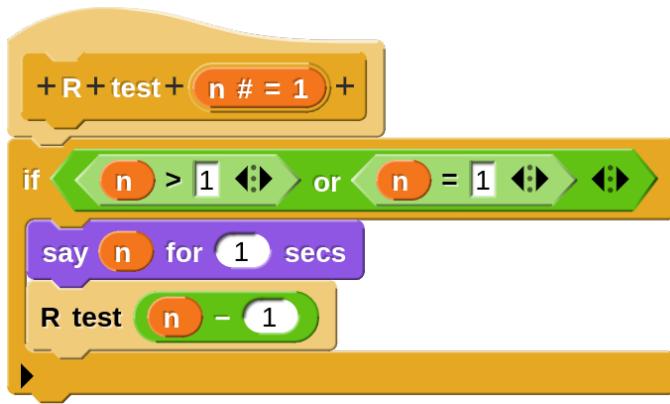


定義ブロックはどこかのスクリプトから呼び出されて実行されるわけですが、実行終了後に呼び出し元に帰る必要があります。呼び出し元のスクリプトの実行を継続するための情報を保存しておき、それを取り出してそこに復帰します。上記のスクリプトでは自分自身への無限呼び出しになり、情報を保存するための場所がなくなってしまいます。ただの無限ループというだけの問題ではありません。したがって、再帰呼び出しでは再帰呼び出しを終了するためのスクリプトが必要です。

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。



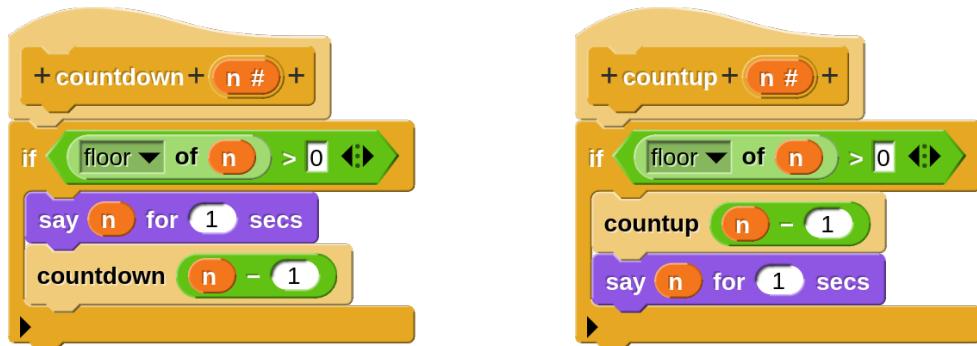
回数に 0.5 を指定しても実行されるので終了条件を変更します。



9.2.2 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。

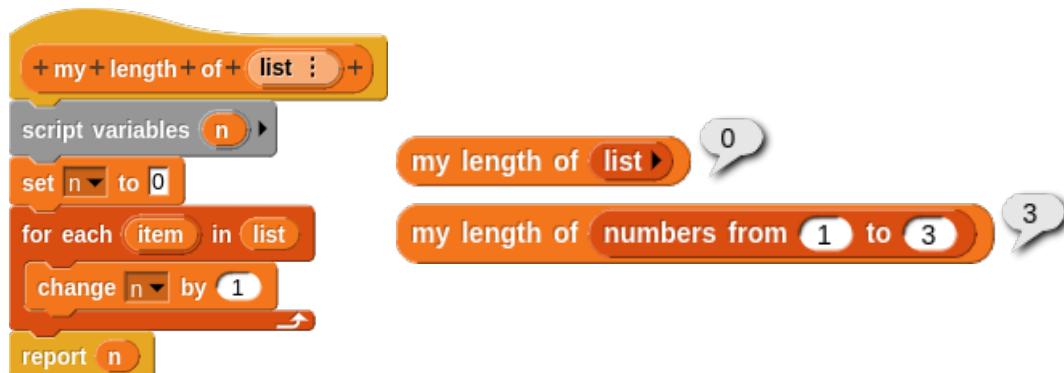
say ブロックのところに処理したいコードブロックを持ってくれば繰り返しの処理ができます。



9.2.3 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。

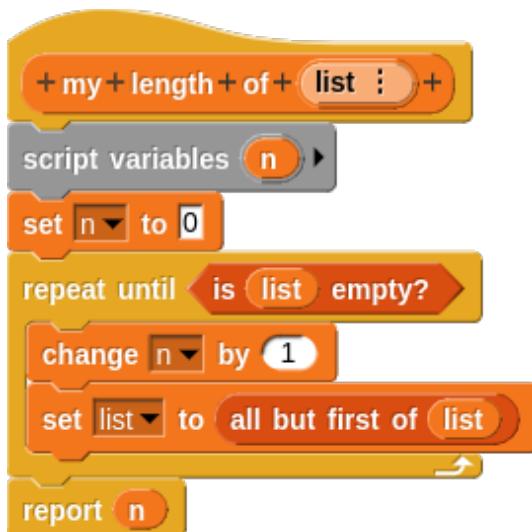


処理にかかる時間を表示します。



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。



16.7

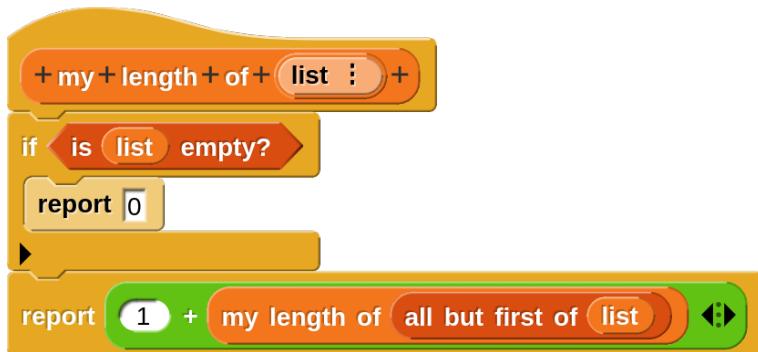
処理にかかる時間を表示します。



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

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

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



本来ならば再帰呼び出しの定義は

```
if (空リストか?) { 空リスト処理 } else { それ以外の処理 }
```

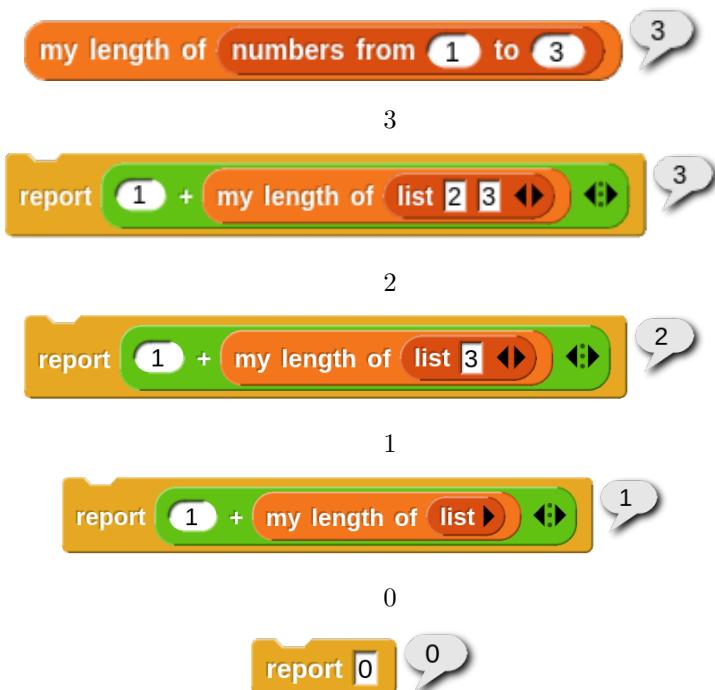
を使うべきですが、表示の関係でこのようにしました。

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。

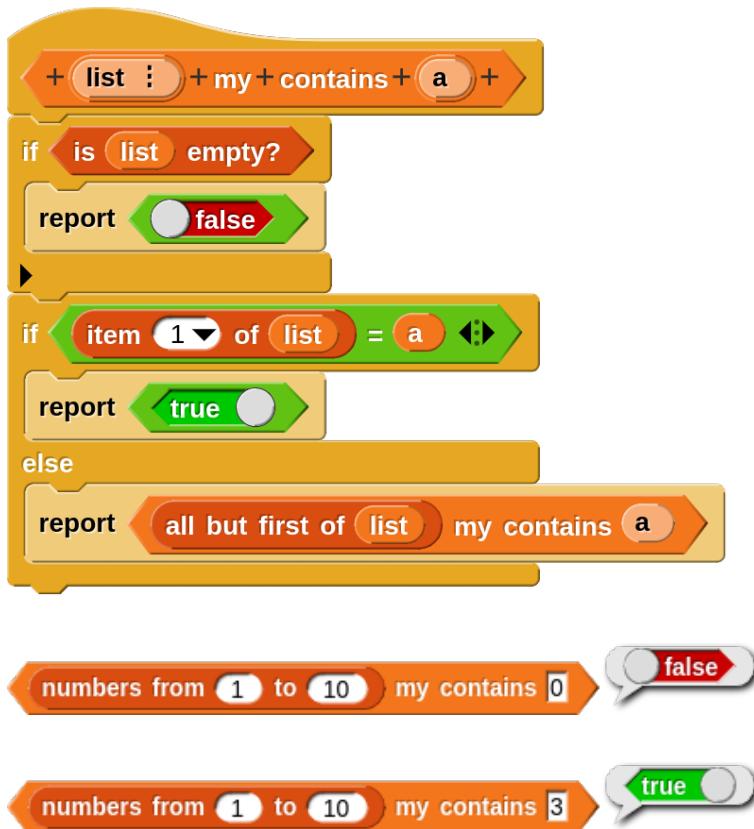
実行される様子を見てみます。 で示す順に再帰呼び出しが実行され、0と値が確定すると、
で示す順に返された値に1を加えて呼び出し元に値を返していきます。最終的に値は3になります。



9.2.4 my contains

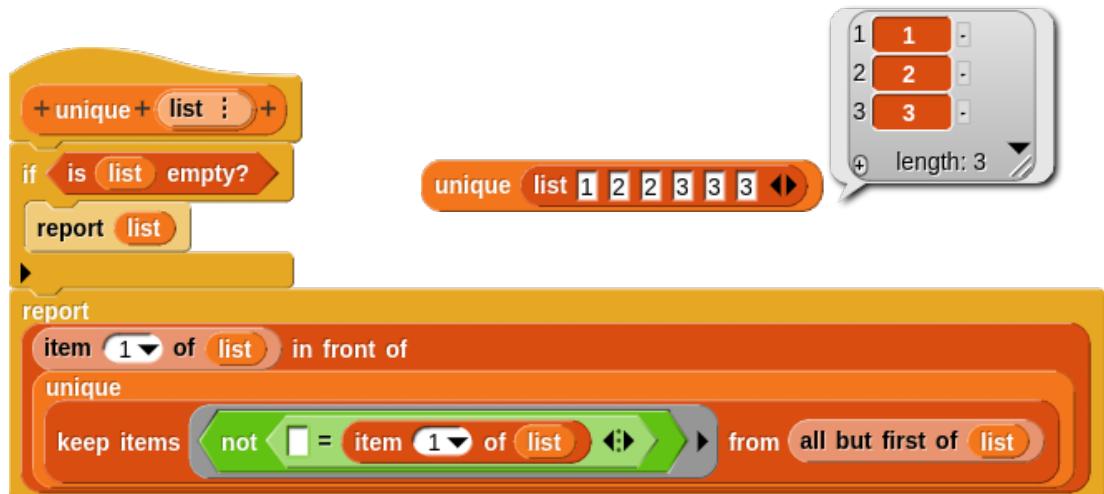
リストの中に指定の要素が存在するかを求める contains ブロックを作ります。

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



9.2.5 unique

Ruby 言語の Array には unique という、配列から重複を取り除く機能が定義されています。



keep を使って、先頭の要素と同じでないものを再帰的に集めています。

9.2.6 リスト要素の巡回

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



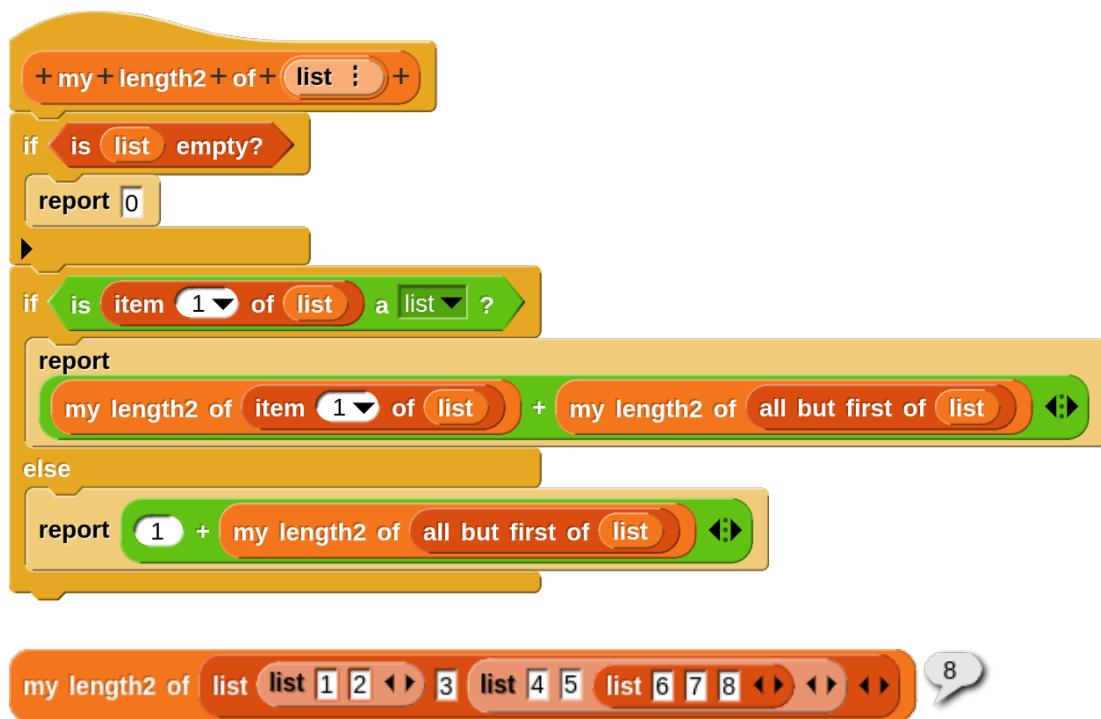
これは my length も同様です。



再帰を使って内部の要素に対してもアクセスしてみます。

処理の内容は次のようにになります。

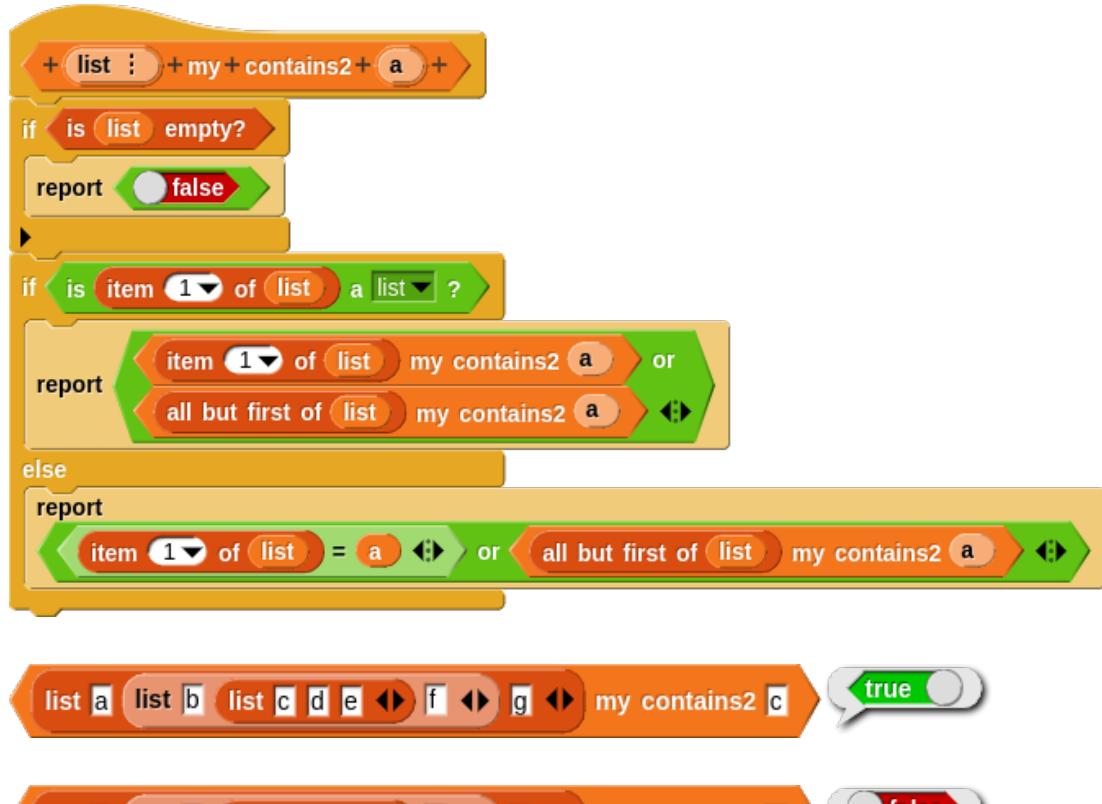
- もしリストが空ならば 0 をリポートする。
- もし先頭の要素がリストならば、そのリストに my length2 をしたものと残りに対して my length2 をしたものを加える。
- そうじゃなかったら、残りに対して my length2 をしたものに 1 を加える。



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



my length2 ブロックを応用すると my contains2 ブロックを作成することができます。true か false かを扱うので演算子は「or」を使用します。



「and」ブロックでは左側のスロットから順にチェックして、`false` ならば残りのスロットのチェックは行いません。それに対して、「or」ブロックでは左側のスロットから順にチェックして、`true` ならば残りのスロットのチェックは行いません。



そのため、リストの先頭からチェックしていくって、指定の要素が見つかればリストの残りはチェックせずにそこで `true` を返して終了になります。



9.2.7 リストからある要素を削除したリストを返す

リストからある要素を削除することは `keep` を使えばできます。しかし、リストの要素がリストの場合はその内部までは対応しないようです。

The Scratch script shows a list of numbers [8, 3, 1, 2, 3] being processed. A condition `not [= 3]` is used with the `keep items` block to exclude item 3 from the list. The resulting list is [8, 3, 1, 2].

```

keep items [not [= 3]] from list [8 3 1 2 3]

```

`my length2` を応用した再帰版です。要素がリストだった場合には、得られたリストを `list` の中に入れることでその構造を維持します。

The Scratch script defines a function `+delete+ [a] [from] [list]` to delete item `a` from list `list`. It handles empty lists and non-empty lists by either reporting the list or performing a recursive deletion.

```

+delete+ [a] [from] [list]
if [is [list] [empty?]]
report [list]
else
if [is [item 1 of list] [a] [list] ?]
report [append [list delete [a] [from] [item 1 of list]] [delete [a] [from] [all but first of list]]]
else
if [item 1 of list] [= a]
report [delete [a] [from] [all but first of list]]
else
report [item 1 of list] [in front of] [delete [a] [from] [all but first of list]]
end
end

```

The Scratch script executes the `delete` function with arguments [3] and [list [8 3 1 2 3]]. The resulting list is [8, 3, 1, 2].

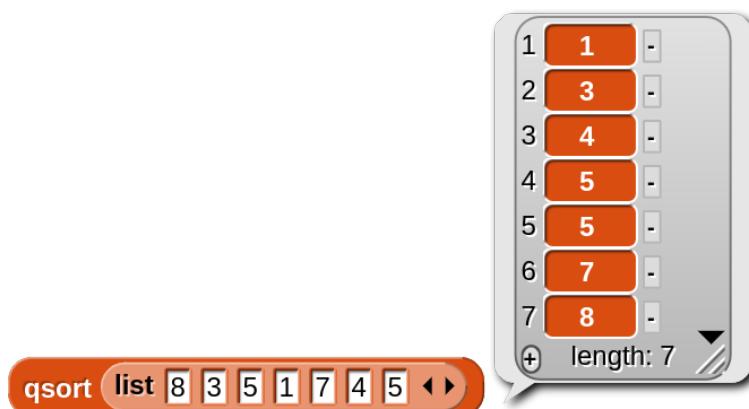
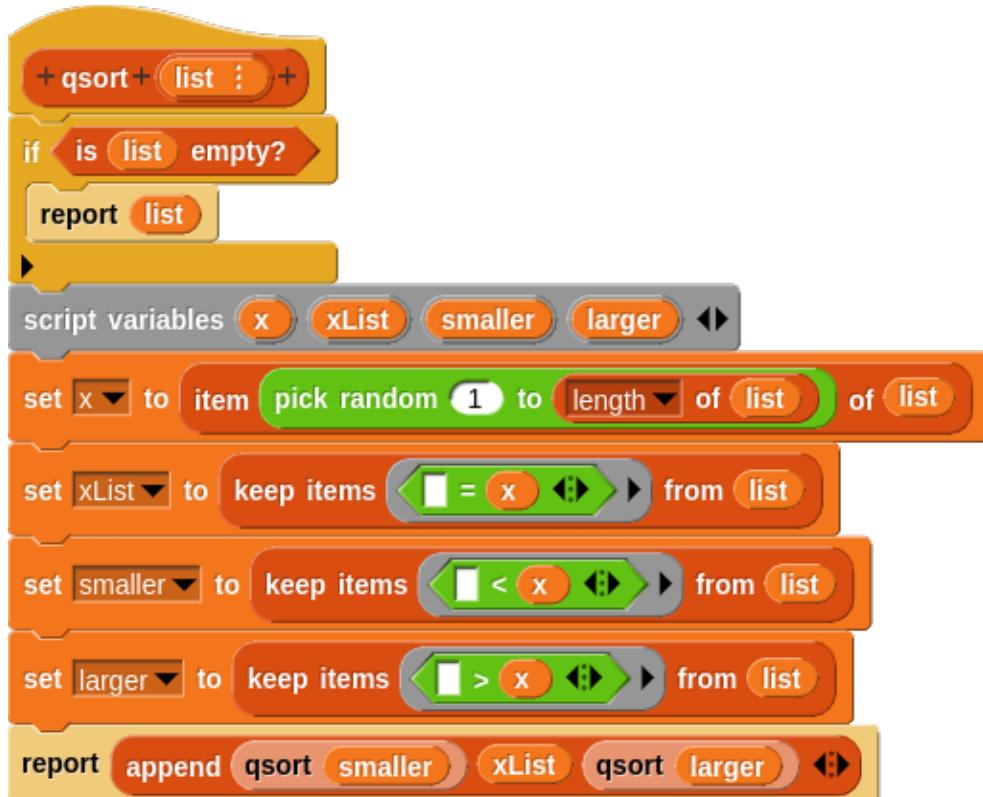
```

delete [3] from list [8 3 1 2 3]

```

9.2.8 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]

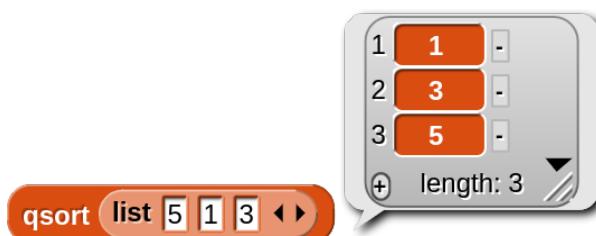
『Programming in Haskell, 2nd edition』

Graham Hutton 著 山本和彦 訳 ラムダノート株式会社 刊

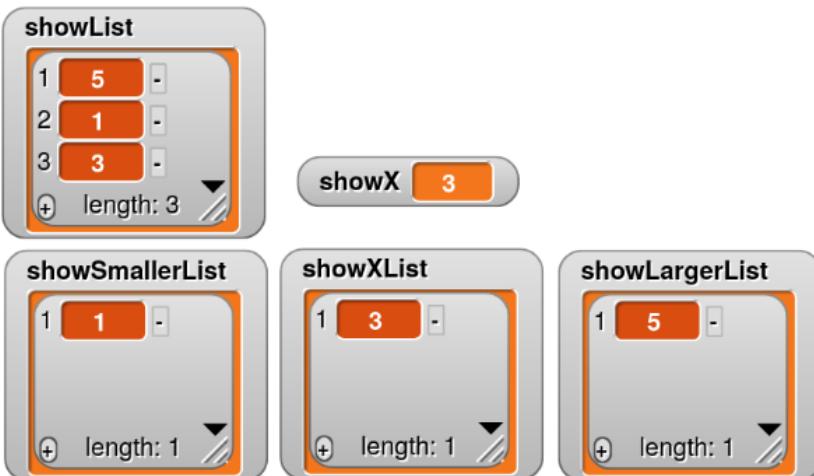
次のように showList, showX, showXList, showSmallerList, showLargerList のグローバル変数を作成し、リストを通して値を表示すると操作の様子が見られます。



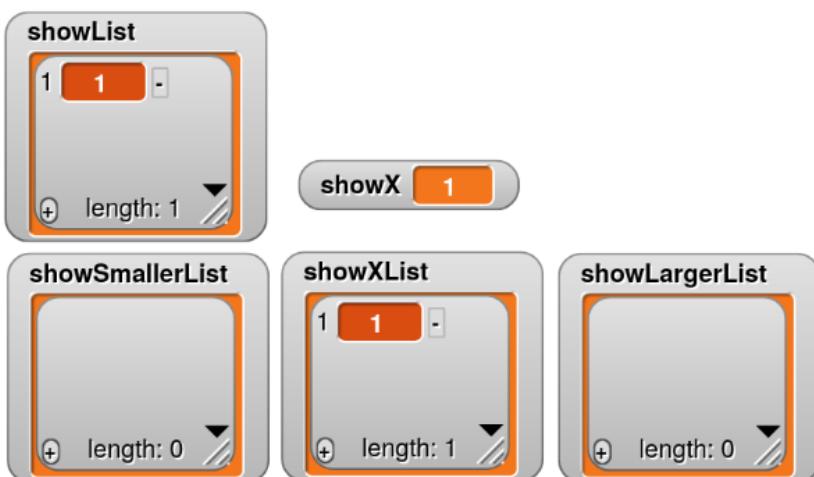
リストの値表示のたびに pause all になります。▶ をクリックして続行してください。



最初に選択された値が 3 だった場合です。



3 より小さい値のグループ処理



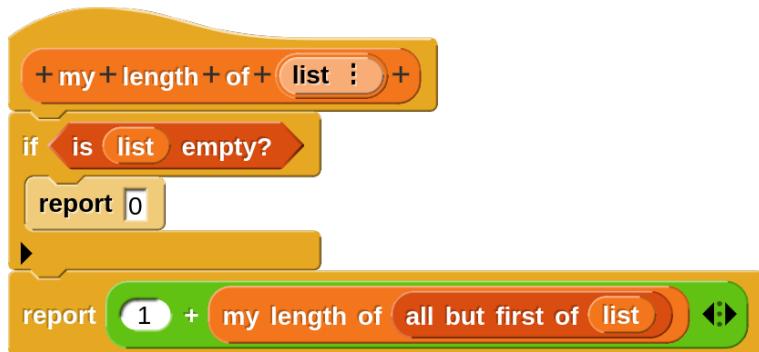
3 より大きい値のグループ処理



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

9.2.9 末尾再帰

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

$$L(1, 2, 3)$$

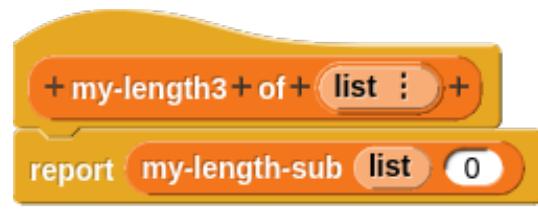
$$\begin{aligned}
 & 1 + L(2, 3) \\
 & 1 + L(3) \\
 & 1 + L() \\
 & 1 + 0 \\
 & 1 + 1 \\
 & 1 + 2 \\
 & 3
 \end{aligned}$$

これに対して、次のようにすると再帰呼出しだけを行うという形にすることができます。

引数 $(n + 1)$ を渡していくことでカウントしています。



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



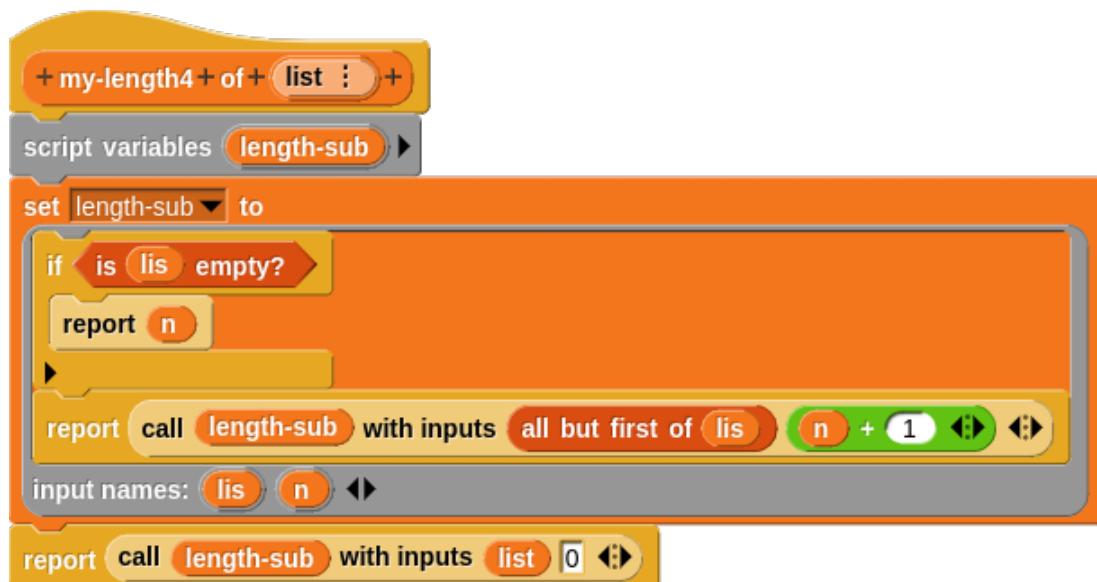
my-length3 of [list 1 2 3 <>] 3 となります。

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

このように、一番最後の再帰呼び出しの返り値がそのまま定義ブロックの値になり、再帰呼び出しから戻る時は単に値をそのまま渡すだけです。このような処理の最後を単独の再帰呼び出しにする形を末尾再帰といいます。

Scheme や Haskell などでは、末尾再帰は最適化されるので効率がよくなりますが、Snap! では変わらないようです。

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



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。

reverse(1, 2, 3, 4, 5)	[L4] + [1]	L5 = [5, 4, 3, 2, 1]
reverse(2, 3, 4, 5)	[L3] + [2]	L4 = [5, 4, 3, 2]
reverse(3, 4, 5)	[L2] + [3]	L3 = [5, 4, 3]
reverse(4, 5)	[L1] + [4]	L2 = [5, 4]
reverse(5)	[L0] + [5]	L1 = [5]
reverse()	[]	L0 = []

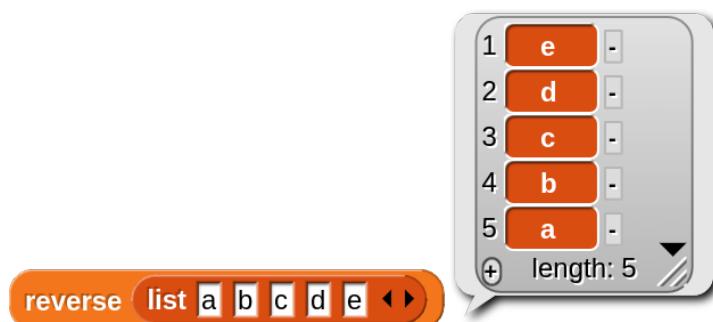


注意点として、append はリスト同士を一つにするものなので、

list [item 1 of list :] のようにしてリストにする必要があります。



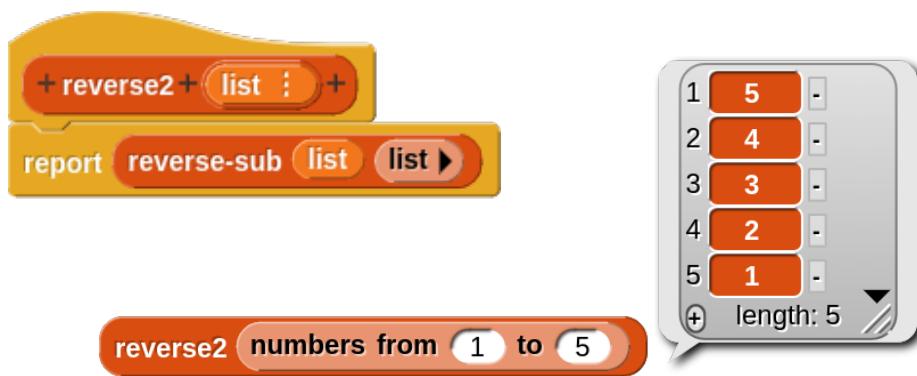
数値じゃなくてもできます。



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。`reverse-sub` の入力スロットに二つの list がありますが、右側のものは初期値としての `list` 空リストです。



局所定義ブロック版です。

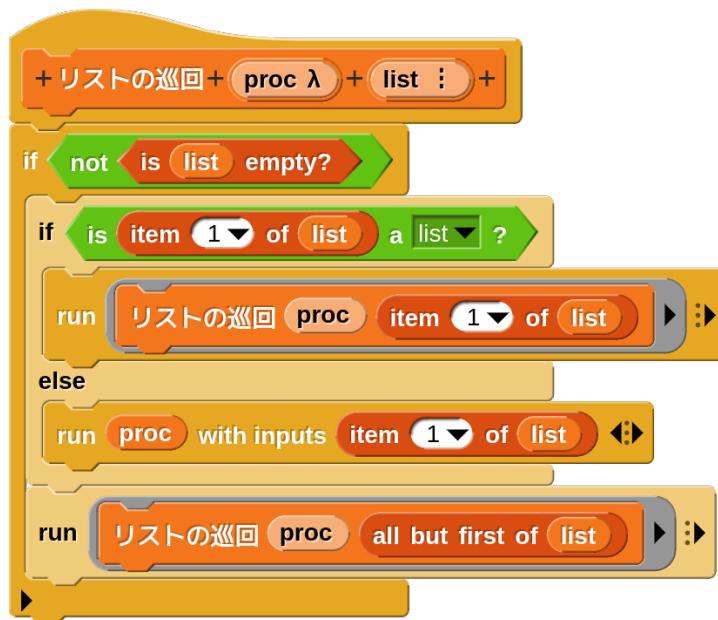


10 高階関数

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

入力スロットにスクリプトブロックを指定する例を示してみます。

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



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

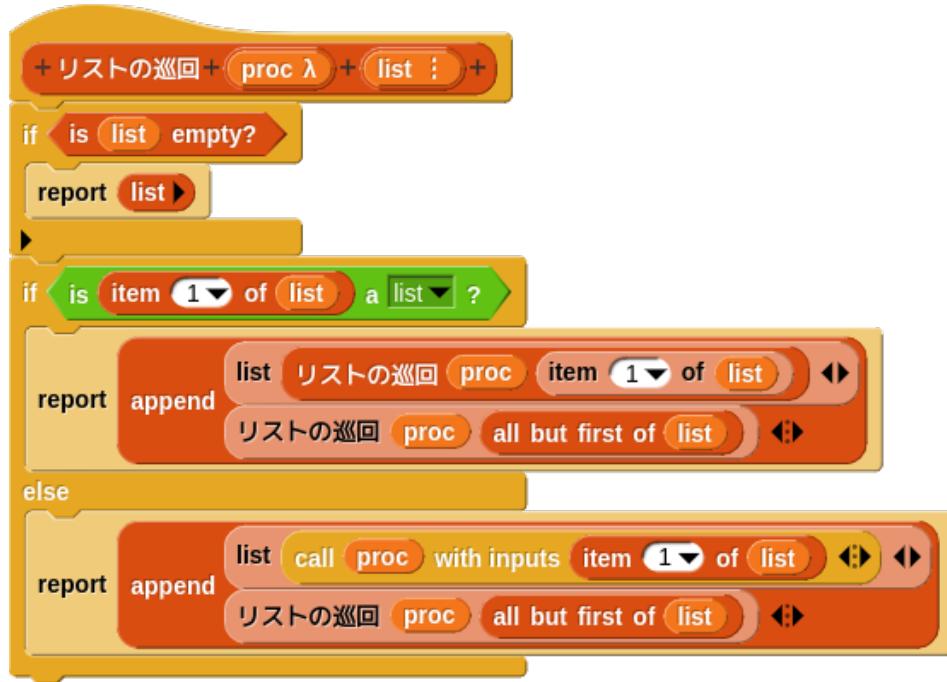


リストの要素の値の合計値を求めます。上とは違い [change (n) by ()] と引数のスロットにリストの要素が入り加算されます。



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

リポーター版です。引数 proc は Reporter 型です。



command 版と同じようにできますが、Reporter 型なので report を入れています。

空きのスロットにはリストの要素が入ります。

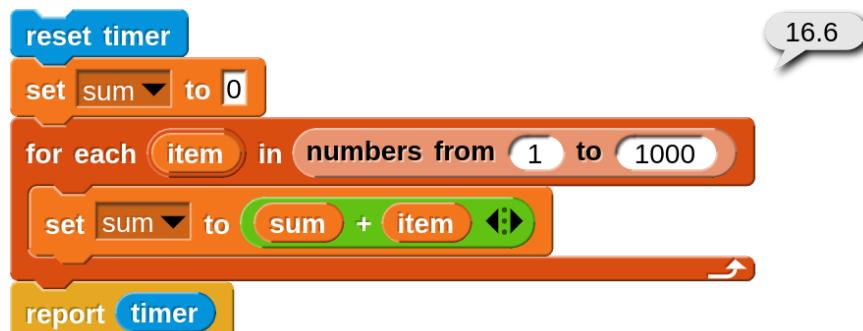
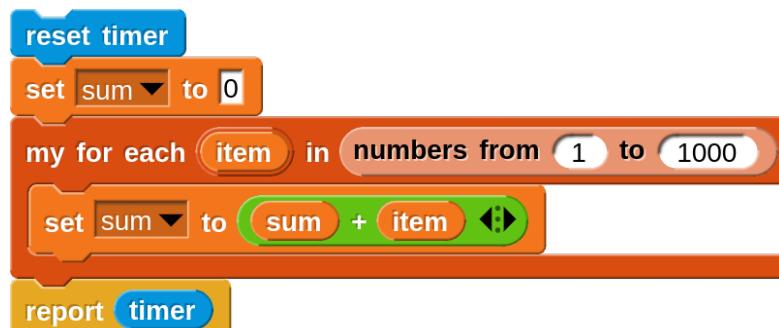
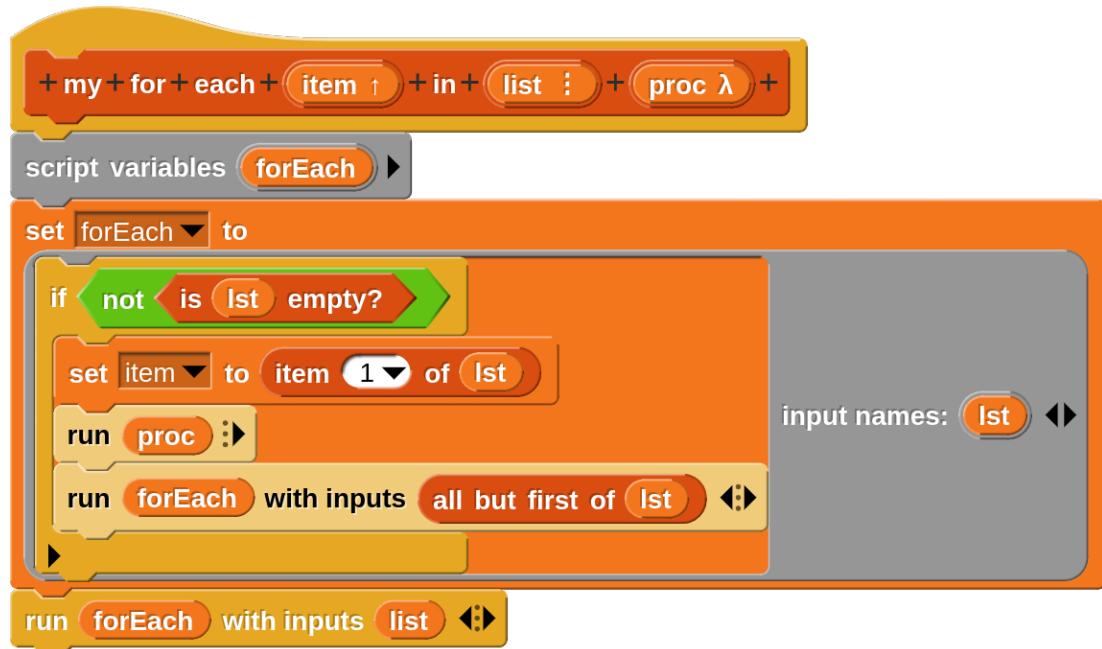


for each item in 目

→ を再帰処理で行うと高速にできます。

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

proc は [Command (c-shape)] で作成します。



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

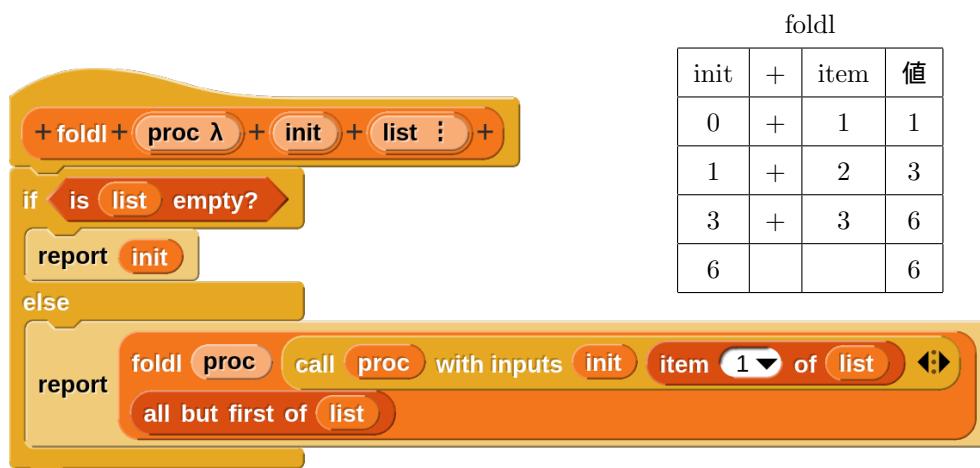
これは、二つの引数をとるリポーターブロック proc と、初期値 init と list の合計 3 つを引数として取る関数です。初期値 init と member (list の先頭要素) の二つの値を引数として proc が何らかの処理をします。得られた値を新たな init とします。その init と member (list の次の要素) を引数として proc を実行 ... ということを list の終わりまで行います。init の値が最終値になります。foldl はリストを左側から順に操作し、foldr はリストを右側から順に操作するものです。

foldl から見ていきます。

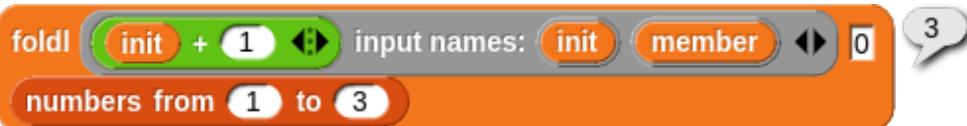


操作内容は、
のようにになります。

フォーマルパラメーターを使用すると次のようにになります。



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



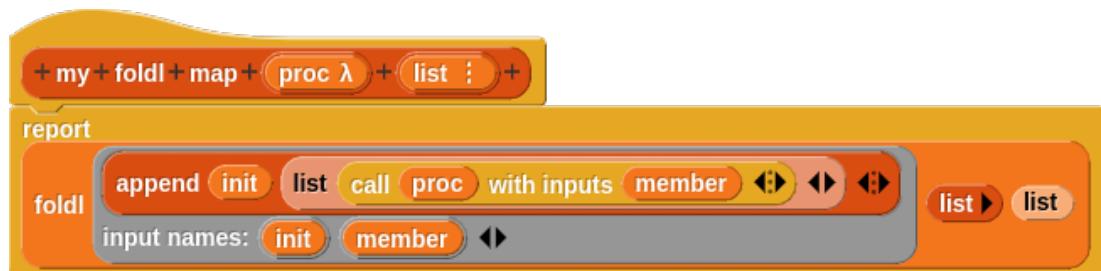
次のようにすると最大値を求めることができます。



map ブロックを再帰呼び出しを使用して作成してみます。引数 proc は Reporter 型です。



foldl 版です。



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



foldl 版です。

A Scratch script titled "report foldl". It contains the following blocks:

- Top block: + my + foldl + keep + items + pred λ + from + list : +
- Control block: if call pred with inputs member then append init list member
- Control block: else init
- Data block: input names: init member
- List block: list

foldr です。

A Scratch script titled "foldr". It contains the following blocks:

- Control block: foldr [+ + <>] [0 numbers from 1 to 3] [6]

操作内容は、
のようになります。

フォーマルパラメータを使用すると次のようにになります。

A Scratch script titled "foldr". It contains the following blocks:

- Control block: foldr [member + init <>] [input names: member init <> 0] [6]
- Data block: numbers from 1 to 3

A Scratch script titled "foldr". It contains the following blocks:

- Control block: + foldr + proc λ + init + list : +
- Control block: if is list empty?
- Control block: report init
- Control block: else
- Control block: report
- Control block: call proc
- Control block: with inputs item 1 of list foldr proc init all but first of list

右侧に表示されるfoldrの実装表:

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

foldr を使っても foldl と同じようなことができます。

A Scratch script titled "foldr". It contains the following blocks:

- Control block: if member > init then member else init
- Control block: input names: member init
- List block: list [8 3 12 10 5]

+ my + foldr + map + proc λ + list : +

report

foldr append list call proc with inputs item :: init :: list list

input names: item init ::

+ my + foldr + keep + items + pred λ + from + list : +

report

foldr

if call pred with inputs member :: then append list member :: init ::
else init

input names: member init ::

list list

foldr がリストを右側から操作することを利用した append です。

+ my + foldr + append + list1 : + list2 : +

report

foldr member in front of init input names: member init :: list2 list1



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

foldl で作成する場合には、リストを逆順にする操作（114 ページ参照）が必要になります。

+ my + foldl + append + list1 : + list2 : +

report

foldl member in front of init input names: init member :: list2
reverse list1

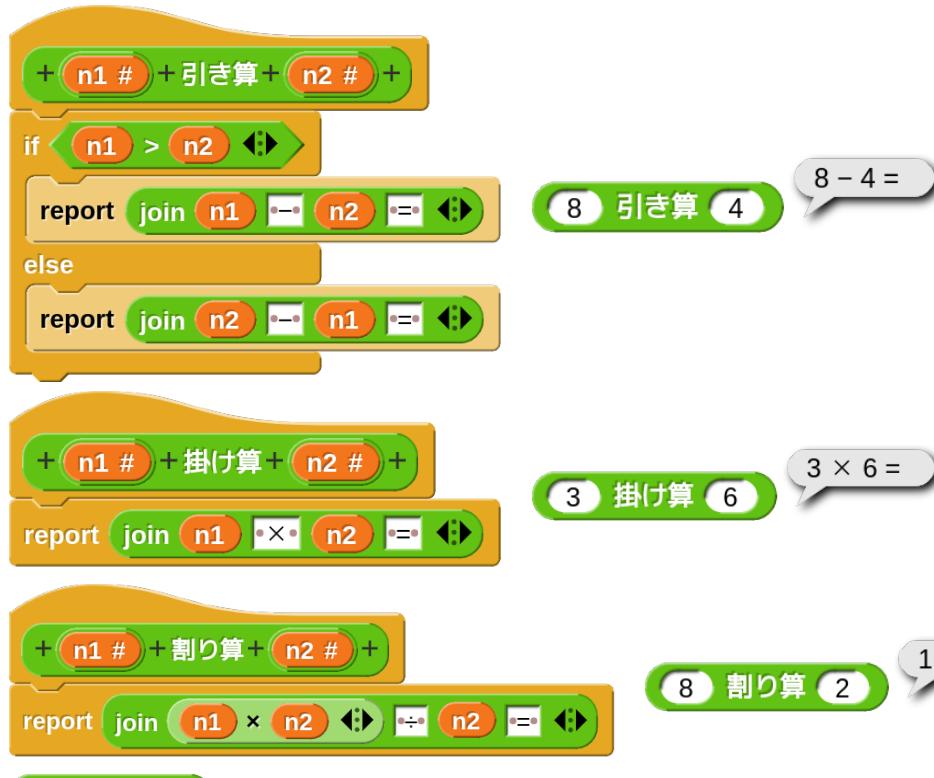
再帰呼出しを使ってリスト処理をすれば効率の良いプログラムになりますが、再帰を使ったプログラム作成はなかなかたいへんです。 fold を使用してなんでも作成できるわけではありませんが、自動的に再帰呼出しを使った処理してくれます。

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

まずは基本となる 2 つの引数を取り、足し算の式として表示するものです。



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

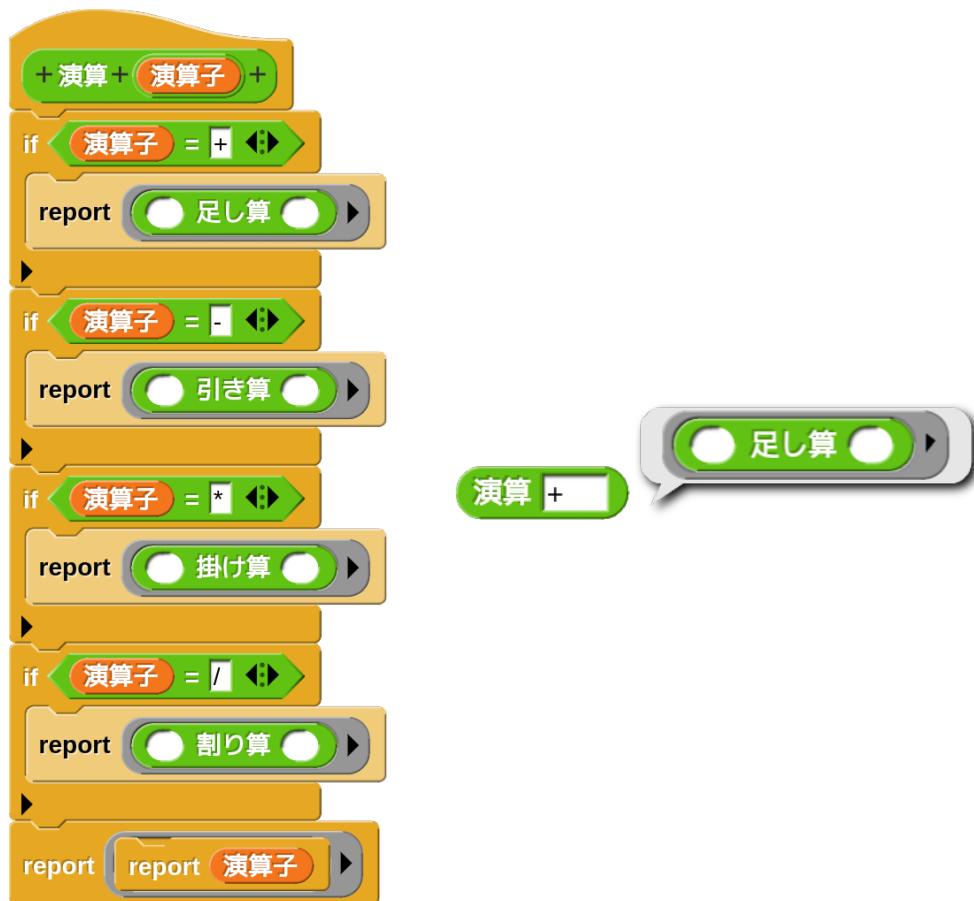


● 足し算 を ringify してリポートすると、

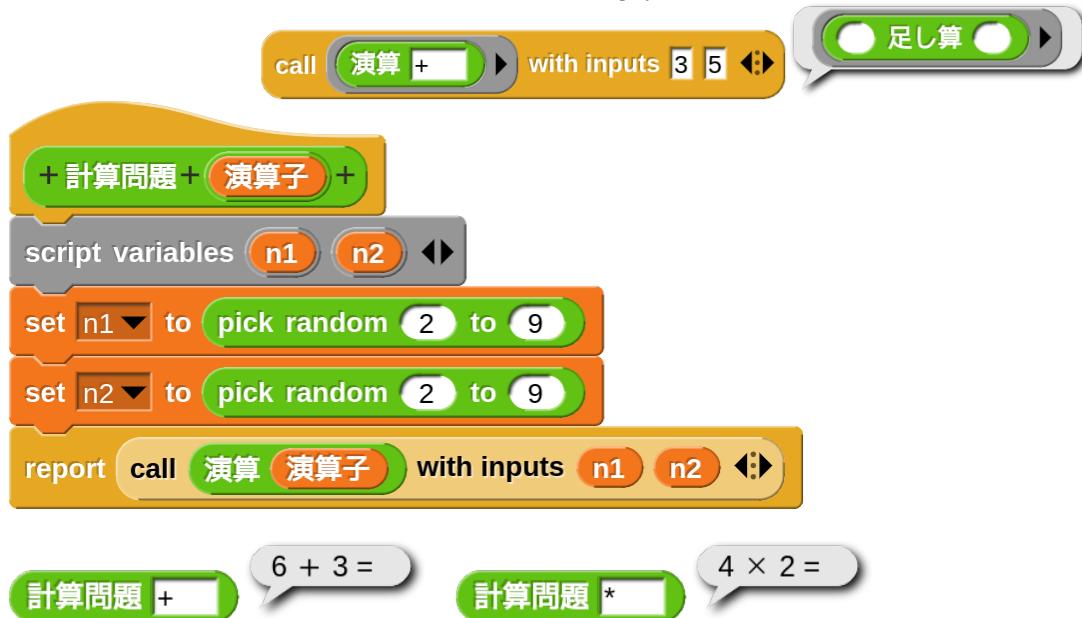


のようにスクリプトブロックをリポートすることができます。

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



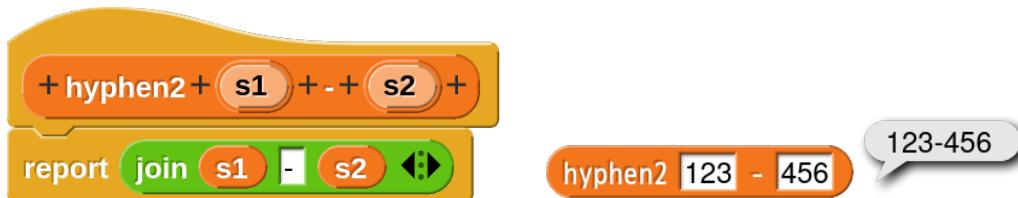
それを呼び出す本体のブロックです。call の入力スロットに **演算 演算子** を入れると ringify **演算 演算子** の状態になります。**演算 演算子** のブロックは ringify されたものを返すものなので、それをまた ringify してしまうと call で実行しても ringify されたものを表示するだけになります。演算のところを右クリックして unringify してください。



10.1 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。



「カリー化」は複数の引数の処理を基本単位の処理の連結で行うためのものです。Haskell などでは「関数は一つしか引数を取らない」という言語仕様上重要な手法のようです。

10.2 Class 的な応用

他のプログラミング言語では、Class という型を作り、変数や操作（メソッド）を設定します。その設計図のようなものを基に作られた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! には表立って Class のような仕組みはありませんが、高階関数を使うことで似たようなことができます。

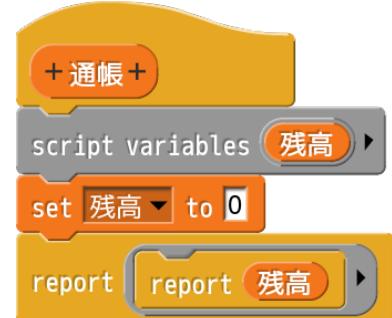
題材としては、通帳を使ってお金の出し入れをする行為は共通のイメージがあると思うので、それを使って説明してみます。

預貯金通帳を定義ブロックで作ってみます。

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。

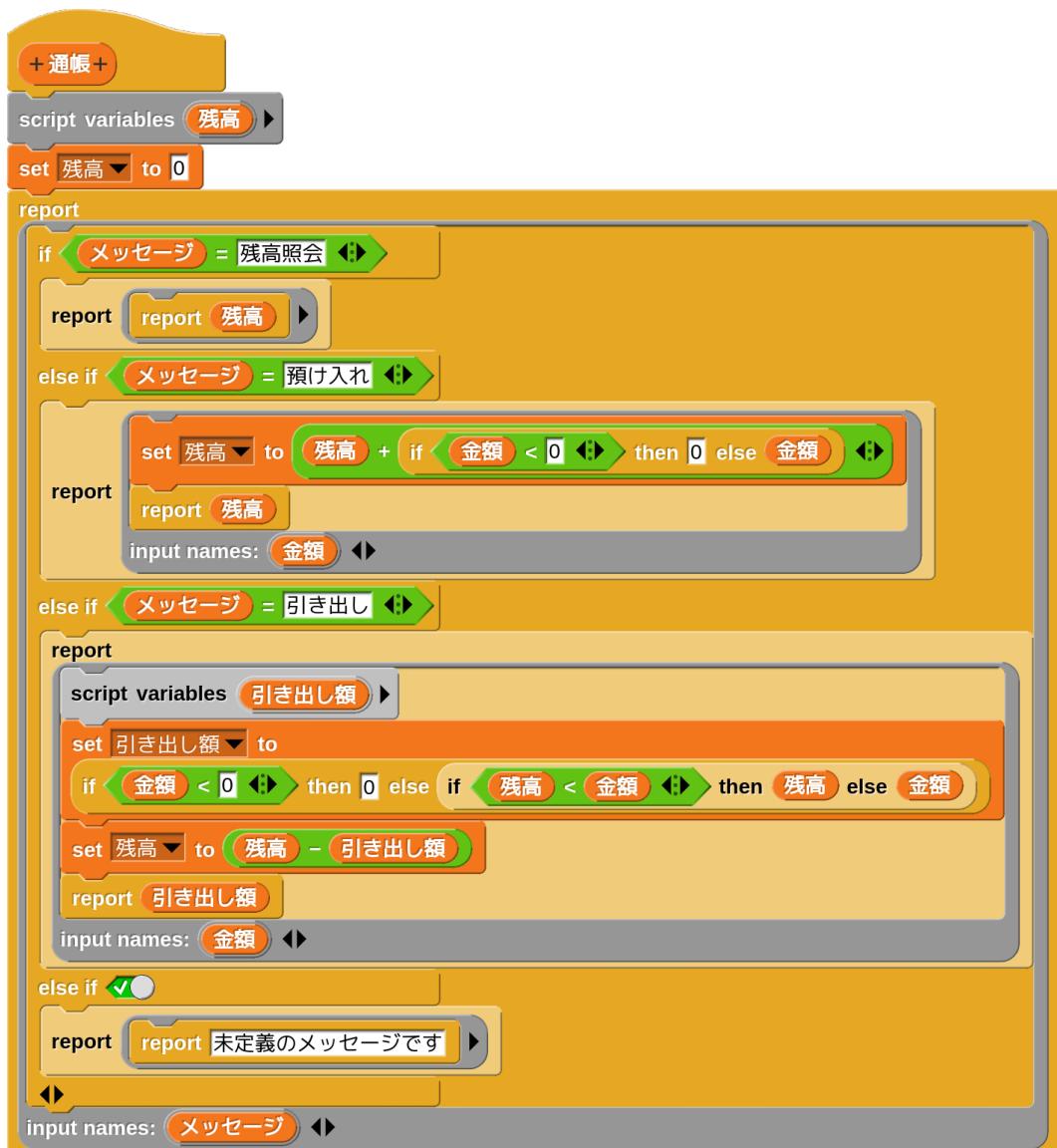


「太郎の通帳」を表示させてみます。変数「残高」は「太郎の通帳」が削除されるまで有効です。ローカル変数ですので「花子の通帳」の変数「残高」とは別物です。



リングで囲まれたブロックなので、残高を表示するには call を使用しなければなりません。

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。「引き出し」の場合は、引き出し額が残高を上回る時は引き出し額を残高としてから残高から減算し、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。
金額を指定するために 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 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。「通帳名」は Object、「操作」は Text, options... と read-only を選択しています。(59 ページ参照)

実は、「通帳名」は「通帳」型の変数を受け取れればいいので Reporter でも Any type でも大丈夫みたいですが、Object のほうがしっくりきます。



残高照会の場合の金額入力は参照しません。



索引

all but first of, 20, 103–106, 112
call, 34
case, 72
combine, 25
composition, 46
Costumes, 9
csv, 6, 13, 29
draggable?, 9
factorial, 100
import, 9, 29, 46
input list:, 34
Iteration, 46
JavaScript, 87
JavaScript extensions, 5
join, 25
json, 6, 13, 29
keep, 24, 109, 120
Language, 5
Libraries, 9
list view, 18
map, 23, 120
New, 9
object, 65, 67
Open, 9
pen trails, 30
pipe, 41
relabel, 11
reshape, 22
ringify, 29, 50
rotation style, 8
run, 33
Save, 9
Save As, 9
scene, 9
script pic, 11
split, 26
switch, 72
table view, 18
transient, 14
turbo mode, 10
unique, 105
unringify, 29
UTC, 92
while, 73
with inputs, 33
Zoom, 5
オフライン版, 5
階乗, 25, 100
カスタムブロック, 7, 42
カリー化, 125
協定世界時, 92
クイックソート, 109
組み合わせ, 25
グローバル変数, 13, 14
高階関数, 116
再帰呼び出し, 100
ジュークボックス, 7
順列, 25
スクリプトエリア, 7
スクリプト変数, 16
ステージエリア, 6
ステップ実行, 8, 45, 75

スライトコラル, 7
スライト变数, 15
ゼブラカラーリング, 11
ソート, 88
ターボモード, 10
大域变数, 14

定義ブロックのインポート, 58
定義ブロックのエクスポート, 58
デバッグ, 8, 75

時計, 92

二進数, 88
日本語化, 5

排他的論理和, 89
配列, 22, 87
バックグラウンド, 7
ハノイの塔, 100
パレットエリア, 7
八口, 12

ピット演算, 88
評価, 67, 69

フォーマルパラメータ, 23, 35
プリミティブブロック, 7
プロトタイプ, 43, 45, 63, 65

並列処理, 95
変数ウォッチャー, 6, 14, 18, 19, 28

未尾再帰, 112

無名関数, 35

ラムダ関数, 35

リスト要素の巡回, 106, 116
リング, 23, 29, 34, 38, 50, 66, 67, 69, 73

ループ变数, 17