

Snap! のこと I

齋藤文康

2026 年 7 月 29 日

Snap! Build Your Own Blocks (ver. 12) の使い方について、Scratch に準じているので、基本的なことは省いて、私が説明できることだけを取り上げました。マニュアルのようにすべての事柄を説明することはできません。

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

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

置き場所は、「Snap! プログラミング」[<https://fumiyasu-aizu.github.io/>] です。容量の関係で分割した部分「Snap! のこと II」もそこに置かれています。このサイトが検索エンジンに無視されて表示されないことがよくあります。間違いの修正や新たに分かったことを加筆していますので上記サイトの最新のものをご利用ください。

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	14
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	Upvar 変数	17
4.5	リスト操作ブロック	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	delete () of ()	21
4.5.6	append () ()	21
4.5.7	for each () in ()	21
4.5.8	reshape () to () ()	22
4.5.9	map () over ()	23
4.5.10	keep items () from ()	25
4.5.11	find first item () in ()	26
4.5.12	combine () using ()	27
4.5.13	combinations () ()	29
4.6	オプションのリスト操作	30
4.6.1	rank of (), dimensions of ()	30
4.6.2	flatten of ()	31
4.6.3	columns of () 転置行列	31
4.6.4	uniques of ()	32
4.6.5	distribution of ()	32

4.6.6	sorted of ()	32
4.6.7	shuffled of ()	33
4.6.8	reverse of ()	33
4.6.9	text of ()	33
4.6.10	lines of ()	33
4.6.11	csv of ()	34
4.6.12	json of ()	34
4.7	リストの演算	34
4.8	変数に入れられるもの	36
5	Control 制御	38
5.1	リポーターの if < > then () else ()	38
5.2	when ()	38
5.3	stop ボタンがクリックされた時の終了処理	39
5.4	run ブロック	40
5.5	call ブロック	41
5.6	launch ブロック	43
5.7	broadcast ブロック, tell to ブロック, request ブロック	46
5.8	pipe	48
6	ブロックを作成する	49
6.1	定数の作成	49
6.2	() $\geq$ () ブロック	51
6.3	help 説明文の作成	54
6.4	for (i) = (start) to (end) step (step)	54
6.5	block variables	64
6.6	Upvar 変数の設定	65
7	ブロック定義について	67
7.1	プルダウン入力	67
7.2	Title Text とシンボル	72
7.3	Input name オプションについて	73
7.3.1	Reporter 型	73
7.3.2	Predicate 型	78
7.3.3	Command 型	79
8	その他	84
8.1	デバッグ	84
8.2	整数への変換	86
8.3	分数の加算	87
8.4	論理演算ブロック	91

8.5	集合	96
8.6	getc	99
8.7	when slot () signals ()	101
8.8	Event Hat ブロック	104
8.9	プリミティブブロックの編集	106
8.10	read-only 入力スロット	108
8.11	入力スロットへのリスト指定	109
8.12	入力スロットの追加	110
8.13	ask	112
8.14	broadcast の検索オプション	113
8.15	クローン	113
8.15.1	テンポラリクローン	113
8.15.2	パーマネントクローン	117
8.16	flat line ends	118
8.17	角度	124
8.18	layer レイヤー	126
8.19	Animation	129
8.20	anchor アンカー	132
8.21	JavaScript function (オプション 5 ページ参照)	135
8.22	match	156
8.23	時計	161
8.24	並列実行について	165
8.25	行列の積	167
8.25.1	inner product	167
8.25.2	outer product	174
8.26	Continuation 継続	177
8.26.1	run と call の with continuation バージョン	177
8.26.2	catch と throw	180
8.26.3	map からの脱出	180
8.26.4	break と continue	182
8.26.5	thread	184

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。

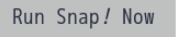
オフライン版は、Snap! のサイトの一番下にある Offline Version のリンクからオフライン版に関するページに移り、Simple Steps: の下の文中のダウンロードサイト

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

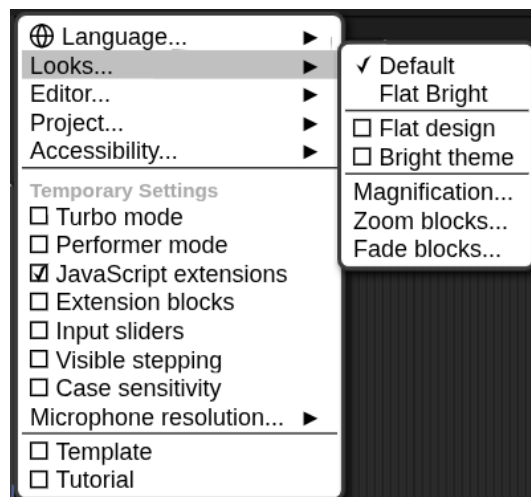
のリンクをクリックすると、オフライン版があるところにたどり着きますから、Source code(zip) か Source code(tar.gz) をダウンロードしてください。

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、 をクリックすれば使用できます。

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



日本語にするには、 のボタンをクリックすると表示される設定メニューの中で

Language... をクリックして日本語を選べば変更することができます。

また、Looks... をクリックすると表示される Magnification... や Zoom blocks... の設定で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ります。

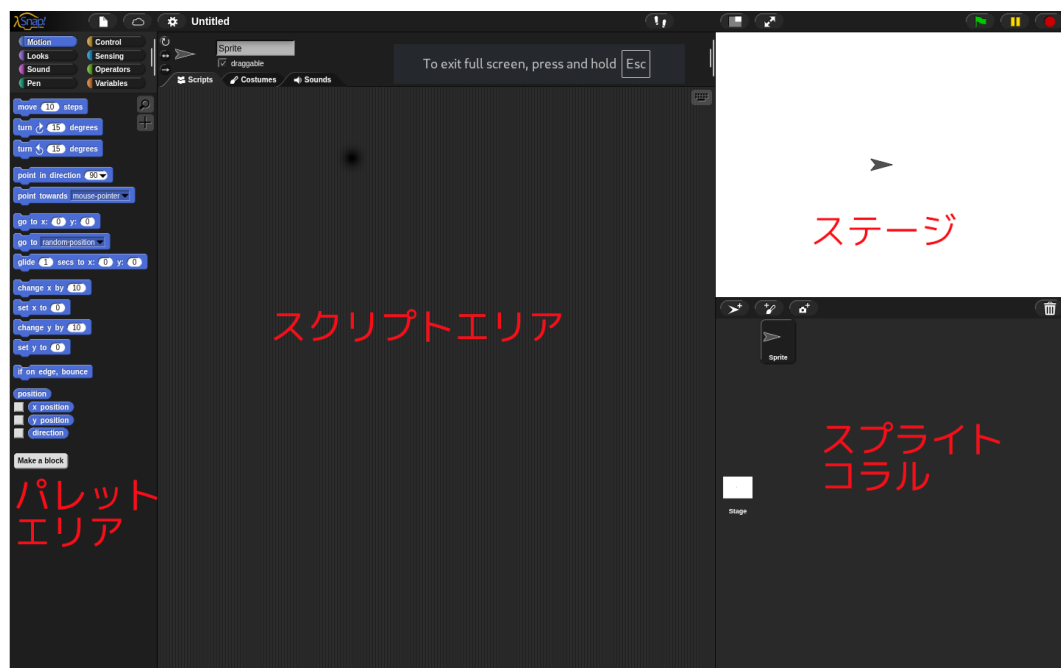
- テキストファイル (.txt) の場合は、変数を作成して読み込みます。
- .csv や .json のファイルは変数を作成し、リストとして読み込みます。

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

英語版でも変数名やデータの内容など日本語が使えます。

2.1 エリア

各エリアには次のような名前がついています。



2.1.1 ステージエリア

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



- edit は、ステージ用のスクリプトの作成です。操作対象を Stage にします。
- show all は、非表示設定になっているものも含めてスプライトを全部表示します。ステージ外に行ってしまったものもステージ内に連れ戻します。変数ウォッチャーも表示されます。不要な変数ウォッチャーは、パレットエリアにドロップしてください。

- pic... は、ステージのスクリーンショットを撮ります。画像はダウンロードフォルダーへ。
- pen trails は、pen や stamp で描いた軌跡を選択されているスプライトのためのコスチュームとしてワードローブに、またはステージのための背景としてバックグラウンドに追加します。このコスチュームの中心は、pen trails された時点の座標になります。

2.1.2 スプライトコラル

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

2.1.3 スクリプトエリア

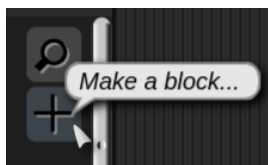
スクリプトエリアは、スクリプトを作成する場所です。ただし、スクリプトエリアの部分はスクリプトを扱う時はスクリプトエリアで、コスチュームを扱う時はワードローブエリア、サウンドを扱う時はジュークボックスと呼び方が変わります。また、ステージの時はワードローブではなくバックグラウンドです。

スクリプトエリアの右上には、 があります。これは後述する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。

ブロックをパレットエリアからスクリプトエリアにドロップすると、 が表示されます。これは、操作のやり直しをするためのものです。 をクリックすれば 1 操作分元に戻せます。これを使うと  が表示され、その変更を元に戻すことができます。

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でもあります。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。



こ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。

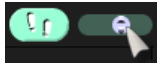
2.2 エリアの大きさ



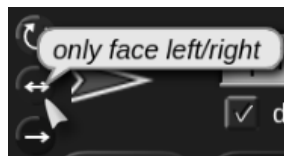
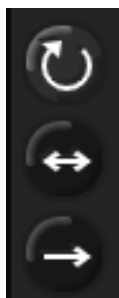
の部分でステージの大きさ、結果的にスクリプトエリアの大きさをすることができます。のボタンのクリックで変わります。のボタンは画面の表示をステージのみにして発表モードにするものです。左図のマウスポインターが置かれて色が薄い紫色になっているところをドラッグしても大きさをすることができます。

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えて横幅のあるブロックの全体を表示させることができます。

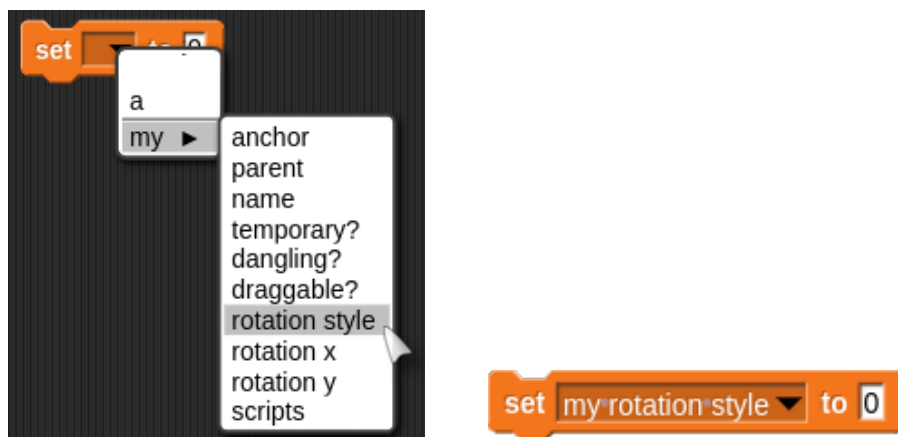
2.3 実行に関するボタン

ステージエリアの上にはのボタンがあります。これはに接続されたスクリプトを実行するボタンです。はスクリプトの実行を終了させるボタンです。はスクリプトの実行を一時停止させるボタンです。のボタンをクリックすると実行中のブロックをハイライトさせてゆっくり実行させたりできます。のマウスポインターが置かれているところをドラッグすると実行のスピードが調整できます。左端だと1動作ごとの1ステップ実行になります。デバッグの時に使えます。(84 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、**set my draggable? to** ☒ **true** 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。

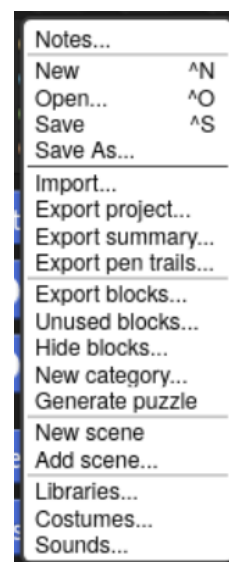
Notes... にはプロジェクトの覚書、注釈が書けます。

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。

switch to scene next and send 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。

「Export blocks...」「Unused blocks...」の項目はユーザー定義ブロックがない場合は表示されません。

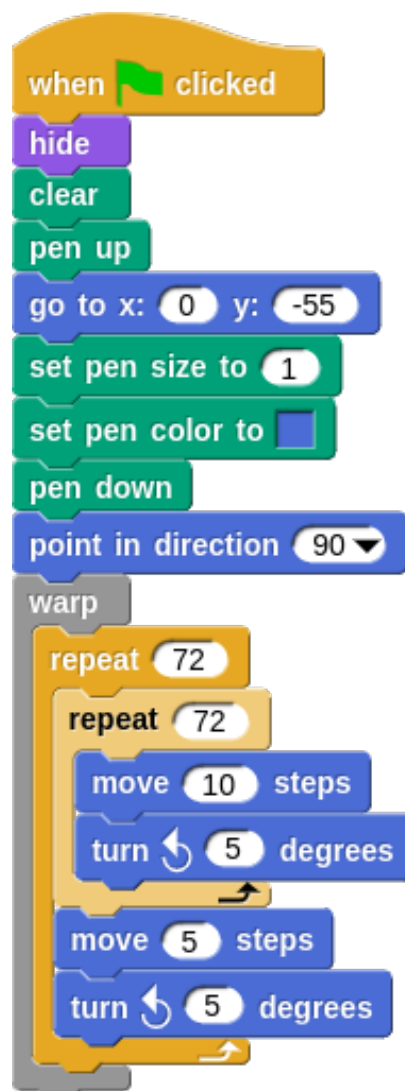
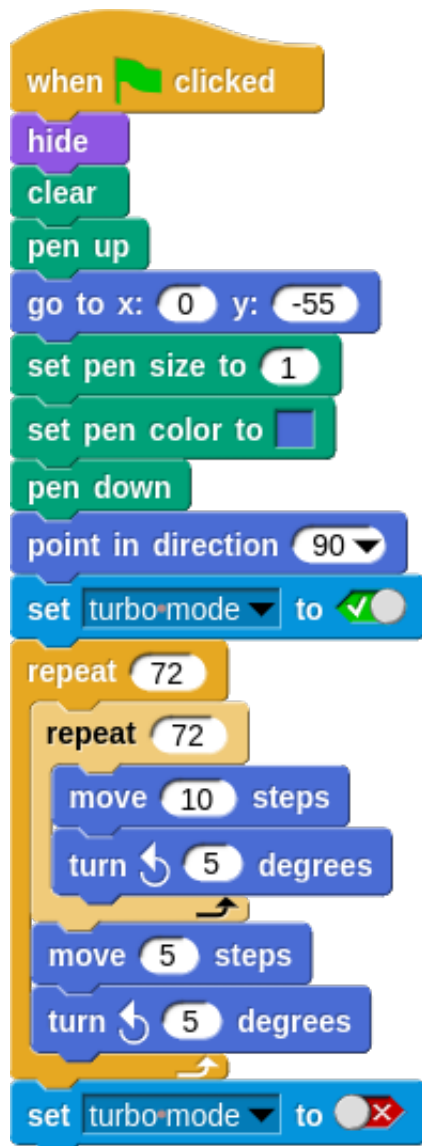
Scratch で高速に実行するためのターボモードが Snap! にもあります。

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

Sensing パレットに  ブロックがあります。これを turbo mode にして六角形の部分を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。

warp (ワープ) ブロックでスクリプトを囲むと、さらに速く処理することができます。

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。ターボモード版は画面のクリアと描画の様子が確認できますが、ワープ版は確認できません。



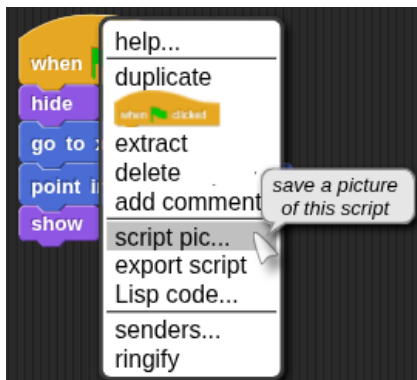
2.4 ブロック表示

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

このような長い式を Snap! でスクリプトにすると、



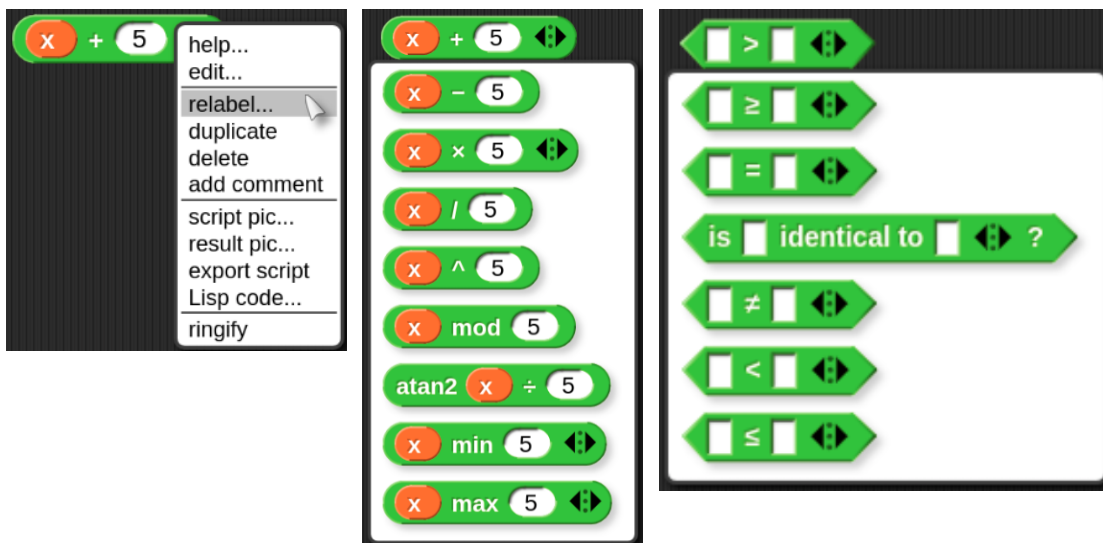
となります。長くなった場合は、自動的に折り畳まれます。また、同じパレットのブロックが重なった場合は、色合いが交互に変化して表示されます。ゼブラカラーリングだそうです。



スクリプトをプリンターで印刷したい場合は、スクリプトを右クリックすると、script pic ... で、画像ファイルとしてダウンロードフォルダーへエクスポートされます。

ファイル名はプロジェクト名 + script pic + (番号) + 画像ファイルを表す拡張子になります。

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

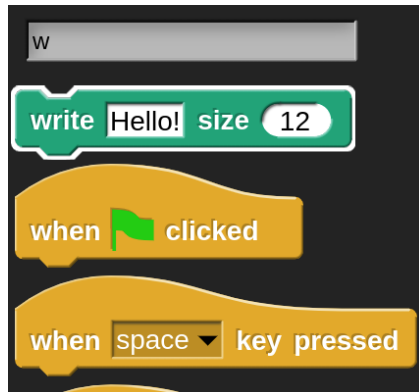


3 キーボード入力

スクリプトエリアには  があります。これをクリックするか、**[Shift]+[Enter]** 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。

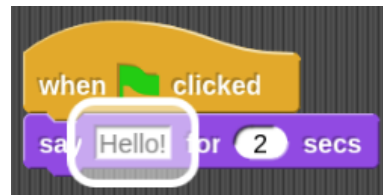
二つの計算結果を表示するスクリプトを作ってみます。



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 **[w]** を打ちます。すると、パレットエリアに左図のように表示されるので、**[↓]** と **[Enter]** で選んでください。

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



ここで **[Enter]** 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。**[Enter]** 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。

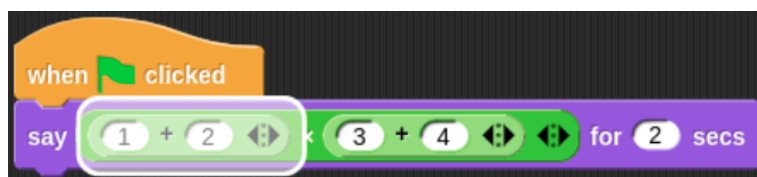
「**(1+2)*(3+4)**」と入れてみてください。パレットエリアに入力に応じて式のブロックが表示されます。**[Enter]** で決定です。



16/4/2



16/(4/2)

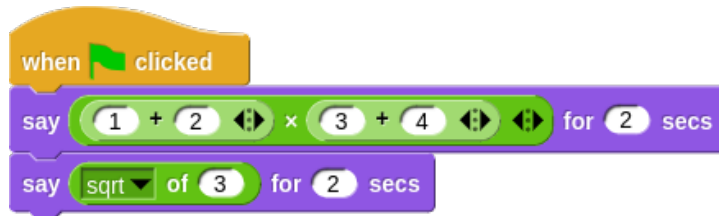


[←] **[→]** で入力スロットを移動して変更できます。**[↓]** で次のブロックに移ります。

次に、ルート 3 の値を表示させてみます。ルートは sqrt で求めます。

また、say Hello! for 2 secs を出してください。

say の最初の入力スロットの部分で、 (of の「o」です) と打ちこんで  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  +  +  で実行できます。



どこかをクリックするか  などでもキーボード入力モードから抜けれます。

4 変数

変数を作成するにはいくつか方法があります。

-  をクリックする。
-  を利用する。
- block variables を利用する。(ブロック内変数 64 ページ参照)
- for ブロックなどの変数を利用する。
- .txt .csv .json などのファイルを Snap! 画面にドロップする。

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



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

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



〔 Make a variable 〕 をクリックすると、



が出来ますから for all sprites を選んで、varAll



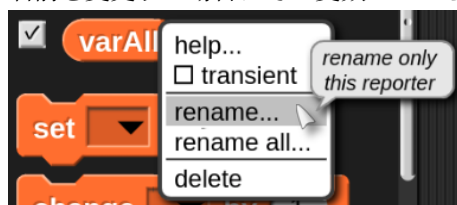
という変数を作ります。そうすると、パレットエリアに
こうして作られた変数は全部のスプライトで使用できます。このようにどこからでも使用できる変数をグローバル変数とか大域変数と言います。

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

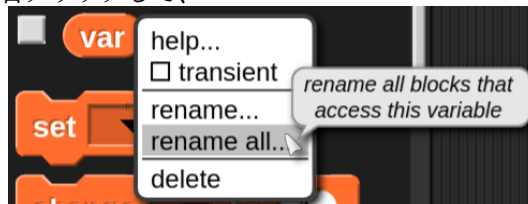


変数を削除する場合は、
で Delete a variable をクリックすると登録されている変数が表示されるので、そこから選んで削除します。

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、

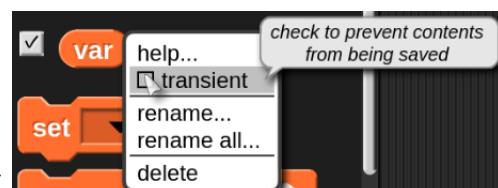


か



で変更

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箇所を変更します。



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。

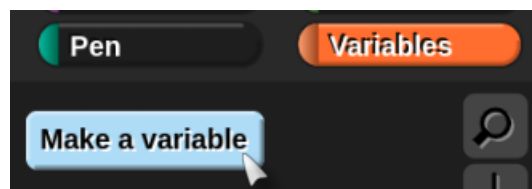
スクリプトが何もない状態で、変数 var を for all sprites を選んで作成します。この段階では var は、 です。


 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 8.5KB でした。これを実行すると、
変数 var に 10000 個の要素のリストが入ります。

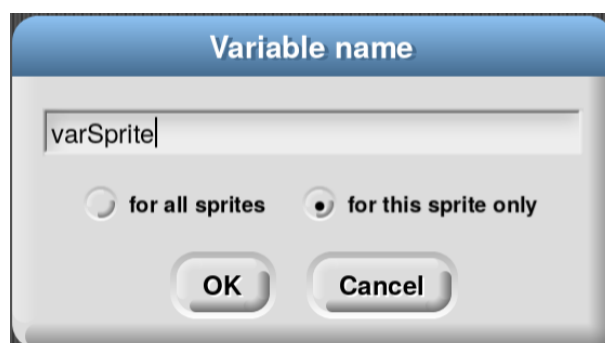
この状態で save します。すると、ファイルの容量は 59.8KB でした。この保存したプロジェクト
を改めて Open... で読み込むと、保存された時の変数の値になっています。つまり、何もしな
いと変数が値ごと保存されてプロジェクトファイルの容量が大きくなるということです。transient
をチェックして save すると、ファイルの容量は 10.9KB でした。この数値は実行環境など状況に
よって違うかもしれません。この保存したプロジェクトを改めて Open... で読み込むと、変数 var
の値は初期値 0 になっています。これが transient の機能です。このスクリプトの場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

グローバル変数はどこからでも使えて便利なのですが、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。

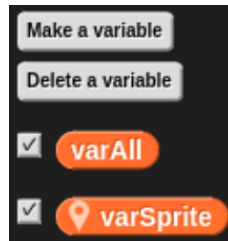
4.2 for this sprite only スプライト変数



〔 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** a のところをクリックすると変えられます。



a の隣りにある右向きの三角をクリックすると変数を追加できます。

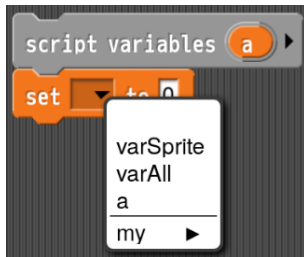


左向きの三角をクリックすると削除できます。

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



このように、変数リストに表示されません。

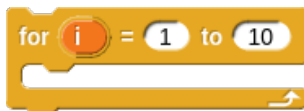


このように接続されてからでないと使えません。



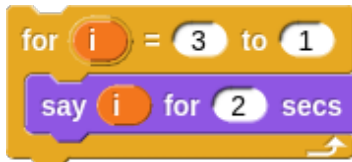
このようにその前でも有効になりません。

4.4 Upvar 変数

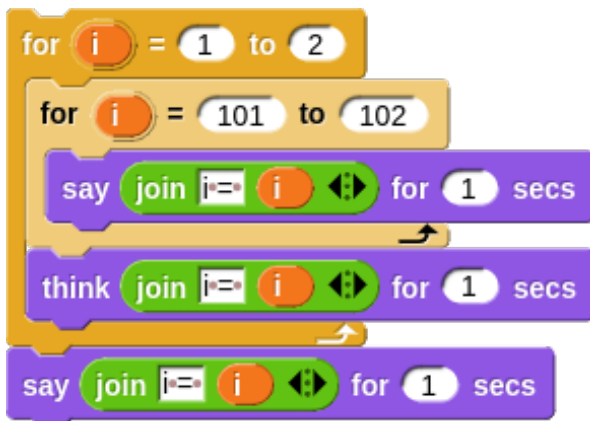


for ループのようにあらかじめ用意されている変数があります。

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



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



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



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

4.5 リスト操作ブロック

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。

Make a variable
Delete a variable
☒ alist

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



set aList to numbers from 1 to 5

を使うと、

set aList to list
for i = 1 to 5
add i to aList

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

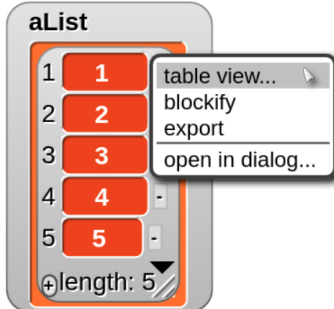
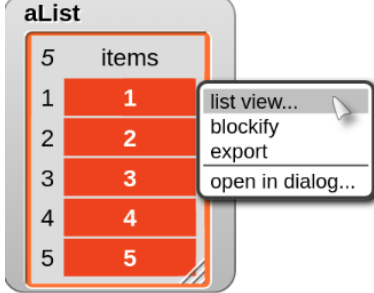


table view... を選択すると



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



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



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



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

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

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

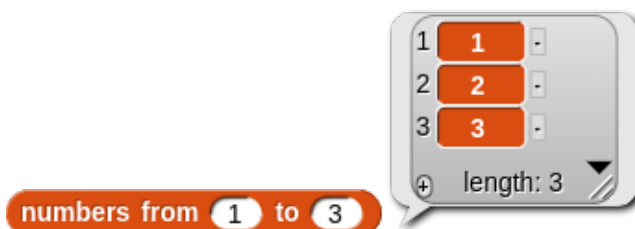


たとえば、**aList** や **list** のように。なお list view では、一度に扱える要素数は 100 までになっています。次の 101 から 200 の要素に移るには、下向きの三角をクリックすると出てくる範囲から選びます。

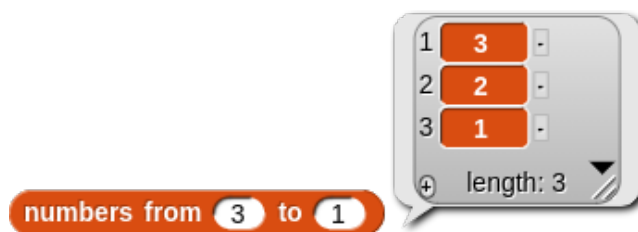


4.5.1 numbers from () to ()

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レポート



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レポートします。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レポートします。指定されたリスト自体は変更しません。

4.5.4 index of () in ()

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



4.5.5 delete () of ()

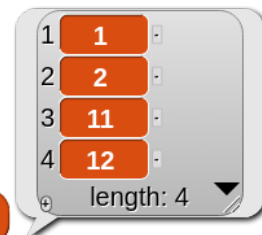
これは、指定されたリストから指定された位置の要素を削除します。



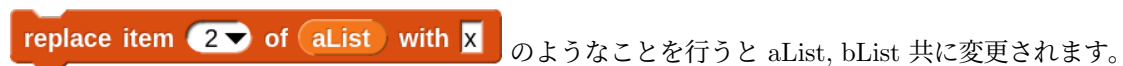
位置が要素数よりも大きい場合は何もしません。ただし、位置を 0 とすると last を指定したのと同じになってしまいます。-1 にすると、最後から -1 番目、-2 にすると、最後から -2 番目ということになります。要素数 -1 以上の値を負数にした値、要素数 5 の aList では -4 以下の数値を指定すると、1 番目の要素を削除することになります。

4.5.6 append () ()

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



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



4.5.7 for each () in ()

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



4.5.8 reshape () to () ()

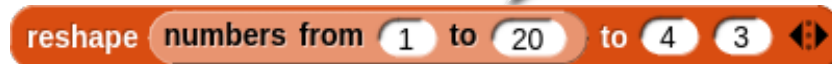
リストの要素にリストを含むものを配列、縦横二次元的なものを行列と言います。

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



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

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



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

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



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。



4.5.9 map () over ()

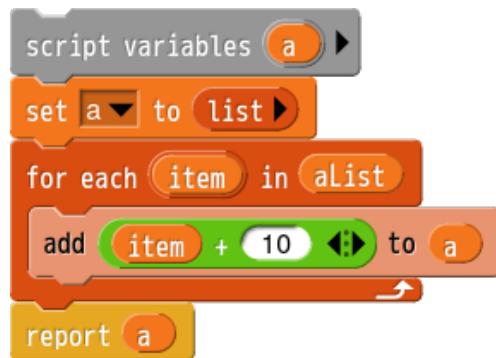


これは、指定されたリストの各要素に対して指定された操作を行ったリストをレポートします。



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

for each で同様のことをすると、次のようになります。



入れ子になったリストも map を入れ子にすることで対応できます。

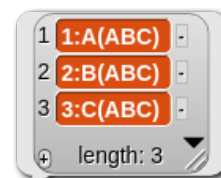
	A	B	C
2			
1	2	3	
2	4	5	6



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

value には指定されたリストの要素が順番にセットされます。index はリストの何番目の要素を処理しているかを示します。list は指定されたリストを表します。

value = item (index) of (list) の関係になっています。



次のようにすると特定の値の要素を入れ替えることができます。



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

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

まずは簡単にするために aList の各要素を 1 にします。



次のようにして list の要素の値を書き換えています。そうしないと、 $(n-1)$ 項 と $(n-2)$ 項 の値を得られないからです。



その結果、aList の要素の値も入れ替わります。

(入力したリストが変更されることを示すために変数を使いましたが、**reshape 1 to 7** を使用することもできます。その場合は、それによって作成されたリストが変更されます。)

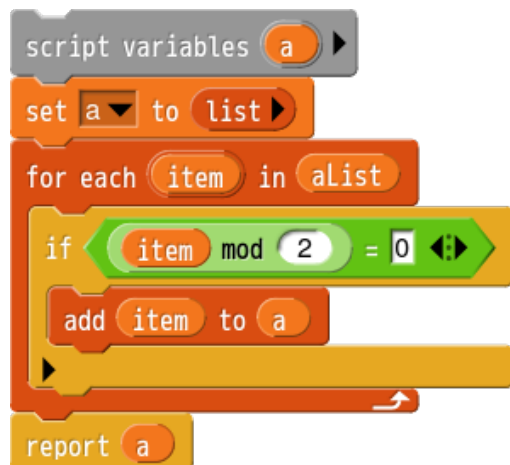


4.5.10 keep items () from ()



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

for each で同様のことをすると、次のようになります。

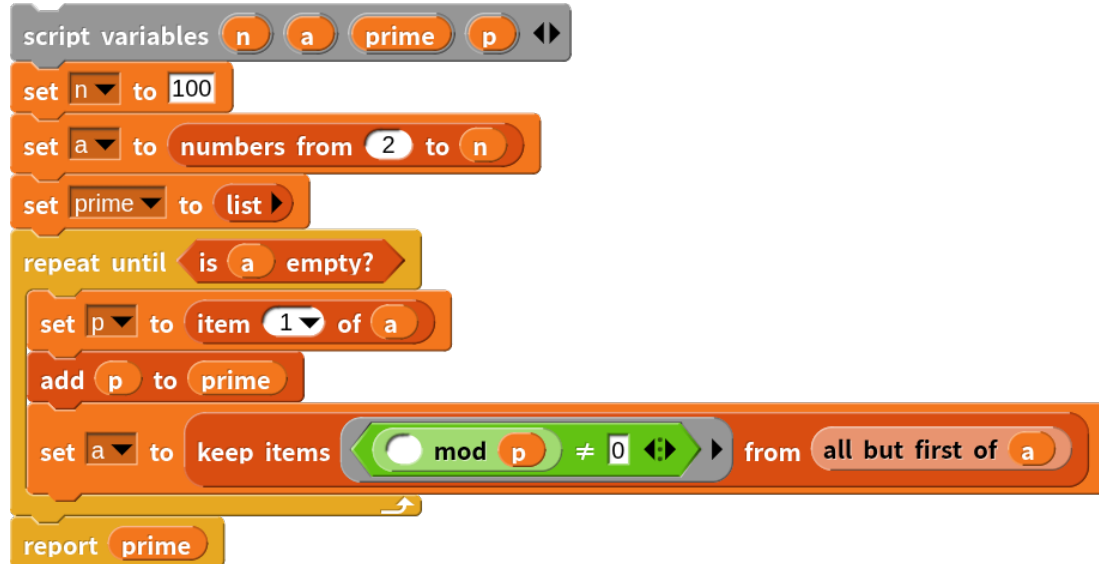


delete 1 of のブロックでリスト内の位置を指定して要素を削除できますが、次のように **not** を使用すると、指定した条件のものを削除したリストが得られます。



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

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



2000 年から 2200 年までのうるう年のリストを求めてみます。

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



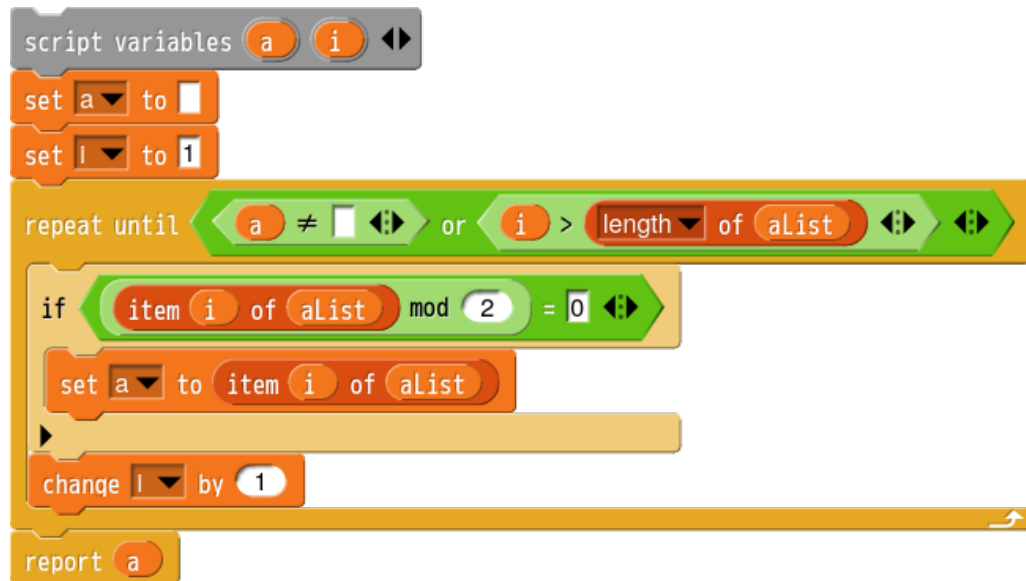
空の入力スロットが複数ヶ所あります。2000 年の場合、すべてにリストの要素である 2000 が入ります。フォーマルパラメータ value をすべてにセットするのと同じです。

4.5.11 find first item () in ()



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

repeat until で同様のことをすると、次のようになります。



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

4.5.12 combine () using ()

combine using は、指定された演算（引数が 2 つ）をリストの要素に行い値をレポートします。指定されたリスト自体は変更しません。

combine numbers from 1 to 5 using + は、指定されたリストの合計をレポートします。1 + 2 + 3 + 4 + 5

for each で同様のことをすると、次のようになります。



109 ページで述べているように、演算ブロックの右端の三角のところにリストを入れてやることでリスト全要素に対しての演算の値を得ることができるので、そちらでほぼ処理できます。

sum numbers from 1 to 5 15

combine numbers from 1 to 5 using × 120 は、5 の階乗を求める操作になります。 1 × 2 × 3 × 4 × 5 120

product numbers from 1 to 5 120

pipe (48 ページ参照) ブロックを使った処理でも代用が可能です。

pipe numbers from 1 to 10 → combine using + 55

pipe numbers from 1 to 10 → sum list input names: list

リストに対応していないユーザー定義ブロックでは、combine が必要になります。

ユーザー定義ブロック「最大公約数」は 2 つの引数を受け取り、ユークリッドの互除法を使ってその最大公約数を求めるものです。

最大公約数 45 75 15

combine を使用すると、リストの全要素の最大公約数を求めることができます。

combine list 84 210 196 98 126 using 最大公約数 14

```
+ 最大公約数 + m # + n # +
script variables t
repeat until n = 0
  set t to m mod n
  set m to n
  set n to t
report m
```


4.5.13 combinations () ()

指定されたリスト同士を順番に組み合わせたリストをレポートします。

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

combinations list 1 2 list 1 2

for each で同様のことをすると、次のようになります。

```
script variables a
set a to list
for each item_1 in list 1 2
  for each item_2 in list 1 2
    add list item_1 item_2 to a
report a
```

x 座標、y 座標 それぞれ取り得る値が (-100, 0, 100) と (-80, 0, 80) の場合にすべての組み合わせからランダムに選んだ座標へ移動するスクリプトです。

```
set aList to combinations list -100 0 100 list -80 0 80
forever
  go to item pick random 1 to length of aList of aList
  wait 1 secs
```

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

4.6 オプションのリスト操作

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

4.6.1 rank of (), dimensions of ()

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

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

a 0 のようにリストではない場合です。

rank ▼ **of** **a** 0

dimensions ▼ **of** **a** length: 0

a length: 3

1	1	-
2	2	-
3	3	-

要素にリストを含まないリストの場合です。

rank ▼ **of** **a** 1

dimensions ▼ **of** **a** length: 1

set a ▼ **to** **reshape** **numbers from** **1** **to** **6** **to** **2** **3** **↔**

a length: 2

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

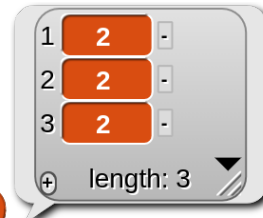
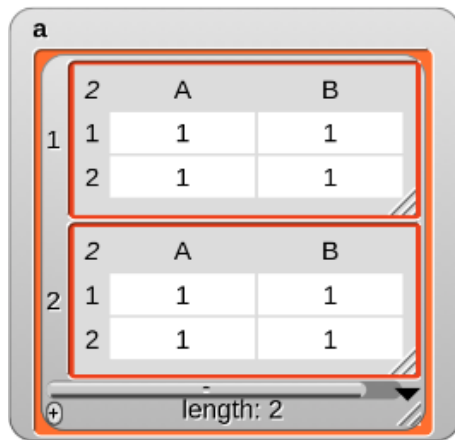
行列（マトリックス）の場合です。

reshape で指定した形が dimensions の値になります。

rank ▼ **of** **a** 2

dimensions ▼ **of** **a** length: 2

rank は dimensions のリストの要素数を示します。



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

4.6.2 flatten of ()

rank 2 以上のリストを rank 1 に整形してレポートします。



4.6.3 columns of () 転置行列

リストの行と列を入れ替えた転置行列をレポートします。



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

aList

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

columns of aList

4.6.4 uniques of ()

リスト中の同じ要素を削除してリポートします。
頻度順になるようです。

1	3
2	2
3	1

length: 3

uniques of list 1 2 2 3 3 3

4.6.5 distribution of ()

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

3	A	B
1	c	3
2	b	2
3	a	1

distribution of list a b b c c c

4.6.6 sorted of ()

リストの要素を整列してリポートします。
数値は後方になるようです。

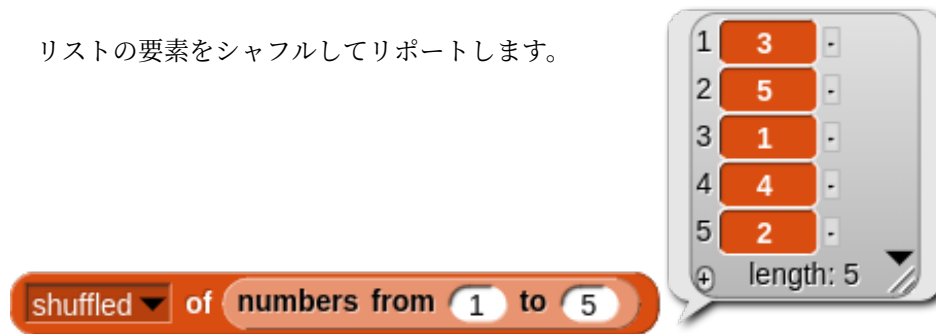
1	a
2	c
3	e
4	1
5	2
6	5

length: 6

sorted of list 2 5 1 e a c

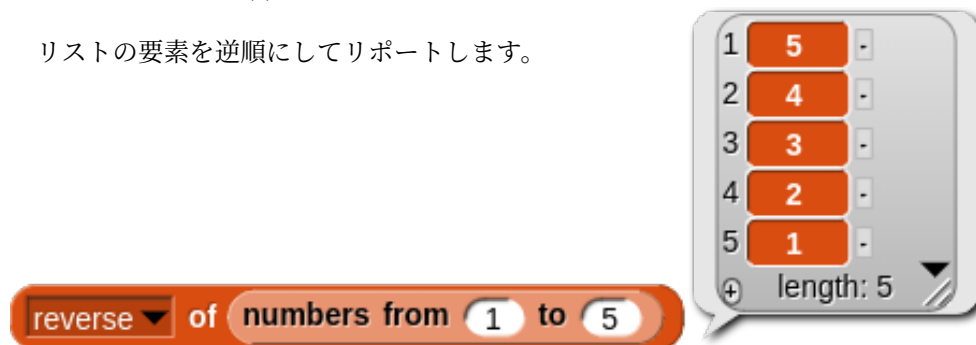
4.6.7 shuffled of ()

リストの要素をシャフルしてリポートします。



4.6.8 reverse of ()

リストの要素を逆順にしてリポートします。



4.6.9 text of ()

リストの要素を空白を間に入れた文字列としてリポートします。



4.6.10 lines of ()

リストの要素を文字列行としてリポートします。

これが使えるのは rank 1 のリストです。



4.6.11 csv of ()

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

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



4.6.12 json of ()

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

rank 3 のリストにはこちらを使うことになります。



4.7 リストの演算

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

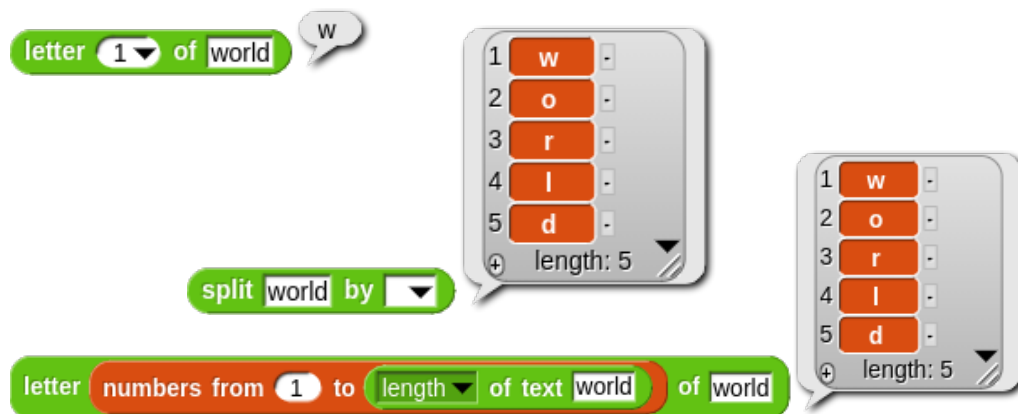




リスト同士で演算をすることもできます。いずれも同じインデックスの要素同士を演算します。要素数が合わない場合は対応する部分だけを行います。



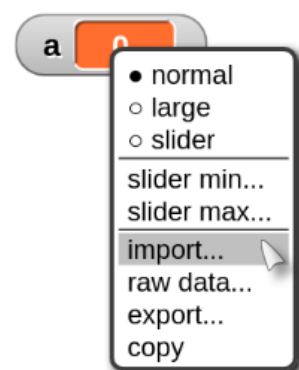
リストに文字列の各要素を入れることもできます。





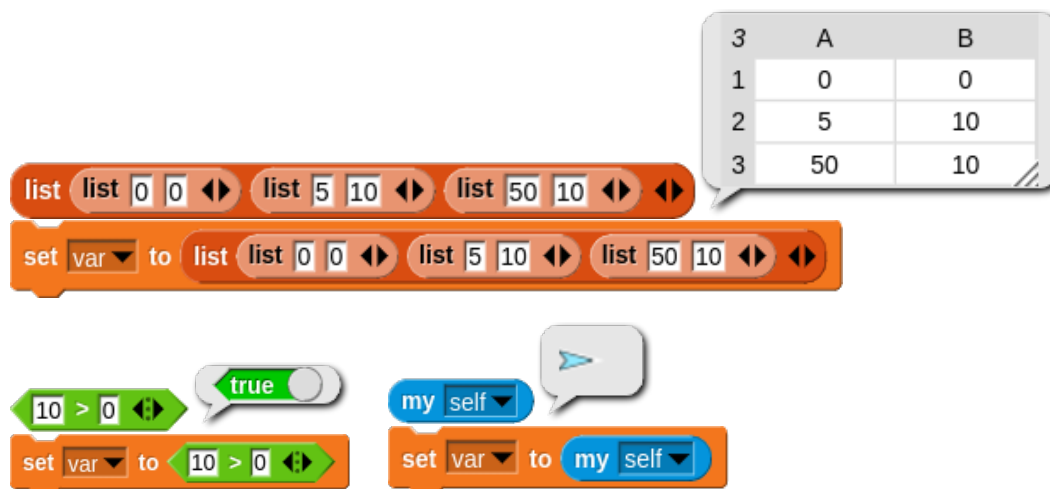
4.8 変数に入れられるもの

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。

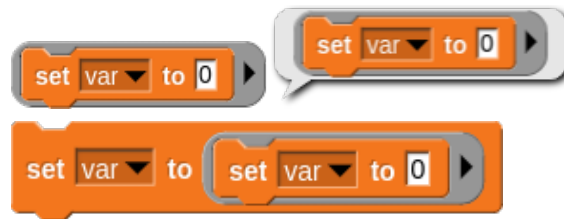
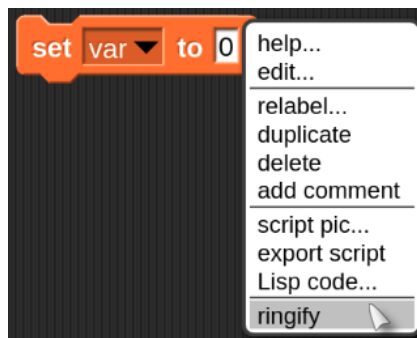


「import...」で、指定したファイルの内容を取り込むことができます。.csv や .json のファイルの場合は、そのデータ形式に従ってリストを作成します。.csv ファイルではカンマと改行をセパレーターとした要素をリストにします。単なるテキストデータとして取り込みたい場合は、「raw data...」を使います。

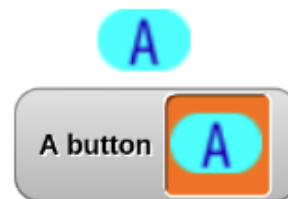
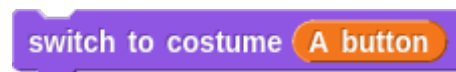
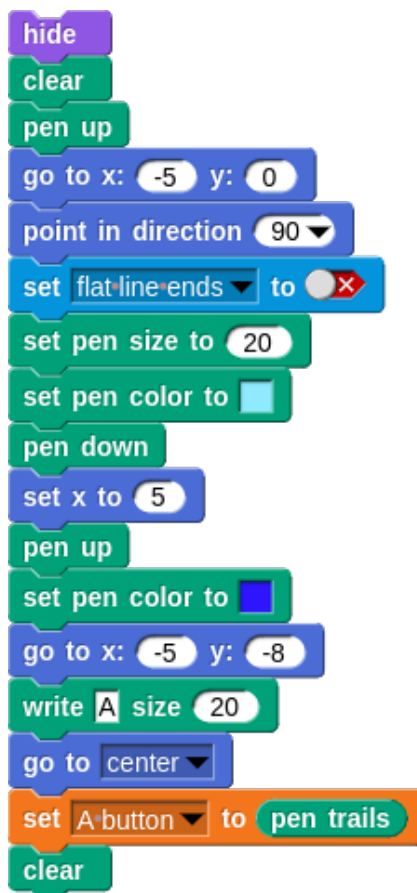
Snap! の変数には、値をレポートするものを入れられます。



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



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



5 Control 制御

Snap! で使用できる制御ブロックについて。

5.1 リポーターの if < > then () else ()

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



5.2 when ()

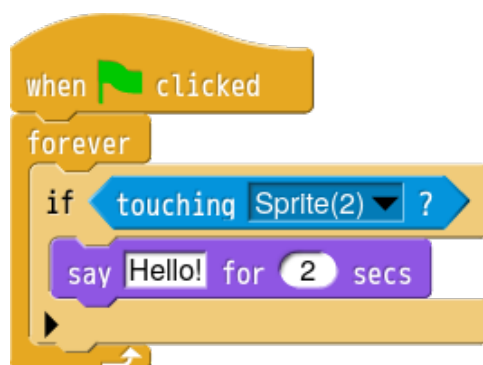


はずっとチェックを続け、指定された条件ブロックが True 真になると

実行します。

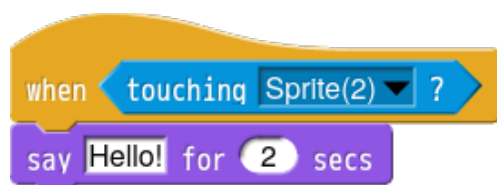


Sprite, Sprite(2) の二つのスクリプトがあって Sprite(2) が Sprite に触れた時になにかをさせる場合、次のようなやり方があります。



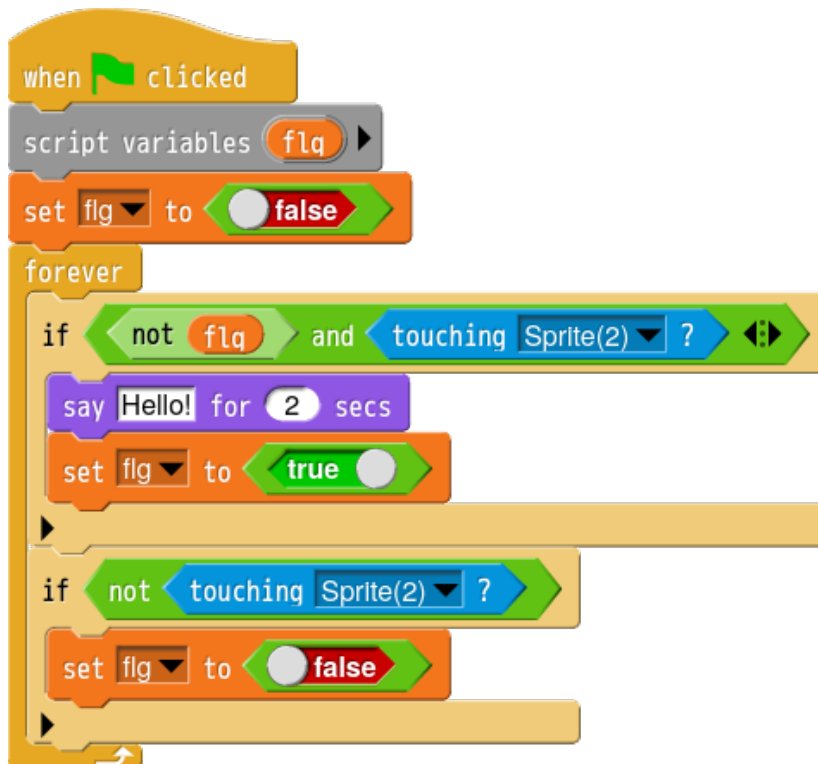
この場合は、触れている間ずっと say が実行されます。

これに対して when を使うと、

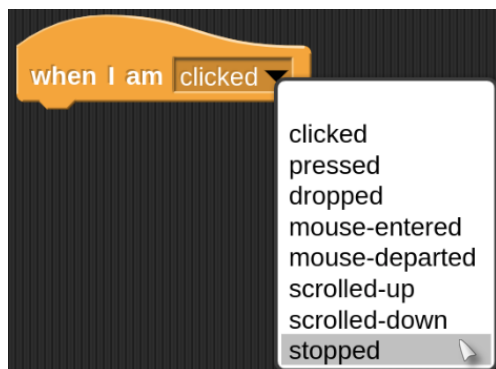


触れた時 1 回だけ実行されます。離してまた触れれば実行されます。

これは次のスクリプトと同じ動作です。



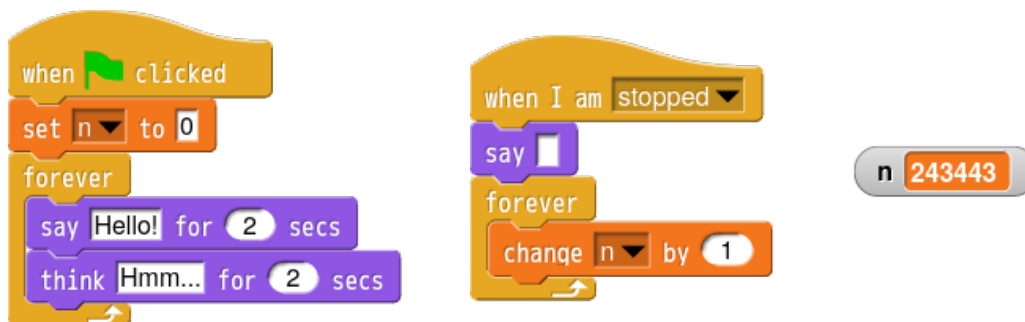
5.3 stop ボタンがクリックされた時の終了処理



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。たとえば、接続されているロボットのモーターを止めるとかなどです。

普通のプログラムでも  ボタンによる終了時に画面を clear したり、スプライトを hide したりと、散らかった画面を整理して終わらせることができます。

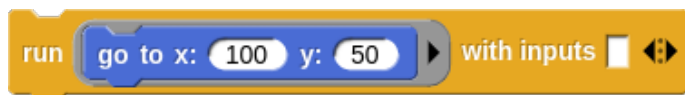
次のスクリプトで動作の確認ができます。変数 n を ストップ後に +1 しています。けっこうな回数 n のカウント（スクリプトの実行）ができるようです。動作環境によって値は変わります。



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、run go to x: 100 y: 50 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

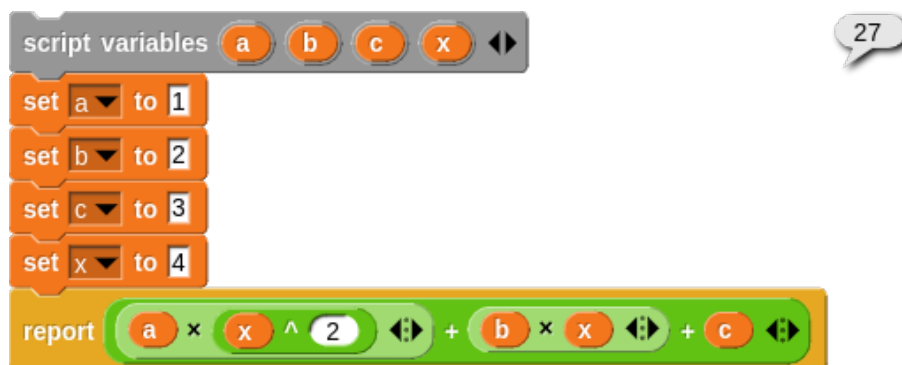
run ブロックに入れられるのが値をレポートしないスクリプトブロックだったのに対して、call ブロックに入れられるのは、実行または評価することによって何らかの値または true か false をレポートするブロックです。call ブロックは、得られた値をレポートします。



call ブロックを使ってちょっとした関数のようなもの（入力に対応する値を返すブロック）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ようになります。



変数の入力スロットを空にして、式をリングで囲みます。



リングの端の三角をクリックしてフォーマルパラメータを出します。



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

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



このリングを変数に入れてやれば定義ブロックのようにも使えます。



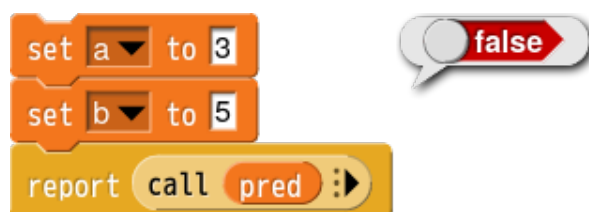
a = 2, b = 1 の時に  とすると変数 pred は次のようになります。



pred にはその時点の a, b の値でテストした結果の true がセットされているだけなので次のようにしても pred の値は変わりません。



 とすることによって、テストを行うことができます。



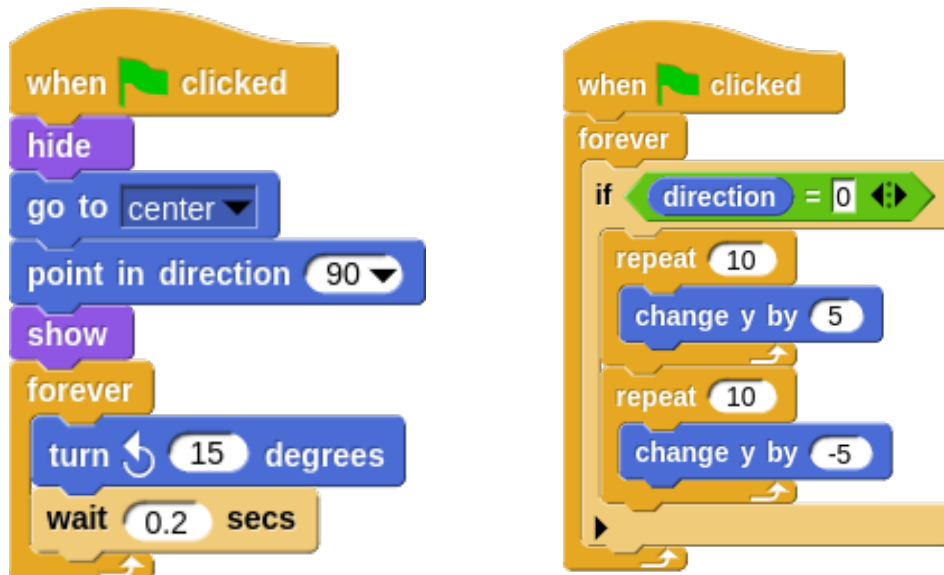
run や call ブロックは、ユーザー定義ブロックなどで入力されたブロックを実行したり値を求めるために使用されます。

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。

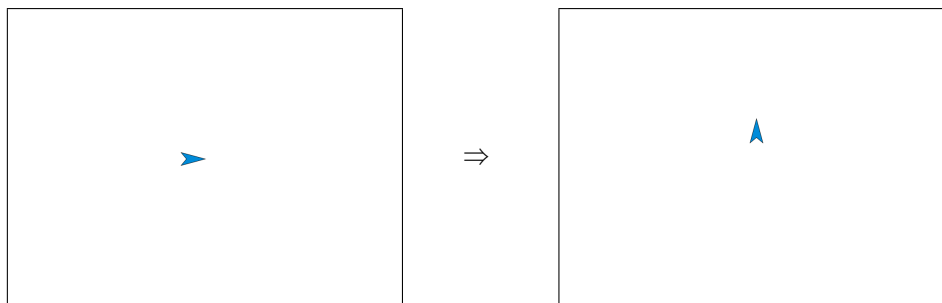
1 個のスプライトに対して、回転させるスクリプトと、角度が上（0 度）の時にジャンプさせるスクリプトを並列で実行してみます。

並列実行は、普通次のようにして 2 つのスクリプトを同時に実行します。

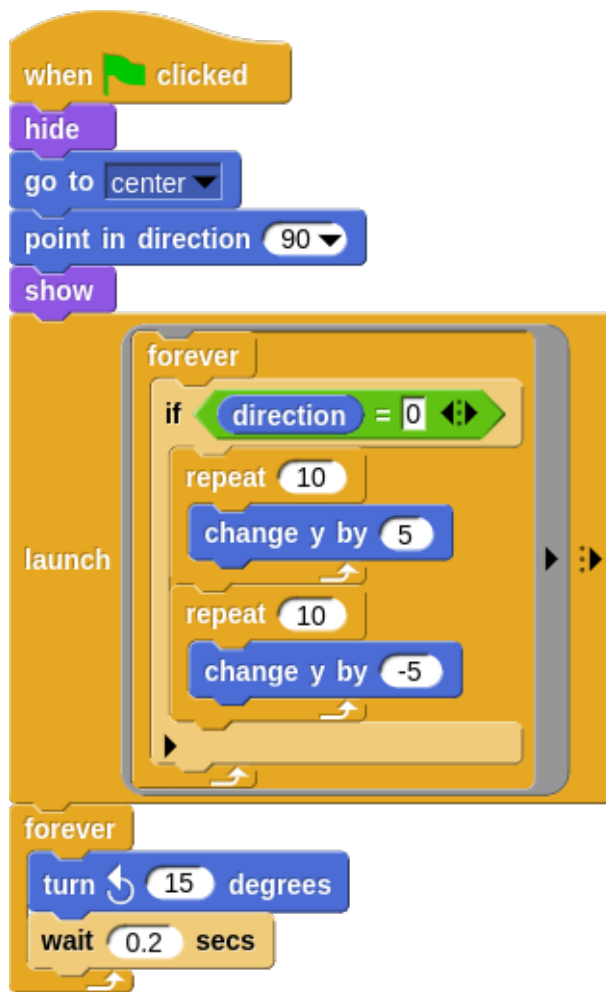


左のスクリプトで、ずっと回転させる動作を行います。

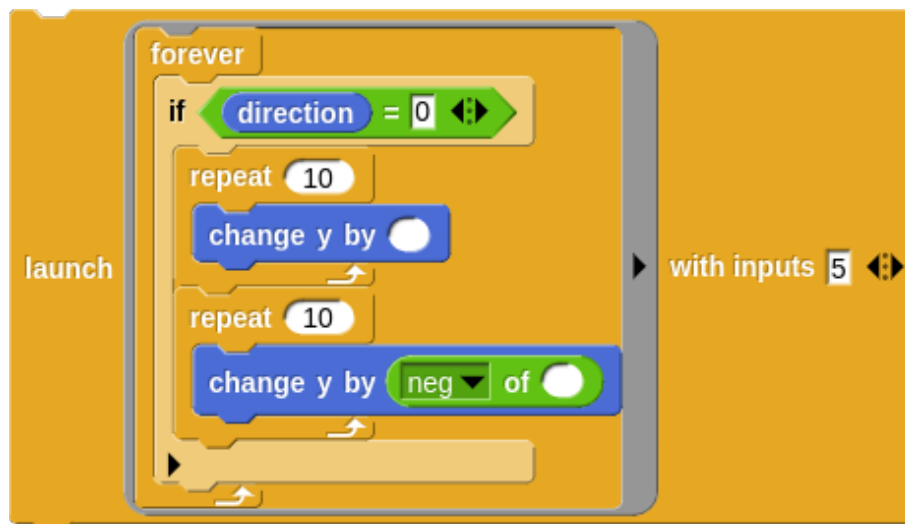
右のスクリプトで、上を向いた時だけジャンプする動作を行います。



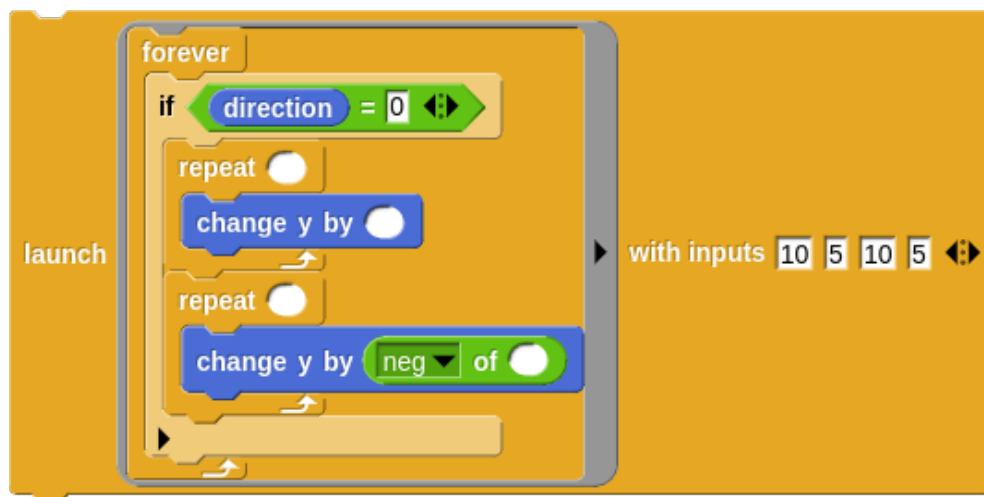
launch を使うことで 1 つのスクリプトで実行することができます。



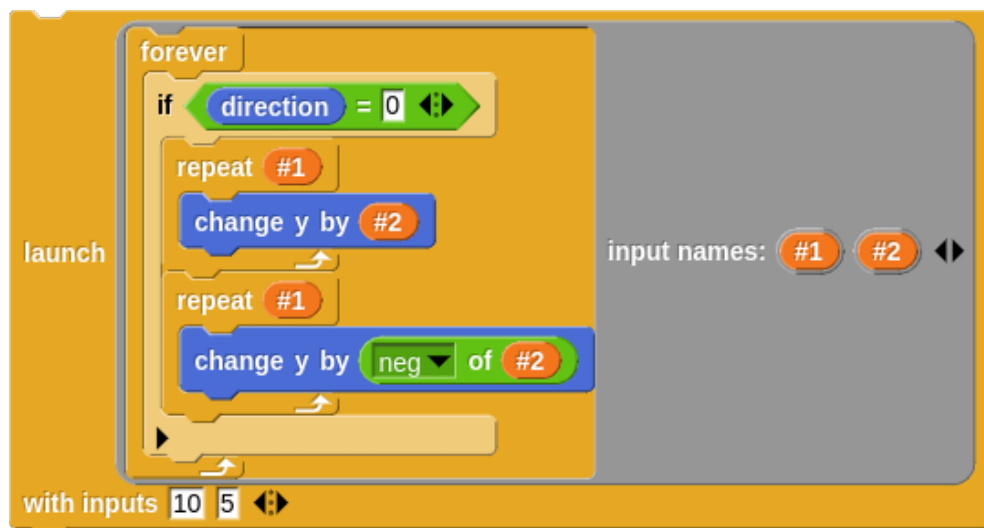
launch の右端の右向き三角をクリックすると、入力値を設定することができます。y の値の設定を空（空白ではなく）にして、入力値をひとつだけ設定すると、その入力値がすべての空の部分の値になります。



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

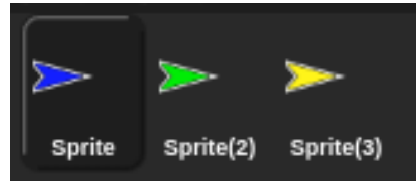


この場合は、リングではフォーマルパラメータが使えるので、次のようにすることができます。リング、灰色の部分の右端の三角をクリックして外側の入力 (10, 5) の個数と同じ個数の変数を用意します。対応する位置に目的の変数を置きます。



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

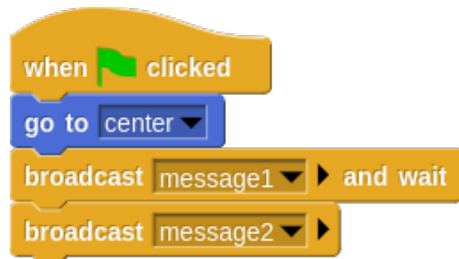
5.7 broadcast ブロック, tell to ブロック, request ブロック



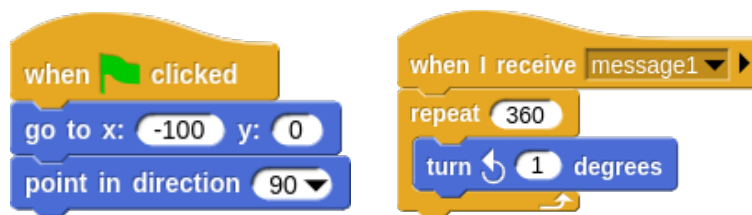
次はスプライトを3個使います。 を用意してください。

Sprite から司令を出して Sprite(2) と Sprite(3) を順番に動かします。 broadcast を使うと次のようになります。

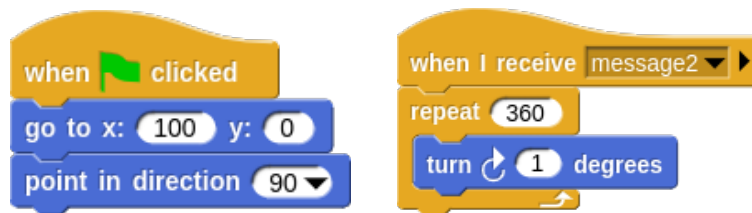
Sprite 用



Sprite(2) 用



Sprite(3) 用

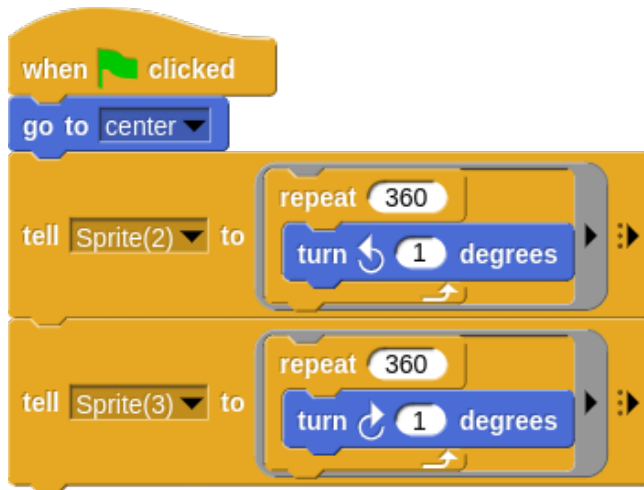


broadcast and wait ブロックは、メッセージを受け取ったがわの when I receive で指定されたスクリプトの実行が終わるまで待ちます。



のブロックを使用すると broadcast を使わなくても Sprite から Sprite(2), Sprite(3) を直接操作することができます。

Sprite 用

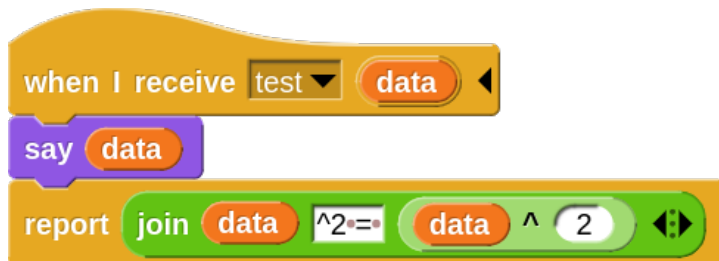


ver. 11 から broadcast and wait の機能を持つ request ブロックが追加されました。メッセージを受け取った側で値を返し、送信した側ではそれを受け取ることができます。

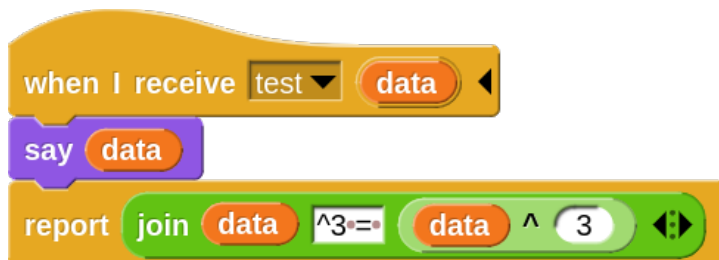
Sprite 用 （すべてのスプライトに対して 5 をデータとしてメッセージを送ります。）

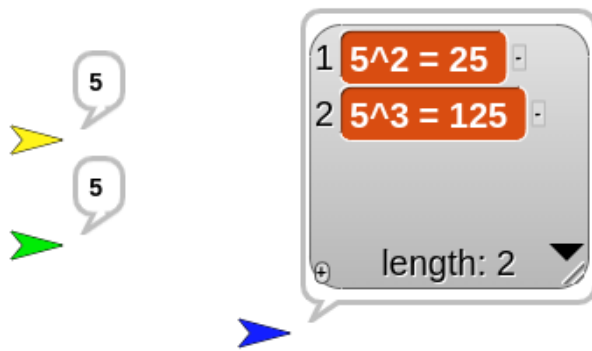


Sprite(2) 用 （受け取ったデータを 2 乗して返します。）



Sprite(3) 用 （受け取ったデータを 3 乗して返します。）





返されたデータはリストにまとめられますが、期待した順番にはならないかもしれません。

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って別な処理をして出力する pipe (パイプ) というデータのリレーのようなことが行われます。

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

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

"We have been ~ at the heart of Snap!." をコピーしてエディターで "SnapText.txt" という名前で保存します。それをスクリプトエリアにドロップすると、SnapText という変数が作成されます。変数 SnapText は文字列をリポートします。文章から単語に分解するには **split** [] by **word** [] です。単語の抽出は空白や改行などでなされるようなので辞書的な正確さはありません。リストの要素数を求めるには **length** [] of [] です。順番に実行してみてください。

pipe SnapText → [] []

pipe SnapText → **split** [] by **word** [] []

pipe SnapText → **split** [] by **word** [] [] **length** [] of [] []

67

受け取った値は指定のスロットに入って処理されます。

一つの入力を受け取って処理し一つの値を出力するというリレーです。それぞれを小さくて分かりやすい処理単位にすることで、テストもしやすく見通しの良いプログラムになります。因みに、UNIX(Linux) ではテキストを出力するのが cat で、単語数を数えるのが wc -w です。「|」がパイプを表します。UNIX では二つの処理しか必要としませんが、Snap! と同じ結果になります。

```
[~]$ cat SnapText.txt | wc -w
67
```

6 ブロックを作成する

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

6.1 定数の作成

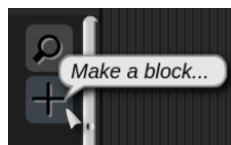
一般的なプログラミング言語では値が変更されない定数を定義することができます。Scratch では、変数で代用するしかありませんでした。



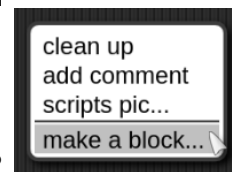
変数は変更されることを前提としているものなので定数として使用するのはいままでのScratchでは好ましくないことではあります。piは良いとしてもeなどは普通の変数と思われてしまいそうです。

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

ブロックを作成するには、パレットエリア内にある **Make a block**、

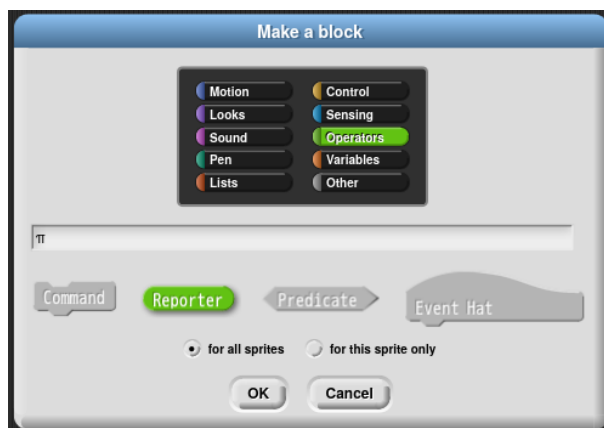


、スクリプトエリア内で右クリックすると出てくる
のど
れかの **make a block** をクリックすれば始められます。



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もかまいません。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。カテゴリで Other「その他」というものを選ぶと、置き場所は Variables のところになります。Command は、値をリポートしないブロックです。Reporter は、ブロック内で設定した値をリポートします。Predicate は、述語と訳されます。true か false をリポートします。Event Hat はなんらかのイベント発生用です。(104 ページ参照) それぞれの形が、できたブロックの使い方を示しています。ブロック名入力欄に π を入れて、上の欄でボタン Operators を、下の段で Reporter を選択してください。Operators を選択することはパレットの種類を決めることであり、置き場所を決めることでもあります。

Ok をクリックするとブロックエディターが表示されます。



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

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

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

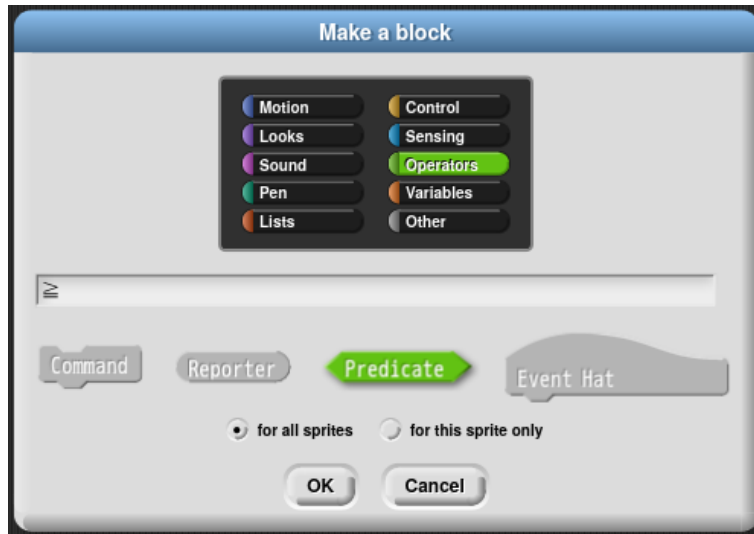


同じように e も作成することができます。



6.2 () ≥ () ブロック

引数を使用する例として () ≥ () のブロックを作ってみます。11 ページで示したように ≥ ブロックがあるので必要ないですが、最初の練習としては適当です。

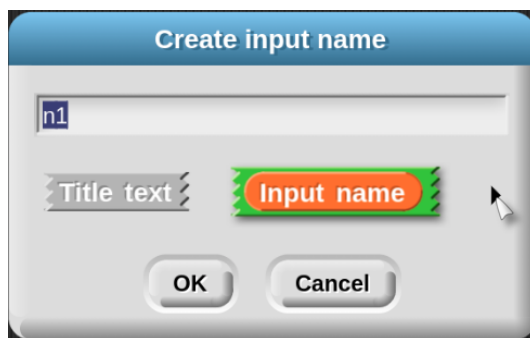


ブロック名入力欄に ≥ を入れて（全角文字「きごう」とか「だいたいいいこーる」で変換できます）、上の欄でボタン Operators を、下の段で Predicate を選択してください。プリミティブの「>」ブロックと同じにします。Predicate は形から分かるように、Control コントロールブロックの条件式のところなどで使用されて true または false を返すためのものです。

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。



「≥」の左側の + ボタンをクリックしてください。



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

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



現在は  Any type が選択されています。Any つまり数値でも文字でも受け付けるタイプです。このままでもいいのですが、あえて  Number にしてみます。これは数値しか受け付けないタイプです。Any type を選んだ場合は、変数の表記に「#」が付かないだけで以下の操作は同じです。もう一つ、下の方に  Single input. Default Value: が出来ます。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。



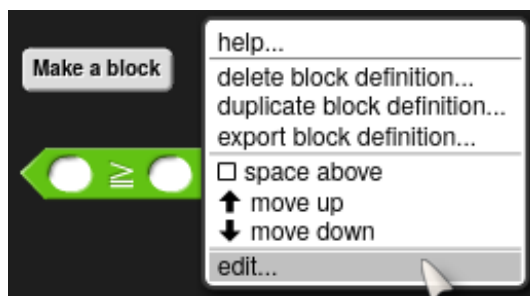
パレットエリアからブロックを持ってきて完成させてください。



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。



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



ブロックを右クリックすると、edit... で下記のように内容の編集ができます。

export block definition をクリックすると、このブロック定義だけをファイルに書き出すことができます。



変数 n1# を左クリックすると、変数名や設定の変更をすることができます。

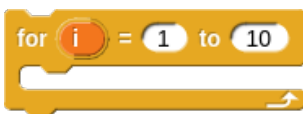
 を使ってステップ実行する場合に、作成したブロックの内部をステップ実行させたければ  オンにしてからそのブロックをブロックエディターで表示させておく必要があります。順序が逆だとそのブロック内のステップ実行はされません。(84 ページ参照)

6.3 help 説明文の作成

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



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

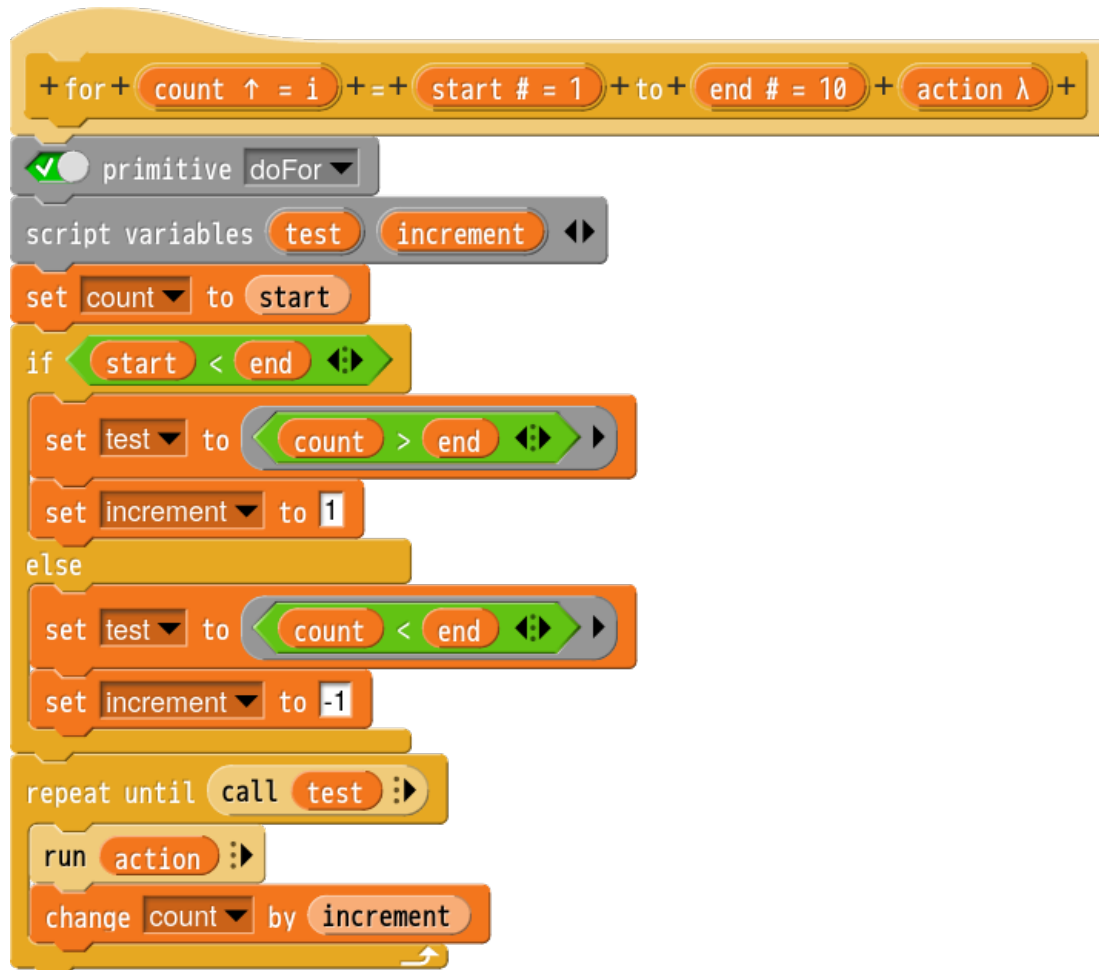
Snap! には  ブロックがあります。これを基にして増減分が指定できるものを作成してみます。

増減分を指定する位置は 3 通りほど考えられます。



今回は 2 番目のブロックを作成します。

次のように既存の for ブロックのところで右クリックして edit... を指定します。そうすると、for ブロックの定義が表示されます。



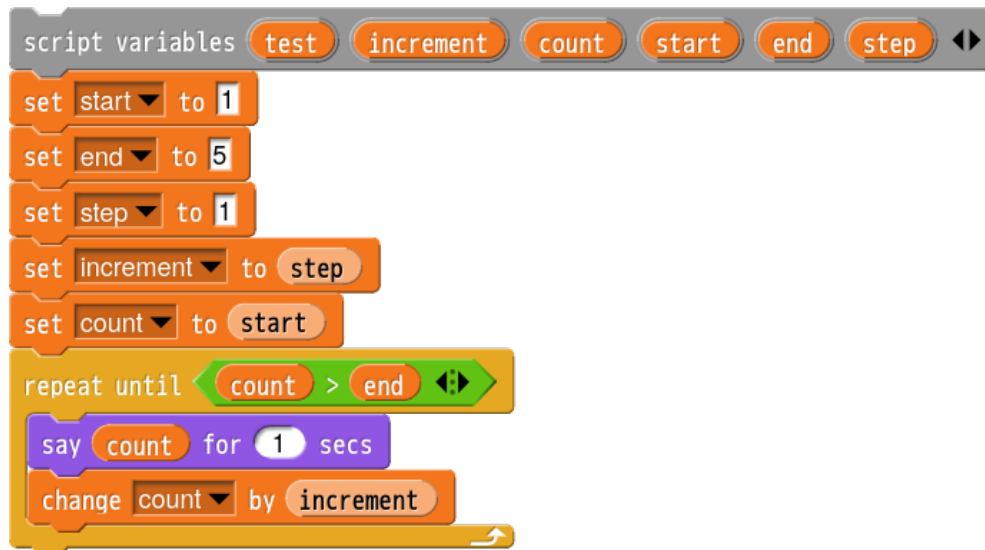
このスクリプトを基に、my for (i) = (start) to (end) step (step) というブロックを作成します。

ここで、 はそれ以降のスクリプトは使わず内部の定義を実行することを示します。これを  にすると、内部での定義は使わずにそれ以降のスクリプトを実行することになります。この定義ブロックを変更するだけでは練習にならないので、別な定義ブロックとして作成します。

この定義を見ると、increment という変数があり、これによって count が増減されています。これに step の値をセットすれば求める動作になりそうです。

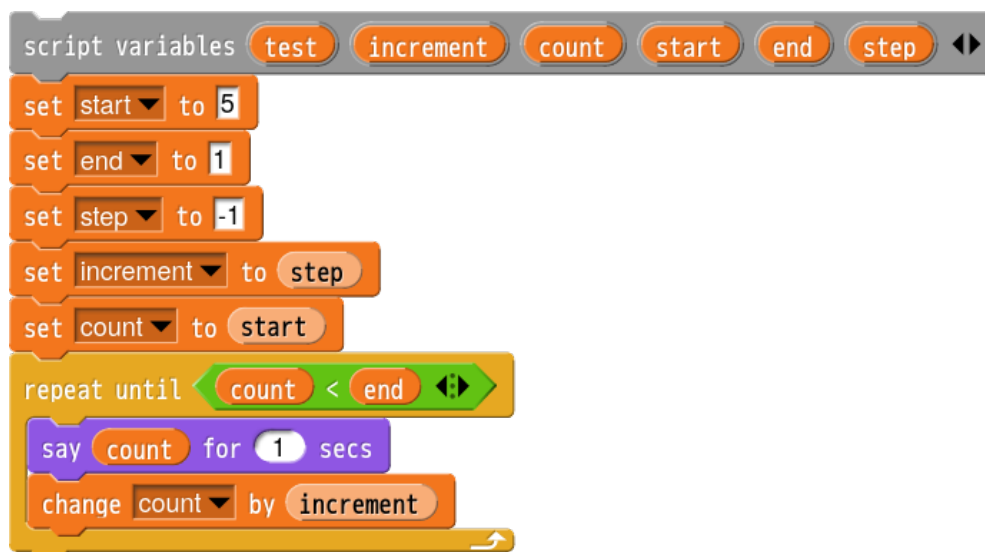
まずは、スクリプトエリアで for ブロックの動作を確認しながら組み立ててみます。
increment が増加していく場合です。

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



start を 5、end を 1、step を -1 にすると、 $5 > 1$ で、 の終了条件を満たしてしまうので実行されません。

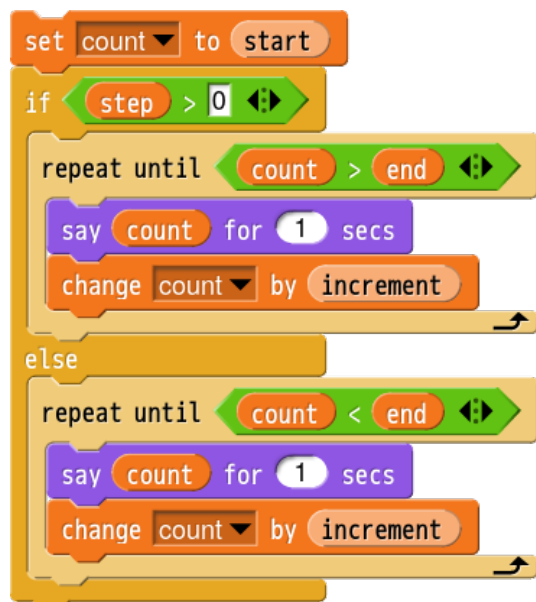
減少カウントにする場合は、終了条件を  にして



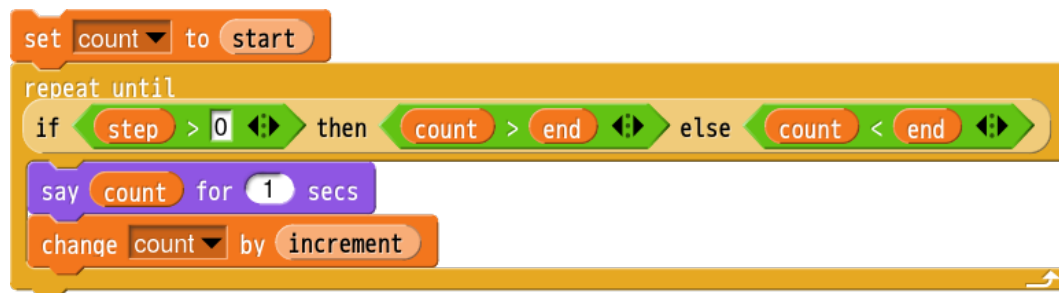
のようになければなりません。

増加でも減少でも対応させると、for ループは次のようになります。

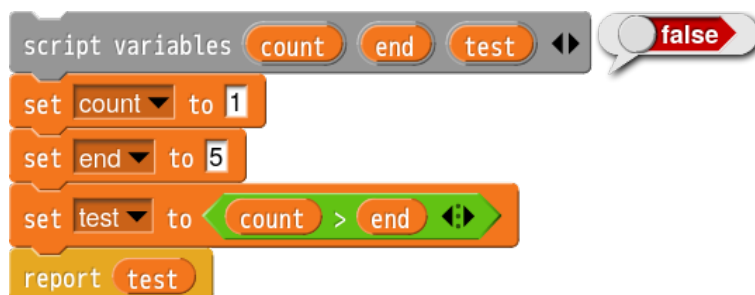
上記スクリプトから  以降を表示します。



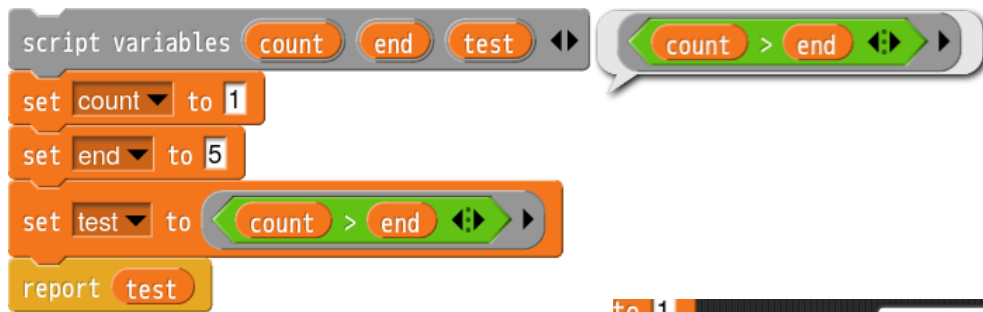
ループのテスト部分をまとめると次のようにすることができます。



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



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



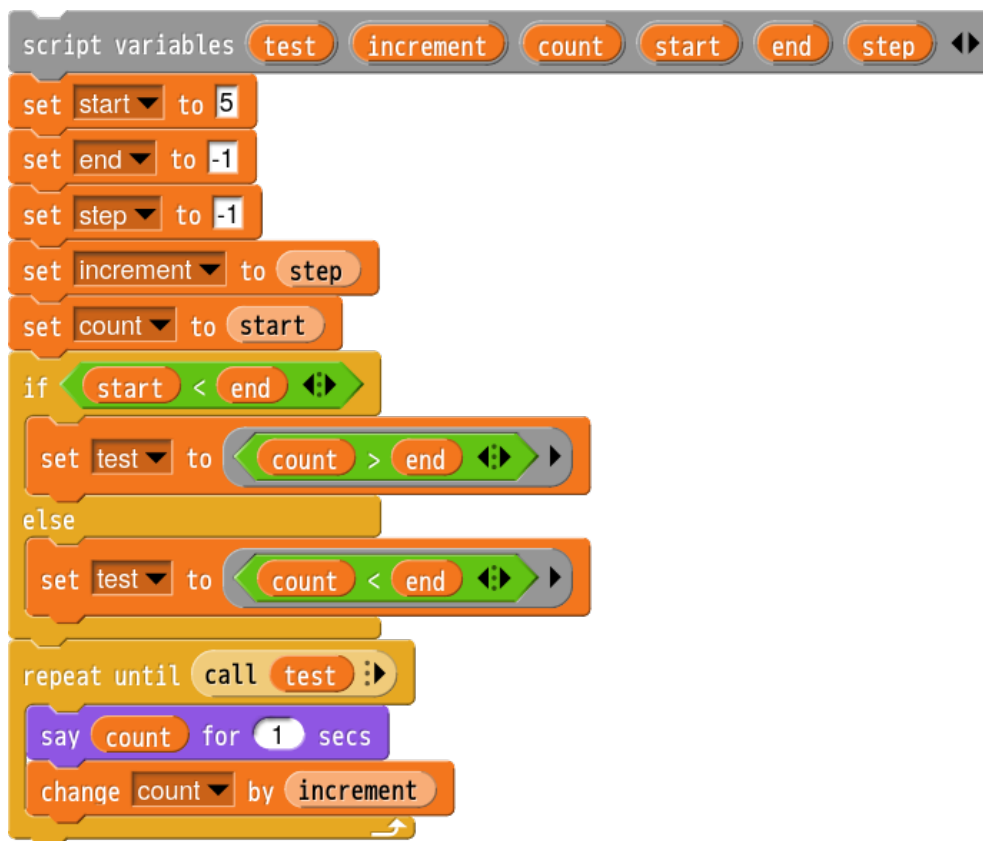
テスト項目自体を変数に入れます。

 をパレットエリアからもってくる必要はありません。右クリックしてメニューから ringify をすればできます。



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

これを for ループに使用します。



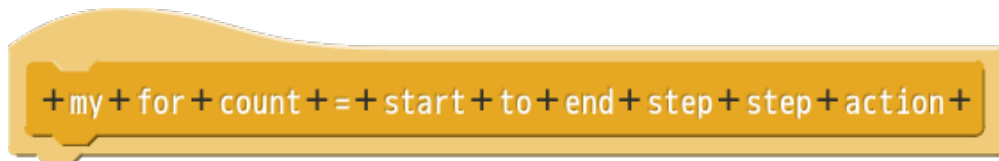
これで一応 for ループの動作が実現できましたがちょっと問題があります。step の値が 0 の場合と、step の値が負であるべきなのに正値を指定すると無限ループになってしまうのです。どちらの場合も続行は好ましくないのでエラー終了させます。

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 count = start to end step step action と入力して OK で次に進んでください。



そうすると、要素がすべて Title text のプロトタイプになります。



プリミティブの for ブロックのプロトタイプ部分です。



count の要素をクリックすると、Edit input name が出ますから右端の三角マークをクリックして設定ウィンドウを出してください。





ここで設定されているのは、Default Name: に `i` が指定され、Upvar が選択されていることです。Title text ではなく Input name を選択すると初期設定として、Any type と Single input がセットされます。Upvar を指定すると、Any type などの上段の指定と Single input は解除されます。

`start` の設定です。



ここでは、Number と Single が選択され Default Value: に 1 が設定されています。end も数

値なので同じように設定されています。

action の設定です。

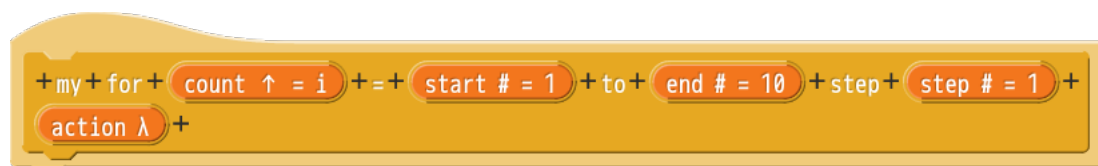


Command (C-shape) と Single input が選択され、ループマークにチェックが付いています。これによって C 型ブロックにループマークが表示されます。

さらに、右下の設定ボタンをクリックすると右図のように static にチェックが付いています。



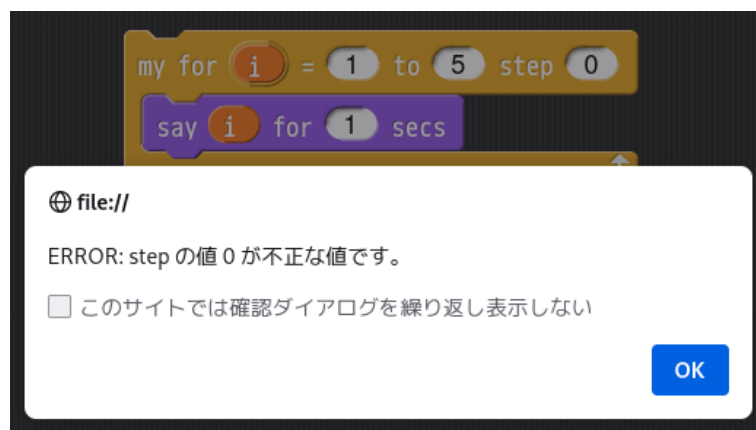
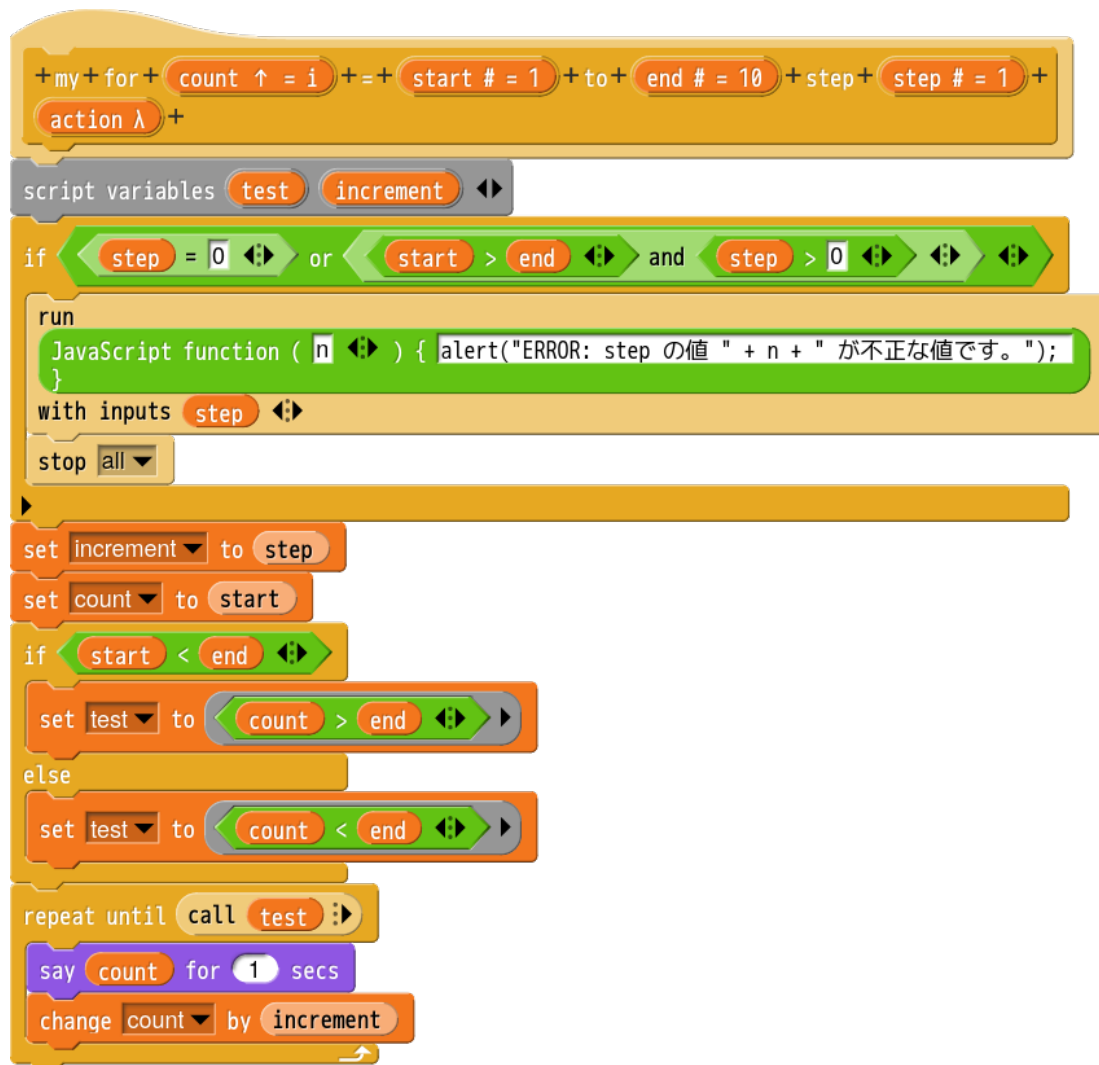
同じように my for のプロトタイプに設定を行います。対応する Text の要素を Input name に変更して設定します。step も数値なので start の設定と同様に規定値を 1 にして設定します。



そうすると、次のようなブロックになります。



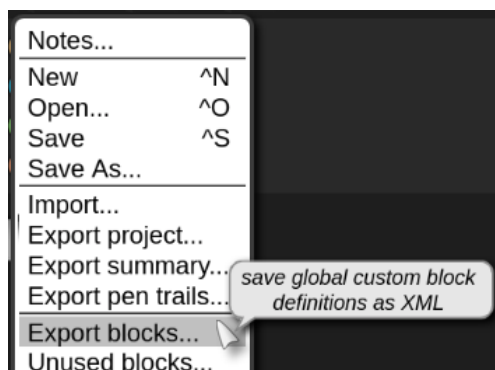
定義の内容は次のようになります。エラーに関する部分は後述する JavaScript の項目を参照してください。



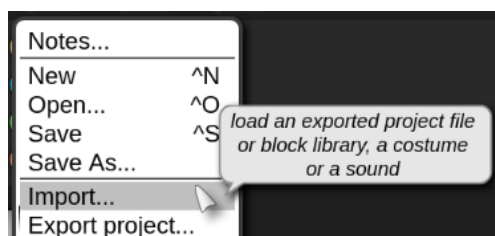
ところで、Libraries... からブロック定義をインポートしてプロジェクトを作成した場合に、普通に保存すると未使用のブロック定義も含んだファイルになります。プロジェクトを公開する場合は、次のようにして未使用のブロック定義を削除してファイルの容量を小さくしたほうがいいかもしれません。



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。



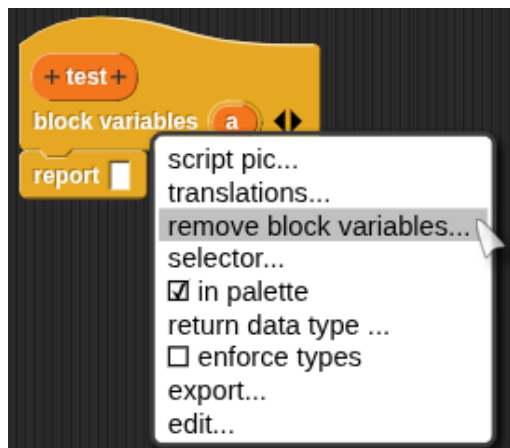
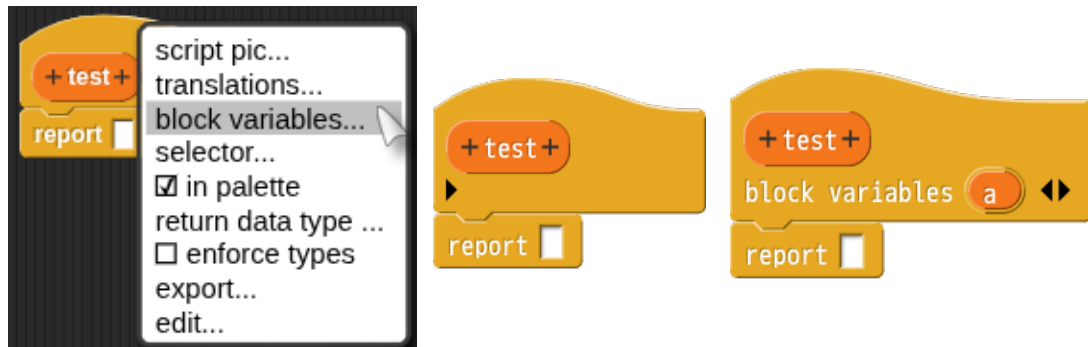
OK をクリックすると、ダウンロードフォルダに「????? blocks.xml」というファイル名で保存されます。????? のところにはプロジェクト名または untitled が入ります。適宜リネームしてください。これを利用する時は、次のようにしてインポートすれば使用できるようになります。



6.5 block variables

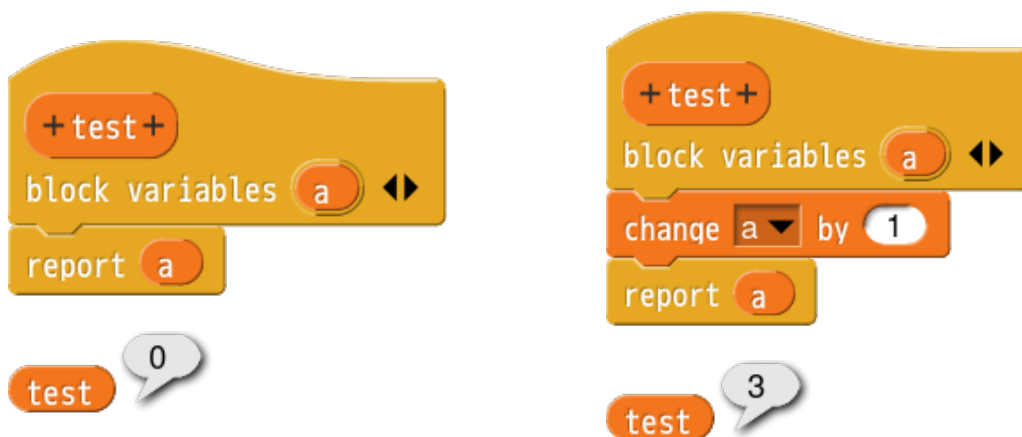
ブロックのプロトタイプのところでも右クリックして、block variables... オプションを選択すると、ブロック変数を使用できるようになります。右向きの三角をクリックすると変数の作成で、左向きの三角をクリックすると削除です。リネームもできます。

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



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

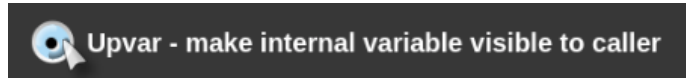
ブロック変数を作成すると次のように 0 に設定されるようですが、右のようにして同じブロックをなんども実行すると a の値は増加されますから set で初期設定してください。



この解説では紙面の都合でブロック変数を使用していますが、基本的にはスクリプト変数の使用をお勧めします。

6.6 Upvar 変数の設定

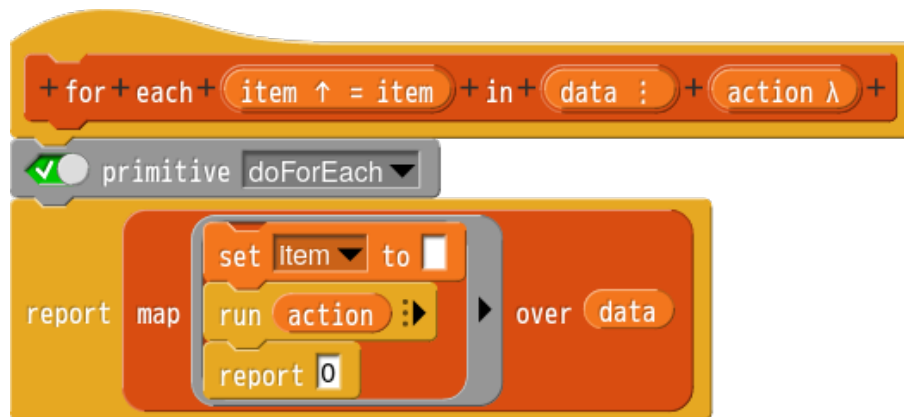
ブロック定義 プロトタイプ内に作成する入力変数は設定で Upvar 指定することで Upvar 変数になります。



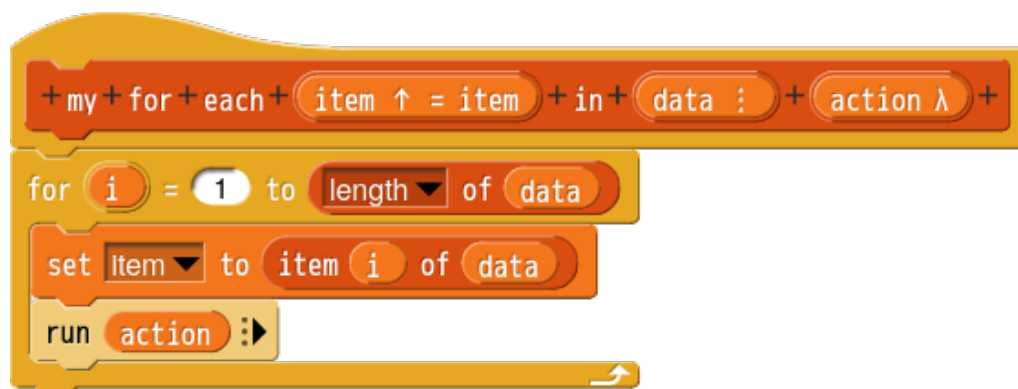
これだけでドラッグ&ドロップして使用できる特別な変数になります。自動的に Number や Any type のオプションはクリアされます。変数名は変更可能ですし、いくらでも取り出して使用できます。変数名のところに Upvar を示す「↑」が表示されます。

Upvar 変数はブロック定義内で変数の値を設定更新でき、その値を持った変数をスクリプトで使用します。for ループでは i の値を指定された範囲で 1 ずつ更新し、for each ループでは item にリストの全要素を順にセットします。

プリミティブの for each ブロックでは次のように map を使用しています。map によってリストである data から各要素を item にセットしながら action を実行します。



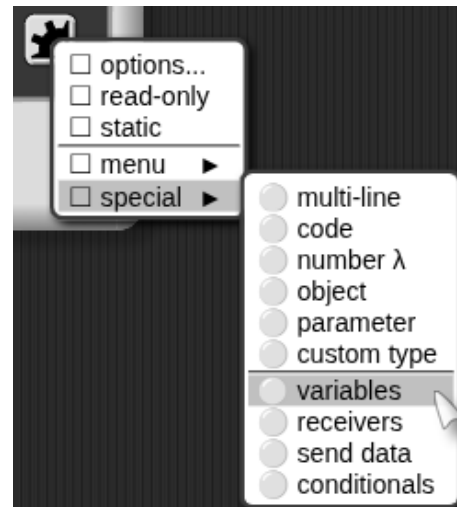
分かりやすいように for ブロックで作成してみます。ループ内で item に data の各要素をセットしながら指定された action を実行します。action は Command(C-shape) タイプです。



このように Upvar 変数は定義ブロック内で更新のような操作が可能な変数で、その変数使用時はその操作された値を利用するものです。

スクリプト変数のようにブロックから変数を取り出して利用できる仕組みがあります。こちらで作成すると Upvar 変数のようにブロック定義で値を操作することはできません。変数名も a からとスクリプト変数同様になります。

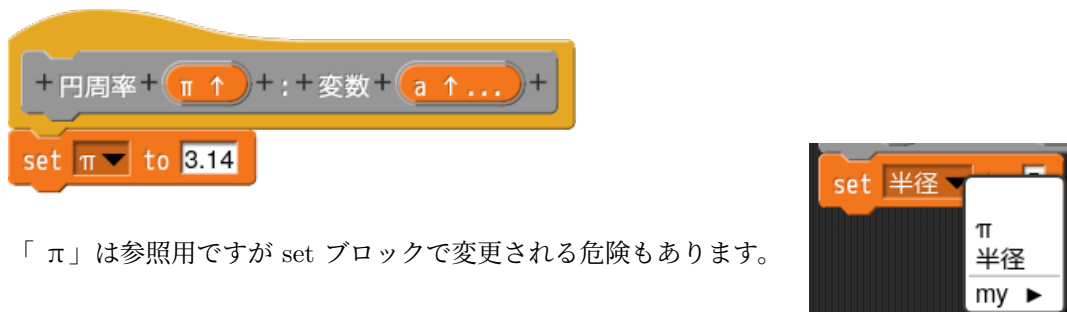
変数の設定で右下の歯車のようなアイコンをクリックすると右図のメニューが出るので、special から variables を指定します。変数名は変更もできますし、スクリプト変数と同じように増やせます。



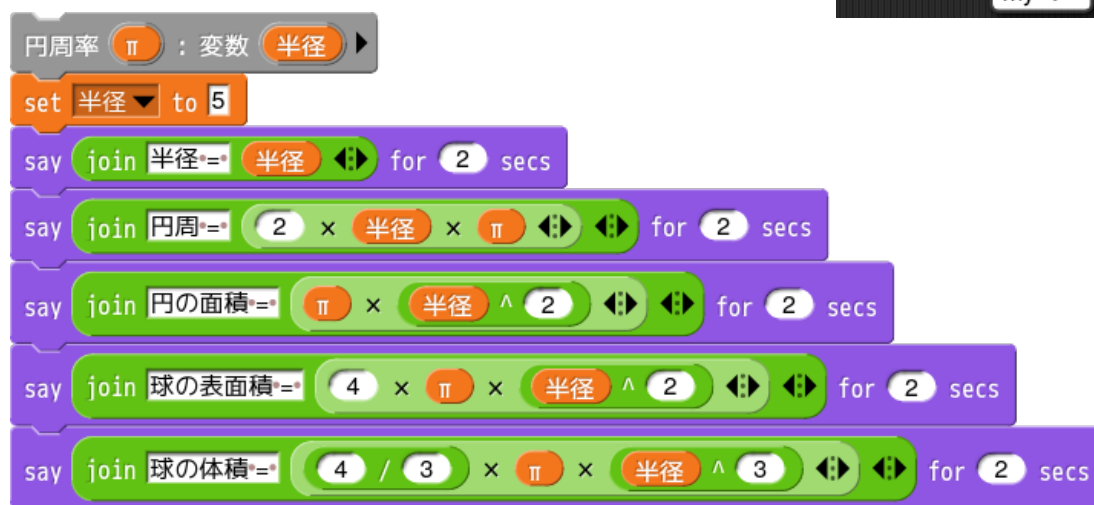
これを使用すると、script variables ブロックと同じものが作れてしまいます。



Upvar 変数を定数定義のように使ってみます。上記の variables 設定した変数も使用しています。



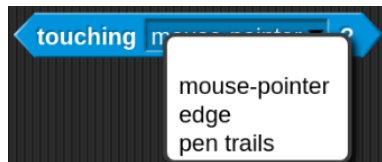
「π」は参照用ですが set ブロックで変更される危険もあります。



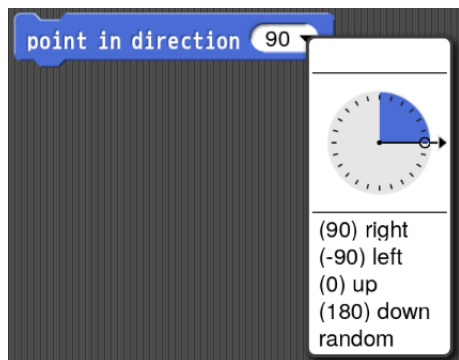
7 ブロック定義について

7.1 プルダウン入力

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



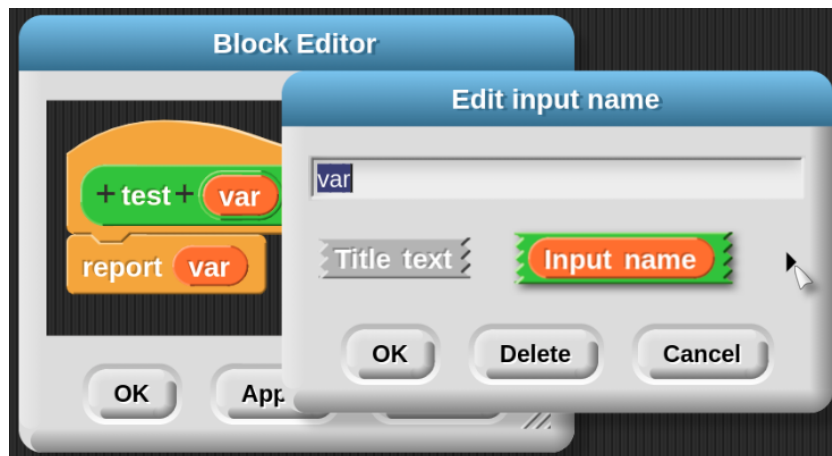
入力スロットに値をキーボードから入力することはできなくなっています。これは、後で出てくる read-only のオプションが指定されているためです。



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

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

説明のために、test というブロックを作成します。



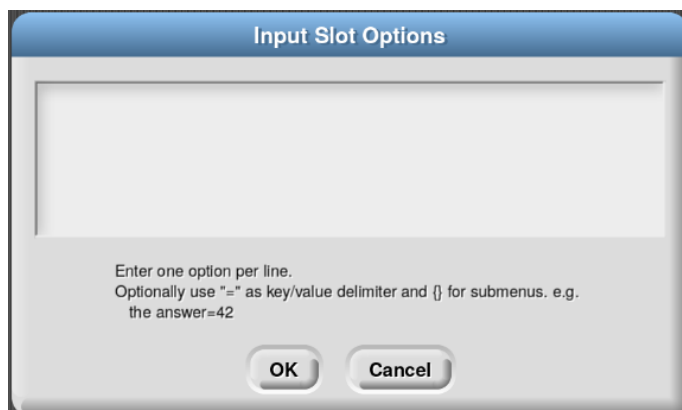
プルダウン入力を行うには、Input name の設定ダイアログを開きます。



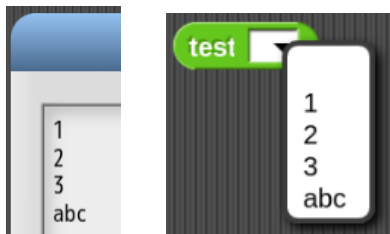
の var をクリックすると Edit input name のダイアログが開きますから、Input name の右側にある三角をクリックします。これで、大きな Edit input name のダイアログが開きます。ここの暗い灰色の領域で右クリックするか、 ボタンをクリックします。すると、このようなメニューが表示されます。



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

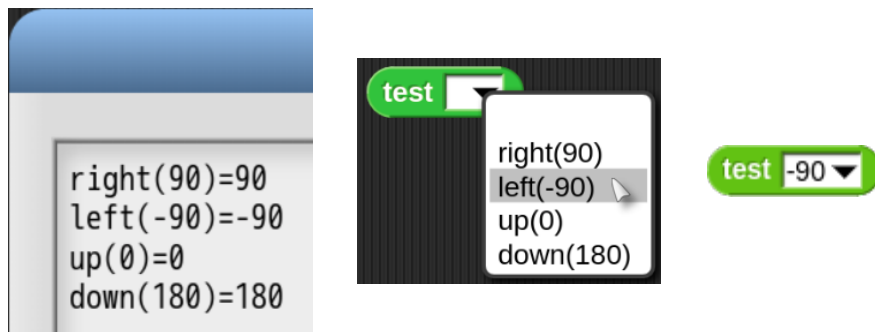


ここの各行にオプションを入れてプルダウンメニューを作っていきます。
左のように設定して、ブロックエディターを Apply すると、右のような結果になります。

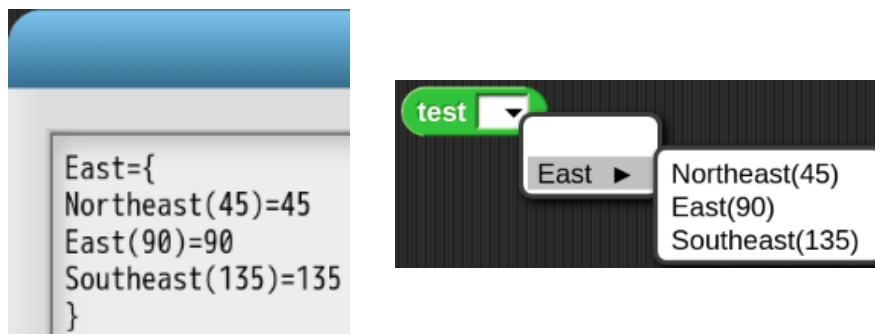


設定したものがそのまま表示され、クリックするとそれが入力値になります。

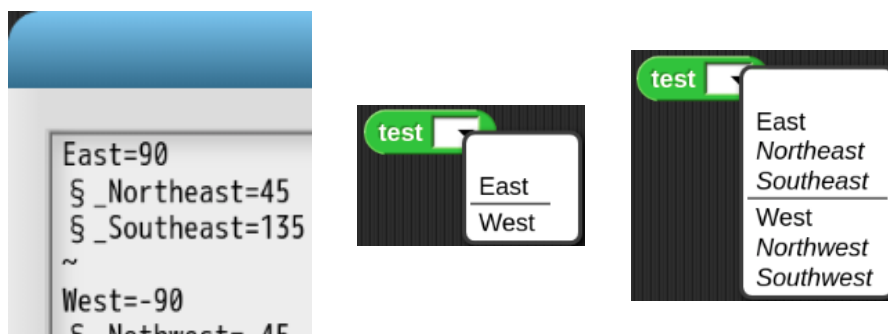
次のように、「=」で値を設定すると、メニューには「=」の左側の項目が表示されますが、クリックされると「=」の右側にある項目が入力値になります。



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、**[Ctrol]+[Shift]+[u]**を押してから、a7を入力して[⌨]で出すことができます。a7は0167の十六進数コードです。OSを問わず、日本語入力モードで「せくしょん」または「きごう」と入力して変換して出すこともできます。

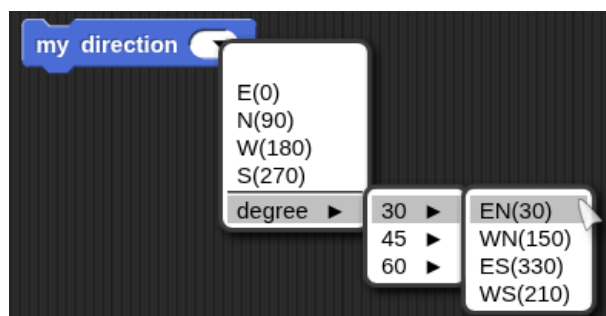
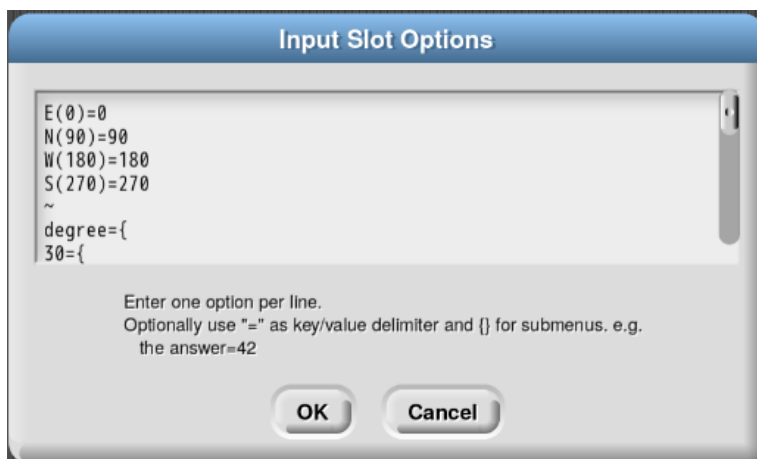
Single input. Default Value: に値を設定すると、選択肢のデフォルトになります。

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

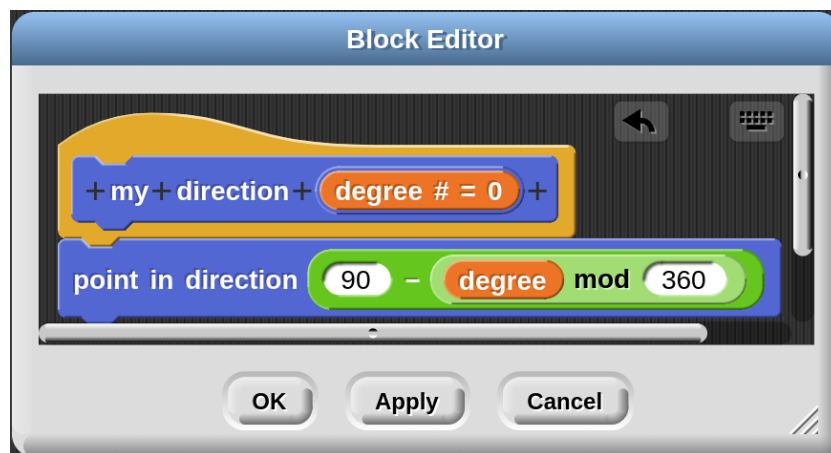
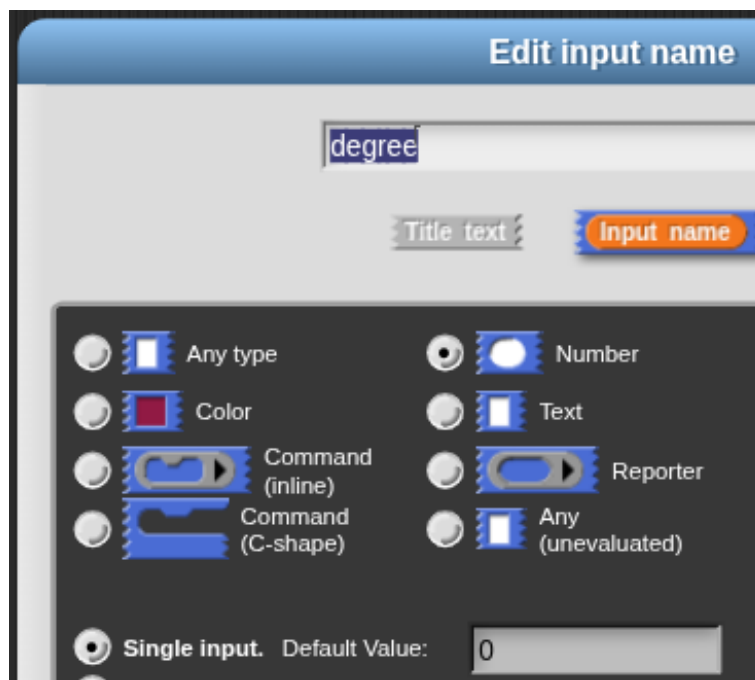
```

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

```

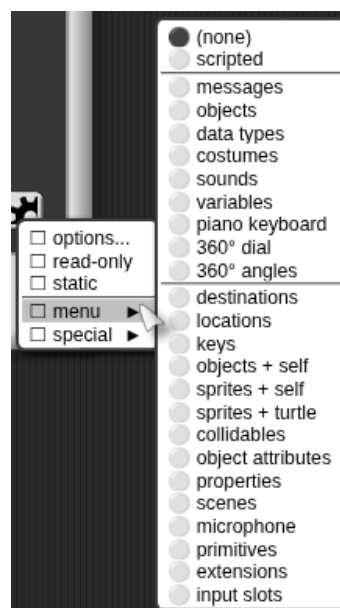


degree から、EN, WN, ES, WS のサブサブメニューを作っています。あくまで機能説明の作例ですので、あまり意味のあるものではありません。



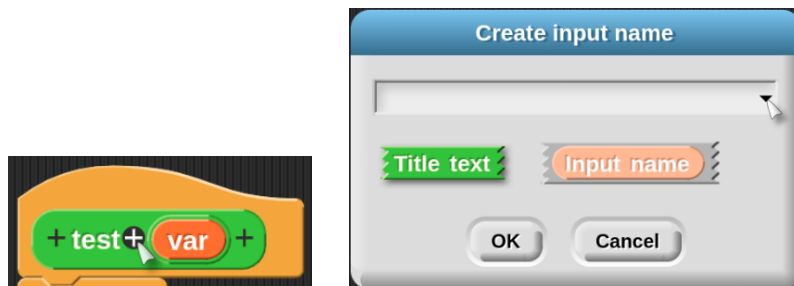
また、menu サブメニューから選択することで、入力スロットへの指定の仕方をいろいろ設定することが可能になります。

degree の場合、360° angles にすることで右水平方向が 0 の値を設定することが可能です。



7.2 Title Text とシンボル

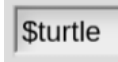
プリミティブブロックの中には、表示に  の回転する矢印のようにシンボルが含まれているものがあります。カスタムブロックでもシンボルを使用できます。ブロックエディターで、プロトタイプのシンボルを挿入したい位置のプラス記号をクリックします。すると、ダイアログが開きますから Title text にしてください。



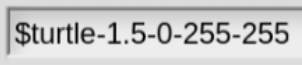
次に、入力待ちのテキストボックスの右端にある  をクリックします。するとシンボルのメニューが表示されます。まとめて表示すると次のようになります。(Title text としてセットされたものを右クリックしてもシンボルメニューが表示されます。)

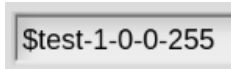
■ square	✓ tick	➡ arrowRight
▶ pointRight	☑ checkedBox	➡ arrowRightOutline
▶ stepForward	□ rectangle	➡ arrowRightThin
⚙ gears	■ rectangleSolid	🤖 robot
⚙ gearPartial	○ circle	🔍 magnifyingGlass
⚙ gearBig	● circleSolid	🔍 magnifierOutline
📁 file	• dot	🖱 selection
🖥 fullScreen	○ ellipse	○ polygon
📏 grow	— line	🖌 closedBrush
🖥 normalScreen	✚ cross	🎵 notes
📏 shrink	✚ crosshairs	📷 camera
📺 smallStage	🎨 paintbucket	📍 location
📺 normalStage	🧼 eraser	👣 footprints
🐢 turtle	🧴 pipette	⌨ keyboard
➡ turtlePlus	💬 speechBubble	⌨ keyboardFilled
➡ turtleOutline	💬 speechBubbleOutline	🌐 globe
📺 stage	🔄 loop	🌐 globeBig
⏸ pause	↶ turnBack	📋 list
🚩 flag	➡ turnForward	📋 listNarrow
⬆ octagon	⬆ arrowUp	⋮ verticalEllipsis
☁ cloud	⬆ arrowUpOutline	🔄 flipVertical
☁ cloudGradient	⬆ arrowUpThin	🔄 flipHorizontal
☁ cloudOutline	⬆ arrowUpDownThin	🗑 trash
🔄 turnRight	⬅ arrowLeft	🗑 trashFull
🔄 turnLeft	⬅ arrowLeftOutline	📦 cube
🔄 turnAround	⬅ arrowLeftThin	📦 cubeSolid
📦 storage	↔ arrowLeftRightThin	🧱 blocks
🖨 poster	⬇ arrowDown	🔊 speaker
⚡ flash	⬇ arrowDownOutline	♾ infinity
🖌 brush	⬇ arrowDownThin	📶 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。

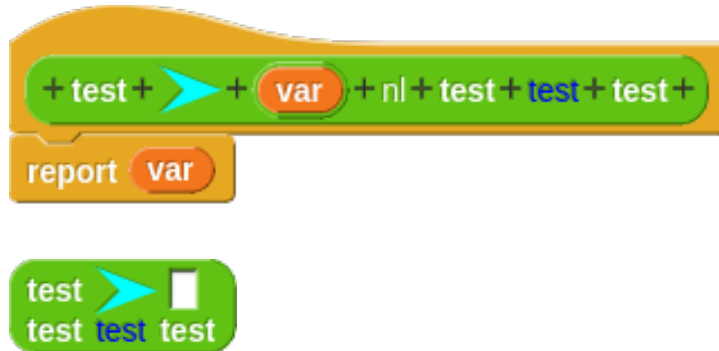
すると、入力欄に  がセットされます。

OK して Apply すると、 になります。

定義を  に変更すると、 になります。シンボルの後の「-1.5-0-255-255」は、表示倍率 (1.5) 指定、RGB(Red=0, Green=255, Blue=255) によるカラーコード指定です。表示倍率だけの指定もできます。

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

定義は、こうなります。



7.3 Input name オプションについて

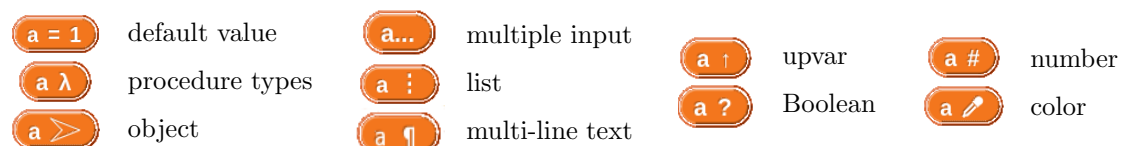
ブロックを作成する時の Input name のオプションについて見ていきます。間違っているかもしれませんが、こういう考え方をするとオプションを選ぶ目安になるのではないかと思います。

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に入れられるものです。Reporter 型は、なにかしらの値をレポートする、楕円形のリポーターブロックに入れられるものです。Predicate 型は、真理値 true か false をレポートする、六角形のブロックに入れられるものです。

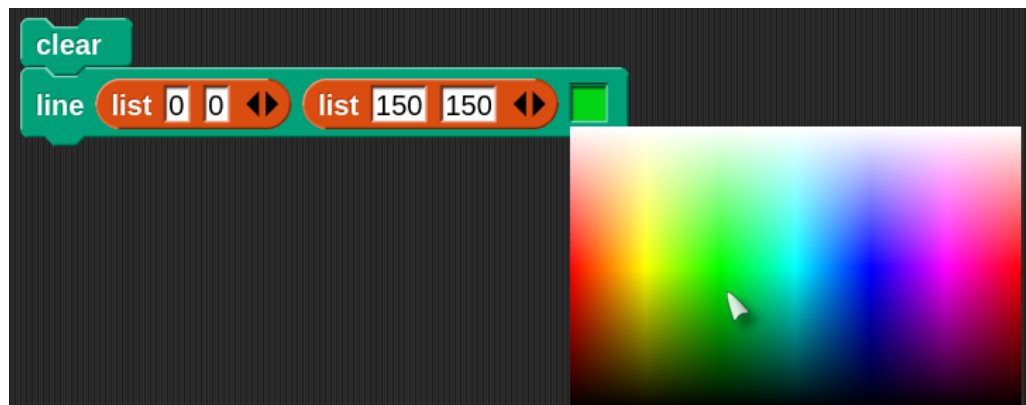
また、それぞれについて下の段の 3 種類のオプション（Single input, Multiple input, Upvar）を設定できる場合があります。設定された内容によってプロトタイプ内では次のように表示されます。



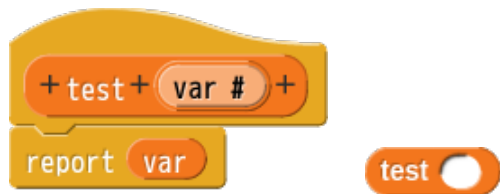
7.3.1 Reporter 型

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

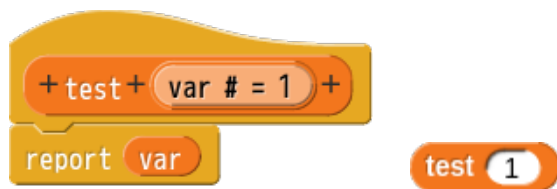
同じように色指定をすることができます。



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

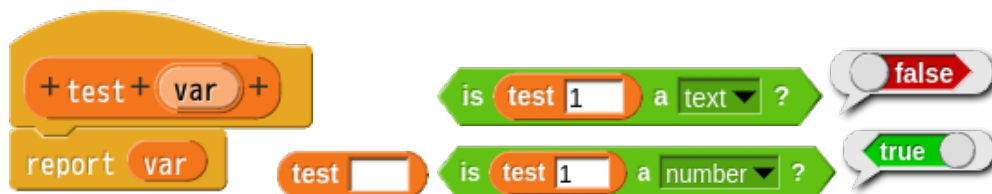


Default Value を指定すると、



既定値を設定することができます。

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



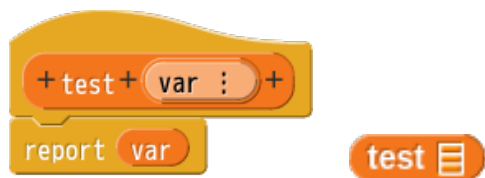
- special のオプションから multi-line または code を選択して次のような定義にすると、入力スロットにキーボードから改行を含むテキストを入力することができます。



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



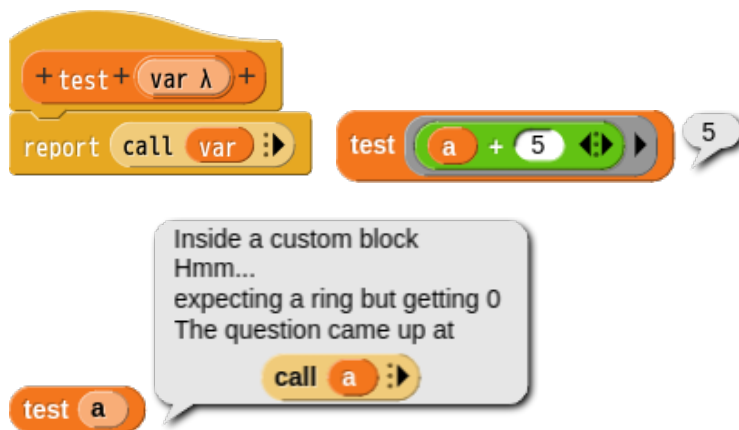
- リストの入力を求めていることをアピールするのが、List です。



- 入力スロットにリングで囲ったものを扱う必要がある場合があります。使用することにより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。

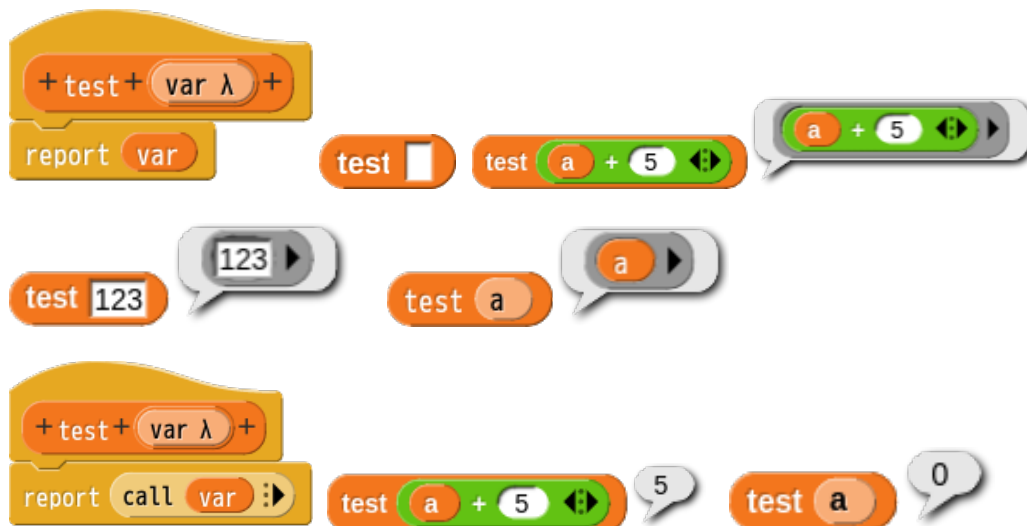


ただし、変数単体だと次の Any(unevaluated) とは違い **test a** とリング付きではなくなくなってしまいます。実際には call ブロックを使ってリングブロックの値を求めます。変数単体だと次のようになるので、場合によっては手動でリングを付ける必要があります。

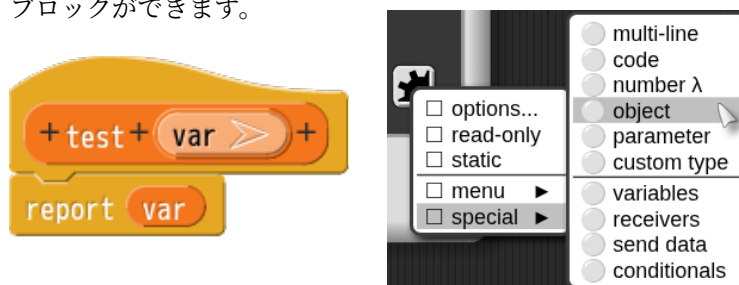


- Reporter から外観上リングをなくし、キーボードから入力できるようにしたのが

Any(unevaluated) です。unevaluated 評価されていない、つまり、値を求めるような操作はされていないということです。評価について... 数字の「1」を評価して1という数値を得るというような表現もするので、実行するというのとはちょっとニュアンスが違っています。



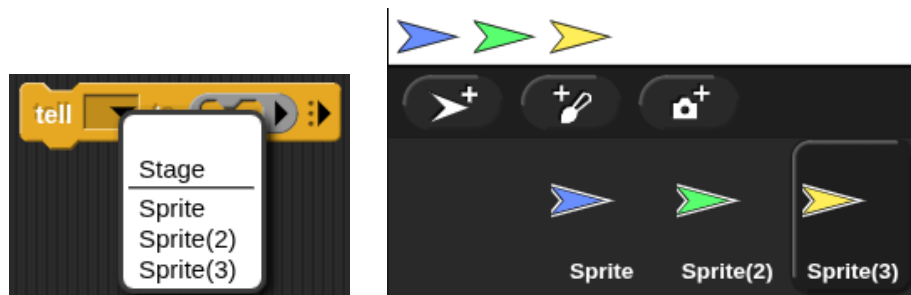
- special のオプションから object を選択して次のような定義にすると、**test** というブロックができます。



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

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

に指定できるオブジェクトが表示されます。



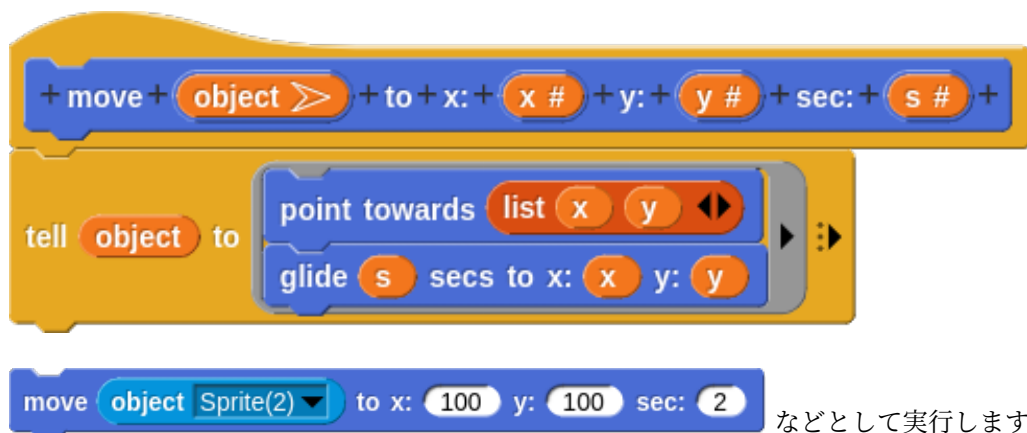
この入力スロットにオブジェクトを示す変数などをドロップすることもできます。Sensing パレットにある Object リポーターブロックの入力メニューを開くと tell ブロックと同様に指定可能なオブジェクトを選択することができます。



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



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



などとして実行します。

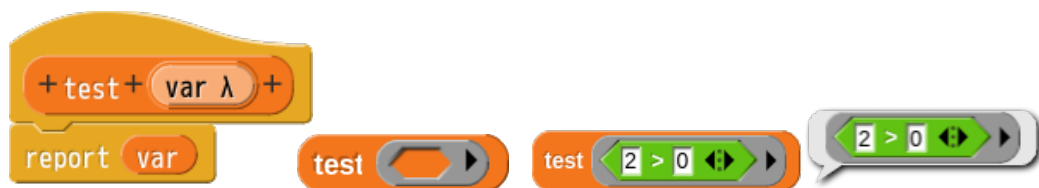
7.3.2 Predicate 型

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

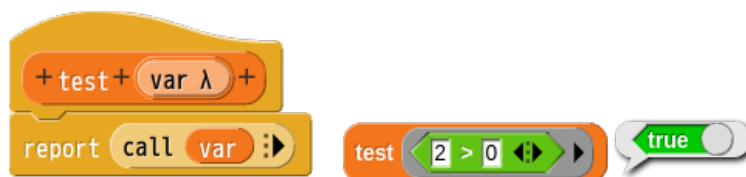
- Boolean です。



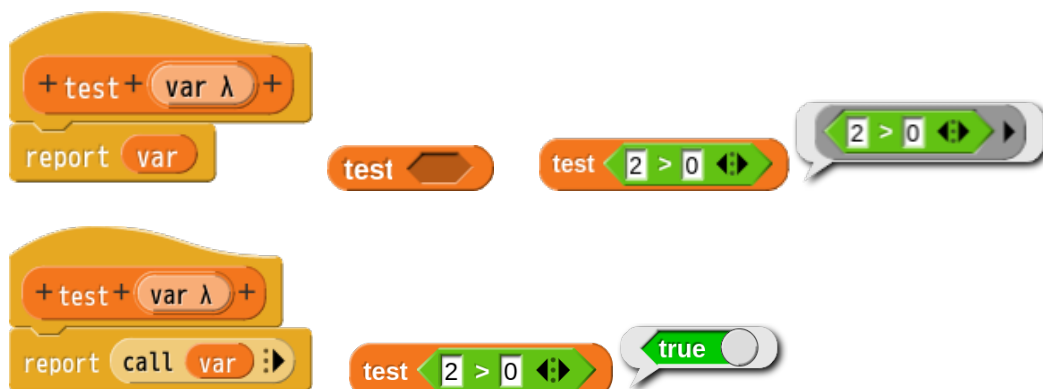
- Predicate です。 (変数単体だと Reporter のようにリングが消えてしまうので注意が必要です。)



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



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



7.3.3 Command 型

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

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

c 言語では、

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

のようになると、0、1、2、3、4 と表示するプログラムになります。

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

i = 0 の部分が初期設定で、初めに一回だけ実行されます。

i < 5 の部分のテストが true なら繰り返しを続行します。

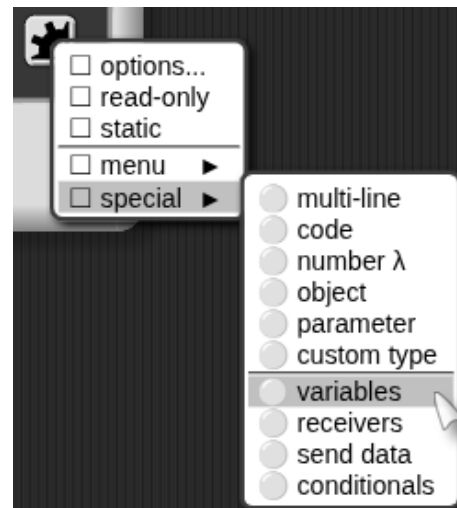
i++ の部分は次の繰り返しに進む前に実行されます。i++ は i に 1 加算する処理をします。

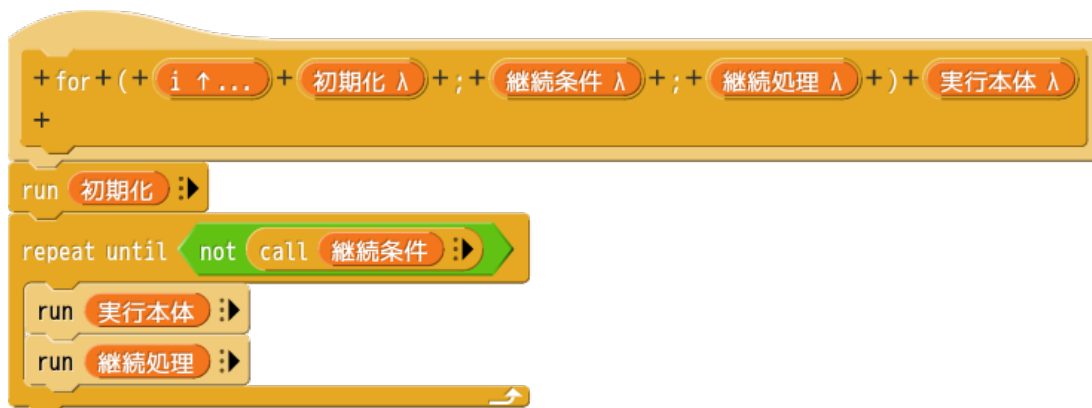
{ } の内部がループの本体です。

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

変数 i を作成します。設定で右下の歯車のようなアイコンをクリックすると右図のメニューが出るので、special から variables を指定します。変数名を i に指定してもスクリプト変数と同じように a に設定されるようです。

(66 ページ参照)

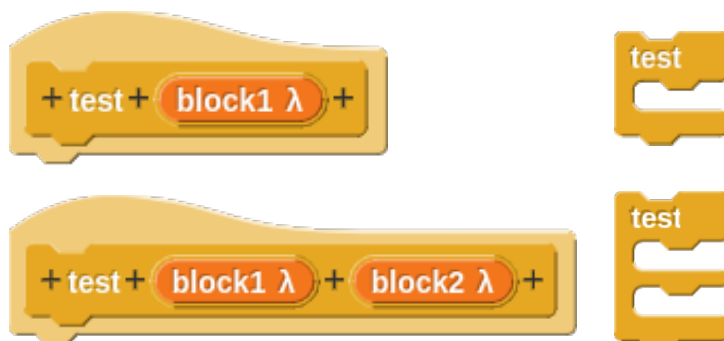




なお、「初期化」と「継続処理」は Command(inline) 型で、「継続条件」は Predicate 型です。「実行本体」は Command (C-shape) 型です。ループ表示ボタンをチェックしてください。(61 ページ参照)



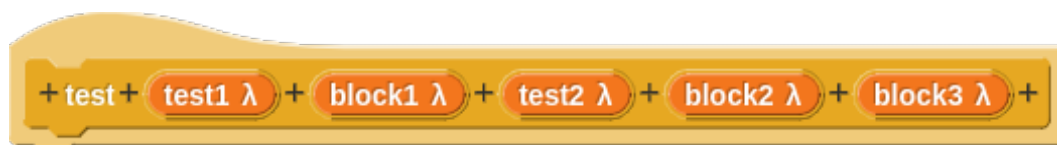
for ループは C 型が 1 つでしたが、2 つにすると E 型になります。if else の形ですね。Command (C-shape) 型の変数を 並べればいくらでも増やせます。



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

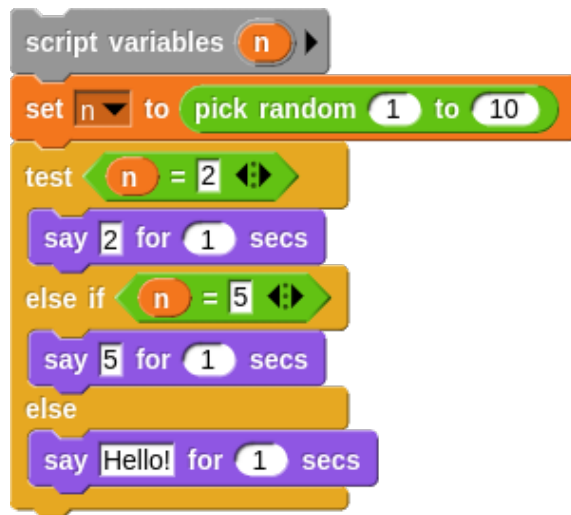


変数 block2 の前にも Boolean(unevaluated) の変数を入れ、block3 を追加すると変わった if else ブロックができます。





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



C 言語などで利用できる switch case ブロックを作ってみます。

switch の入力スロットで調べる変数を指定して、case ブロックで指定した値と一致した場合にその処理をします。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。

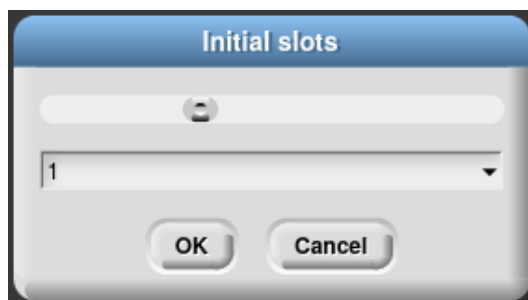


case ブロックは単に指定された値と実行ブロックの対をリストとしてレポートするだけです。

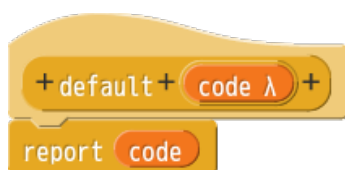


case ブロックは Reporter 型です。code は Command(C-shape) 型です。

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



default ブロックは単に実行ブロックをレポートするだけです。

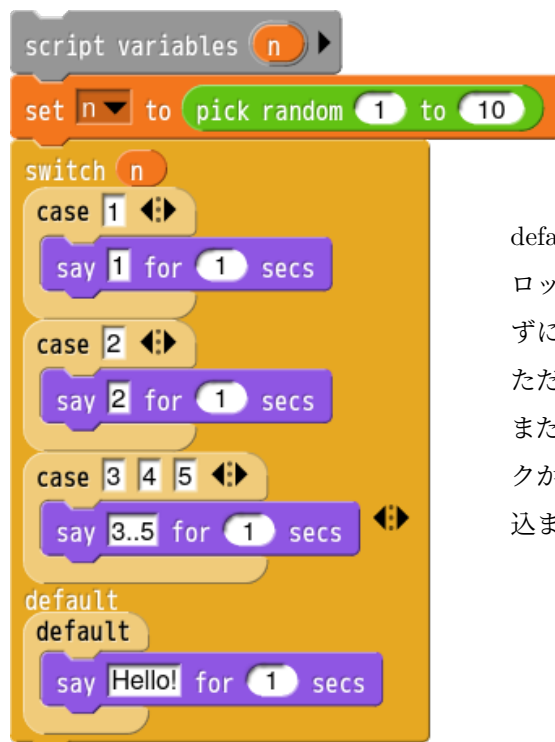


default ブロックは Reporter 型
code は Command(C-shape) 型
です。

switch の定義ブロックです。cases 変数は Command(inline) かつ Multiple inputs(value is list of inputs) を使って追加できるようにしています。case の変数「値」のように初期入力スロット数を 1 にしてください。これには値と実行ブロックの対のリストが入っているので、switch で指定した値と case の値がマッチしたら指定されたブロックを実行します。マッチするものがなかったら default で指定されたブロックを実行します。default は Command(inline) 型です。default は無指定、空のままでも大丈夫です。(「nl」は 73 ページ参照)



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」を参考にさせていただきました。



default ブロックは一般的な switch の仕様や case ブロックの見た目に合わせるためのもので、これを使わずに実行ブロックを直接入力することもできます。ただし、default ブロックの位置は固定です。また、case で指定された処理を終えたら switch ブロックから脱出します。c 言語の case 文に break が組み込まれているような Go 言語仕様のものです。

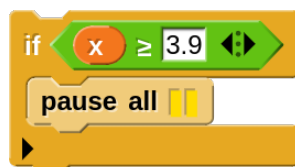
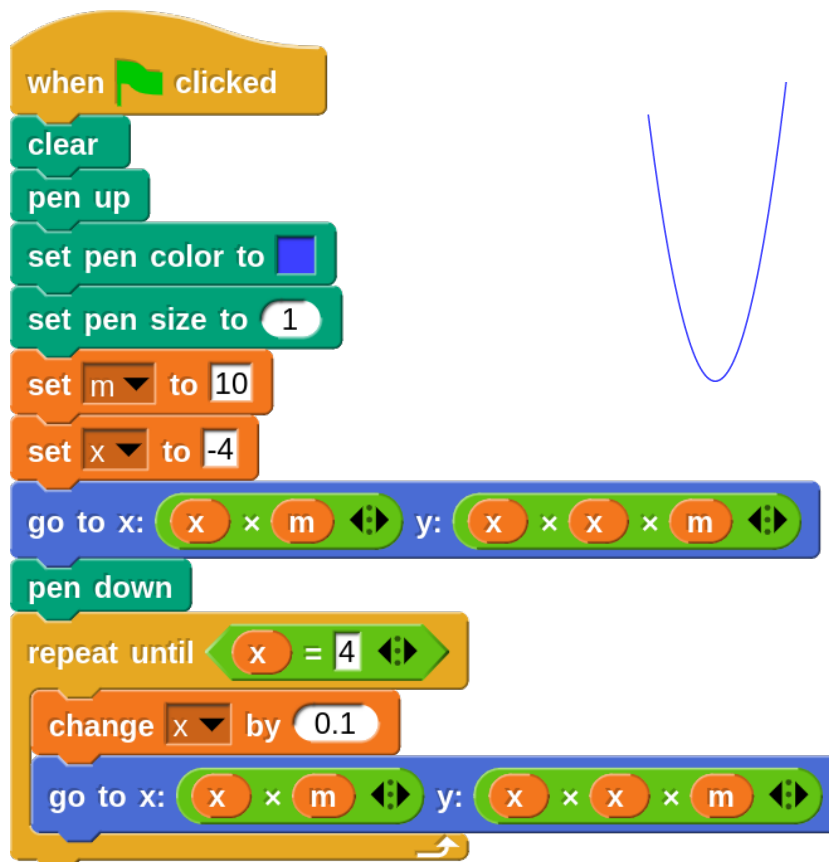
8 その他

8.1 デバッグ

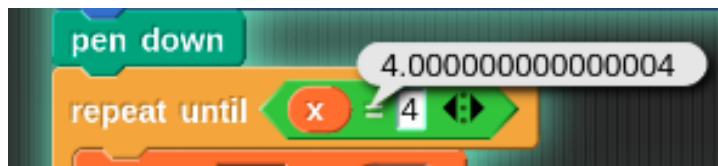
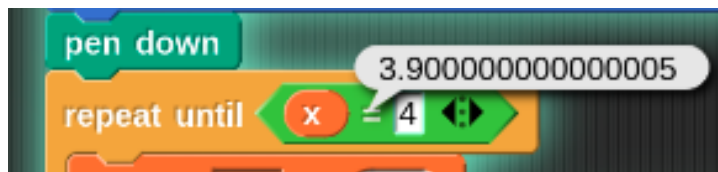
Snap! にはデバッグの機能が装備されています。

 **pause all** ブロックにより指定の箇所でプログラムの実行を一時停止させることができます。 の操作でプログラムの実行スピードを遅くしたり、1 ステップ実行にしたりできます。デバッグ中は実行中のブロックをハイライトしてくれます。変数などの値もリポートしてくれます。 のクリックで一時停止した実行を再開できます。

次のスクリプトは x の値が -4 から 4 になるまで 0.1 ずつ増やしながらグラフを描くものです。 $x = 4$ になったら終わりにしたいのですが、終了しません。



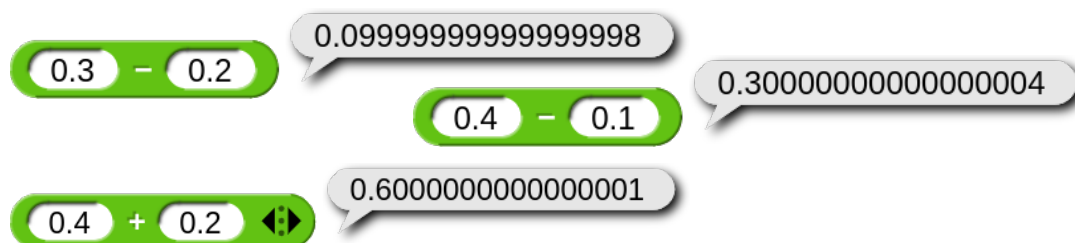
を repeat until 内の最後に追加してから、スクリプトを直接クリックして実行させて下さい。そうすると、 $x \geq 3.9$ の時点で一時停止します。 ボタンをクリックして のように、スライダーを左端にして1ステップ実行モードにします。 をクリックして1ブロックずつ実行しながら x の値を確認して下さい。



このように、 x は 4 にはならないので無限ループになります。
 repeat until の終了条件を $x \geq 4$ にする必要があります。
 次のように、Julia 言語で 0.1 を表示させてみます。「0.1」というのは十進数で表現された文字列です。コンピューター内部ではこれを二進数に変換して数値として扱います。println はそれを十進数値文字列として丸めて 0.1 と表示します。printf を使うと、指定桁表示できます。
 0.000000000001 のような小さな数から 10000000000 のような大きな数まで扱える仕組みが浮動小数点数ですが、十進数で表された小数点以下の値は二進数へ正確には変換できません。Snap! も同じ仕組みになっています。

```
julia> println(0.1)
0.1

julia> @printf("%.30f\n", 0.1)
0.100000000000000000005551115123126
```



!! ボタンをクリックすると、デバッグモード  のオンオフができます。
 ただし、 のボタンで実行すると、変数などの値をレポートしてくれません。直接スクリプトをクリックして実行してください。
 ユーザー定義ブロック内についてもデバッグする場合は、デバッグモードにしてからユーザー定義ブロックを edit で表示させれば見ることができます。
 通常は edit で開くと一番下に (Ok) (Apply) (Cancel) が表示されますが、デバッグモードでは (OK) だけ表示されます。
 ユーザー定義ブロックのデバッグは必要ないならば、表示させなければユーザー定義ブロック内は通常速度で実行されます。

8.2 整数への変換

小数点が付いた数値である浮動小数点数を整数に変換するには、**ceiling** of 10 (切り上げ)、**floor** of 10 (切り捨て)、**round** (四捨五入) があります。

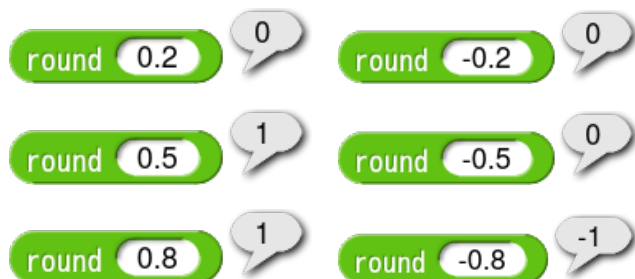
ceiling は引数以上の最小の整数を返します。



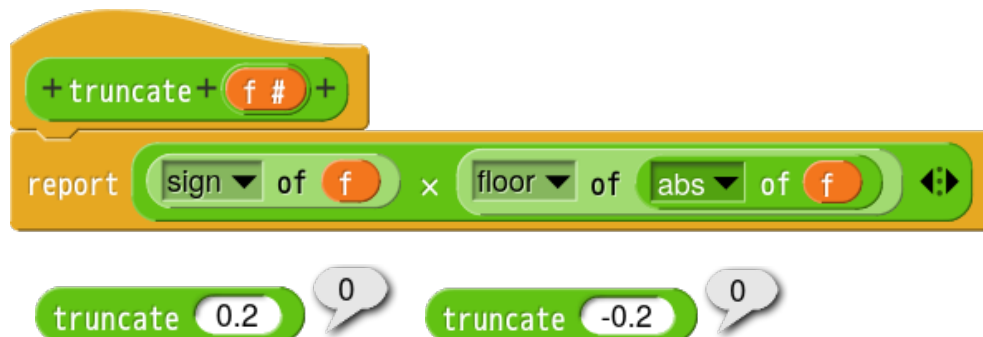
floor は引数以下の最大の整数を返します。



round は最も近い方の整数を返します。ただし小数部分が 5 ならば引数以上の最小の整数を返します。



他言語では **truncate** という単に整数部分を返す関数が定義されていたりします。



8.3 分数の加算

分数の計算では、 $\frac{1}{2} + \frac{1}{3} = \frac{3}{6} + \frac{2}{6} = \frac{5}{6}$ になります。

Snap! では数値を浮動小数点数で扱っているので、分数のような形でも浮動小数点数としての計算になります。

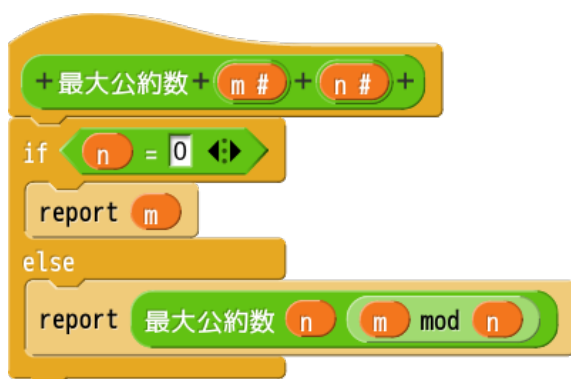


ですが、分数を文字列として処理することは可能です。



分数を通分するには最小公倍数を使用し、約分するには最大公約数を使用します。

最大公約数を求めるにはユークリッドの互助法を用います。28 ページでも提示していますが、ここでは「Snap! のこと II」で扱っている再帰呼び出しを使ったバージョンを示します。



ここで定義した分数は文字列であり、算術演算で使用するとエラーになります。



分数文字列を数値に変換するブロックです。分数を表すスラッシュが含まれないならばそのまま返しています。



浮動小数点数では分数を正確に表せないので、必要になるまで数値化しないことや分数として扱うことが有効な場合もあります。

$\frac{1}{2} + \frac{1}{3}$ 0.8333333333333333

$\frac{5}{6}$ 0.8333333333333334

分数数値化 分数 $\frac{1}{2} + \frac{1}{3}$ 0.8333333333333334

分数 分数 $\frac{1}{2} + \frac{1}{3} + 1$ $\frac{11}{6}$

combine list $\frac{1}{2} \frac{1}{3} \frac{1}{4} \frac{1}{5} \frac{1}{6}$ using 分数 $\frac{\quad}{\quad} + \frac{\quad}{\quad}$ 29/20

刻み幅を 1/10 にしてループする場合、浮動小数点数では 1 にならないためループが終了しません。 $x \geq 1$ でチェックすれば大丈夫です。

```
script variables x
set x to 0
repeat until x = 1
  say x for 1 secs
  change x by 1 / 10
```

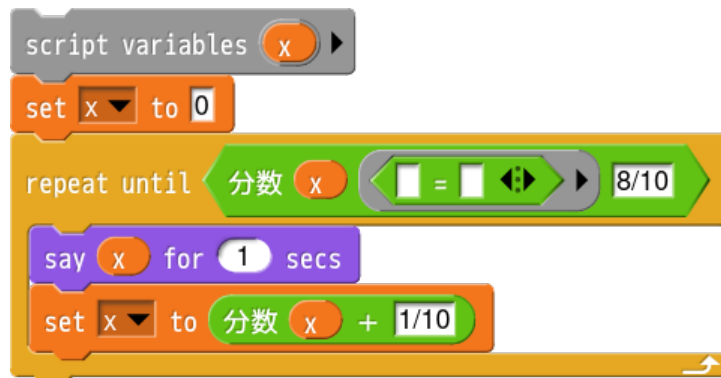
▶ 0.1 でも同様です

こちらは x の値 8/10 が 4/5 に約分されるため、文字列として比較すると 「4/5」 ≠ 「8/10」 なのでチェックをすり抜けます。

```
script variables x
set x to 0
repeat until x = 8/10
  say x for 1 secs
  set x to 分数 x + 1/10
```

▼
文字列なので chage ブロックは使えません //

分数は文字列で表しているので専用の大小比較ブロックを定義する必要があります。通分をし、分子部分で比較します。比較用ブロックを入力スロットにドロップして指定します。
変数 pred は Predicate 型です。



8.4 論理演算ブロック

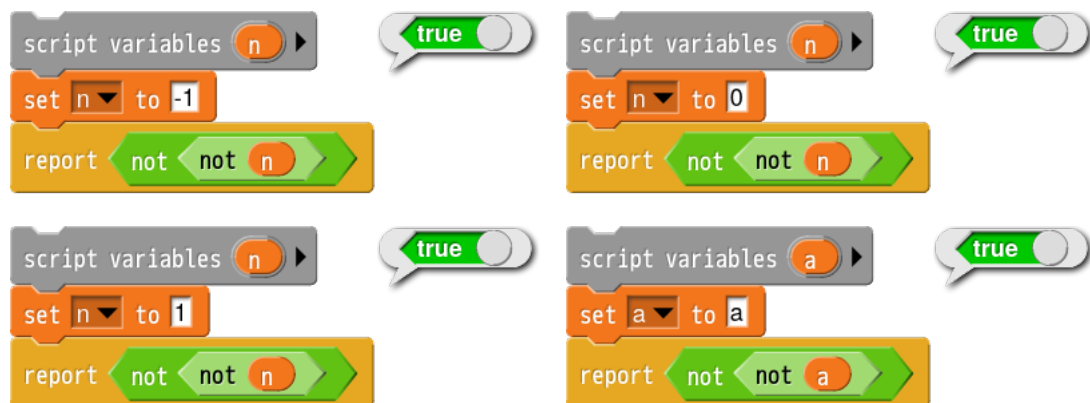
真理値（真偽値または論理値とも表現されます。）は   のような六角形のブロックが返す true か false の値で、if ブロックなどの条件分岐に使用されます。

   が重要な論理演算ブロックです。

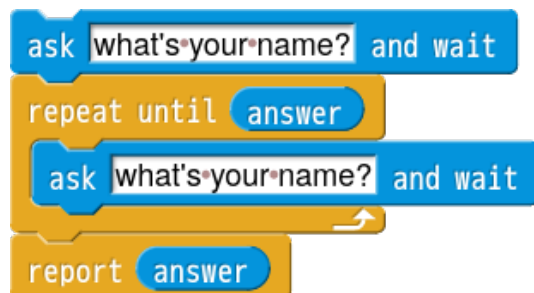
 は真理値を反転するものです。



論理ブロックへの入力には true か false を返すブロックが大前提なのですが、数値や文字などのリポーターブロックも基本的に true 値として受け付けてくれます。



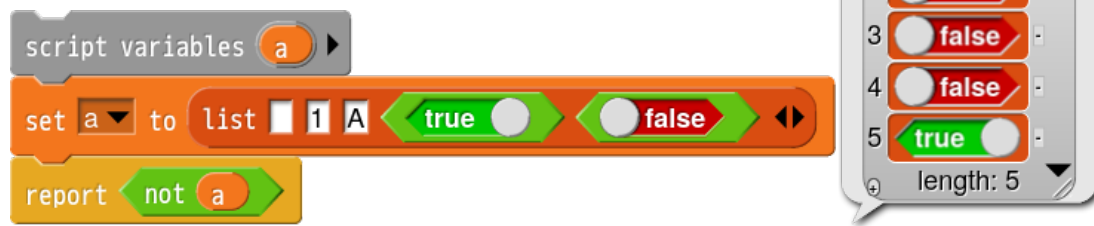
false になるのは空文字や値がセットされていない変数の場合です。次の例では入力でエンターキーだけを押し、false が返されるのでなにか入力されるまで繰り返します。



次の例は変数に値がセットされていないので false が返されます。

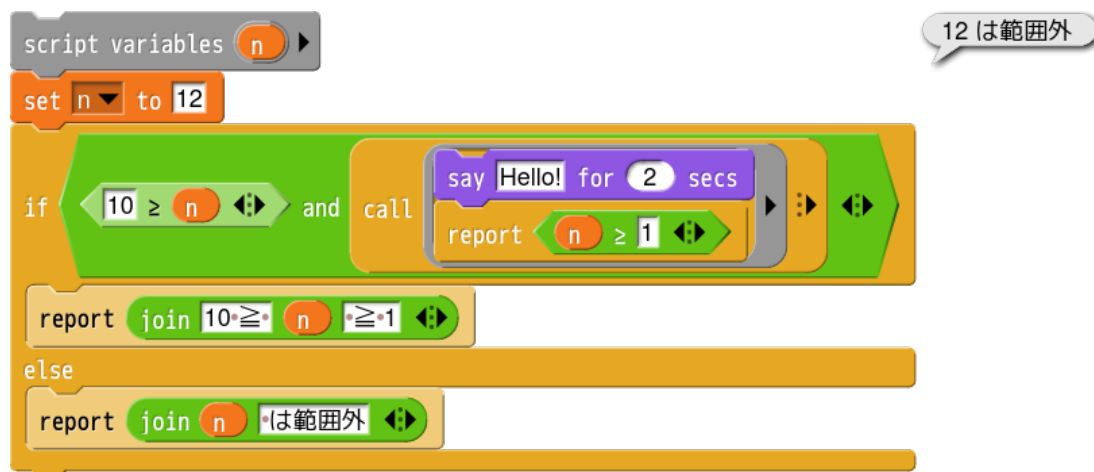


リストの場合は、各要素に対して not 演算がされるようです。



and と or 演算を使ってそれぞれ $10 \geq n \geq 1$ のテストをするスクリプトを示します。

and は両方の入力スロットが true の場合だけ true を返します。入力スロットの評価は左側から順に行われます。もしも値が false ならば、and 演算の場合 false が確定なので次の入力スロットは評価しません。次のスクリプトでは Hello! は表示されません。

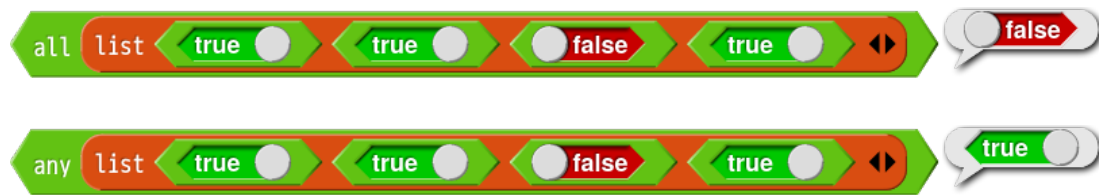


or はどちらかの入力スロットでも true ならば true を返します。もしも左の入力スロットが true ならば、or 演算の場合 true が確定なので右の入力スロットは評価しません。次のスクリプトでは Hello! は表示されません。



and や or ブロックは入力スロットを増設できます。and の場合はすべての入力スロットが true の場合だけ true になり、or の場合はどれか一つでも true ならば true になります。

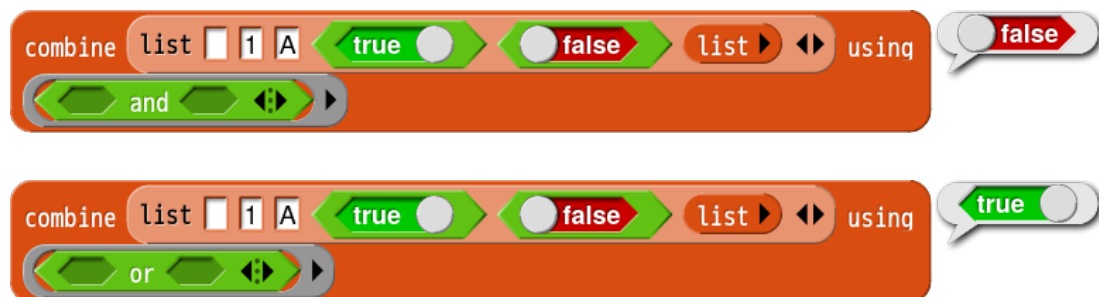
リストの要素に対して and または or 操作をする場合はブロックの右端に有る両向きの三角マークのところにリストをドロップすれば結果が得られます。



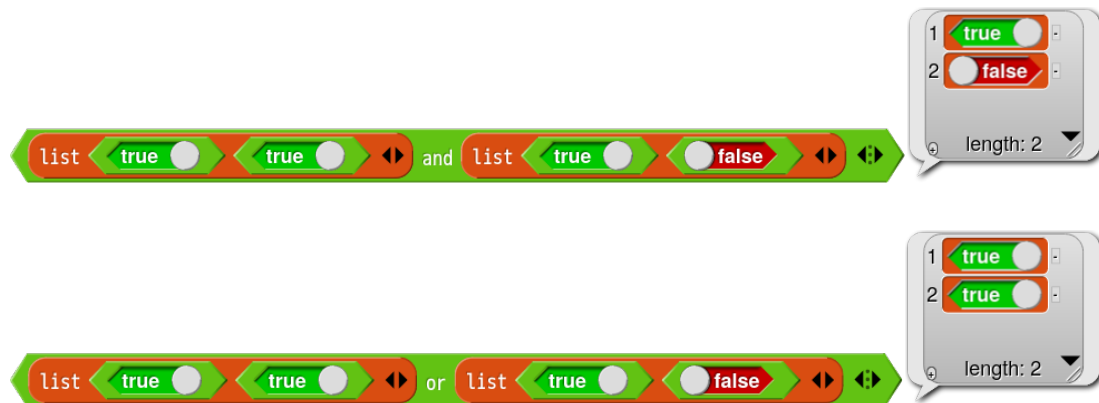
しかし、次のように真理値以外が含まれる場合はエラーが発生します。



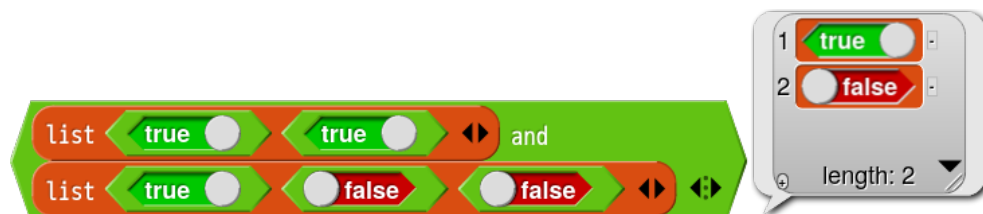
combine ブロックを使用すると大丈夫なようです。



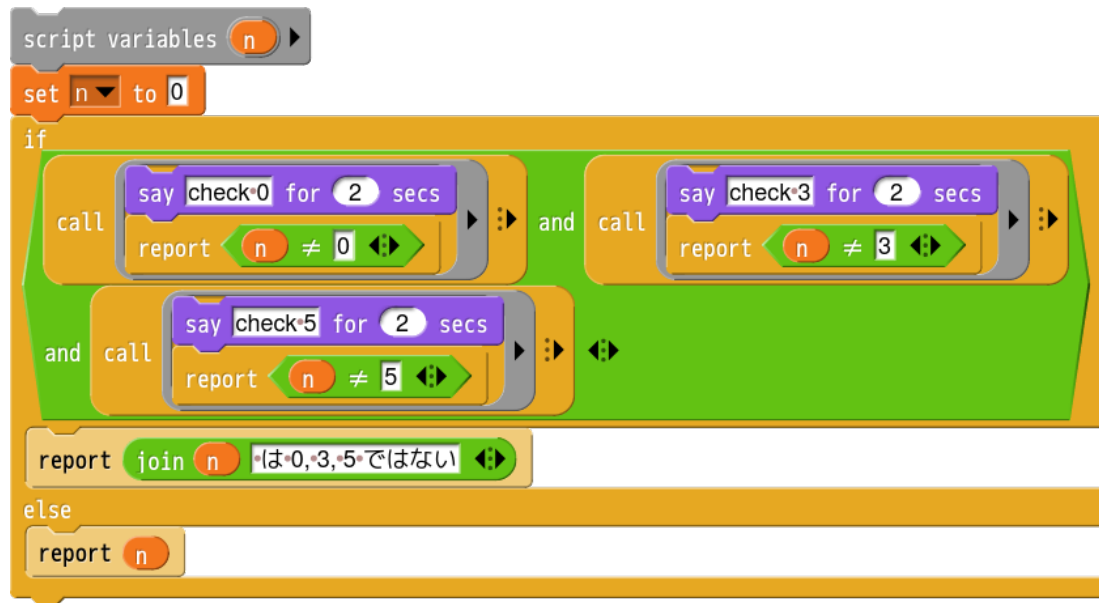
次のようにリスト同士の論理演算をすると先頭要素から同じ位置同士の要素で演算が行われます。



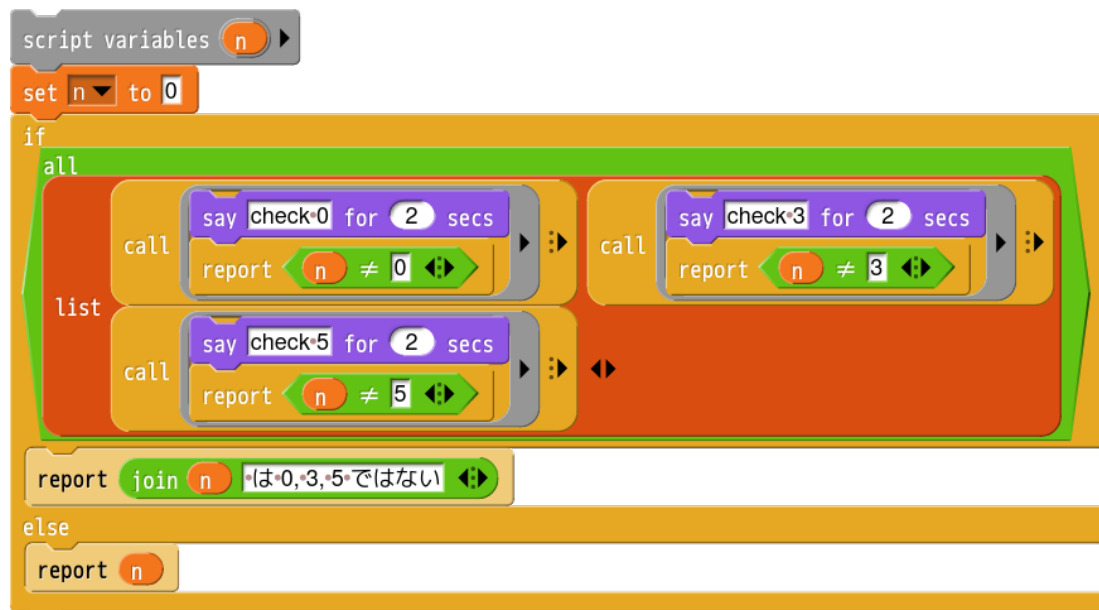
要素数が違うリストの場合は短いリストに合わせた要素だけで行われます。

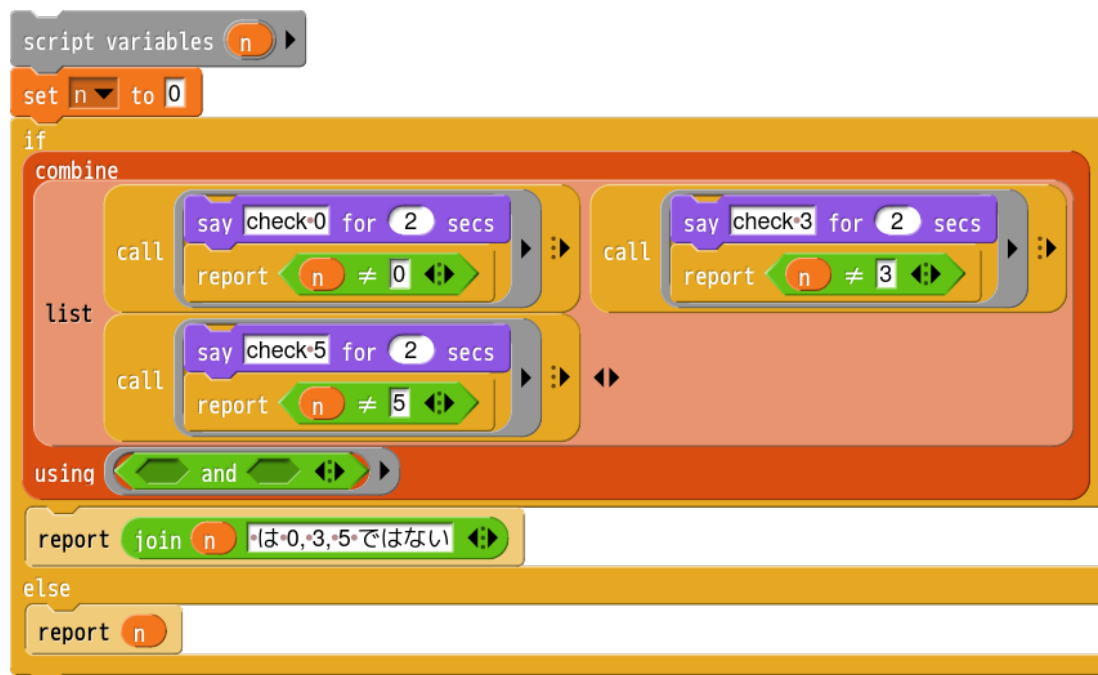


and や or 演算では結果が確定した場合には次のチェックを行いません。次のスクリプトでは最初のチェックで false が確定なので「check 3」「check 5」は表示されません。

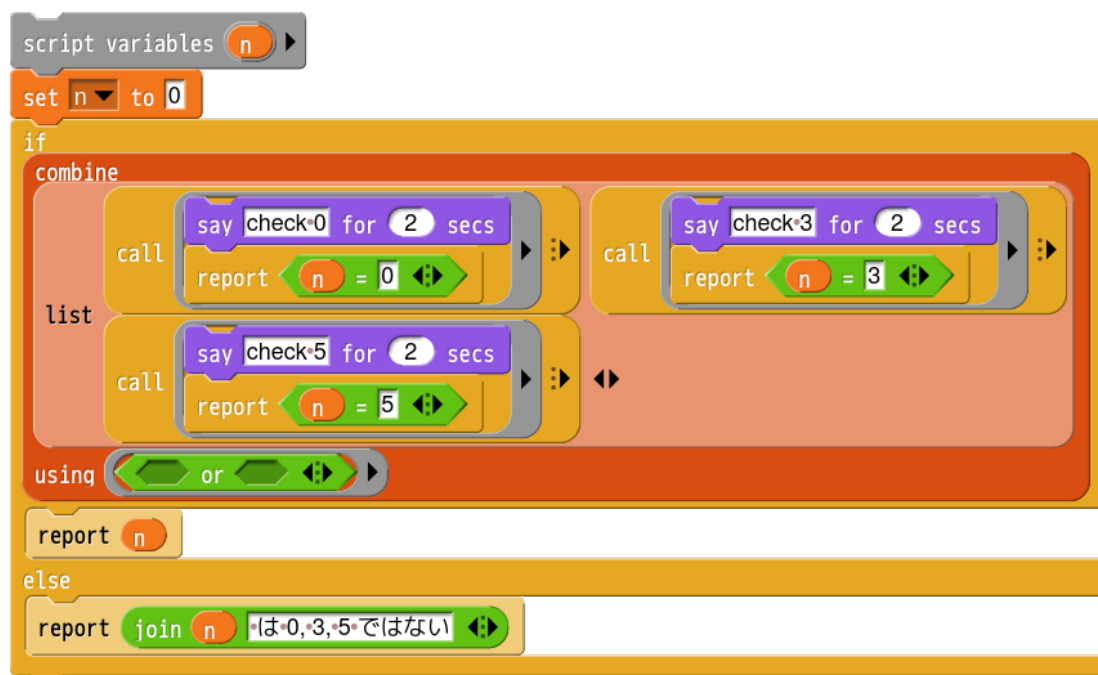


ですが、次のようにリストを使った場合にはリスト要素すべてのチェックが行われます。





or 演算でも同様です。



8.5 集合

Python や Julia では要素に重複がない値の集まり、集合を扱えます。集合を作成するブロック、和集合を求めるブロック、積集合を求めるブロックを作成してみます。

重複や空要素を取り除いて集合を作成します。本来は不要ですが並び替えをしています。



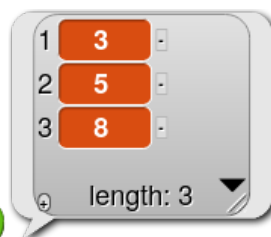
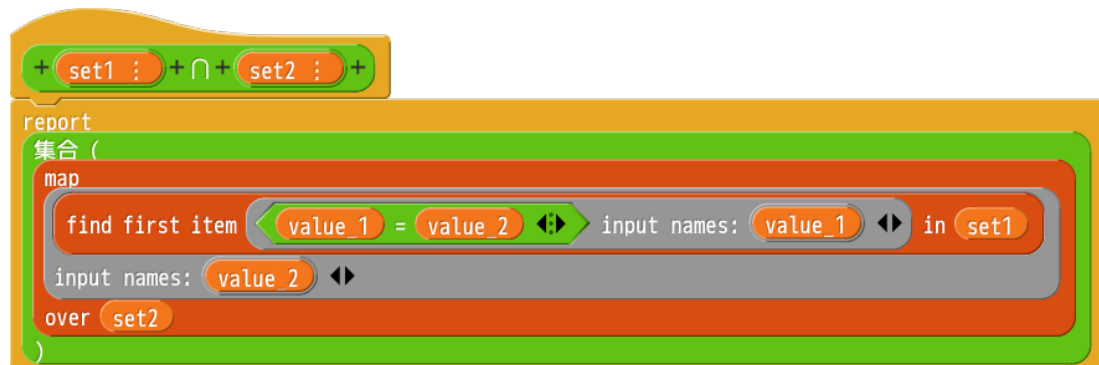
二つの集合を作成します。



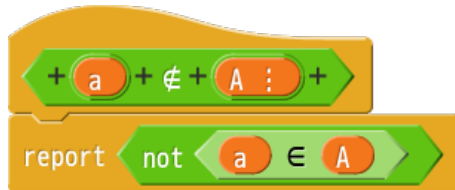
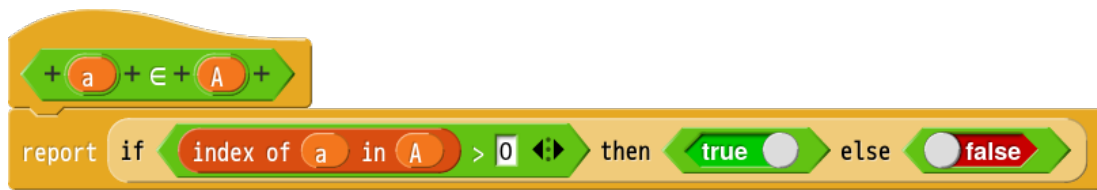
集合同士をまとめ、重複を取り除いた和集合を求めます。



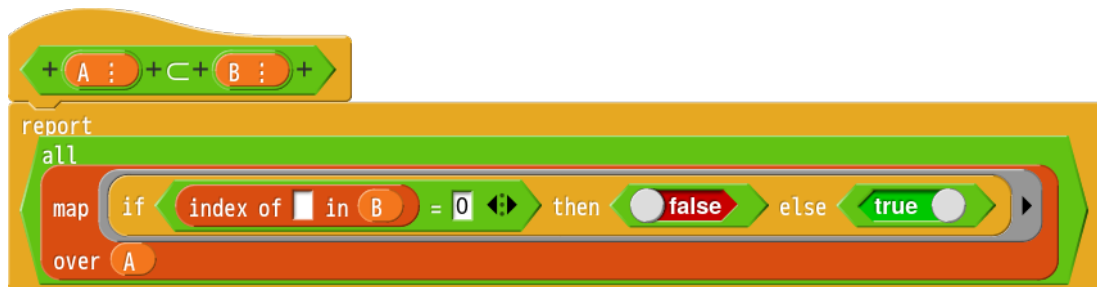
集合同士で同じ値の要素の集まりの積集合を求めます。



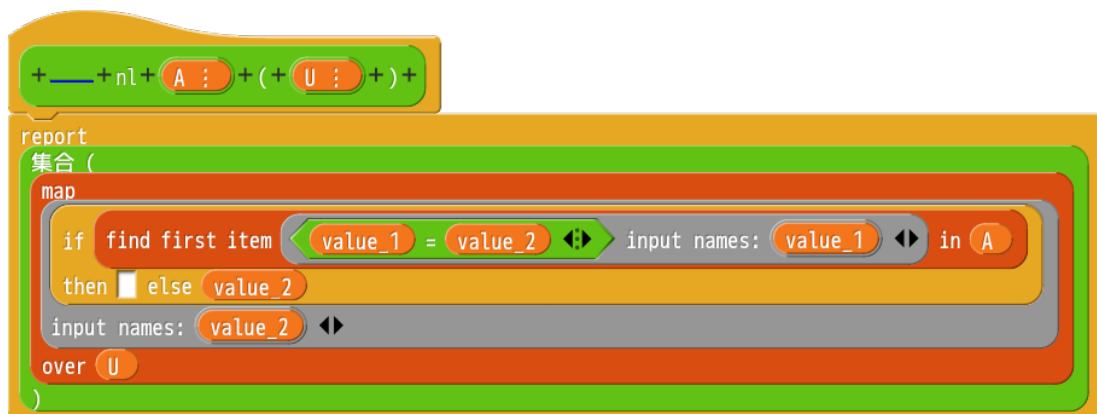
集合に関するブロックをいくつか示します。
要素 a が集合 A に属するか？ 属さないか？



集合 A は集合 B の部分集合か？

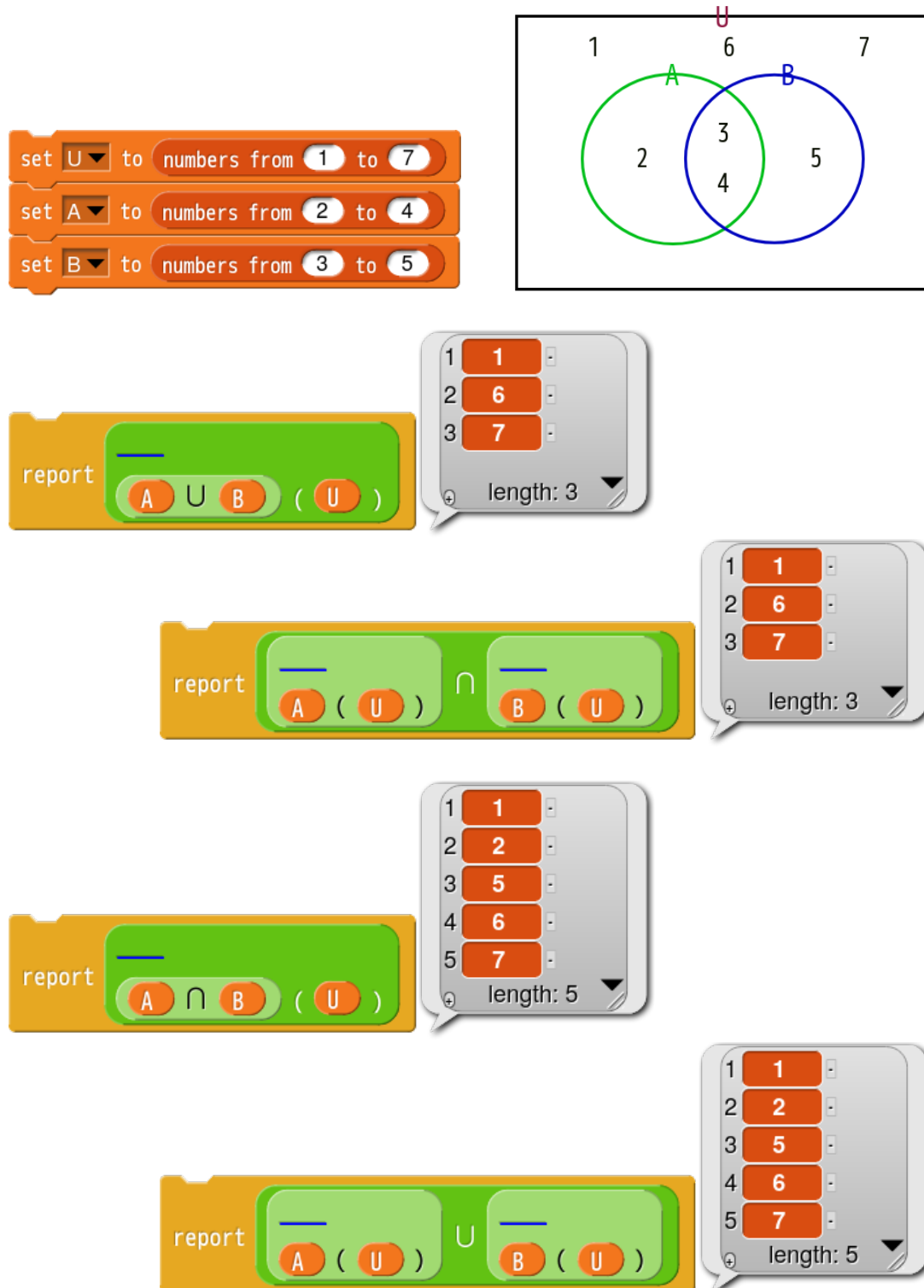


全体集合 U における部分集合 A の補集合。なお、補集合のマークは Title text で \$-2.0-0-0-255 (一は全角のダッシュです) で設定しています。



ド・モルガンの法則で確認してみます。

$$\overline{A \cup B} = \overline{A} \cap \overline{B}, \quad \overline{A \cap B} = \overline{A} \cup \overline{B}$$

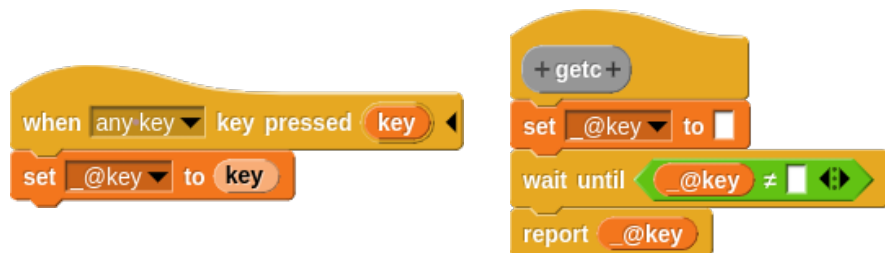


8.6 getc

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



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



で表示させてみると、

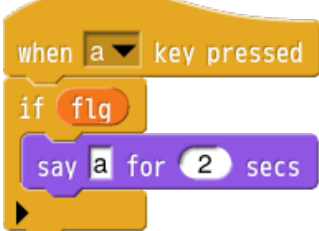
入力キー Shift 入力キー Control
入力キー Backspace 入力キー right arrow 入力キー enter のようにいろいろなキーが使えるのが分かります。大文字のアルファベットも使えます。それぞれ表示されている単語でテストできます。

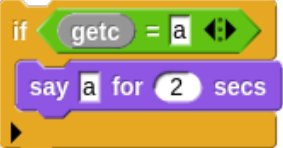
スペースキーは 入力キー 0 と表示されますが、unicode of 入力キー 32 なので

入力キー = true 半角空白でテストできます。

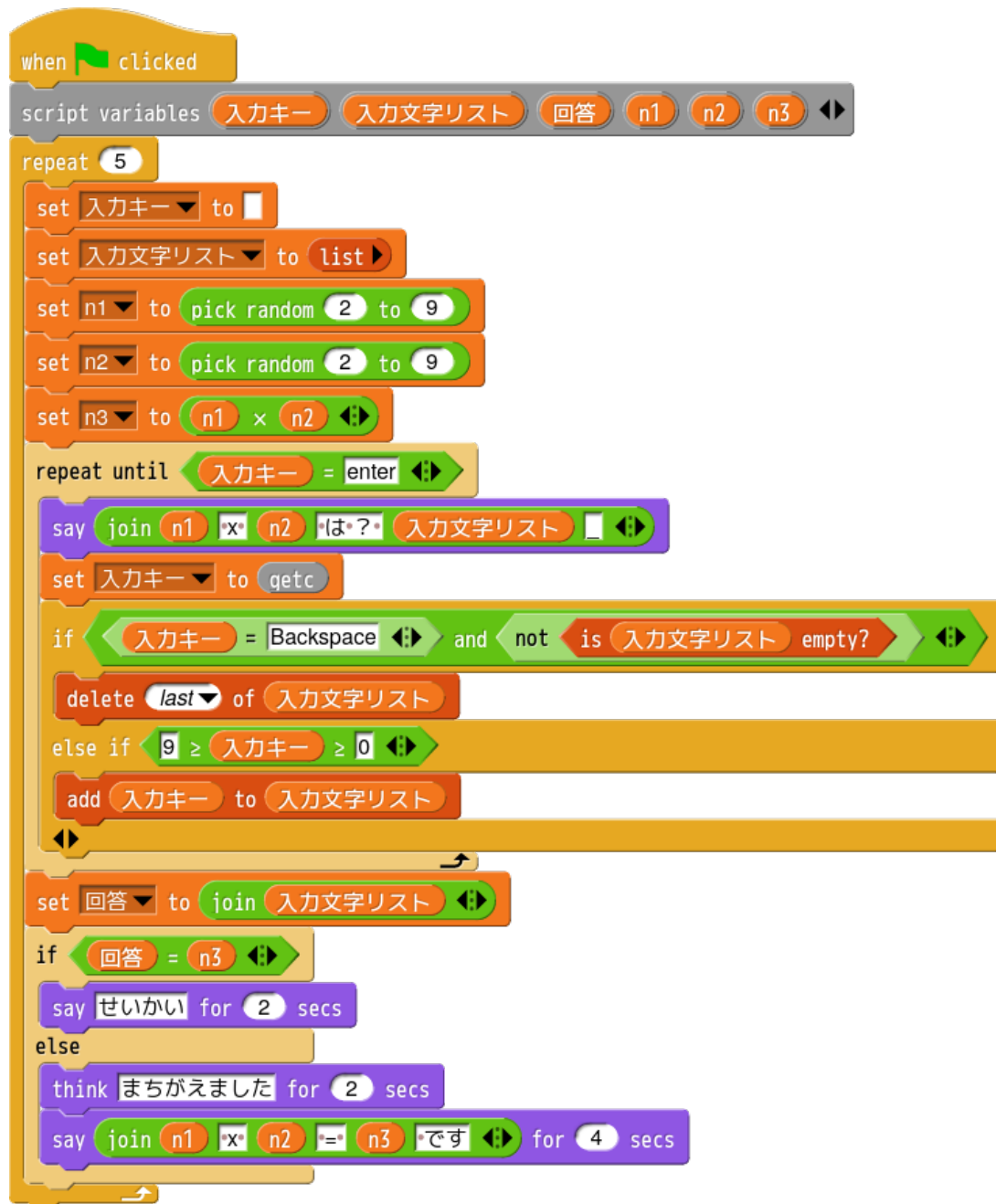
getc を使うことの利点として、キー入力は getc を呼び出した時だけ有効になることです。

例えば、 say a for 2 secs とした場合は a キーを押せばいつでもこのブロックが実行

されてしまうので、 などとする必要があります。

 とするこの時だけになります。

文字列入力の例です。(say ではカーソルとしてアンダースコア _ を表示しています。)
最初に変数「入力キー」に空文字をセットしています。続けて使用する場合に、enter キーが入っていると空入力になってしまいます。それを防ぐためです。



join にリストを入れると文字列 (text) になり、



数字文字列なので数値として扱うことができます。



8.7 when slot () signals ()

 ブロックは Block Editor 内部にしかドロップできません。slot () で指定したカスタムブロックの入力スロットが signals () で指定した操作が行われた時に実行されます。

機能説明のために、numX という値を保持するリポーターブロックを作ってみます。入力スロット X (0), set (set), n1 (1), n2 (10) と、それぞれカッコ内の値をデフォルト値にして作成します。



変数 set は、read-only にしてください。

カスタムブロック numX を作成している Block Editor 内に when slot ブロック、set slot ブロックをドロップして次のように作成してください。slot からは numX に設定されている入力スロット変数名が選択できます。



when slot ブロックを使用すると、次のようなことが可能です。

このブロックをクリックすると先頭の入力スロットに表示されている値をレポートします。

単に  という定義なので、実行されたことによって値が変更されることはありません。



[set] の入力スロットをクリックすると 3 番目と 4 番目の値を使って

 で得た値を先頭の入力スロットにセットします。

この場合は、pick random により 3 が設定されました。



3 番目と 4 番目の入力スロットをクリックして変更可能な状態にしたり、変更すると先頭の入力スロットに新たな値が設定されます。



signals で、menu なら指定の入力スロットがクリックされた場合に指定された処理が実行されます。



edited なら文字通り、指定の入力スロット内で編集作業が行われた場合に指定された処理が実行されます。



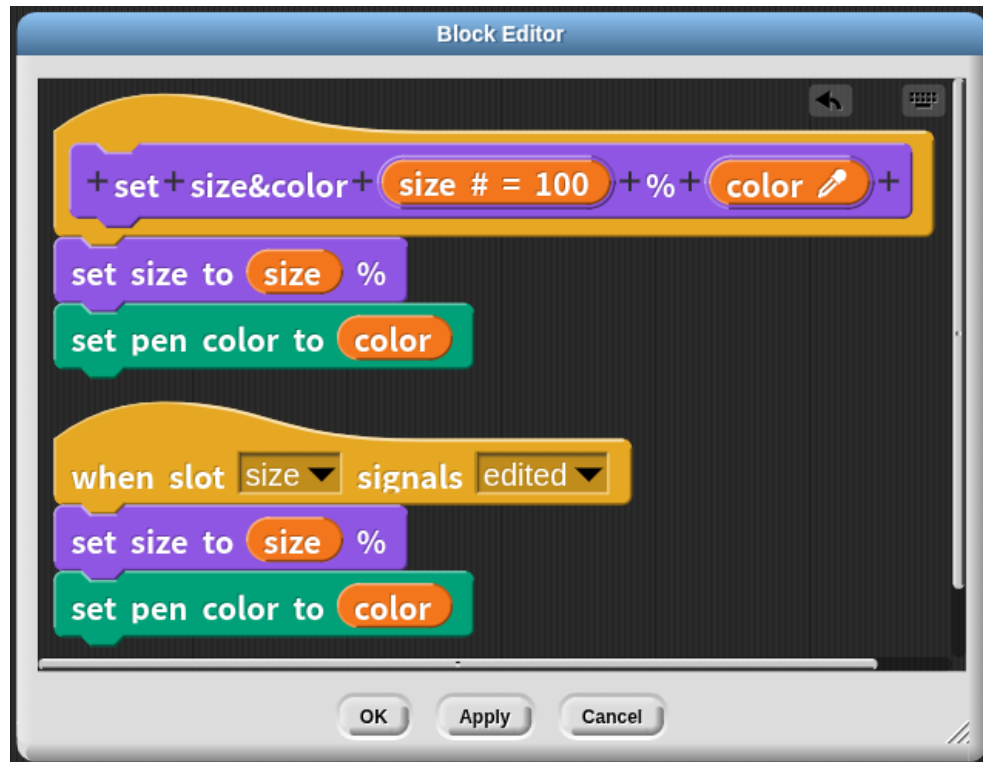
X 先頭の入力スロットの変数に値をセットします。



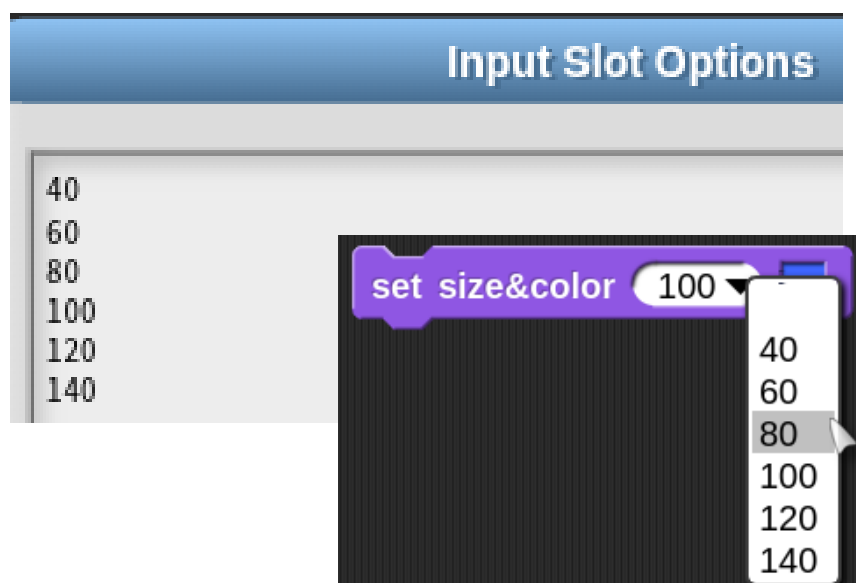
もう一つ例として、スプライトの Turtle コスチュームの大きさと色を変更するブロックです。サイズの部分を編集すると、大きさと色が変更されます。残念ながら color 型の入力スロットは slot で選択できません。色だけの選択では自動的に変更されないのでブロックをクリックする必要があります。



size はデフォルト設定しています。こちらでは set slot ブロックは使用していません。

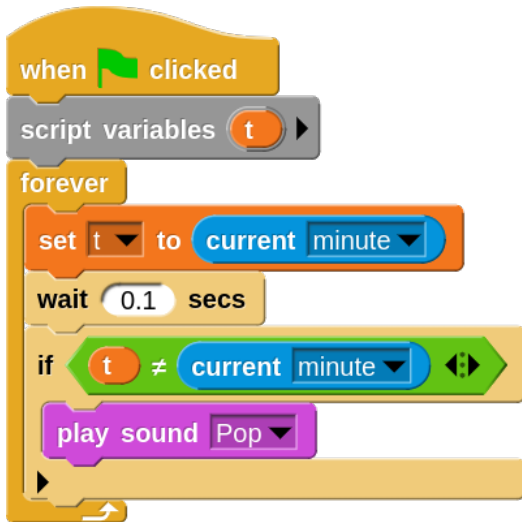


size にオプション設定すると、手入力も可能なので便利です。



8.8 Event Hat ブロック

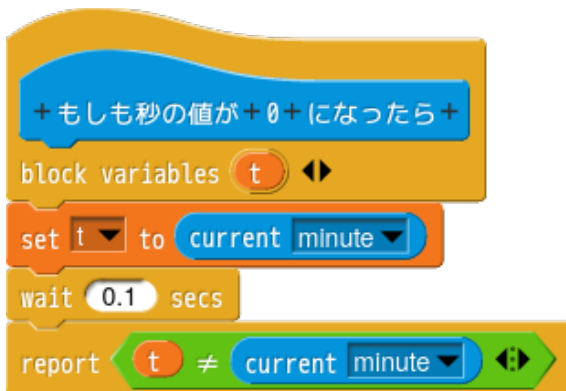
例えば 1 分ごとに音を鳴らす場合、次のようなやり方があります。



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



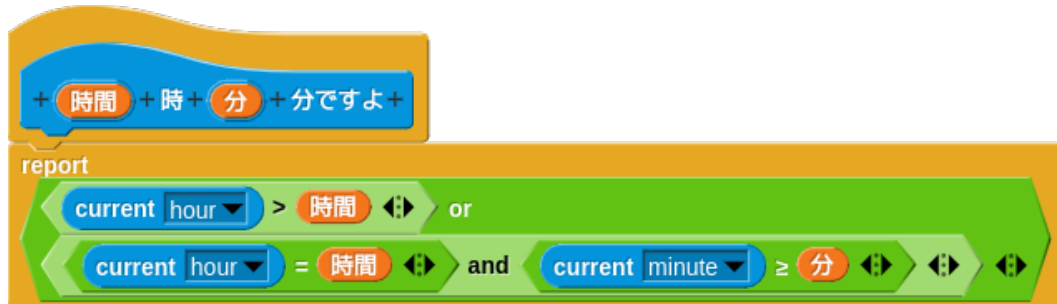
期待する状態になった時に true を返すスクリプトを作成します。



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



指定した時間になったらイベントを発生させるブロックです。

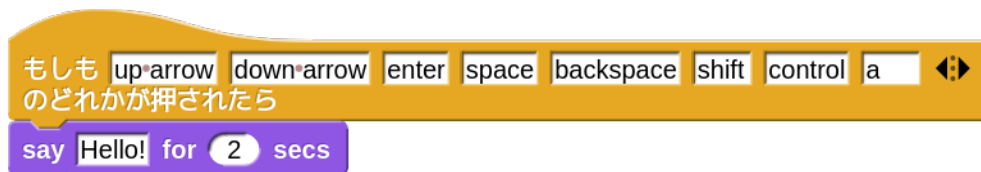


いくつかのキー入力に対して同じ処理を行いたい時には、次のようにできます。
ハットブロックの色は選んだパレットと同じになります。



key は Multiple inputs (value is list of inputs) にしてください。 or ブロックの右端三角マークに map ブロックを入れると any ブロックになります。(109 ページ参照)

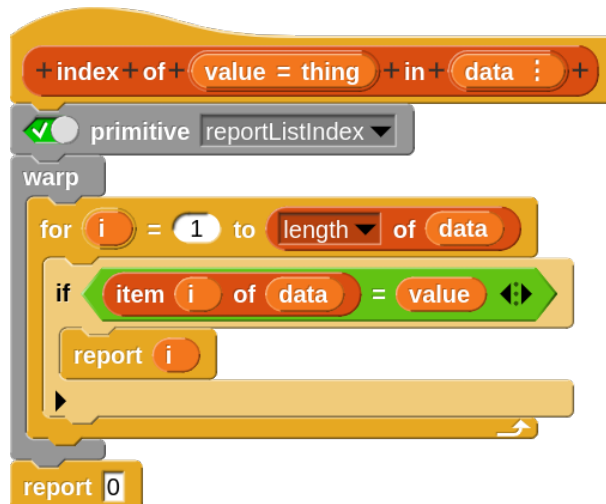
次のようにすると、カーソル↑↓キーか エンターキーか スペースキーか バックスペースキーか シフトキーか コントロールキーか a を押した時に表示させることができます。



8.9 プリミティブブロックの編集

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

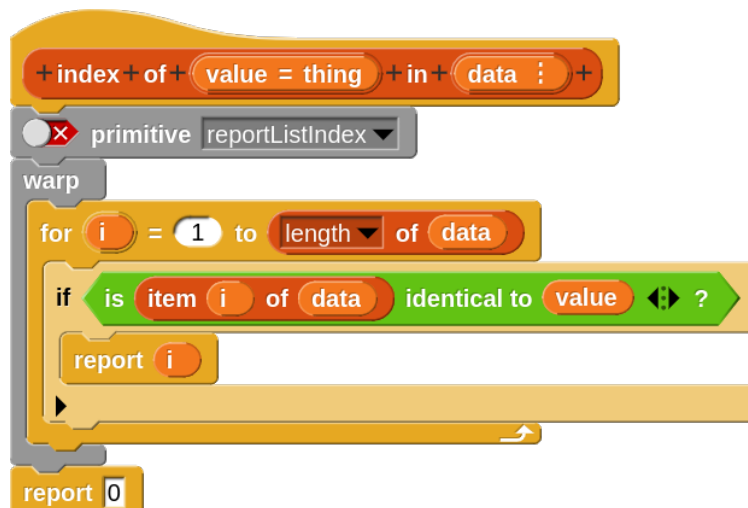
index of thing in を edit で開くと次のようなスクリプトになっています。primitive のスイッチによってプリミティブのコードか、指定されたスクリプトを実行するかを決定します。スイッチがオンなので、この場合プリミティブのコードが実行され warp 以降のスクリプトは実行されません。



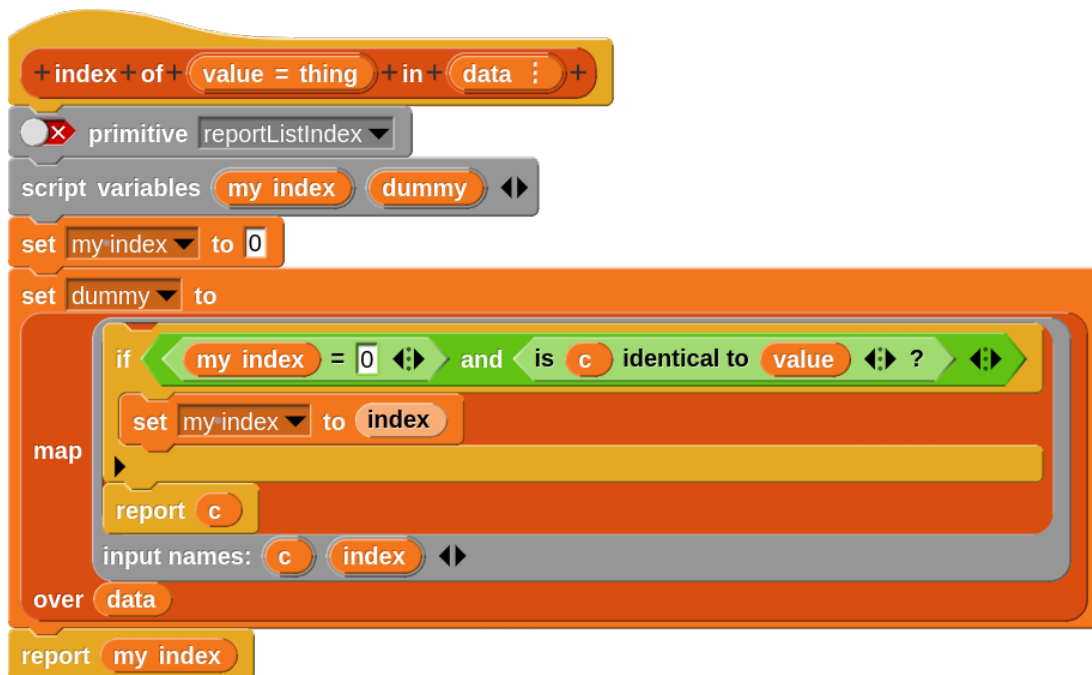
このスクリプトだと、アルファベットの大小文字を区別しないため大文字を指定しても小文字にヒットしてそのインデックスを返してしまいます。



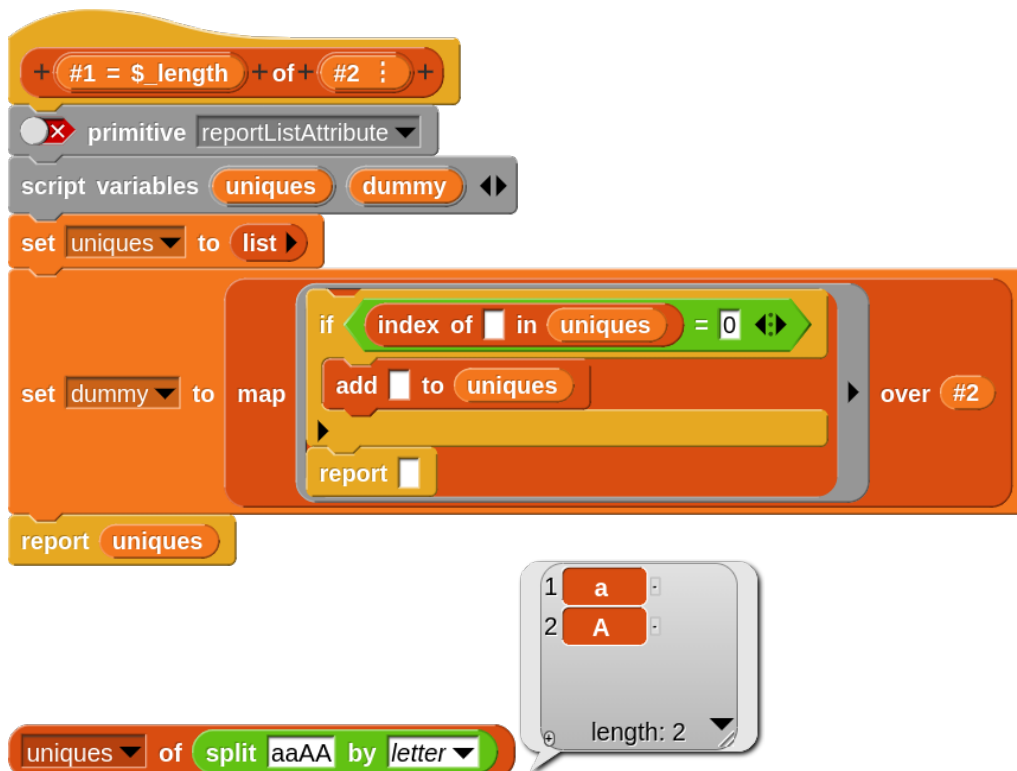
大文字小文字を区別するにはイコールブロックをリラベルして、プリミティブスイッチをオフにします。



map を使用して次のようにすることもできます。



プリミティブの uniques ブロックは大文字のアルファベットを小文字に変更してしまいます。上記のように index ブロックが変更されていると、uniques ブロックも大文字小文字を区別することができます。



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

8.10 read-only 入力スロット

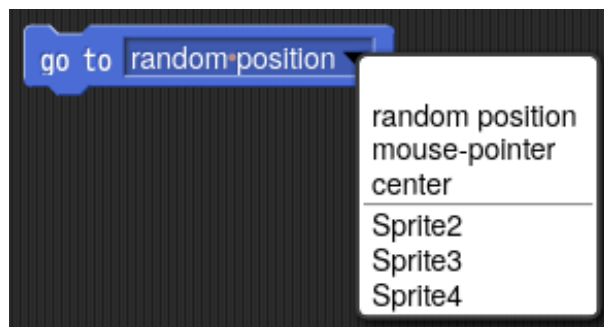
次のように背景が白い入力スロットはキーボードからの入力を受け付けます。



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



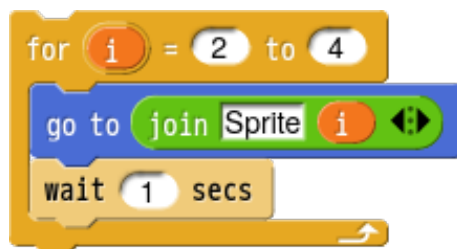
例えば、Sprite, Sprite2, Sprite3, Sprite4 がある場合、次のようになります。



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



これを利用すると次のような使い方ができます。



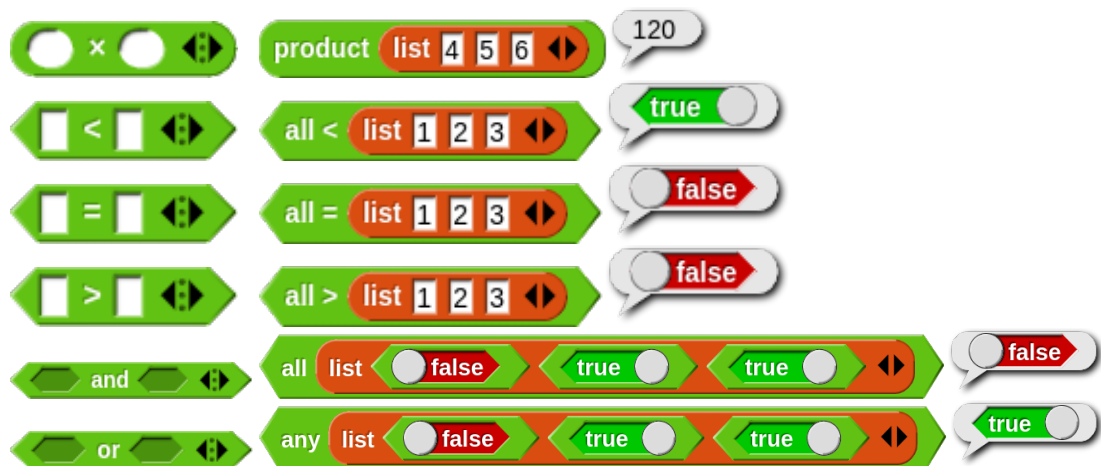
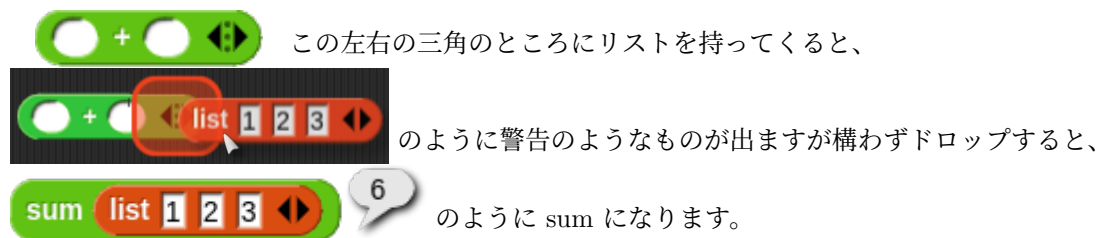
乱数を使って次のようにすることもできます。



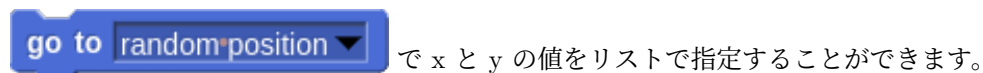
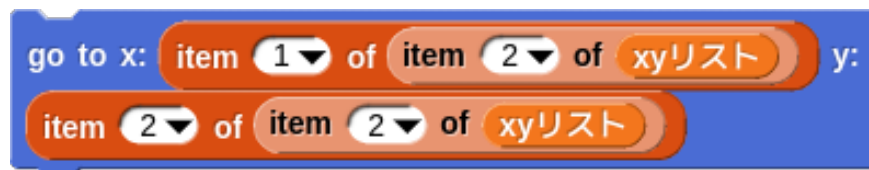
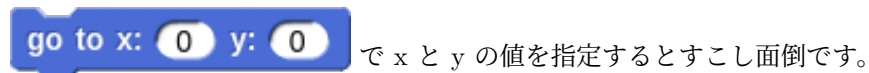
このやり方は他の tell や switch to costume などでも有効です。

8.11 入力スロットへのリスト指定

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



xy 座標を次のようにリストにした場合、

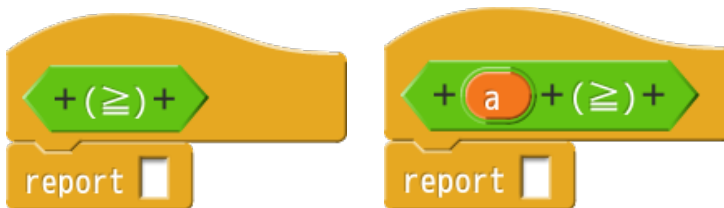


8.12 入力スロットの追加

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



a という変数を作成します。



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

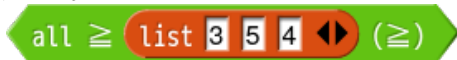
右下の歯車のようなアイコンをクリックすると右図のメニューが出るので separator, collapse, initial slots の項目をそれぞれ変更します。



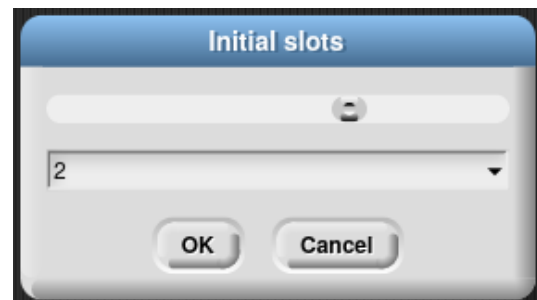
Separator は各スロットの間に入る項目です。
 この場合は「 $\geq$ 」になります。全角文字「きごう」とか「だいたいこーる」で変換すると出てきます。



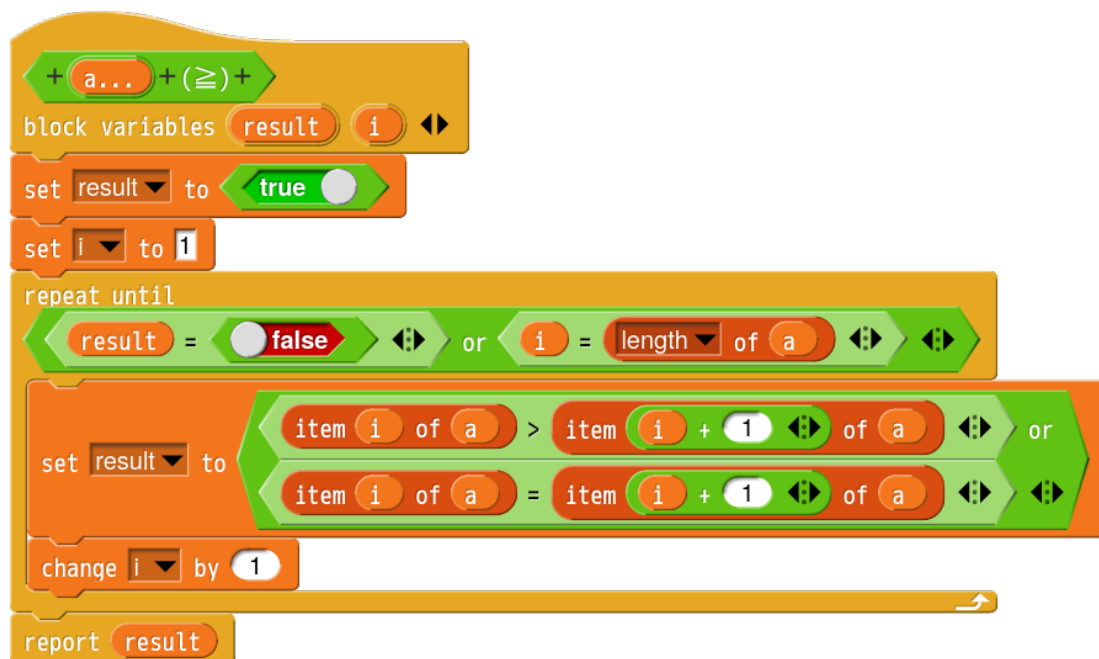
Collapse はリストを入力して、スロットがない場合の表示項目です。この場合は、「all $\geq$ 」になります。



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



変数 a の表示が変更されます。ブロック定義の本体は次のようになります。入力を値 2 つではなくリストとして受け取るので少し複雑になります。

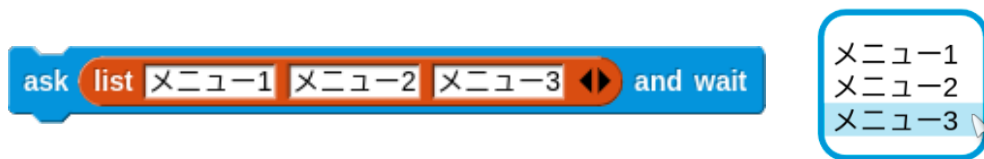


($\geq$) を削除するとプリミティブブロックのような表示になりますが、続けて $\leq$ のブロックを作成した場合にプロトタイプが同じになるので (2) が表示されてしまいます。なにか方法があるのかもしれませんが。

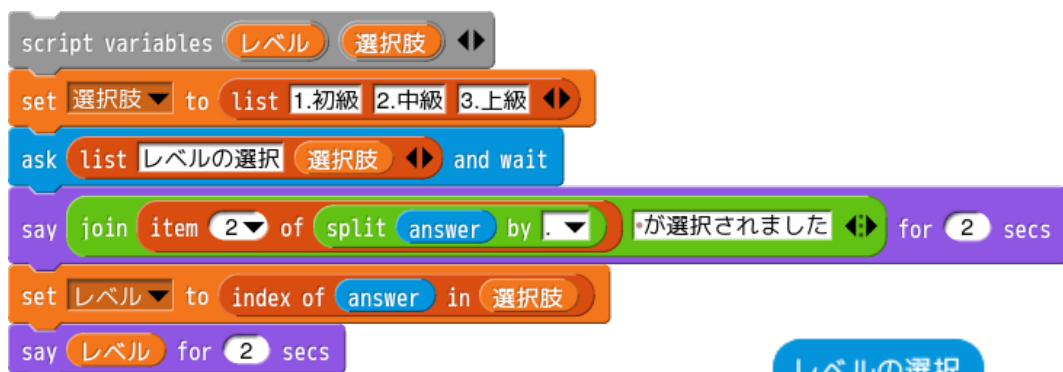
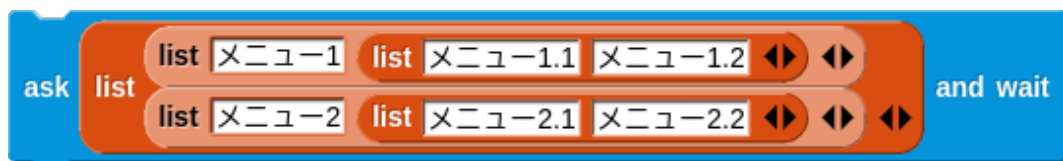
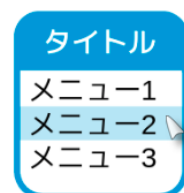
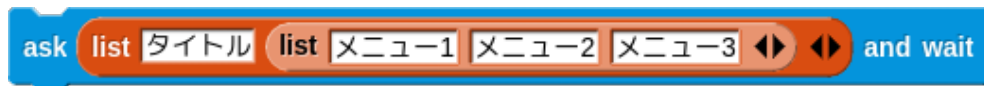
8.13 ask

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。



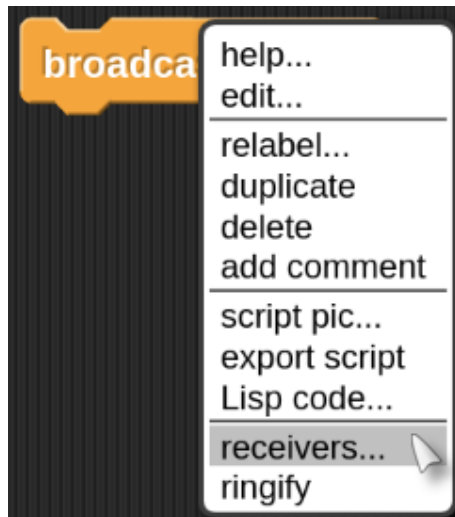
リストを組み合わせるとタイトル（説明文）を付けたり、サブメニュー構造にすることもできます。



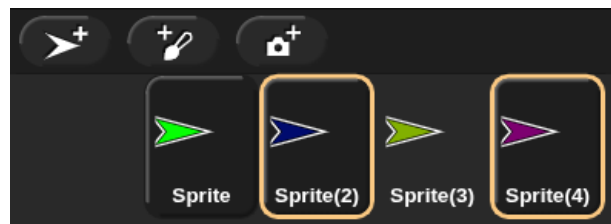
8.14 broadcast の検索オプション

例えば、Sprite で **broadcast test** を使用して、

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... オプションが役に立ちます。



broadcast test ブロックを右クリックすると表示されるオプションから receivers... を選択します。すると、スプライトコラルの該当するスプライトが一瞬マークアップ表示されます。



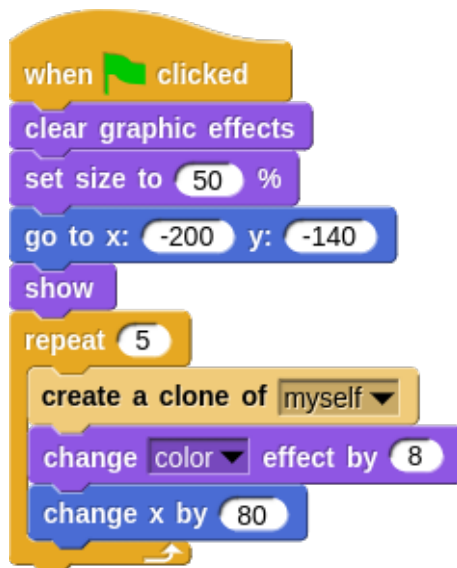
8.15 クローン

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。

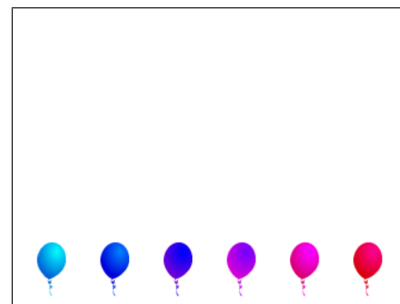
8.15.1 テンポラリクローン

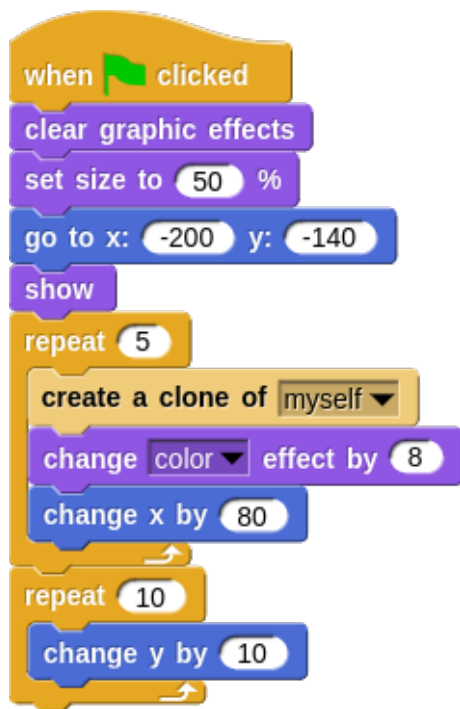
テンポラリクローンは Scratch でのクローンと基本的には同じものです。

次のようにして風船のクローンを作成してみます。

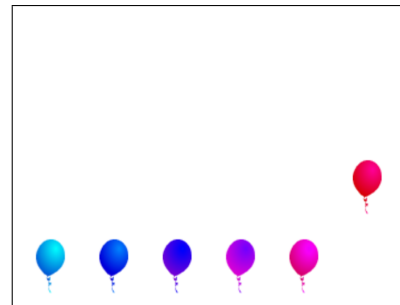


このスクリプトでは「show」により、作成された5個のクローンが左から表示され、右端にクローンではない本体も表示されるので合計6個の風船が出現します。

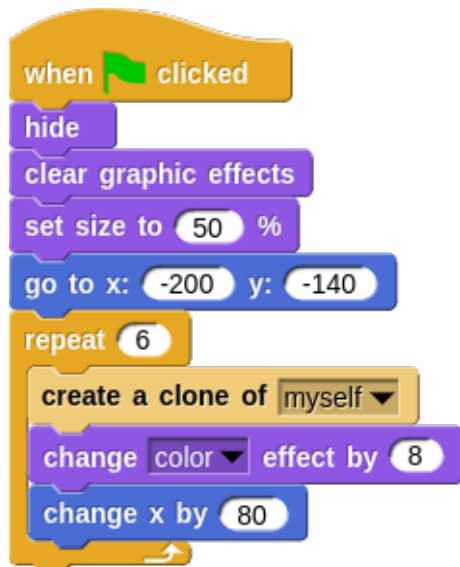




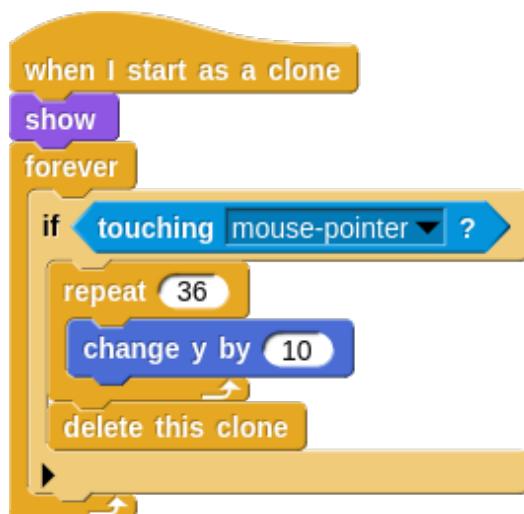
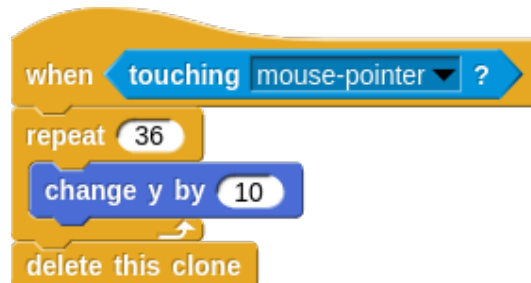
このようにして風船を移動させるスクリプトを追加すると、本体だけが移動します。つまり、本体用のスクリプトではクローンの操作はできません。普通は本体を存在させると都合が悪いので、非表示にしてクローンのみを操作します。



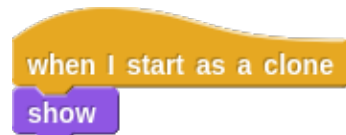
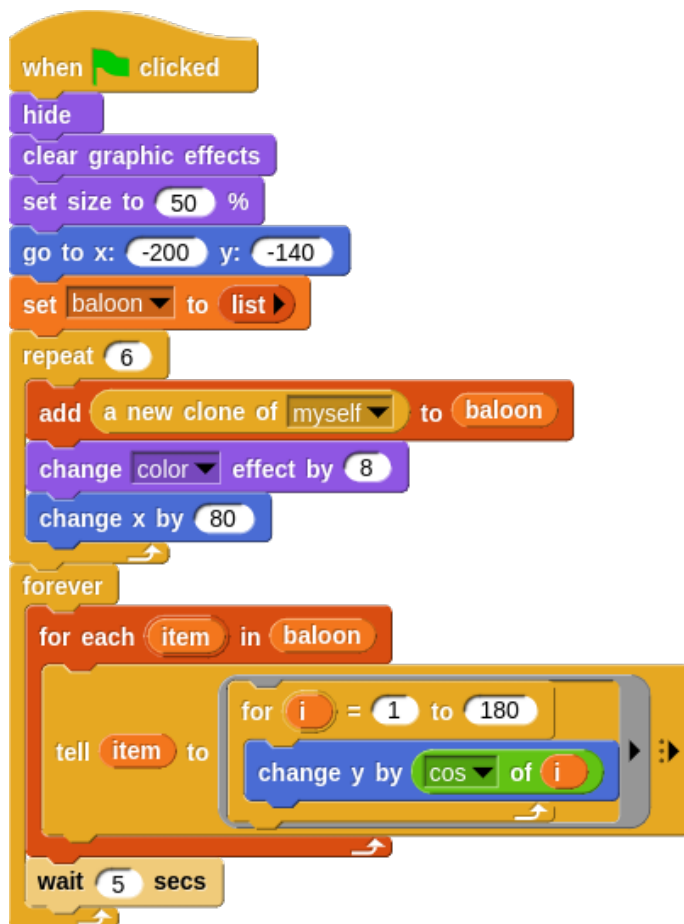
表示は「when I start as a clone」内で行います。



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。



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

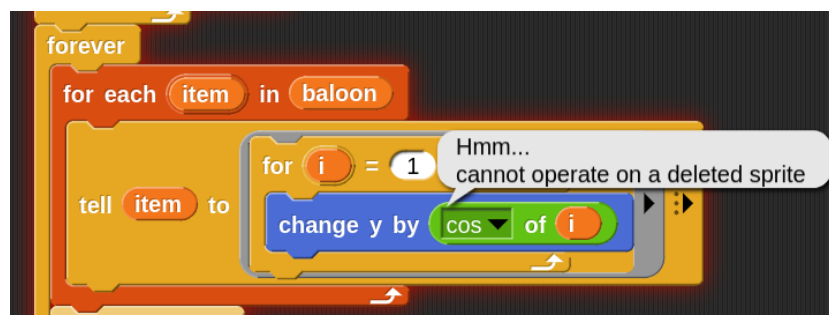


Snap! では作成したクローンを一時的にリストなどの変数に入れることができるので、それを使って対象を指定して操作することができます。



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こります。

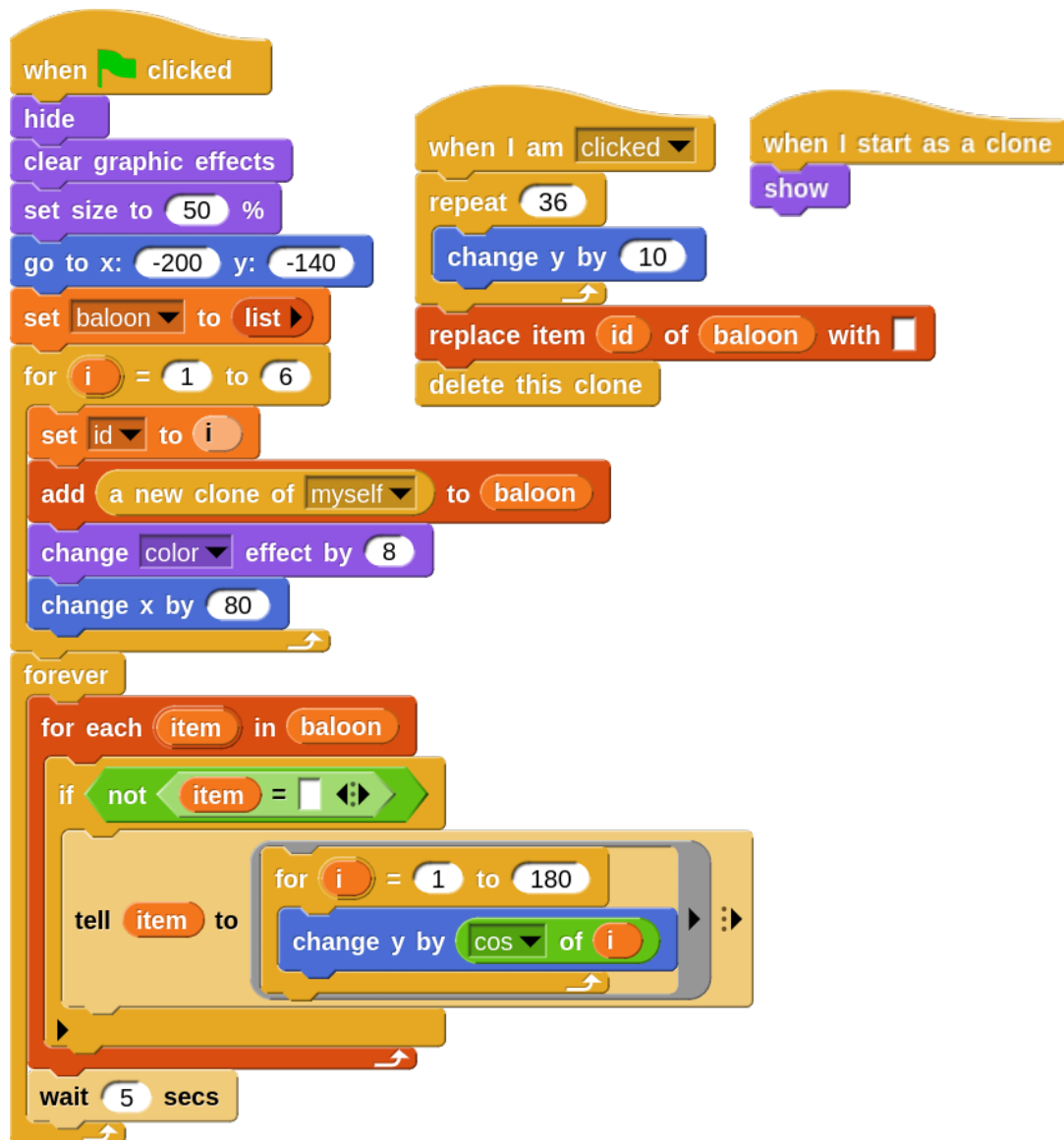
クローンを格納したリストの要素が削除済みということで、「X」になっています。「削除されたスプライトに対して操作することはできない」というエラーになります。





このようにすると、クリックされたクローンは削除されます。しかし、wait している時なら問題ないのですが、for each 実行中にクリックすると期待するような item の順番にならないようです。

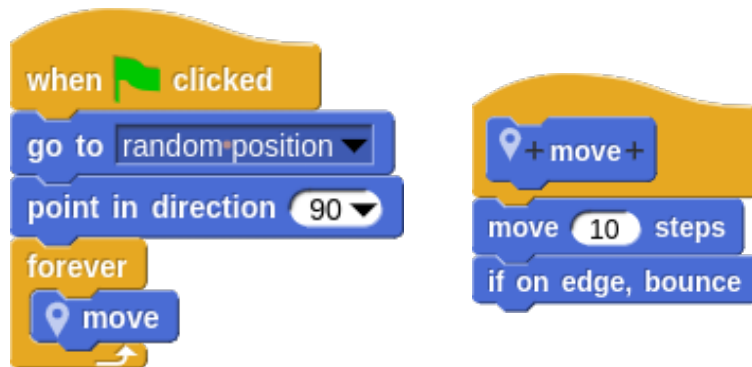
これに対処する一つの方法です。スプライト変数は各スプライトの中だけで有効な変数です。クローンもスプライトなので、スプライト変数を使用するとそれは同じ名前でも各クローン独自のものです。スプライト変数  **id** に baloon リストの位置番号を記憶させます。そうすると、クローン1は1、クローン2は2、... と、各クローンはリスト位置を示す変数 id を持ちます。削除対象のクローンの id をインデックスとして baloon リストの要素を 空 で書き換えます。baloon の要素が 空 の時は、tell 操作をしません。



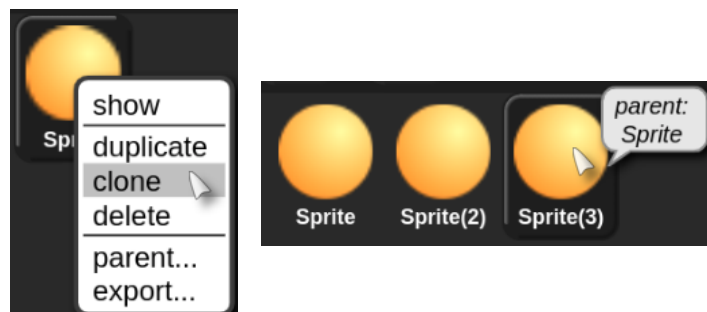
8.15.2 パーマネントクローン

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。

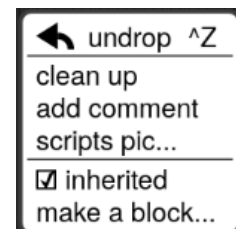


スプライトコラールで Sprite を右クリックして、クローンを二つ作成します。作成されたクローンのところにマウスカーソルを置くと、parent: 親が Sprite であることを表示します。



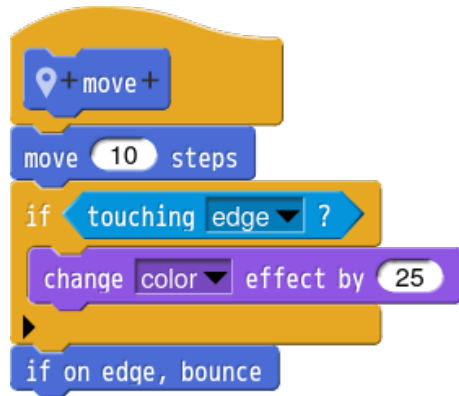
クローンである Sprite(2), Sprite(3) には、Sprite と同じスクリプトがコピーされています。親である Sprite のスクリプトを変更すると、子である Sprite(2), Sprite(3) のスクリプトに反映されます。例えば、サイズを 50 などと変えてみてください。

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

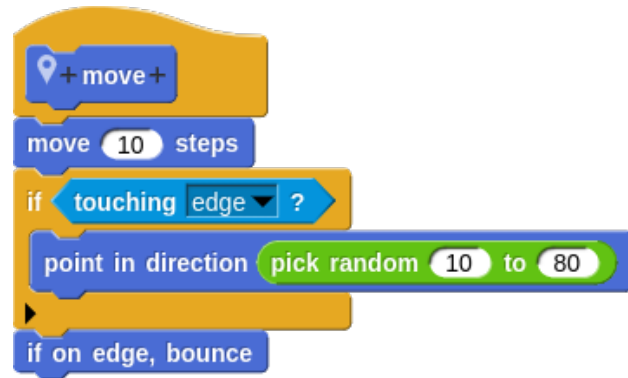


クローンを作成した時に親側のユーザー定義ブロックも子側にコピーされています。しかし、この move ブロックは for this sprite only で作成されているので、親側で変更しても子側は変わりません。子側で変更しても inherited の状態は変わりません。次のように子側の move 定義を変更してください。各スプライトで同じ名前の move ブロックですが、それぞれ違った動作をさせることができます。

Sprite(2)



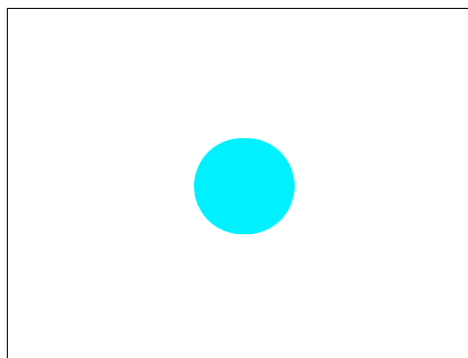
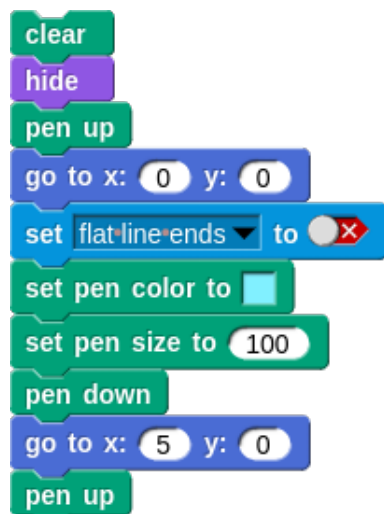
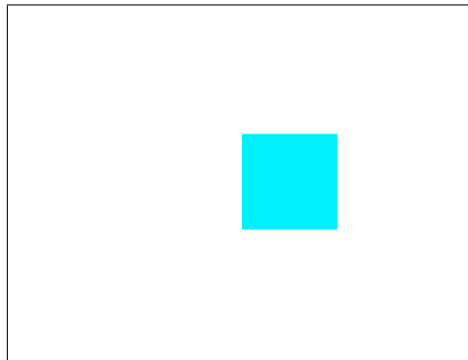
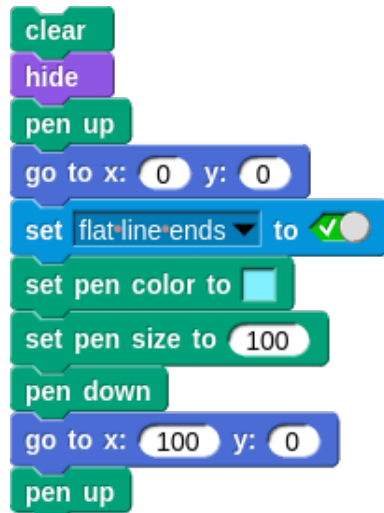
Sprite(3)



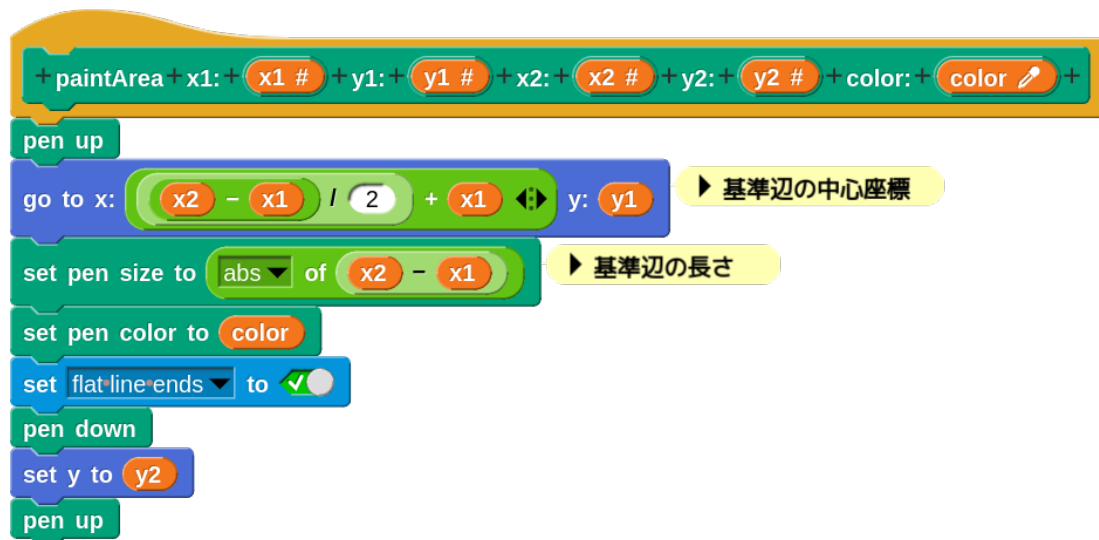
8.16 flat line ends

Sensing のパレットに **set video capture to** があります。

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

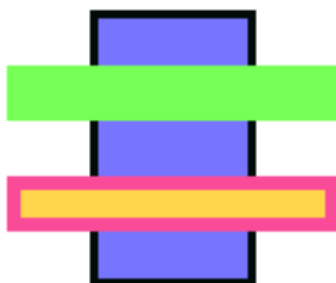
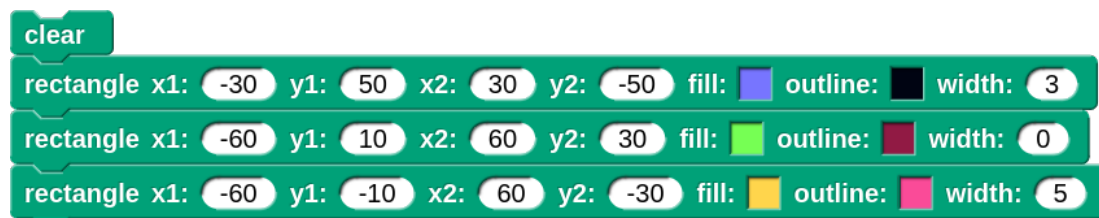


flat line ends を利用すると、対角の座標を指定してその矩形エリアを塗りつぶすことができます。color: で色の指定ができます。背景の色を指定するとクリア指定になります。



他の言語には rectangle のような矩形塗りつぶし命令があります。Snap! には fill ブロックがあるので、枠線で囲い内部を塗りつぶすことができますが、他の図形と重なっていたりすると一度の fill では塗りつぶせないですし、塗りつぶすために内部の座標を指定するのも面倒なことです。paintArea のようなやり方をすると rectangle 相当ブロックを作成することができます。

fill: は塗りつぶす色の指定で、outline: は枠線の色です。width: は枠線のサイズの指定です。サイズ 0 で枠線なしになります。なお、この定義では矩形の大きさは枠線のサイズにかかわらず指定のものになります。



定義です。枠線色の矩形の中に枠線分だけ小さい fill 色の矩形を描くことで rectangle 命令文と同じような描画になります。他のスクリプトに影響を与えないようスプライトの位置やペンの情報をキープさせています。



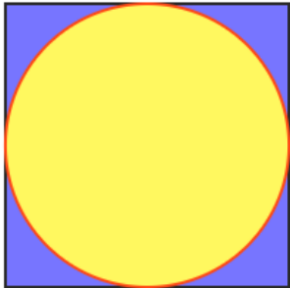
fill 色で指定の矩形を描き、各角の頂点を指定のサイズと枠線色の線で結べば一般的な rectangle 命令文のように枠線のサイズ分大きな矩形になります。

参考までに、flat line ends を利用すると円を描くこともできます。r: は円の半径の指定です。なお、この定義では円の大きさは枠線のサイズにかかわらず指定のようになります。

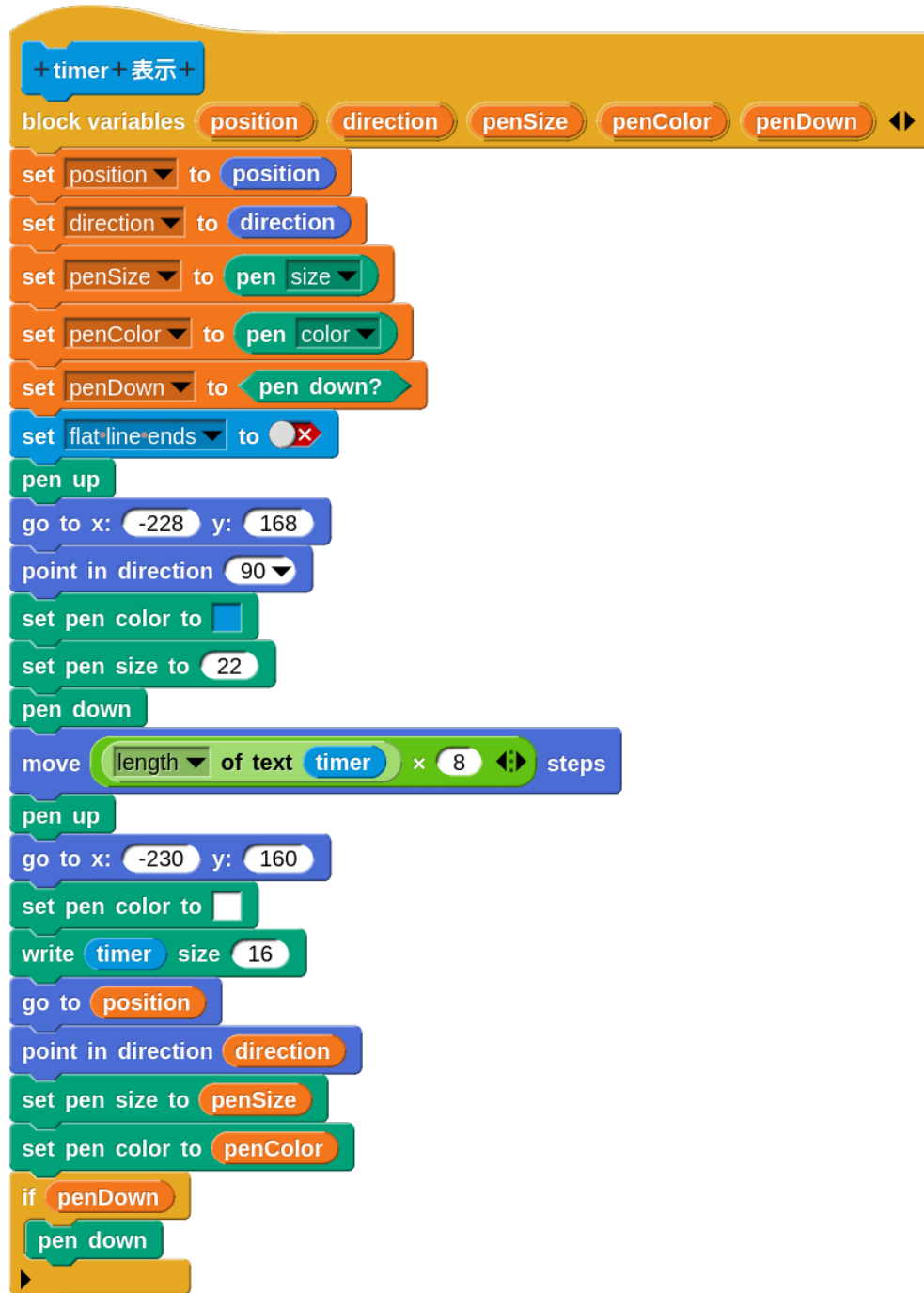
```

+circle+x:+ x # +y:+ y # +r:+ r # +fill:+ fill +outline:+ outline +width:
+width # +
block variables position pinSize penCollor
set position to position
set pinSize to pen size
set penCollor to pen color
pen up
set flat-line-ends to
set pen size to r × 2 ▶ 円の直径
go to x: x y: y ▶ 円の中心へ
set pen color to outline
pen down
go to x: x y: y ▶ 枠線で全体円を描く
set pen size to r - width × 2 ▶ 枠線分を除いた直径
set pen color to fill
go to x: x y: y ▶ 枠線を除いた円を描く
pen up
go to position
set pen size to pinSize
set pen color to penCollor
clear
rectangle x1: -100 y1: 100 x2: 100 y2: -100 fill: outline: width: 2
circle x: 0 y: 0 r: 100 fill: outline: width: 2

```



プログラムの実行中に何かデータを表示するには変数ウォッチャーを使ってリポートすることができますが、大きさや色などを変更したい場合があります。次のように Pen 機能を使用すると、timer 表示の代わりとすることができます。



100 ページで getc での入力を say ブロックで行いました。代わりに Pen パレットの機能を使ってみます。99 ページで定義した getc, when any key ブロックも使用します。

21

script variables 入力キー 入力文字リスト size xy

```

hide
set 入力キー to 
set 入力文字リスト to list
set size to 40 ▶ 入力文字の大きさ (これにより削除エリアも調整される)
set flat-line-ends to 
clear
pen up
go to x: -200 y: 0
set pen color to 
write 3*x*7*は*?* size size
set xy to position ▶ 入力位置
repeat until 入力キー = enter
  write  size size ▶ カーソルとして下線を表示
  set 入力キー to getc
  if 入力キー = Backspace and not is 入力文字リスト empty?
    delete last of 入力文字リスト
  else if is 入力キー a number? and length of 入力文字リスト < 2
    add 入力キー to 入力文字リスト
  go to xy ▶ 入力エリアをクリア
  change y by size / 3
  set pen color to 
  set pen size to size + 2
  pen down
  change x by size + size / 2 ▶ 3 文字分として調整
  pen up ▶ 入力文字列を表示
  go to xy
  set pen color to 
  write join 入力文字リスト size size
report join 入力文字リスト

```

削除エリアを分かりやすくするための色付き
本来は背景色にする

8.17 角度

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

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

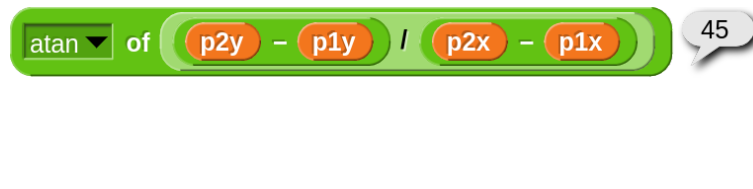
一般的な角度で指定するには、



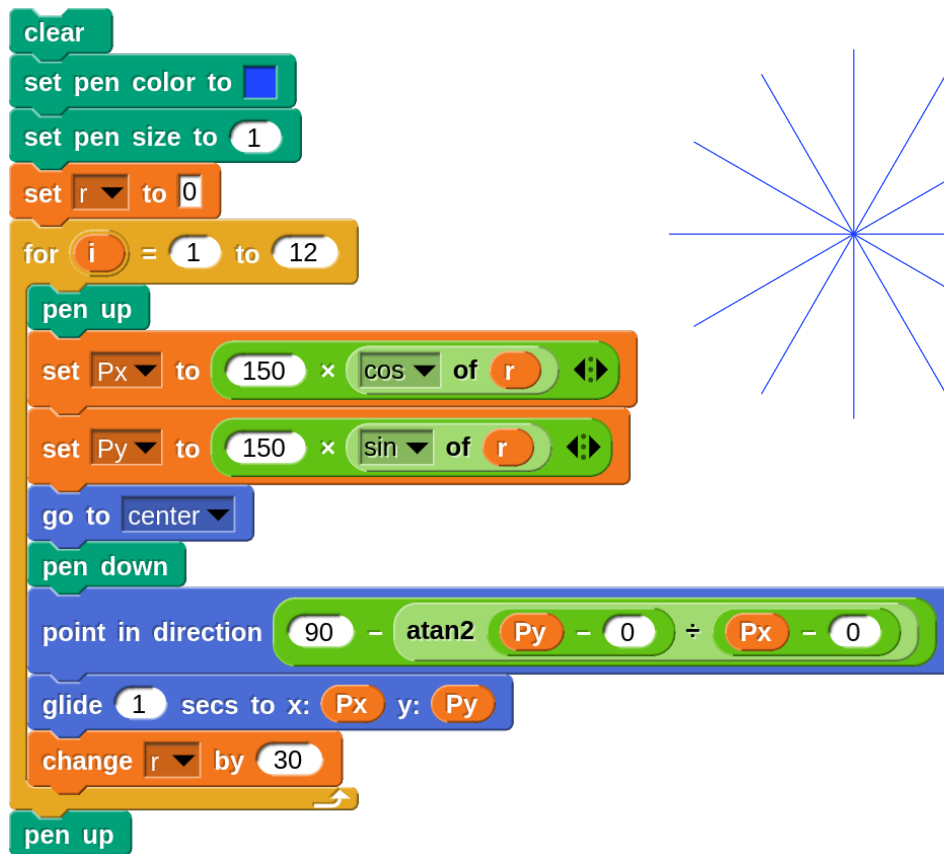
または、

point in direction **90 - r** とします。

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



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

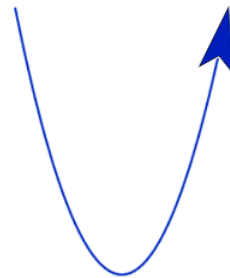


方程式のグラフを描く場合、微分によって接線の傾きが求められるのでスプライトの角度に利用することができます。

```

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

```



```

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

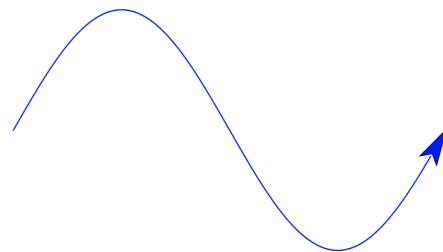
```

次の座標位置へ向くというやり方もあります。また、atan2 ではなく方向の指定を Sensing にある direction を使う方法もあります。

```

direction to list x + 1 sin of r + 1 * m

```



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

```

100 / 0
atan of 100 / 0
atan2 100 ÷ 0

```

8.18 layer レイヤー

スプライトはそのスプライト用の透明な板のようなものに表示され、複数のスプライトがあるならばそれらが画面上に重ねられて見えるというようなイメージです。その板のような層がレイヤーであり、そのレイヤーの位置を指示することでどちらが手前側に表示されるかを設定します。

レイヤーを設定するブロックです。



スプライト用のレイヤーの奥に Pen 機能で描画に使用されるレイヤーがあり、一番奥に Background 背景があります。

例えば、 と  のスプライトを重ねると、設定によって



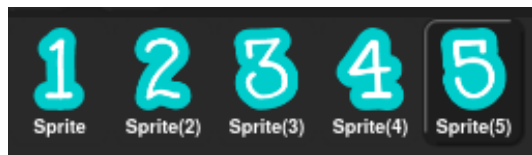
入れ物のスプライトをスタンプするか奥のレイヤーにし、ペンギンを上図右側のようにその手前のレイヤーにします。入れ物のスプライトから作成した  を一番手前のレイヤーにして重ねると入れ物の中に入っているように見せることができます。



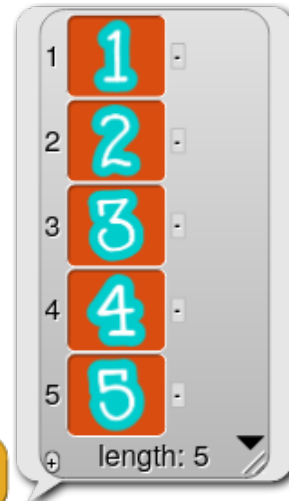
これらをステージ最下部に置き、ペンギンを下から上に動かすと、入れ物の中から出てくるように見えます。



次のような数字のスプライトを 1 から 5 の順に作成すると、



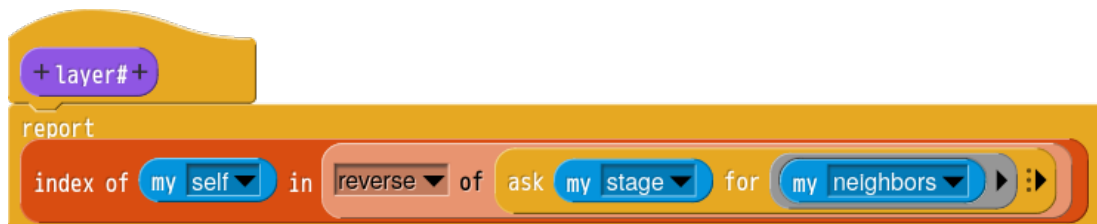
各スプライトのレイヤーの位置は、右のように作成された順に奥から配置されるので 5 が一番手前に表示されることになります。
ドラッグされるとそれが一番手前に変わります。



次のようにすると、そのスプライト (5) の奥からの順番が求められます。



ask ブロックからの値を reverse させると手前からの順番になります。手前からの順番をレポートするブロックです。

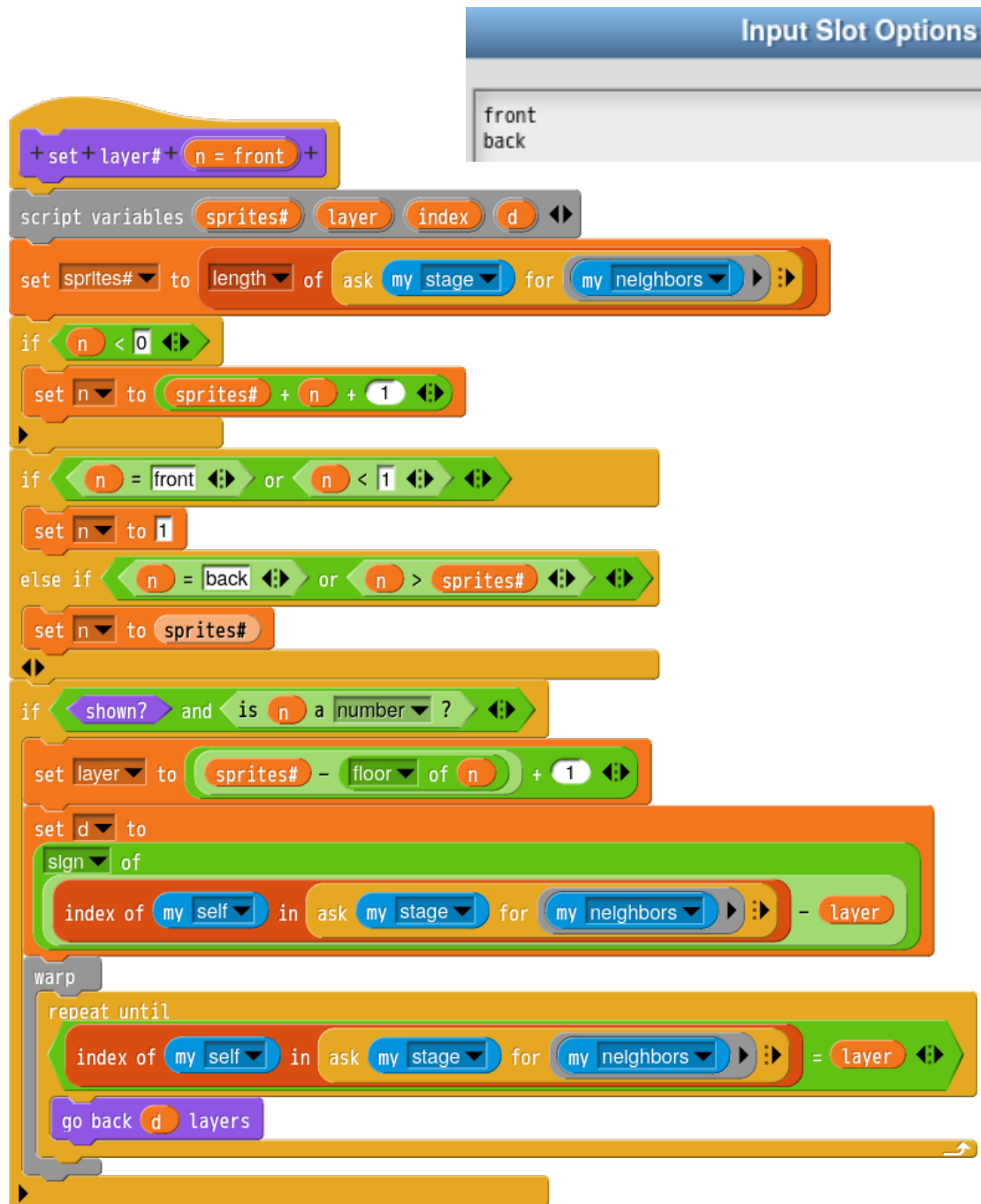


これらを利用すると、スプライトの手前からの位置を指定してレイヤー位置を設定する定義ができます。

5 のスプライトを手前から 4 番目のレイヤー位置に変更します。(5 のスプライトのスクリプトエリアで実行します)



定義です。変数 n のオプションに front と back を設定しています。



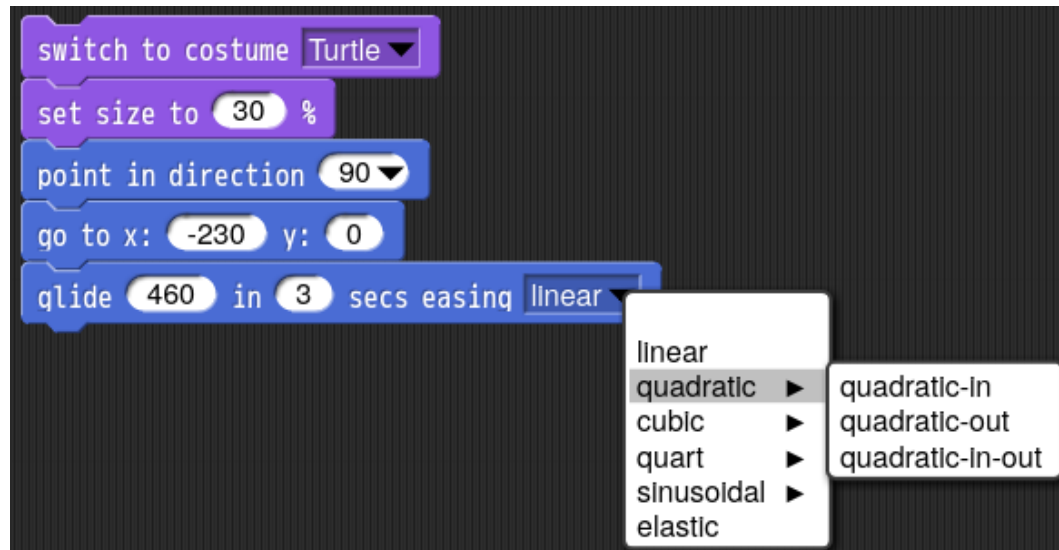
負の数を指定すると、後ろから数えた位置になります。-2 は後ろから二番目です。範囲外の値ならば、1 よりも小さければ front、スプライト数よりも大きければ back として扱います。

index と layer の差から `go back 3 layers` のようにセットすることもできますが、`go to front layer` の定義を真似て 1 段階ずつにしています。こうすると、位置のチェックをするため確実です。ただし、この定義はスプライトが表示されている状態でしか機能しません。

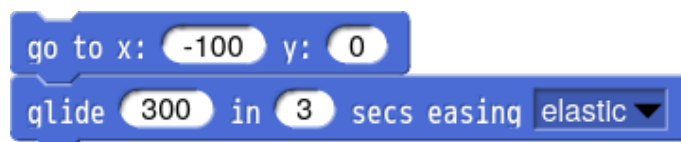
8.19 Animation

Animation (animation_module.xml) ライブラリーをインポートすると、スプライトを動的に変化させるブロックが使えます。

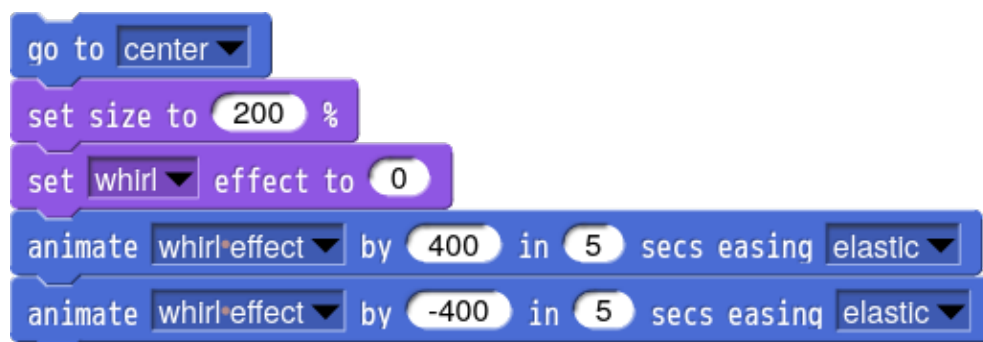
Animation の glide は移動距離と実行秒数、easing の機能で動き方を指定することができます。linear は普通の移動です。quadratic, cubic, quart はそれぞれ、経過時間に対して 2 乗, 3 乗, 4 乗の計算式を使って移動スピードを変化させます。次のスクリプトを実行するとスピードの変化が分かります。-in (徐々に加速する), -out (減速して終了), -in-out (徐々に加速して減速して終了) のオプションがあります。sinusoidal は sin を計算式に使用します。quadratic よりも加速はゆっくりです。



elastic は、一、二、三と反動を使って動き始めるような動きです。移動範囲の関係で上のスクリプトの go to 以下を次のように変更する必要があります。



animate を使うと、いろいろな項目をその時点の値から指定した変化量になるまで指定した時間内で easing のオプションにより変化させることができます。



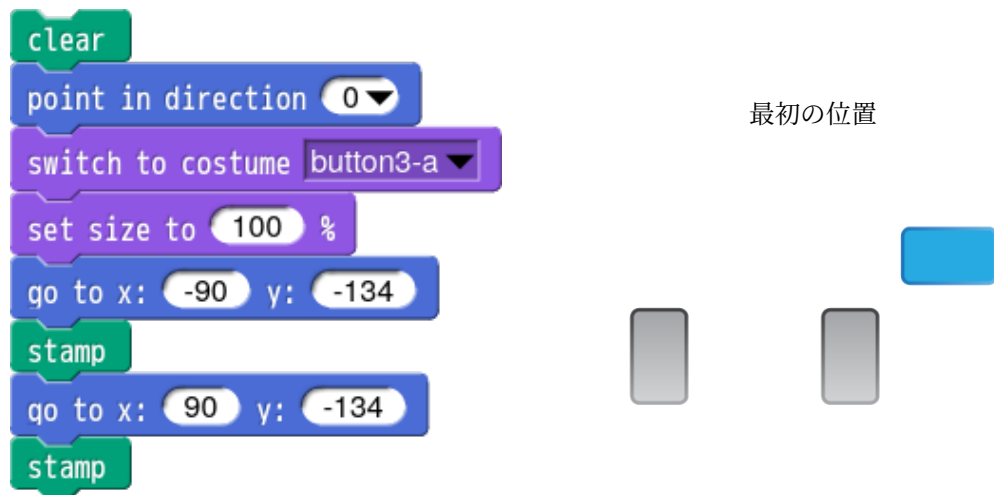
Animation の for ブロックを使っても同じようなことができます。easing の指定で関係するのは指定された秒数と Upvar 変数 i の値です。

```
go to center
set size to 200 %
for i = 0 to 400
  in 5 secs easing elastic
  set whirl effect to i
for i = 400 to 0
  in 5 secs easing elastic
  set whirl effect to i
```

```
set size to 100 %
set color effect to 0
point in direction 90
for i = -200 to 230
  in 5 secs easing quart-In-out
  set color effect to i × 0.1
  go to x: i y: 0
```

```
set size to 150 %
set color effect to 0
go to x: 0 y: 0
for i = 0 to 360
  in 3 secs easing quart-In-out
  set color effect to i × 0.2
  point in direction i
```

駐車シミュレーションをしてみます。上から見た車のコスチュームがないので button3-a b で代用します。まず、両側の車をセットします。



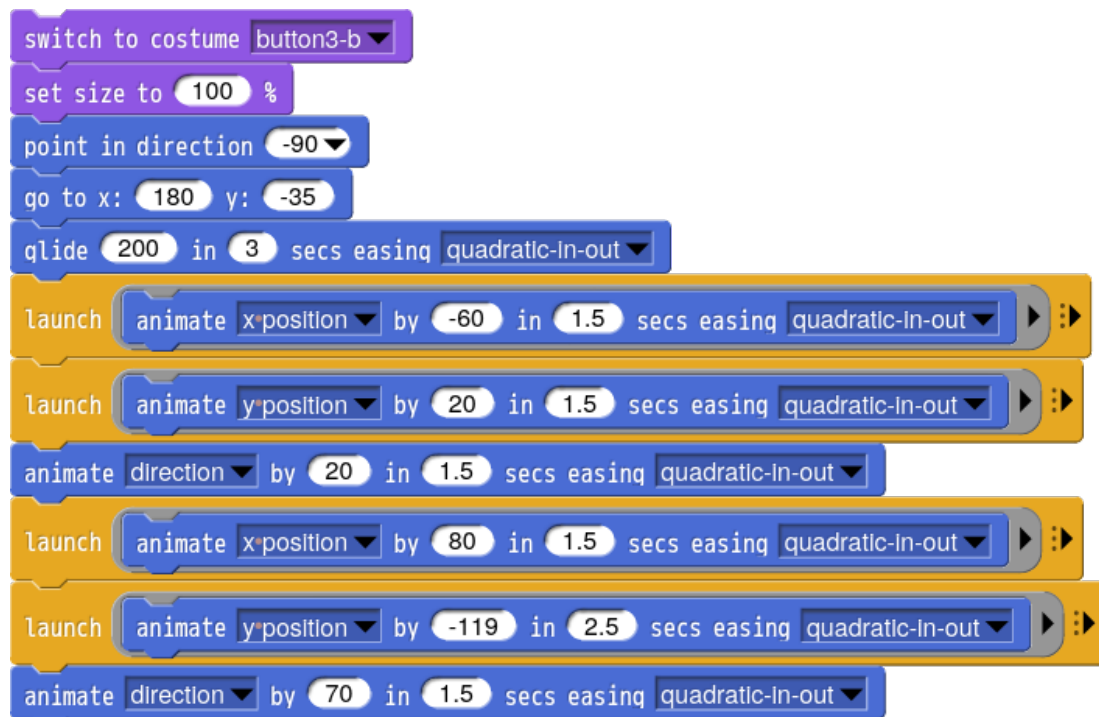
右に回転しながら位置決め



駐車完了



launch を使って位置と向きの変化を同時進行させています。



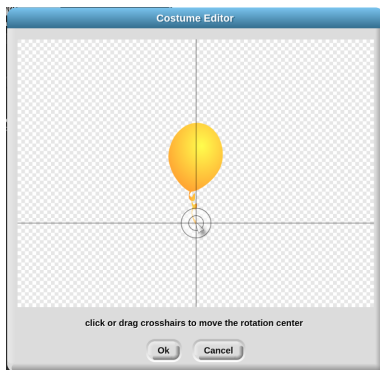
8.20 anchor アンカー

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



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

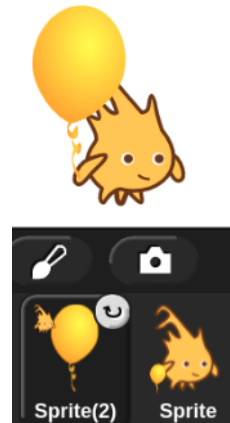
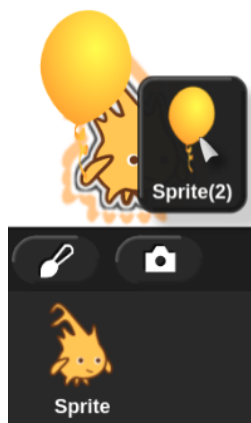
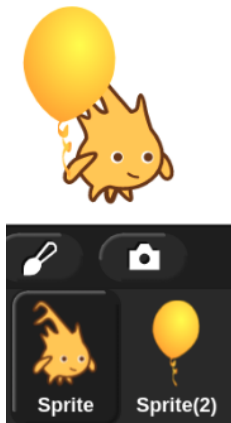
ステージ上で alonzo の手に風船を持たせてください。



balloon をエディターで開いて、動きや回転の中心を糸の端にします。

alonzo 側の中心位置変更の必要はありません。

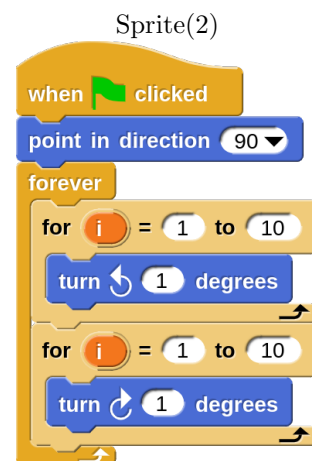
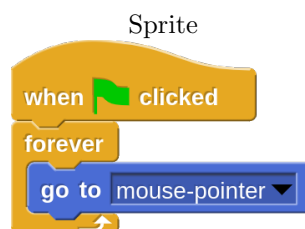
アンカー操作には二つのやり方があります。コラールにある風船のスプライトをステージ上の alonzo のスプライトのところに持っていくとハロがでます。ドロップするとコラールのスプライトの表示が変化します。

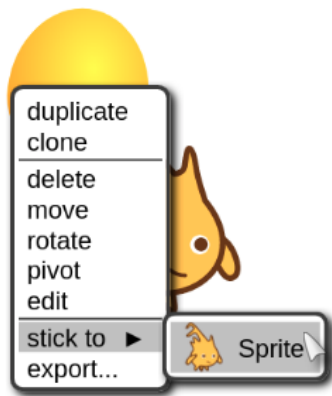


テストスクリプトです。

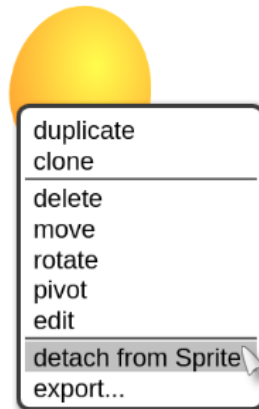
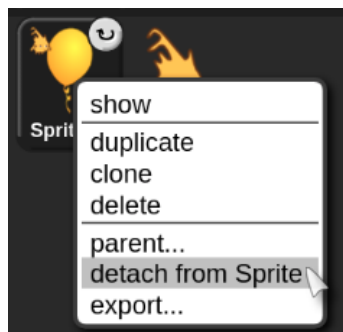
alonzo は動く風船を持ちながらマウスポインターに付いて行きます。

alonzo のサイズを変更すると、連動して風船のサイズも変更されます。





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

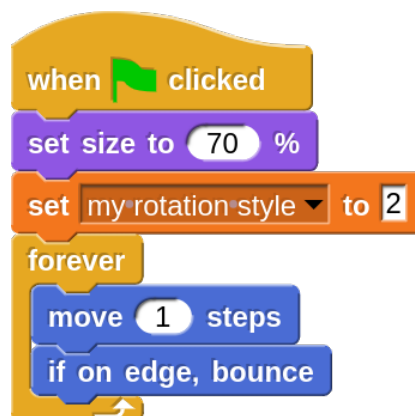


アンカーを解除するには、ステージ上かコラールにある風船のスプライトを右クリックして、オプションから detach from Sprite を選択します。

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

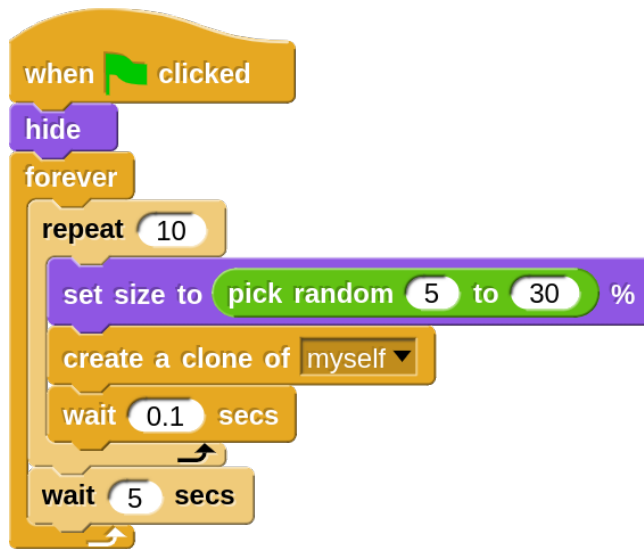


ダイバーのスクリプトです。

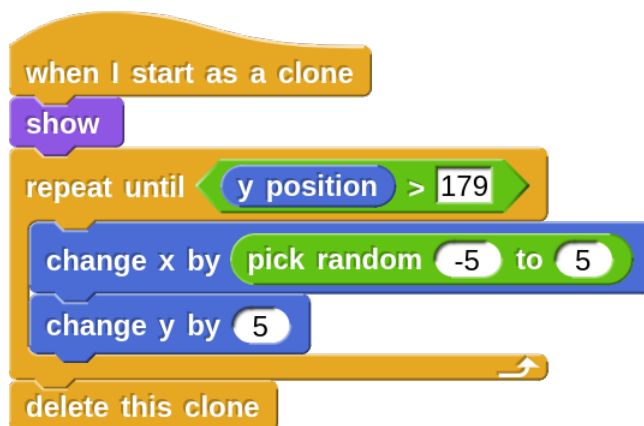


ボールのスク립トです。ボール本体はダイバーに付帯していますが、クローンはその座標を起点にして浮上していきます。 **go to** **ダイバー** を使ってもできることではありますが。

サイズを変化させて、気泡っぽく見せています。



x 座標をランダムに変化させて揺れながら上昇する感じを出します。

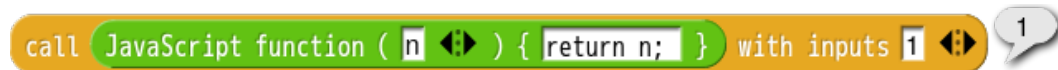


8.21 JavaScript function (オプション 5 ページ参照)

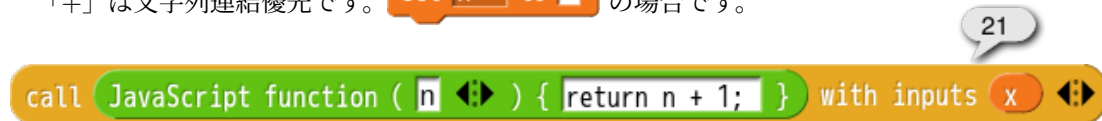
Snap! には、JavaScript コードを実行させるブロックがあります。次の例は JavaScript 側から数値を返すものです。



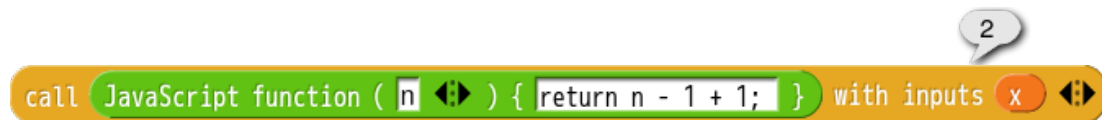
call の入力スロットに入れたものが対応する JavaScript の入力スロットにセットされて JavaScript 側からアクセスできるようになります。



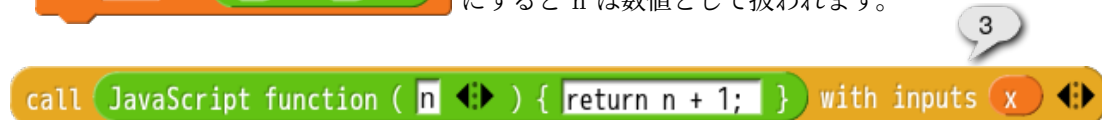
「+」は文字列連結優先です。set x to 2 の場合です。



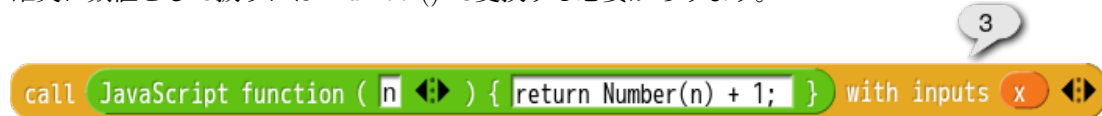
「n - 1」は算術演算されて数値 1 になり数値同士の算術演算により 1 + 1 -> 2 となります。



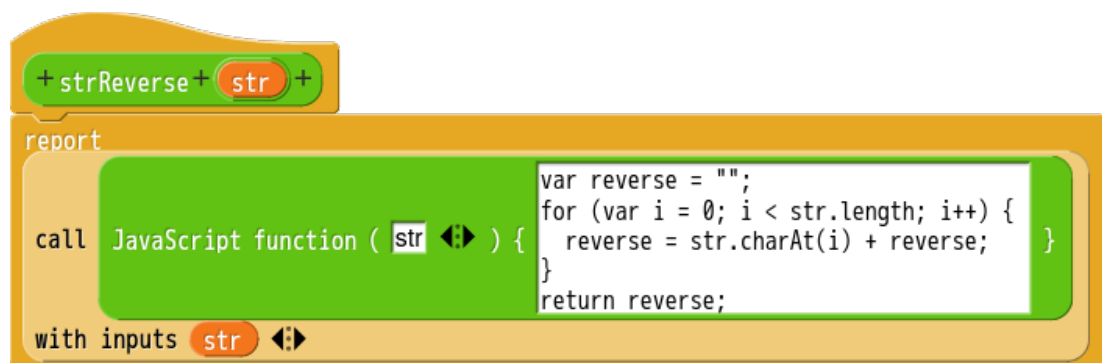
set x to 1 + 1 にすると n は数値として扱われます。



call の with inputs スロットにはどのような値を持つ変数が入力されるか分かりません。それを確実に数値として扱うには Number() で変換する必要があります。



リバースした文字列を返します。str.length は文字数を返します。JavaScript のインデックスは 0 から始まるので length - 1 が最後尾の位置になります。str.charAt(i) は i 番目の文字を返します。



strReverse abc123

321cba

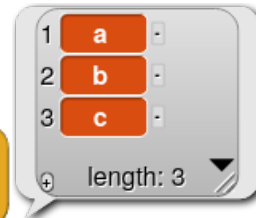
次のように Snap! 側では JavaScript の配列を扱うことはできないようです。

call JavaScript function () { return [1,2,3]; }

«1»

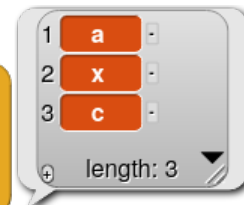
受け取ったリストをただ返すだけならば問題ありません。

call JavaScript function (list) { return list; }
with inputs list a b c



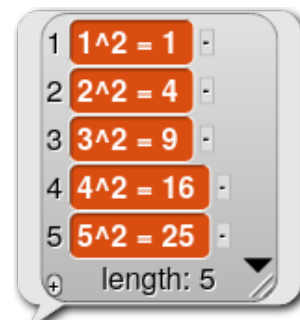
Snap! のリストを JavaScript 側で操作するには `asArray()` で配列に変換した変数に対して操作する必要があります。操作の結果はリストに反映されます。必ず入力スロットにセットされたリスト変数を返してください。この場合、1ではなく `list` です。配列の先頭のインデックスは 0 です。

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

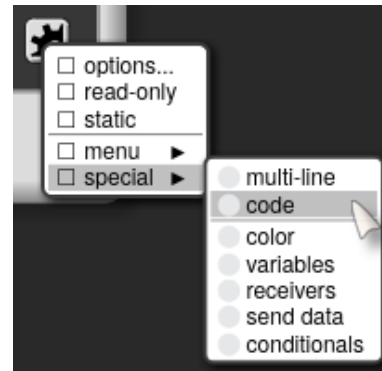


次の例は、JavaScript 側から 1 から 5 の二乗のリストを返すものです。

JavaScript 側で作成した配列を Snap! 側に返すには必ずリストとして返さなければならないので、データとしては必要なくても Snap! 側から JavaScript 側にリストを引数として渡す必要があります。空の配列に `push` で要素を追加して配列を作成するのですが、空の配列を用意するために `l = [];` としてしまうとリストへの参照がなくなってしまいます。list とは別物になり、1 への操作が list に反映されなくなります。Snap! 側から空のリストを渡すことにはリストという入れ物の受け渡しとリスト（配列）の初期化という二つの重要な意味があります。



call JavaScript function (list) {
var l = list.asArray();
for (var i = 1; i <= 5; i++) {
l.push(i + "^2 = " + i * i);
}
return list;
}
with inputs list



入力スロットのオプションには code があります。

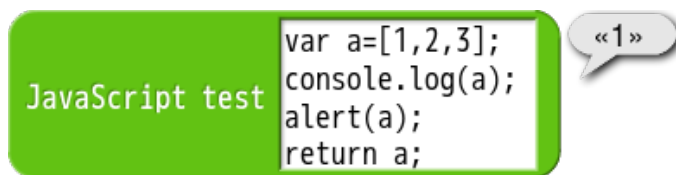
これを利用すると複数行のテキストが入力できるので JavaScript の練習用ブロックが作れます。



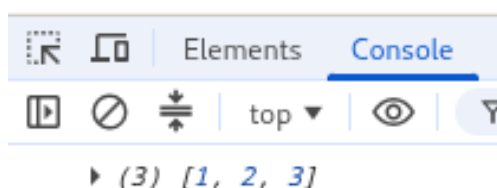
結果は console.log() や alert() で表示するのと return で値を返す方法がとれます。

Snap! では JavaScript の配列を扱えないので、表示するには console.log() か alert() を使用する必要があります。以下に示すように Snap! ではそれぞれ配列の表示の仕方が違ってきます。

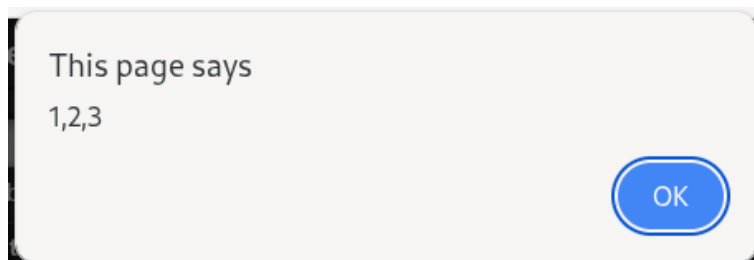
実際には最後に値が表示されるスクリプトです。



console.log() による表示です。web ブラウザーの開発者ツールのコンソールを表示させておく必要があります。

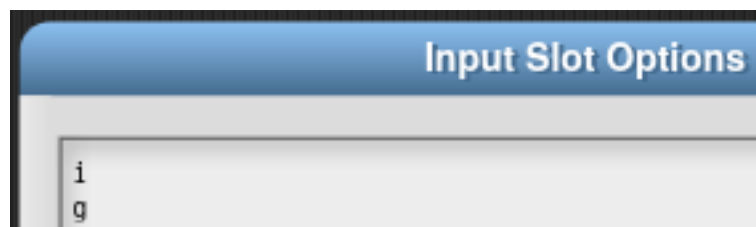
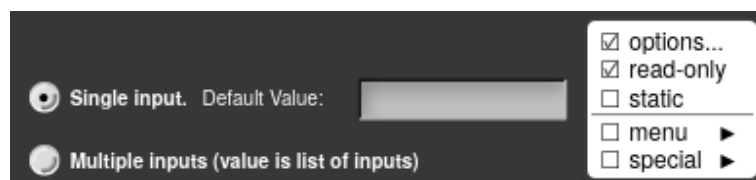


alert() による表示です。次へ進むには OK ボタンをクリックします。

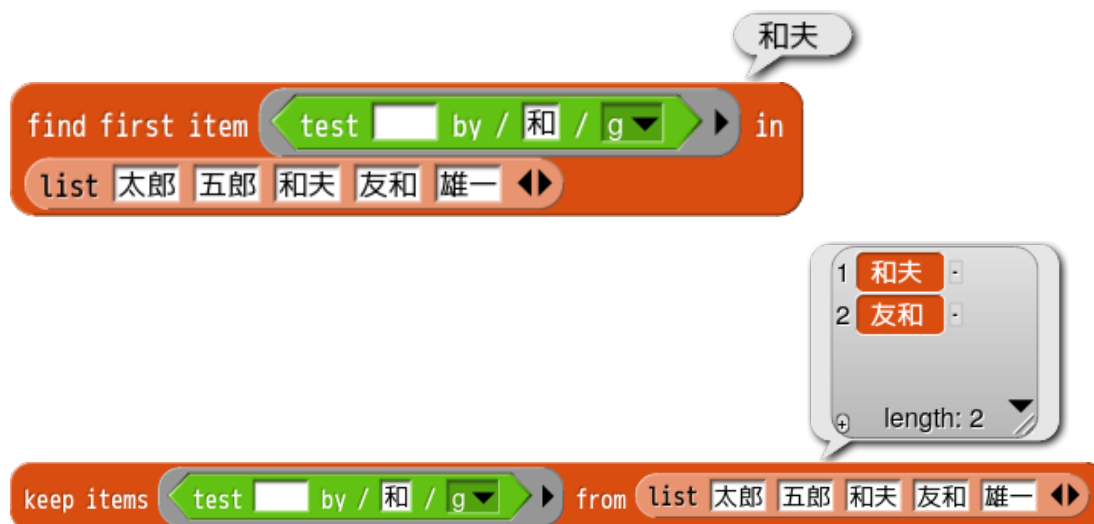


文字列操作で威力を発揮するのが正規表現です。Snap! でも JavaScript の正規表現を利用することができます。例えば郵便番号を正規表現で表すと、`/^\d{3}-\d{4}$/` になります。「先頭 (^) から数字 (\d) が三桁 ({3})、ハイフン、数字が四桁で末尾 (\$) のパターンをスラッシュで囲んでいます。ここにメソッドによって、フラグで機能指定をすることができます。Snap! 側からスラッシュ内部のパターンと機能指定を送ることでメソッドを利用することができます。

パターンと一致するかをテストするブロックです。RegExp() で正規表現パターンに変換しています。test() の機能指定は flag で行いますが、このブロックで使用できるのは i(ignore 大文字小文字の区別をしない)、g(global) にしています。g は本来複数ヶ所のマッチを可能にするものですが、find で使用する際に入力スロットを空にしない目的で使用します。flag は read-only、options... (i, g) で設定しています。Default value は無指定です。



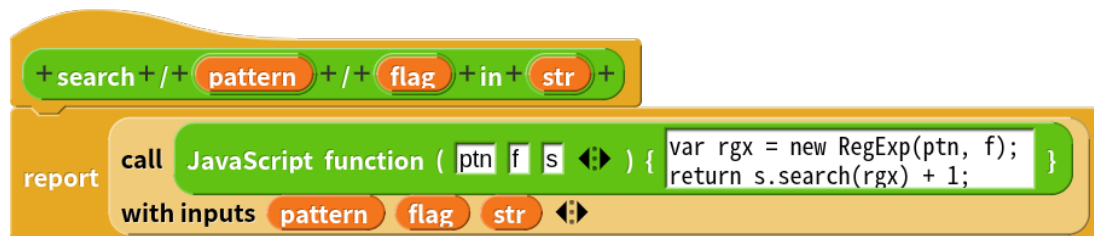
find などの条件指定に test を使用するのも有益です。ただし、flag が無指定の場合はリングから value を取り出して str にセットしなければならなくなるので、必要のない g 機能を指定しています。



map で使用すると true か false のリストになるのでそこからインデックスを求められます。



str の中から指定された pattern を探してマッチした位置を返す search です。ない場合は 0 を返します。pattern を Any type、str を Text に設定して入力スロットの大きさを変えています。flag は read-only、options... (i) で設定しています。Default value は無指定です。



i flag の有無により大文字小文字を区別するか指定します。



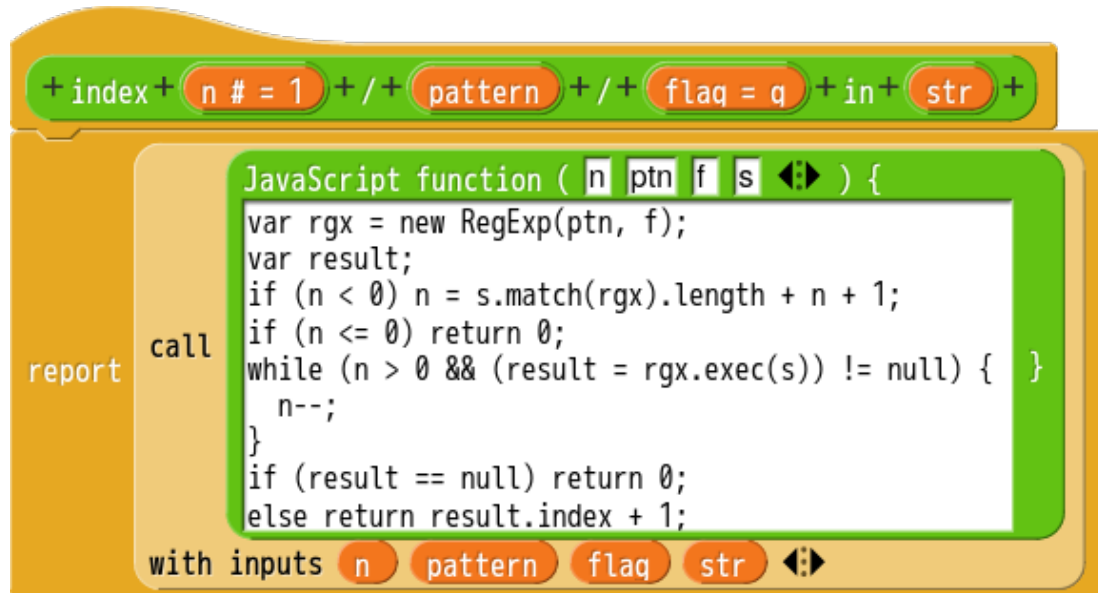
次のようにすると、末尾にないなら 0、末尾なら位置を表示します。



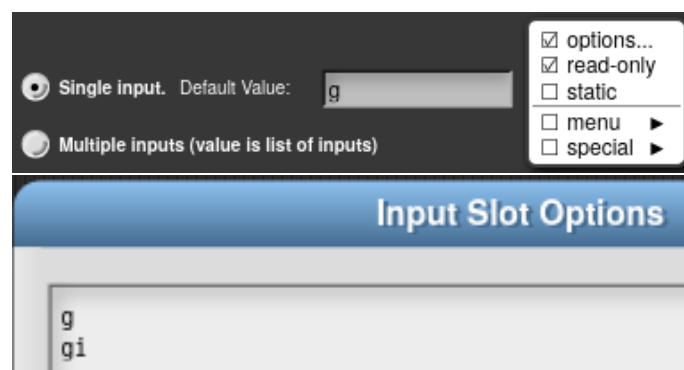
次のようにすると、最後にマッチした位置を表示します。



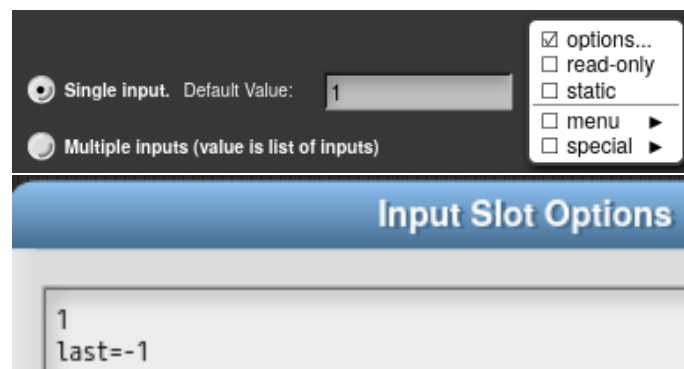
何番目にマッチしたかを指定して位置を求める index です。flag に g は必須です。match() はマッチした要素の配列を返すので、そこからマッチした要素数を求めます。g flag 付きの exec() は継続検索を実行します。



flag の設定です。
read-only, options...(g, gi)
Default は g です。



n の設定です。
数値入力可なので read-only で
はありません。
Default が 1 で、負数で最後か
らのカウントになります。-1 で
最後にマッチしたものになりま
す。options...(1, last=-1) になり
ます。



index 2 / ab / g in 1abc5abd9ab 6

index -1 / ab / g in 1abc5abd9ab 10

index -1 / \d / g in 1abc5abd9ab 9

replace です。str の中で pattern とマッチしたものを word で入れ替えます。マッチしない場合は str をそのまま返します。flag は read-only、options... (g, gi) で設定しています。Default value は g です。g はマッチした複数箇所すべての要素の入れ替えを指定します。g が指定されないと最初にマッチした一箇所だけになります。i が指定されると大文字小文字の区別なく変更します。

+replace+/+ pattern /+ flag = g +with+ word +in+ str +

report

call JavaScript function (ptn f wd s) { var rgx = new RegExp(ptn, f); return s.replace(rgx, wd); }

with inputs pattern flag word str

replace / ab / g with xy in 1ab2AB3 1xy2AB3

replace / ab / g| with xy in 1ab2AB3 1xy2xy3

正規表現「\d」は数字を表し、「+」で 1 個以上の指定になります。

replace / \d+ / g with N in 1a2b345c NaNbNc

正規表現「\s」は空白を表します。半角全角どちらの空白にも対応します。

replace / \s+ / g with in 1 a2b 1a2b

正規表現で指定して「-」の左右の要素を交換します。かっこで囲むとグループ化されて、replace ではマッチした要素が左から \$1、\$2、\$3、... で参照できます。「.」は、改行以外の一文字を表します。「*」で 0 個以上の指定になります。

replace / (.*)-(.*?) / with \$2-\$1 in 一二三四

JavaScript には、substring や slice という部分文字列を抽出するメソッドがあります。これを使って文字列の何番目から何文字を取り出すブロックを定義してみます。JavaScript では文字のインデックスは 0 から始まります。

```

+ str + .substring(from# + from # + ,length# + length # + ) +
report
call
JavaScript function ( text from len ) { return text.substring(from-1,from-1+len); }
with inputs str from length

```

半角文字も全角文字も正しくカウントされます。

12345一二三四五 .substring(from# 4 ,length# 4)

replace ブロックを使って（部分文字列）をマッチさせて取り出すこともできます。

「.{n}」で改行以外の一文字を n 個の意味になります。

```

+ subStr+RegExp+ str + from # + length # +
block variables l
set from to from - 1
set l to length of text str - from + length
report replace / join ^{ from } (.*).{ l }$ / with $1 in str

```

subStr RegExp 12345一二三四五 4 4

match メソッドを使うと、正規表現を使って取り出した要素をリストで返してくれます。

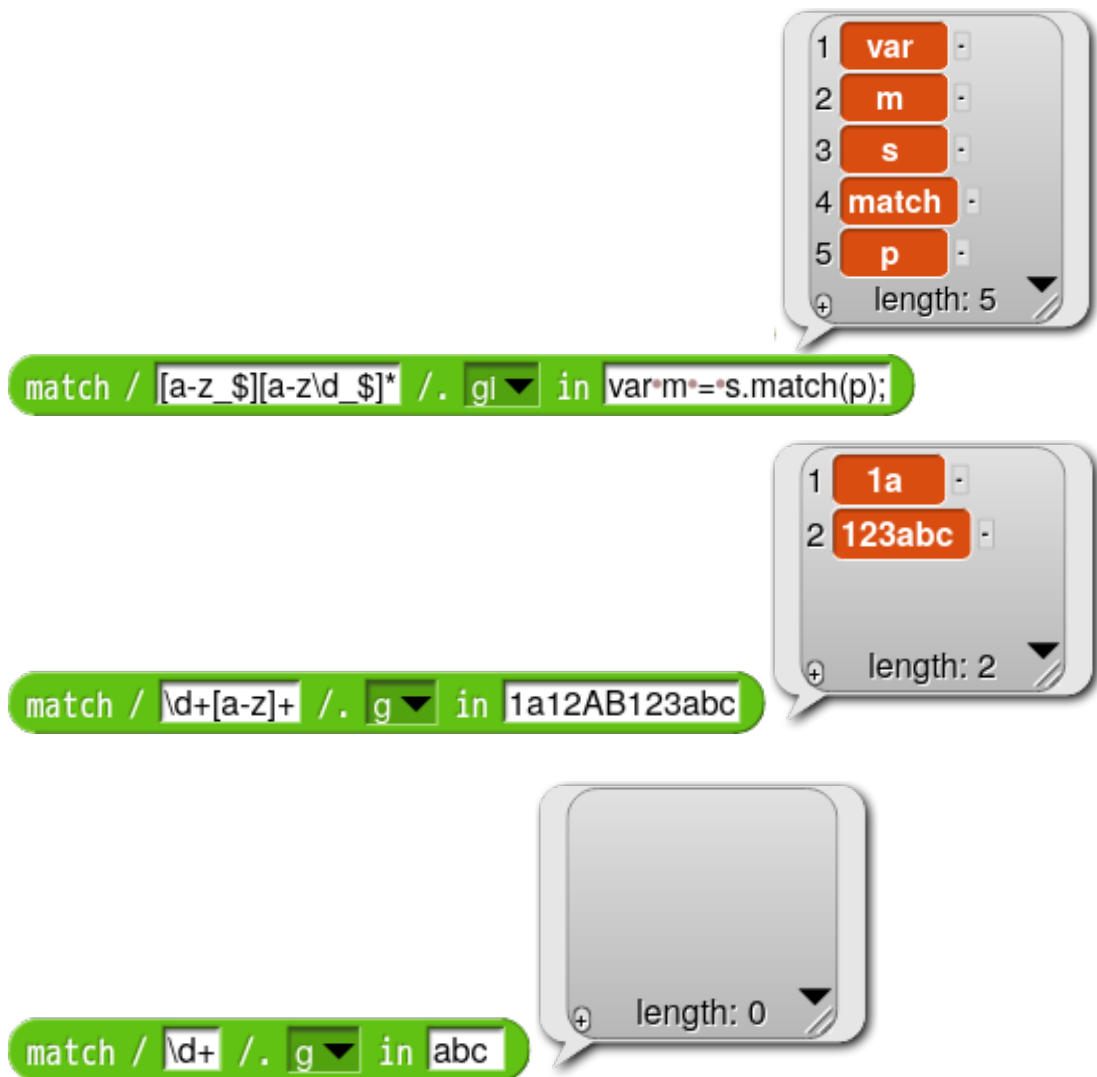


A Scratch-style block for the JavaScript `match` function. The block has a header with labels: `+match+/+`, `pattern`, `+/.`, `flag`, `+in+`, and `str`. Below the header is a `report call` button. The main body is a green box containing the following JavaScript code:

```
JavaScript function ( ptn f s l ) {  
    var rgx = new RegExp(ptn, f);  
    var ml = l.asArray();  
    var m = s.match(rgx);  
    if (m != null) {  
        for (var i = 0; i < m.length; i++) { }  
        ml.push(m[i]);  
    }  
    return l;  
}
```

At the bottom, there is a `with inputs` section with labels: `pattern`, `flag`, `str`, and `list`.

flag は read-only、options... (g, gi) で設定しています。Default value は g です。
変数名を抽出します。(予約語にもマッチします。)



Three examples of the `match` function block in use:

- Example 1:** The block is configured with `match / [a-z_$][a-z\d_$]* /. gl` in `var m = s.match(p);`. The output list shows 5 items: `var`, `m`, `s`, `match`, and `p`. The length is 5.
- Example 2:** The block is configured with `match / \d+[a-z]+ /. g` in `1a12AB123abc`. The output list shows 2 items: `1a` and `123abc`. The length is 2.
- Example 3:** The block is configured with `match / \d+ /. g` in `abc`. The output list is empty, and the length is 0.

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

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

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

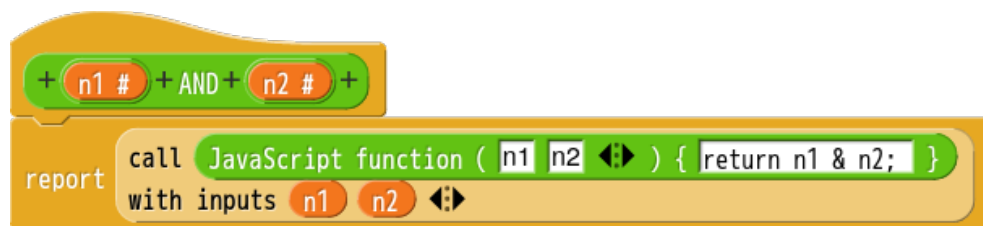
主要なビット演算として、ビット反転 (NOT)、論理積 (AND)、論理和 (OR)、排他的論理和 (XOR) があります。2 つの 1 ビットの数 n1, n2 が取り得る値 0, 1 に対してのそれぞれのビット演算の結果を示します。

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

論理積 (AND) は両方の値が 1 の時だけ 1、
論理和 (OR) は片方だけでも 1 ならば 1、
排他的論理和 (XOR) は双方の値が異なる時だけ 1 になります。

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

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



1 AND 0 → 0, 1 AND 1 → 1, 1 OR 0 → 1, 1 XOR 0 → 1. "&" の部分を "|" にすれば OR 演算, "^" にすれば XOR 演算になります。

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



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

ります。つまり、1111 は 0001 に加えると 0000 になる数である十進数の -1 ということになります。ニプルで表せる符号付き数は 1000 ～ 0111、十進数の -8 ～ 7 になります。

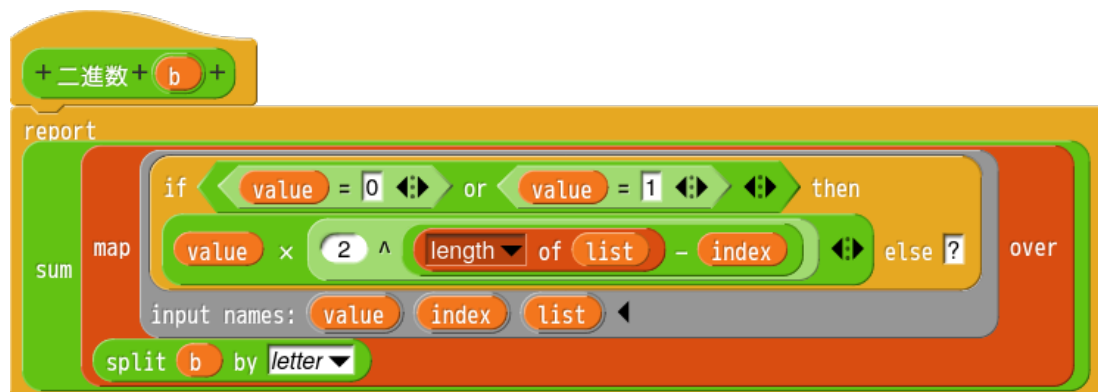
0001 + 1111 = 0000 のような桁上がりの有無を機械語ではキャリーとして利用します。

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



二進数で数値を指定するリポーターブロックを作成してみます。b の入力のタイプは text です。split で二進数値を各桁要素のリストにしていますが、letter all of b でも可能です。

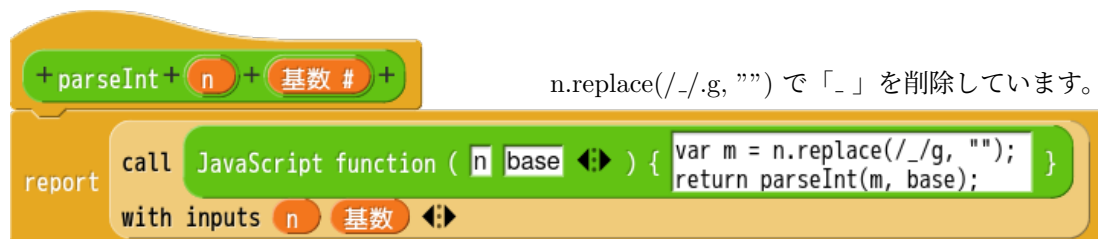
二進数 1100 の場合、 $1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$ なので、それぞれの桁ごとにエラーチェックをしながら map で十進数に変換して、得られたリストを sum で合計します。



二進数では 0 と 1 しか使わないので、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。



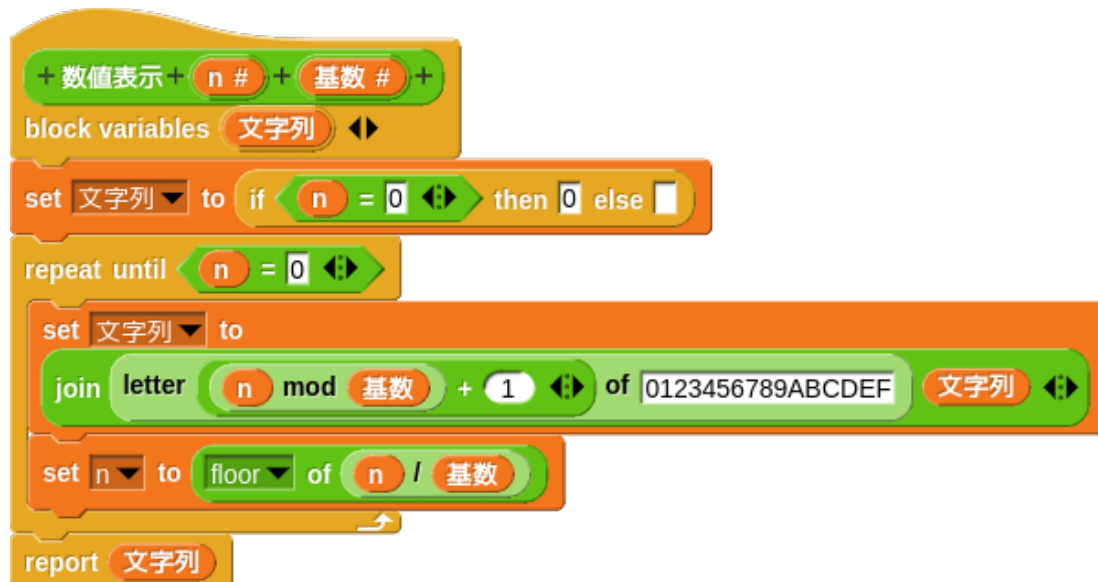
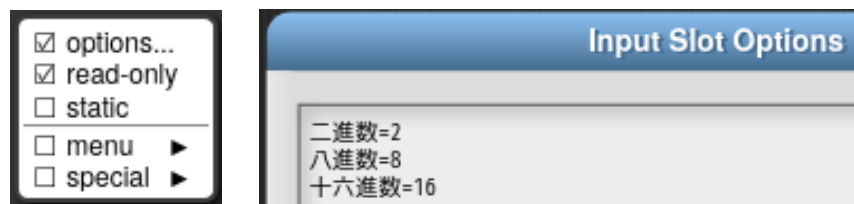
JavaScript には文字列を整数に変換する parseInt 関数があります。2 番目の引数で基数を指定できます。2 で二進数、16 で十六進数の文字列からの変換になります。なお、「base」と「基数」が合っていませんが、JavaScript の引数には名前ではなく inputs のスロット位置に対応した値がセットされます。規定の文字以外（_を除く）が現れた時点で変換を終了します。エラー処理はしません。二進数などで 1000.1010 のように「_」を挿入可にしてみました。



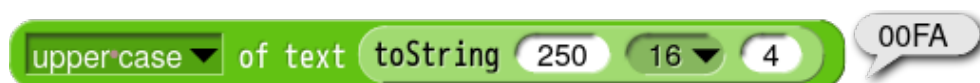
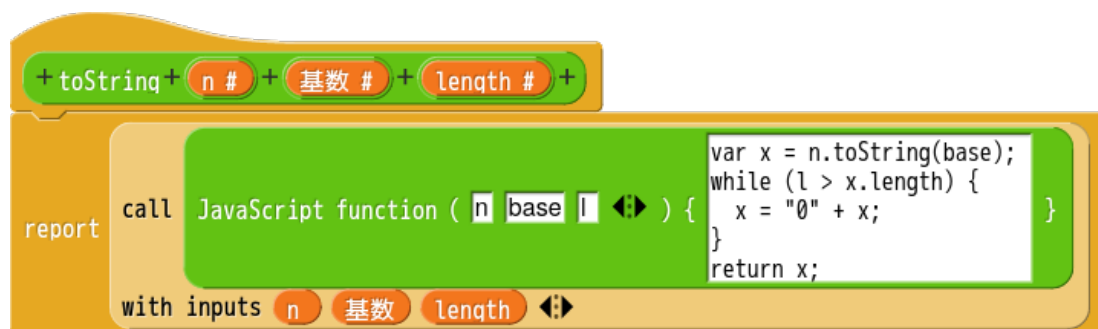
`n.replace(/_/g, "")` で「_」を削除しています。



こうなると、数値を二進数表示するブロックも必要です。どうせなら基数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 二, 八, 十六進数です。
変数「基数」は、read-only、options... (2, 8, 16) を設定しています。



JavaScript では toString がそれをしてくれます。
桁数が指定できて 0 で埋めます。無指定または必要な桁数以下なら無視します。
while では、指定の桁数になるまで "0" を連結することを繰り返します。



AND と XOR の使用例です。

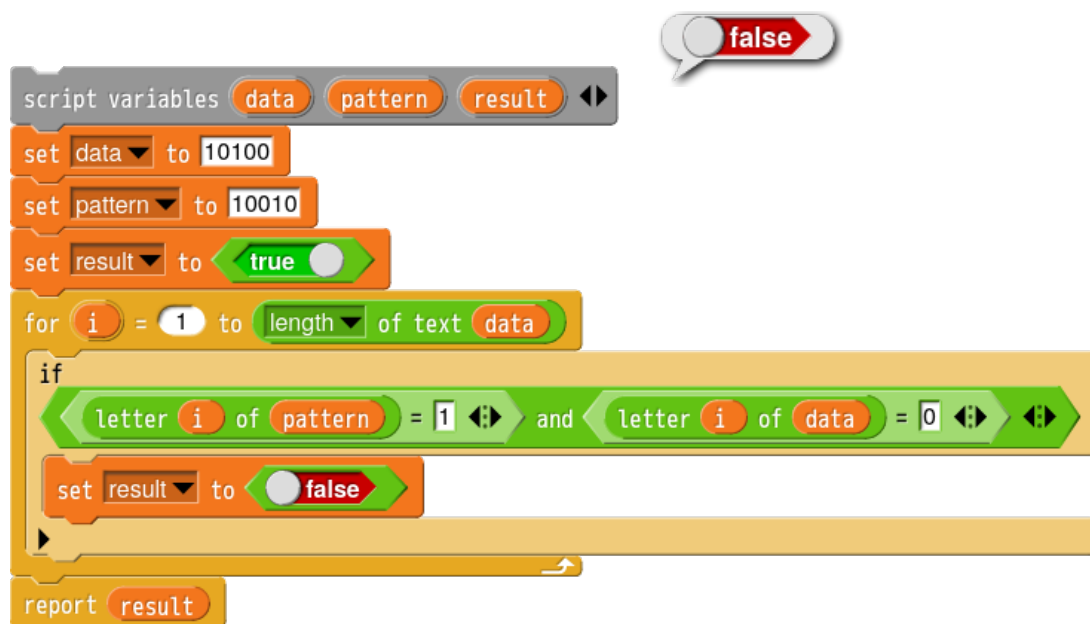
5つの設問に対する回答 ○ と X を 1 と 0 で表したデータのリスト

list 10100 11010 01011 00101 10010 に対して、

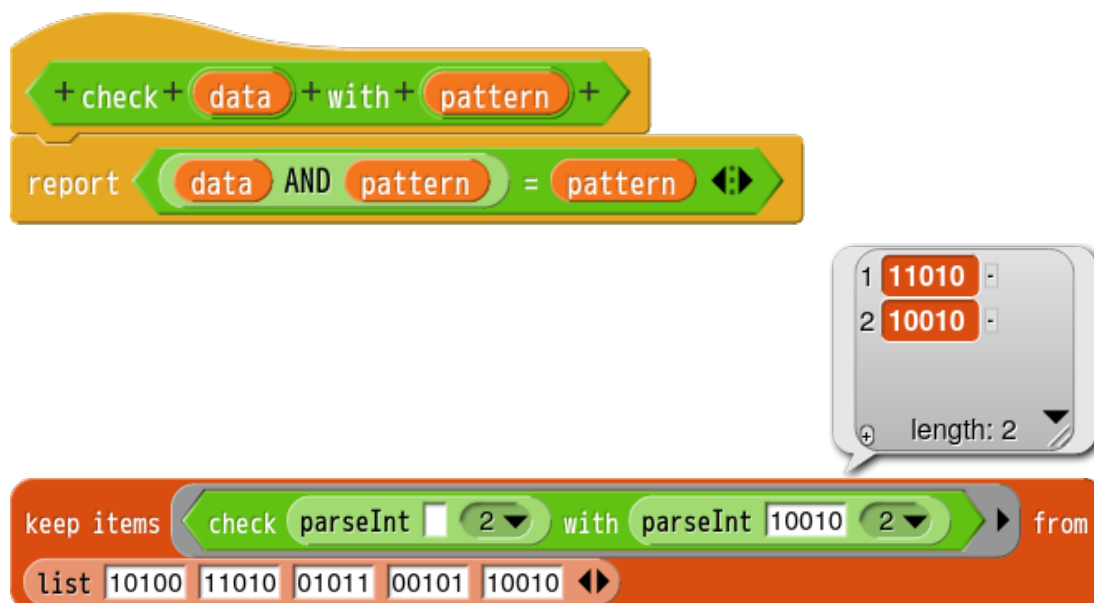
一番目と四番目が 1 のデータを検索するとします。他の位置はどちらでもかまいません。

アンケートなどで、ある項目に関心がある人を把握するような場合です。

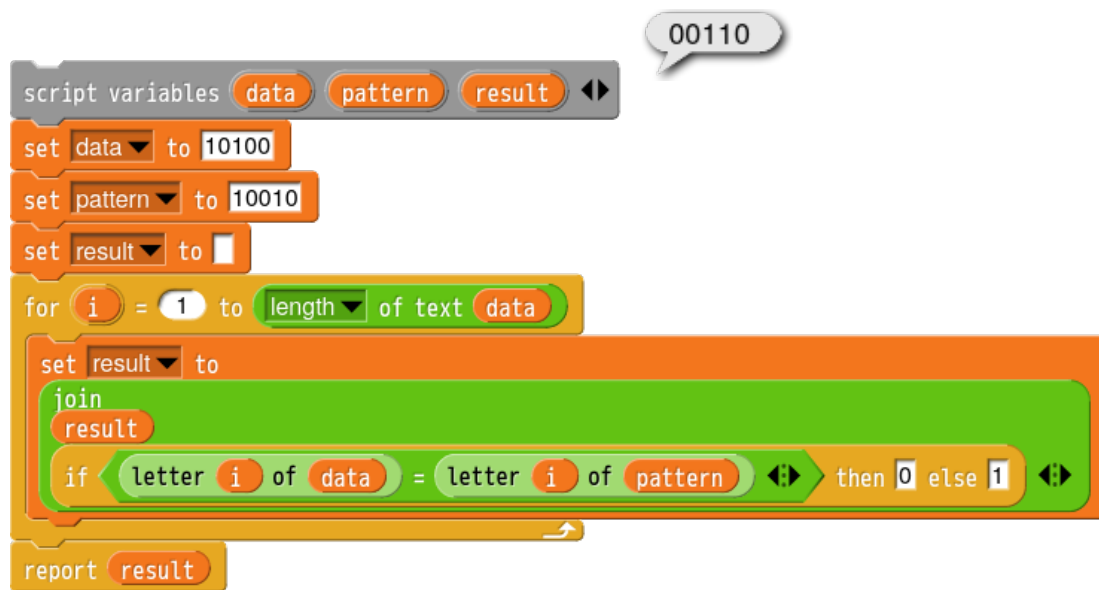
これを for ブロックで作るとこのようになります。一番目と四番目なので 10010 でチェックします。



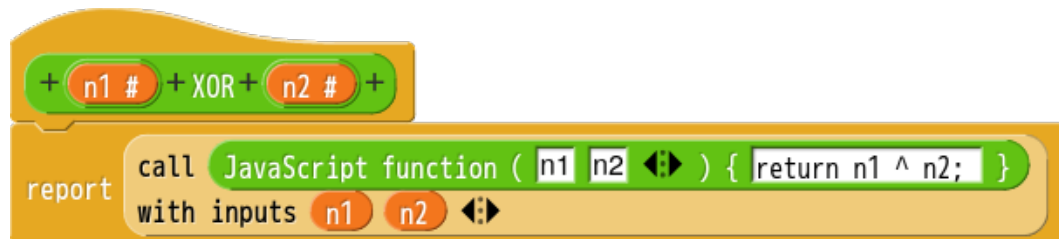
AND を使うと簡単です。AND は特定のビットの状態を部分的に取り出すために使用されます。データを二進数として扱うために parseInt で数値化する必要があります。



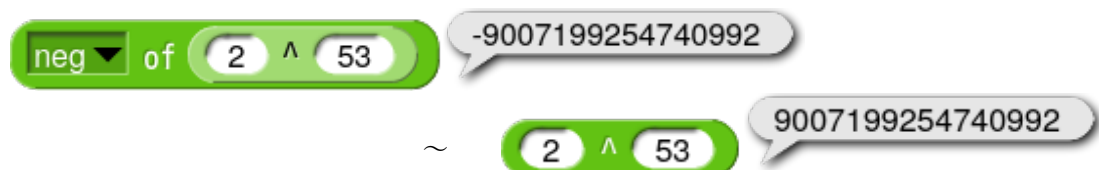
先のデータがテストの解答だとすると、正解のデータ 10010 と違うところ（間違いの箇所）を表示するスクリプトは次のようになります。



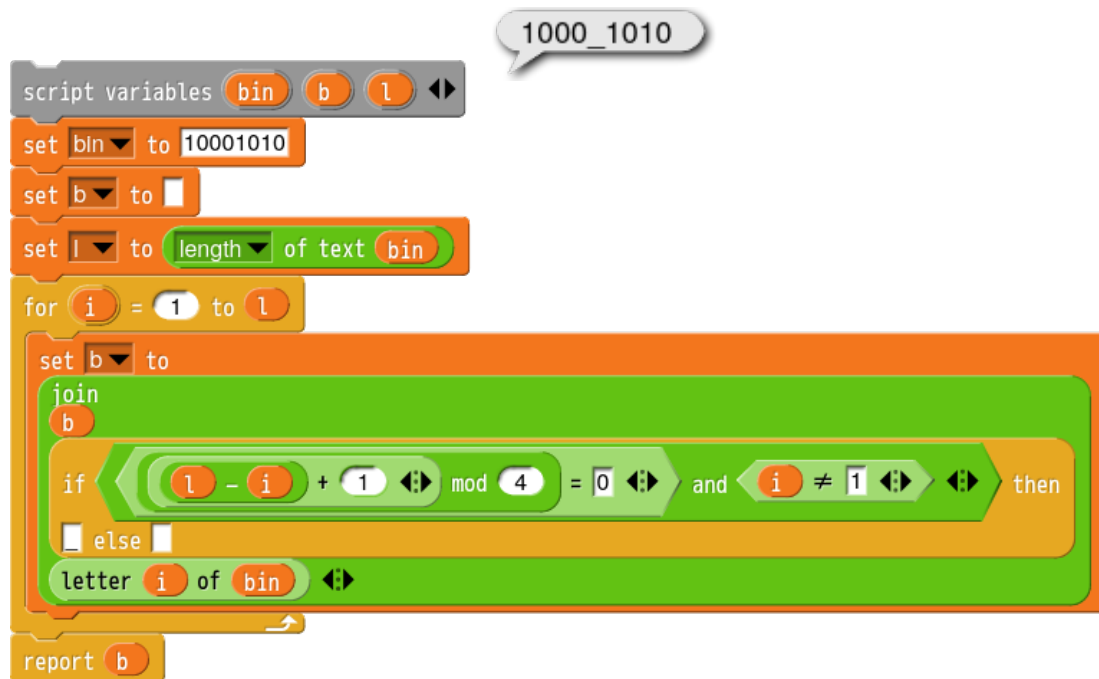
正解のデータ 10010 と XOR を行うことで
間違えたところを 1 で示すことができます。
五番目のデータは全問正解なので 00000 です。
基準のデータとの違いがチェックできます。



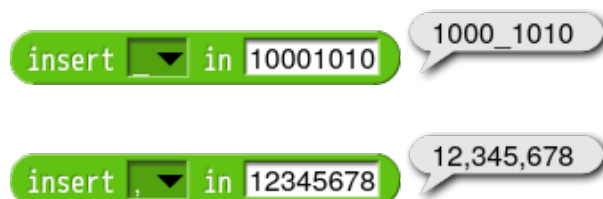
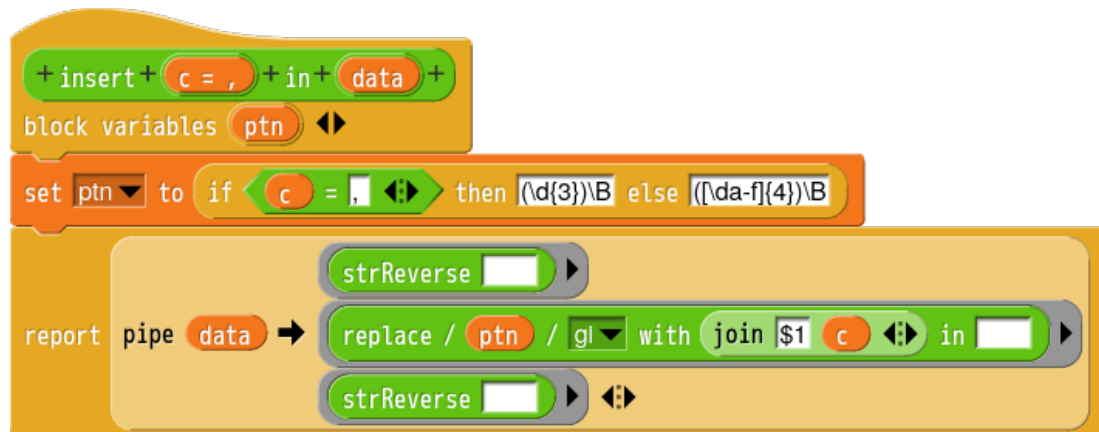
数値は、整数として扱える範囲には限界があります。
JavaScript の数値形式では以下の範囲になるそうです。つまり、AND や XOR などの演算もその
範囲の値に限定されます。



二進数などを四桁ごとに区切る方法です。(1000010001 -> 10_0001_0001)
 mod 4 (4 桁ごと) を利用したスクリプトです。
 mod 3 (3 桁ごと) に変更し、区切り文字をコンマにすることで十進数用になります。

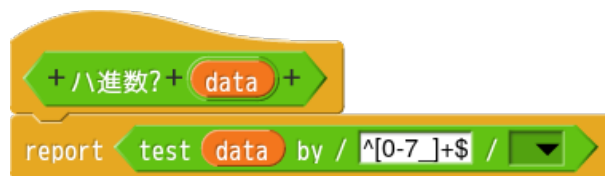


正規表現を使う方法です。十六進数などでも使えます。(strReverse は 135 ページ参照)
 最下位桁側から操作する必要があるので逆順にして処理します。\\B (単語境界ではない位置) 付き
 でマッチさせて末尾に区切り文字がこないようにします。処理後に正順に戻します。
 区切り文字をコンマにすることで十進数用になります。

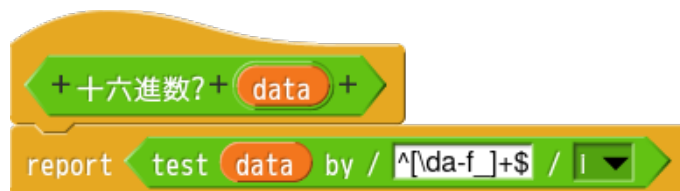


二進数、八進数、十六進数のデータをチェックするブロックです。

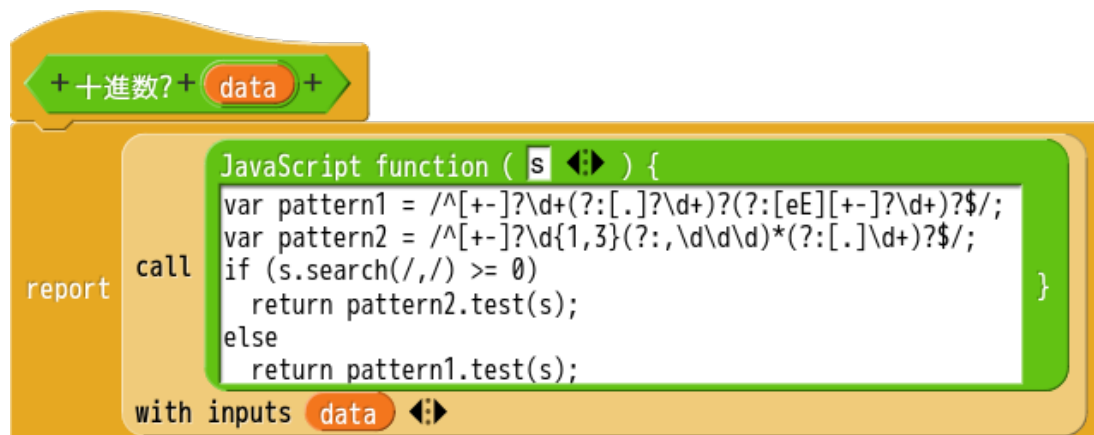
データの先頭を表す「^」と末尾を表す「\$」の間にそれぞれで使える文字が 1 個以上並ぶ構成になります。「[]」のかっこで囲むとその中のどれかの要素が可能であることを示します。単独だと 1 個だけを表しますが、「?」で 0 か 1 個、「*」で 0 個以上、「+」で 1 個以上の指定になります。「0-7a-f」のように「-」でつなぐと、0 ～ 7、a ～ f の連続した文字を指定することになります。「_」は、二進数などを扱う場合に 1000_1111 のように 4 ビットで区切ると読みやすくなるので使えるようにしています。



十六進数では 10 ～ 15 の数値を A ～ F で表しますが、test の flag に i を指定しているので大文字小文字どちらもマッチします。



十進数では小数点や指数表記のものがあ、コンマを使って三桁で区切る場合もあります。それらが正しく表記されているかチェックします。「.123」や「123.」のような表記は false としてみました。



このような二列三行の白丸と黒丸のパターンがある場合、



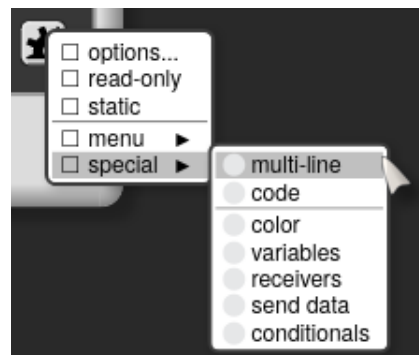
左端のパターンの白丸と黒丸をそれぞれ 0 と 1 で表し、リストにすると次のようになります。



この 0 と 1 の三行のパターンを改行をなくして結合すると、100000 と二進数値とみなすことができます。十進数に変換すると 32 になり、データとしてはリストよりもコンパクトになります。実は、ページ先頭の 5 つのパターンは左から点字の「ア イ ウ エ オ」を表しています。実際には凸化されるのは黒丸の部分だけになります。このようなパターンを数値に変換するスクリプトです。



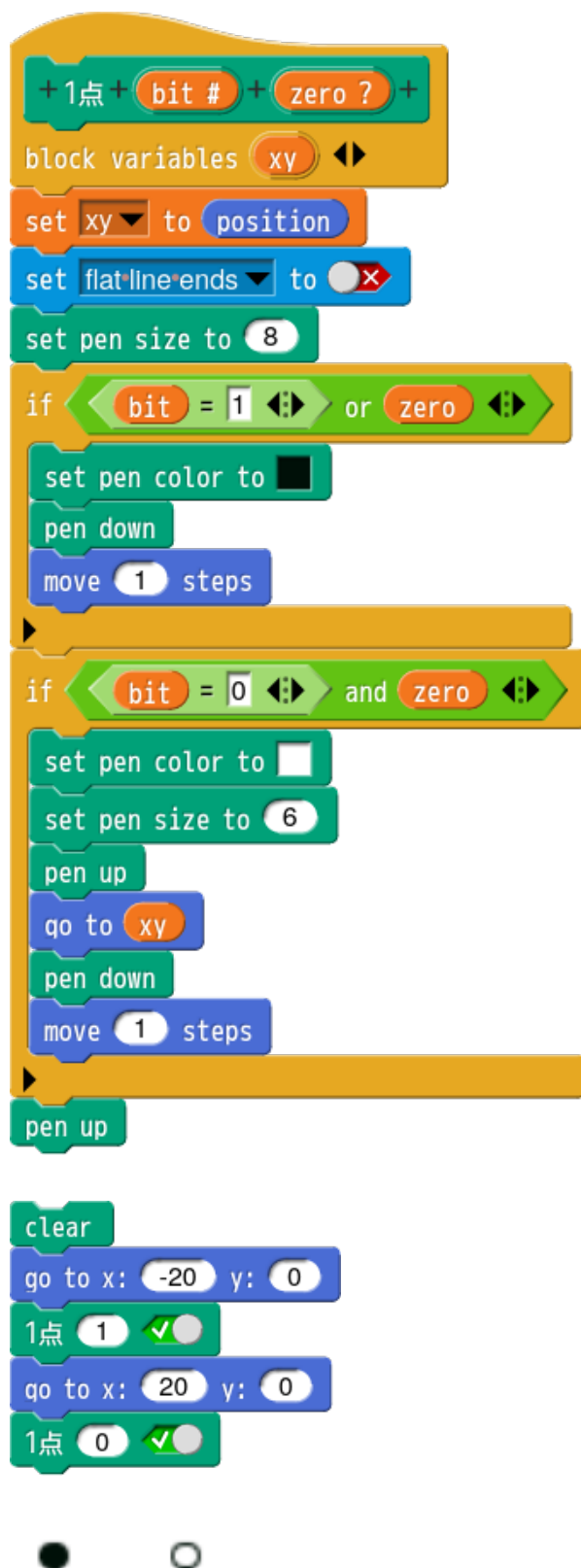
二進数を点の位置に対応させて行分割入力できるように変数 a に special -> multi-line 機能を指定します。



カナ文字とパターンデータの組はこのように表すことができます。



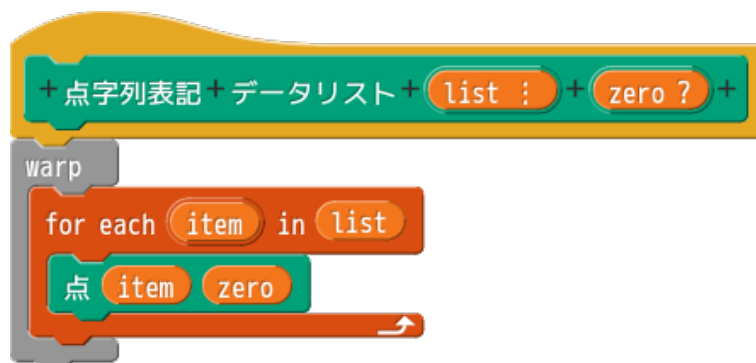
1点を描くスクリプトです。変数 zero を true オンにすることによって白の点を表示することができます。true を Default にしています。



点字一文字を描くスクリプトです。



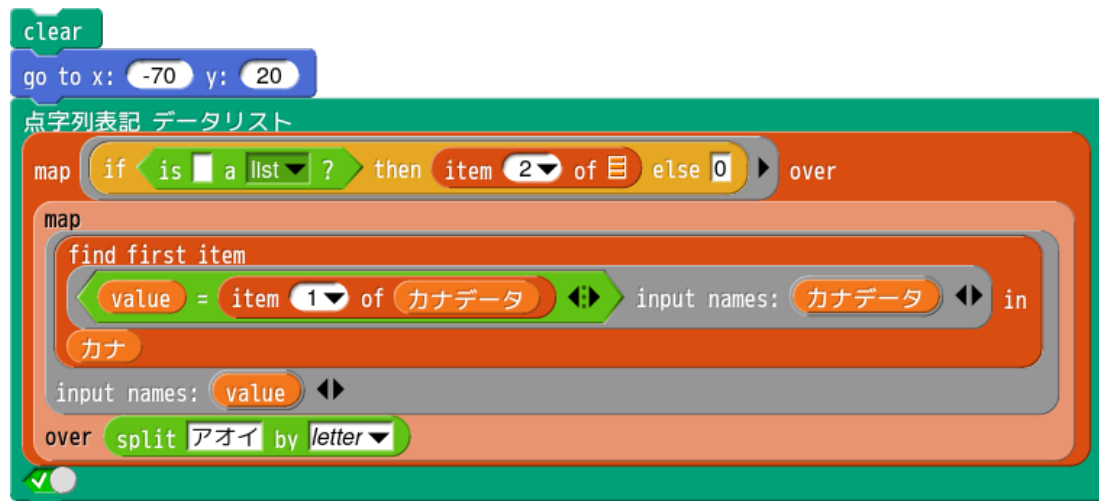
データのリストを点字で表示するスクリプトです。



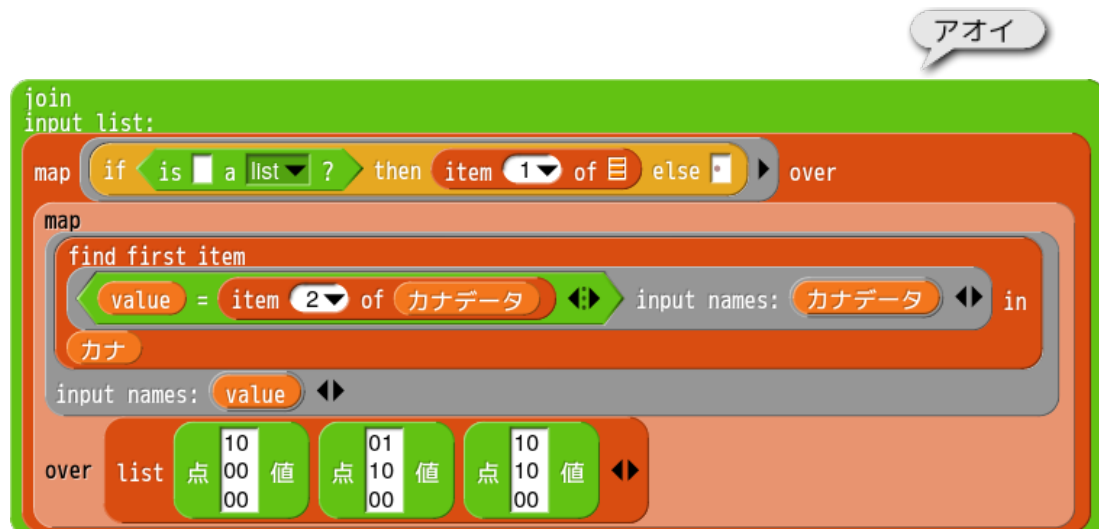
数値データで指定する場合は。



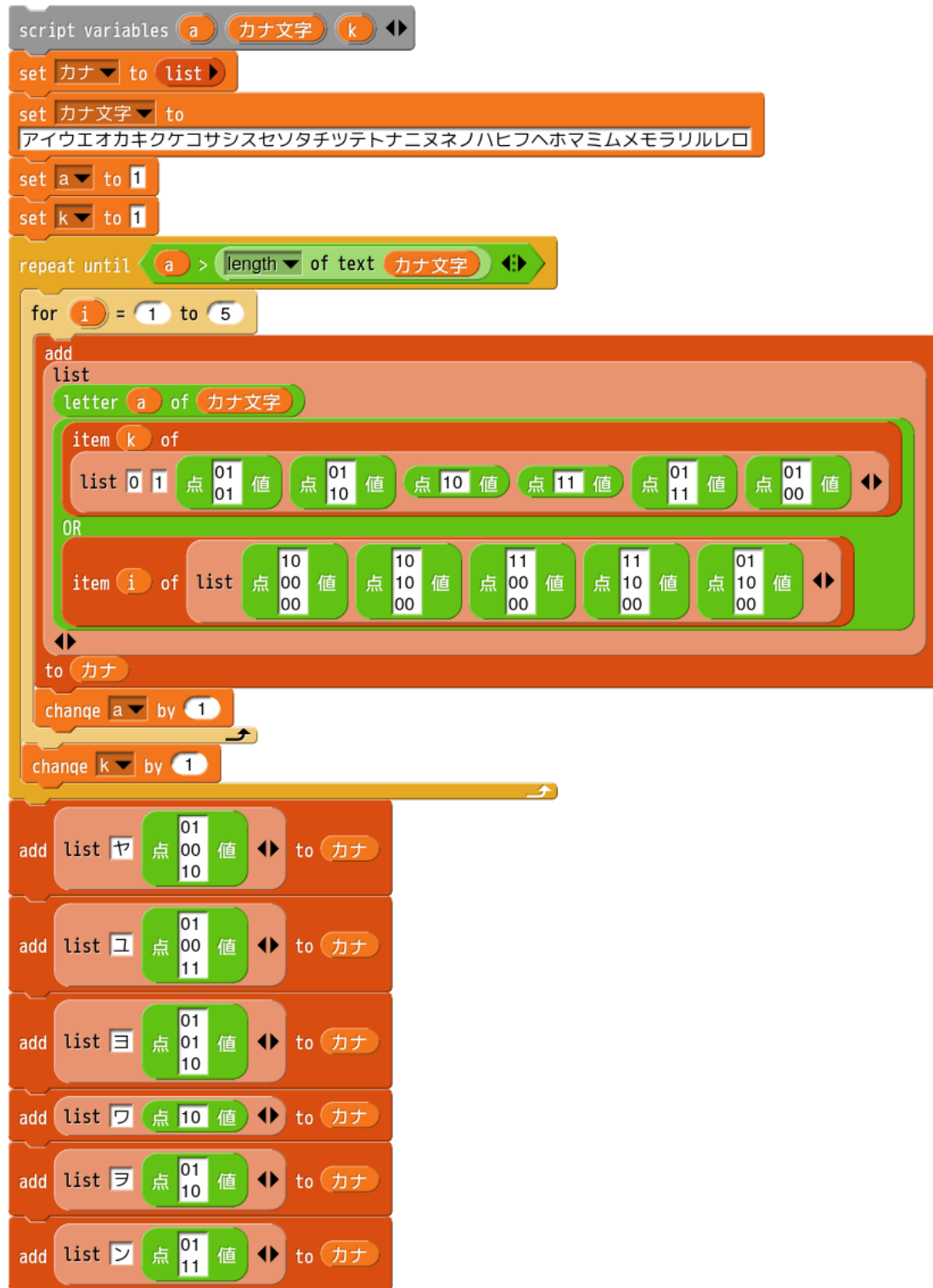
文字で検索して、対応する数値データにより点字を表示します。検索にヒットしない時は空白にしています。



逆変換です。



変換用リスト「カナ」を設定するスクリプトです。母音を表す「ア行」のデータに「カ行」「サ行」などを表すデータを OR するとそのデータが得られます。「ヤ行」などの例外があります。(OR ブロック作成については 144 ページ参照)



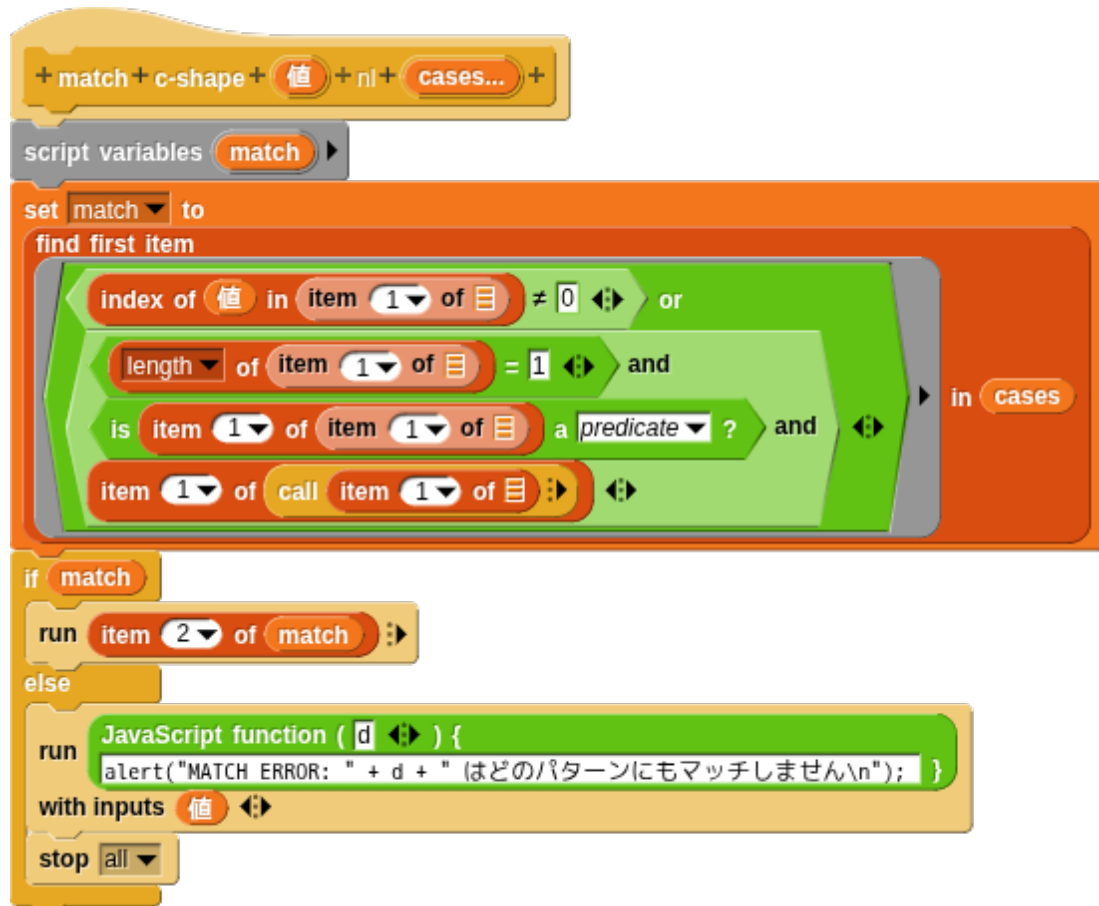
実際には「ガ」「シュ」などのような文字は二文字になりますし、この変換リストだけでは対応できません。数字やアルファベットの文字もありますし、それを示すための文字を置く必要があります。

8.22 match

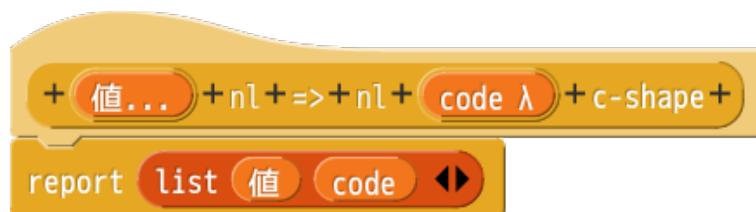
81 ページで switch や case ブロックについてふれました。

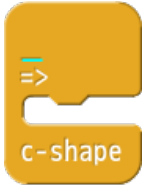
Rust 言語や Scala 言語では switch ではなく、match が使われます。case 文に相当する「() =>」でいろいろな指定の仕方ができます。「|」を使って複数の値を指定したり、パターンや範囲指定の条件で指定したりすることができます。マッチした場合はその「() =>」文で指定された処理だけを実行します。Snap! でエミュレートしてみます。

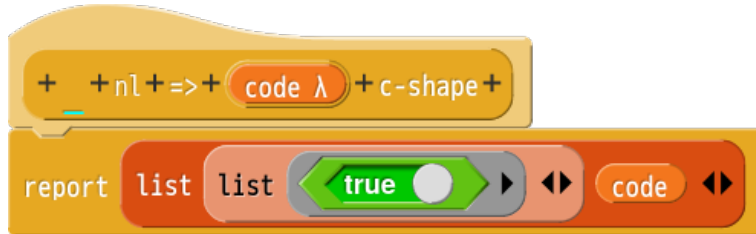
以下が定義ですが、match プロトタイプ部分の入力変数の仕様は「値」が Any type、cases が Command inline かつ Multiple inputs です。



case 相当ブロック **c-shape** は 83 ページのものと同様に「値」を増設できるものを使用します。code は Command(C-shape) type です。「|」で複数の値を指定するのではなく、入力スロットを増設することで対応します。



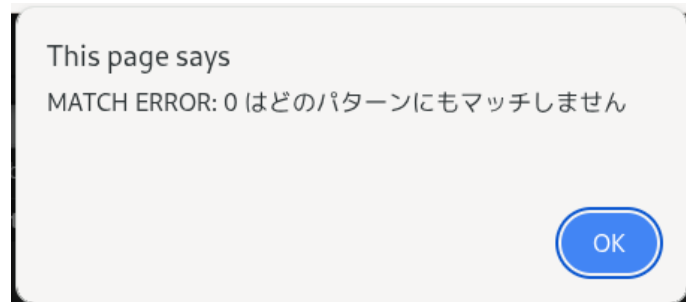
また、Rust 言語では default ではなく「_」ワイルドカードを使うので、 で行っています。「_」部分は Title text で「\$_-1.8-0-255-255」です。(72 ページ参照)



ワイルドカードでは条件判断部分を true にして必ずマッチするようにしています。C 言語タイプの default はどれにもマッチしなかった場合に実行されるものなのでどこに置いても良かったですが、ワイルドカードは置かれた場所でマッチしてしまうので以降のパターンは無意味になります。

Rust では match 文でパターンの指定漏れがあればコンパイルできません。match ブロックでは、JavaScript で alert を使ってエラーを表示します。OK ボタンをクリックしてください。すべてのスクリプトを終了します。

これは値が 0 で、すべての条件に当てはまらなかった場合のアラートです。



右のようにいろいろな指定の仕方ができます。なお、条件ブロックは ringify してください。そうしないと、

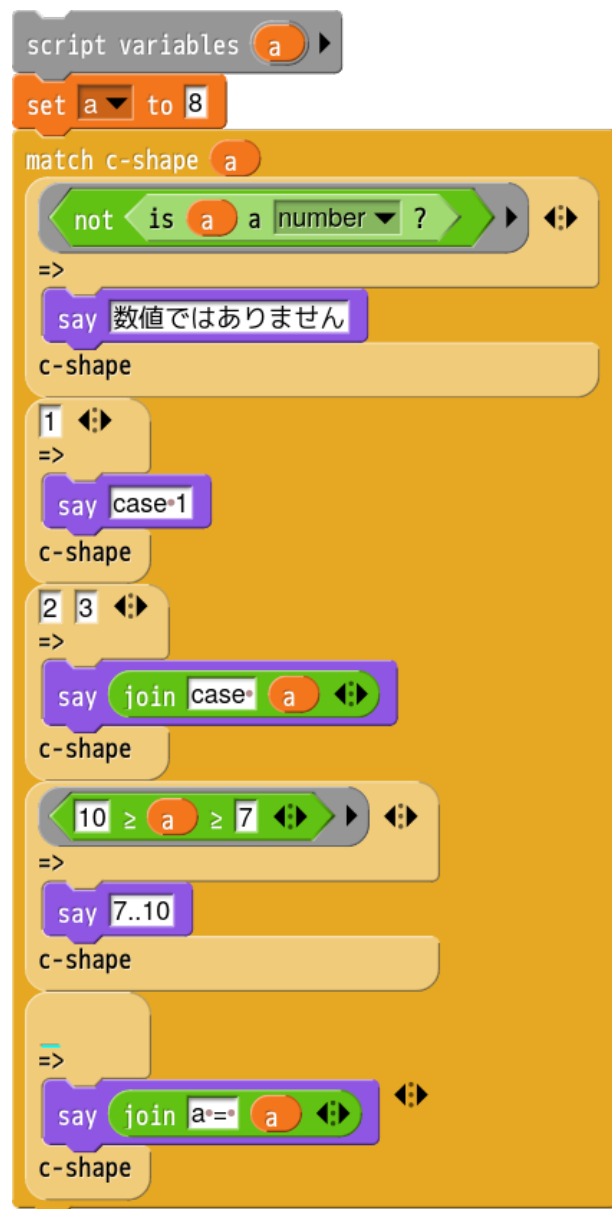


a が true の場合にマッチする条件になってしまいます。

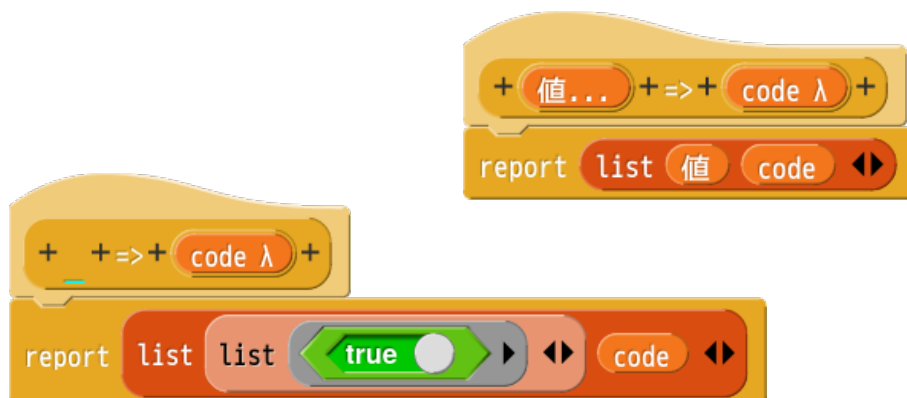
条件ブロックでの指定は 1 項目の時だけ有効です。複数項目の場合は条件ブロックとしてではなく同値としてのチェックになります。

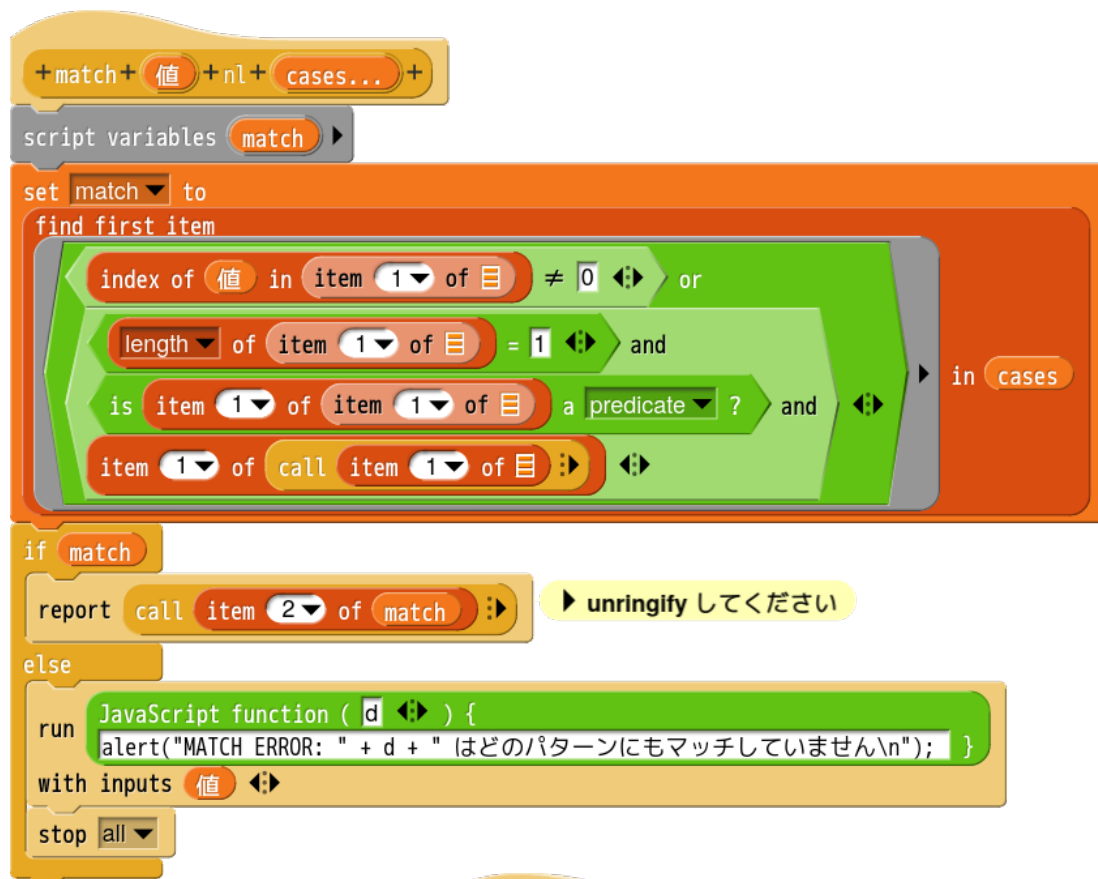
ワイルドカードを指定すればマッチの条件漏れは起きません。

「値」の属性などでもチェックできますし、数値の範囲での指定もできます。そのようなことは if ブロックでも可能ではありますが。

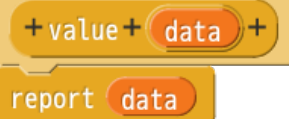


Rust の match は、「式」ということで値を返すことができます。リポーター型の match です。
() => や _ => で、変数 code は Reporter type です。



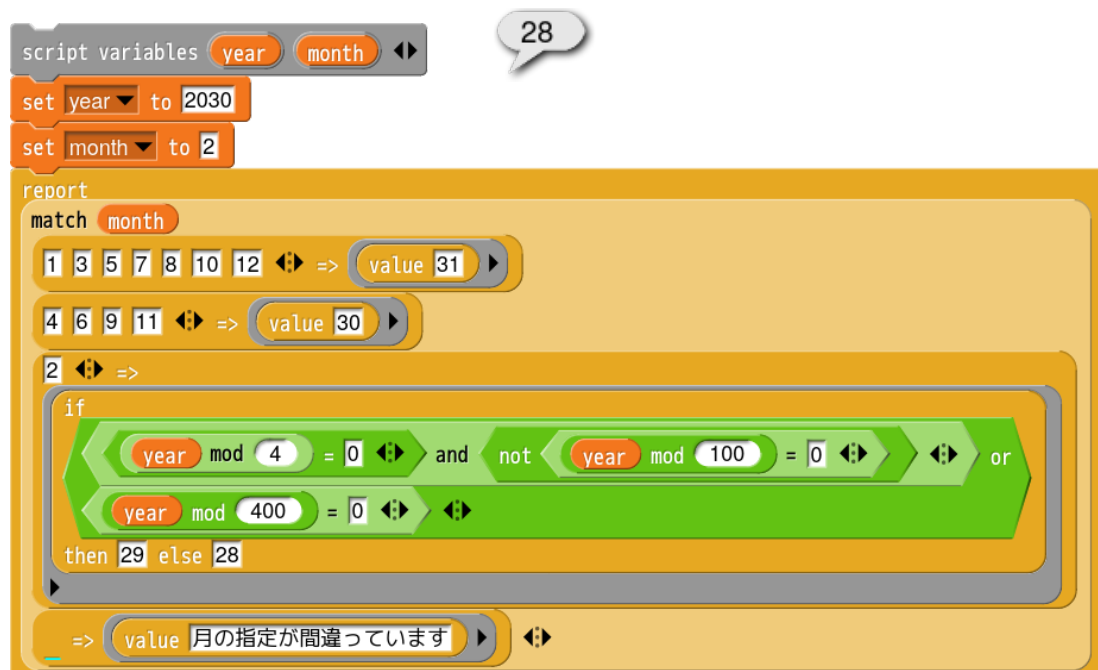


report を使っても同じですが、



で値をレポートします。

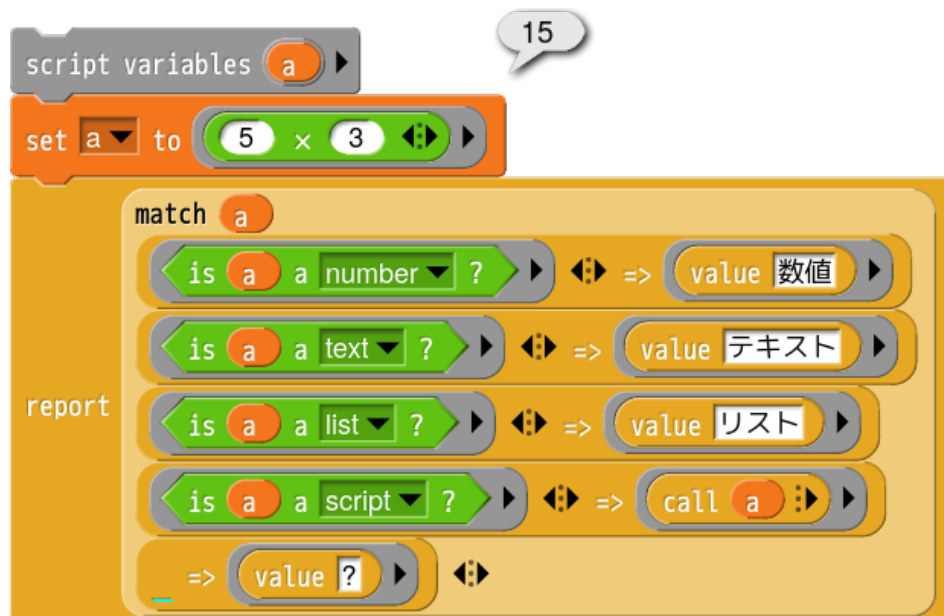
西暦年と月を指定してその月の日数を求めるスクリプトです。



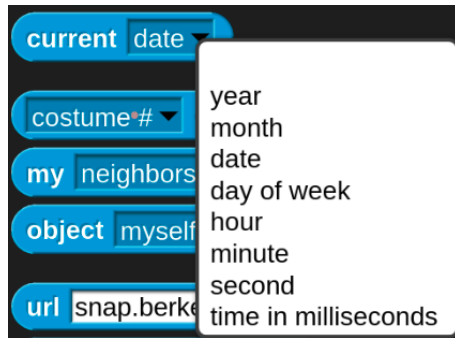
正規表現を使った例です。n 進数の数値を十進数で表示します。二進数は「B」、八進数は「O」（アルファベット）、十六進数は「X」を先頭に付けます。（138 ページ以降参照）



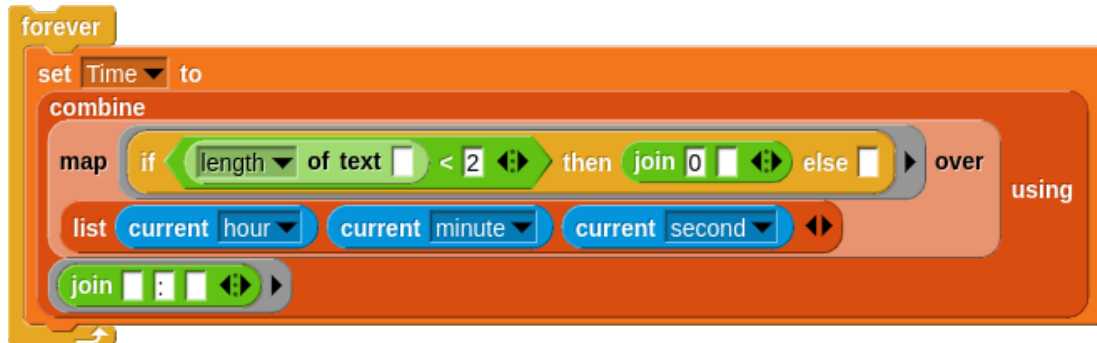
あまり意味はありませんが、型別に処理を分けています。



8.23 時計



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



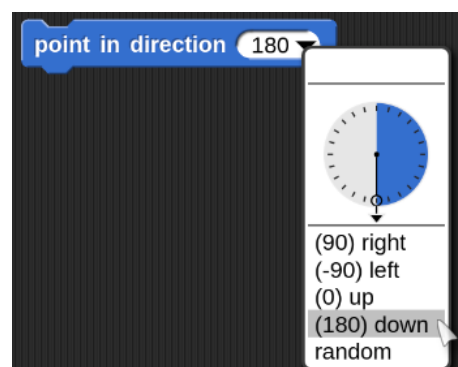
時間のデータを各針の角度に変換すればアナログ時計ができます。針はスプライトで表現してもいいですし、その都度ペンで clear 描画を繰り返してもいいです。

秒数は、**current second** で求められます。

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



秒数をアナログ時計の秒針の角度に変換するには $\text{秒数} \times 360 (\text{度}) \div 60 (\text{秒})$ になります。30 秒の場合、 $30 \text{ 秒} \times 360 (\text{度}) \div 60 (\text{秒}) \Rightarrow 180 \text{ 度}$ です。



秒針の角度です。(60 秒で 1 回転)



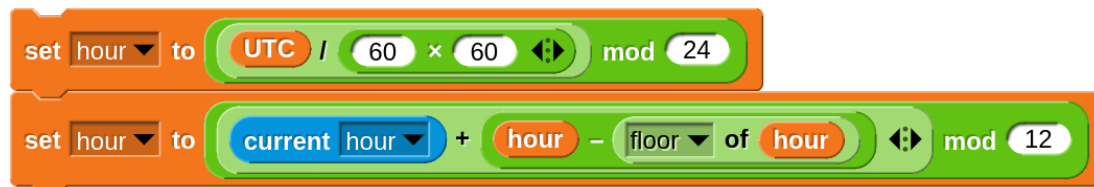
秒単位の時間を 60 で割って分単位の時間にしてから、それを 60 で割った余りを求めれば分数が求められます。



分針の角度です。(60 分で 1 回転)



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



時計の角度です。(12 時間で 1 回転)

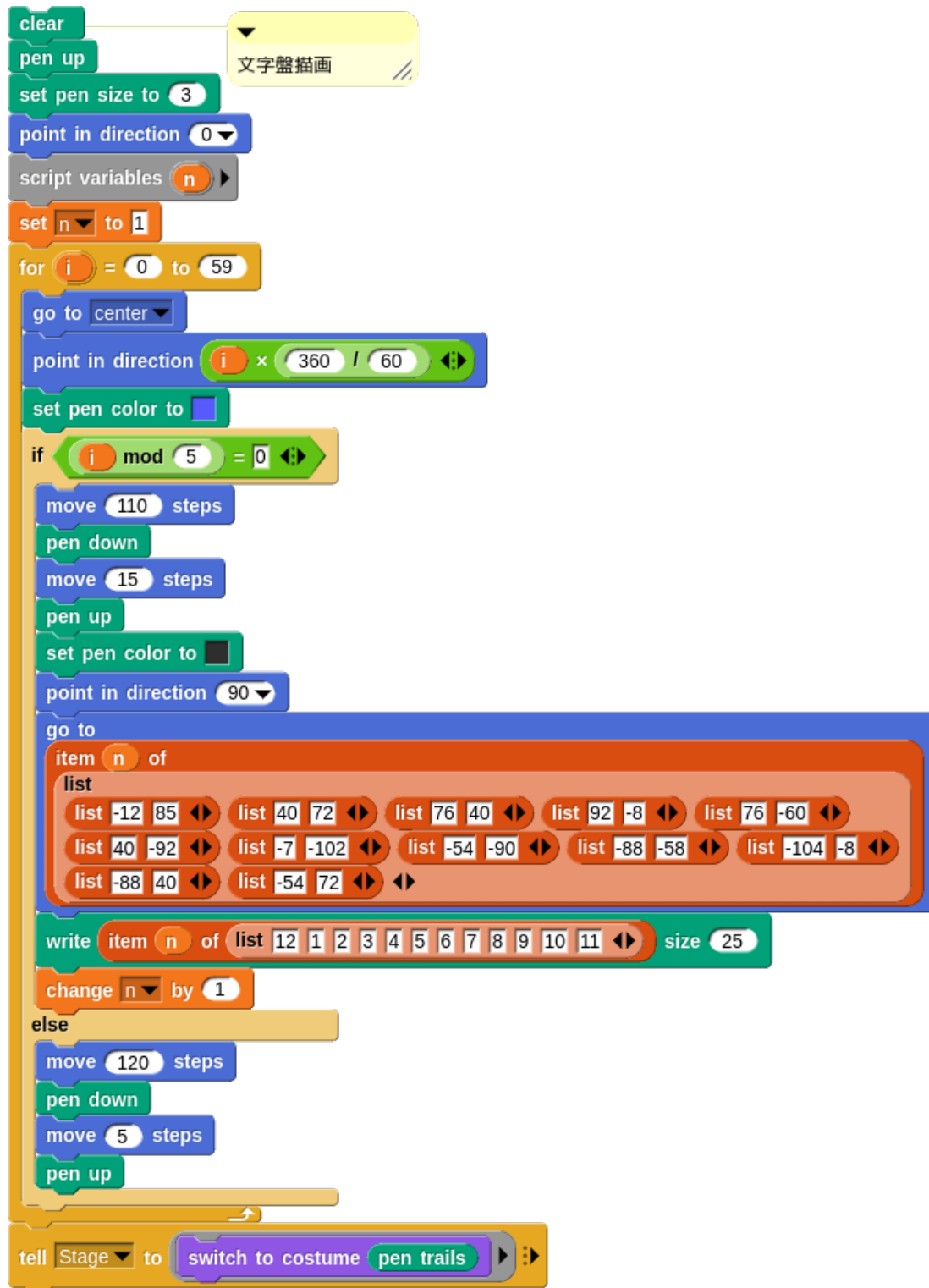


ペンで描く場合はこの逆の順序で描けば実際の時計と同じ重なりになります。

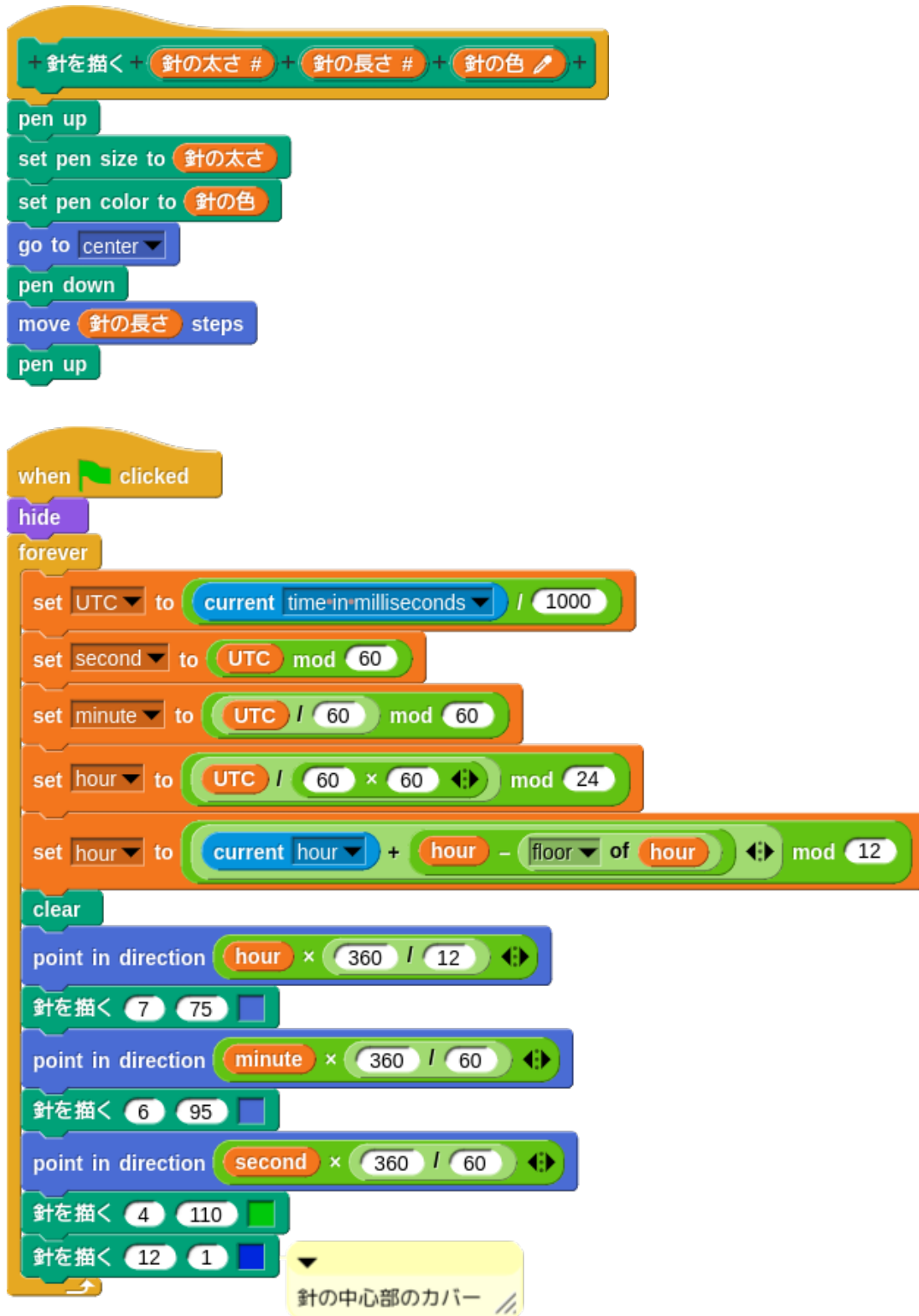
Scratch や Snap! の角度指定で画面上方向が 0なのは時計やメーターの針の角度を指定しやすくするためかもしれません。

次のページから文字盤を描画し、ペン描画による時計のスク립トを示します。

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



針を描くための定義ブロックです。



8.24 並列実行について

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

スプライト 1



スプライト 2



実行順序

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

length: 6

set 実行順序 to list

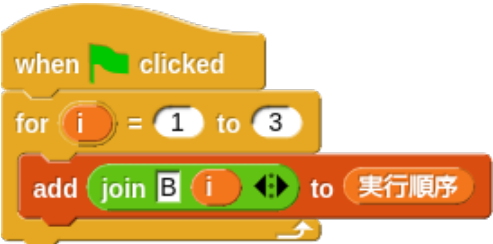
を実行してから  をクリックすると、左のようにリストに実行順序が表示されます。1 ブロックずつ交互に実行されるのではなく、一方はもう一方のスクリプトの実行が終わるまで待たされます。実行の順番はスプライト、スクリプトが作成された順番になるようです。

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

スプライト 1



スプライト 2



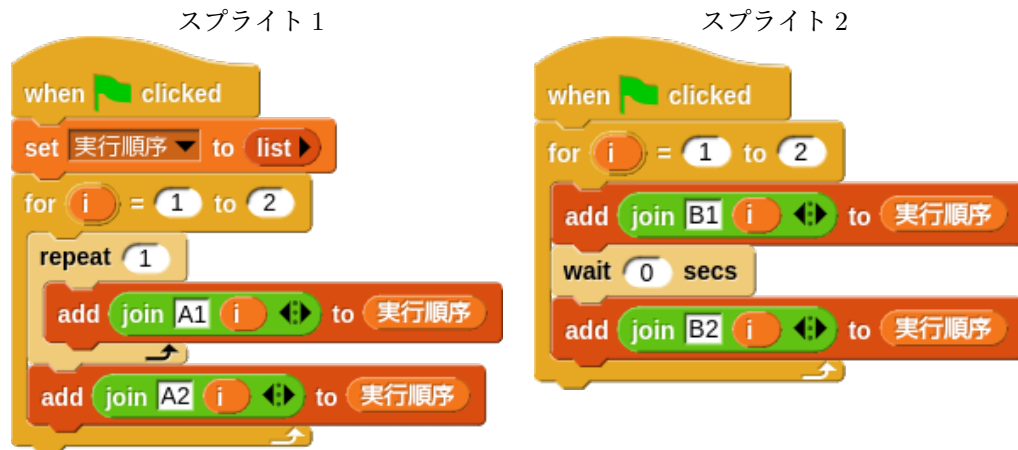
実行順序

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

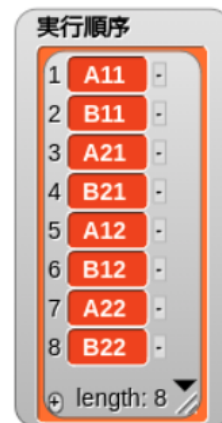
length: 6

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

次のように、repeat 1 の C 型ブロックで囲んだり wait 0 を入れることでも処理を 1 ブロックずつ交互に行うことができます。



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



プログラムを統括するメインの役割を Stage に持たせるやり方があります。プログラムによっていろいろなスプライトが用いられますが、Stage はどんなプログラムにも必ず存在するためと思われれます。しかし、次のようにスクリプトの実行順序も最下位になっているので、ここでプログラムの初期設定をするには Stage のスクリプトを最初に実行させる工夫（他のスクリプトの先頭に wait 0 を入れるとか broadcast で起動させるとか）が必要になります。



8.25 行列の積

8.25.1 inner product

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

2 行 2 列の行列同士では次のような計算方法になります。

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

a 側の行列の列数と b 側の行列の行数は同じ必要があります。

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

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

1	17
length: 1	
2	39
length: 1	
length: 2	



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

1	A	B
1	40	46



$\begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$ は 1 行 3 列の行列です。list 1 2 3 としてしまうと 3 行 1 列の行列になってしまうので、

1	A	B	C
1	1	2	3

にする必要があります。

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

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

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

一番わかりやすい例として、 $(1, 0)$ の点を 90 度反時計回りに回転させると $(0, 1)$ の点になります。

左側の行列、回転させるための変換行列 R を作成するブロックです。



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

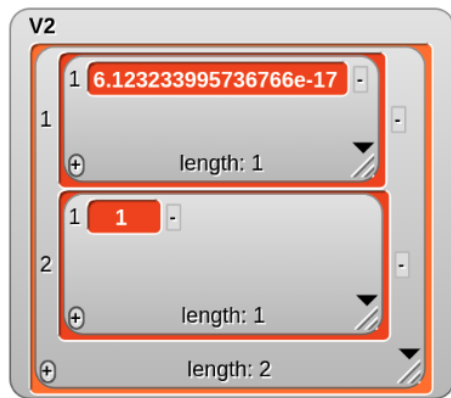


この行列 $\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$ と座標 (x, y) $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ の積

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



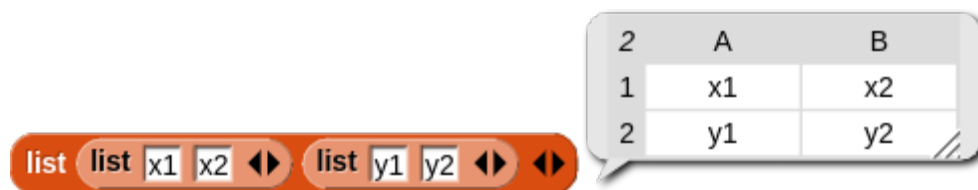
$V0$ が変換前の座標で、 $V2$ が変換後の座標です。



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

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

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

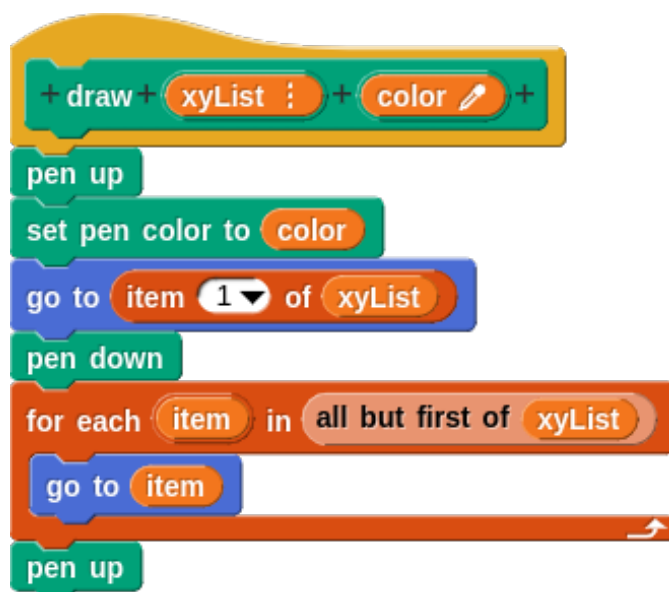


転置行列を使うと、普通の (x, y) 座標の指定の仕方ができます。



(x, y) 座標のリストで与えられた図形を一筆書きで描くブロックです。

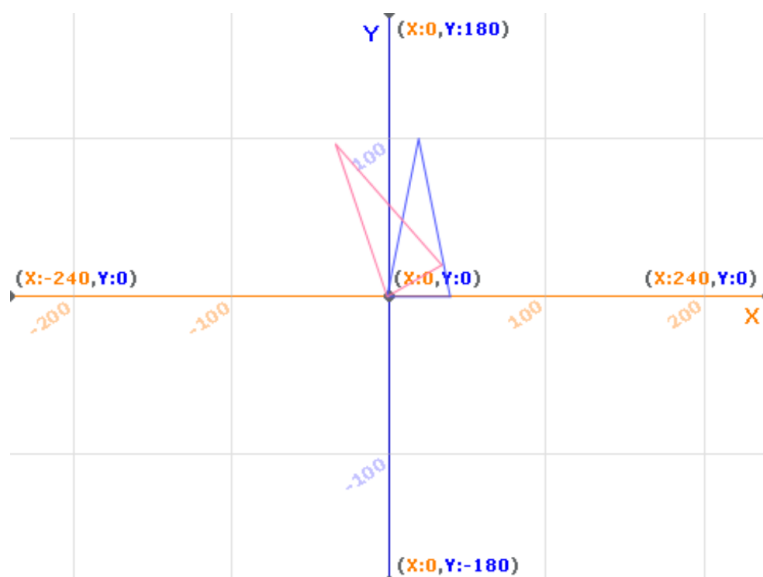
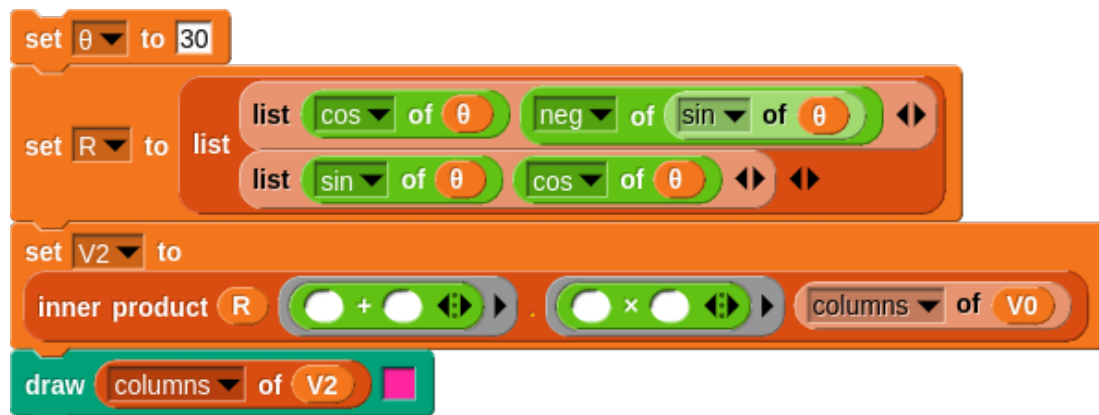
go to ブロックは **go to random position** にリストをドロップしたものです。



三角形  を描くスクリプトです。



それを半時計回りに 30 度回転させるスクリプトです。



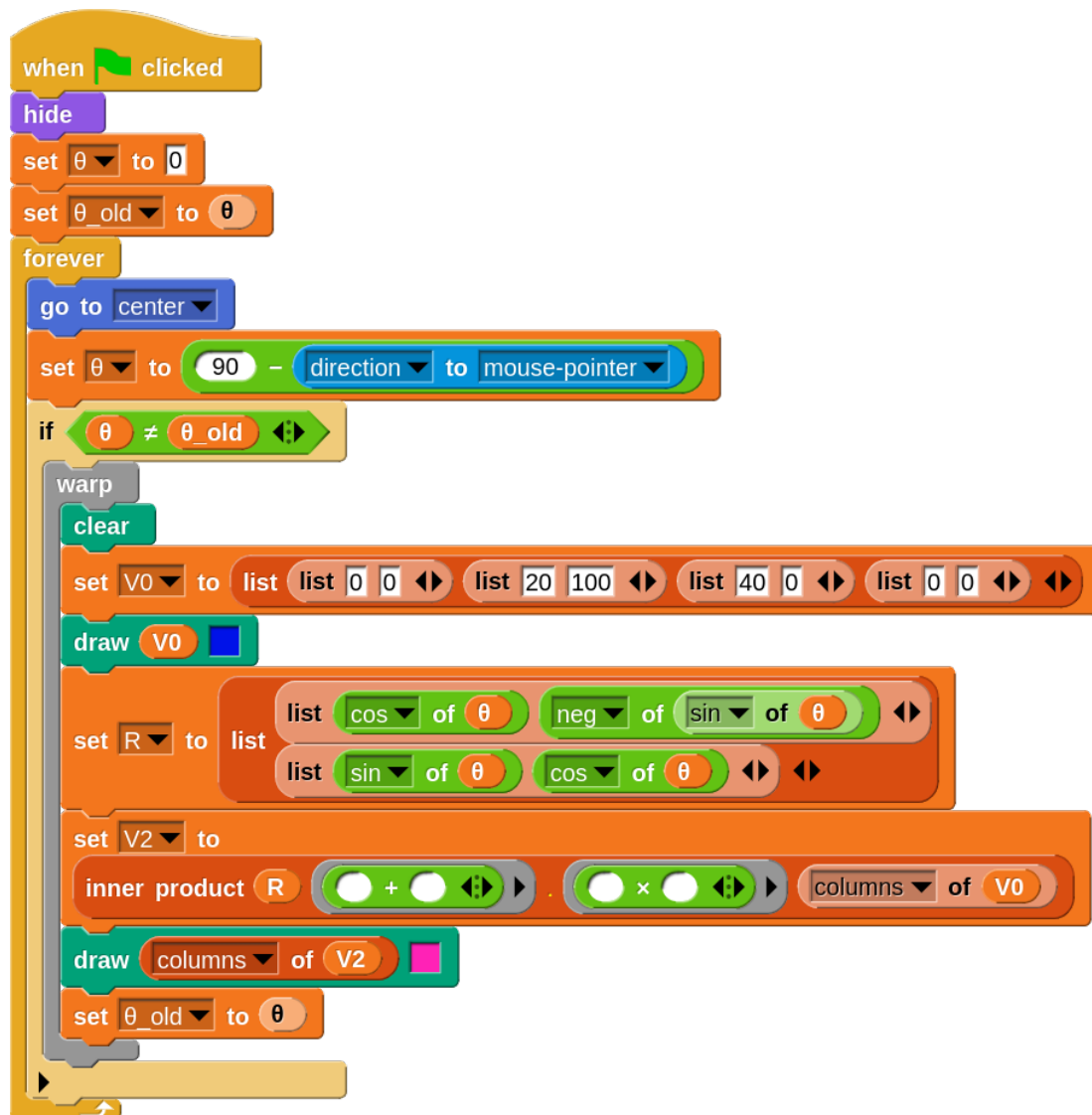
[参考文献]

『Python ではじめる数学の冒険

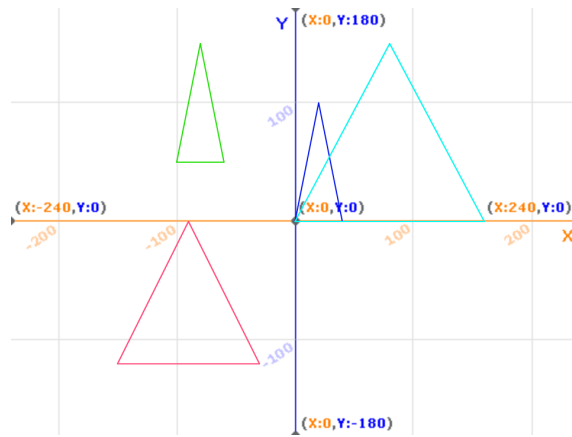
プログラミングで図解する代数、幾何学、三角関数』

Peter Farrell 著 鈴木幸敏 訳 オライリー・ジャパン 刊

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



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



x 軸方向に 4 倍、y 軸方向に 1.5 倍します。

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



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

x 軸方向に -100、y 軸方向に 50 平行移動します。

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



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



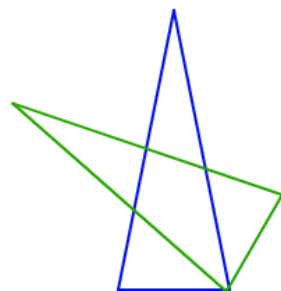
平行移動を利用すれば図形の任意の位置を中心として回転させることができます。三角形 V0 の重心の座標を中心にして 60 度回転してみます。

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

```

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

```



8.25.2 outer product

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

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

この記号のようなものはギリシャ文字の ι イオタです。

次のようにすると、九九の表の一部ができます。

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

outer product    

計算方法を表示させてみます。

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

outer product    

単位行列なども作成できます。

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

outer product    then 1 else 0 

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

outer product    then 1 else 0 

[参考文献]

『基礎からの APL 解説と例題例解』

西川利男/日本アイビー・エム 共著 サイエンスハウス 刊

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

set N to 5

1 ~ 5 の整数のリスト N を作り、その中の素数を求めてみます。

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

(174 ページ、outer product の計算方法で $\times$ の代わりに mod になります。)

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

pipe N → outer product mod = 0

A の列の意味は、1 ~ 5 の値を 1 で割った余りが 0 かどうかを表しています。これはすべて割り切れるので全部 true です。B の列の意味は、1 ~ 5 の値を 2 で割った余りが 0 かどうかを表しています。この場合、2 と 4 の時だけ true です。他の C, D, E ではどちらも同じ値の時だけ true です。

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

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

combine in rows (reduce by column vectors) +

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

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

pipe N → outer product mod = 0 combine in rows (reduce by column vectors) +

行中の true の要素の合計が 2 のものを map を使って求めます。

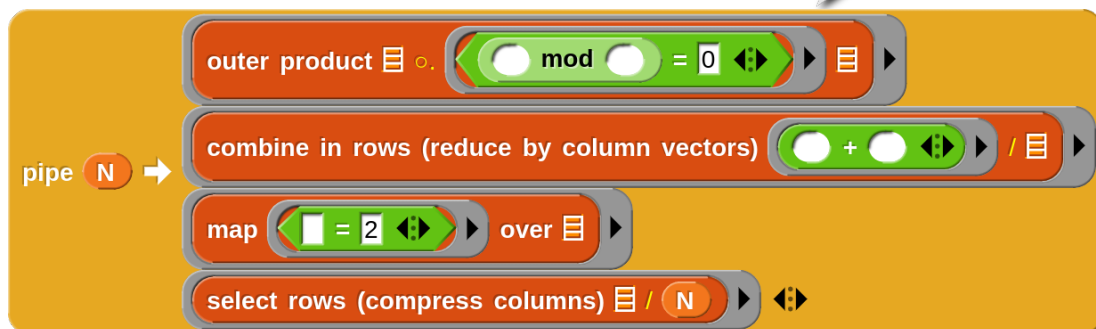
1	false
2	true
3	true
4	false
5	true
length: 5	



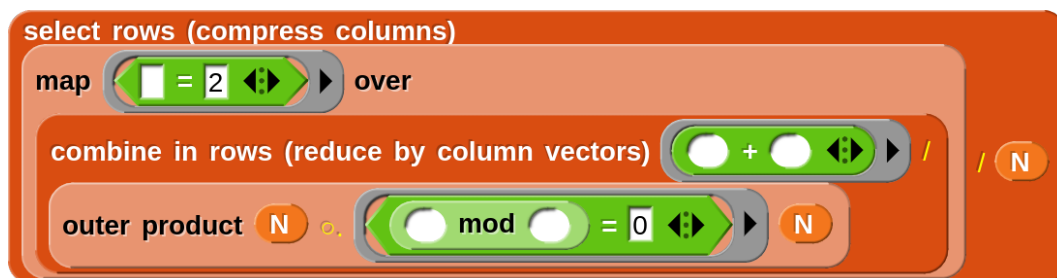
2, 3, 5 の位置が true になりました。その位置に対応する N の要素のリストを求めます。
select rows ブロックはリストの中から true で指定した位置の要素を集めてリストにします。
これにより、素数のリストが求められます。

select rows (compress columns) / N

1	2
2	3
3	5
length: 3	



次のようにまとめることもできます。



8.26 Continuation 継続

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

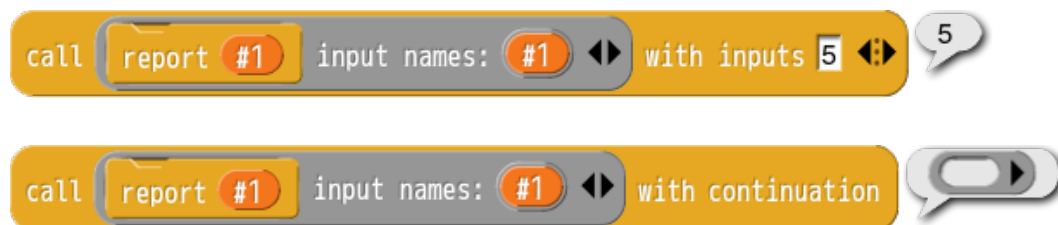
8.26.1 run と call の with continuation バージョン

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

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

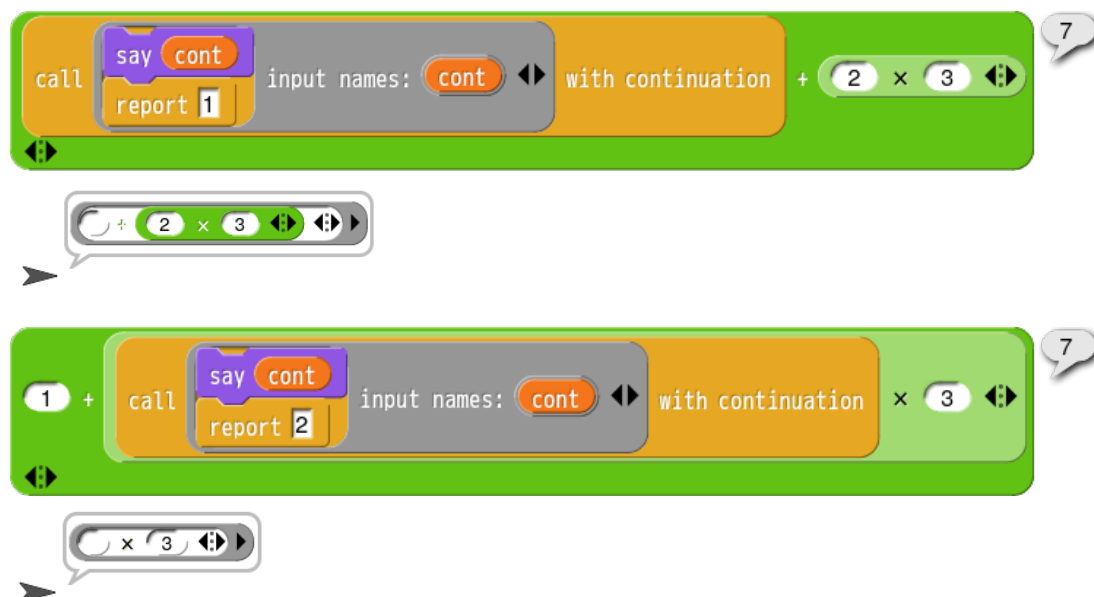


しかし、リング端の三角をクリックしてフォーマルパラメーターを使用してみると、call with continuation の方は空のリングが表示されます。



call with continuation のフォーマルパラメーターにはこのブロックに継続するスクリプトがセットされます。この場合はこの後に処理すべきものが何もないので空です。

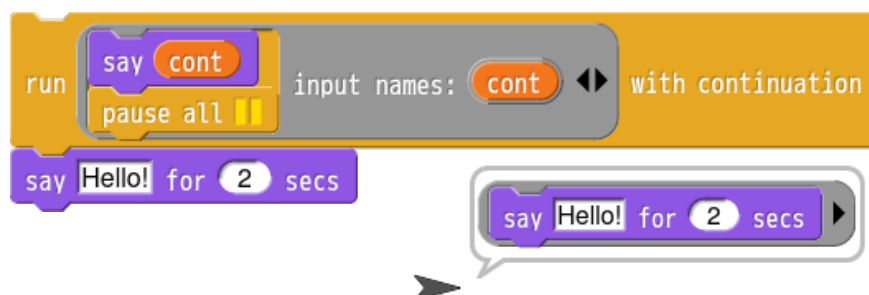
 の各スロットにこれをセットして、その位置での継続を表示させてみます。それぞれその時点での継続、その後処理すべき内容がセットされます。(フォーマルパラメーターを cont にリネームしています。)



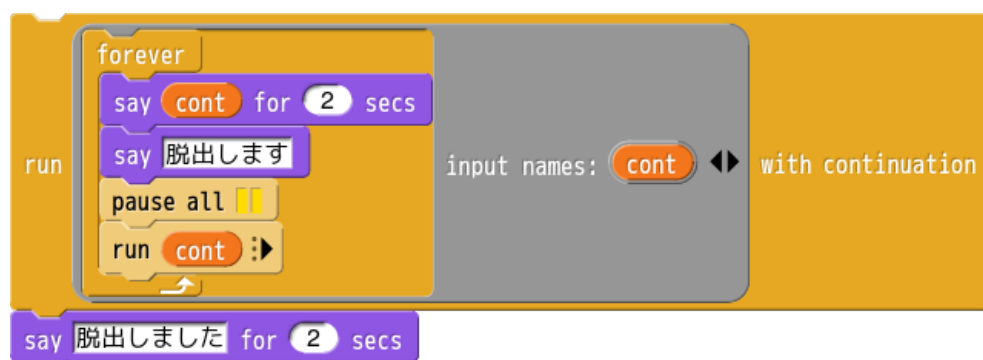


用途としては、call with continuation や run with continuation のリング内部にあるスクリプトで、ある条件の時にその外（継続）への脱出に使われます。

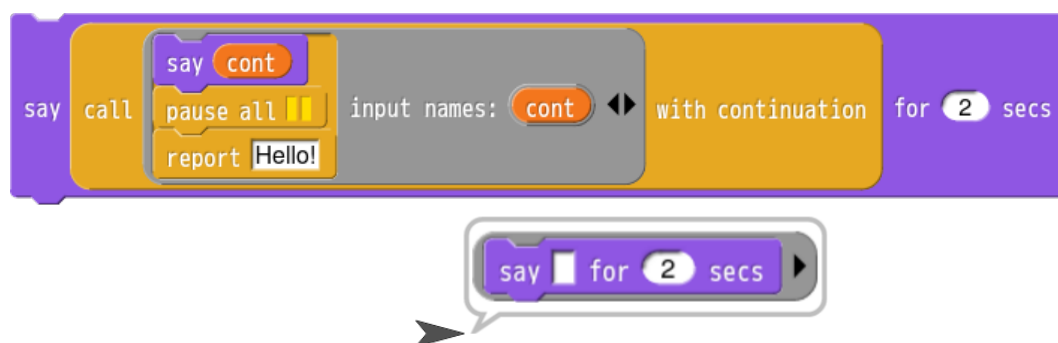
このスクリプトを実行すると、run with continuation の継続が表示されます。



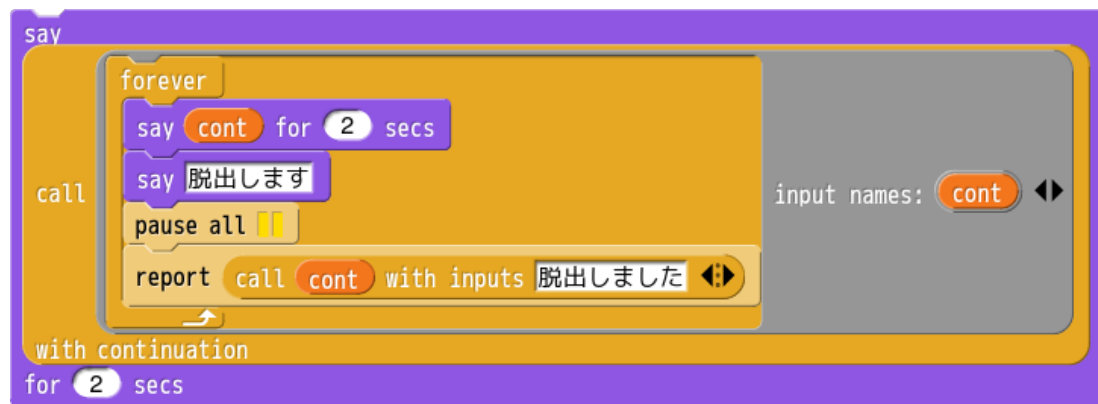
say や pause all が置かれているところにループを置き、その中で run cont とすると、run with continuation の中から外へジャンプすることができます。



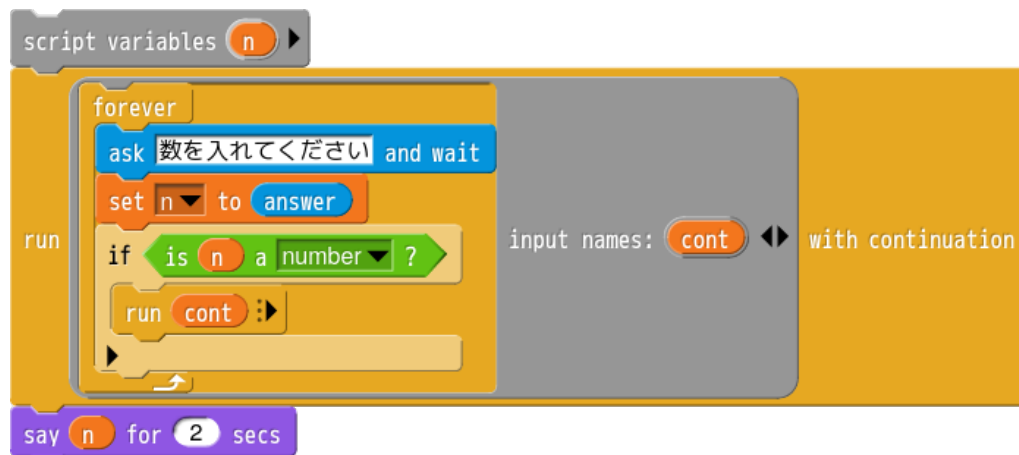
このスクリプトを実行すると、call with continuation の継続が表示されます。自分自身が入っているので、say の入力为空のブロックが表示されます。



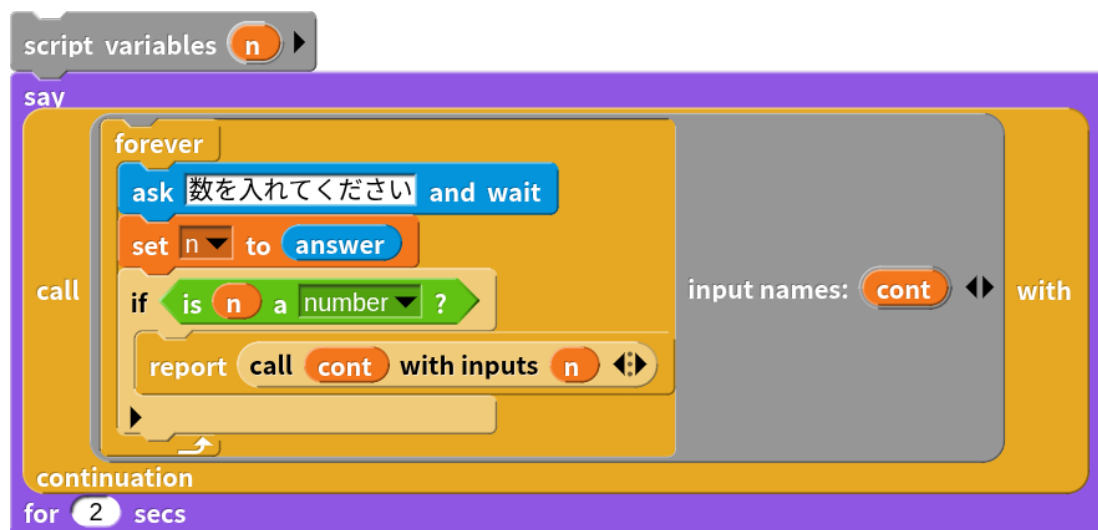
say や pause all が置かれているところにループを置き、その中で call cont とすると、call with continuation の中から外へジャンプすることができます。call with continuation はリポータブロックなので値を返さなければなりません。call cont とすることで脱出になりますが、with inputs で値を指定することでその値「脱出しました」を返すことができます。



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

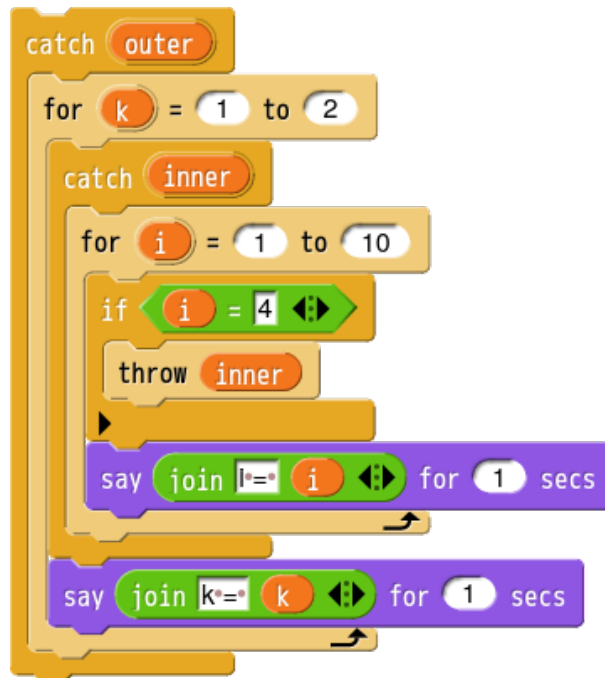


call with continuation 版です。



8.26.2 catch と throw

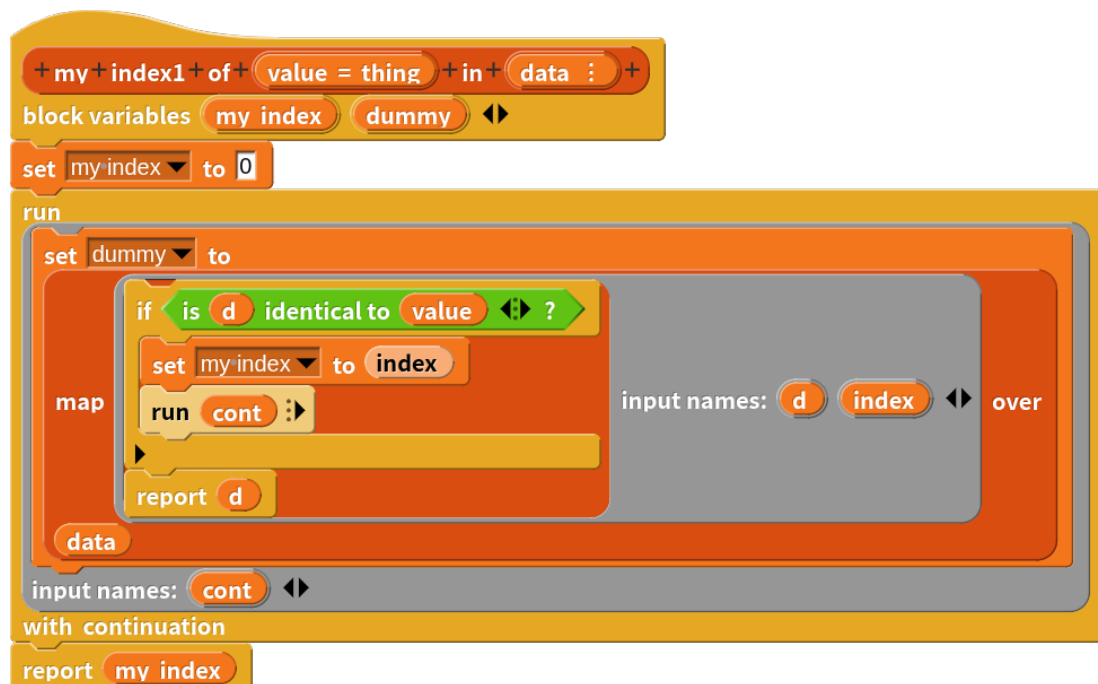
継続を意識することなくループなどからの脱出に使用できるものが、Libraries の Iteration, composition にある catch と throw です。



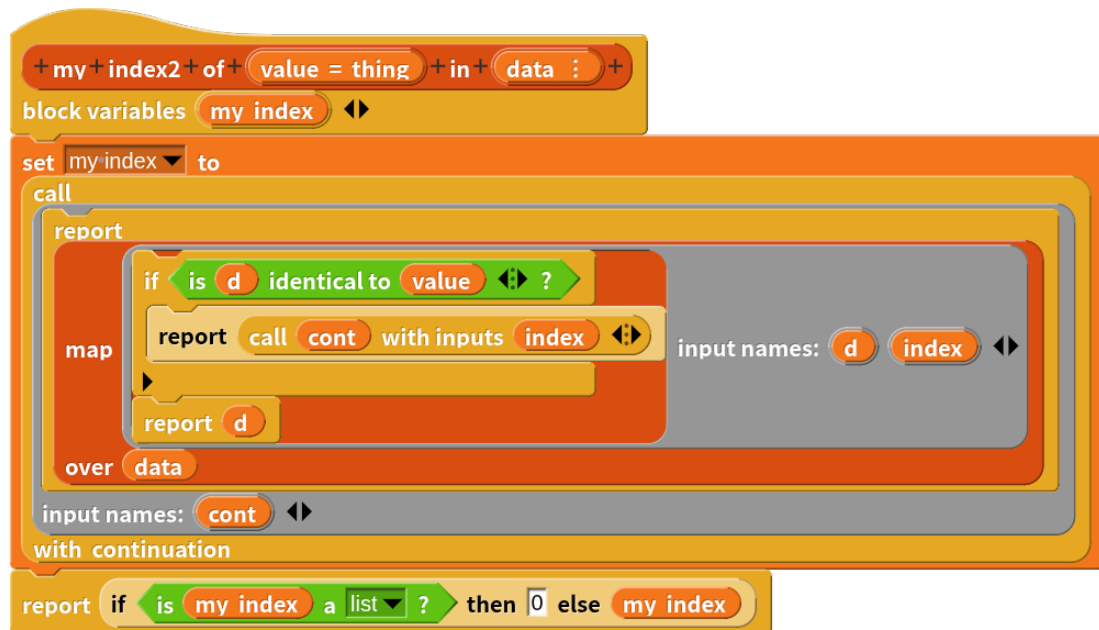
これを実行すると、i の値が 4 になった時に内側のループから脱出し、catch inner の次のスクリプトに進みます。throw outer にすると、外側のループから脱出し、catch outer の次のスクリプトに進みます。ある条件の時に、throw で指定したラベルの付いた catch ブロックの外側に脱出するという仕組みです。

8.26.3 map からの脱出

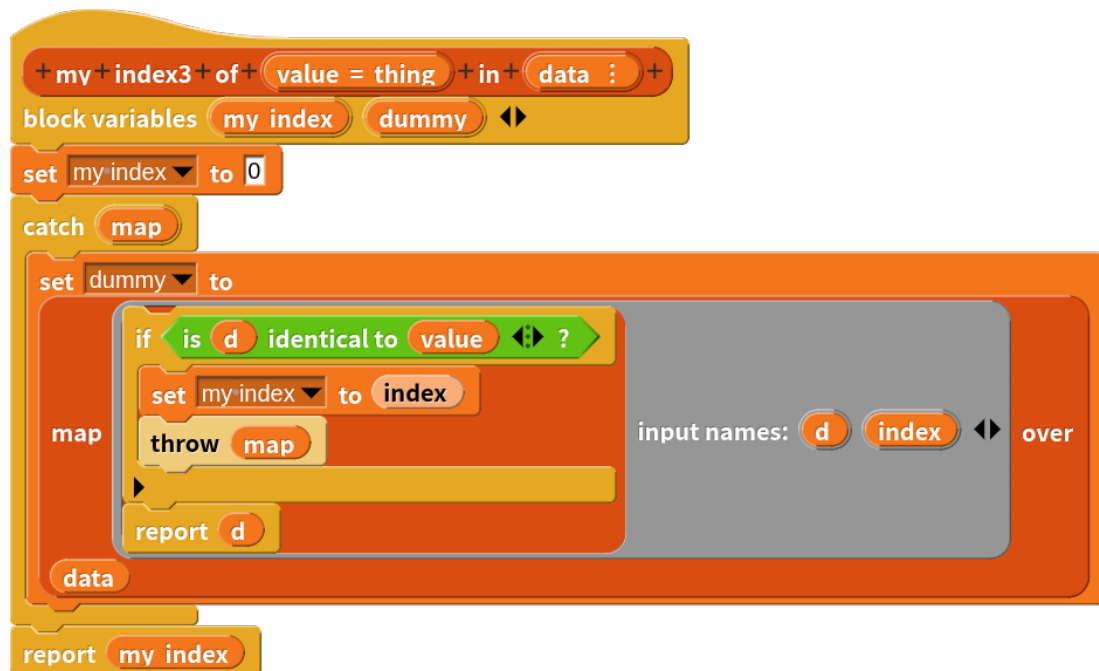
107 ページで index of thing in 目 を扱いました。この方法だとリストのすべての要素を走査しています。次のようにすると、要素が見つかった時点で脱出することができます。run with continuation 版では map を使用するにはダミーの変数 dummy が必要です。



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

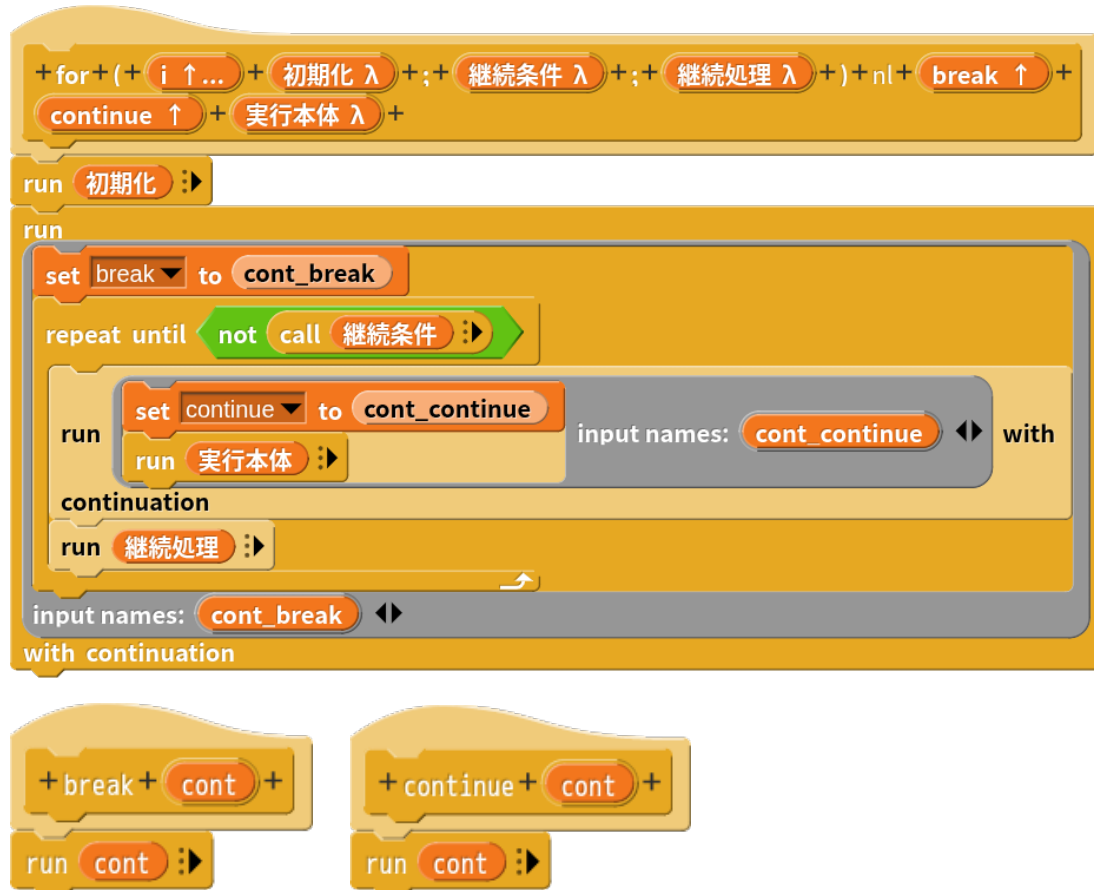


catch throw 版です。

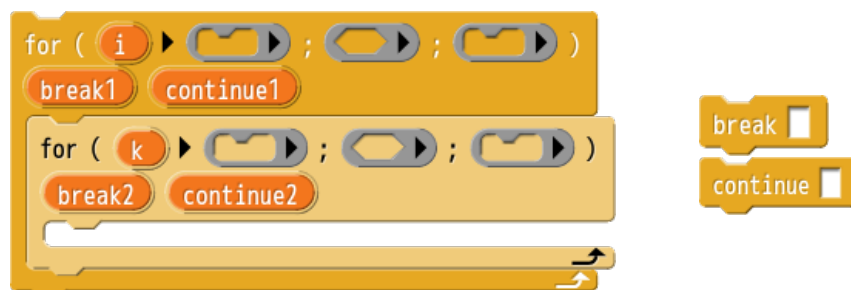


8.26.4 break と continue

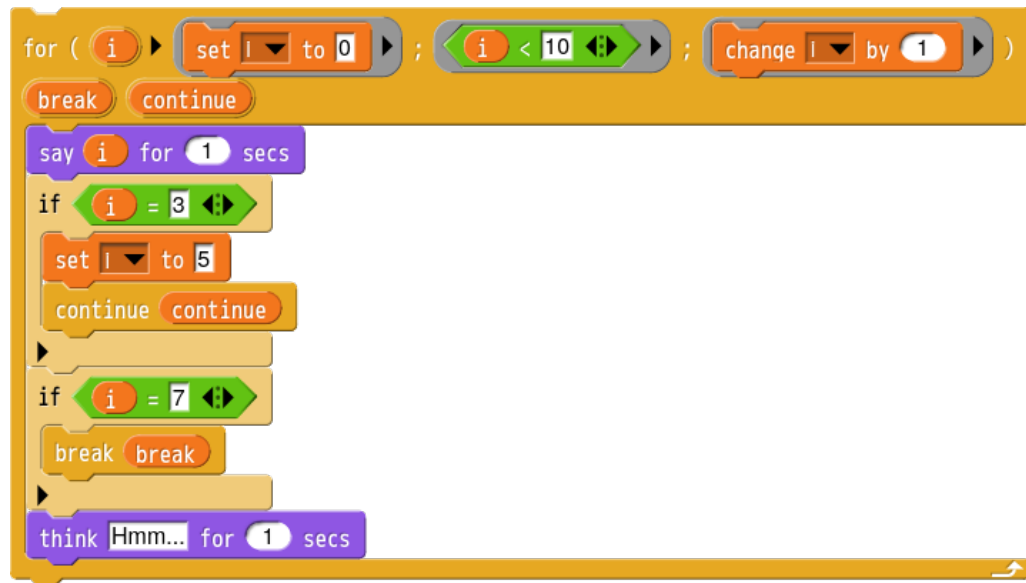
継続の機能を利用すると、C 言語風 for (; ;) ループに break や continue の機能を持たせることができます。break はループから脱出する機能で、continue は「継続処理」へジャンプして次の繰り返しを開始する機能です。紛らわしいですが、ここで入力変数名に使われている「継続」はループ処理を続けるという意味で、continuation に関するものではありません。変数 i は special から variables に設定します。(66 ページ参照) break と continue 定義中の cont は Any type 変数です。



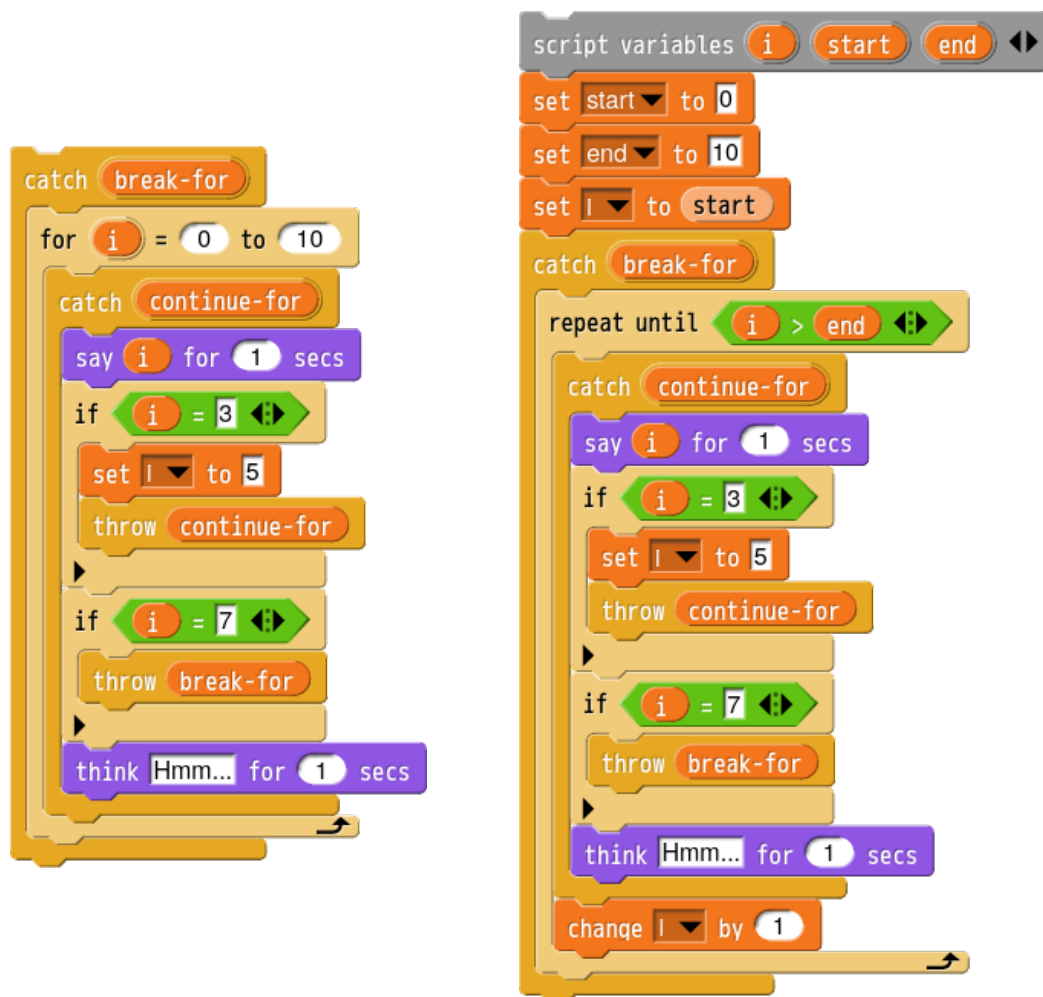
break と continue は、upvar オプションの変数です。for (; ;) ブロックから break, continue 変数をそれぞれ break や continue ブロックにドロップして使用します。二重三重のループの場合は、変数名に数字を追加するなどして区別してください。



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



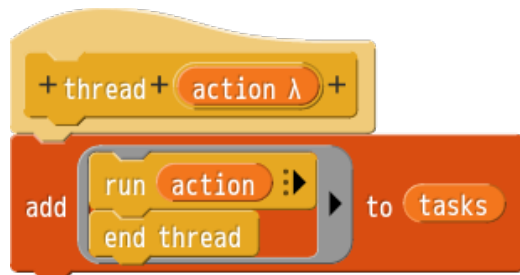
同じことをする、プリミティブの for と repeat until を使った catch と throw 版です。



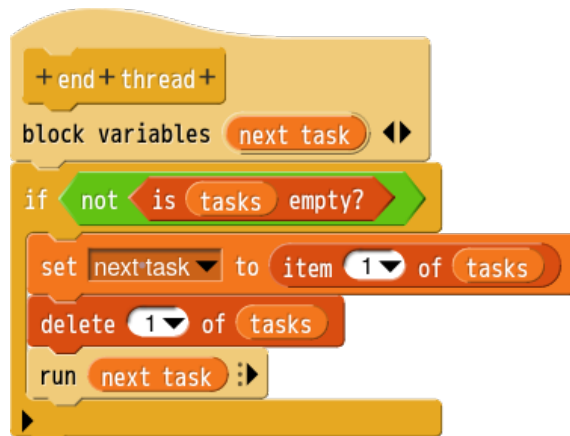
8.26.5 thread

Snap! では複数の **when clicked** が存在する場合、それぞれのスクリプトを並列に実行しているように見せています。継続を使用すると、一つのスクリプト内にある複数の処理ブロックを並列に実行の順番や処理の移行のタイミングをプログラミングすることができます。並列処理されるそれぞれのスクリプトを thread (スレッド) と言います。

処理するスクリプトの記憶場所として **tasks** の変数を作成しておきます。以下に示す 3 個の定義ブロックはスレッド処理を行うための 3 点セットです。

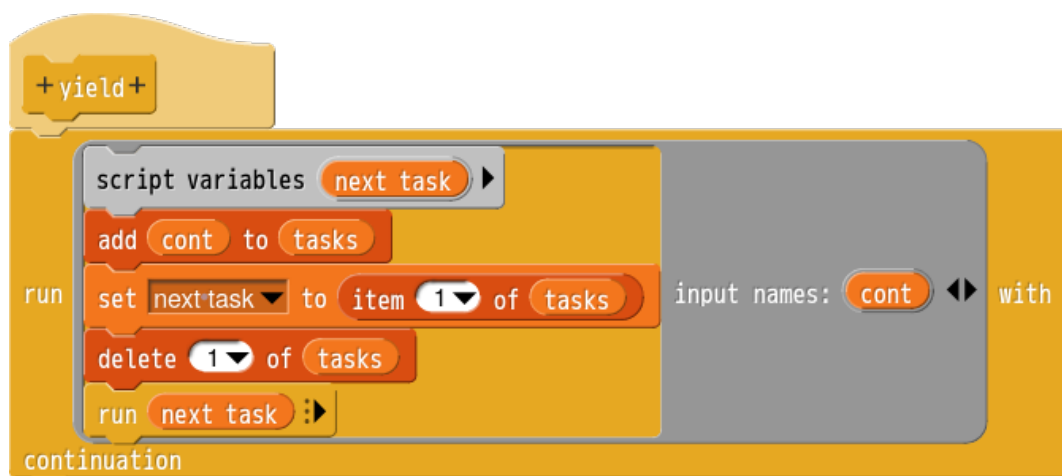


スレッドを設定するブロックです。(設定されたスレッドを tasks リストに加えていきます。) action は Command(C-shape) 型です。



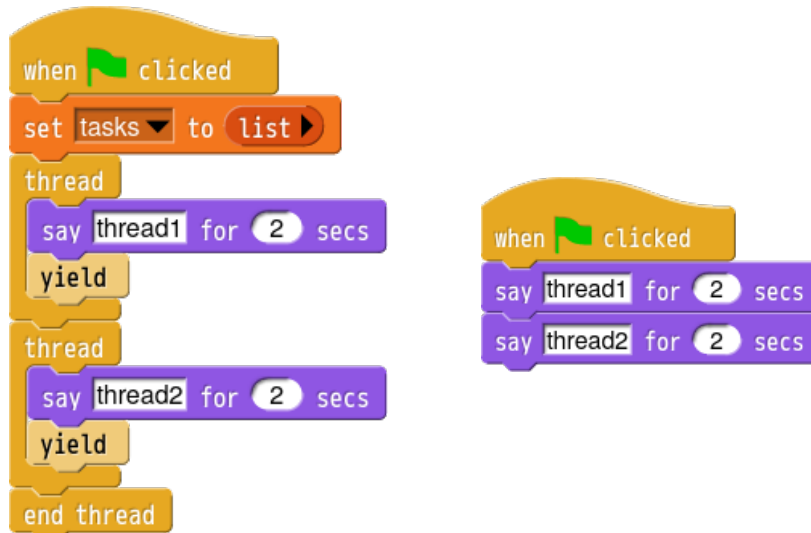
tasks リストが空でなかったら tasks リストの先頭のスクリプトを取り出して実行します。これ以上スレッドが無いことを示します。これによって設定された複数のスレッドの実行が始まります。つまり、これを置かないとスレッドが実行されません。

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



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

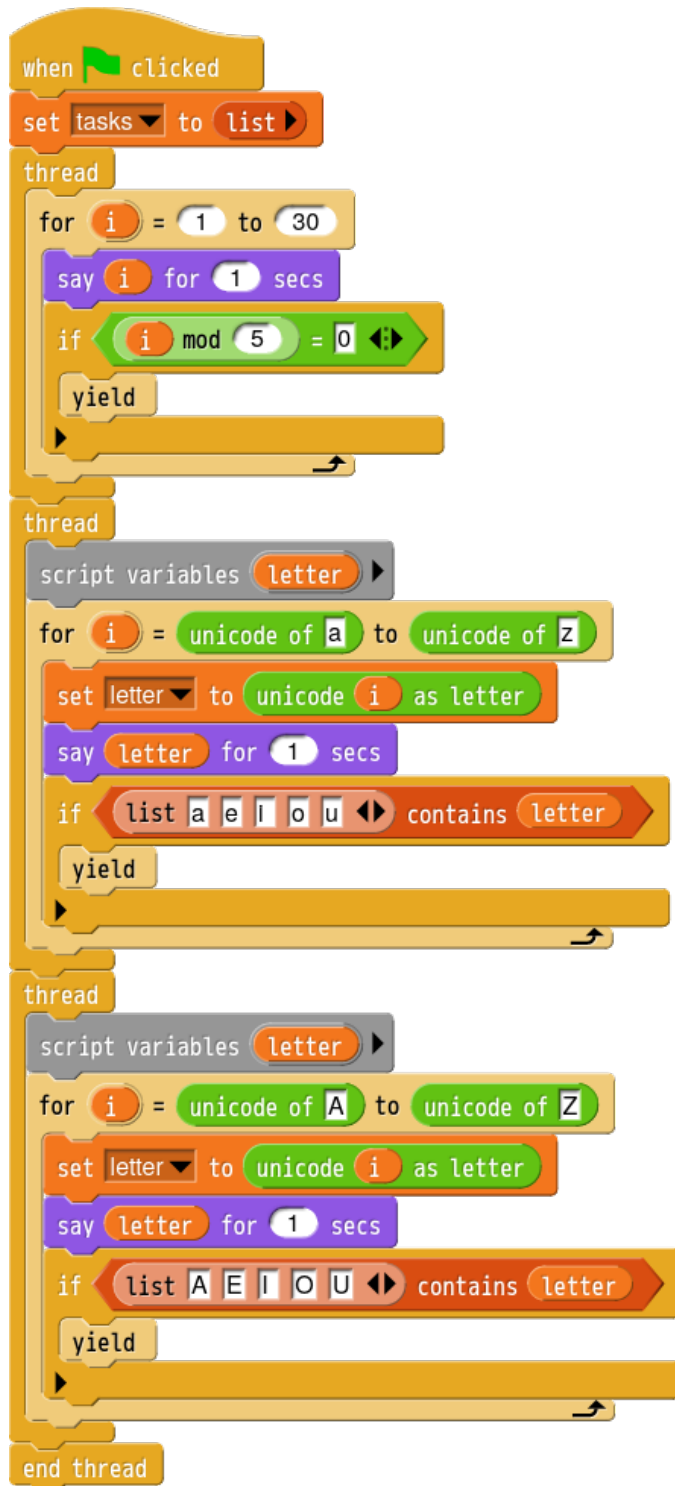
次のスクリプトは、右のスクリプトと同じことなのであまり意味はありませんが、使い方の例です。



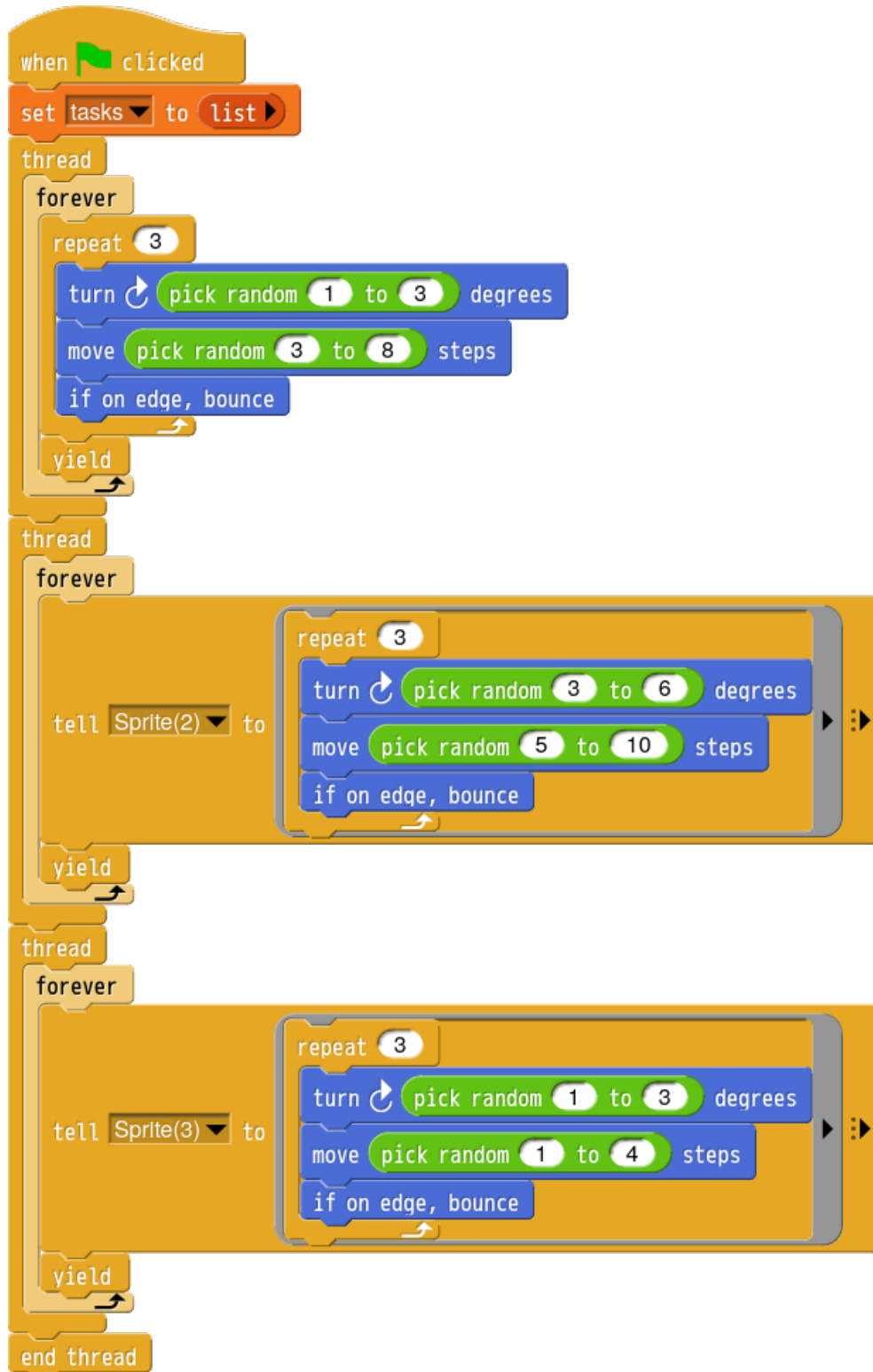
番号を表示するだけの 3 個のスレッドを実行してみます。



マニュアルに掲載されているスクリプトを基にした例です。指定の条件の時に次のスレッドに移行します。最初のスレッドは 1 から 30 までカウントしますが、5 で割り切れる値の時に次のスレッドに移行します。次のスレッドでは a から z までを表示しますが、a, e, i, o, u の時に次のスレッドに移行します。次のスレッドでは A から Z までを表示しますが、A, E, I, O, U の時に次のスレッド（最初のスレッド）に移行します。この操作がそれぞれのスレッドが終了するまで続けます。



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



索引

$\geq$, 51, 110

alert, 137, 157

all, 109

all <, 109

all =, 109

all >, 109

all but first of, 20

AND, 144, 147

and, 92

Animation, 129

any, 109

append, 21

atan, 124

atan2, 124

block variables, 64

break, 182

call, 41

case, 81

catch, 180

ceiling, 86

code, 75, 137

collapse, 111

columns, 169

columns of (), 31

combine, 27, 93, 161

console.log, 137

continuation, 177

continue, 182

Costumes, 9

csv, 6, 13, 36

csv of (), 34

dimensions of (), 30

distribution of (), 32

draggable?, 9

end thread, 184

Event Hat ブロック, 104

find first item, 26

flatten of (), 31

floor, 86

for ブロック, 54, 79, 182

getc, 99

import, 9, 36

() in front of (), 20

index, 140

initial slots, 111

inner product, 167

input list:, 40

iteration-composition, 180

JavaScript, 135

JavaScript extensions, 5

join, 100

json, 6, 13, 36

json of (), 34

keep, 25

Language, 5

layer, 126

Libraries, 9

lines of (), 33

list view, 18

Looks, 5

Magnification, 5

map, 23

match, 143, 156

multi-line, 75, 151

multiple inputs, 83

New, 9

NOT, 144

not, 91

object, 76
Open, 9
OR, 144, 155
or, 92
outer product, 174

parseInt, 145
pen trails, 37
pipe, 48, 173, 175, 176
product, 109

rank of (), 30
read-only, 67, 108
relabel, 11
replace, 141
reshape, 22
reverse of (), 33
ringify, 36, 58
rotation style, 8
round, 86
run, 40

Save, 9
Save As, 9
scene, 9
script pic, 11
search, 139
separator, 111
set slot, 101
shuffled of (), 33
sorted of (), 32
substring, 142
sum, 109
switch, 81

table view, 18
test, 138
text of (), 33
thread, 184
throw, 180
toString, 146

transient, 14
truncate, 86
turbo mode, 10

uniques of (), 32
unringify, 36
Upvar 変数, 17, 65
UTC, 161

variables, 66, 79

warp, 10
when slot, 101
with continuation, 177
with inputs, 40

XOR, 144, 148

yield, 184

Zoom, 5

イベント, 104, 114

エラトステネスのふるい, 26

オフライン版, 5

階乗, 28
回転行列, 168
角度, 124
カスタムブロック, 7, 49

協定世界時, 161
行列の積, 167

グローバル変数, 14

継続, 177

最小公倍数, 87
最大公約数, 28, 87
座標変換, 168

ジュークボックス, 7
集合, 96

真偽値, 91
真理値, 53, 78, 91

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, 54, 84
スプライトコラル, 7
スプライト変数, 15
スレッド, 184

正規表現, 138, 142
積集合, 96
ゼブラカラーリング, 11

素数, 26, 175

ターボモード, 10
大域変数, 14
単位行列, 174

定義ブロックのインポート, 63
定義ブロックのエクスポート, 63
定数, 49
デバッグ, 8, 84
転置行列, 31, 169

時計, 161

二進数, 144
日本語化, 5

排他的論理和 (XOR) , 144
配列, 22
バックグラウンド, 7
パレットエリア, 7
ハロ, 12

ビット演算, 144
ビット反転 (NOT) , 144
評価, 76, 78

フィボナッチ数列, 24
フォーマルパラメータ, 23, 41

プリミティブブロック, 7
ブロック変数, 64
プロトタイプ, 50, 54, 72, 73

並列実行, 165
変数ウォッチャー, 6, 14, 18, 19, 36

無名関数, 41

ユークリッドの互除法, 28

ラムダ関数, 41

リング, 23, 36, 41, 45, 57, 75, 76, 78

レイヤー, 126

ローカル変数, 15
ロケーションピンアイコン, 16
論理積 (AND) , 144
論理値, 91
論理和 (OR) , 144

ワードローブエリア, 7
ワープ, 10
和集合, 96