Snap! のこと

CoderDojo Aizu
齋藤文康

2021 年 12 月 28 日

Snap! Build Your Own Blocks(ver. 7.0.2) 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取り上げました。したがって、マニュアルのようにすべての事柄を扱っているわけではありません。サウンドおよびマルチメディア関係も扱っていません。

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。正しい内容になるように努めましたが、スクリプトを含め無保証です。

目 次

1　始め方

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

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

`https://github.com/jmoenig/Snap/releases/latest`

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

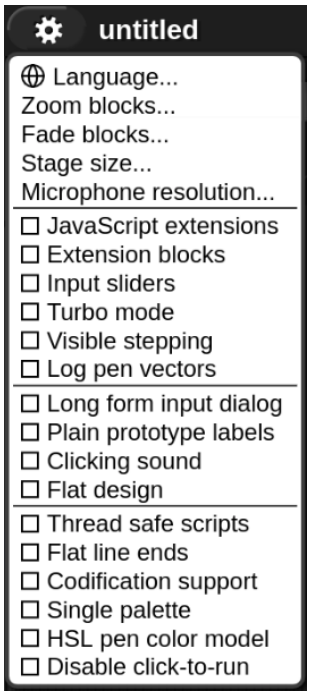
ダウンロード後、展開し、その中にある snap.html を Web ブラウザで開いてください。

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

オンライン版は、 

をクリックすれば使用できます。

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



日本語にするには、  のボタンをクリックすると表示される設定メニューの中で Language... をクリックして日本語を選べば変更することができます。

また、Zoom blocks... をクリックすると、ブロックの大きさを変更することができます。

JavaScript extensions がチェックされていると JavaScript ブロックが使用可能になります。

2　画面まわり

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

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

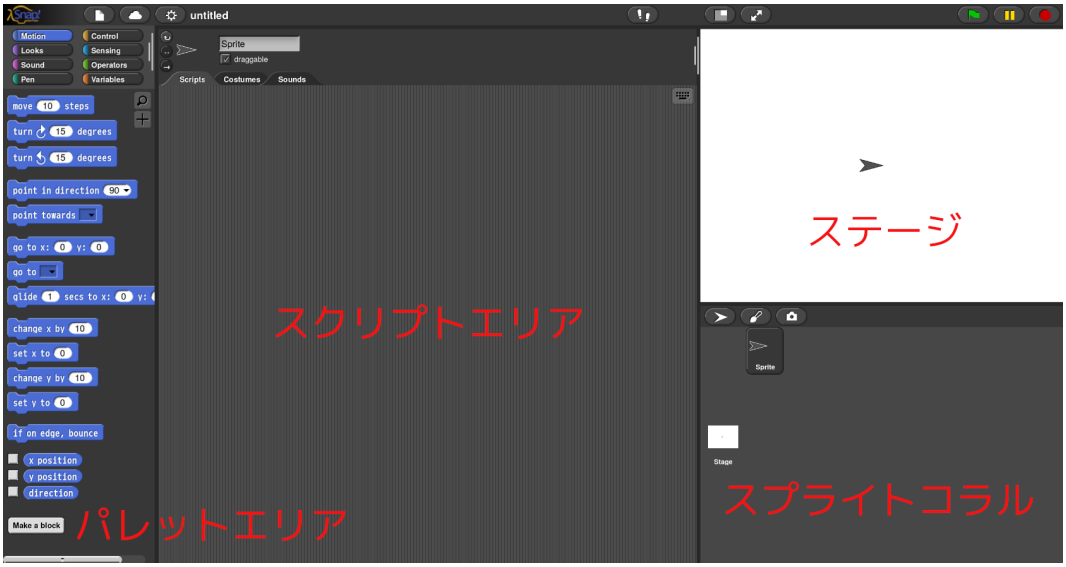
- 画像ファイル (.png, .jpeg など) の場合は、その時点で対象になっているスプライトのコスチュームとしてインポートし、ワードローブに入ります。
- サウンドファイル (.mp3 など) の場合は、その時点で対象になっているスプライトのサウンドとしてインポートし、ジュークボックスに入ります。
- テキストファイル (.txt) の場合は、変数を作成して読み込みます。
- .csv や .json のファイルは変数を作成し、リストとして読み込みます。

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追加します。このコスチュームの中心は、pen trails された時点の座標になります。

2.1.2 スプライトコラル

ここには使用するスプライトやステージが表示されます。 

をクリックすると新しいスプライトを生成できます。すでにあるスプライトを右クリックして出てくるメニューからコピーやクローンを作ることもできます。

2.1.3 スクリプトエリア

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

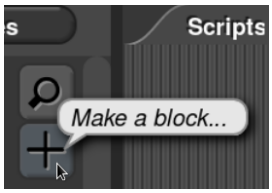
スクリプトエリアの右上には、  と  があります。

ブロックを組んでいて、いらないと思ってパレットエリアに移してしまったものが必要だった場合は、  をクリックすれば元に戻せます。これを使うと  になり、その変更を元に戻すことができます。ただし、これはブロックのドロップ（どこに移動させたか）に関することだけらしくて、入力スロットに設定した値を元に戻すようなことはできないようです。

 は、キーボードを使ってスクリプトを作成するモードへのスイッチです。

2.1.4 パレットエリア

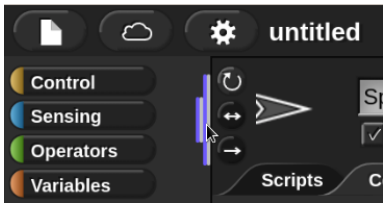
ここからスクリプト作成のためのブロックを持ってきます。要らなくなったブロックを戻す場所でもあります。上部にある 8 個のボタンから選択して、使用する機能のブロックを表示します。



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

2.2　エリアの大きさ

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



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

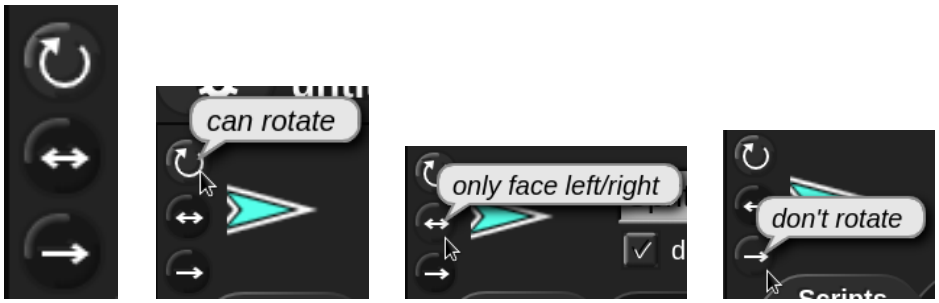
2.3　実行に関するボタン

ステージエリアの上には  のボタンがあります。これは 

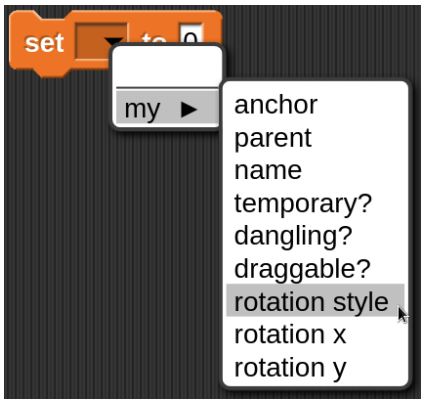
に接続されたスクリプトを実行するボタンです。  はスクリプトの実行を終了させるボタンです。  はスクリプトの実行を一時停止させるボタンです。  のボタンをクリックすると実行中のブロックをハイライトさせてゆっくり実行させたりできます。  のマウスポインターが置かれているところをドラッグすると実行のスピードが調整できます。一番左だと一動作ごとのステップ実行になります。デバッグの時に使えます。  のクリックで実行再開です。スクリプトを止めて確認したいところに 

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、 

です。スクリプトで設定するのが、 

です。こちらは、true か false で設定します。


をクリックすると右のメニューが出てきます。

Notes... にはプロジェクトの覚書、注釈が書けます。

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

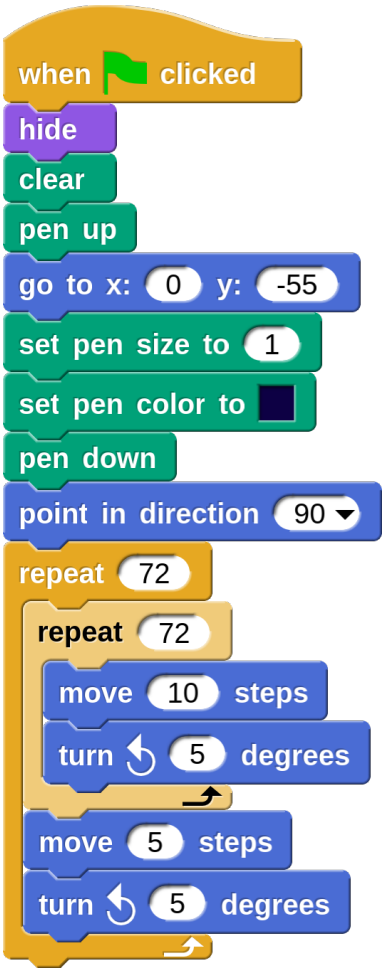
scene は、プロジェクト内にサブプロジェクトを置くものです。新規作成または既存のプロジェクトをオープンしてサブプロジェクトとして加えます。 

で次のプロジェクトに移行することができます。

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

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

次を実行すると、かなり時間がかかります。

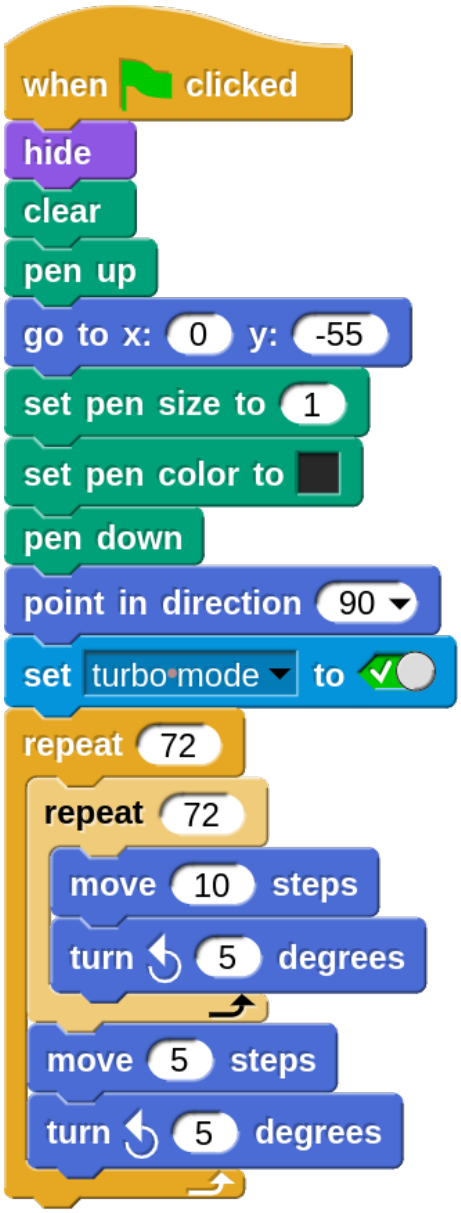


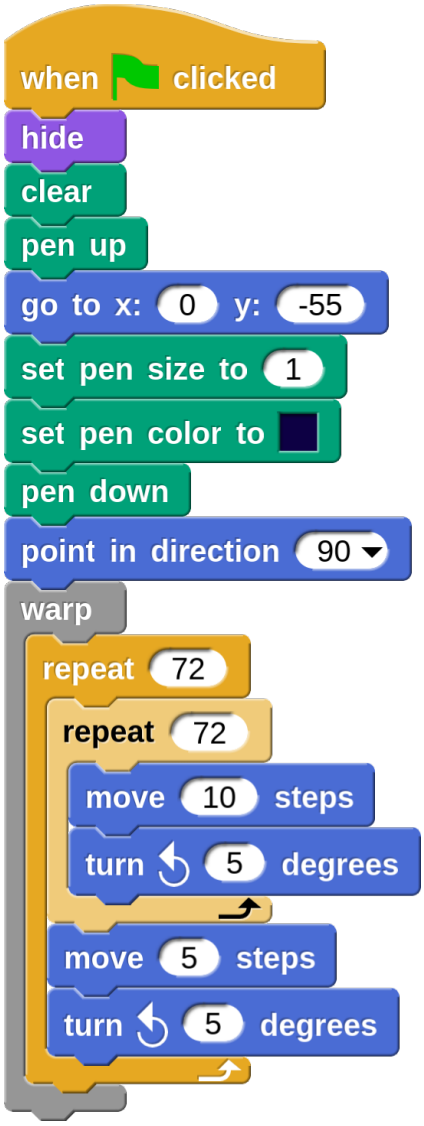
Shift を押しながら  のボタンをクリックすると  (ターボモード) になり、これをクリックすると描画のスピードが速くなります。

Sensing パレットに  ブロックがあります。この六角形の部分をクリックすることで、  ターボモードをオンにしたり  オフにすることができます。スクリプト内で自在にターボモードの切り替えができます。

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

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

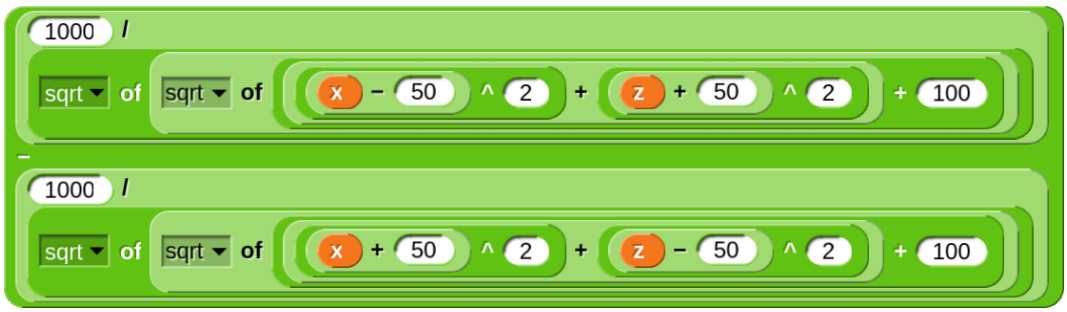




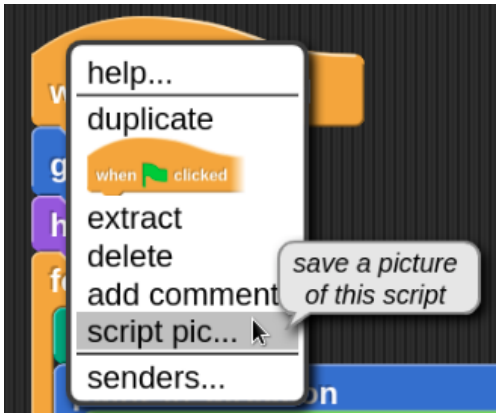
2.4 ブロック表示

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

このような長い式を Snap! でスクリプトにすると、

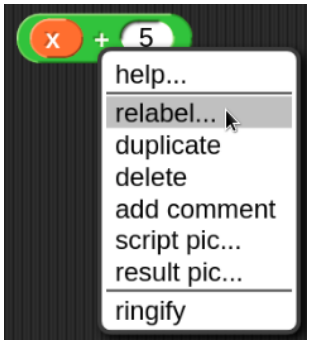


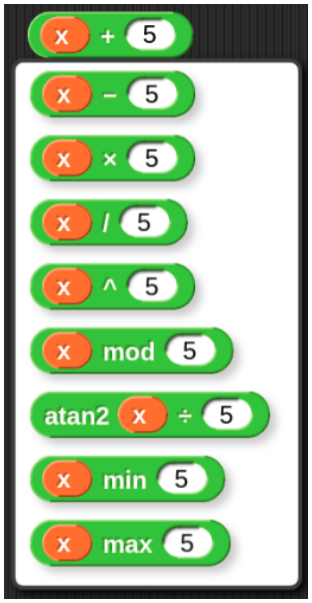
となります。長くなった場合は、自動的に折り畳まれます。また、同じパレットのブロックが重なった場合は、色合いが交互に変化して表示されます。ゼブラカラーリングだそうです。

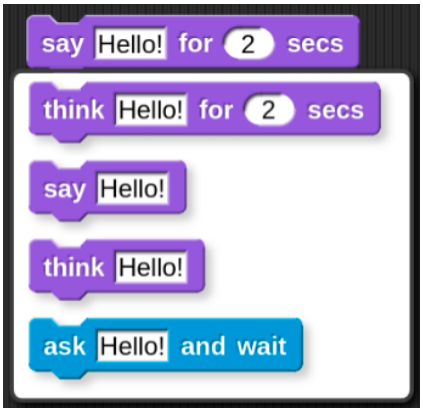


スクリプトをプリンターで印刷したい場合は、スクリプトを右クリックすると、script pic ... で、画像ファイルとしてダウンロードフォルダーへエクスポートされます。
ファイル名はプロジェクト名 + script pic + (番号) + 画像ファイルを表す拡張子になります。

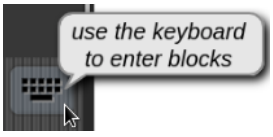
ブロックを組んでいて、間違えたとか違う種類の方にしたい時があります。そういう時はそのブロックを右クリックすると出てくるメニューから、relabel... をクリックすると欲しいものが得られる場合があります。Operators の場合はパレットに無いブロックも使えたりします。





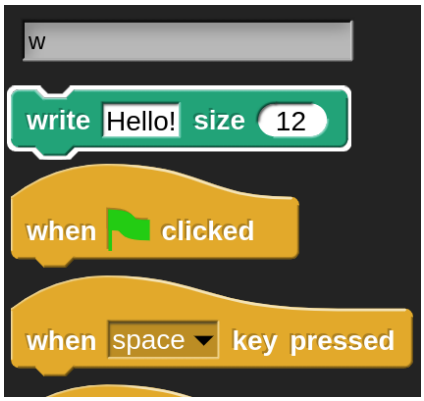


3　キーボード入力

スクリプトエリアには  があります。これをクリックするか、Shift+↵ でキーボードを使ってスクリプトを作成できるモードになります。 

をクリックしてください。すると、スクリプトエリアに白い線が点滅されます。すでにいくつかスクリプトがある場合は、Tab を押すと別のスクリプトのところへ移動します。

2 つの計算結果を表示するスクリプトを作ってみます。

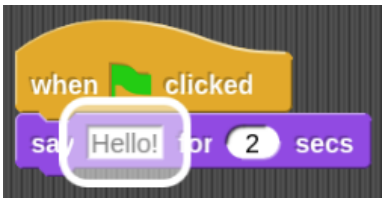


スクリプトが何もない状態から始めます。

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 [↵] 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。

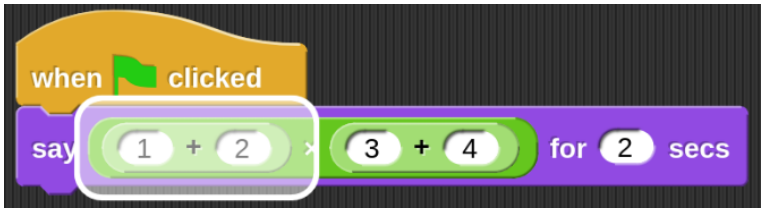


ここで [↵] を押すと、ハロが消えて初期値としての「Hello!」を修正または別なテキストを入力することができます。[↵] を押してハロが出ている状態で文字を打ち込むと、ブロックや変数を候補として出してくれます。数字や「( ) + - * / < = >」を打ち込むと、数式の入力ができます。

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

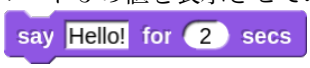
16/4/2

16/(4/2)



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

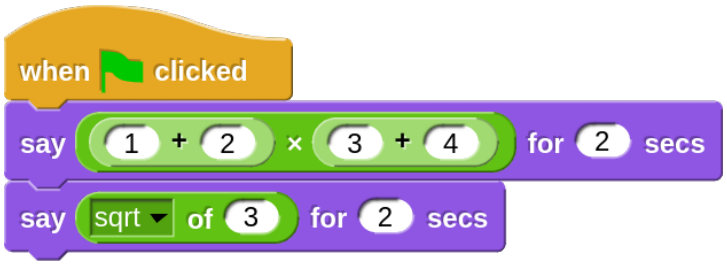
次に、ルート 3 の値を表示させてみます。ルートは sqrt で求めます。

また、 

を出してください。

say の最初の入力スロットの部分で、[o] (of の「o」です) と打ちこんで 

を選びます。この場合は sqrt になっているのでこのままでいいですが、別な演算を選ぶ場合は [↵] を押すと選択肢が表示されます。後は 3 をセットすれば終了です。できたスクリプトは [Control]+[Shift]+[↵] で実行できます。



どこかをクリックするか Esc などでキーボード入力モードから抜けます。

4 変数

変数を作成するにはいくつか方法があります。

- Make variables をクリックする。
- 

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

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

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

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

[Make a variable] をクリックすると、

が出ますから for all sprites を選んで、varAll

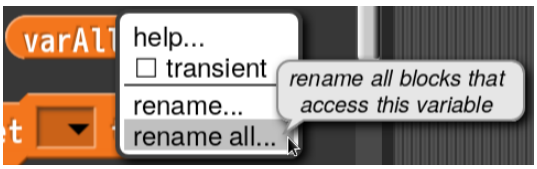
という変数を作ります。そうすると、パレットエリアに が表示されます。こうして作られた変数は全部のスプライトで使用できます。このようにどこからでも使用できる変数をグローバル変数とか大域変数と言います。

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

変数を削除する場合は、 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。

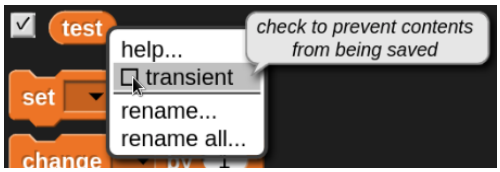
名前を変更する場合はその変数のところを右クリックして、

か



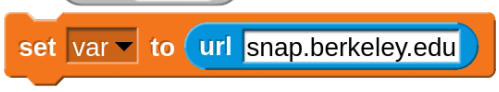
で変更します。all の方を選ぶと、この変数を使用しているすべての個所を変更します。

Make a variable で作成した変数は、右クリックで

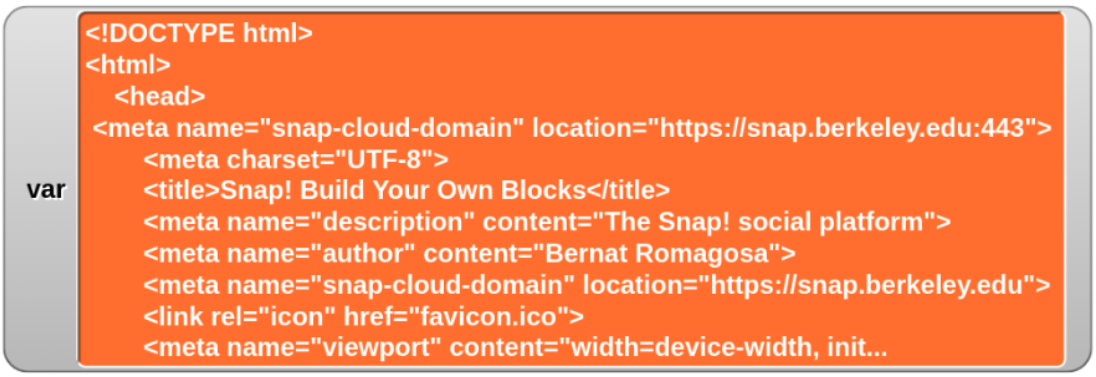


transient のオプションが表示されます。これはプロジェクトの保存の時にこの変数の値を保存させないためのものです。以下は、transient の効果を示すものです。

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



をスクリプトエリアに置きます。これを実行させない状態で save します。すると、ファイルの容量は 11.5KB でした。これを実行すると、変数 var に Snap! のホームページの html ファイルが入ります。（オンラインの Snap! を使用のこと）



この状態で save します。すると、ファイルの容量は 86.4KB でした。この保存したプロジェクトを改めて Open... で読み込むと、保存された時の変数の値になっています。つまり、何もしないと変数が値ごと保存されてプロジェクトの容量を大きくするということです。transient をチェックして save すると、ファイルの容量は 24.3KB 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 の値は初期値になっています。これが transient の機能です。

グローバル変数はどこからでも使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、

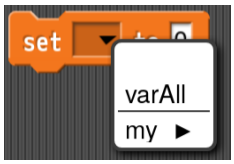
for this sprite only を選んで varSprite を作成します。すると、パレットエリアに

が表示されます。varSprite という名前の左にロケーションピンアイコンが表示されて、これがこのスプライト専用の変数であることを表

します。この変数はこのスプライトのスクリプトエリア内ならばどのスクリプトからも使用することができます。このスプライト限定になりますがカスタムブロック内で使用することもできます。

をクリックして、 

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

を使いますが、 

varSprite は変数名リストに出てきません。

4.3 script variables スクリプト変数

で、その下に接続された範囲だけで有効なスクリプト変数が使えるようになります。変数名は 

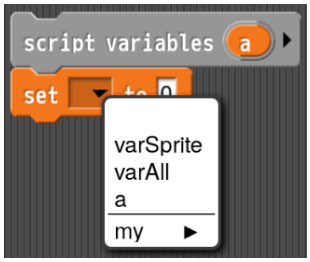
a のところをクリックすると変えられます。

a の隣りにある右向きの三角をクリックすると変数を追加できます。

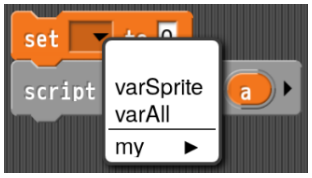
左向きの三角をクリックすると削除できます。

スクリプト変数は 

をスクリプトエリアに持ってくるだけでは使えません。

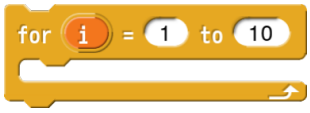


のように接続されてからでないと使えません。



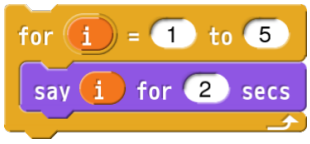
のようにその前でも有効になりません。

4.4 for ループ変数

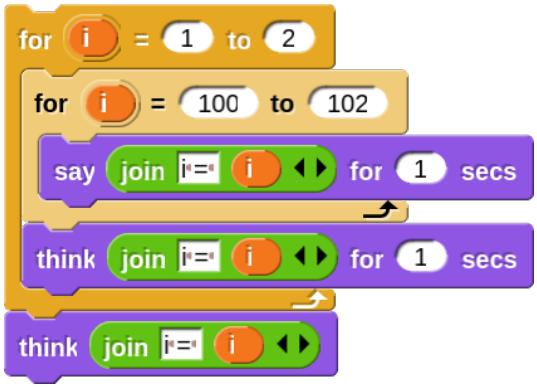


このようにあらかじめ用意されている変数があります。

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



この例は、i の値を 1 から 5 に 1 つずつ増やしながら C 型ブロック内のスクリプトを実行します。1、2、3、4、5 と表示します。



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

4.5 リスト処理用ブロック

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

で aList という変数を作成すると、ステージエリアに 

が表示されます。 

の左向きの三角をクリックして 

とすると、空のリ

ストができます。 

の右向きの三角をクリックして 

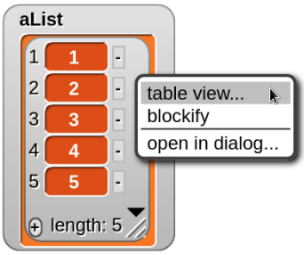
で値を入れてやると (左端の入力スロットに 1 を入れて、Tab キーを押すと次の入力スロットに移れます)、aList は要素を持つリストになり、ステージエリアに 

が表示されます。

を使うと、 

にすることができます。 

で同じことができます。

変数ウォッチャーのリスト表示エリア内を右クリックして 

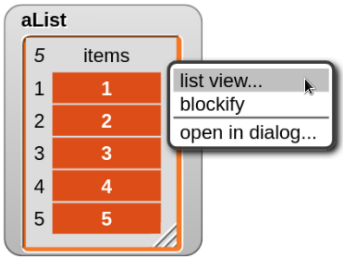


table view... を選択すると
に変更することができます。また右クリックして list view... を選択すると元に戻ります。

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。

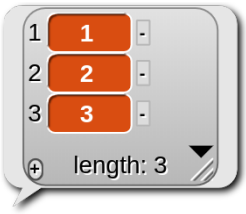
変数ウォッチャーの内部を右クリックして、blockify をクリックすると、リストブロックとして取り出すことができます。

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。

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、 

や 

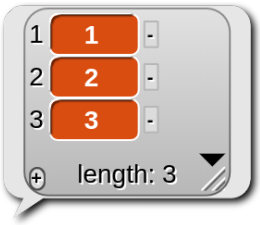


のように。

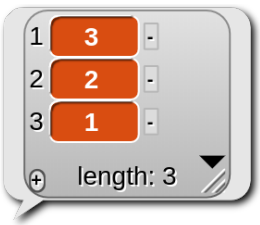
なお list view では、一度に扱える要素数は 100 までになっています。次の 101 から 200 の要素に移るには、下向きの三角をクリックすると出てくる範囲から選びます。

4.5.1 numbers form ( ) to ( )

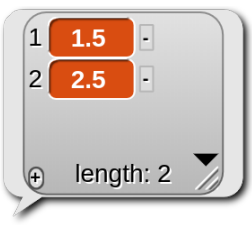
これは指定された範囲のリストをリポートするものです。



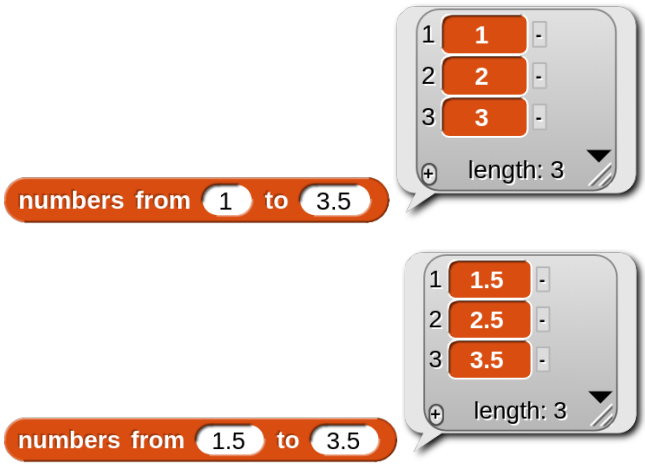
指定された範囲内で 1 ずつ増加するリストをリポート



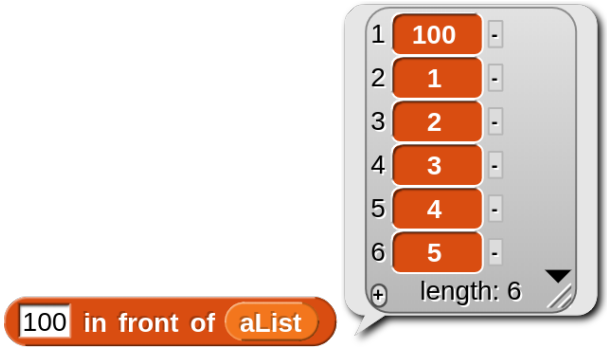
指定された範囲内で 1 ずつ減少するリストをリポート



1 ずつの増加または減少なので指定された値が含まれないこともあります。



4.5.2 ( ) in front of ( )



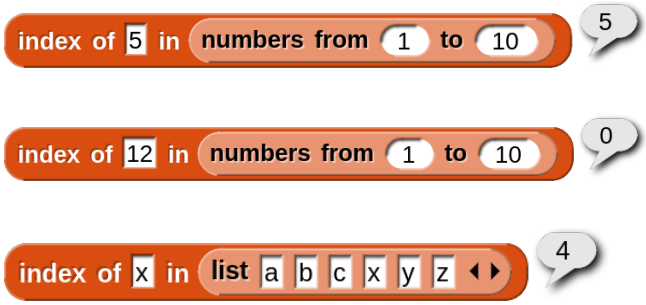
これは、指定された値を指定されたリストの先頭に挿入したリストをリポートします。指定されたリスト自体は変更しません。

4.5.3 all but first of ( )

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

4.5.4 index of ( ) in ( )

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



4.5.5 append ( ) ( )

これは、指定されたリストの要素を繋げたリストをリポートします。

4.5.6 map ( ) over ( )

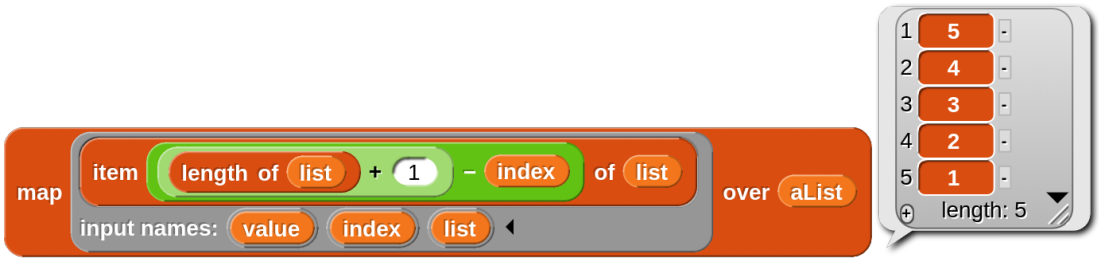
これは、指定されたリストの各要素に対して指定された演算を行ったリストをリポートします。指定されたリスト自体は変更しません。

「+」の左側の入力スロットが空になっています。ここに aList の要素が順番に入って、計算された結果をリストとしてリポートします。空の場所は  でもかまいません。

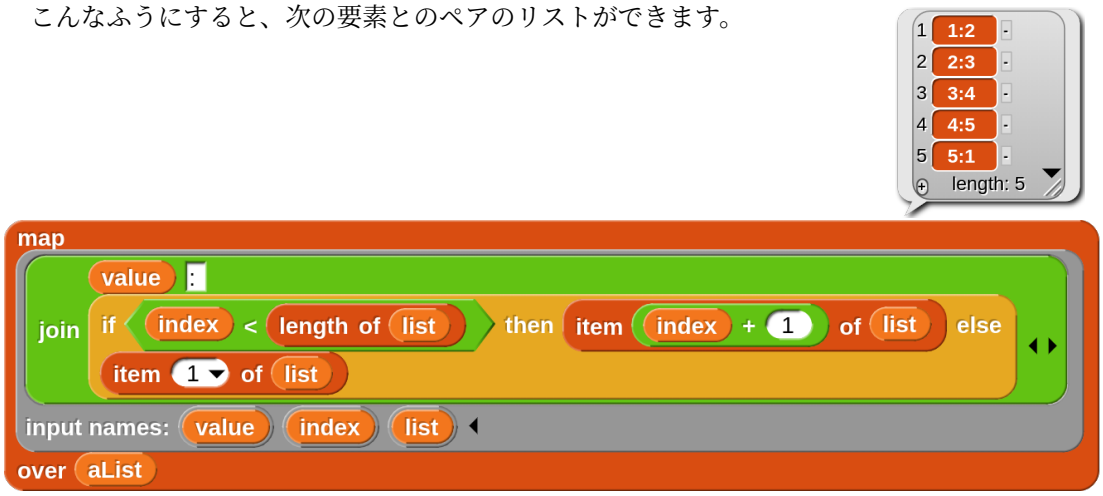
緑の演算子ブロックの外側の灰色の部分をリングと言います。右端の右向きの三角をクリックすると、フォーマルパラメーターが出てきます。パラメーターとは、値を受け取るための変数のようなものです。 map のフォーマルパラメーターには機能が設定されています。

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

2 番目のフォーマルパラメーターは index 、3 番目が list です。これを使うと逆順のリストが作れたりします。この場合 list は aList を表します。



こんなふうにすると、次の要素とのペアのリストができます。



例を示すためにひねり出したもので、あまり意味はありません。あまり複雑にして分かりづらくなるようならば、素直に for ループで作ったほうがいいでしょう。

リストの値を使用しない使い方もあります。

4.5.7 keep items ( ) from ( )

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

4.5.8 find first item ( ) in ( )

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

4.5.9 combine ( ) using ( )

は、指定されたリストの要素に対して指定された演算を行った結果をリポートします。指定されたリスト自体は変更しません。この場合は、

を行うのと同じ内容になります。combine では、

の演算がおもに使われるようです。

4.5.10 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。

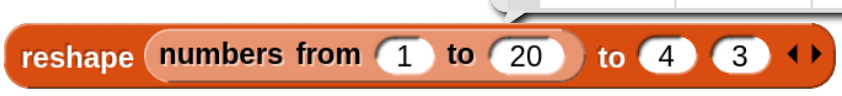
4.6 reshape ( ) to ( ) ( )

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

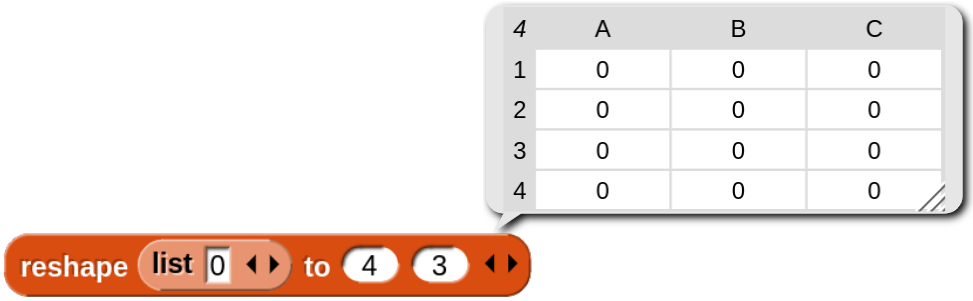
4	A	B	C
1	1	2	3
2	4	5	6
3	7	8	9
4	10	11	12

reshape ブロックを使って、一列のリストと行数と列数を指定することで希望の形の配列を作成することができます。ただし、指定した行数 × 列数個以上のリストの部分は無視されます。

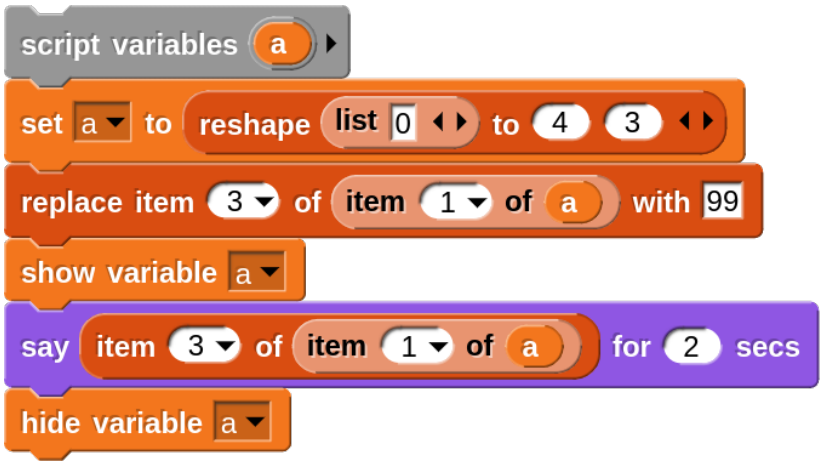
4	A	B	C
1	1	2	3
2	4	5	6
3	7	8	9
4	10	11	12



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。



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。

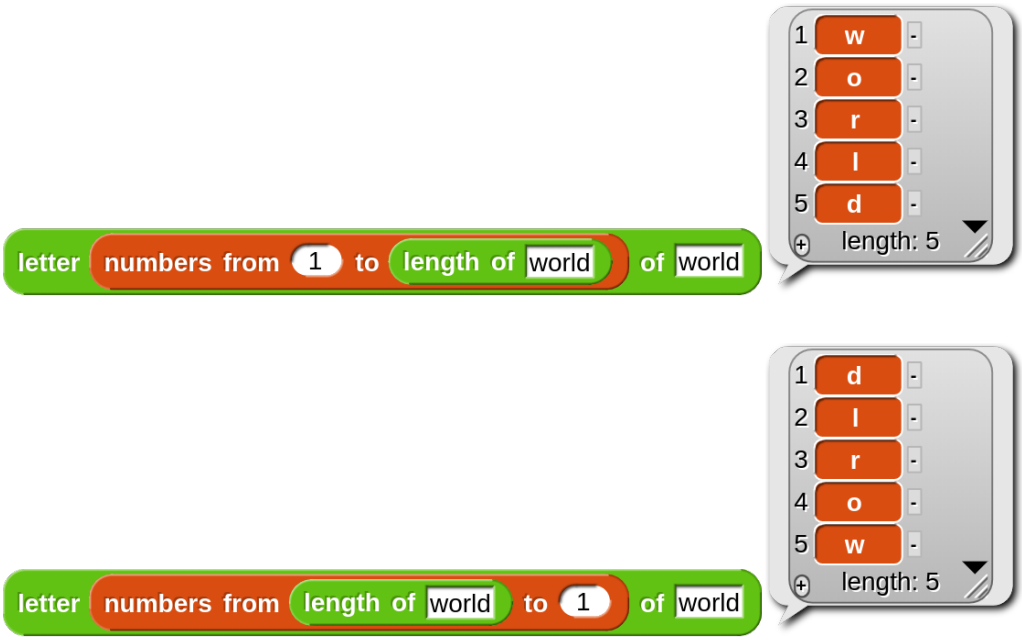


4.7　リストの演算

Snap! では、APL 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。

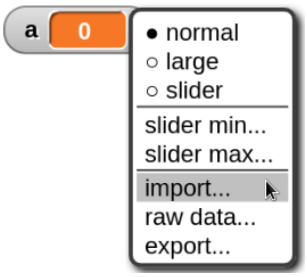
リスト同士で演算をすることもできます。いずれも同じインデックスの要素同士を演算します。要素数が合わない場合は対応する部分だけで行います。

リストに文字列の各要素を入れることもできます。



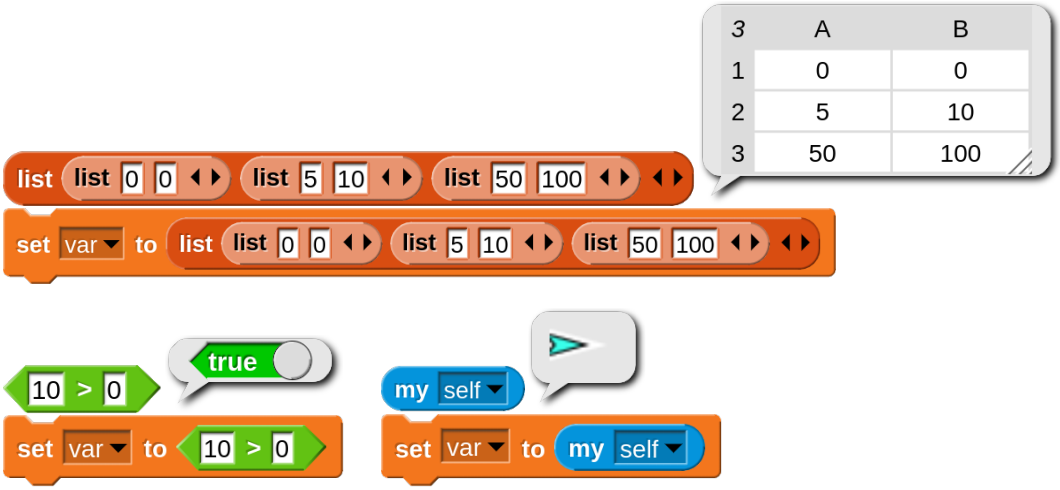
4.8 変数に入れられるもの

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



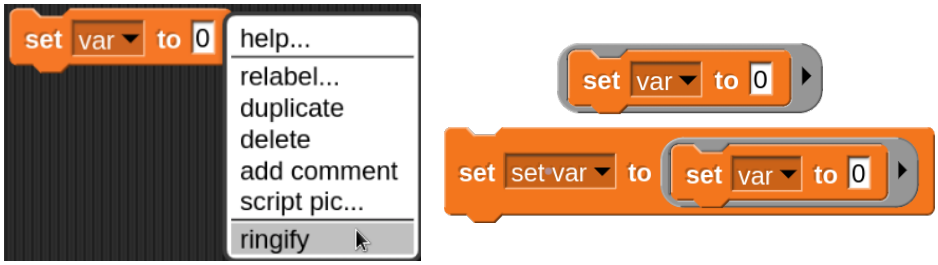
「import...」で、指定したファイルの内容を取り込むことができます。.csv や .json のファイルの場合は、そのデータ形式に従ってリストを作成します。.csv ファイルではカンマと改行をセパレーターとした要素をリストにします。単なるテキストデータとして取り込みたい場合は、「raw data...」を使います。

Snap! の変数には、値をリポートするものは入れられるようです。

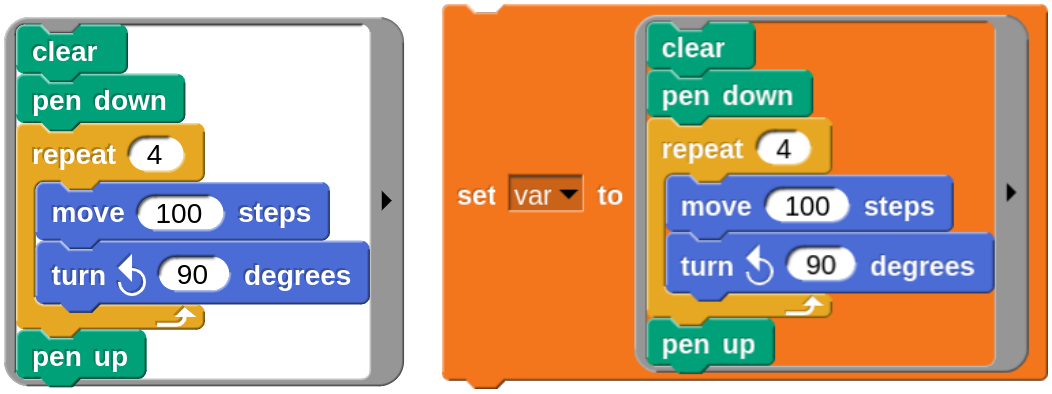


ブロックをリングに入れてやるとリポーターブロックにすることができます。ブロックそのものをリポートするものです。対象のブロックのところで右クリックしてメニューから ringify を選べ

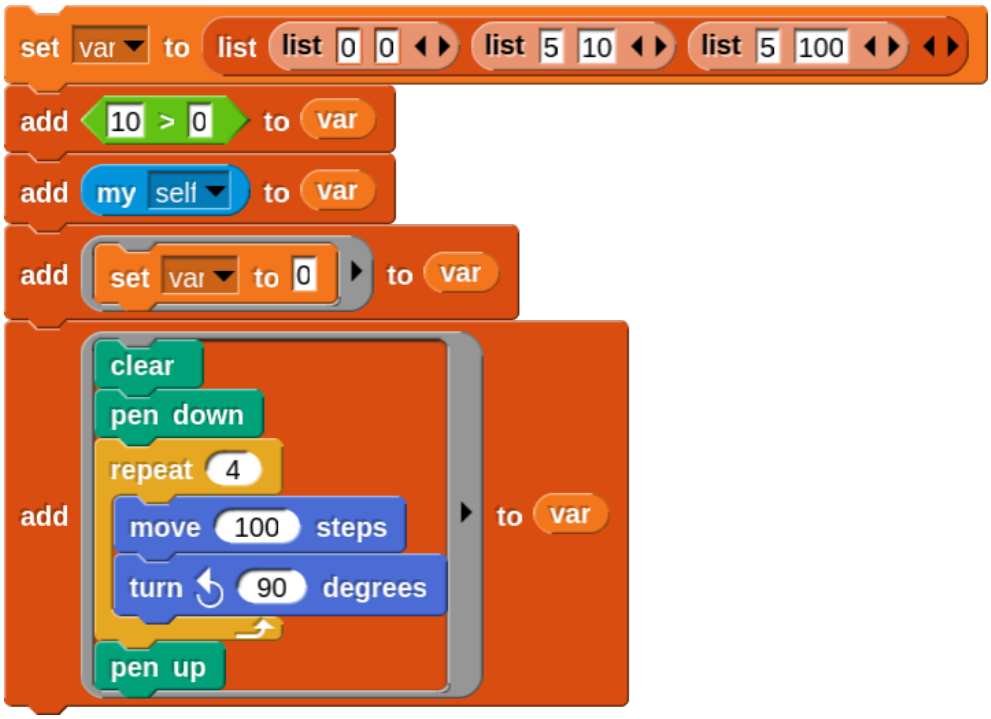
ばリングで囲むことができます。リングを外すには unringify をクリックします。



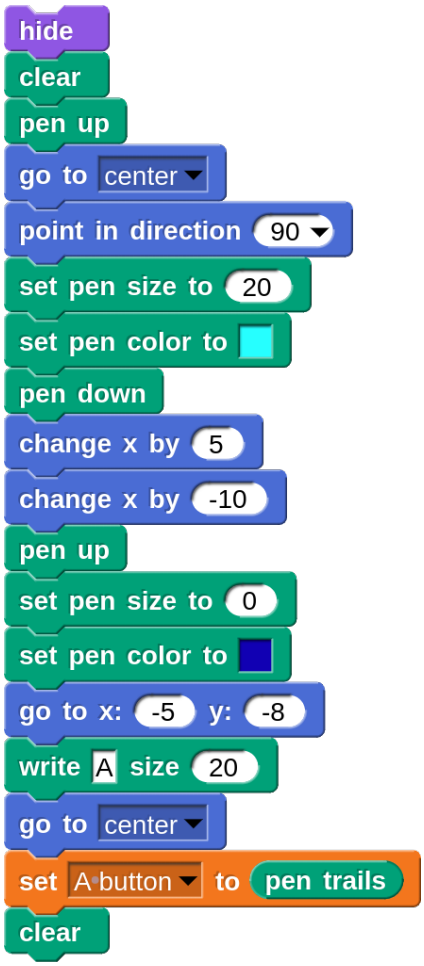
１つのブロックだけではなく、スクリプトもまとめてリングで囲むことができます。それを変数に入れることができます。

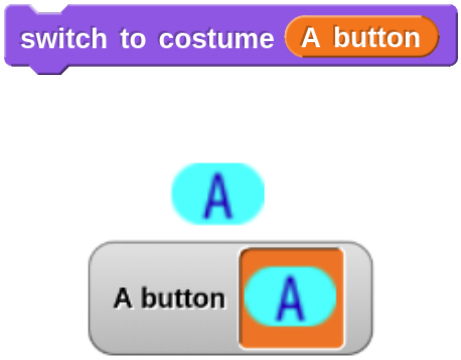


リストに入れることもできます。



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



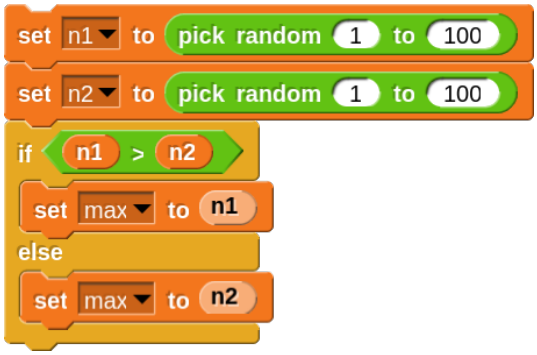


この例は、ステージに pen でボタンを描いてその軌跡を変数にセットします。そして、それを switch to costume でコスチュームにします。これはコスチュームに文字をセットする１つの方法です。
なお、描き終わりの座標がコスチューム、スプライトの中心になるので go to center を入れて調整しています。

5　Control 制御

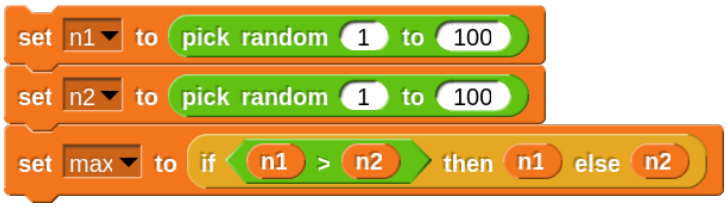
Snap! で使用できる制御ブロックについて。

5.1　リポーターの if then else

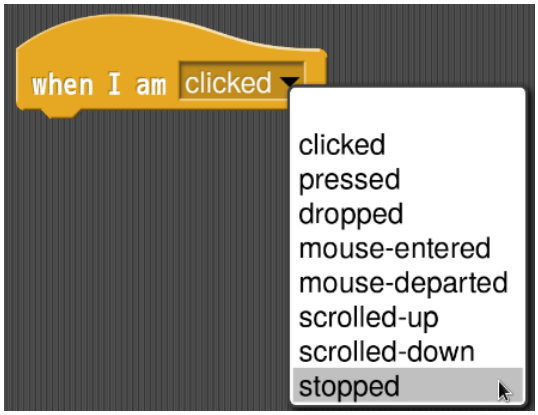


2 つの数のうち大きいほうを max という変数にセットするスクリプトを if else で作ると、こうなります。n1、n2、max の変数を使用します。

これをリポーターを使うとこんなふうにすることができます。if リポーターは、n1 か n2 の大きい方の値をリポートし、それが max にセットされます。



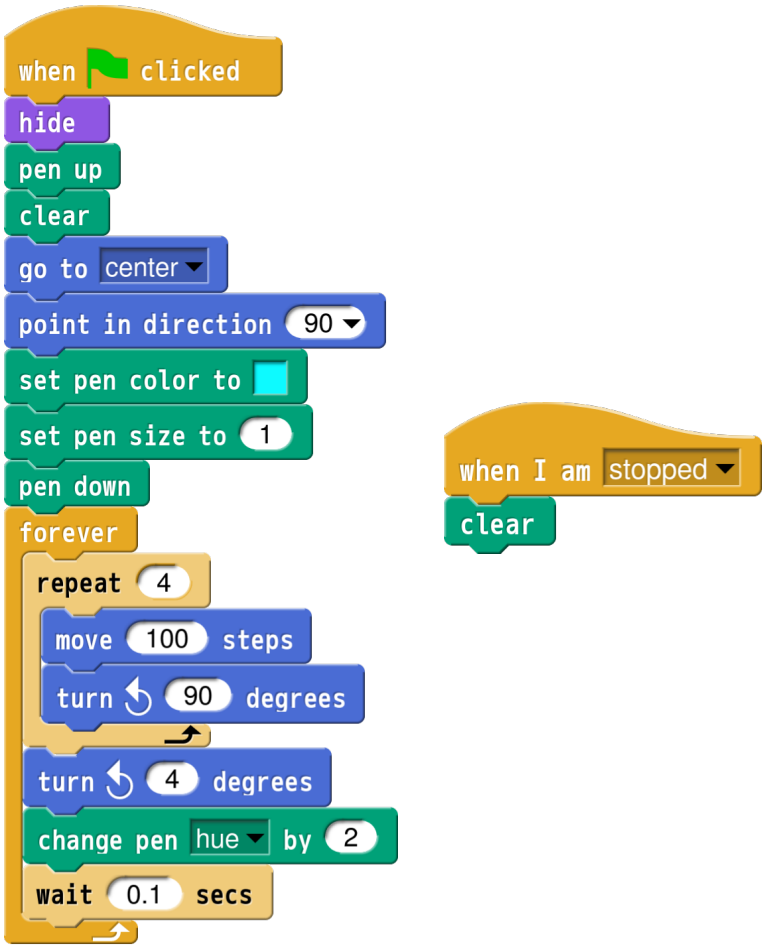
5.2 stop ボタンがクリックされた時の終了処理



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

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

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



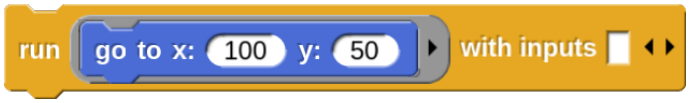
5.3 run ブロック

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

のように。

たとえば、 

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



のようになります。 go to x: y: の各入力スロットを空にして、

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

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

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

5.4 call ブロック

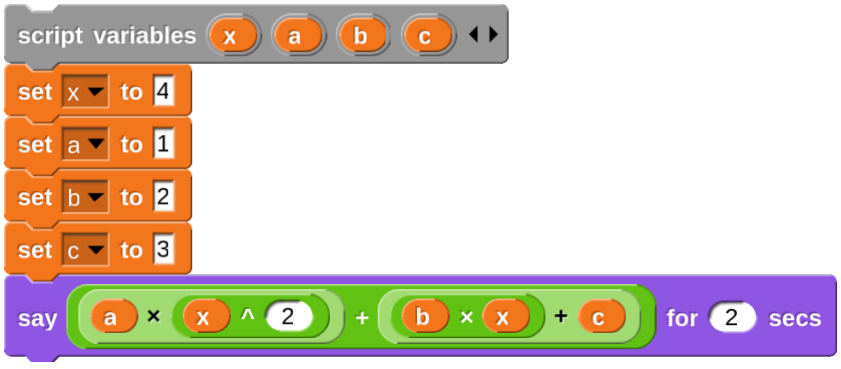
run ブロックに入れられるのが値をリポートしない実行 (Command) ブロックだったのに対して、call ブロックに入れられるのは、実行または評価することによって何らかの値または true か false をリポートするブロックです。call ブロックは、得られた値をリポートします。

call ブロックを使ってちょっとした関数のようなものが作れます。ブロックを定義するまでもないものなら、これで間に合います。

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

この式をブロックにすると、 

で表され、スクリプトで確かめられるようにするとこうなります。



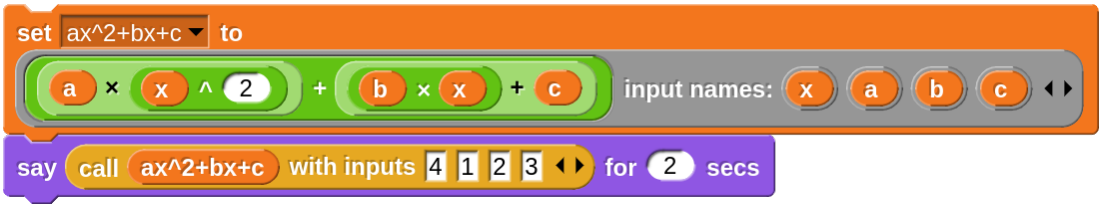
変数の入力スロットを空にして、式をリングで囲みます。

リングの端の三角をクリックしてフォーマルパラメーターを出します。

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

それを式の入力スロットに入れれば、call を使った無名関数になります。with inputs で、順にそれぞれの変数の値を指定します。名前がないのでこの場所限りの使用になります。

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

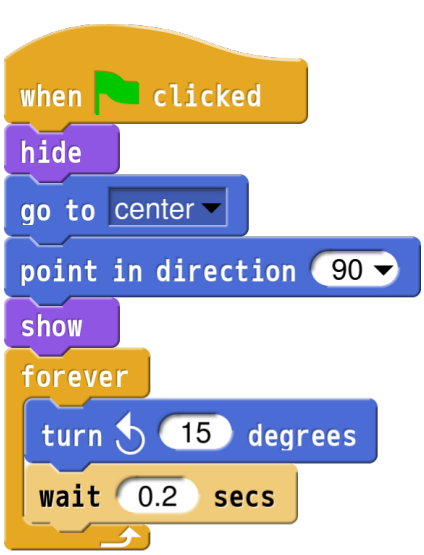


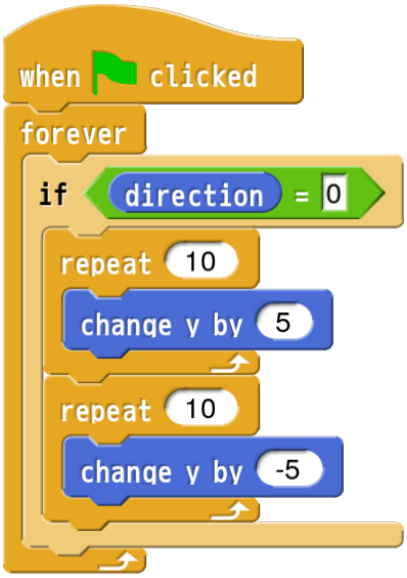
5.5 launch ブロック

launch ブロックは指定されたスクリプトを並列で実行するものです。並列でというのは、launch ブロック内のスクリプトの実行が終わるのを待たずに、それをしながら同時に launch ブロックの次のスクリプトの実行に移ります。

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

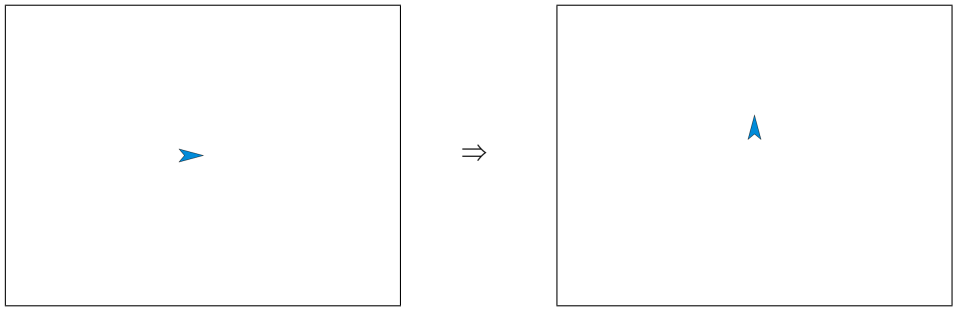
並列処理は、普通次のようにして 2 つのスクリプトを同時に実行します。



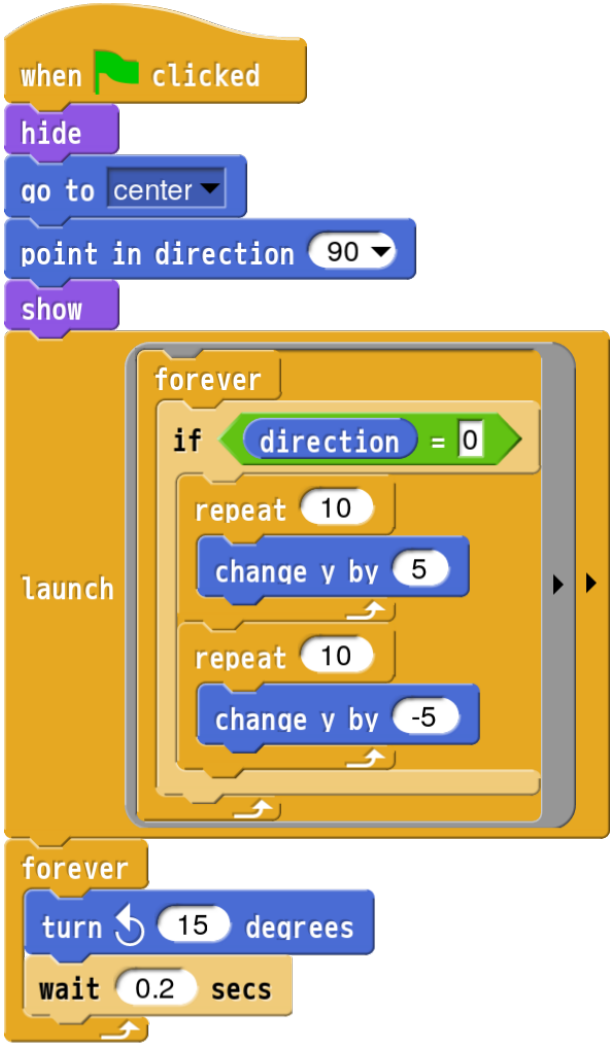


左のスクリプトで、ずっと回転させる動作を行います。

右のスクリプトで、上を向いた時だけジャンプする動作を行います。

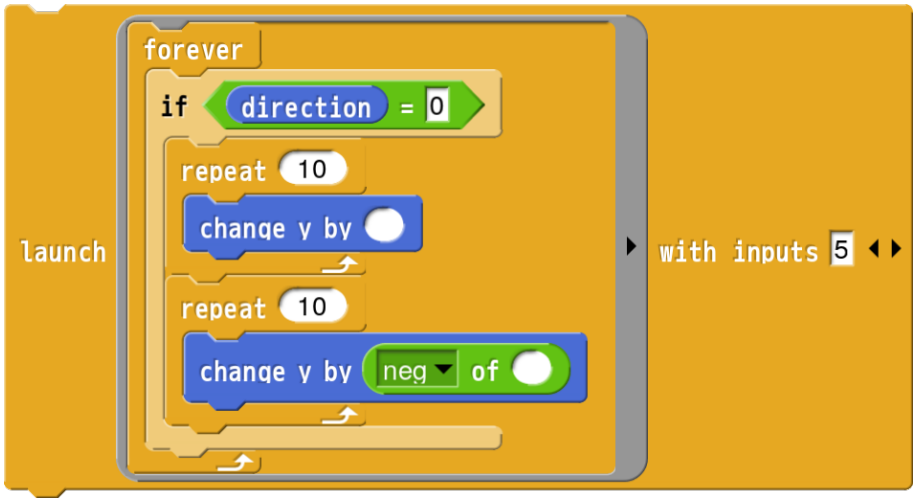


launch を使うことで１つのスクリプトで実行することができます。

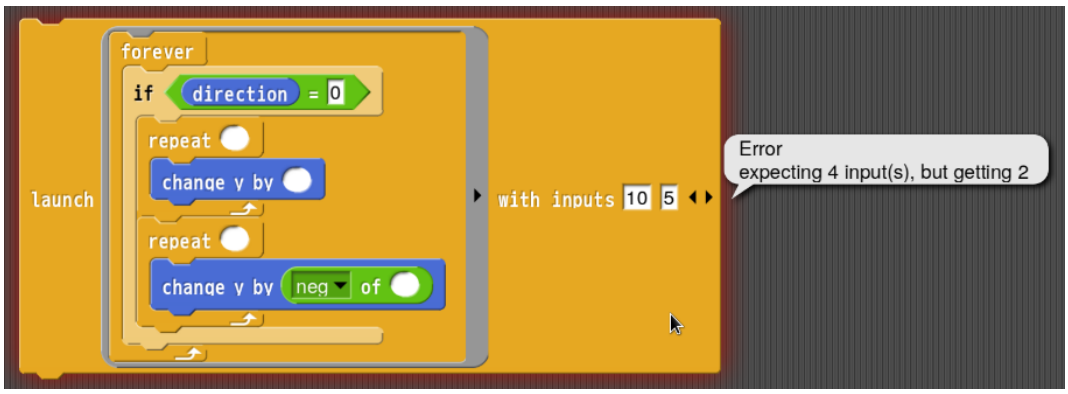


launch の実行終了を待たずに次のスクリプトに進むので、launch 内のスクリプトが forever 終わることがなくても大丈夫です。

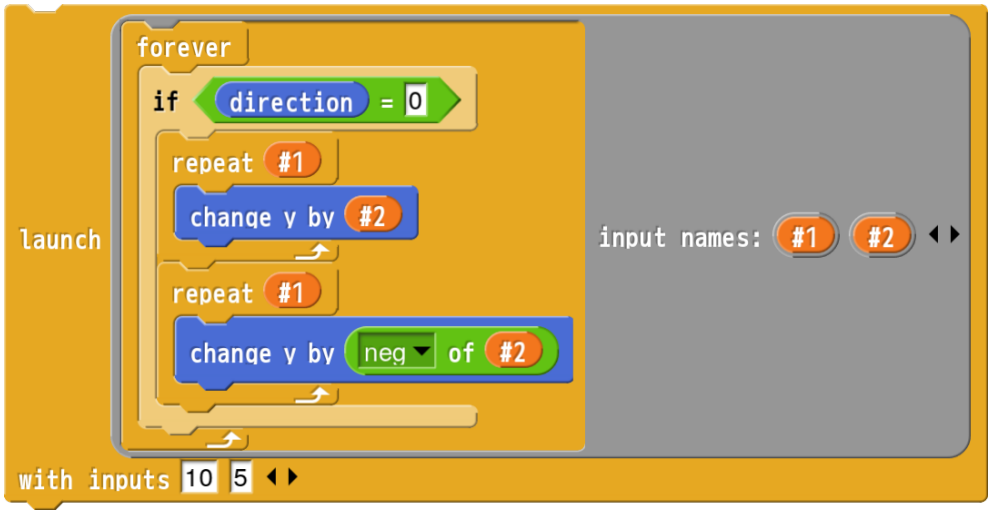
launch の右端の右向き三角をクリックすると、入力値を設定することができます。 y の値の設定を空（空白ではなく）にして、入力値をひとつだけ設定すると、その入力値がすべての空の部分の値になります。



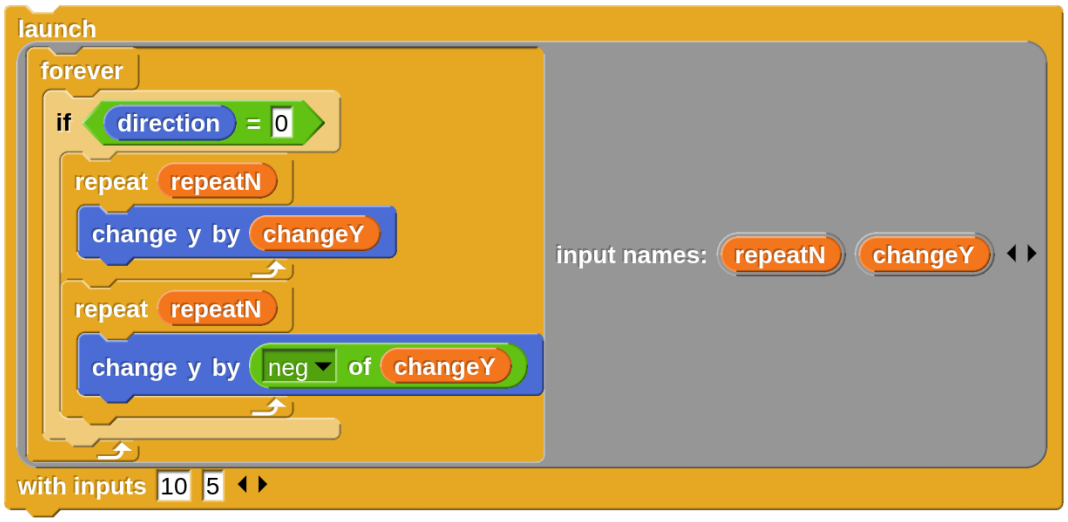
2 個以上の入力値を設定した場合、空の入力スロットの個数が合わないとエラーになります。



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

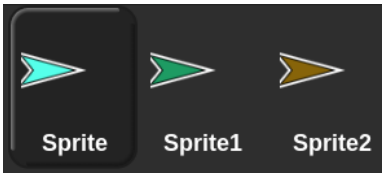


フォーマルパラメーターをクリックして変数名を変更するとスクリプトが分かりやすくなります。



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

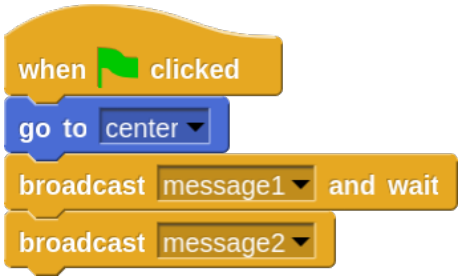
5.6 broadcast ブロック, tell to ブロック

次はスプライトを３個使います。 

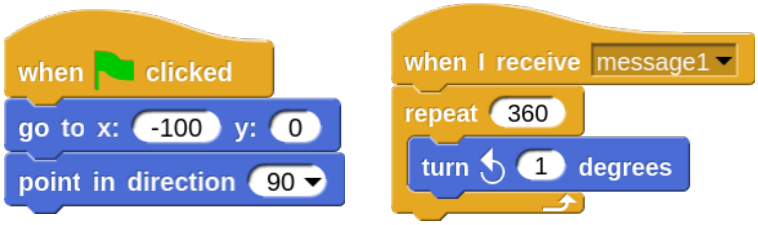
を用意してください。

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

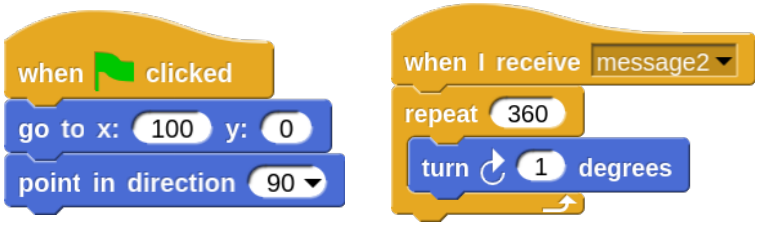
Sprite 用



Sprite1 用

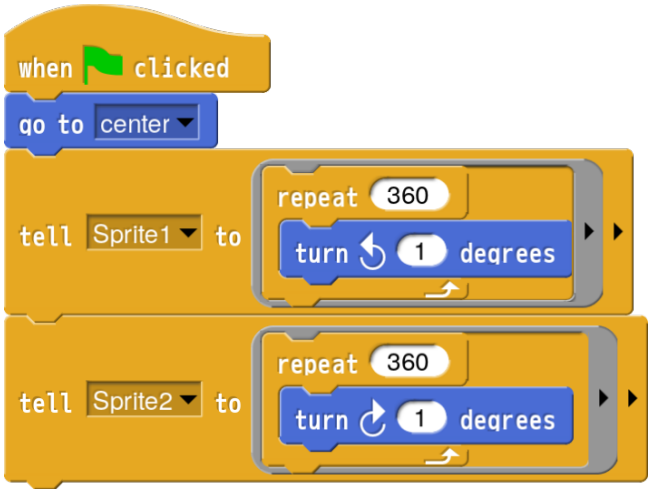


Sprite2 用



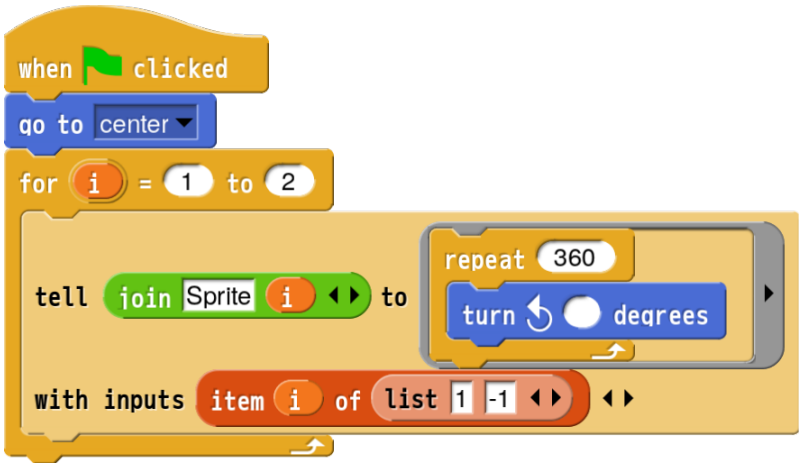
のブロックを使用すると broadcast を使わなくても Sprite から Sprite1, Sprite2 を直接操作することができます。

Sprite 用



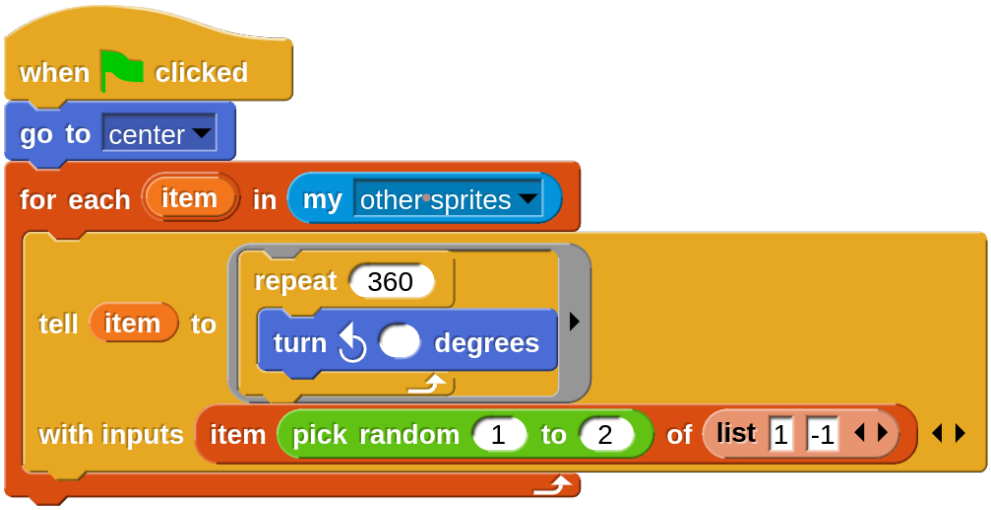
もしも Sprite1 と Sprite2 を同時に動かすのならば、tell Sprit1 のブロックを launch で囲みます。

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



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

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

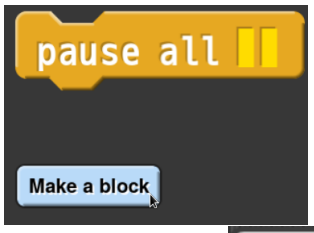


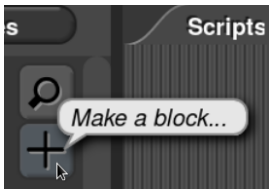
6　ブロックを作成する

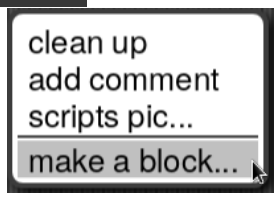
Snap! では、ローカル変数が使え、値をリポートすることもできるので、カスタムブロック (ユーザー定義ブロック) が作りやすくなりました。

6.1　>= ブロック

最初の例として、>= のブロックを作ってみます。
これがあると、`var > 0 or var = 0` ではなく、`var >= 0` とできるのですっきりします。
ブロック内部では `var > 0 or var = 0` を行っているので見た目だけのことですが。

ブロックを作成するには、パレットエリア内にある 

、


、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。

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

カテゴリーで Other「その他」というものを選ぶと、置き場所は Variables のところになります。

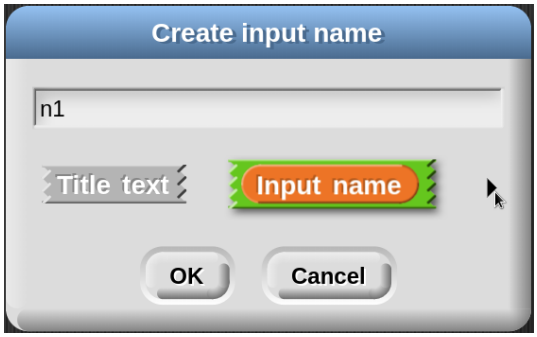
Command は、値をリポートしないブロックです。 Reporter は、なんらかの値をリポートします。 Predicate は、述語と訳されます。true か false をリポートします。それぞれの形が、できたブロックの使われ方を示しています。

ブロック名入力欄に >= を入れて、上の欄でボタン Operators を、下の段で Predicate を選択してください。 Operators を選択することはパレットの種類を決めることであり、置き場所を決めることでもあります。プリミティブの 「>」 ブロックが Operators に置いてあるのでそこにします。 Predicate は形から分かるように、Control コントロールブロックの条件式のところなどで使用されて true または false を返すためのものです。 Ok をクリックするとブロックエディターが表示されます。

定義の先頭の 

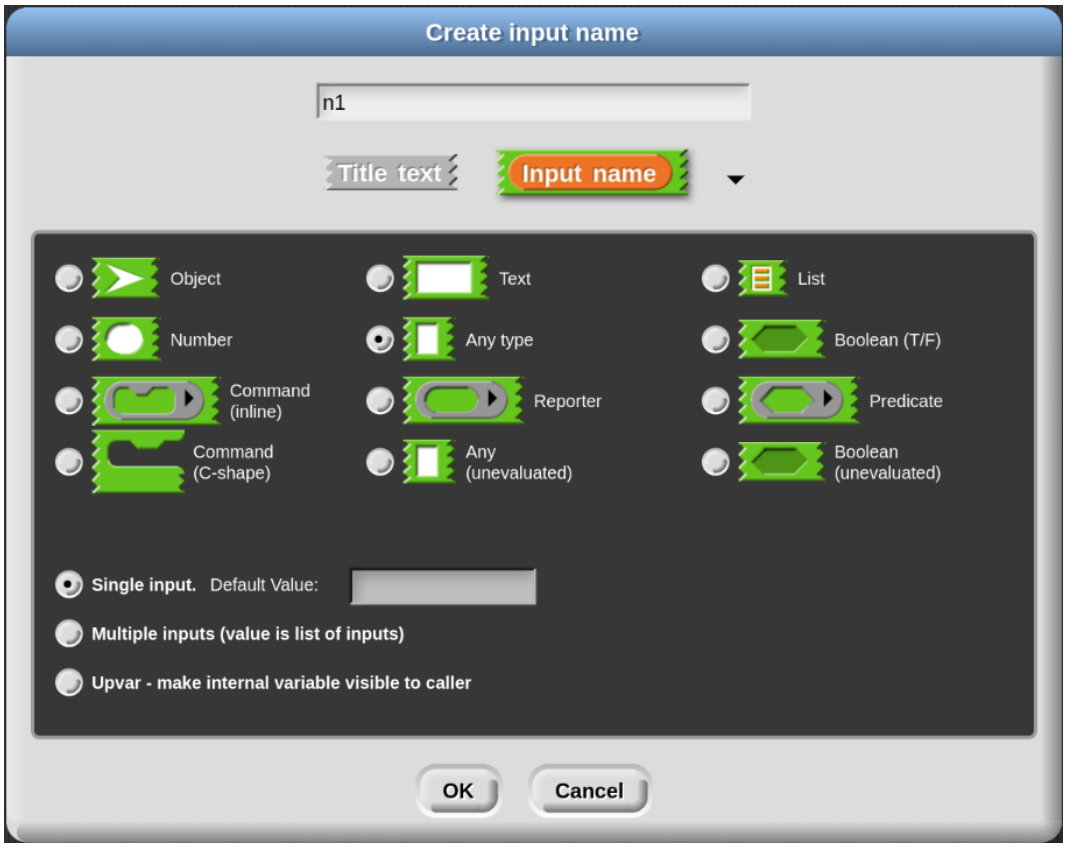
の部分をプロトタイプといいます。ここをクリックするとブロックのカテゴリーを変更することができます。

「>= 」 の左側の + ボタンをクリックしてください。

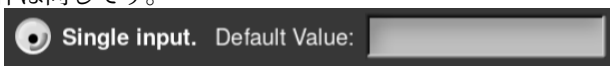


すると、入力ウィンドウが出ます。左側に Title text, 右側に Input name のボタンがあります。ブロックに表示される文字列を指定する時には Title text をクリックして文字列を設定します。ブロックを使用する時にデータを受け取るための変数を指定する時には Input name で設定します。

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

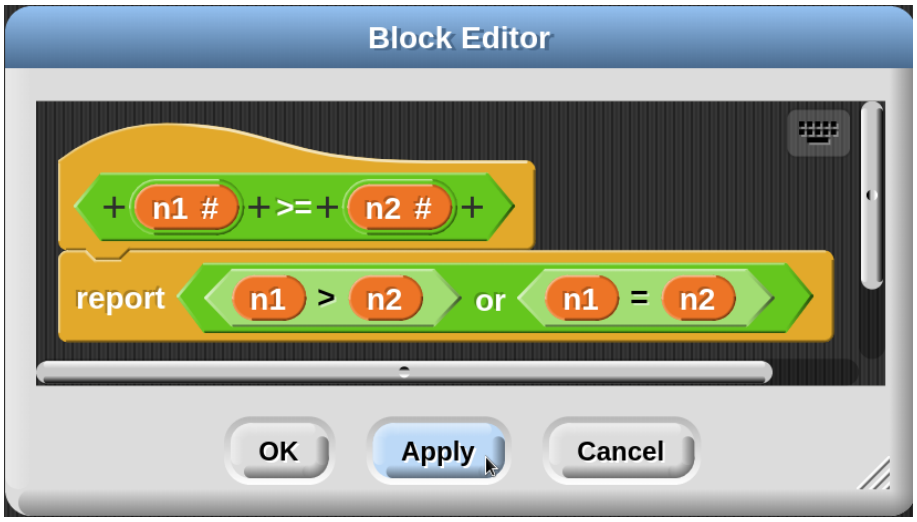


が出ます。現在は  が選択されています。 Any type つまり数値でも文字でも受け付けるタイプです。このままでもいいのですが、あえて  にしてみます。これは数値しか受け付けないタイプです。 Any type を選んだ場合は、変数の表記に「#」 が付かないだけで以下の操作は同じです。

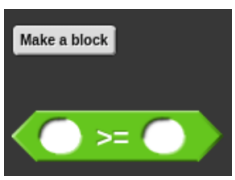
もう一つ、下の方に  が出ます。ここで入力の初期値が設定できるのですが、この場合は関係ないのでこのままにしておきます。 OK をクリックして次に進んでください。

右側の + ボタンをクリックして、同じように n2 の設定をしてください。

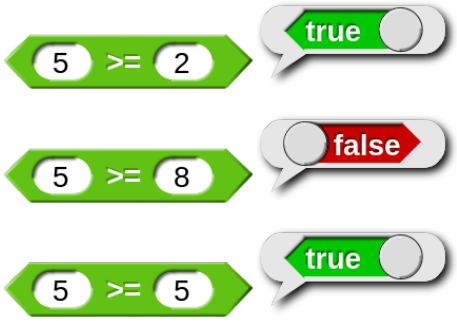
パレットエリアからブロックを持ってきて完成させてください。



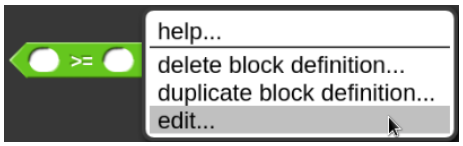
report は値をリポートする（返す）ためのブロックです。この場合は式の結果により真理値、true か false をリポートすることになります。 apply をクリックしてください。

パレットエリアに 

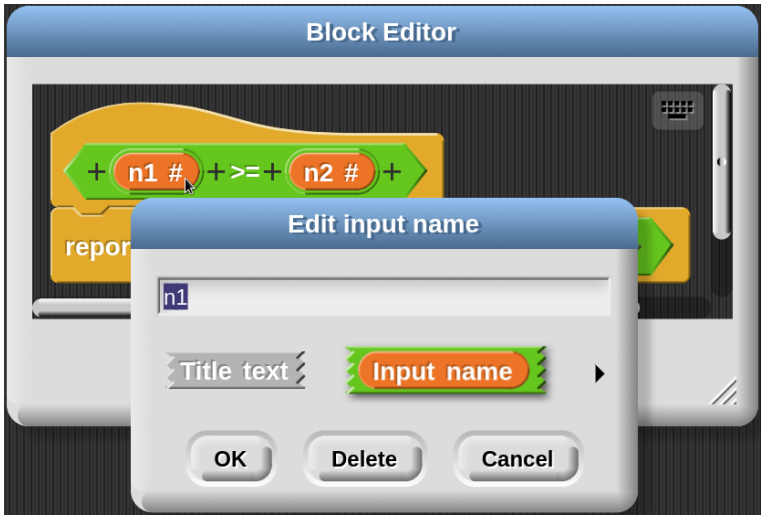
がセットされます。



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



ブロックを右クリックすると、edit... で内容の編集ができます。

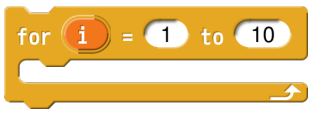


から変更ができます。変数を Any type に設定し直すこともできます。

を使ってステップ実行する場合に、作成したブロックの内部をステップ実行させたければ

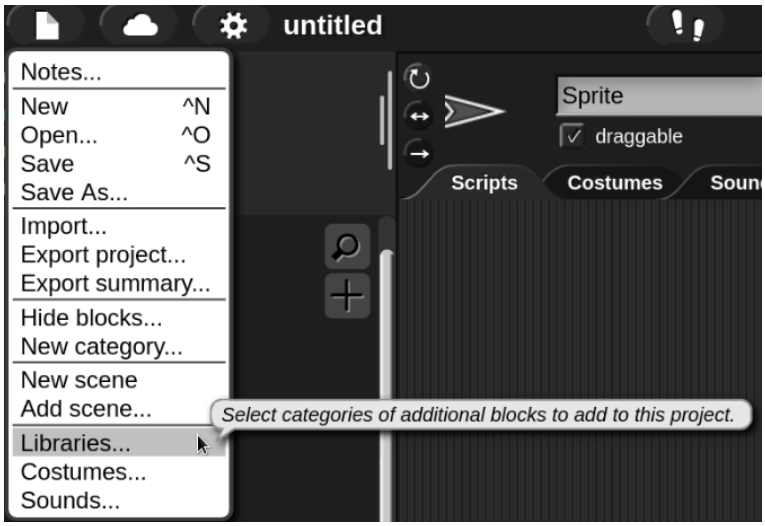
オンにしてからそのブロックをブロックエディターで表示させておく必要があります。順序が逆だとそのブロック内のステップ実行はされません。

6.2 for i = start to end step add

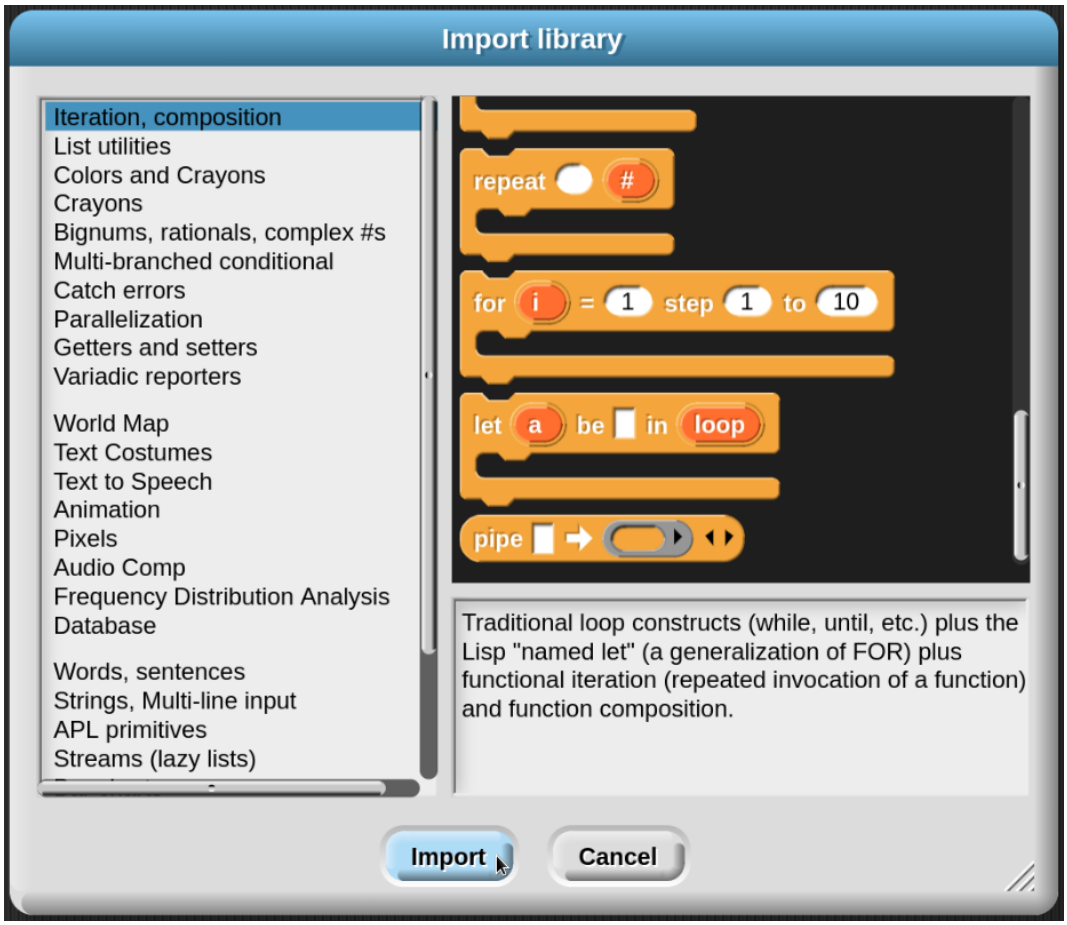
Snap! には 

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

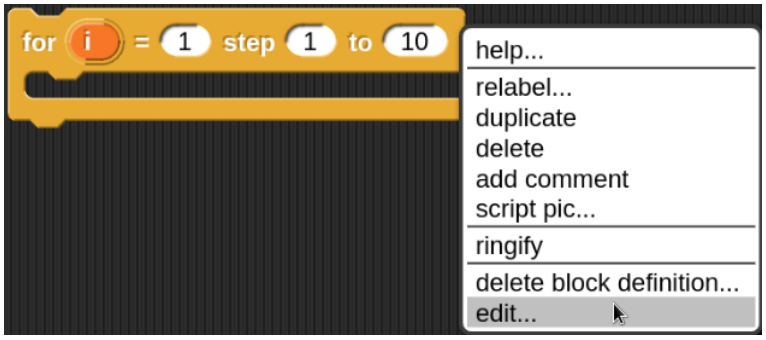
をクリックします。



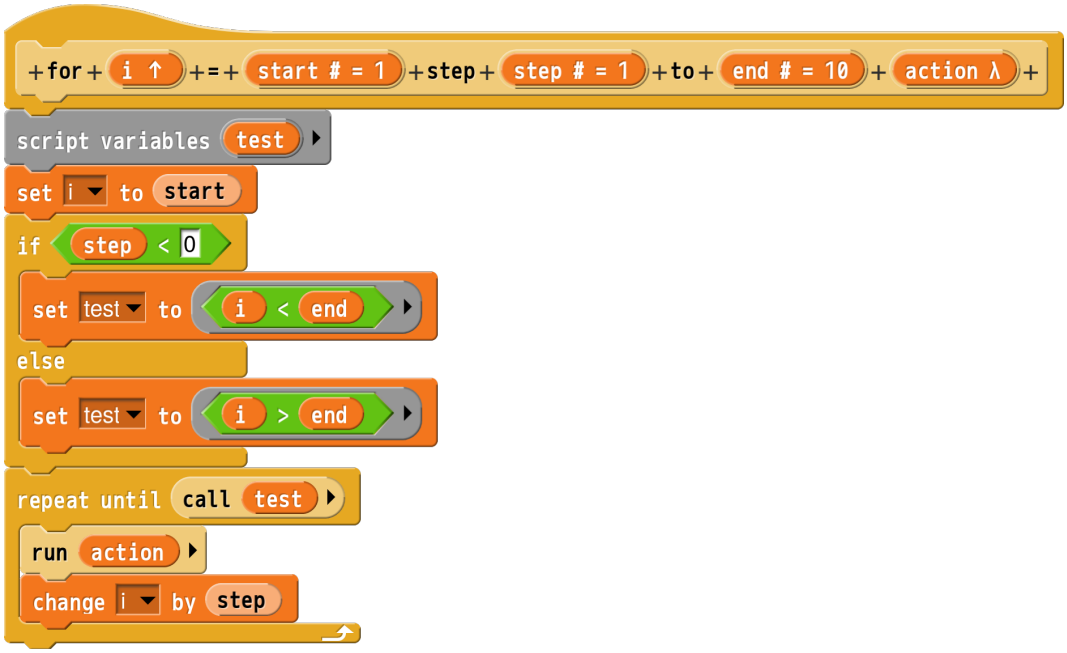
Libraries... をクリックします。Iteration, composition をクリックして内容を確かめてから Import すれば追加できます。



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



を右クリックして、edit... で中身を見てみます。



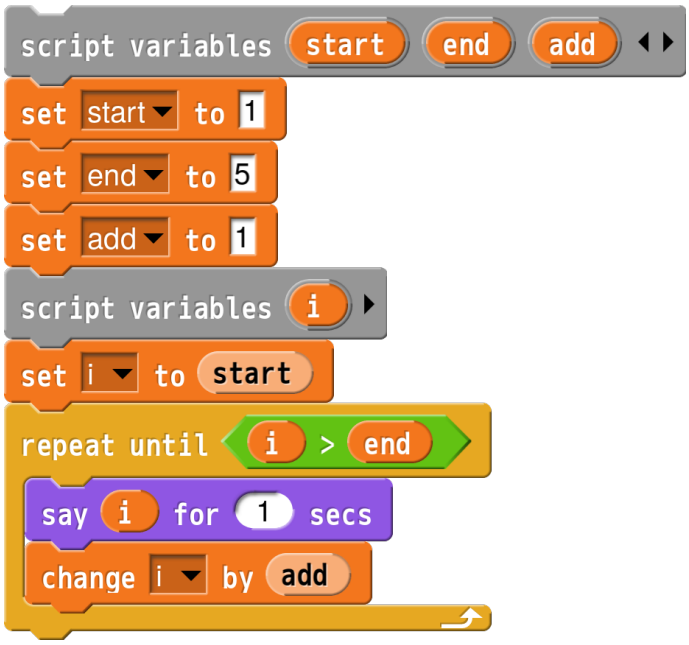
このスクリプトにしたがって、my for i = start to end step add という増加部分の位置が違うだけのブロックを作成してみます。

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

i が増加していく場合です。

start, end, add, i の変数をスクリプト変数を使い 

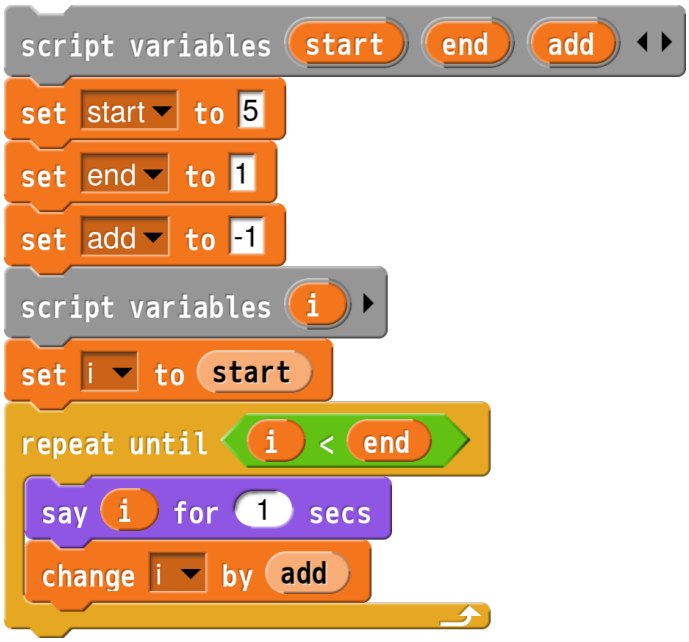
で作ってみます。



start を 5、end を 1、add を-1 にすると、5 > 1 で、 

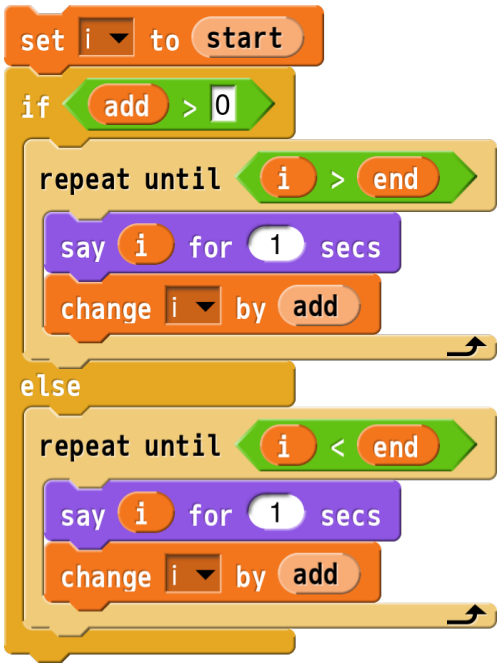
の終了条件を満たしてしまうので実行されません。減少カウントにする場合は、終了条件を 

にして

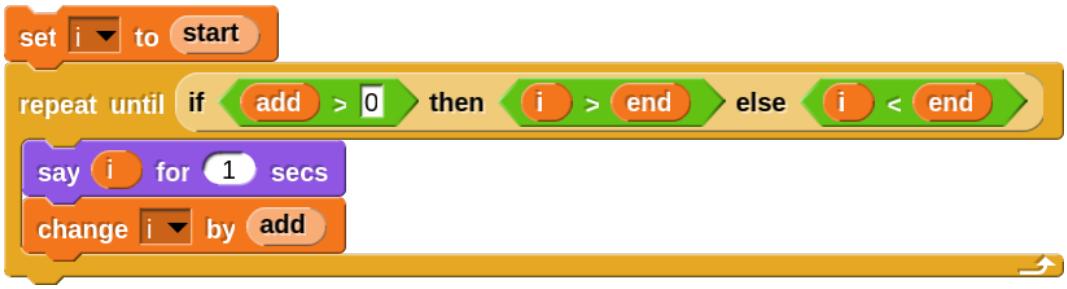


のようにしなければなりません。

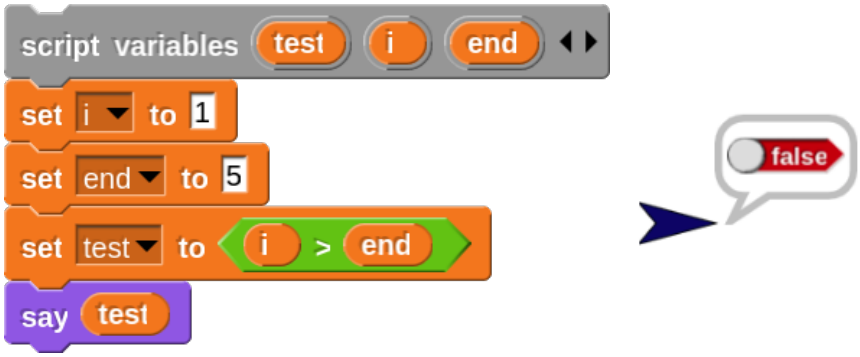
増加でも減少でも対応させると、for ループは次のようになります。



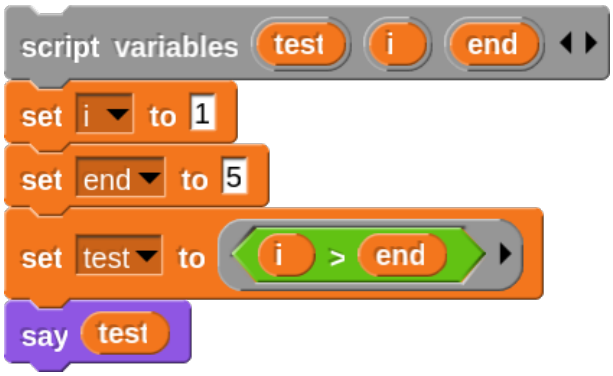
ループのテスト部分をまとめると次のようにすることができます。



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

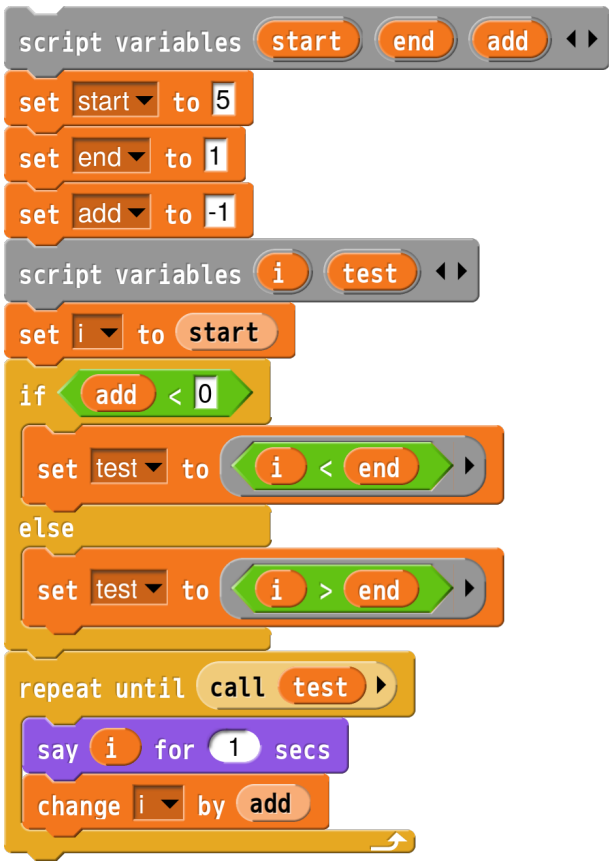
をパレットエリアからもってくる必要はありません。右クリックしてメニューから ringify をすればできます。

は形から分かるようにリポーターブロックで、

つまり、

のブロックをテストした結果をリポートしてくれます。

これを for ループに使用します。



これが Libraries の for ループの内容です。

ところで、このままではちょっと問題があります。増分が 0 の場合に無限ループになってしまうのです。増分が 0 なのだからそれでいいと考えることもできますが、無限ループになるのは嫌です。増分が 0 の場合には何もしないで終わるか、1 回だけ実行するかの選択がありますが、my for では何もしないで終わるようにします。

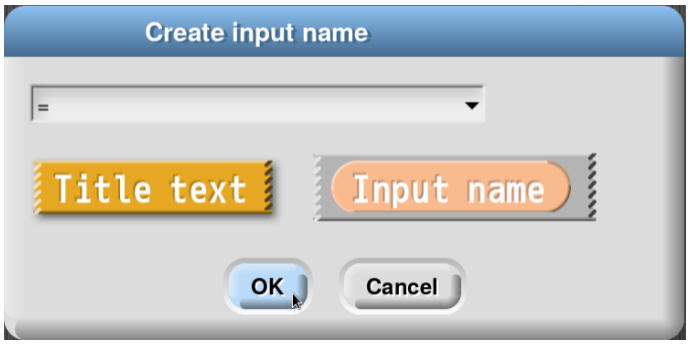
それでは、いよいよブロックエディターで作成していきます。

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

「+」 をクリックして、変数 i の設定をします。

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

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

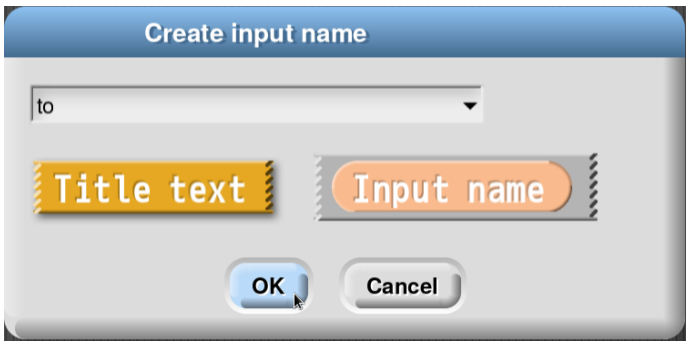


続いて、「+」 をクリックして、変数 start を設定します。

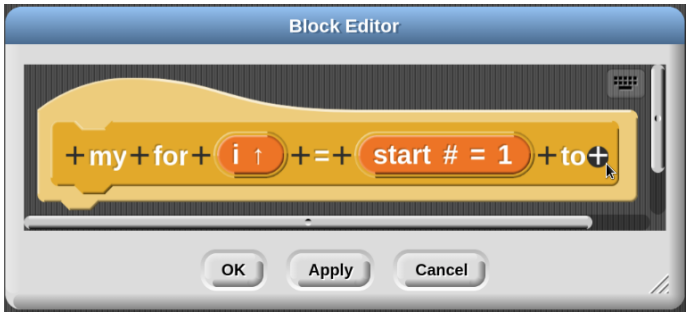
オプションとして、  、数値入力限定で、

規定値を 1 に設定します。
続いて、「+」 をクリックして、

「to」 を、Title text として入力してください。



続いて、「+」 をクリックして、

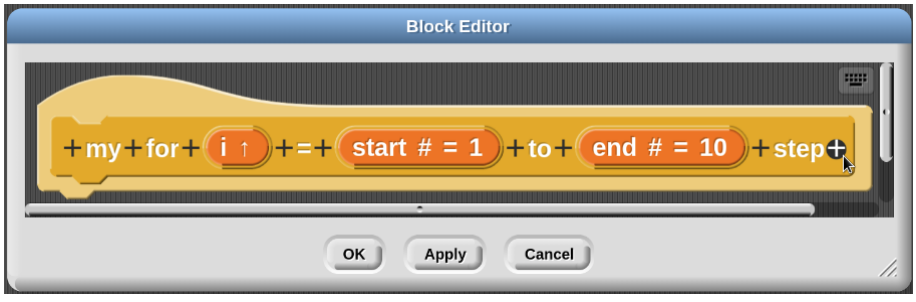


変数 end を設定します。

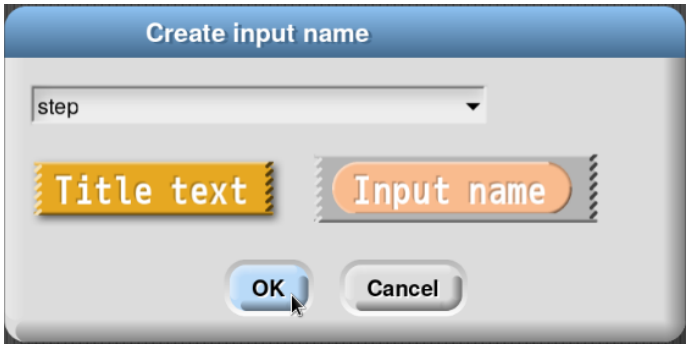
オプションとして、  、数値入力限定で、

規定値を 10 に設定します。

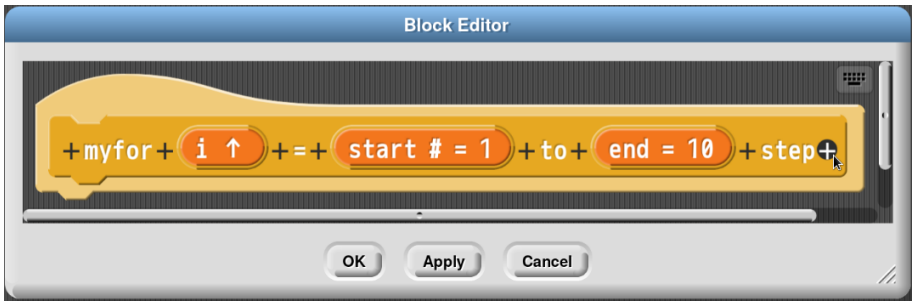
続いて、「+」 をクリックして、



「step」 を、Title text として入力してください。



続いて、「+」 をクリックして、



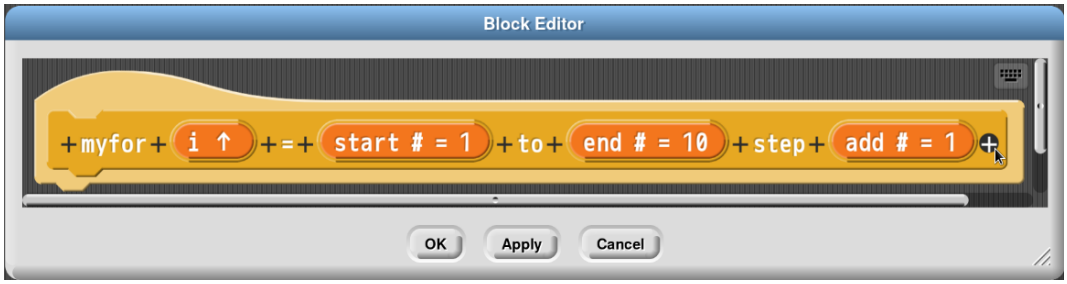
変数 add を設定します。

オプションとして、 

、数値入力限定で、

規定値を 1 に設定します。

続いて、「+」 をクリックして、

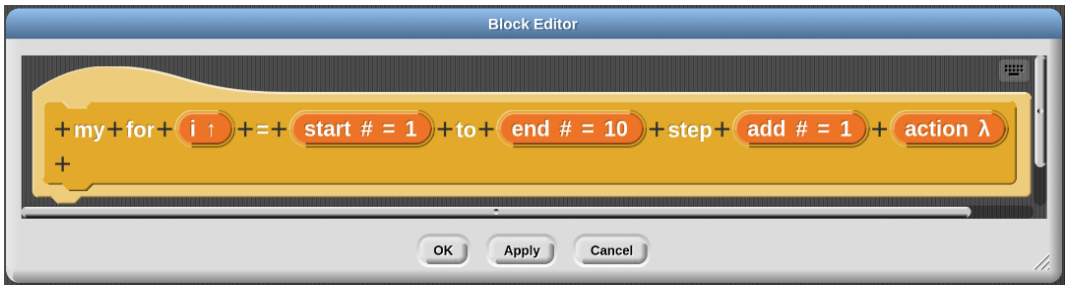


変数 action を設定します。

オプションとして、 

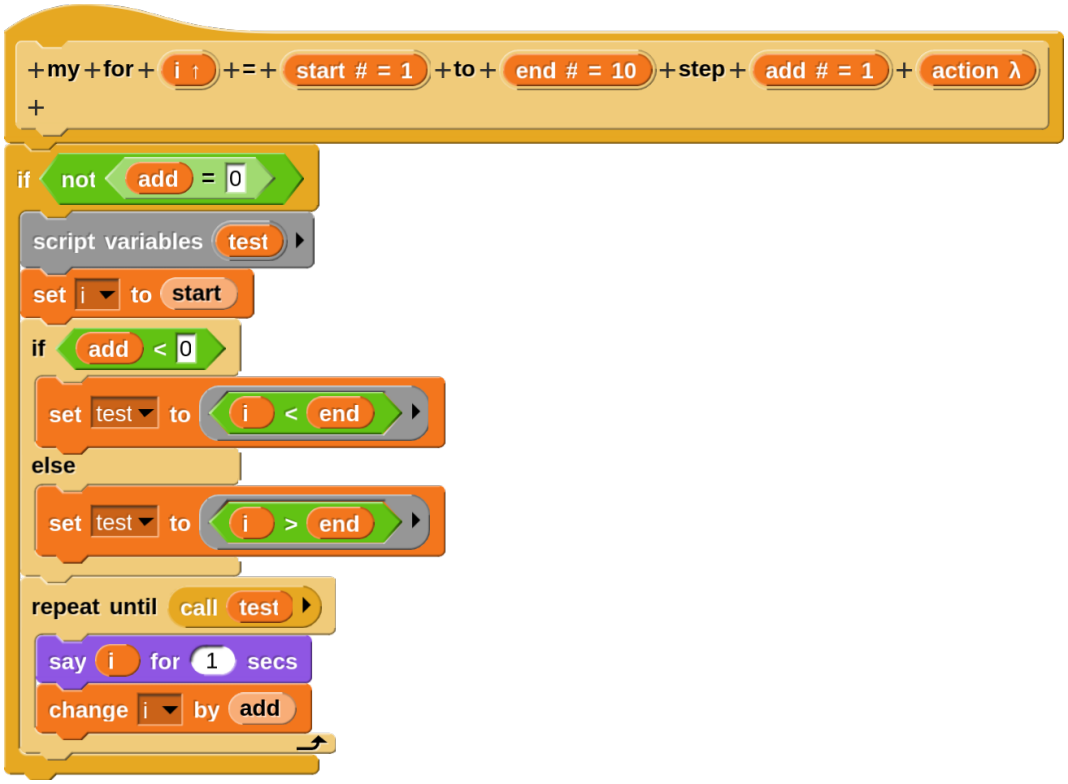
を選択すると、自動的に

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



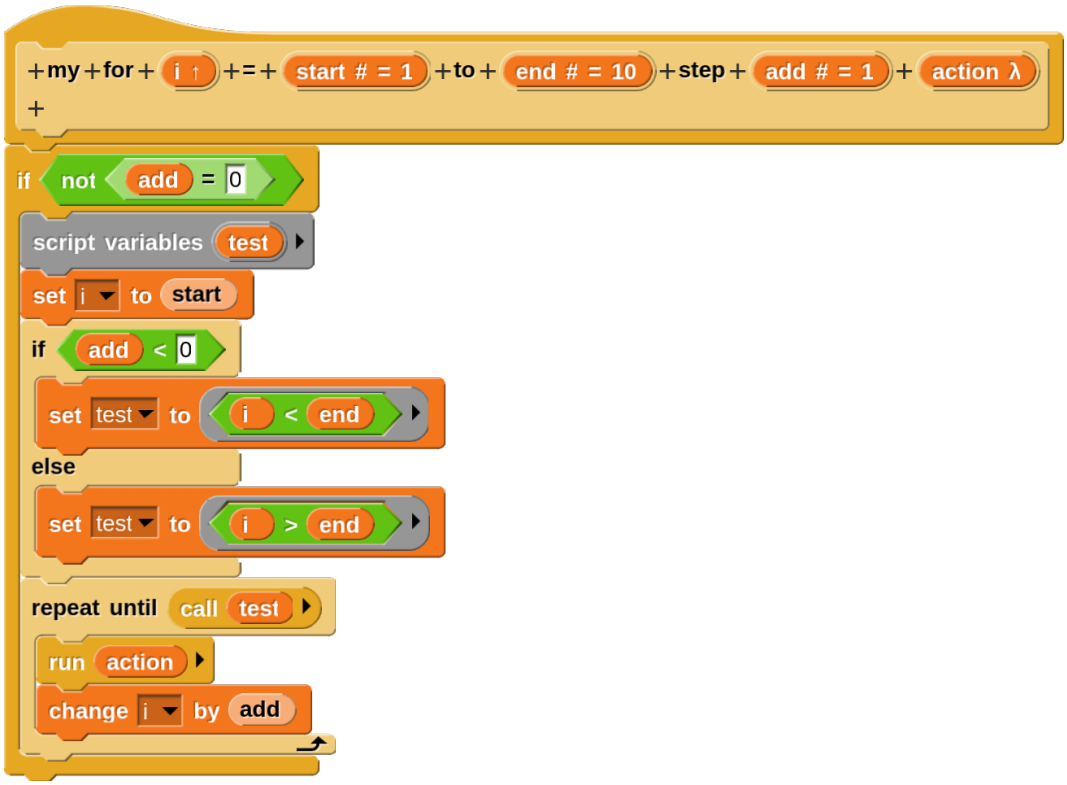
変数 action に特別なマークが付きました。

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

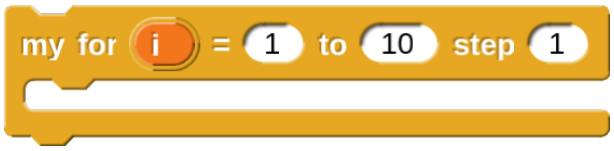


action で指定されたスクリプトを実行するブロックは run です。パレットエリアから持ってきて action をはめ込みます。

これで、実験用のスクリプトを入れ替えます。



Apply するとパレットエリアに作成したブロックがセットされます。



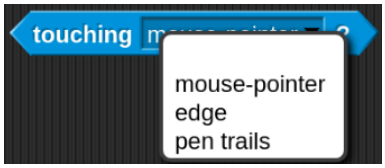
テストをしてみて問題がなければ OK をクリックしてブロックエディターを閉じてください。

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

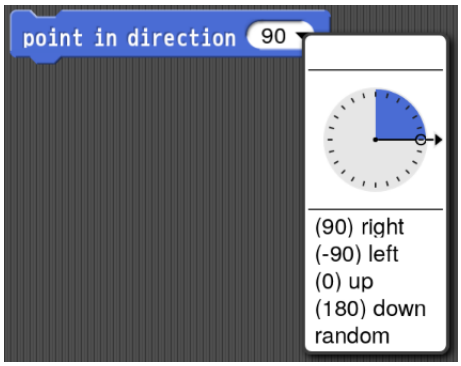
7　ブロック定義について

7.1　プルダウン入力

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



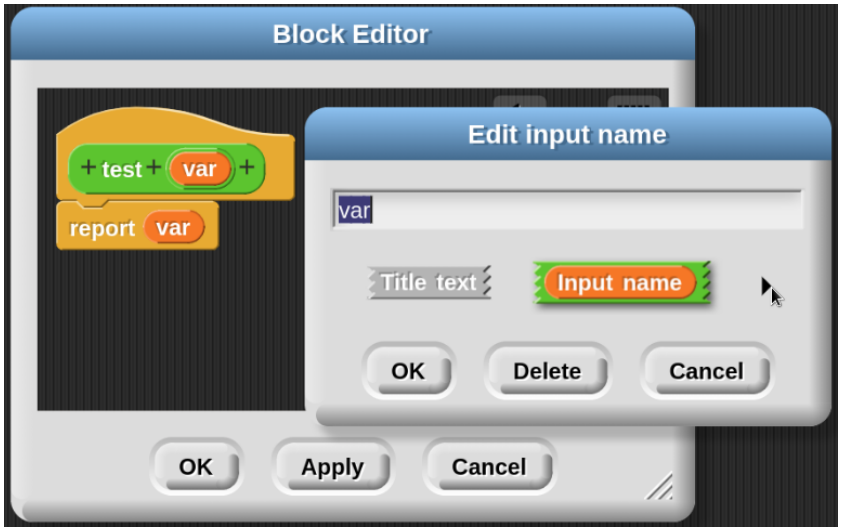
入力スロットに値をキーボードから入力することはできなくなっています。これは、後で出てくる read-only のオプションが指定されているためです。



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

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

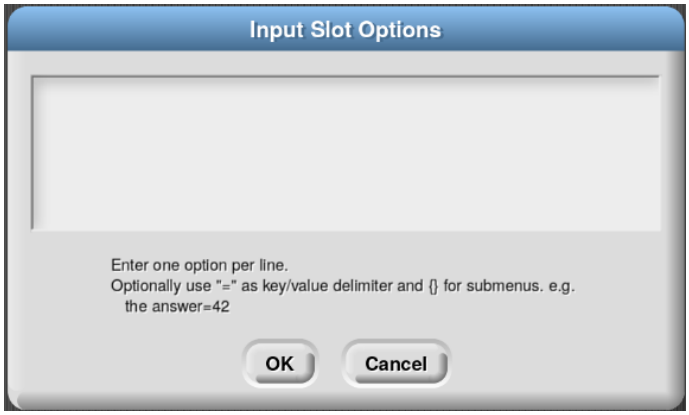
説明のために、test というブロックを作成します。



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

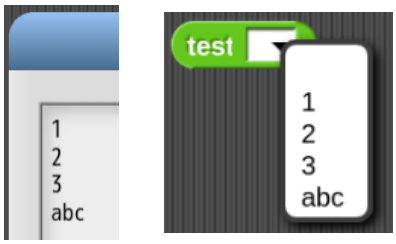
の var をクリックすると Edit input name のダイアログが開きますから、Input name の右側にある三角をクリックします。これで、大きな Edit input name のダイアログが開きます。ここの暗い灰色の領域で右クリックします。すると、このようなメニューが表示されます。

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



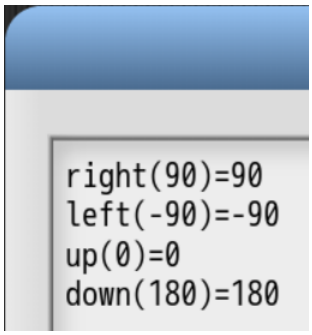
ここの各行にオプションを入れてプルダウンメニューを作っていきます。

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。

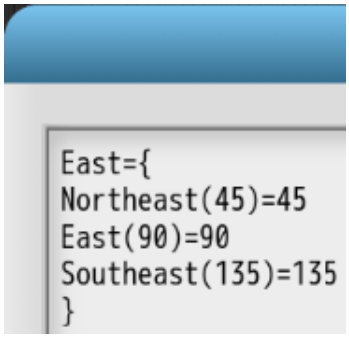


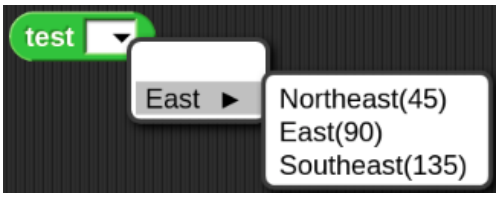
設定したものがそのまま表示され、クリックするとそれが入力値になります。

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。

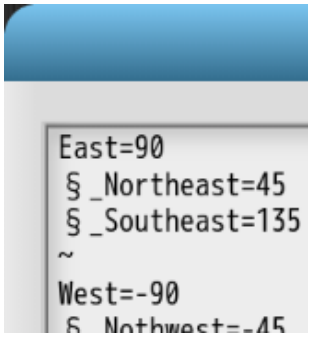


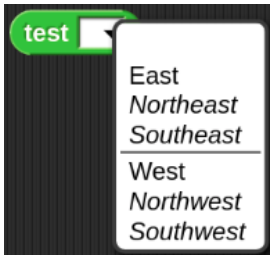
次のように、「 ={ 」で行を終えると、サブメニューを設定することができます。「 ={ 」の左側の項目はサブメニューの名前であり、メニューには「 ▶ 」を付けて表示されます。ここにマウスポインターを置くとその横にサブメニューが表示されます。「 } 」だけの行はサブメニューを終了させます。サブメニューは任意の深さまで入れ子にすることができます。





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。

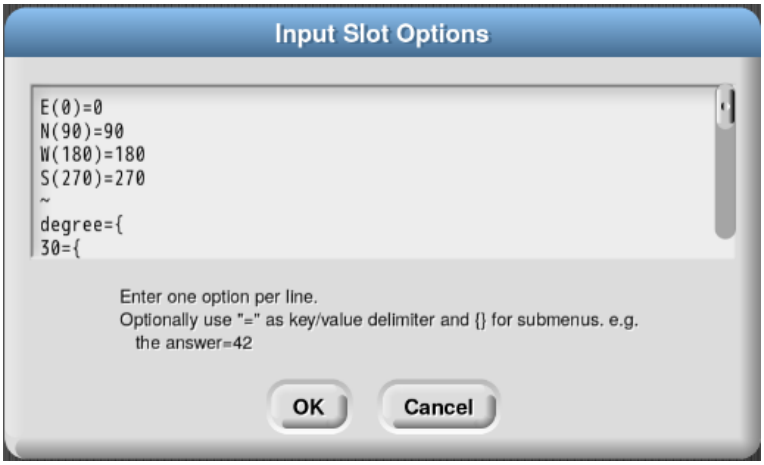


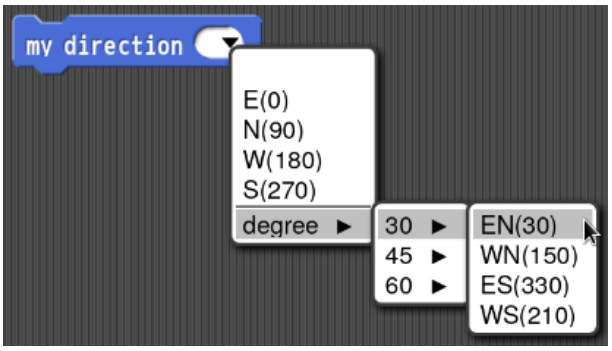


項目の前に、「§_」セクション記号とアンダースコア（アンダーバー）を置くと、シフトキーを押しながらクリックしないとプルダウンメニューに表示されなくなります。因みに、§記号はWindows OS の場合だと、Alt キーを押しながらテンキーから（通常の数字キーではなく）0167と入力、Linux OS の場合だと、Ctrol+Shift+u を押してから、a7 を入力して↵で出すことができます。a7 は 0167 の十六進数コードです。OS を問わず、日本語入力モードで「せくしょん」または「きごう」と入力して変換して出すこともできます。

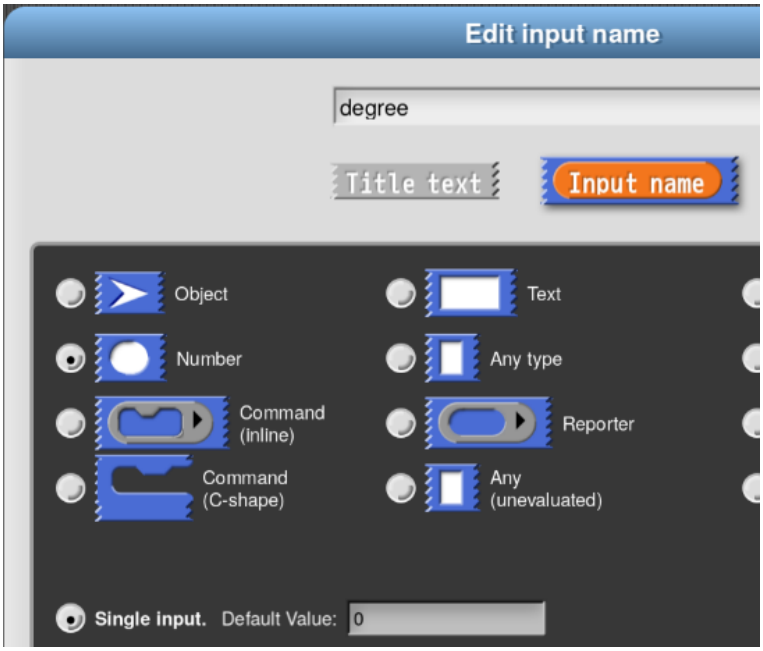
プルダウンメニューの例として、my direction というブロックを作ってみます。 Input name を degree とし、一般的な数学上の角度で向きを指定できるようにします。つまり、右方向が 0 度、上方向が 90 度、左方向が 180 度 という具合です。オプションを以下のように設定します。

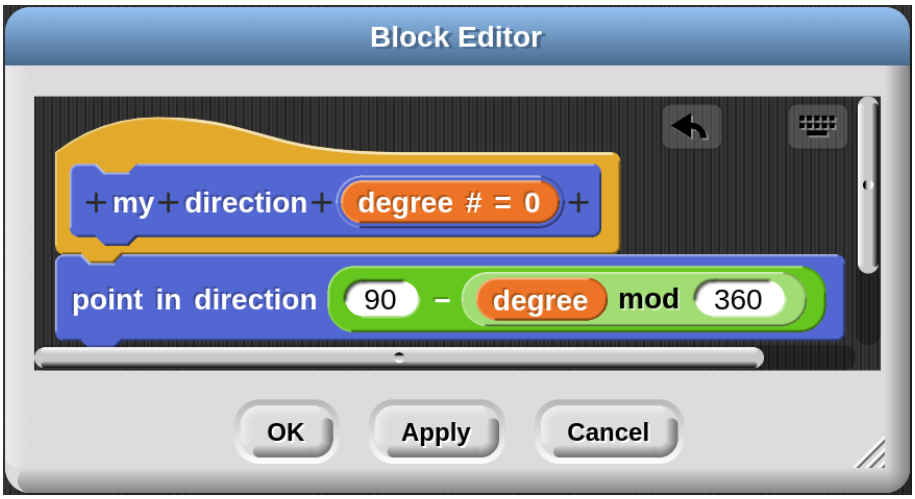
E(0)=0
N(90)=90
W(180)=180
S(270)=270
~
degree={
30={
EN(30)=30
WN(150)=150
ES(330)=330
WS(210)=210
}
45={
EN(45)=45
WN(135)=135
ES(315)=315
WS(225)=225
}
60={
EN(60)=60
WN(120)=120
ES(300)=300
WS(240)=240
}
}



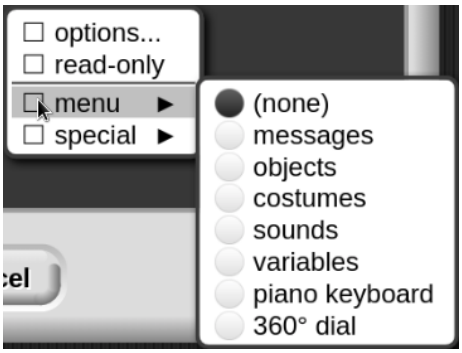


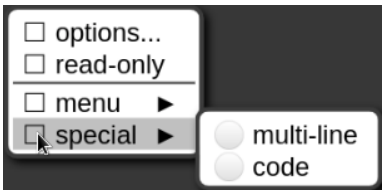
degree から、EN, WN, ES, WS のサブサブメニューを作っています。あくまで機能説明の作例ですので、あまり意味のあるものではありません。





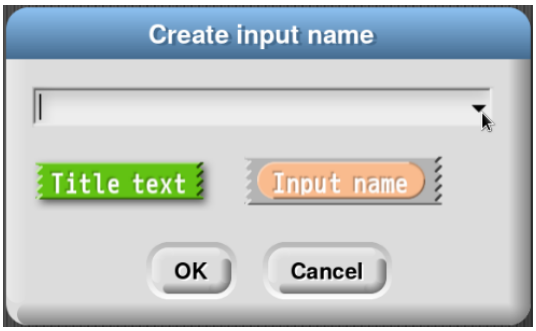
また、menu サブメニューから選択することで、いくつかのプリミティブブロックで使用されている特別なメニューを取得することも可能です。いろいろと試してみてください。





7.2 Title Text とシンボル

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



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

そこからお目当てのシンボルを選択します。 turtle を選んでみます。

すると、入力欄に $turtle がセットされます。

OK して Apply すると、 

になります。

定義を $turtle-1.5-0-255-255 に変更すると、 

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

シンボルメニューの最後に、「new line」があります。 Title text にこれ $nl を設定するとそこで改行されます。また、シンボルじゃなくても、文字列の頭に「$」 $test-1-0-0-255 を付けるとシンボルのように倍率と色の指定ができます。反面、シンボルと同じ文字列は使用できないということですが。

定義は、こうなります。

7.3 Input name オプションについて

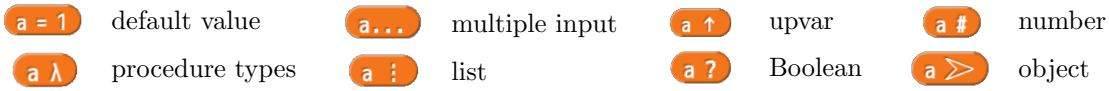
ブロックを作成する時の Input name のオプションについて見ていきます。間違っているかもしれませんが、こういう考え方をするとオプションを選ぶ目安になるのではないかと思います。

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必要があります。（指定しなかった場合は、Any type, Sngle input になります。） 期待する入力のタイプを入力スロットの形で示すためであったり、機能を設定するためです。

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

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

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



7.3.1 Reporter 型

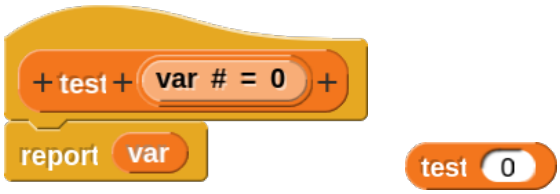
Object を選択して次のような定義にすると、 

というブロックができます。

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

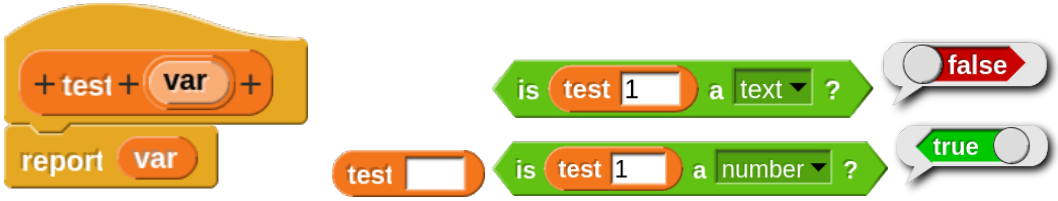
入力スロットへのキーボードからの入力を数値限定にしたのが、Number です。

Default Value を指定すると、

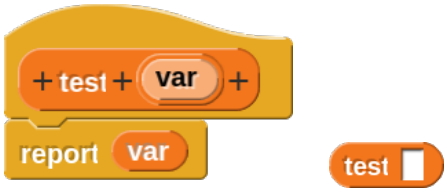


既定値を設定することができます。

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



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



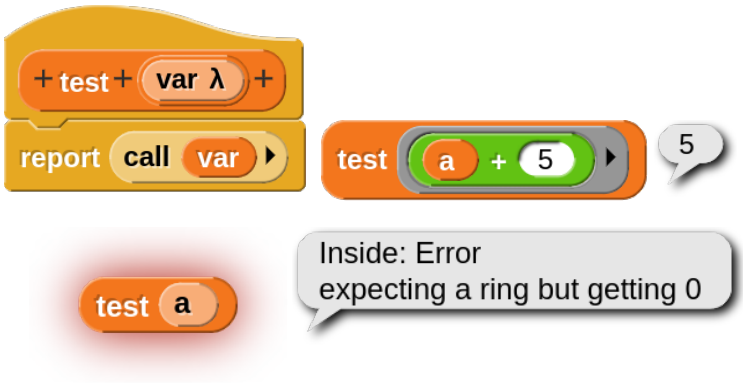
リストの入力を求めていることをアピールするのが、List です。

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

変数単体だと次の Any(unevaluated) とは違い 

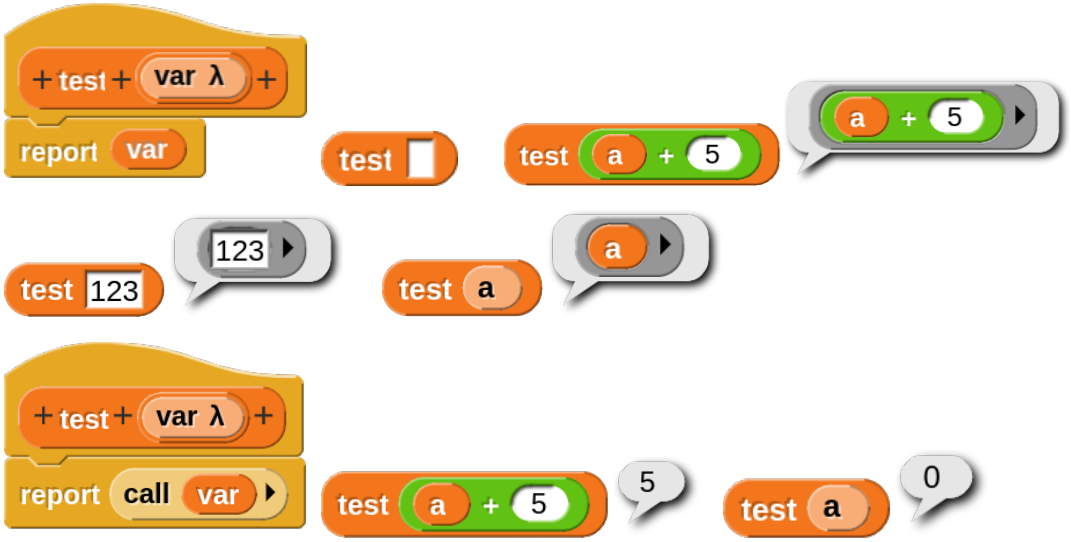
とリング付きではなくなってしまいます。(ver. 7.0.2 現在)

実際には call ブロックを使ってリングブロックの値を求めます。



a にリングが付いていない状態なのでエラーになります。(ver. 7.0.2 現在)

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



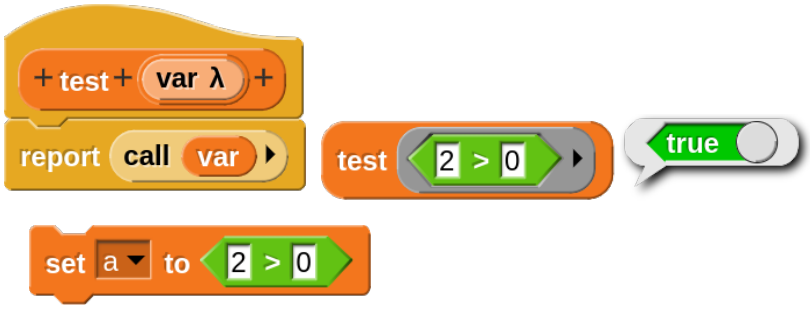
7.3.2 Predicate 型

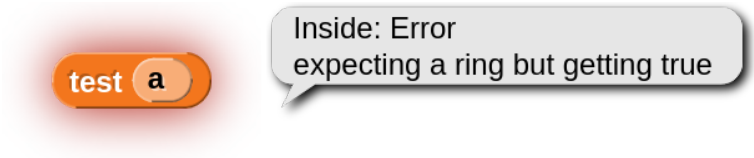
Predicate 型には、Boolean, Predicate, Boolean(unevaluated) があります。Reporter 型での Any type → Reporter → Any(unevaluated) の関係が、Predicate 型にも当てはまります。

Boolean です。

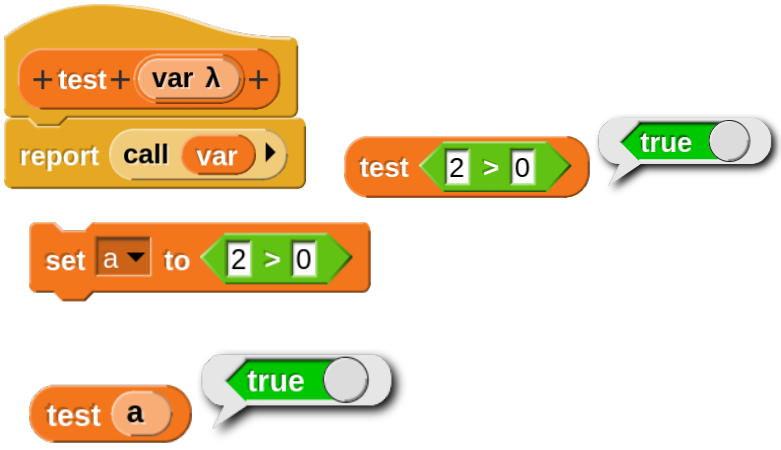
Predicate です。

リングで囲われた Predicate 型変数をテストするには call ブロックを使って、評価し、真理値を求めます。変数単体に対して、Reporter と同じことがここでも起こります。(ver. 7.0.2 現在)





Boolean(unevaluated) です。unevaluated つまり、評価されていない、値を求めるような操作はされていないということです。



7.3.3 Command 型

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

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

c 言語では、

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

のようにすると、0、1、2、3、4 と表示するプログラムになります。（ ; ; ）のようにカッコの中がセミコロンで 3 つの部分に分けられています。

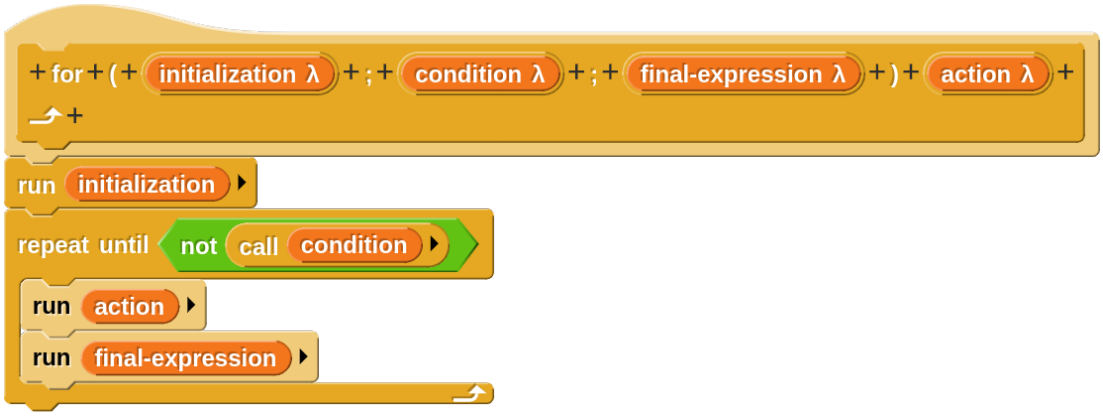
`i = 0` の部分が初期設定で、初めに一回だけ実行されます。

`i < 5` の部分が実行を続けるかどうかのテストをします。

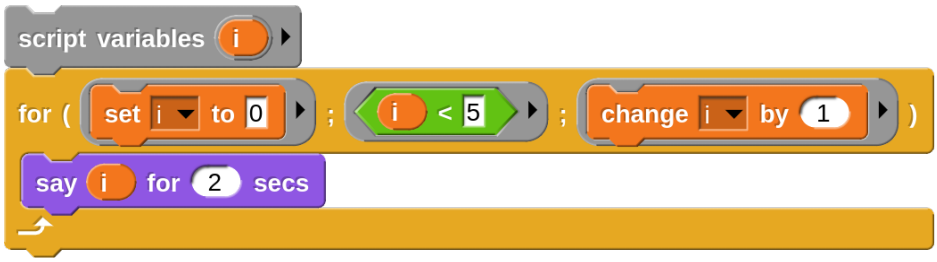
`i++` の部分は次の繰り返しに進む前に実行されます。 `i++` は `i` に 1 加算する処理をします。

`{ }` の内部がループの本体です。

以下が定義です。定義自体は run や call に丸投げするだけなので my for よりもシンプルですね。

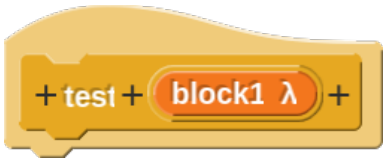


なお、initialization と final-expression は Command(inline) 型で、condition は Predicate 型です。action は Command（C-shape）型 です。



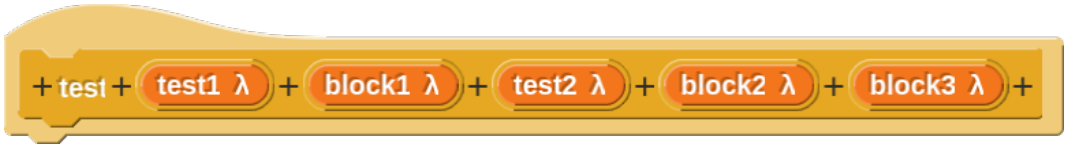
for ループは C 型が 1 つでしたが、 2 つにすると E 型になります。 if else の形ですね。

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

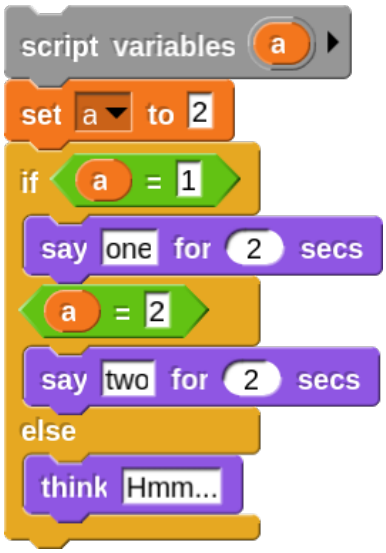


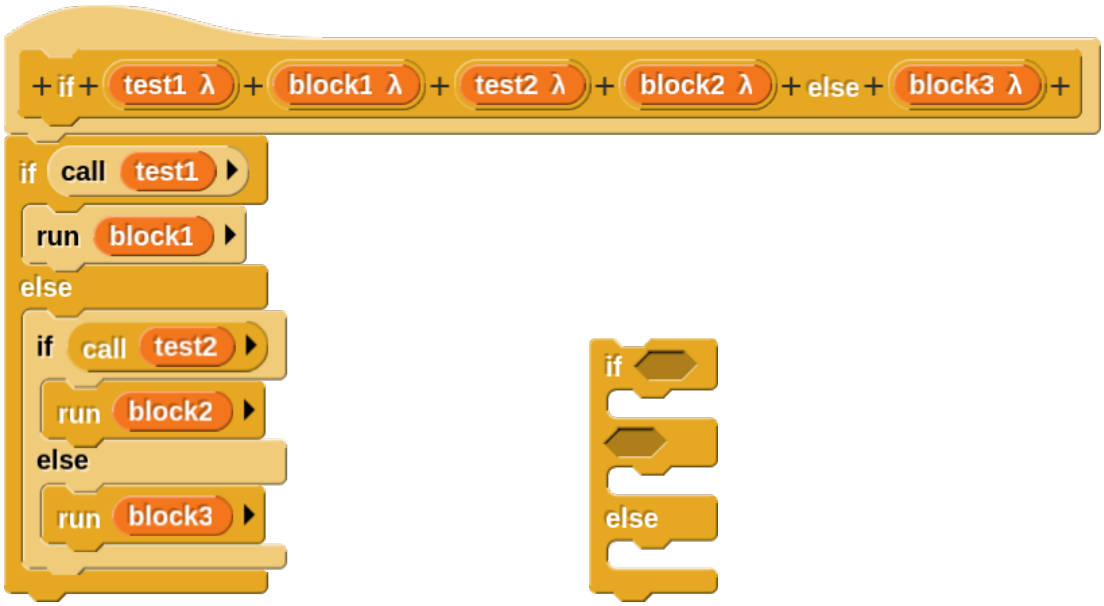
変数 block1 の前に Boolean(unevaluated) の変数を入れると、if else ブロックができます。

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



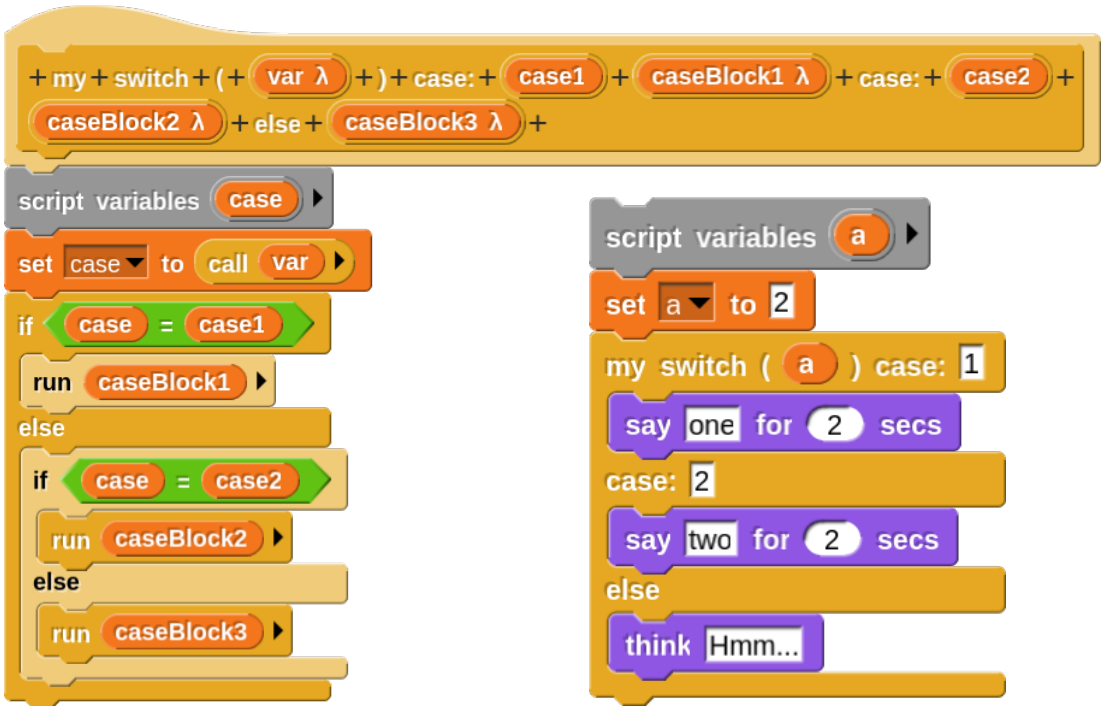
if else ブロックとしての体裁を整えてこんなふうに使えるようにしてみます。





これと同じようなことをする、switch case ブロックを作ってみます。

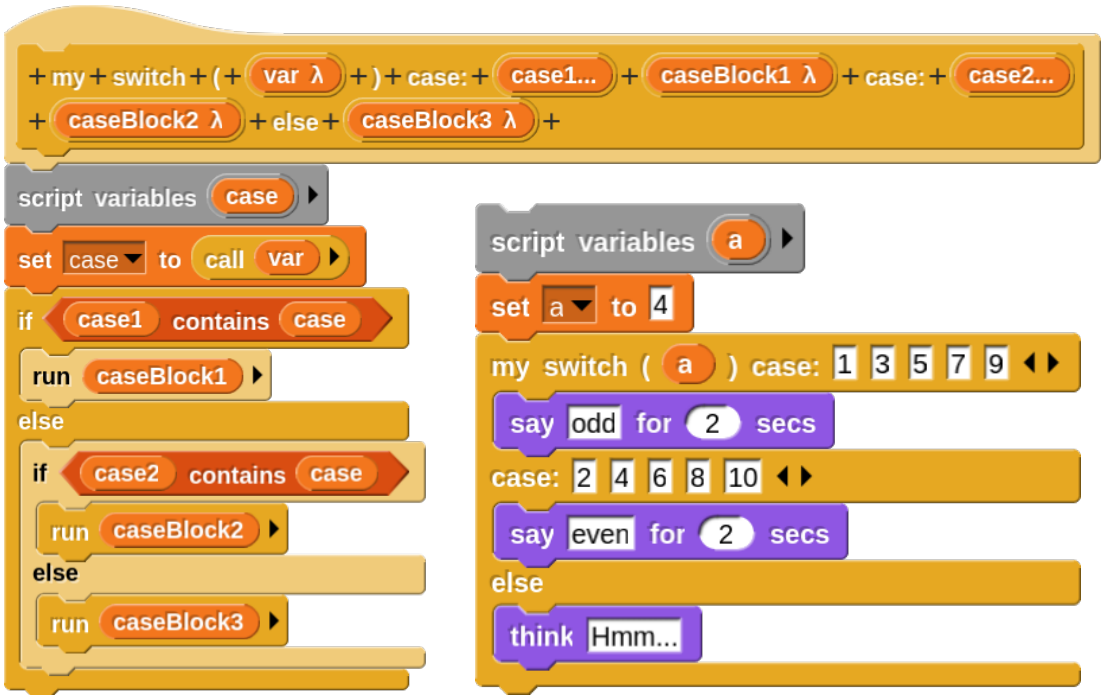
switch の（　）で調べる変数を指定して、case: で一致した場合にその処置をします。



変数 var は、Any(unevaluated) にしています。

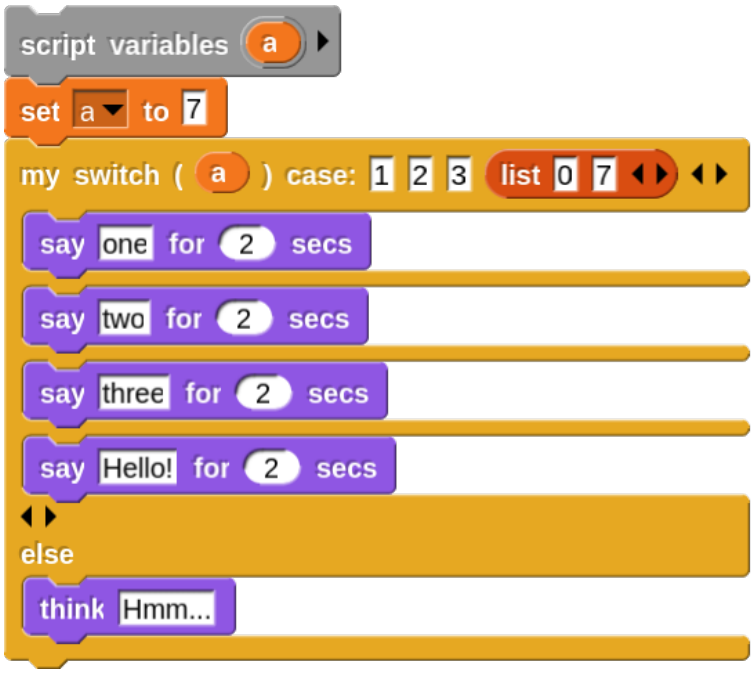
ここでは、変数 case1、case2 を Any type, Single input にしていますが、これを Any type, Multiple inputs にすると、case: で指定する入力スロットを追加することができます。その場合は、入力スロットの値をリストで受け取りますから、一致のチェックに 

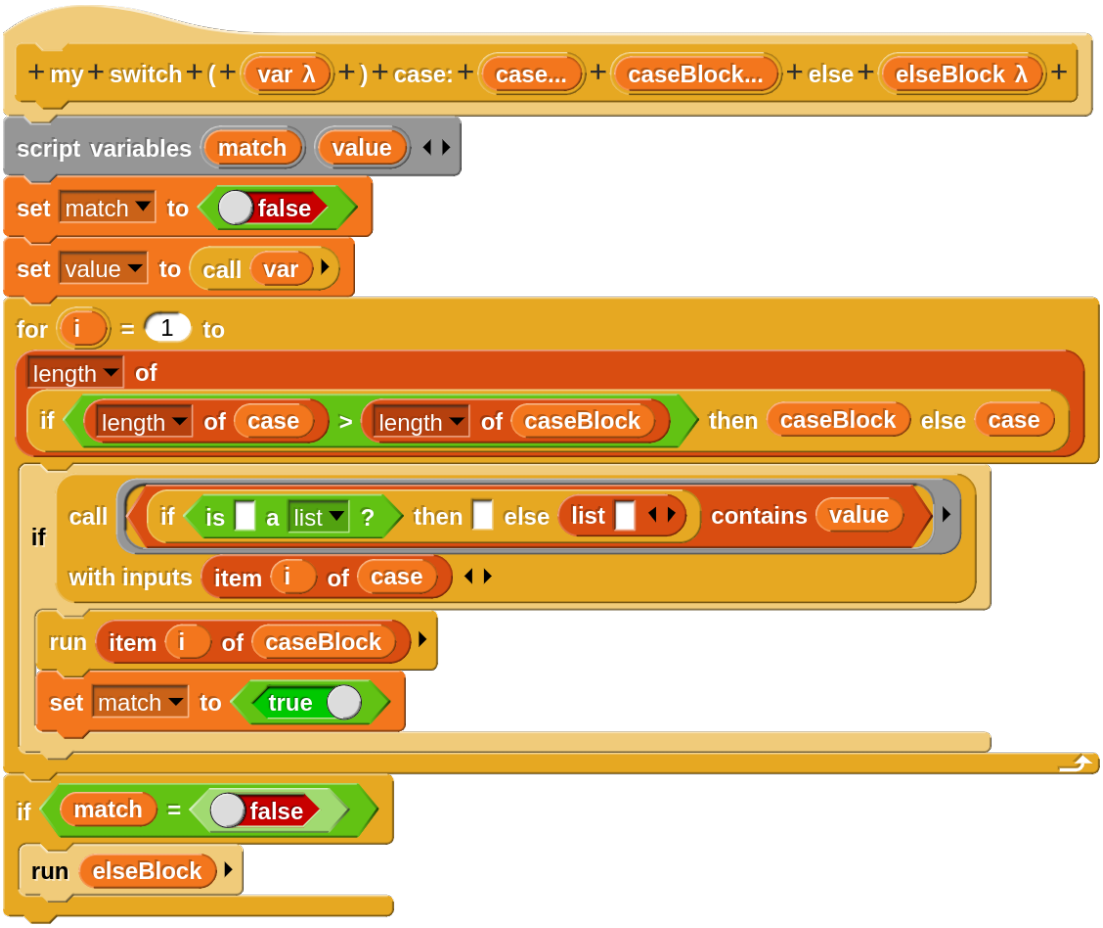
を使います。



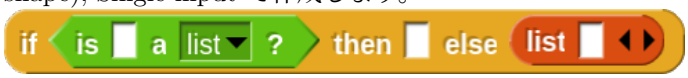
C 型ブロックも Multiple inputs で作成すると追加することができます。

改良版です。次のようにして実行します。case の値は 1 つの数値でも、リストで複数の数値でも指定でき、マッチした位置に対応したスクリプトを実行します。case の値および実行ブロックの指定は右向きの三角をクリックして追加することができます。





定義は、var を Any(unevaluated), Single input で、case を Any type, Multiple inputs で、caseBlock を Command(C-shape), Multiple inputs で作成します。 elseBlock を Command(C-shape), Single input で作成します。

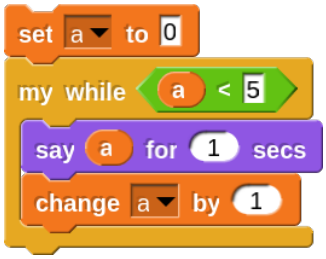


の部分は case の値がリストの時はそのまま、１つの数値の場合はリストにします。リストにしないと

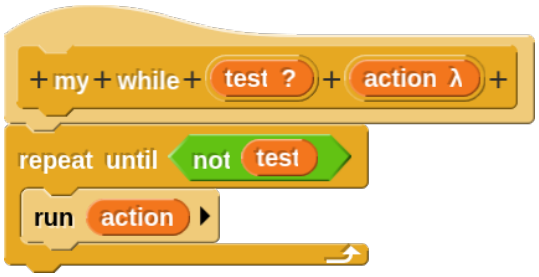
が使えませんから。case で指定された値（リスト）に、value の値が含まれるかを contains でチェックしています。この定義では、複数の case の値にマッチした場合にその都度指定のスクリプトを実行します。変数 match の真理値によって、どこかでマッチしたら他は実行しないようにすることもできます。

for ブロックの繰り返し回数設定がごちゃごちゃしていますが、case の項目数と実行ブロック項目数が合わない場合は少ない方を選択します。

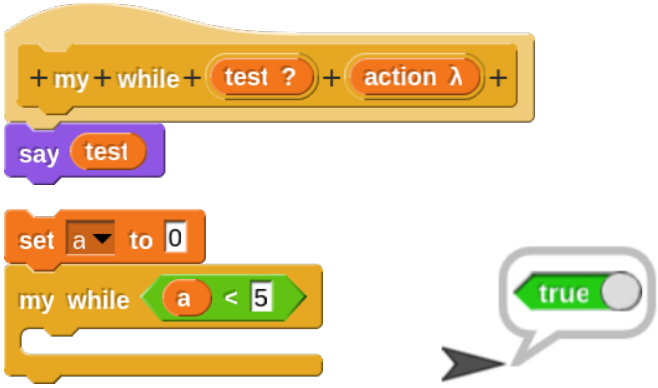
while ループを作成しながら、Predicate 型の補足説明をします。



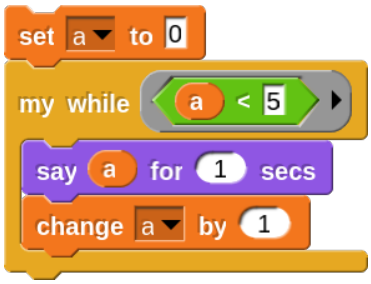
これは、 a < 5 のテストが true ならば、my while 内のループを繰り返します。
test を Boolean で、action を Command(C-shape) で次のように作成します。



しかし、これを実行するとテストがうまくいかないようで、終わらなくなります。
定義を変更してテストしてみます。

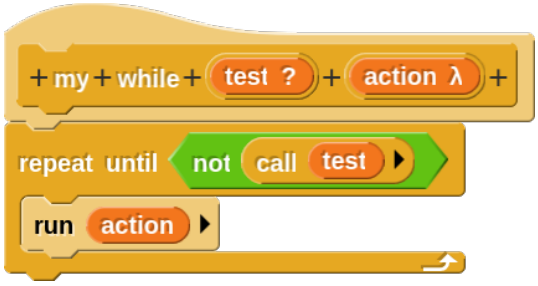


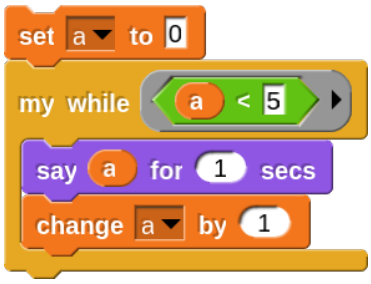
これは、test が受け取ったのは a < 5 という式ではなくて true という値だということです。
定義をもとに戻して、 a < 5 をリングで囲ってやってみます。



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

受け取った式から真理値を得るために call を使う必要があります。

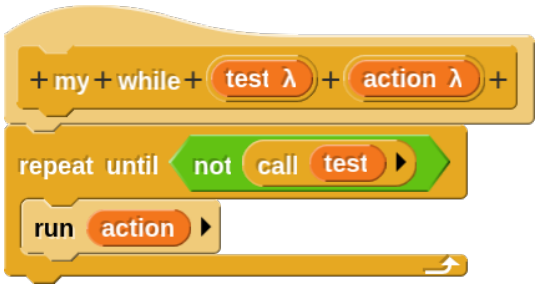


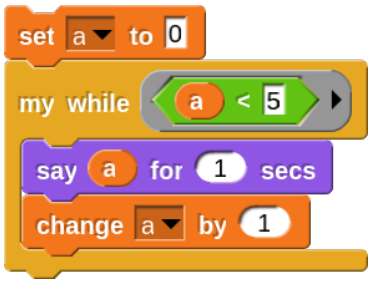


今度はうまくいきました。

test の型を Boolean 型から Predicate 型に変更すると、 

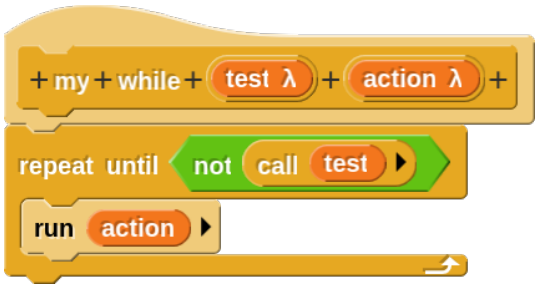
になるため、式をリングで囲ってからドロップする必要がなくなります。

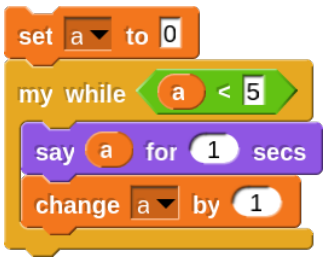




さらに、Predicate 型から Boolean(unevaluated) 型に変更すると、 

リングが消えてすっきりします。





8 Continuation 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。

8.1 w/continuation

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

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

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

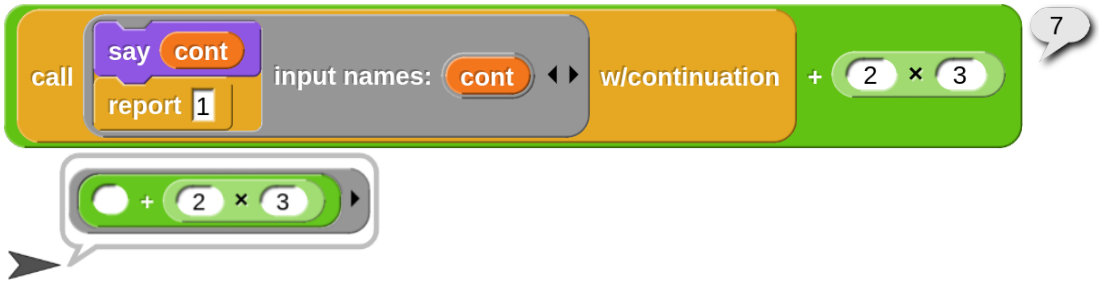
これは外側の入力スロットからの入力を受け取るためのものなので、入力がないためエラーになります。外側に入力をセットするとこうなります。

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

フォーマルパラメーターを表示させてみます。

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

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、その後に処理すべき内容がセットされます。（フォーマルパラメーターを cont にリネームしています。）

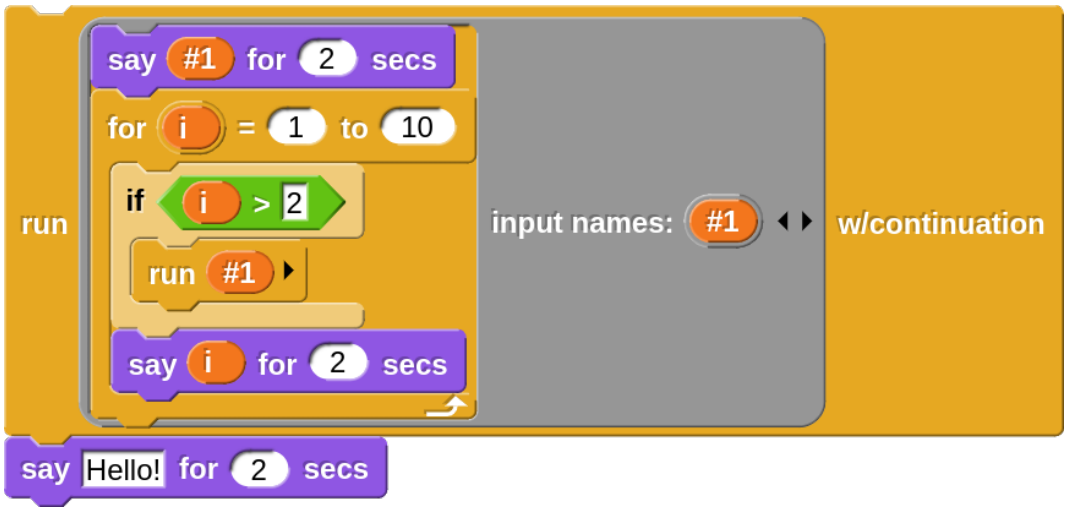


同様にスクリプト中での継続もあります。

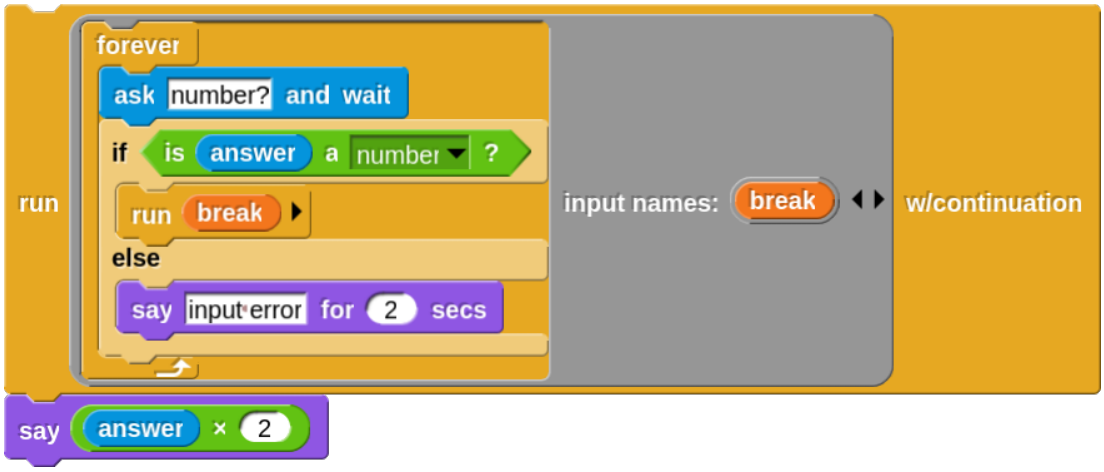
マニュアルには継続の詳しい使用例が載っています。ここではわかりやすい用途として、繰り返しなどのブロックからの脱出にしぼります。

の中に繰り返しのブロックを入れてやります。その中でフォーマルパラメーター #1 にセットされる継続（このブロックに続くスクリプト）を run すると、繰り返しブロックから脱出して継続するスクリプト動作に移行します。

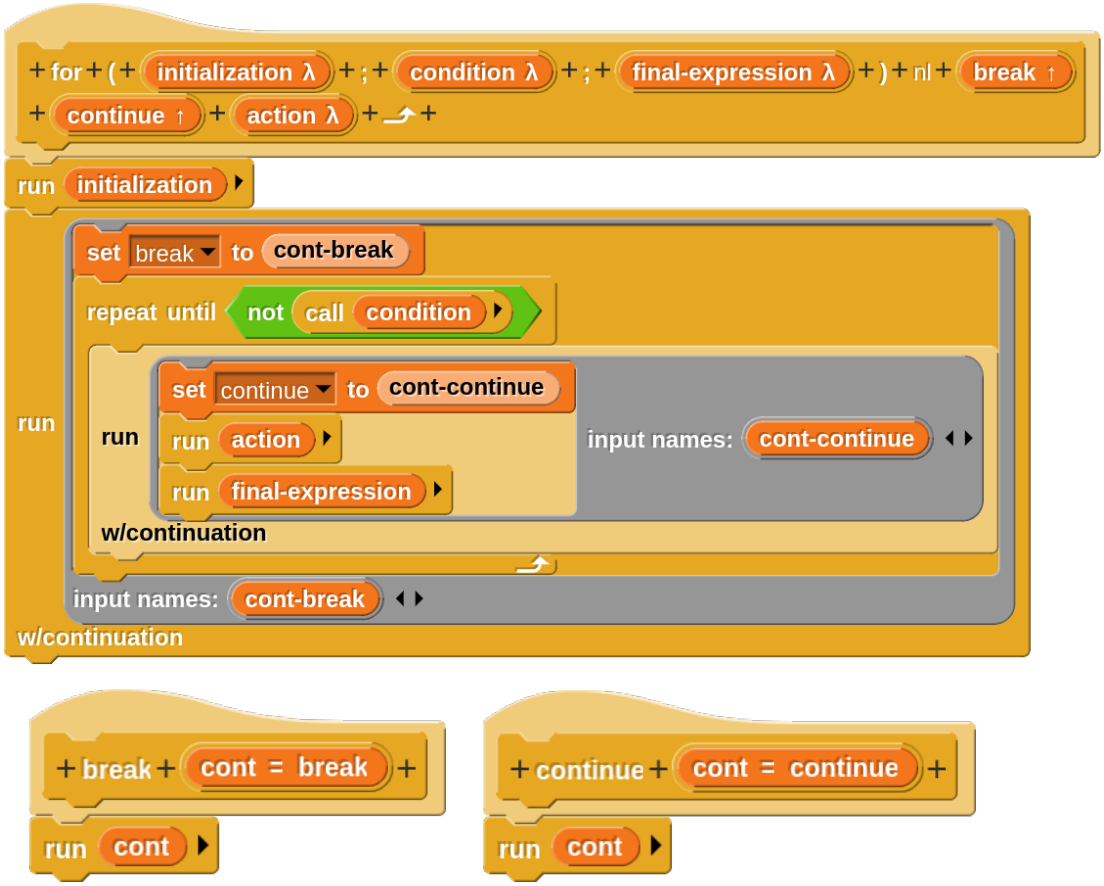
このスクリプトでは、i の値が 3 以上になったら継続スクリプトの実行に移行、つまり繰り返しブロックから脱出します。（最初にフォーマルパラメーター #1 にセットされる継続部分を表示します。）



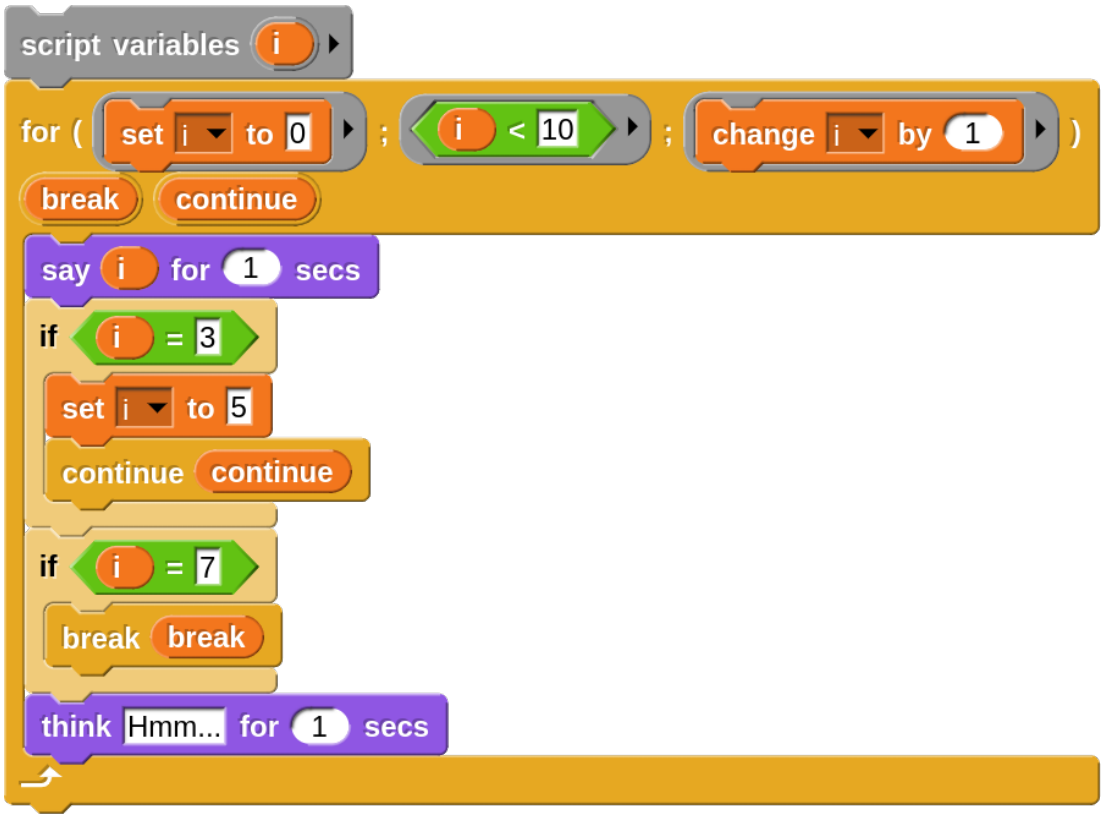
同じような例です。これを使わなくても repeat until でできますが。



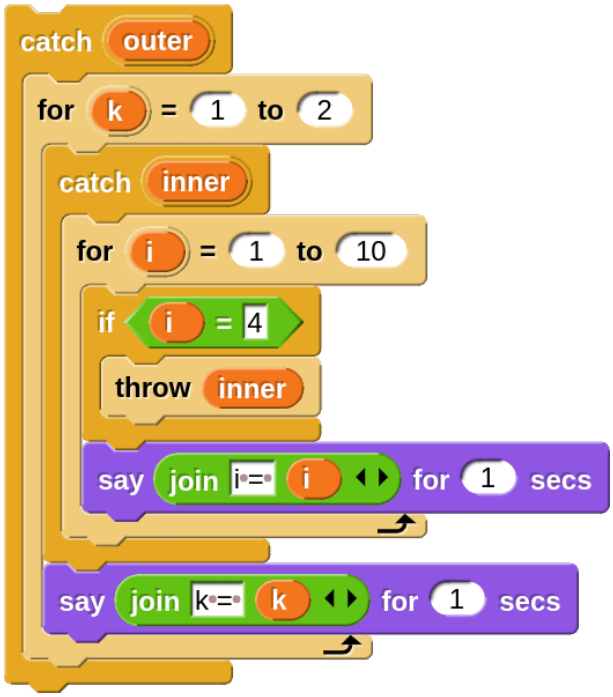
継続の機能を利用すると、C 言語風 for (; ;) ループに break や continue の機能を持たせることができます。break はループから脱出する機能で、continue はループの先頭に戻る機能です。



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



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



これを実行すると、i の値が 4 になった時に内側のループから脱出し、catch inner の次のスクリプトに進みます。 

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です。

9　その他

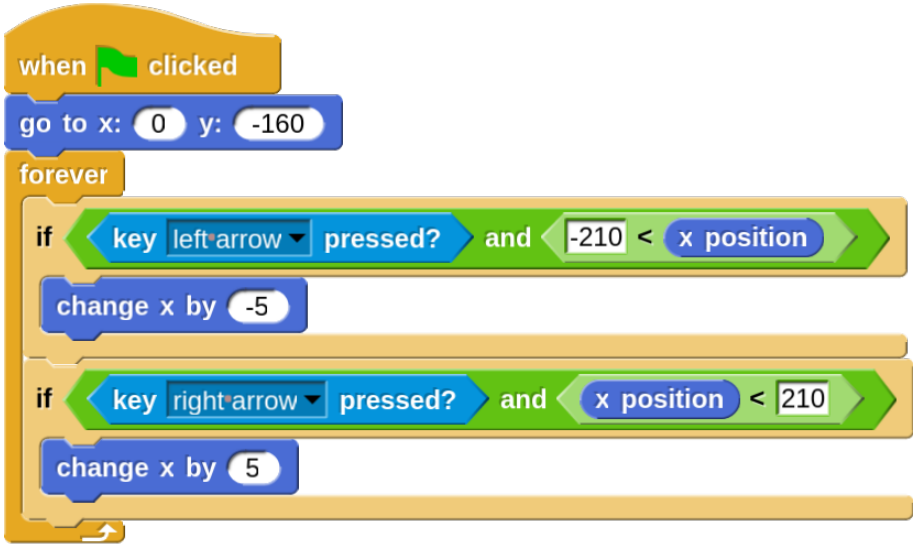
9.1　クローン

Snap! には、テンポラリクローンとパーマネントクローンがあります。

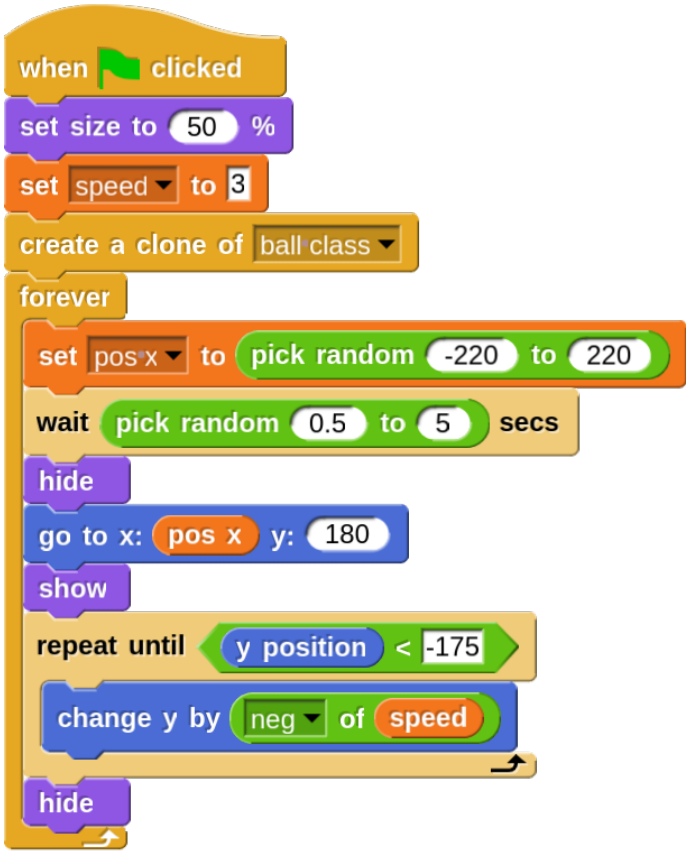
9.1.1　パーマネントクローン

パーマネントクローンは Scratch でおなじみのものとは違って、ストップボタンをクリックしても消滅しません。パーマネントクローンを使って、落ちてくるボールをキャッチするスクリプトを作ってみます。ball とキャッチするための bowl のスプライトを用意します。

bowl のスクリプトです。



ball のスクリプトです。使用する 2 つの変数は for this sprite only で作成します。

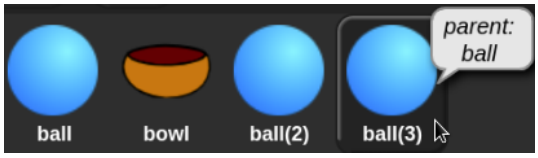


この段階ではキャッチの処理は入っていません。

スプライトコラルで ball のスプライトを右クリックすると、duplicate と clone の項目があります。比較のために両方を作成してみます。

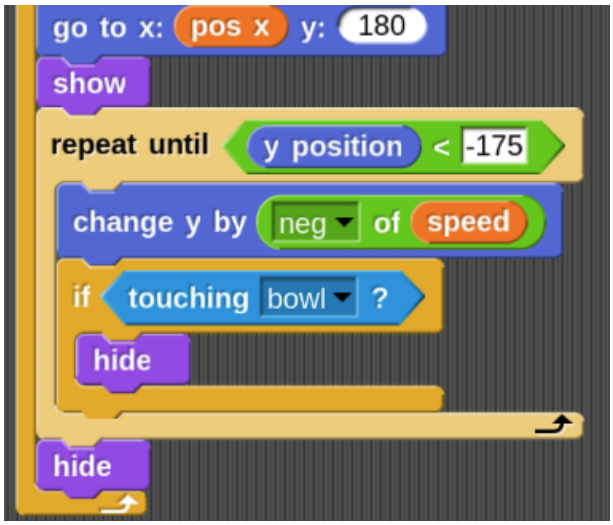
duplicate でできたのが ball(2) です。ただのコピーです。

clone でできたのが ball(3) です。ball(3) のところにマウスポインターを持っていくと parent 親が何になっているかを表示します。



作成されたもののスクリプトはどちらも ball のものと同じになっています。ボールの表示が同じだと区別がつかないので、別な色のコスチュームにしてみました。コスチュームの変更は親子関係に影響しません。

ball のスクリプトにキャッチされた時の処理を加えます。



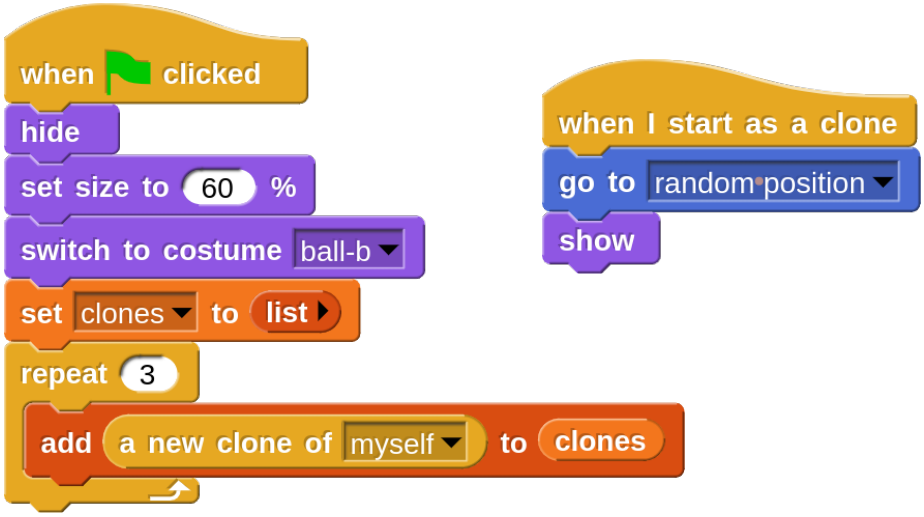
そうすると、clone である ball(3) にも同じスクリプトがセットされています。これが親 parent と 子 child の関係です。ただのコピーである ball(2) にはなんの変化もありません。ただし、子のスクリプトに変更を加えた場合、たとえば、size とか speed を変えるとかした場合には、子が独立したとみなされます。そうすると、親のスクリプトの変更が子のスクリプトに反映されなくなります。元の関係に戻すには 

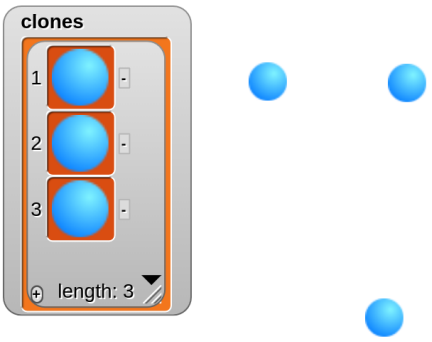
をクリックするか、子のスクリプトに対して inherit オプションを実行します。

9.1.2 テンポラリクローン

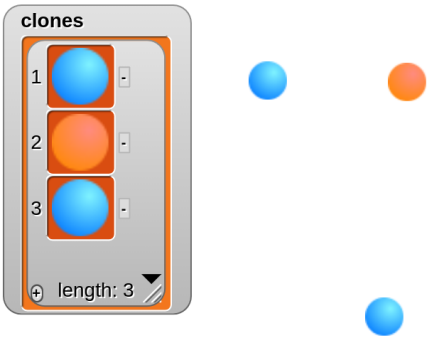
テンポラリクローンは Scratch でのクローンと同じものです。Snap! にはクローンを作成し、それをリポートするブロックがあるので、変数やリストに格納して、それを使ってクローンを操作することができます。

以下のようなスクリプトを作成すると、リストにクローンを格納することができます。





クローンを操作するにはこのようにすることができます。



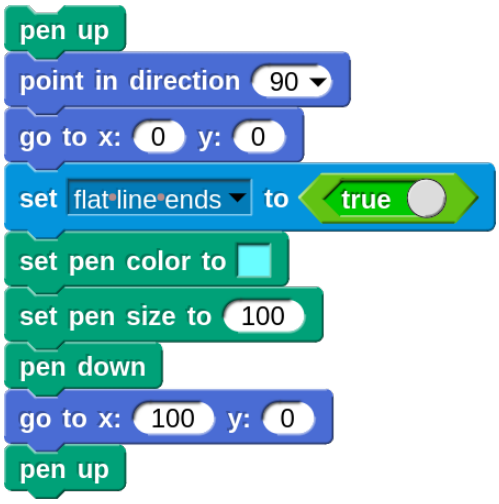
9.2 flat line ends

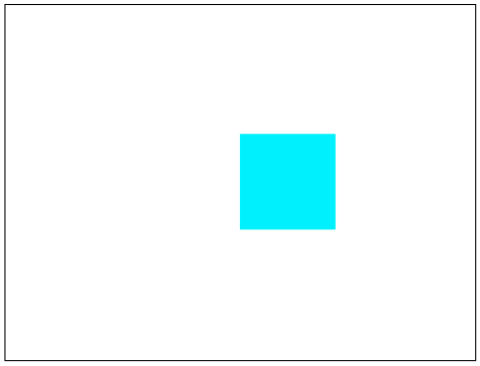
Sensing のパレットに 

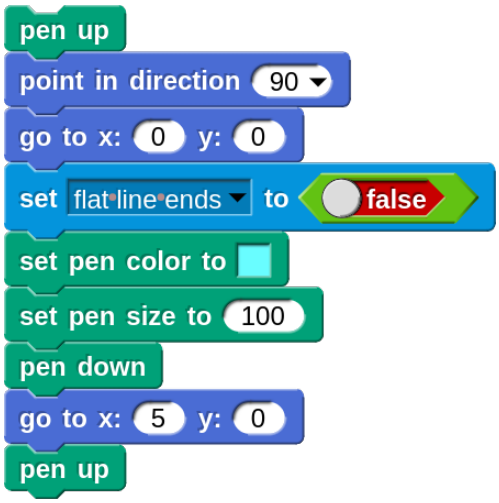
があります。

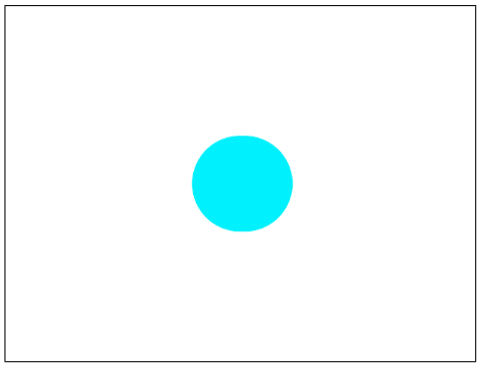
これを 

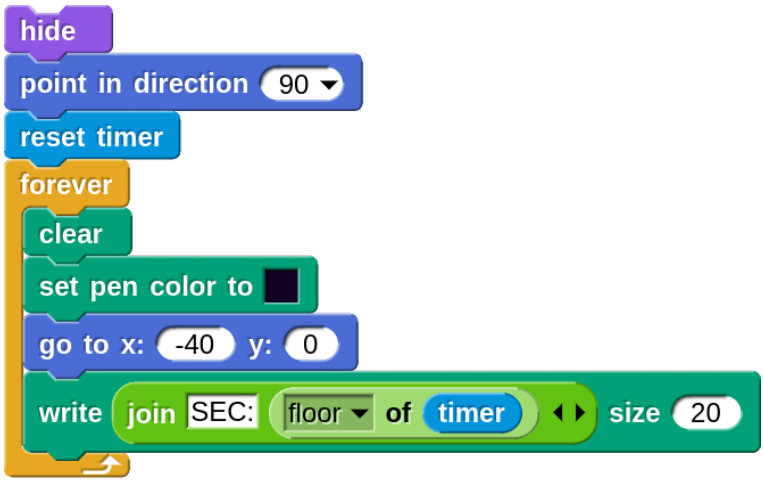
にすると、ペンで描く線の端を平ら (true) にするか丸く (false) するかを指定できます。





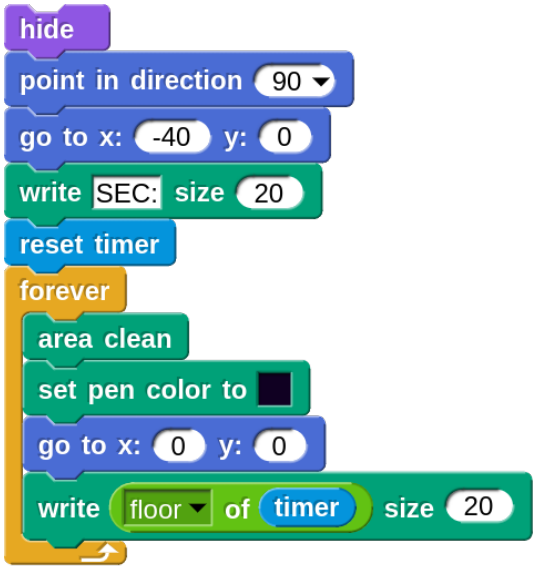


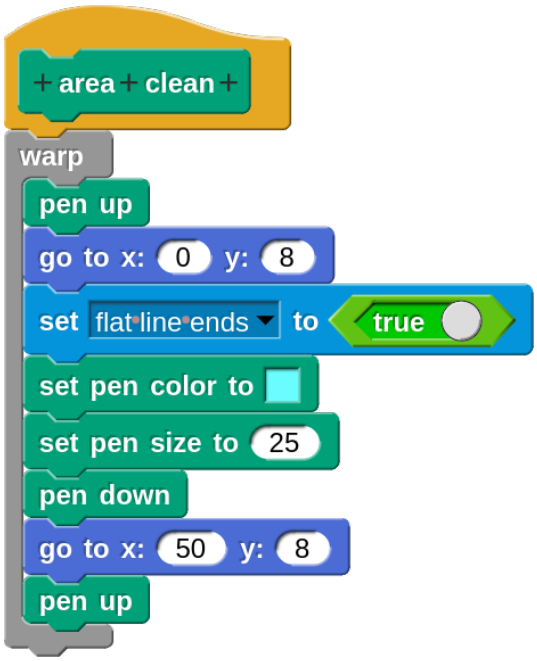




データの部分だけを消すやり方です。

ステージにデータを表示する場合に普通は変数ウォッチャーを使いますが、Pen を使って左のようにすることができます。ステージ全体を消しているので他の部分で Pen 描画をしている場合は、それらも消してしまいます。



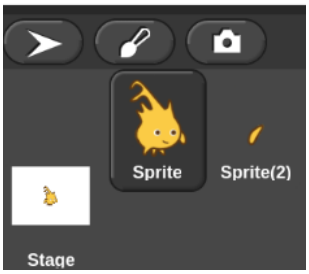


で色を指定している部分をクリックしてから、ステージ上のデータを表示させたいところでクリックするとそこの色を指定することができます。

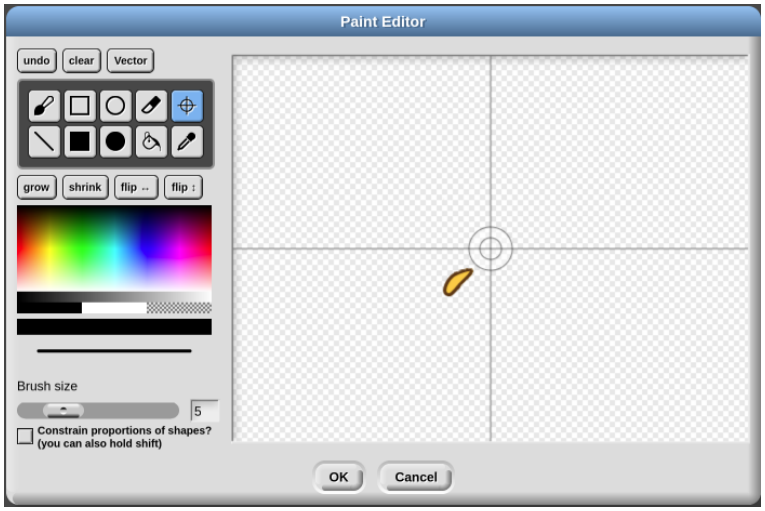
9.3 anchor アンカー

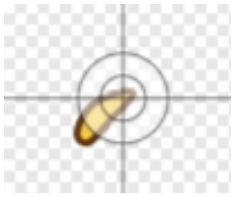
スプライトに関して、ボディーとパーツを別に用意してくっつけるというやり方があります。そうすると、パーツをボディーの一部として付随しながらパーツ自体の動きをさせることができます。

alonzo というコスチュームを利用して、ボディーと腕のパーツを作ります。そして、腕のパーツをボディーの取り付けたいところに持っていきます。

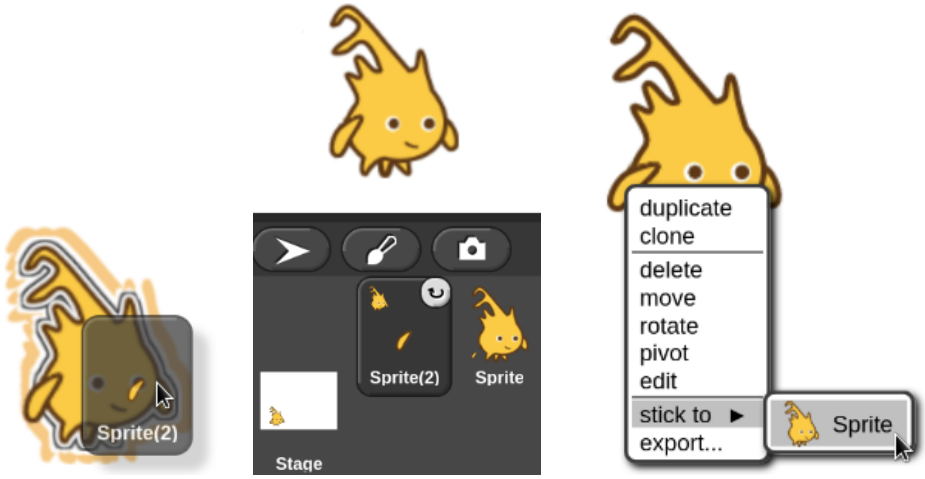


腕のパーツですが、コスチュームエディターで以下のように回転の中心を設定します。

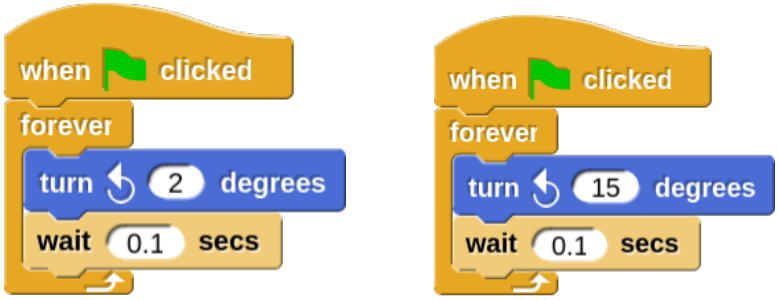




スプライトコラルにある腕のパーツのスプライトをステージにあるボディーのところへドラッグ&ドロップします。すると、スプライトコラルにある腕のパーツのスプライトの表示が変化します。ステージ上の腕のパーツを右クリックして、stick to ですることもできます。

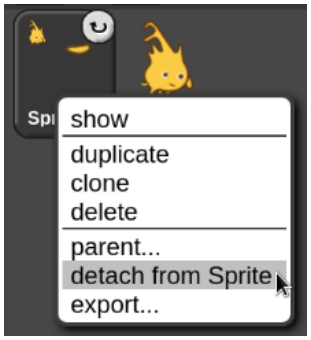


ボディー用のスクリプト　　腕用のスクリプト



こんなふうに腕を回しながらボディーも回転します。

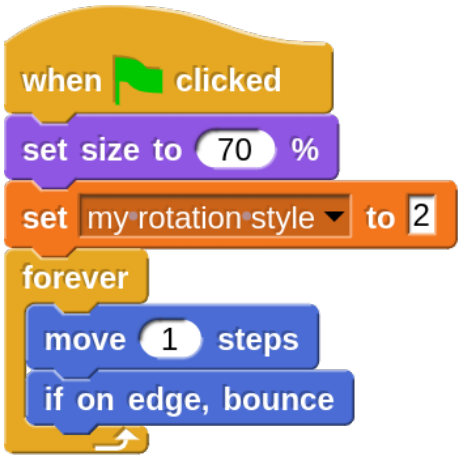
anchor の解除はスプライトコラルにある腕のパーツのスプライトを右クリックして detach を選択します。



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

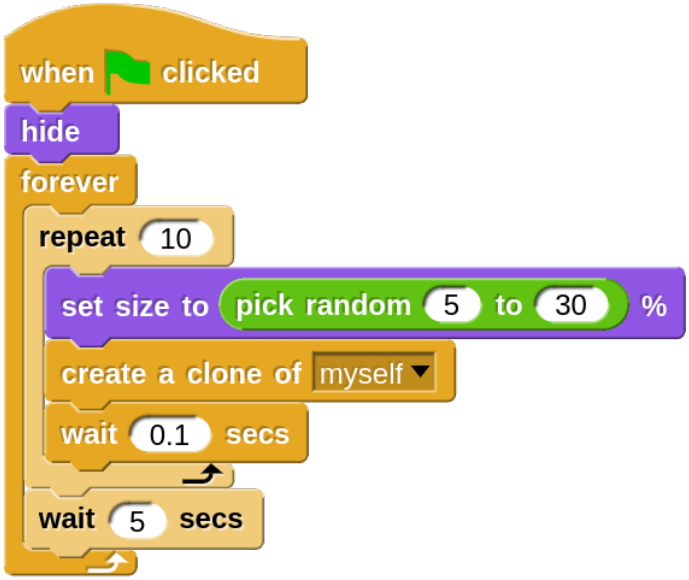
青いボールを anchor を使ってダイバーに付帯させて、吐いた息のように見せてみます。このように両者の y 座標を合わせてセットしてからボールをダイバーに stick させます。

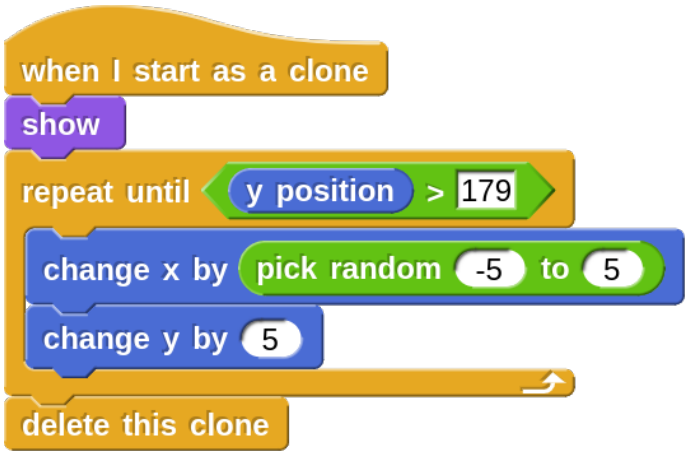
ダイバーのスクリプトです。



ボールのスクリプトです。ボール本体はダイバーに付帯していますが、クローンはその座標を起点にして浮上していきます。 

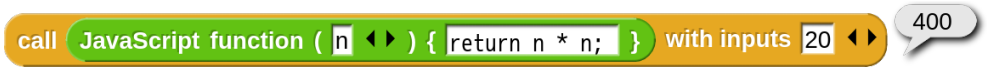
を使ってもできることではありますが。



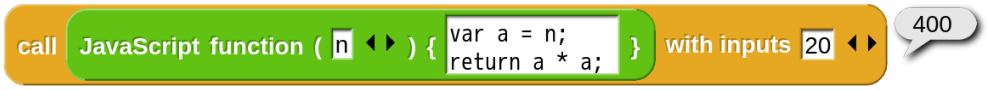


9.4 JavaScript function (オプション 5 ページ参照)

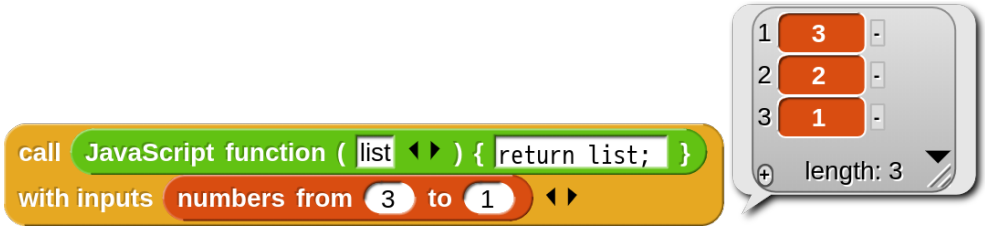
Snap! には、JavaScript コードを実行させるブロックがあります。次のようにすると数値をやり取りすることができます。call の入力スロットに入れたものが JavaScript の入力スロットに設定されて JavaScript 側からアクセスできるようになります。



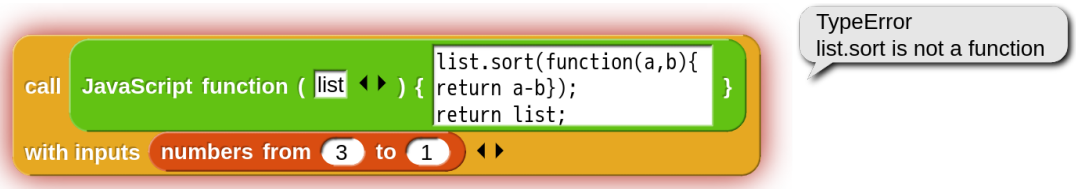
変数を使って演算をしてそれを返すこともできます。



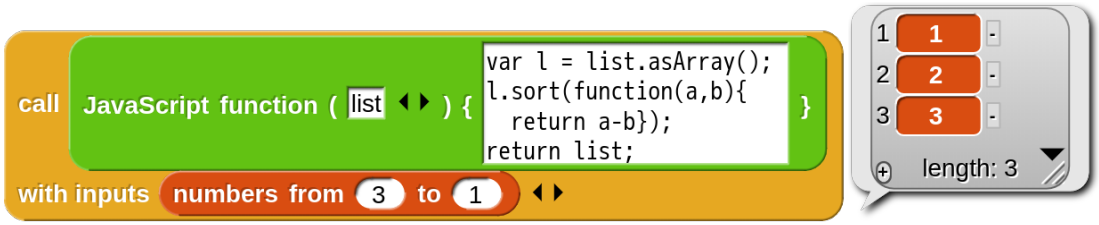
しかし、Snap! のリストはそのまま JavaScript 側で配列として操作することはできません。ただ返すだけならば問題ありません。



JavaScript の配列ソートを利用しようとするとエラーになります。

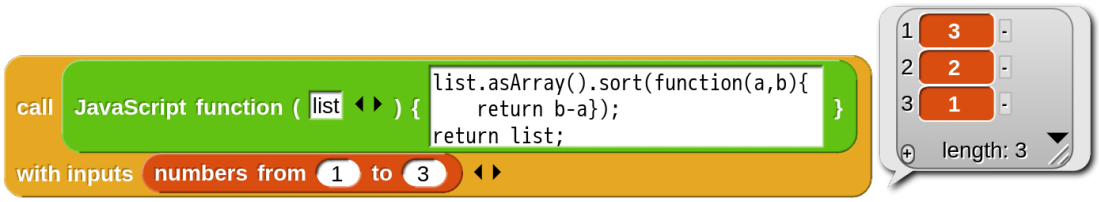


var l = list.asArray() で変換して配列にすると、配列として操作ができるようになります。 配列は参照型ということでなのか、結果的に list 自体がソートされるため、list を返すことができるようです。



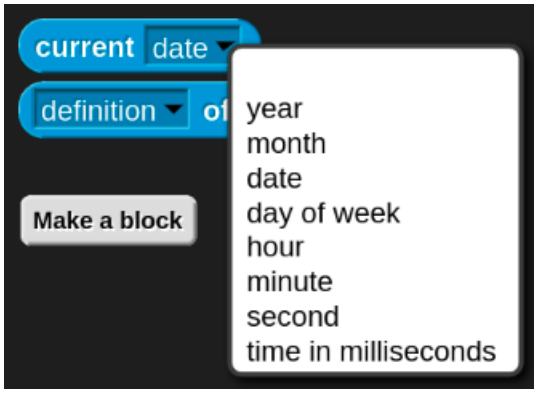
なお、return l; とすると、この場合「1,2,3」というものが返されます。これは、テストしてみると、number, list, text のチェックで false になります。つまり、数値でもリストでもテキストでもないということで使用できません。

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

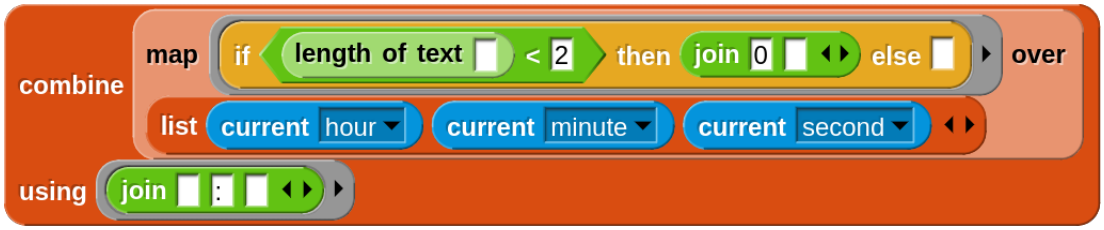


JavaScript function の使用例としてソートをやってみましたが、APL ライブラリーにソートブロックが用意されています。

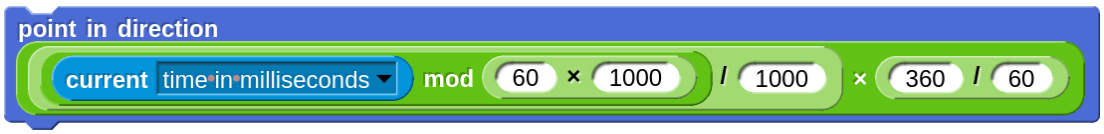
9.5　時計



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。

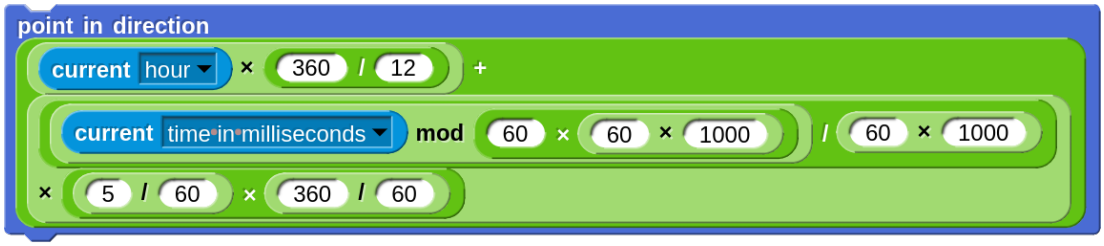


この時間のデータを各針の角度に変換すればアナログ時計ができます。針はスプライトで表現してもいいですし、その都度ペンで clear 描画を繰り返してもいいです。秒針をスムーズに動かしたければ、[current time in milliseconds] を使用すればできます。このブロックがリポートする値は、1970 年 1 月 1 日からの経過秒数です。(UTC 協定世界時) milliseconds とあるように、1/1000 秒単位の値になります。この値から秒のデータを取り出すには 1 分 (60 秒 × 1000 ミリ秒)、つまり 60000 で割った余りを求め、それを 1000 で割れば ○.○ 秒が得られます。



分針の角度です。

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



ペンで描く場合はこの逆の順序で描けば実際の時計と同じになります。

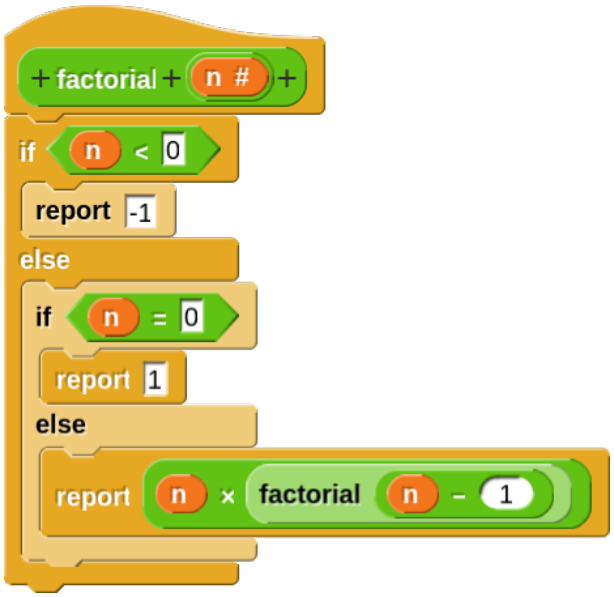
10　再帰

10.1　再帰の例

ブロックを作る時に再帰呼び出しを使うことがあります。いくつか例を示します。Scratch では値を返せなかったので、階乗やフィボナッチ数列はできませんでした。

10.1.1　階乗

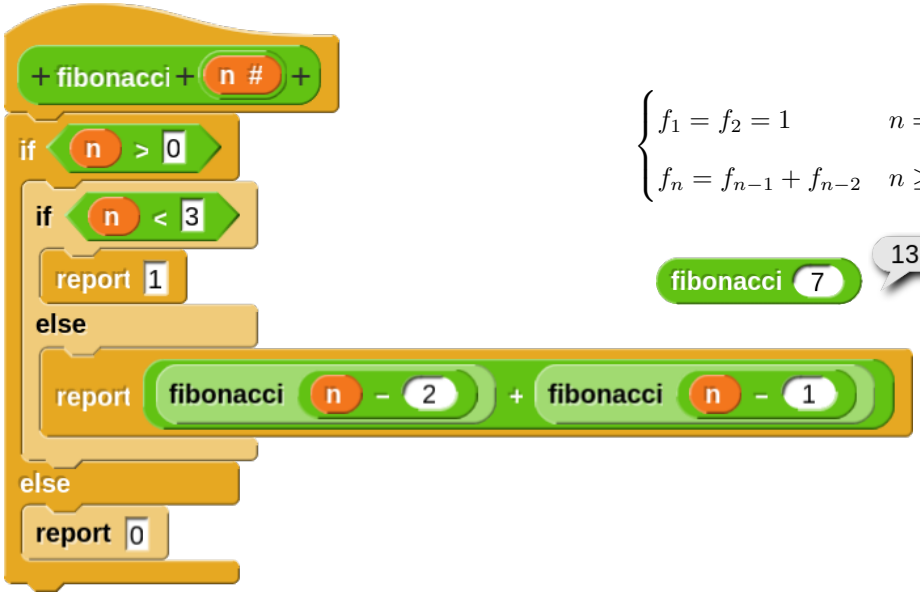
factorial 階乗は再帰の例としてよく使用されます。



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

10.1.2　フィボナッチ数列

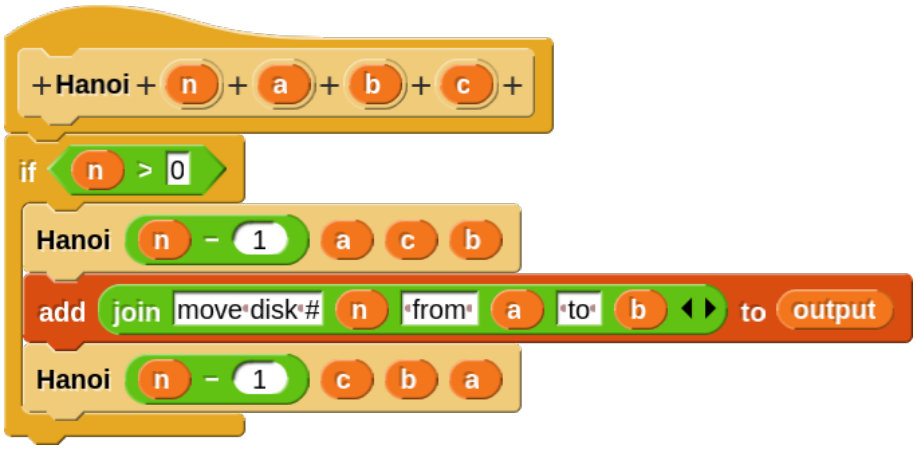
フィボナッチ数列も再帰の例としてよく使用されます。

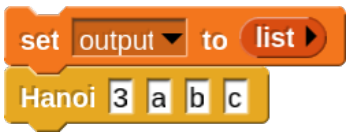


$$\begin{cases} f_1 = f_2 = 1 & n = 1, 2 \\ f_n = f_{n-1} + f_{n-2} & n \geq 3 \end{cases}$$

10.1.3　ハノイの塔

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



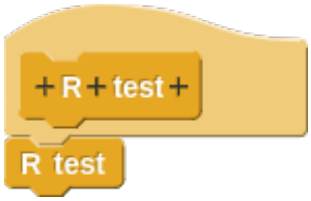


10.2　再帰の使用

例に示したものは特別で、普通は再帰などあまり使用するものでもないように思われますが、Scheme、Snap! では繰り返し処理の手法として再帰を使うのが一般的で、効率的だったりします。

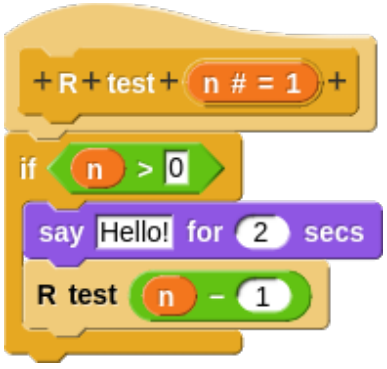
10.2.1　繰り返し

まずはただ自分自身を呼び出してみます。(実行しないでください)

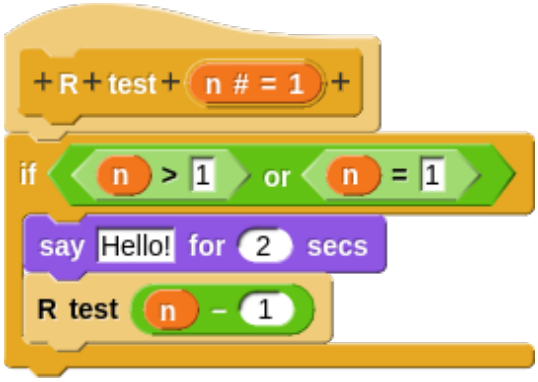


当たり前ですが、再帰呼び出しが終了するコードブロックがないので無限ループになります。普通の処理系ではスタックオーバーフローでエラー終了するのですが。

指定した回数だけ「Hello!」と言わせてみます。



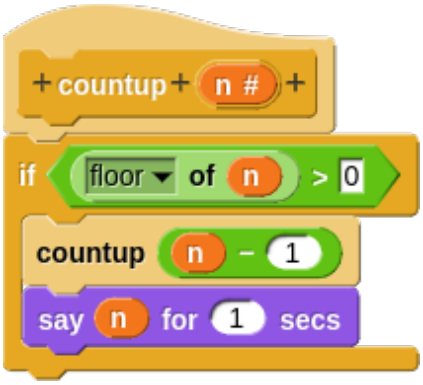
回数に 0.5 を指定しても実行されるので終了条件を変更します。



10.2.2 カウントダウンとカウントアップ

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

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



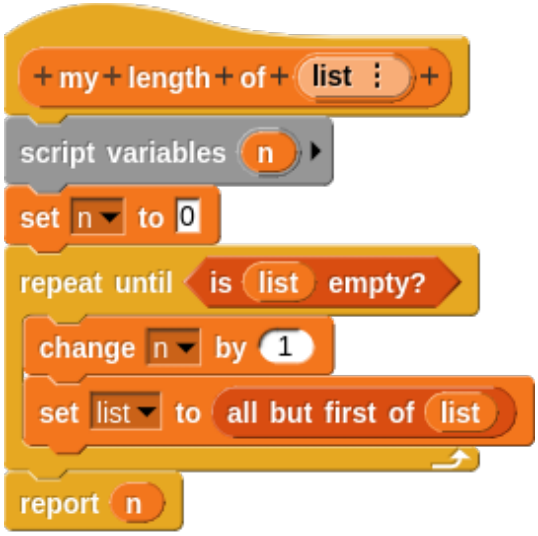
10.2.3 my length

リストの要素数を求める length ブロックを作ってみます。

普通に考えると for each ブロックを使うやり方になると思います。

処理にかかる時間を表示します。

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



処理にかかる時間を表示します。

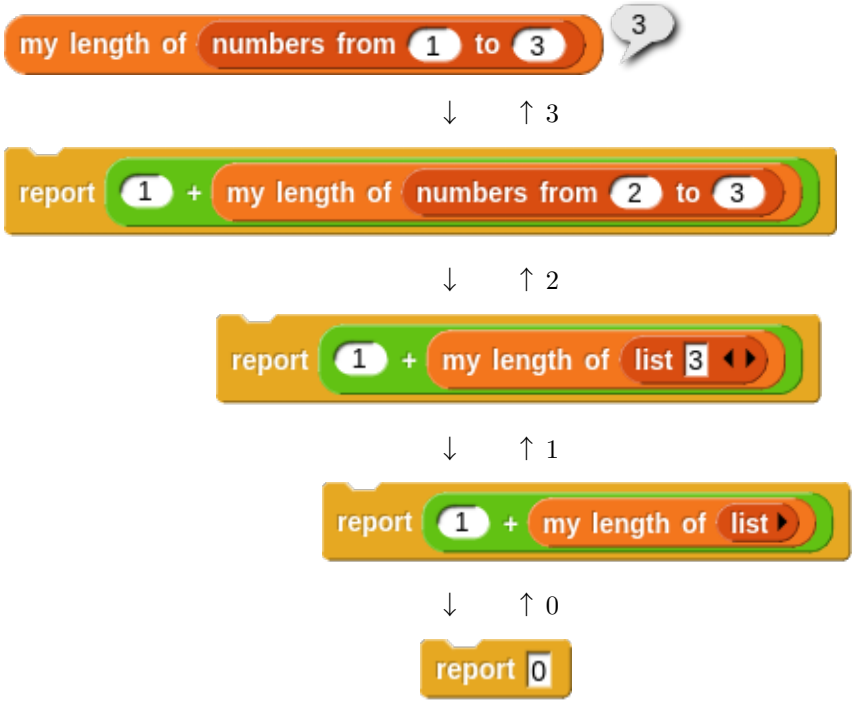
再帰版です。カウント用の変数が無いので理解しにくいですが、report が返す値がカウント用変数の役割を果たしています。 

でリストが空になるまで再帰呼び出しされて、0, 1, 2, ... と、report が返す値+1 を積み重ねて、結果的に 0 からのカウントアップで要素数を求めることができます。

処理にかかる時間を表示します。

再帰呼び出しを使わないものに比べてとても効率が良いことがわかります。

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



10.2.4 my contains

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

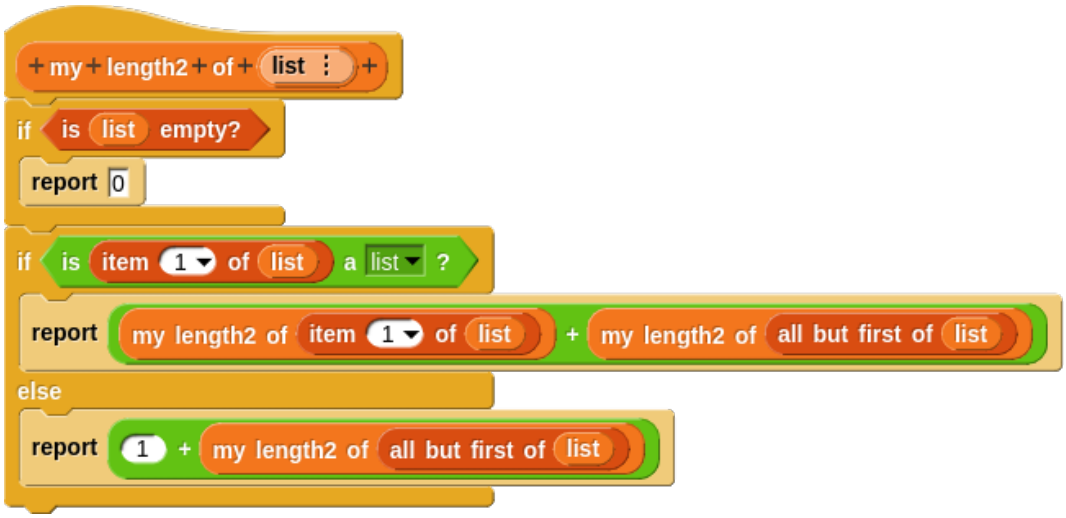
10.2.5 リスト要素の巡回

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

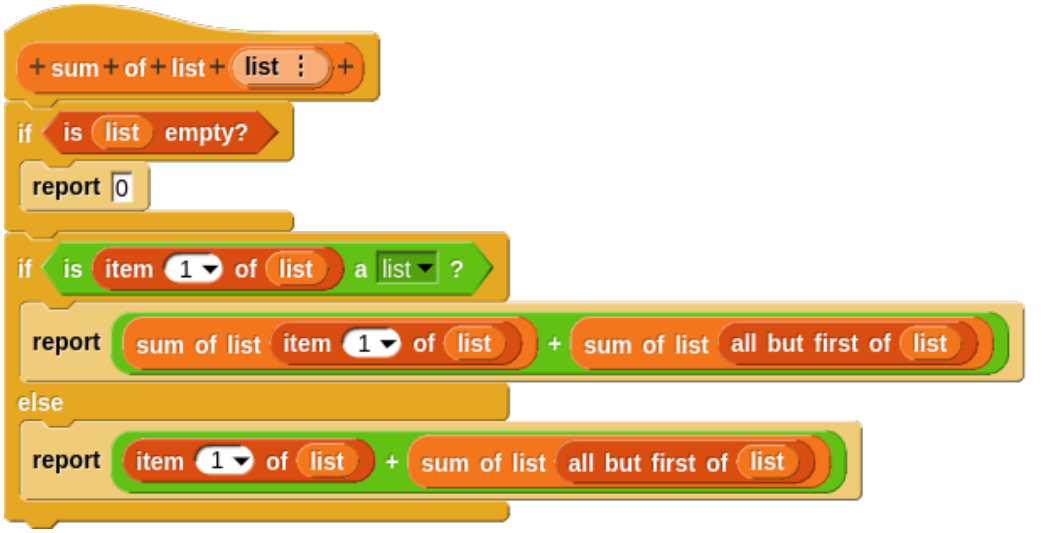
これは my length も同様です。

再帰を使って内部の要素に対してもアクセスしてみます。
処理の内容は次のようになります。

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



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



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

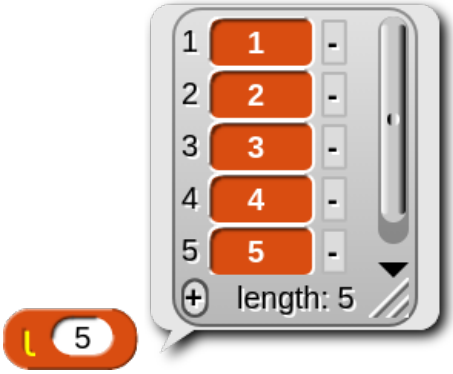
11　APL ライブラリー

APL ライブラリーをインポートすると、APL 言語的なブロックが使用できるようになります。基本的なことがらについてだけ説明します。ver. 6.9.0 時点ではライブラリー定義内で JavaScript ブロックが使われているものがいくつかありますから JavaScript extensions オプションを有効にする必要があります。(5 ページ参照)

11.1　形

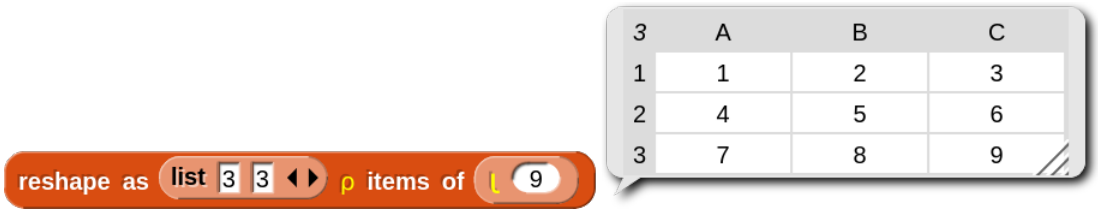
リストではない、ただの一個の数値や文字のデータをスカラーと言います。スカラーを一列に並べたものをベクトルまたはベクターと言います。要素にリストなどを含まないリストです。数学などで扱う、方向の要素を持ったベクトルではありません。(行列では要素ではなく成分と言うようです。)

ベクトルを生成するのに、

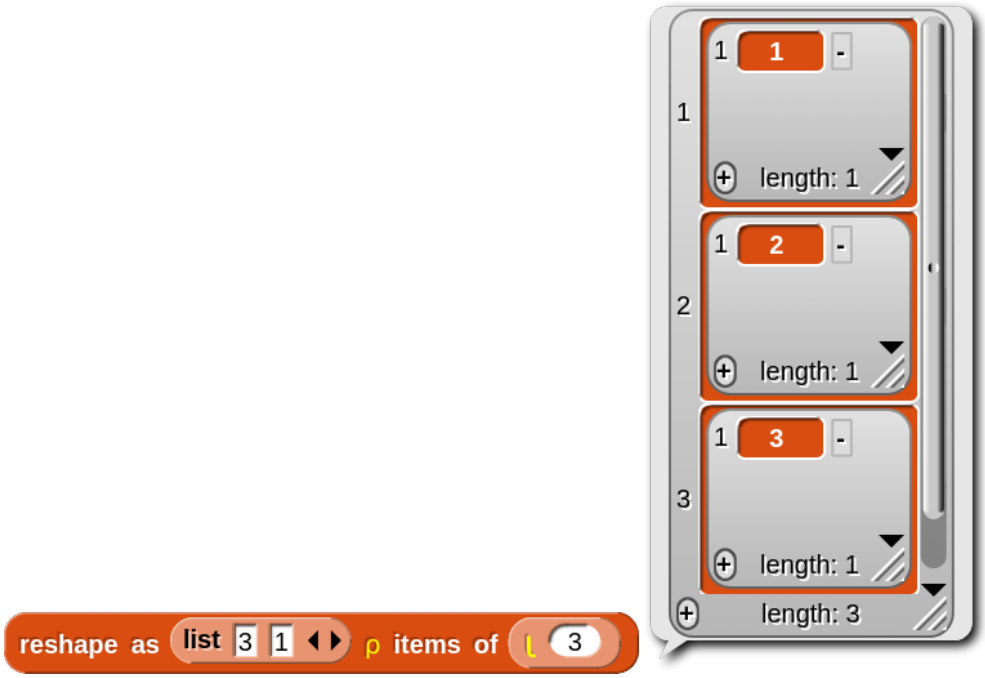


とすれば、1 から指定の数値までのベクトルが得られます。ちなみにこの記号のようなものはギリシャ文字の ι イオタです。

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

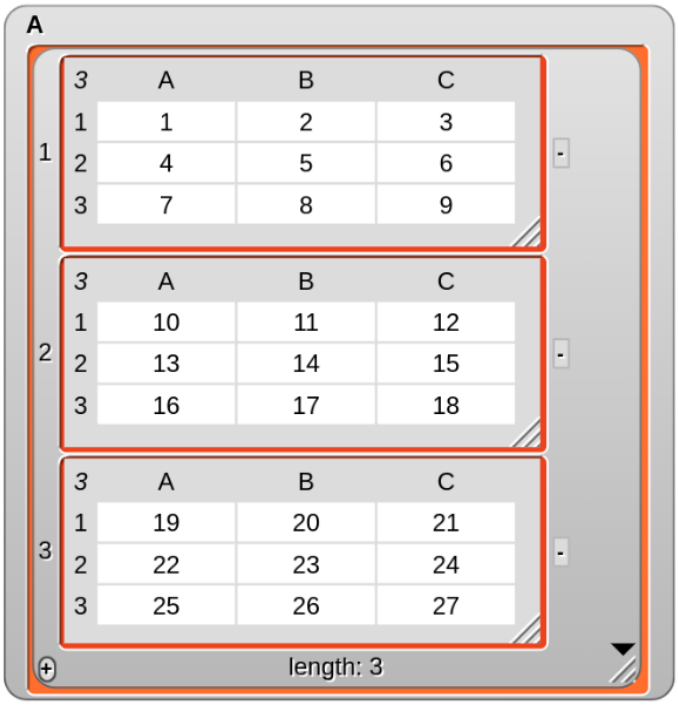


形はリストを使って指定します。次のように 1 行 3 列や、

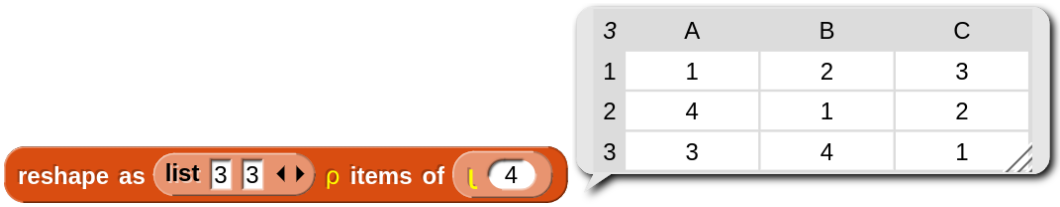


のように 3 行 1 列にすることもできます。

reshape のリストを次のようにすると、

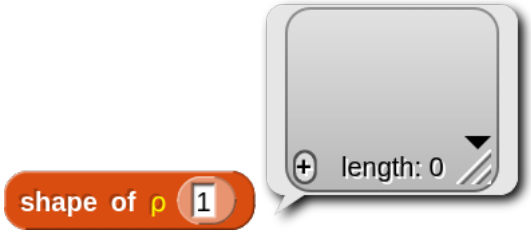


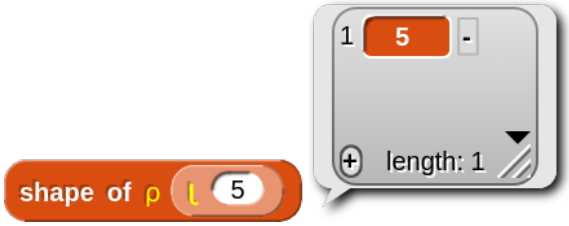
3 行 3 列の配列が三次元的に三個連なります。 ι イオタブロックで指定した値よりも配列の要素数が少ない場合は、ベクトルの残りは使用されません。逆に配列の要素数よりも少ない場合は、不足分をベクトルの先頭に戻って供給します。

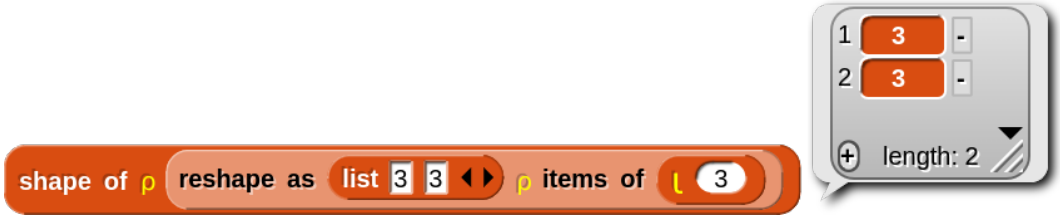


shape of ρ は指定されたものの形をリストで返します。

スカラーの形 shape は空です。

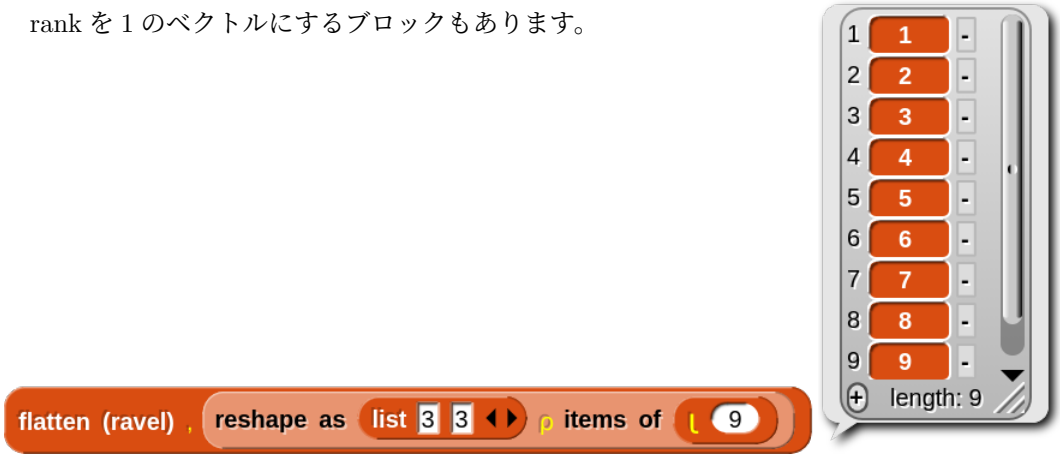






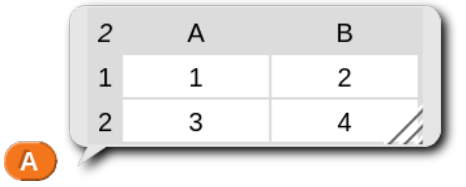
形 shape とは別に rank rank of ρρ というものがあります。これは、一次元、二次元、三次元のような階層を表す数値です。スカラーは 0 になります。

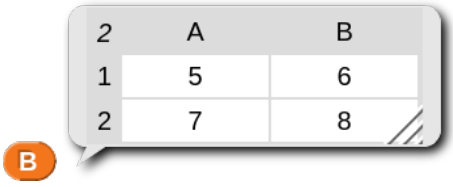
rank を 1 のベクトルにするブロックもあります。

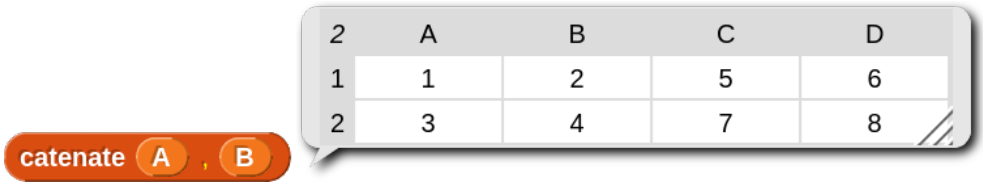


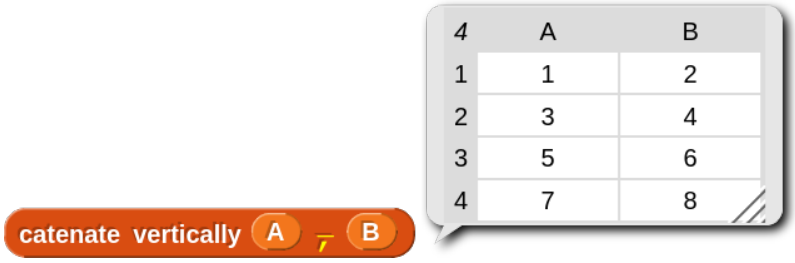
11.2 配列の連結

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





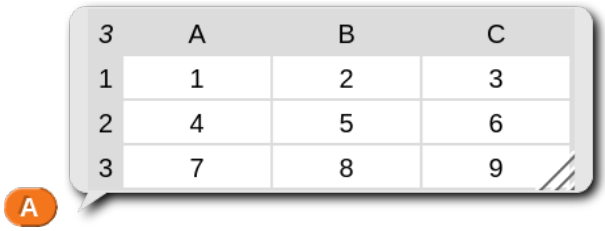




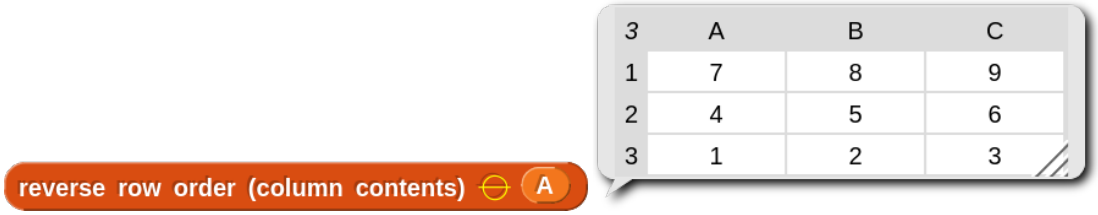
連結される箇所の双方の要素数が同じである必要があります。

11.3 配列要素の配置転換

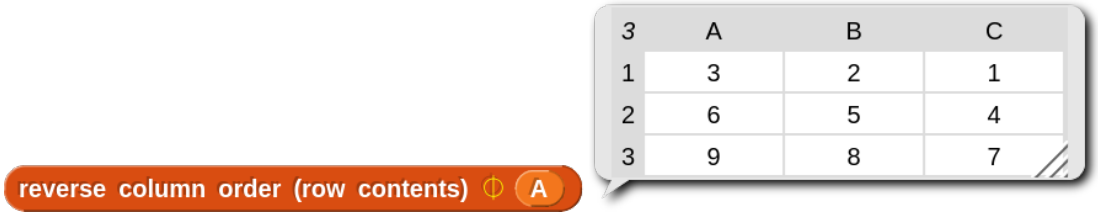
配列内の要素の位置を入れ替える演算子があります。変数 A に 3 行 3 列の配列をセットします。



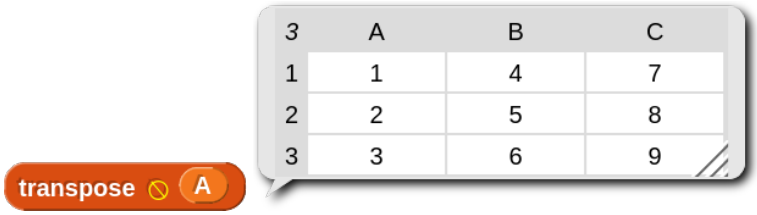
・記号が示しているように上下方向、垂直方向に入れ替えます。



・記号が示しているように左右方向、水平方向に入れ替えます。

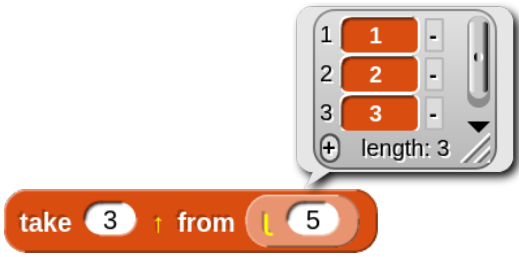


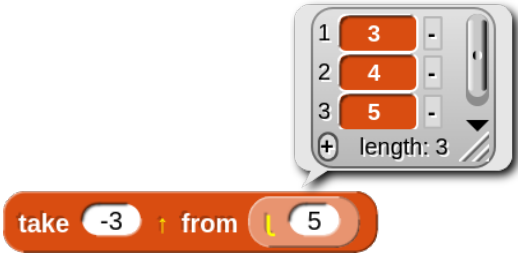
・記号が示しているように対角線を軸にして入れ替えます。（転置）



11.4　ベクトル、配列の範囲指定、選択

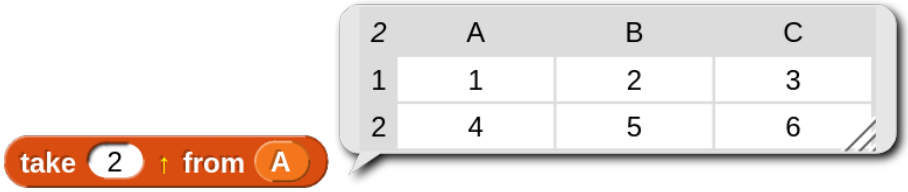
take は、ベクトルの先頭から指定した個数の要素のリストをリポートします。負の数で指定すると、ベクトルの最後尾から指定された絶対値の個数の要素のリストをリポートします。



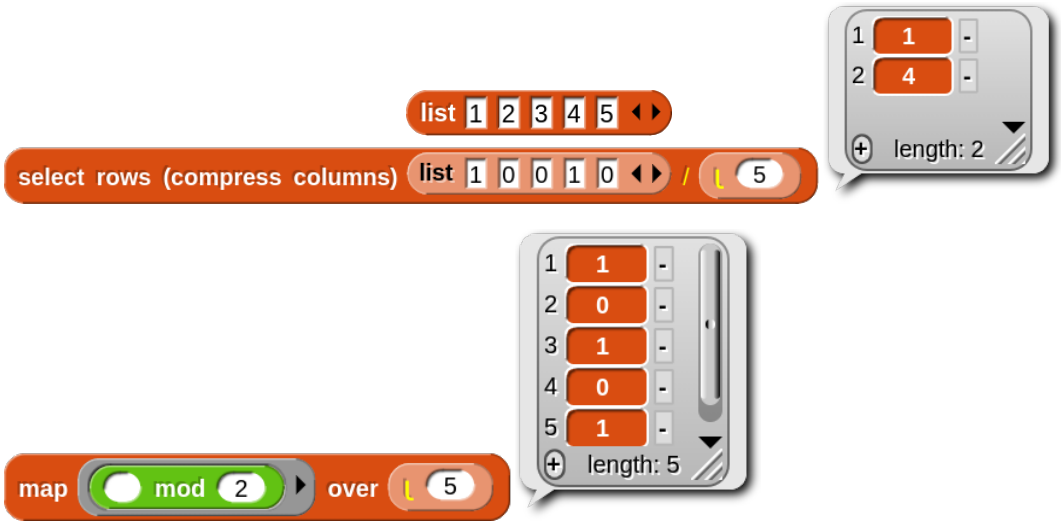


drop は、ベクトルの先頭から指定した個数の要素を除外し、残りの要素のリストをリポートします。負の数で指定すると、ベクトルの最後尾から指定された絶対値の個数の要素を除外し、残りの要素のリストをリポートします。

配列が指定された場合は、行単位の指定になるようです。

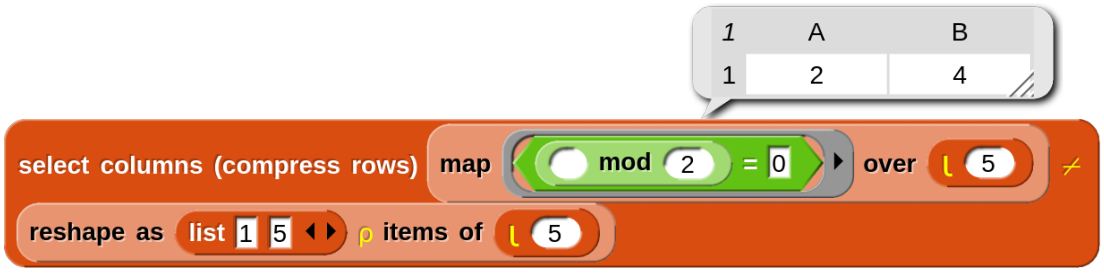


0(非選択)、1(選択) で指定リストにして要素を選択することができます。指定リストの要素数は選択される側の要素数と一致させる必要があります。

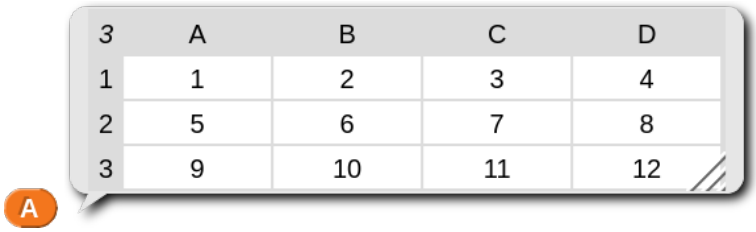


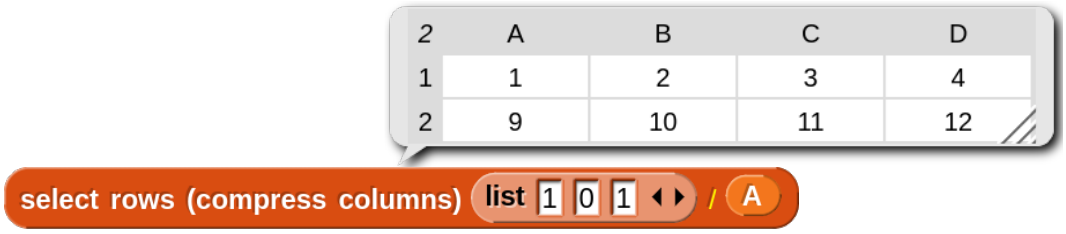
これを使用すると奇数番の要素を選択できます。

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



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





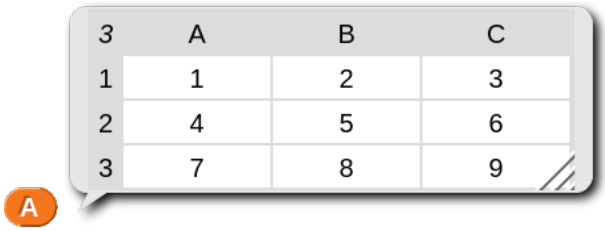
11.5 配列要素に対する演算

APL では、配列に対してスカラーの演算をすることができます。

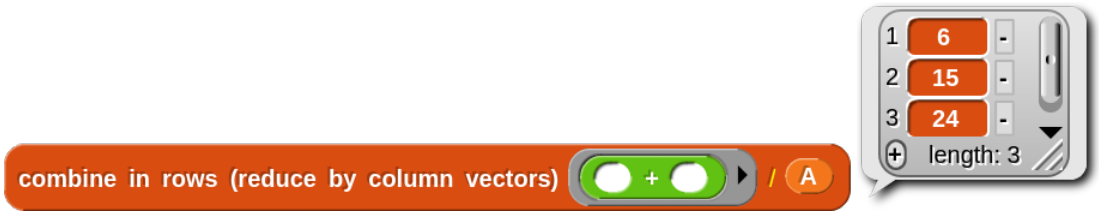
これは、スカラーの値 10 を 

のように同じ形の配列に変換 (整合) して、対応する配列要素同士で演算するようになっています。

の配列版が用意されています。



配列の 1 行内の各列の要素に対して指定された演算をします。



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

この場合の演算子は＋なので、1 行内の各列要素の合計になります。

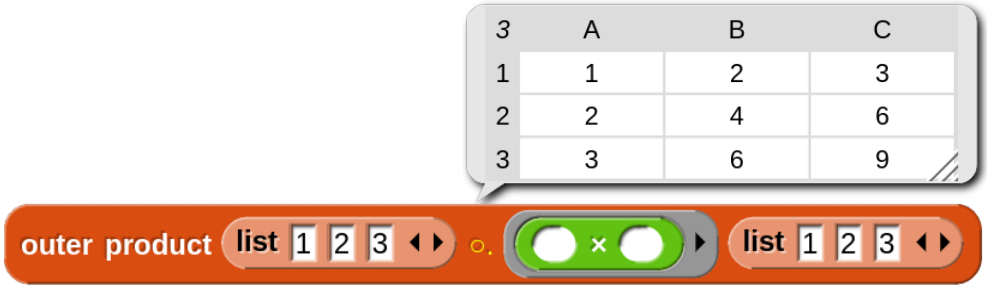
reduce(減らす)... 配列マトリックスだったものがベクトルにランクが減るということです。

配列の 1 列内の各行の要素に対して指定された演算をします。

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

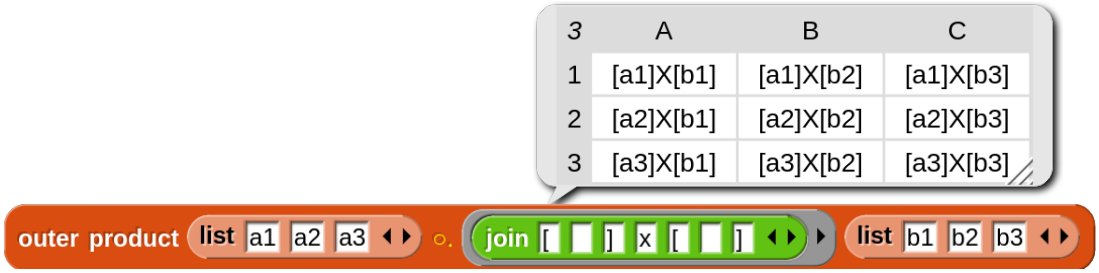
11.6　outer product

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



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

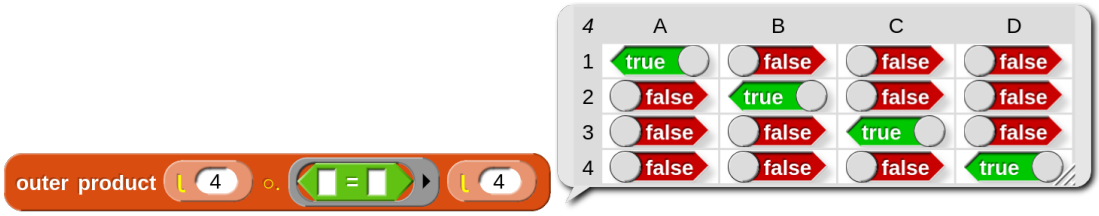
計算方法を表示させてみます。



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

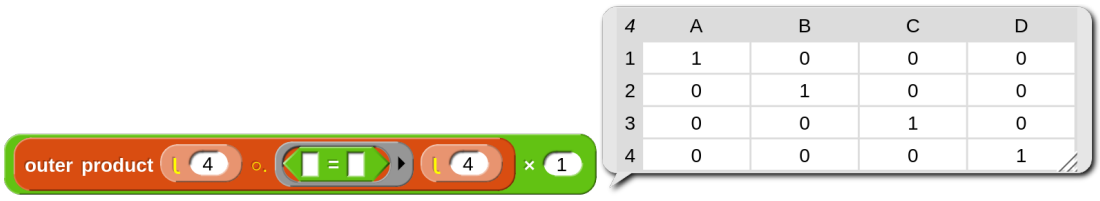
$$\begin{bmatrix}1\\2\\3\end{bmatrix} \bigcirc.\times \begin{bmatrix}1\\2\\3\end{bmatrix} \Rightarrow \begin{bmatrix}1\times1 & 1\times2 & 1\times3\\ 2\times1 & 2\times2 & 2\times3\\ 3\times1 & 3\times2 & 3\times3\end{bmatrix} \Rightarrow \begin{bmatrix}1 & 2 & 3\\ 2 & 4 & 6\\ 3 & 6 & 9\end{bmatrix}$$

単位行列なども作成できます。



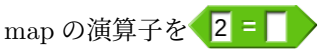
4	A	B	C	D
1	true	false	false	false
2	false	true	false	false
3	false	false	true	false
4	false	false	false	true

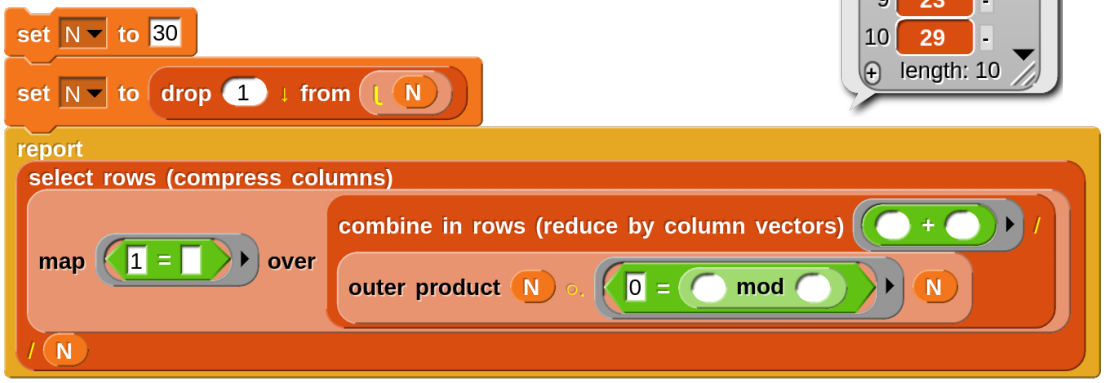
0, 1 で表すには、



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

『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
の 101 ページから　「6.5　適用例 — 素数を求める問題」が解説されています。APL ライブラリーの使用例としてとても参考になります。

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



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

一番内側の outer product ブロックから、結果を表示させながら順に一層ずつブロックを構築していくと、スクリプトの仕組みが理解できるかもしれません。

11.7 inner product

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

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

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

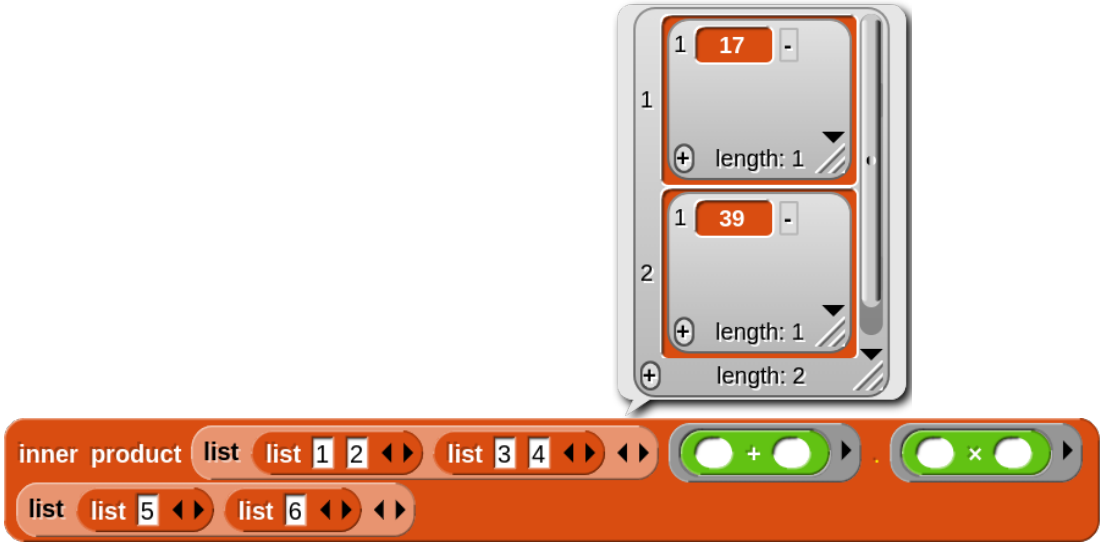
ブロックで求めることができます。

2 行 2 列配列同士では次のような計算方法になります。

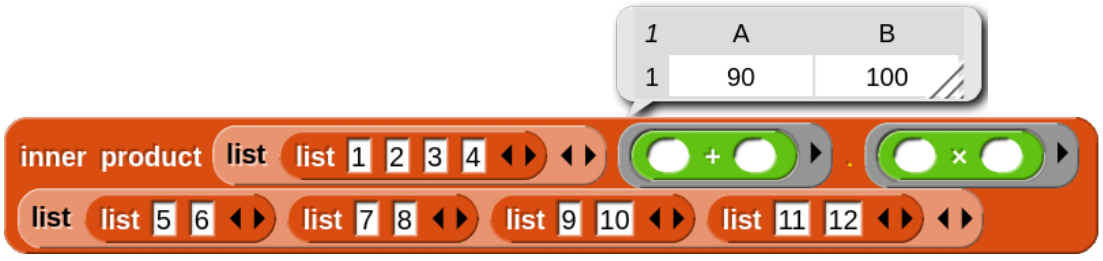
$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} a \times e + b \times g & a \times f + b \times h \\ c \times e + d \times g & c \times f + d \times h \end{bmatrix}$$

左側の配列の列数と右側の配列の行数は同じ必要があります。

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



$$\begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix} \begin{bmatrix} 5 & 6 \\ 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix}$$
$$= \begin{bmatrix} 1 \times 5 + 2 \times 7 + 3 \times 9 + 4 \times 11 & 1 \times 6 + 2 \times 8 + 3 \times 10 + 4 \times 12 \end{bmatrix}$$
$$= \begin{bmatrix} 90 & 100 \end{bmatrix}$$



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) を中心にして半時計回りに θ度回転させた座標を求めるものです。

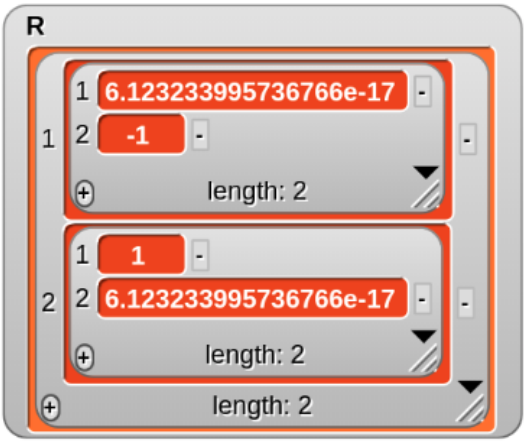
$$\begin{bmatrix} cos\theta & -sin\theta \\ sint\theta & cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

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

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

左側の配列、回転させるため係数を作成するブロックです。

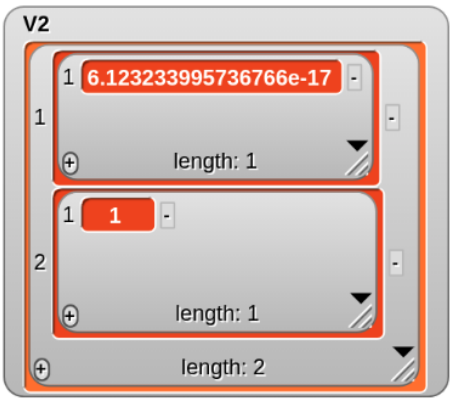
それを表示すると、次のような値の要素の配列になります。6.123233995736766e-17 は 0 とみなします。



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

この配列 $\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$ と座標 (x, y) $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ の積

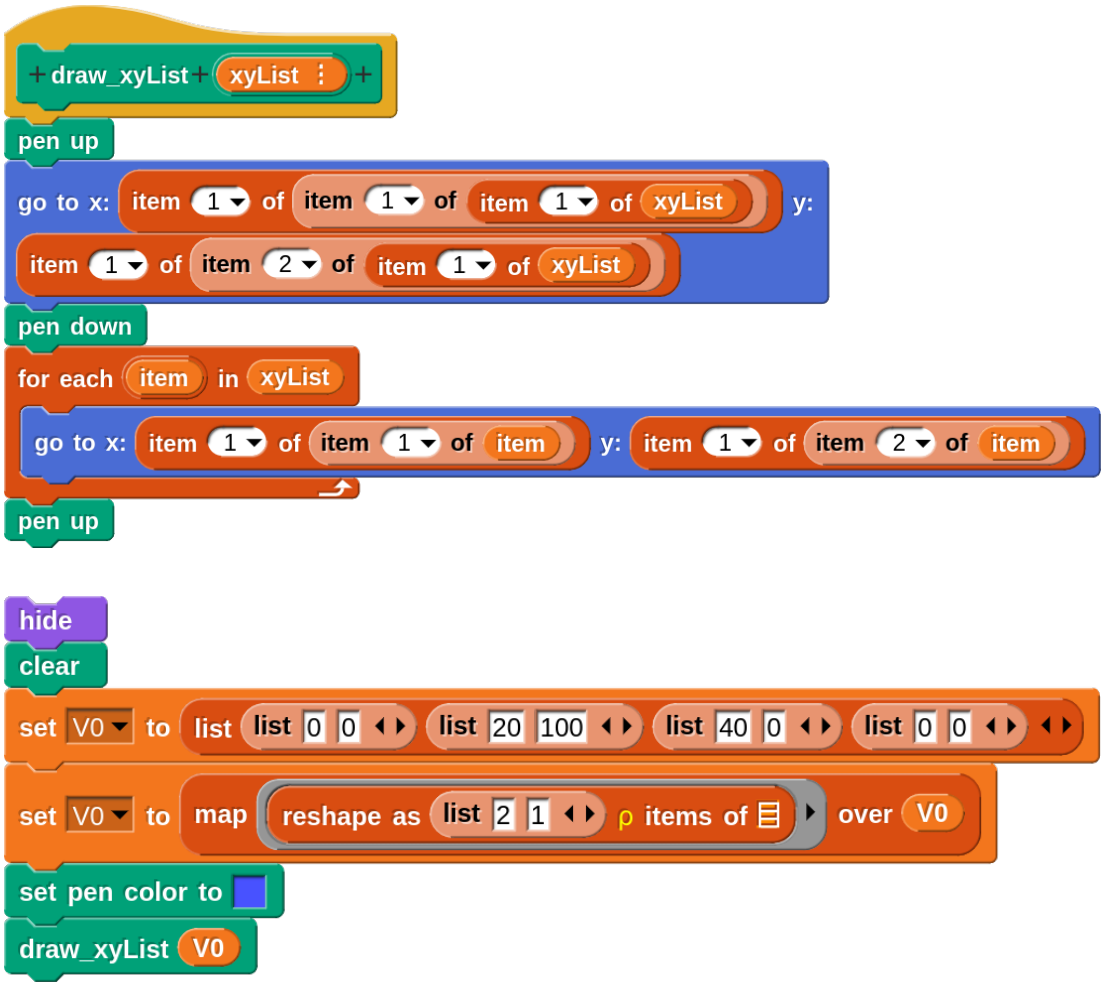
$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 \times 1 + (-1) \times 0 \\ 1 \times 1 + 0 \times 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ を求めます。



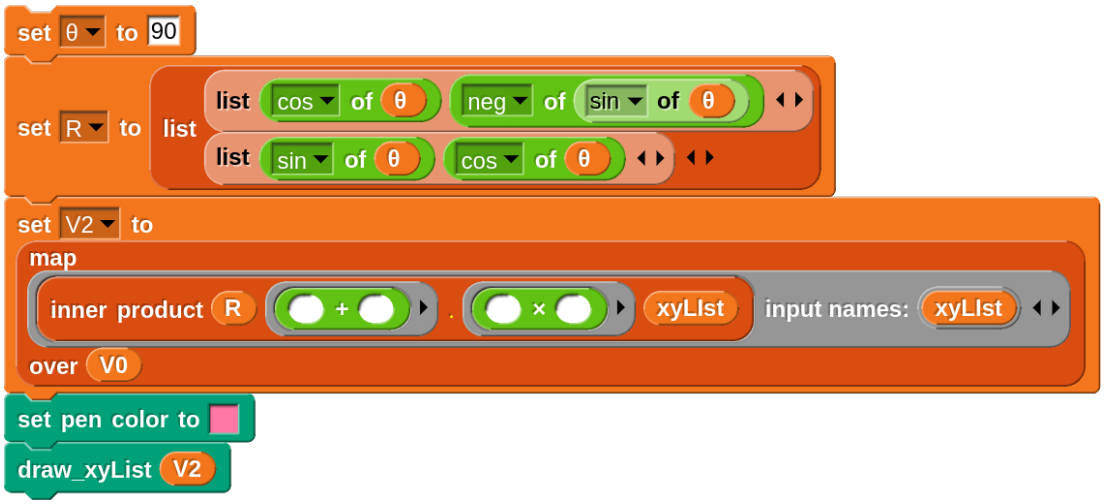
結果として、座標 $\begin{bmatrix}1\\0\end{bmatrix}$ を 90 度回転させた座標 $\begin{bmatrix}0\\1\end{bmatrix}$ が求められました。

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

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



これを実行すると、  が表示されます。



これを実行すると、  回転後の図形が追加表示されます。

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

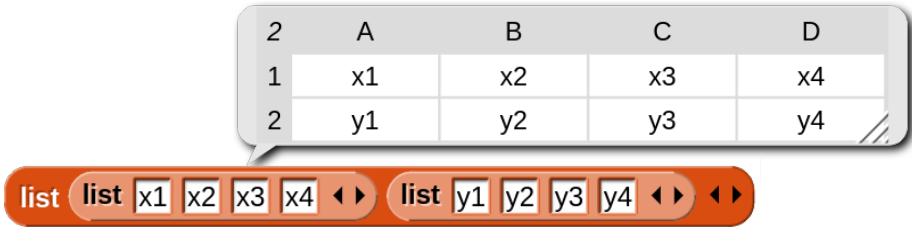
行列の積の定義により、左側の配列による変換が右側のそれぞれの列に対して行われます。

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

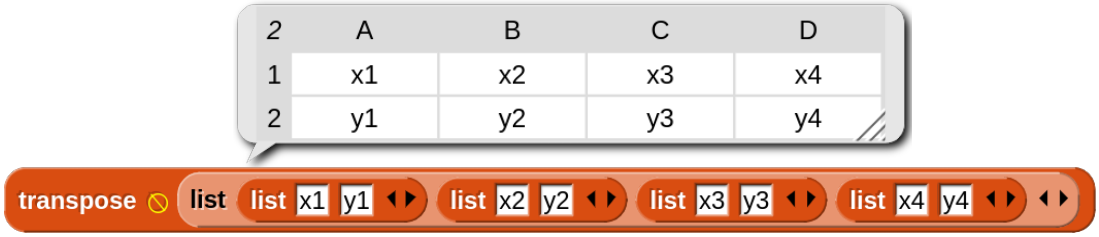
つまり、右側の配列は列を増やしていくとそれぞれの列に対して同じように変換されるということです。先の例では三角形の座標 (x, y) のリストから各点の座標を取り出して個別に変換していましたが、

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x1 & x2 & x3 & x4 \\ y1 & y2 & y3 & y4 \end{bmatrix}$$

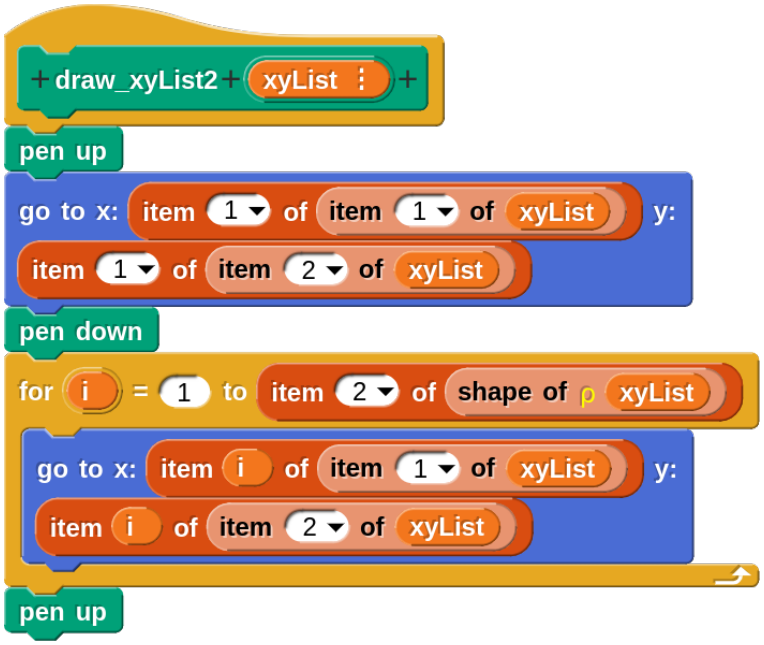
で、一括して変換することができます。ただし、このような形の配列で指定するには、x 座標, y 座標を 2 行に分けて指定する必要があります。



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

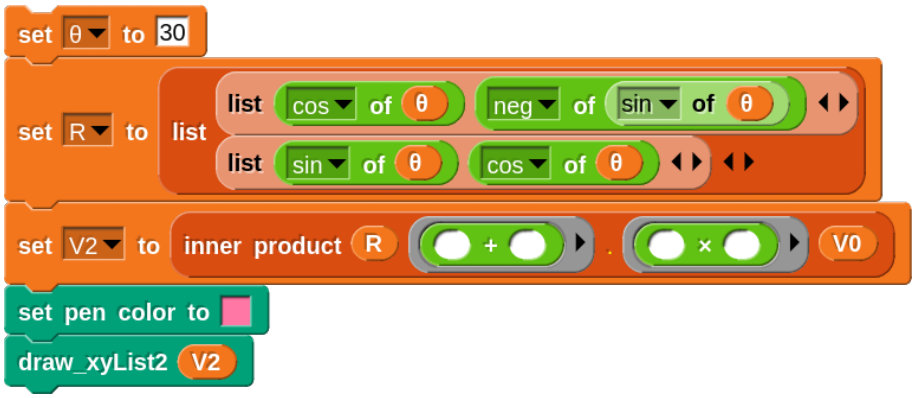


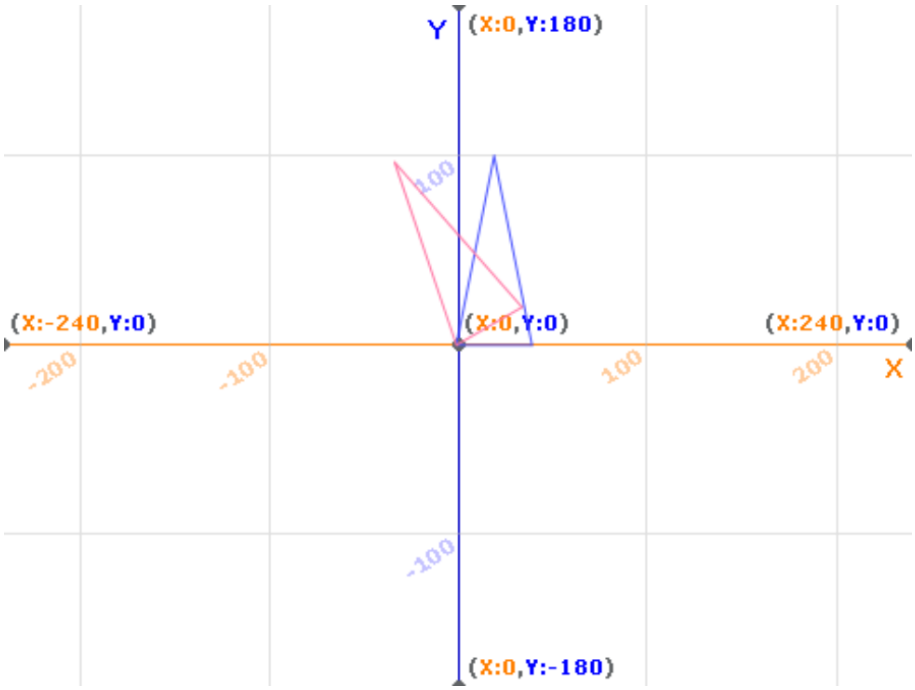
また、このような形の配列で指定された座標の図形を描画するには draw_xyList も変更しなければなりません。



変換前の図形描画スクリプトです。

30 度半時計回りに回転するよう変換して図形描画するスクリプトです。





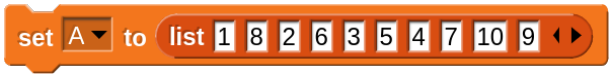
［参考文献］

『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』

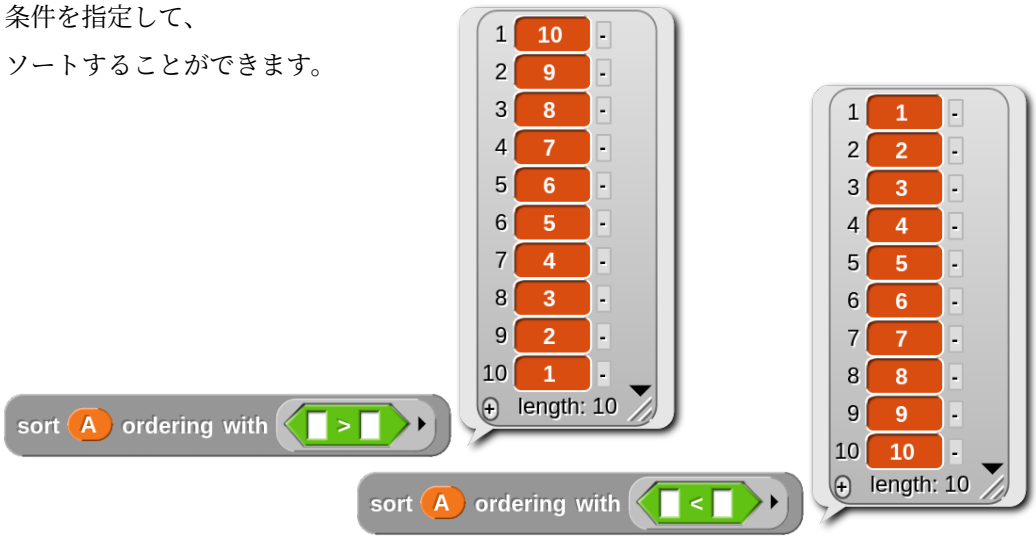
Peter Farrell 著　鈴木幸敏　訳　オライリー・ジャパン　刊

11.8　ソート、順位付け



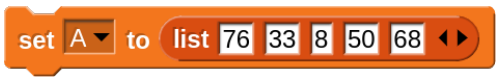
とした場合、

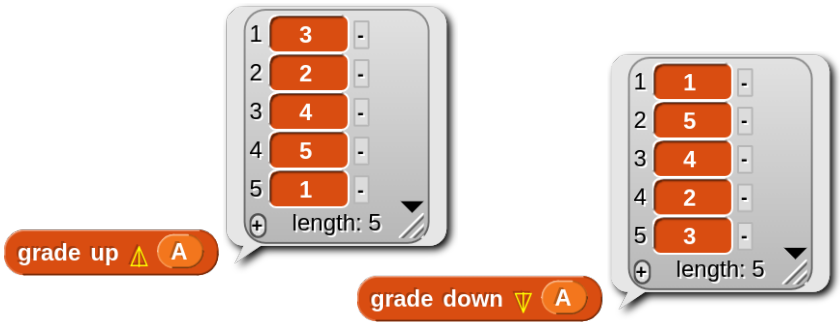
条件を指定して、
ソートすることができます。



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

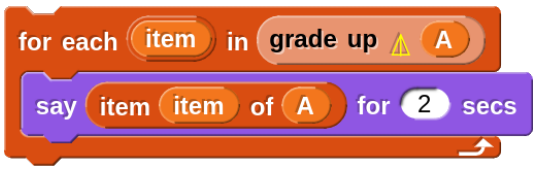
記号が示しているように、昇順降順です。





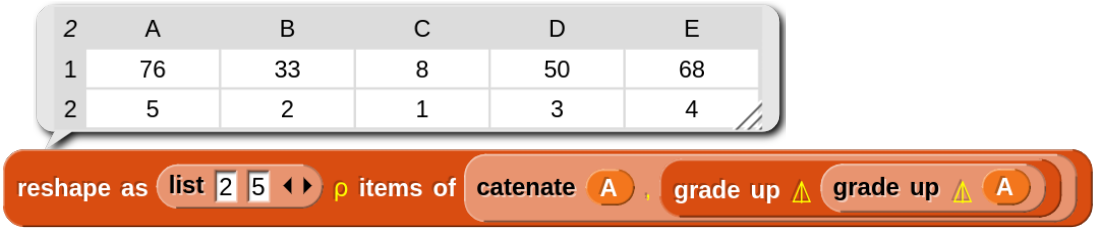
grade up(昇順) の場合は 1 番小さいのは A リストの 3 番目の 8、2 番めは A リストの 2 番目の 33 なので、リストは (3, 2, 4, ..) となります。grade down(降順) の場合は 1 番大きいのは A リストの 1 番目の 76、2 番めは A リストの 5 番目の 68 なので、リストは (1, 5, 4, ..) となります。

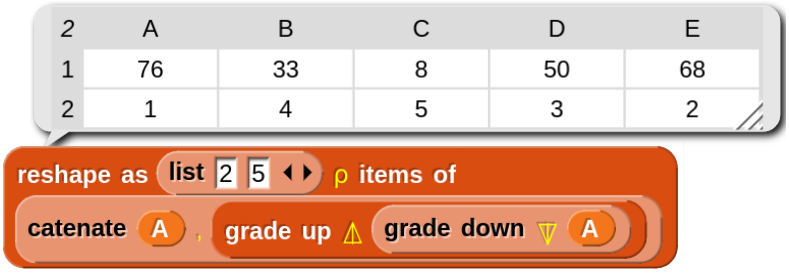
これを使ってソートされたリストを表示するのであれば次のようにする必要があります。



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

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





索 引