

Snap! のこと

齋藤文康

2024 年 6 月 17 日

Snap! Build Your Own Blocks(ver. 9) の使い方について、Scratch に準じているので、基本的なことは省いて、私が説明できることだけを取り上げました。マニュアルのようにすべての事柄を説明することはできません。

見た目は Scratch に似ていますが、Snap! Build Your Own Blocks が示すように、Scratch とは異次元のブロックが作れるという楽しみがあります。ステップ実行などデバッグのための機能もあるので、プログラミングやアルゴリズムの実験にも有益と思われます。

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

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

目 次

| | |
|---|----|
| 1 始め方 | 5 |
| 2 画面まわり | 5 |
| 2.1 エリア | 6 |
| 2.1.1 ステージエリア | 6 |
| 2.1.2 スプライトコラル | 7 |
| 2.1.3 スクリプトエリア | 7 |
| 2.1.4 パレットエリア | 7 |
| 2.2 エリアの大きさ | 8 |
| 2.3 実行に関するボタン | 8 |
| 2.4 ブロック表示 | 11 |
| 3 キーボード入力 | 12 |
| 4 変数 | 13 |
| 4.1 for all sprites (全部のスプライトで使えるグローバル変数) | 13 |
| 4.2 for this sprite only スプライト変数 | 15 |
| 4.3 script variables スクリプト変数 | 16 |
| 4.4 for ループ変数 | 17 |
| 4.5 リスト操作用ブロック | 18 |
| 4.5.1 numbers form () to () | 19 |
| 4.5.2 () in front of () | 20 |
| 4.5.3 all but first of () | 20 |
| 4.5.4 index of () in () | 21 |
| 4.5.5 append () () | 21 |
| 4.5.6 for each () in () | 21 |
| 4.5.7 reshape () to () () | 22 |
| 4.5.8 map () over () | 23 |
| 4.5.9 keep items () from () | 25 |
| 4.5.10 find first item () in () | 26 |
| 4.5.11 combine () using () | 26 |
| 4.5.12 combinations () () | 27 |
| 4.6 オプションのリスト操作 | 28 |
| 4.6.1 rank of (), dimensions of () | 28 |
| 4.6.2 flatten of () | 29 |
| 4.6.3 columns of () 転置行列 | 29 |
| 4.6.4 uniques of () | 30 |
| 4.6.5 distribution of () | 30 |
| 4.6.6 sorted of () | 30 |

| | | |
|----------|--|-----------|
| 4.6.7 | shuffled of () | 31 |
| 4.6.8 | reverse of () | 31 |
| 4.6.9 | Σ of () | 31 |
| 4.6.10 | text of () | 31 |
| 4.6.11 | lines of () | 32 |
| 4.6.12 | csv of () | 32 |
| 4.6.13 | json of () | 32 |
| 4.7 | リストの演算 | 32 |
| 4.8 | 変数に入れられるもの | 34 |
| 5 | Control 制御 | 36 |
| 5.1 | リポーターの if < > then () else () | 36 |
| 5.2 | when () | 36 |
| 5.3 | stop ボタンがクリックされた時の終了処理 | 37 |
| 5.4 | run ブロック | 38 |
| 5.5 | call ブロック | 39 |
| 5.6 | launch ブロック | 41 |
| 5.7 | broadcast ブロック, tell to ブロック | 44 |
| 5.8 | pipe | 45 |
| 6 | ブロックを作成する | 46 |
| 6.1 | () >= () ブロック | 46 |
| 6.2 | help 説明文の作成 | 49 |
| 6.3 | for (i) = (start) to (end) step (step) | 50 |
| 6.4 | block variables | 62 |
| 7 | ブロック定義について | 63 |
| 7.1 | プルダウン入力 | 63 |
| 7.2 | Title Text とシンボル | 67 |
| 7.3 | Input name オプションについて | 69 |
| 7.3.1 | Reporter 型 | 69 |
| 7.3.2 | Predicate 型 | 72 |
| 7.3.3 | Command 型 | 73 |
| 8 | その他 | 77 |
| 8.1 | デバッグ | 77 |
| 8.2 | = と identical | 79 |
| 8.3 | 連想配列、辞書 | 81 |
| 8.4 | 入力スロットへのリスト指定 | 82 |
| 8.5 | ask | 83 |
| 8.6 | broadcast の検索オプション | 83 |

| | | |
|-----------|---|------------|
| 8.7 | クローン | 84 |
| 8.7.1 | テンポラリクローン | 84 |
| 8.7.2 | パーマネントクローン | 88 |
| 8.8 | flat line ends | 89 |
| 8.9 | 角度 | 91 |
| 8.10 | anchor アンカー | 93 |
| 8.11 | JavaScript function (オプション 5 ページ参照) | 94 |
| 8.12 | 時計 | 99 |
| 8.13 | 並列実行について | 103 |
| 8.14 | 行列の積 | 105 |
| 8.14.1 | inner product | 105 |
| 8.14.2 | outer product | 111 |
| 9 | 再帰呼び出し | 114 |
| 9.1 | 再帰呼出しの例 | 114 |
| 9.1.1 | 階乗 | 114 |
| 9.1.2 | ハノイの塔 | 114 |
| 9.2 | 再帰呼び出しの使用 | 115 |
| 9.2.1 | 繰り返し | 115 |
| 9.2.2 | my length | 116 |
| 9.2.3 | リストをリポート | 118 |
| 9.2.4 | my contains | 119 |
| 9.2.5 | unique | 119 |
| 9.2.6 | リスト要素の巡回 | 120 |
| 9.2.7 | 指定の要素に対する delete と replace | 122 |
| 9.2.8 | クイックソート (整列 / 並べ替え) | 124 |
| 9.2.9 | フィボナッチ数列 | 127 |
| 9.2.10 | 末尾再帰 | 129 |
| 10 | 高階関数 | 131 |
| 10.1 | 操作を指定するリストの巡回 | 131 |
| 10.2 | my for each (item) in (list) | 133 |
| 10.3 | foldl, foldr | 134 |
| 10.4 | 関数を返す関数 | 138 |
| 10.5 | カリー化 | 140 |
| 10.6 | Class のような使い方 | 141 |

1 始め方

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

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

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

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

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

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

オンライン版は、Run Snap! Now をクリックすれば使用できます。

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



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

2 画面まわり

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

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

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

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

英語版でも変数名やデータの内容など日本語が使えます。

2.1 エリア

各エリアには次のような名前がついています。



2.1.1 ステージエリア

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



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

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

2.1.2 スプライトコラル



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

2.1.3 スクリプトエリア

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

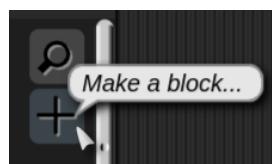
スクリプトエリアの右上には、 と があります。

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

は、キーボードを使ってスクリプトを作成するモードへのスイッチです。

2.1.4 パレットエリア

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



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

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

2.2 エリアの大きさ



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



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

2.3 実行に関するボタン

ステージエリアの上には のボタンがあります。これは に接続された
スクリプトを実行するボタンです。 はスクリプトの実行を終了させるボタンです。 はスクリプトの実行を一時停止させるボタンです。 のボタンをクリックすると実行中のブ
ロックをハイライトさせてゆっくり実行させたりできます。 のマウスポインター
が置かれているところをドラッグすると実行のスピードが調整できます。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。(77 ページ参照) デバッグモードでは、実行
中のブロックをハイライトしたり、ブロック中の変数やリストの値を表示してくれます。 のクリックで実行再開です。スクリプトを止めて確認したいところに を入れて
も、そこで一時停止させることができます。

スクリプトエリアの上に次のようなボタンがあります。



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



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



をクリックすると右のメニューが出てきます。

Notes... にはプロジェクトの覚書、注釈が書けます。

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

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

switch to scene [next v] and send [flag v] で次のプロジェクトに移行することができます。

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



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

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

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

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

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

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



2.4 ブロック表示

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

このような長い式を Snap! でスクリプトにすると、

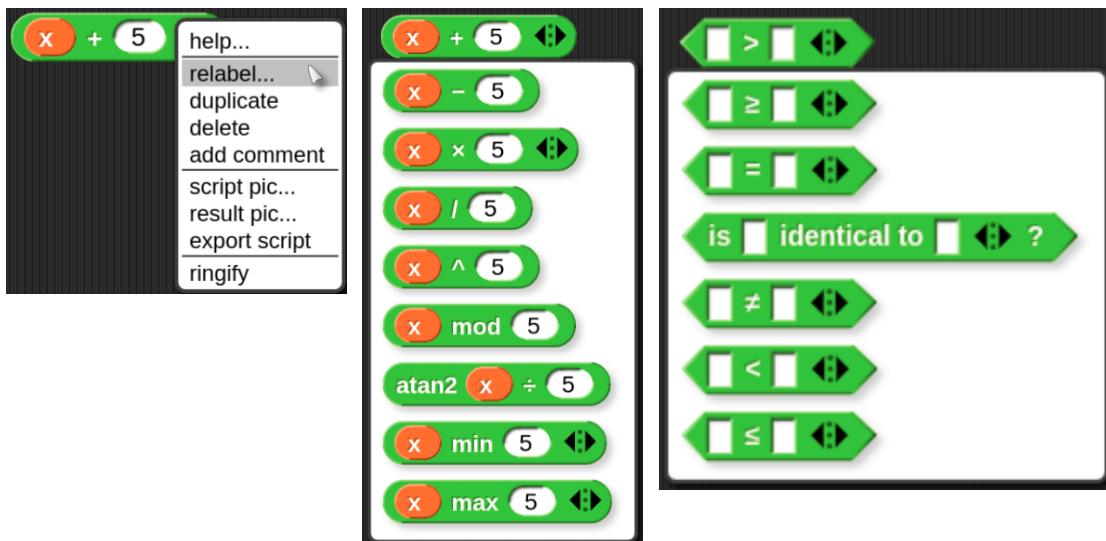


となります。長くなった場合は、自動的に折り畳まれます。また、同じパレットのブロックが重なった場合は、色合いが交互に変化して表示されます。ゼブラカラーリングだそうです。



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

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



3 キー ボード 入力

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

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

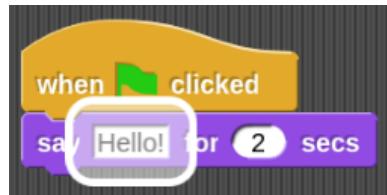
二つの計算結果を表示するスクリプトを作ってみます。



スクリプトが何もない状態から始めます。

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

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



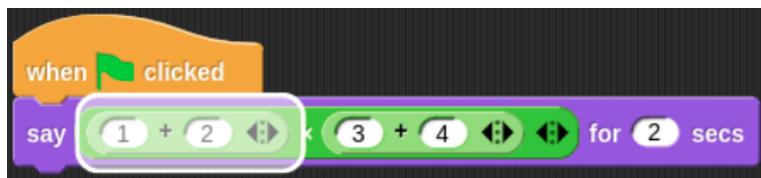
ここで  を押すと、ハロが消えて初期値としての「Hello!」を修正または別なテキストを入力することができます。 を押してハロが出ている状態で文字を打ち込むと、ブロックや変数を候補として出してくれます。数字や「() + - * / < = >」を打ち込むと、数式の入力ができます。

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

16/4/2



16/(4/2)



  で入力スロットを移動して変更できます。 で次のブロックに移ります。

次に、ルート 3 の値を表示させてみます。ルートは sqrt で求めます。

また、say Hello! for 2 secs を出してください。

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



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

4 変数

変数を作成するにはいくつか方法があります。

- Make variables をクリックする。
- script variables □ を利用する。
- block variables を利用する。(ブロック内変数 62 ページ参照)
- for ブロックなどの変数を利用する。
- .txt .csv .json などのファイルを Snap! 画面にドロップする。

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



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

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



[Make a variable] をクリックすると、



が出ますから for all sprites を選んで、varAll



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

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



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

名前を変更する場合はその変数のところを右クリックして、



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



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

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

 をスクリプトエリアに置きます。これを実行させない状態で save します。すると、ファイルの容量は 8.1KB でした。これを実行すると、変数 var に Snap! のホームページの html ファイルが入ります。(オンラインの Snap! を使用のこと)

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

```

この状態で save します。すると、ファイルの容量は 1.6MB でした。この保存したプロジェクトを改めて Open... で読み込むと、保存された時の変数の値になっています。つまり、何もしないと変数が値ごと保存されてプロジェクトの容量を大きくするということです。transient をチェックして save すると、ファイルの容量は 13.2KB でした。この数値は実行環境など状況によって違うかもしれません。この保存したプロジェクトを改めて Open... で読み込むと、変数 var の値は初期値 0 になっています。これが transient の機能です。

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

4.2 for this sprite only スプライト変数



(Make a variable) をクリックして、



for this sprite only を選んで varSprite を作



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

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

set [varAll v] to [0] を使いますが、varSprite は変数名リストに出てきません。

4.3 script variables スクリプト変数

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



a の隣りにある右向きの三角をクリックすると変数を追加できます。



左向きの三角をクリックすると削除できます。

スクリプト変数は をスクリプトエリアに持ってくるだけでは使えません。



のように接続されてからでないと使えません。



のようにその前でも有効になりません。

4.4 for ループ変数



このようにあらかじめ用意されている変数があります。

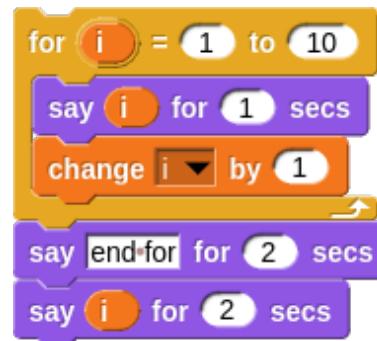
これはスクリプト変数のように変数名を変更することができます。また、この変数をいくつでもドラッグ&ドロップして使用することができます。この変数は、ループする間に 1 ずつ増加または減少していくので、その変化していく変数の値をスクリプトで使用するということですが。



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



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



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

4.5 リスト操作用ブロック

Snap! には Scratch のようなリスト専用の変数はありません。変数に数値や文字列を入れるよう 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

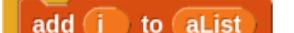


れます)、`aList` は要素を持つリストになり、ステージエリアに

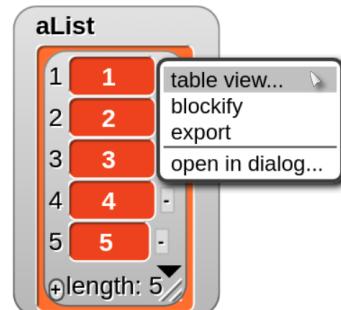


 を使うと、





で同じことができます。



変数ウォッチャーのリスト表示エリア内を右クリックして

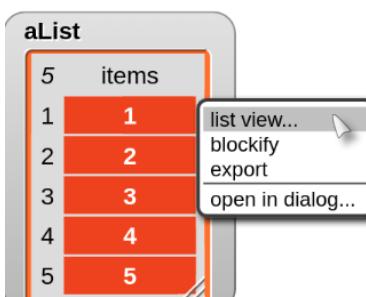


table view... を選択すると

に変更することができます。また右

クリックして list view... を選択すると元に戻ります。



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



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



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

export をクリックすると、変数の内容がテキスト表示できてリスト以外ならば .txt 形式で、リストの場合は .csv 形式、複雑なリストの場合は .json 形式のファイルとして書き出されます。

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



たとえば、**aList** や **list 1 2 3 ↵**



のように。

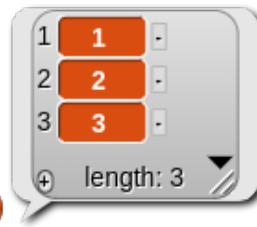
なお list view では、一度に扱える要素数は 100 までになっています。次の 101 から 200 の要素



に移るには、下向きの三角をクリックすると出てくる範囲から選びます。

4.5.1 numbers form () to ()

これは指定された範囲のリストをリポートするものです。



指定された範囲内で 1 ずつ増加するリストをリポート



指定された範囲内で 1 ずつ減少するリストをリポート



1 ずつの増加または減少なので指定された値が含まれないこともあります。



4.5.2 () in front of ()



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

4.5.3 all but first of ()



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

4.5.4 index of () in ()

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

index of 5 in numbers from 1 to 10 5

index of 12 in numbers from 1 to 10 0

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

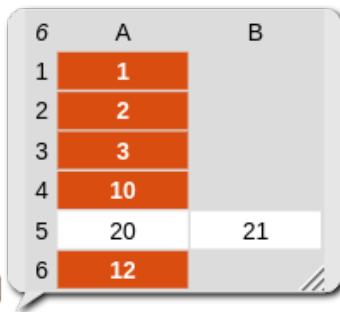
4.5.5 append () ()

これは、指定されたリストを結合したリストをリポートします。

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



append list 1 2 3 list 10 list 20 21 12 13 length: 6



4.5.6 for each () in ()

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

for each item in aList
say item for 1 secs

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

4.5.7 reshape() to()()

リストの要素にリストを含むものを配列、縦横二次元的なものを行列と言います。

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

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

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

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

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

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

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

reshape [list 0 ← to 4 3] ←

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

+ length: 5

reshape [list 0 ← to 5] ←

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

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

replace item 3 of item 1 of aList with 99

4.5.8 map () over ()



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



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

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

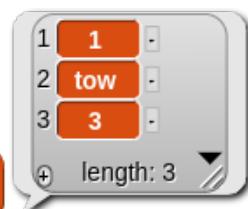
value には指定されたリストの要素が順番にセットされます。**index** はリストの何番目の要素を処理しているかを示します。**list** は指定されたリストを表します。この場合 **list** は **aList** を表します。**value = item (index) of (list)** の関係になっています。



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



replace item 1 of list with thing のブロックを使うとリスト内の位置を指定して要素を入れ替えることができます。次のようにすると特定の値の要素を入れ替えることができます。



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



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

まずは簡単にするために **aList** の各要素を 1 にします。



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

次のようにして list の要素に対して値を入れ替えていきます。そうしないと、 $(n - 1)$ 項と $(n - 2)$ 項の値を得られないからです。

```

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

```

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

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

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

```

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

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

```

4.5.9 keep items () from ()



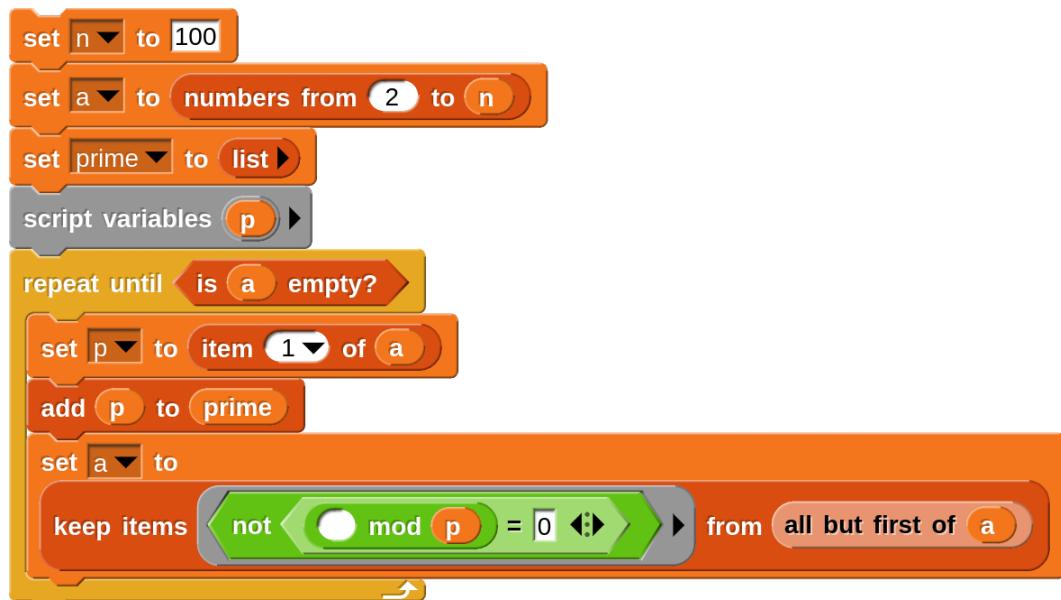
これは、指定されたリストの要素から指定された条件に合った値のリストをリポートします。指定されたリスト自体は変更しません。

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



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

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



2000 年から 2200 年までのうるう年のリストを求めてみます。

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



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

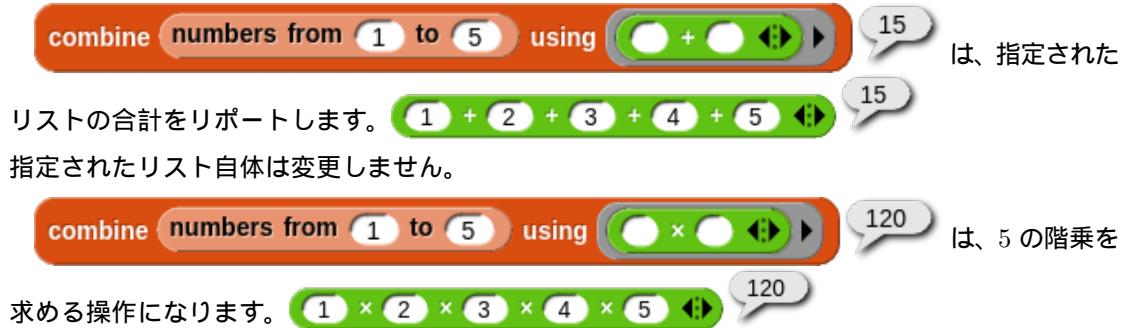
4.5.10 find first item () in ()



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

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

4.5.11 combine () using ()



combine では、以下の演算がおもに使われるようです。



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

join を指定すると文字列の連結になります。(入力スロットは空です。)



これと逆の操作をするのが split ブロックです。



リスト要素の最大値や最小値を求めます。

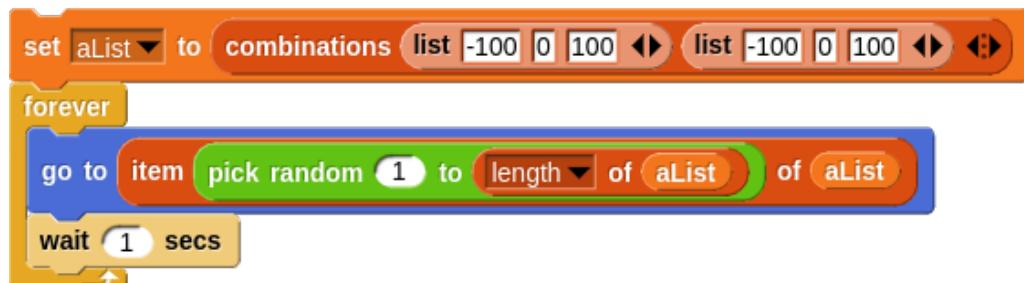


4.5.12 combinations () ()

指定されたリスト同士を順番に組み合わせたリストをリポートします。



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



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

4.6 オプションのリスト操作

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

4.6.1 rank of (), dimensions of ()

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

rank ランク (階) は配列の次元を、dimensions は形 (shape) をリポートします。APL 言語では、データはすべて配列です。

a のようにリストではない場合です。

The screenshot shows the 'rank' and 'dimensions' blocks for a list. The 'rank' block outputs '0' and the 'dimensions' block outputs 'length: 0'. Below it, a list block contains three elements: 1, 2, and 3, with a total length of 3.

要素にリストを含まないリストの場合です。

The screenshot shows the 'rank' and 'dimensions' blocks for a list containing a list. The 'rank' block outputs '1' and the 'dimensions' block outputs 'length: 1'. Below it, a list block contains a single element, which is itself a list with two elements: 1 and 3.

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

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

行列 (マトリックス) の場合です。

reshape で指定した形が dimensions の値になります。

The screenshot shows the 'rank' and 'dimensions' blocks for a reshaped matrix. The 'rank' block outputs '2' and the 'dimensions' block outputs 'length: 2'. Below it, a list block contains two elements: 1 and 2, with a total length of 2.

rank は dimensions のリストの要素数を示します。

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

| a | | |
|-----------|---|---|
| 2 | A | B |
| 1 | 1 | 1 |
| 2 | 1 | 1 |
| length: 2 | | |

rank of a 3

dimensions of a

| | |
|-----------|---|
| 1 | 2 |
| 2 | 2 |
| 3 | 2 |
| length: 3 | |

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

4.6.2 flatten of ()

rank 2 以上のリストを rank 1 に整形してリポートします。

flatten of list 1 list 2 list 3 4

| | |
|-----------|---|
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 4 |
| length: 4 | |

4.6.3 columns of () 転置行列

リストの行と列を入れ替えた転置行列をリポートします。

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

aList

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

columns ▾ of aList

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

4.6.4 uniqueness of ()

リスト中の同じ要素を削除してリポートします。

頻度順になるようです。(119 ページで別バージョンを作成)

uniques ▾ of list 1 2 2 3 3 3 ↵

| | | |
|-------------|---|---|
| 1 | 3 | - |
| 2 | 2 | - |
| 3 | 1 | - |
| + length: 3 | | ▼ |

4.6.5 distribution of ()

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

distribution ▾ of list a b b c c c ↵

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

4.6.6 sorted of ()

リストの要素を整列してリポートします。

sorted ▾ of list 2 5 1 e a c ↵

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

4.6.7 shuffled of ()

リストの要素をシャフルしてリポートします。

| | |
|---|---|
| 1 | 3 |
| 2 | 5 |
| 3 | 1 |
| 4 | 4 |
| 5 | 2 |

(+) length: 5

shuffled ▾ of numbers from 1 to 5

4.6.8 reverse of ()

リストの要素を逆順にしてリポートします。

| | |
|---|---|
| 1 | 5 |
| 2 | 4 |
| 3 | 3 |
| 4 | 2 |
| 5 | 1 |

(+) length: 5

reverse ▾ of numbers from 1 to 5

4.6.9 Σ of ()

リストの要素を合計してリポートします。

Σ ▾ of numbers from 1 to 10 55

要素がリストでも中の値すべて期待したように合計してくれます。

Σ ▾ of list [5 5] list [1 2 3] 16

| | |
|---|----|
| 1 | 11 |
| 2 | 12 |
| 3 | 13 |

(+) length: 3

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

4.6.10 text of ()

リストの要素を文字列としてリポートします。

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

combine numbers from 1 to 5 using join [] 1 2 3 4 5

4.6.11 lines of ()

リストの要素を文字列行としてリポートします。

これが使えるのは rank 1 のリストです。



4.6.12 csv of ()

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

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



4.6.13 json of ()

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

rank 3 のリストにはこちらを使うことになります。



4.7 リストの演算

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

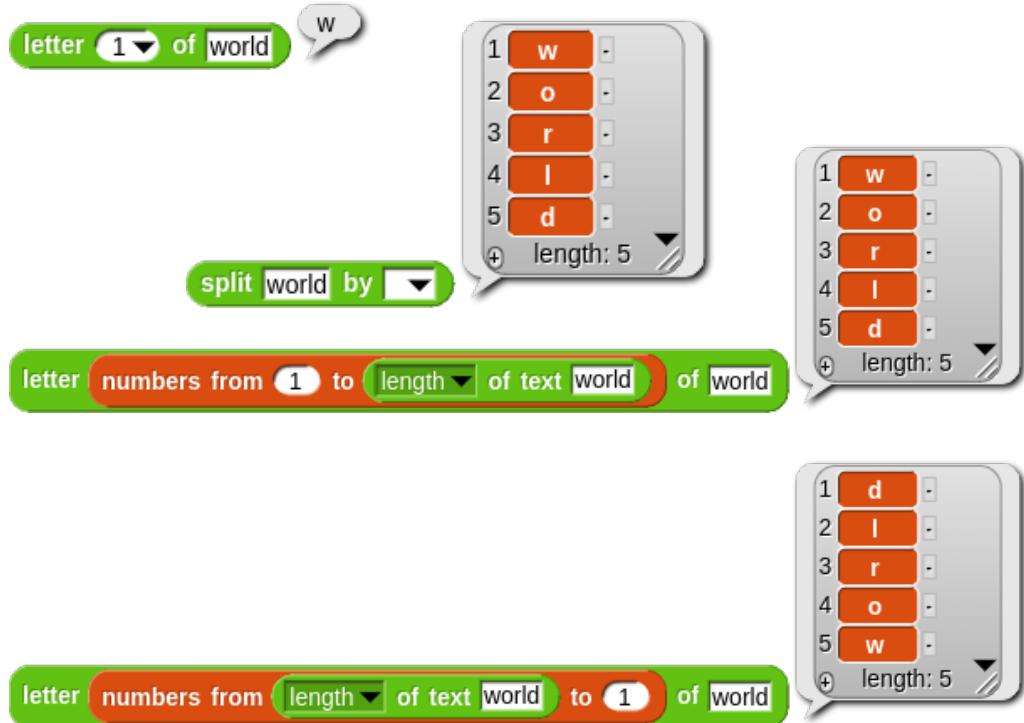




リスト同士で演算をすることもできます。いずれも同じインデックスの要素同士を演算します。
要素数が合わない場合は対応する部分だけで行います。

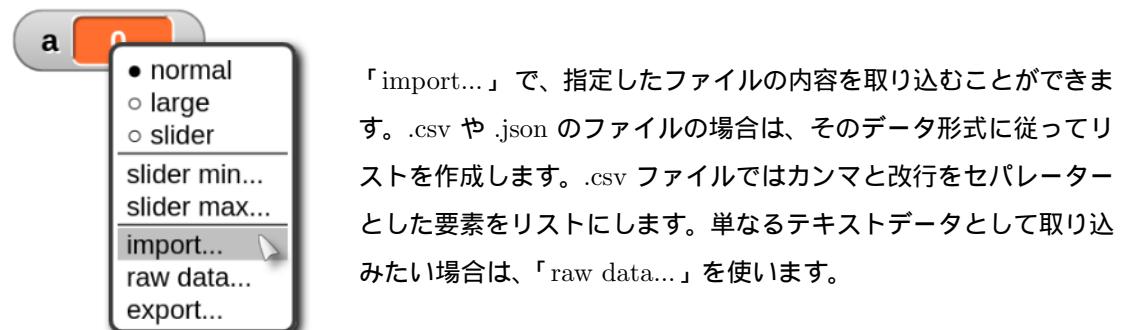


リストに文字列の各要素を入れることもできます。



4.8 変数に入れられるもの

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

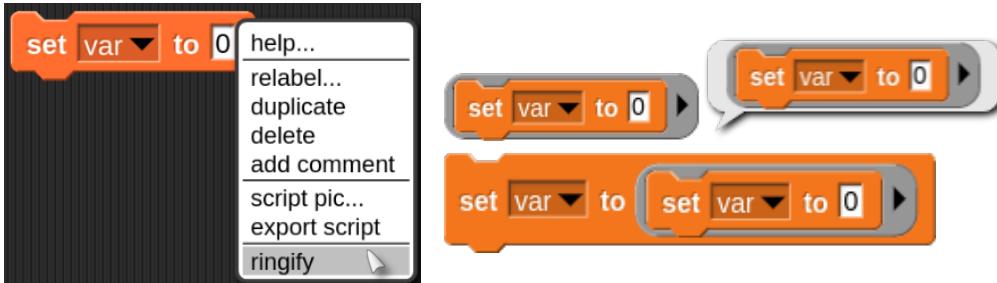


Snap! の変数には、値をリポートするものを入れられます。

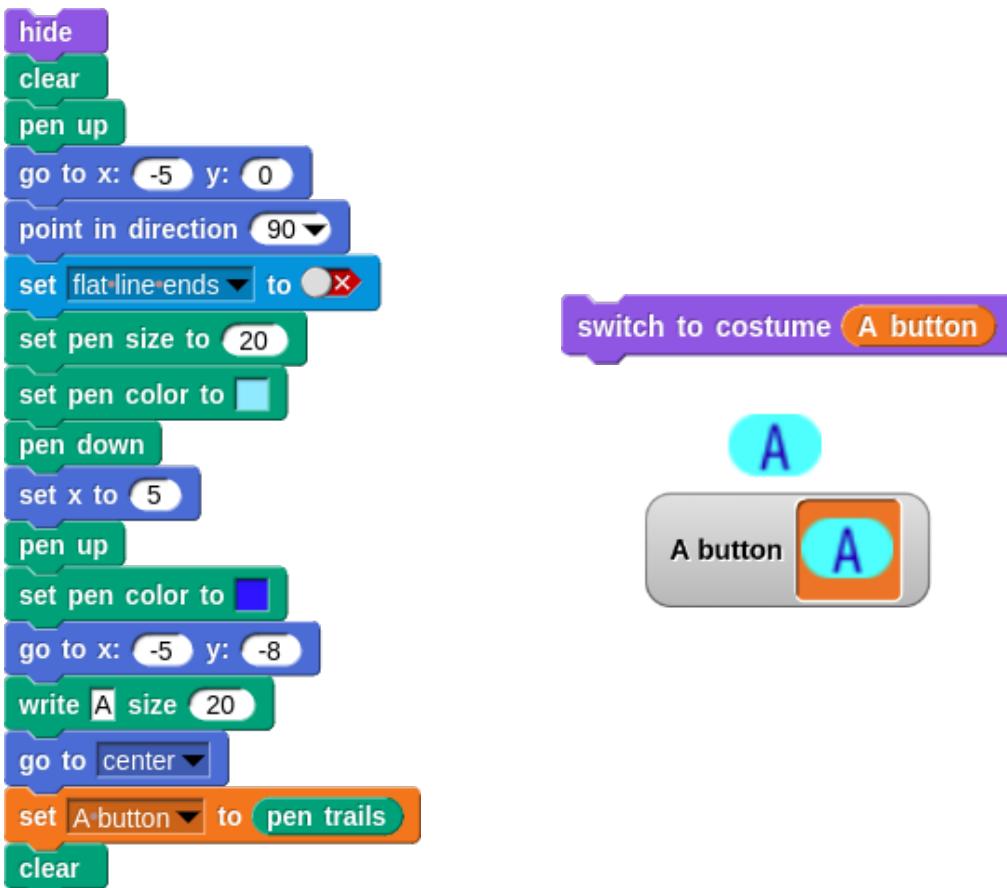




ブロックをリングに入れてやるとリポーターブロックにすることができます。ブロックそのものをリポートするものです。対象のブロックのところで右クリックしてメニューから ringify を選べばリングで囲むことができます。リングを外すには unringify をクリックします。



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



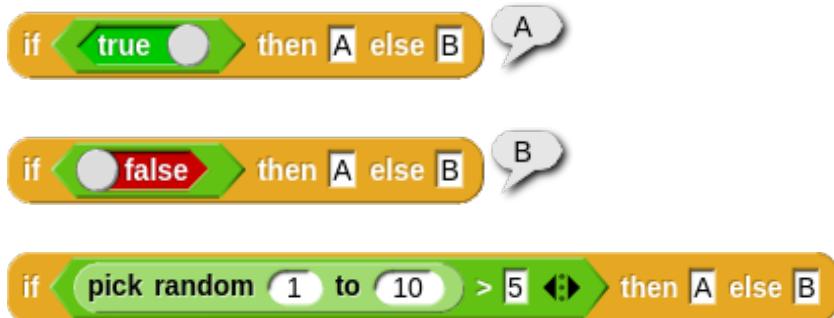
5 Control 制御

Snap! で使用できる制御ブロックについて。

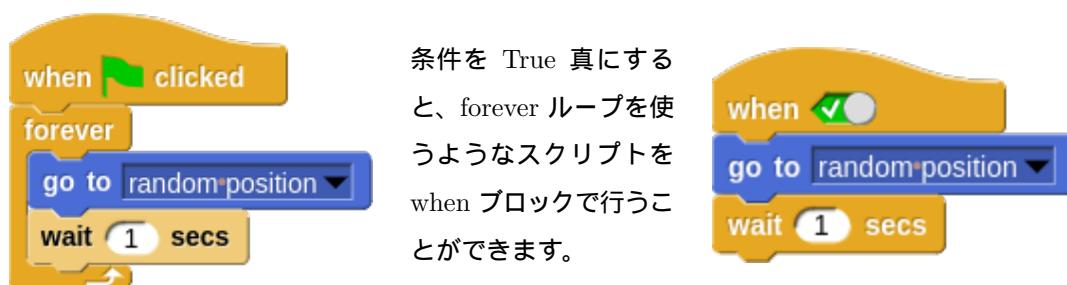
5.1 リポーターの if < > then () else ()

if < > then () else () 制御ブロックは true か false かによりどちらかのスクリプトを実行しますが、

リポーターブロックは true か false によりどちらかの値をリポートします。

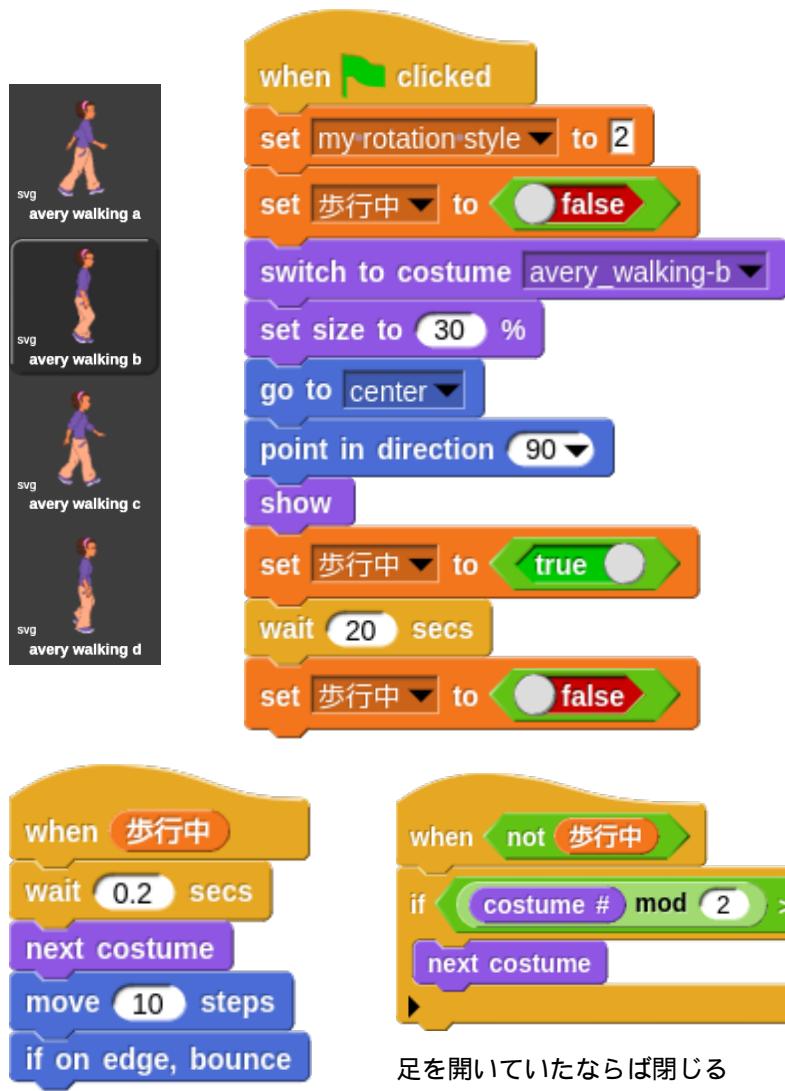


5.2 when ()

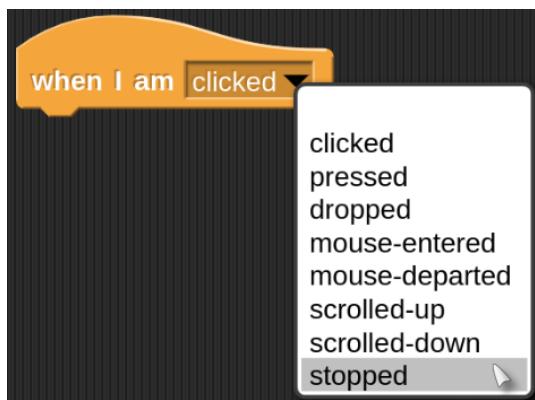


when ブロック版の場合は、 をクリックすると のように形が四角になります、それをクリックして再開することができます。

when ブロックを使って、人が歩く動作をしてみます。変数 歩行中 を使います。



5.3 stop ボタンがクリックされた時の終了処理



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

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

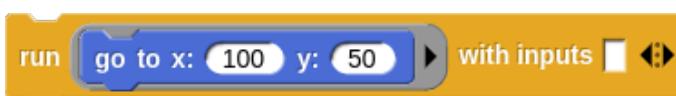
次のスクリプトで動作の確認ができます。変数 n を作成して下さい。けっこうな回数 n のカウントができるようです。



5.4 run ブロック

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

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



のようになります。go to x: y: の各入力スロットを空にして、



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



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



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

5.5 call ブロック

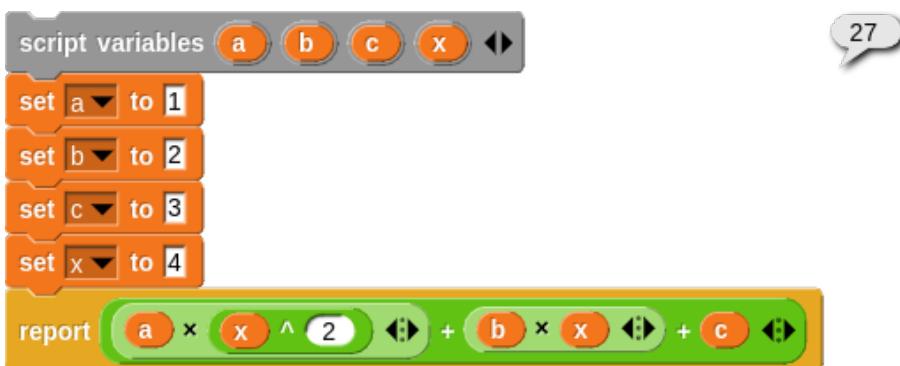
run ブロックに入れられるのが値をリポートしない実行 (Command) ブロックだったのに対して、call ブロックに入れられるのは、実行または評価することによって何らかの値または true か false をリポートするブロックです。call ブロックは、得られた値をリポートします。



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

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

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



変数の入力スロットを空にして、式をリングで囲みます。



リングの端の三角をクリックしてフォーマルパラメータを出します。



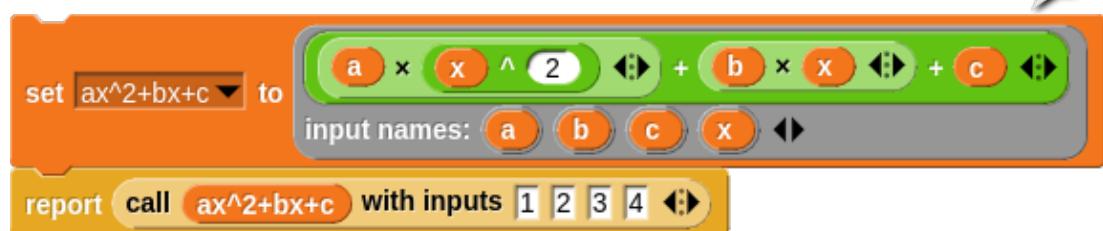
フォーマルパラメータは、クリックして変数名を a、b、c、x に変更します。

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



このリングを変数に入れてやれば定義ブロックのようにも使えます。

27

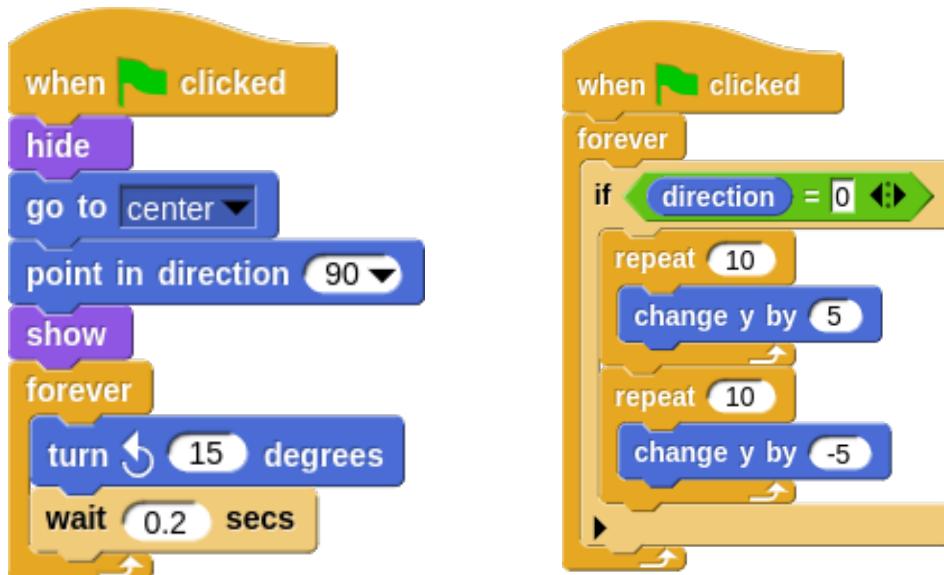


5.6 launch ブロック

launch ブロックは指定されたスクリプトを並列で実行するものです。

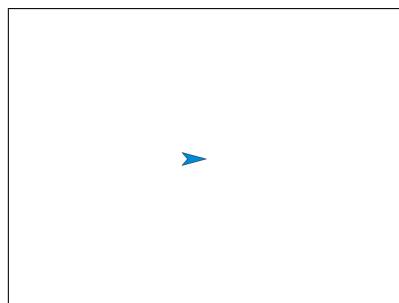
1 個のスプライトに対して、回転させるスクリプトと、角度が上（0 度）の時にジャンプさせるスクリプトを並列で実行してみます。

並列実行は、普通次のようにして 2 つのスクリプトを同時に実行します。



左のスクリプトで、ずっと回転させる動作を行います。

右のスクリプトで、上を向いた時だけジャンプする動作を行います。



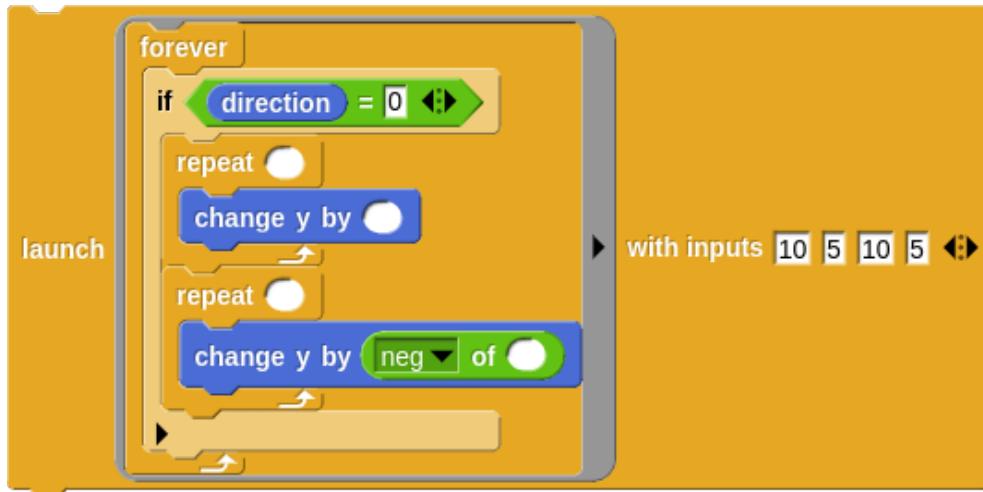
launch を使うことで 1 つのスクリプトで実行することができます。



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



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



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



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

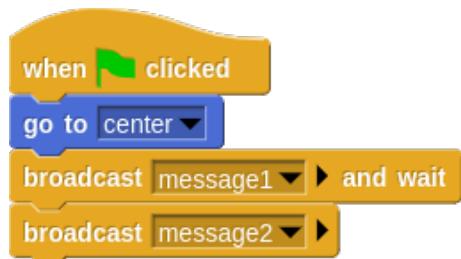
5.7 broadcast ブロック, tell to ブロック



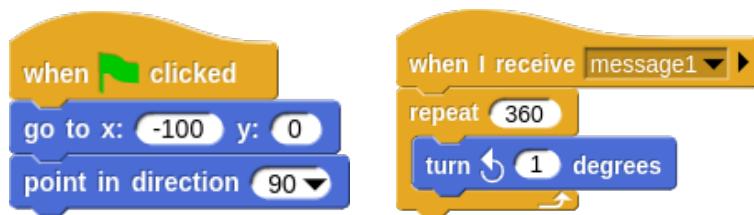
次はスプライトを3個使います。 Sprite, Sprite(2), Sprite(3) を用意してください。

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

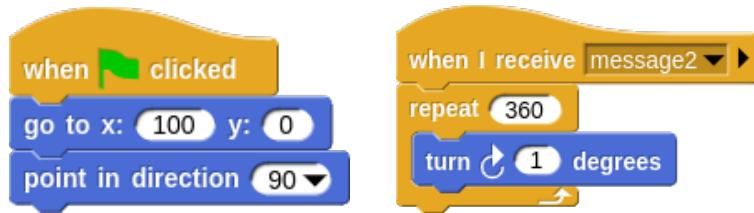
Sprite 用



Sprite(1) 用

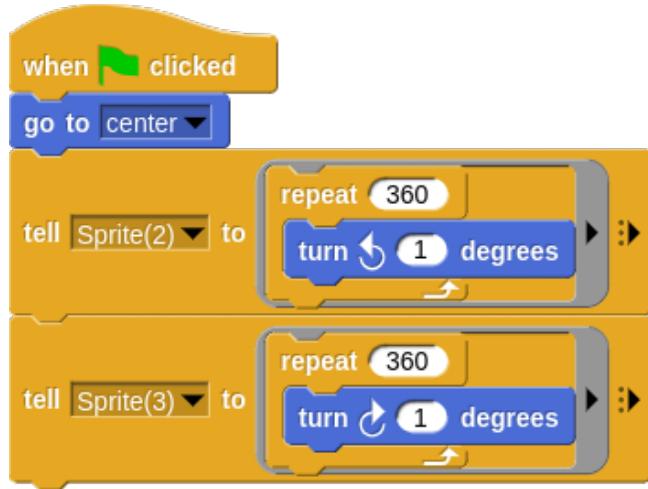


Sprite(2) 用



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

Sprite 用



5.8 pipe

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

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

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

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

pipe SnapText

pipe SnapText

pipe SnapText
length of list

67

受け取ったリポートは指定
のスロットに入って処理さ
れます。

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）が作りやすくなっています。

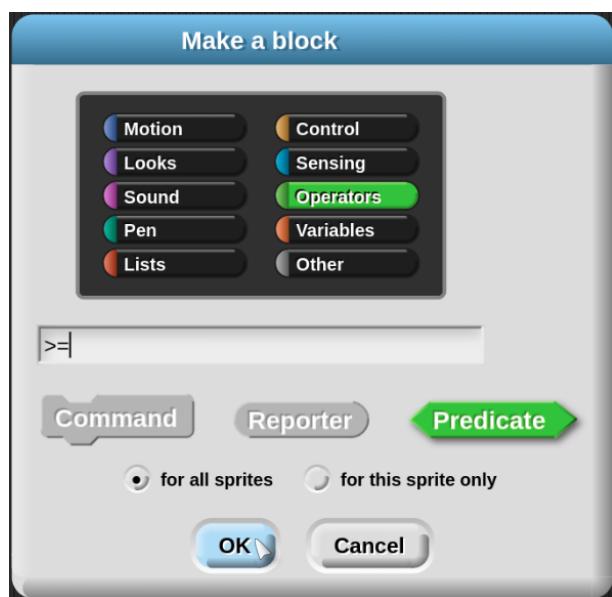
6.1 () \geq () ブロック

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

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かまいません。設定用のウィンドウが現れます。選択できるカテゴリーには Other がありますが、 をクリックしてメニューから New category... を選択すると、独自のカテゴリーを作成してパレットに追加することができます。パレットの色も指定できます。

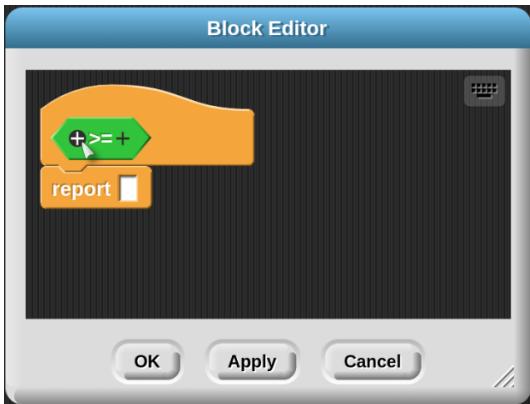


パレットのカテゴリーを選択するボタンや、新しいブロックの名前を入れる欄、ブロックの機能の種類を選択するボタン、このブロックをこのスプライトだけの機能にするかのボタンがあります。

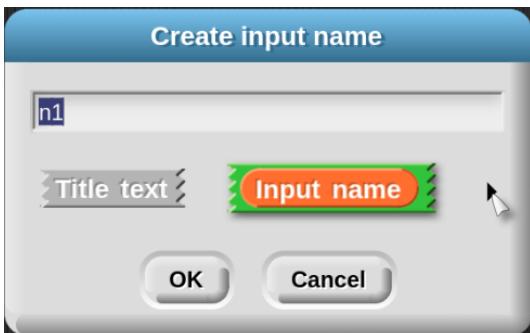
カテゴリーで Other 「その他」というものを選ぶと、置き場所は Variables のところになります。

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

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

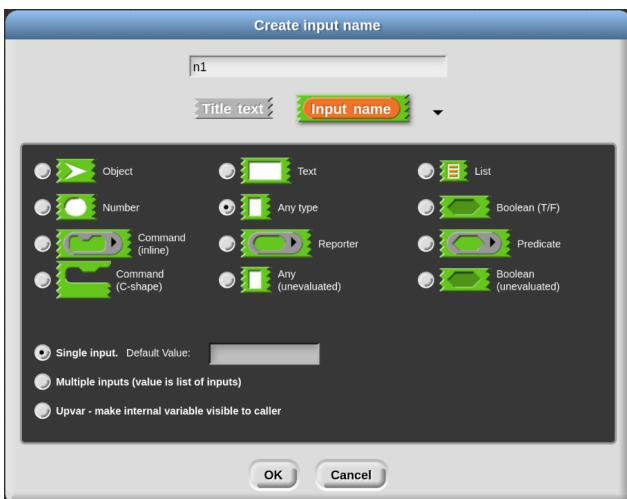


定義の先頭の の部分をプロトタイプといいます。ここを左クリックするとブロックのカテゴリーを変更することができます。右クリックして block variables を選択するとブロック内変数が使用可能になります。「>=」の左側の + ボタンをクリックしてください。



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、



Any type が選択されています。現在は Number にしてみます。これは数値しか受け付けないタイプです。Any type を選んだ場合は、変数の表記に「#」が付かないだけで以下の操作は同じです。

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



右側の + ボタンをクリックして、同じように n2 の設定をしてください。



パレットエリアからブロックを持ってきて完成させてください。

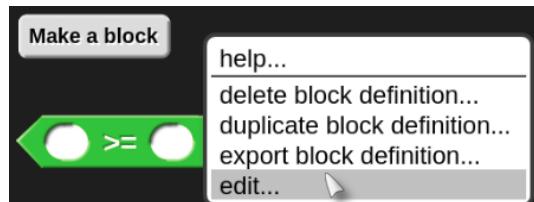


report は値をリポートする(返す)ためのブロックです。この場合は式の結果により真理値、true か false をリポートすることになります。apply をクリックしてください。



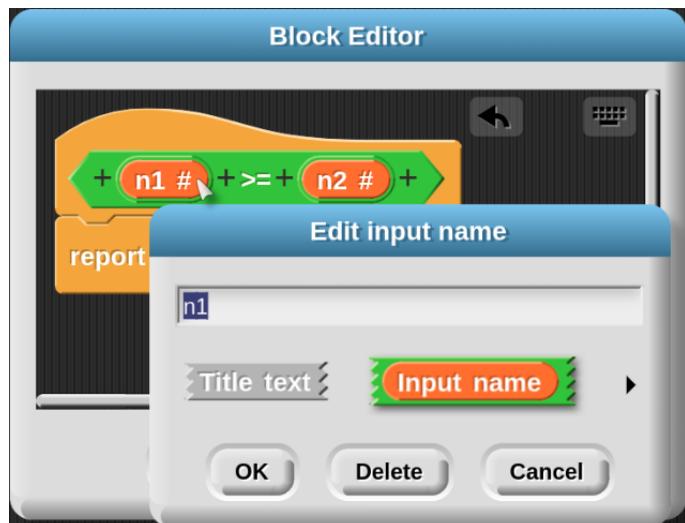


正しい値をリポートしないならばスクリプトを見直してください。正常ならば OK をクリックして終了です。



プロックを右クリックすると、edit... で内容の編集ができます。

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



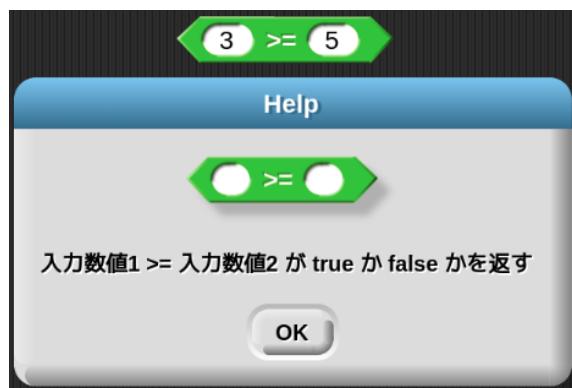
から変更ができます。変数を Any type に設定し直すこともできます。

! を使ってステップ実行する場合に、作成したプロックの内部をステップ実行させなければ **!** オンにしてからそのプロックをブロックエディターで表示させておく必要があります。順序が逆だとそのプロック内のステップ実行はされません。(77 ページ参照)

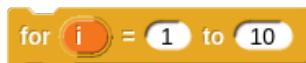
6.2 help 説明文の作成

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





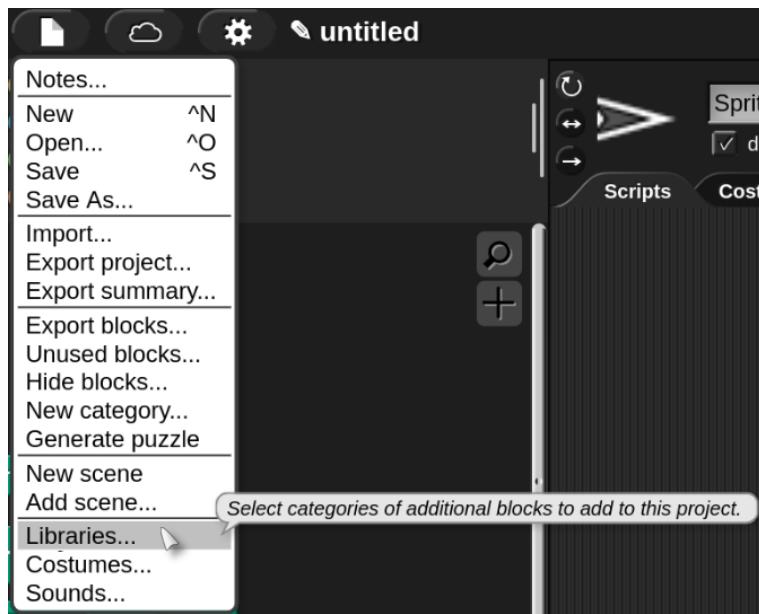
6.3 for (i) = (start) to (end) step (step)



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



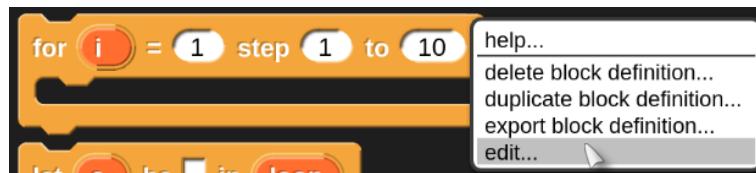
をクリックします。



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



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



を右クリックして、edit...
で中身を見てみます。



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

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

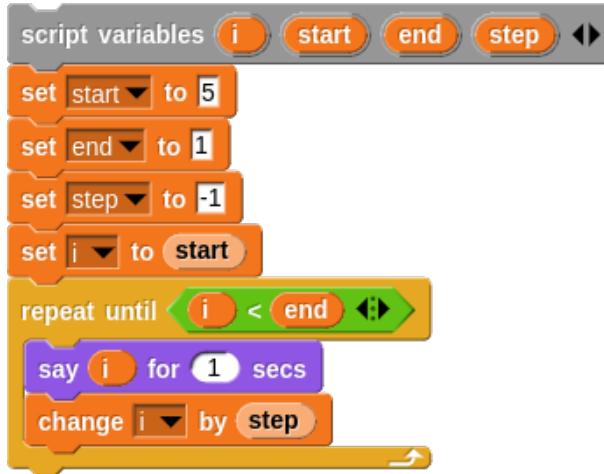
`i` が増加していく場合です。



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



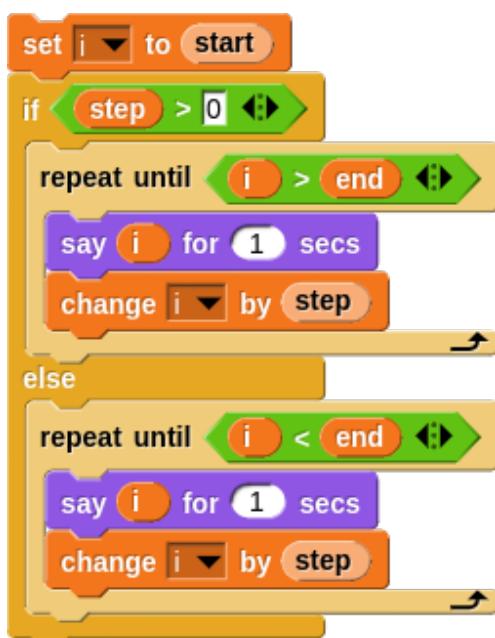
`start` を 5、`end` を 1、`step` を -1 にすると、 $5 > 1$ で、`[i > end]` の終了条件を満たしてしまうので実行されません。減少カウントにする場合は、終了条件を `[i < end]` にして



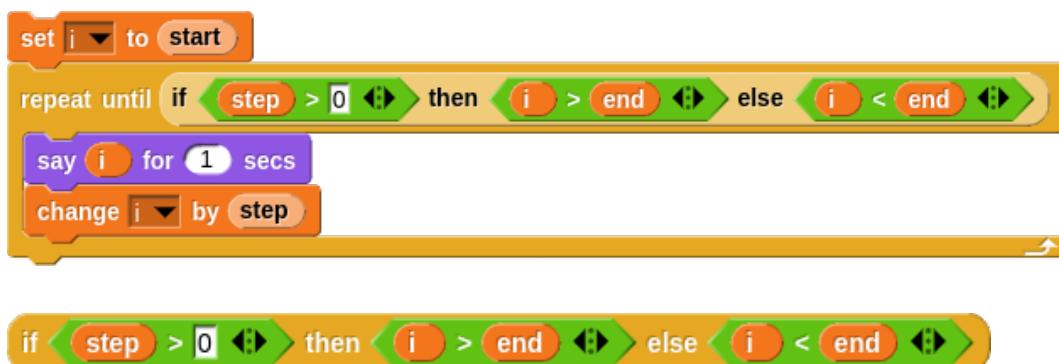
のようにしなければなりません。

増加でも減少でも対応させると、for ループは次のようにになります。

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

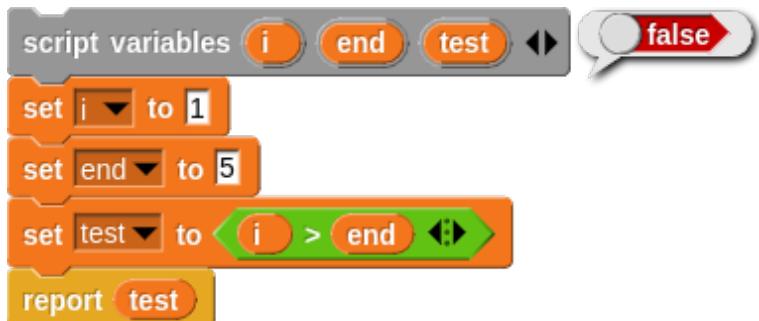


ループのテスト部分をまとめると次のようにすることができます。

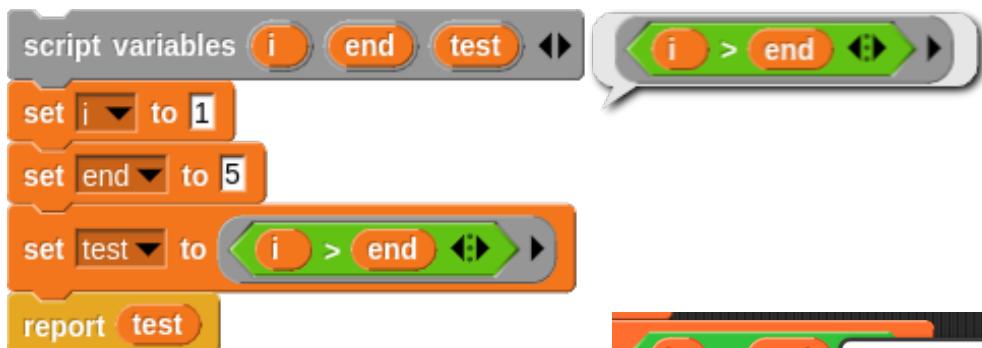


の部分で、`step` の値によって `[i > end]` か `[i < end]` がテストされる

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



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



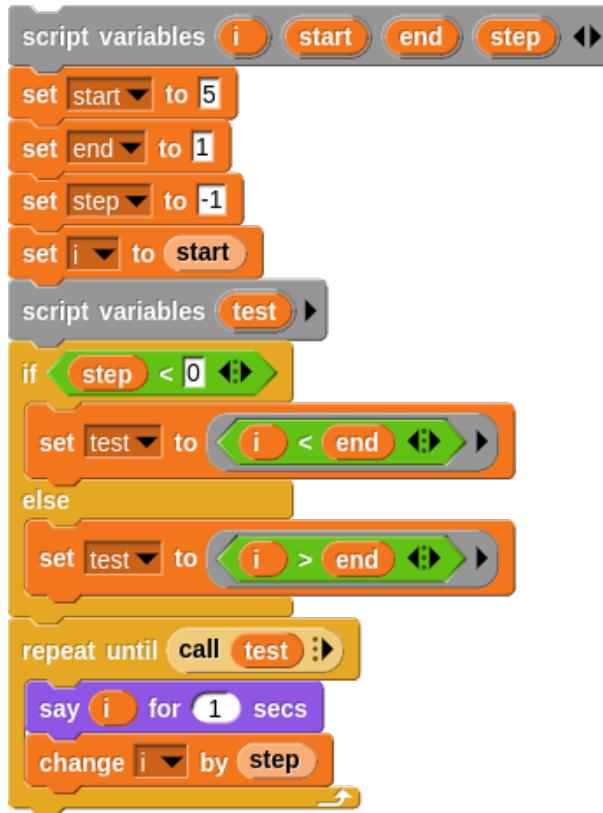
テスト項目自体を変数に入れます。

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



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

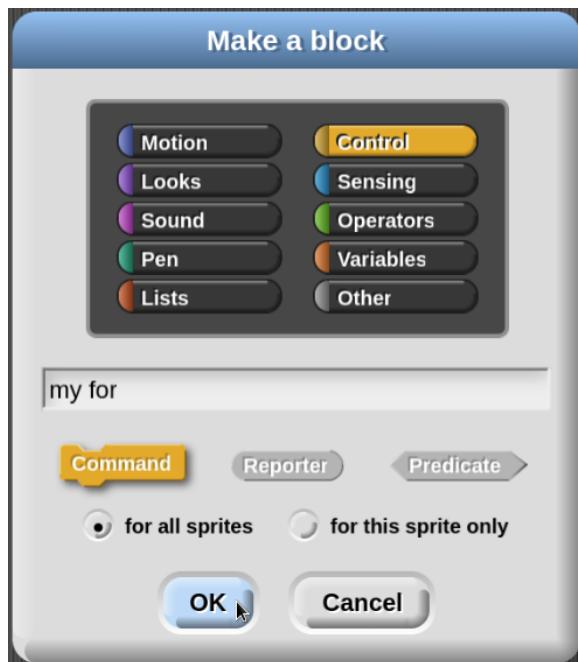
これを for ループに使用します。



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

それでは、いよいよブロックエディターで作成していきます。



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



「+」をクリックして、変数 i の設定をします。



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



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





続いて、「+」をクリックして、変数 start を設定します。

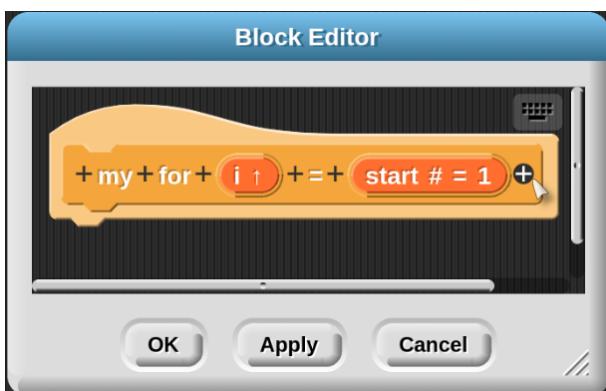


オプションとして、 Number 、数値入力限定で、



規定値を 1 に設定します。

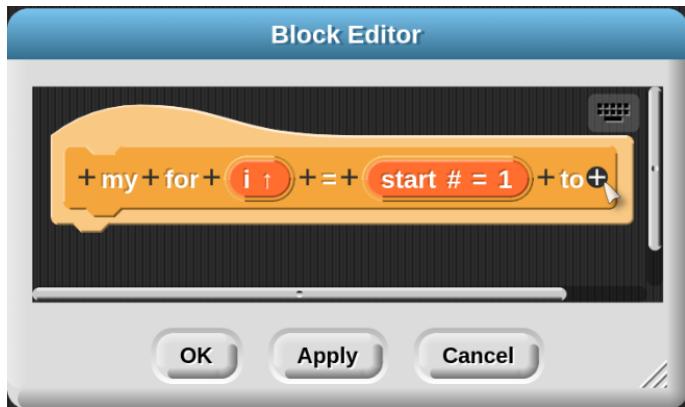
続いて、「+」をクリックして、



「to」を、Title text として入力してください。



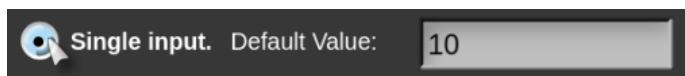
続いて、「+」をクリックして、



変数 end を設定します。

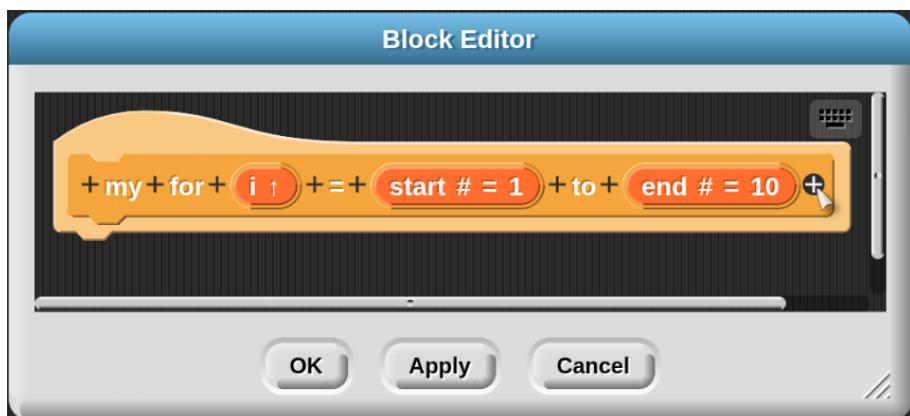


オプションとして、、数値入力限定で、



規定値を 10 に設定します。

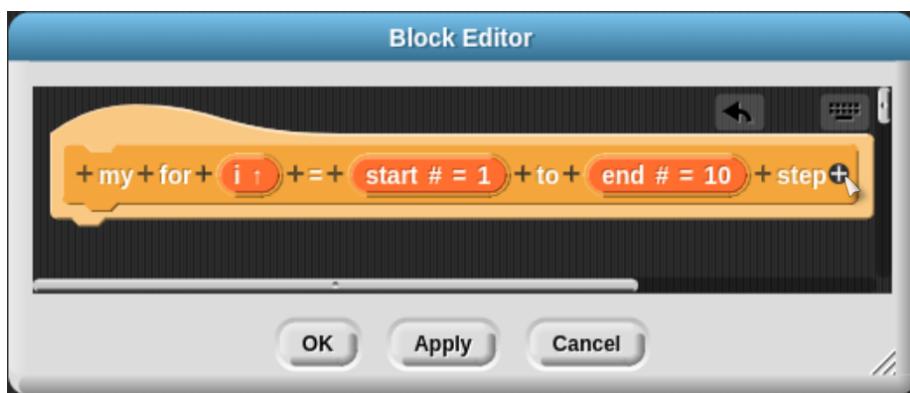
続いて、「+」をクリックして、



「step」を、Title text として入力してください。



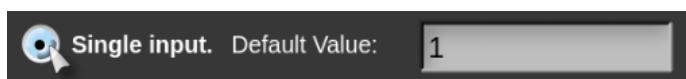
続いて、「+」をクリックして、



変数 step を設定します。



オプションとして、 Number、数値入力限定で、



規定値を 1 に設定します。

続いて、「+」をクリックして、



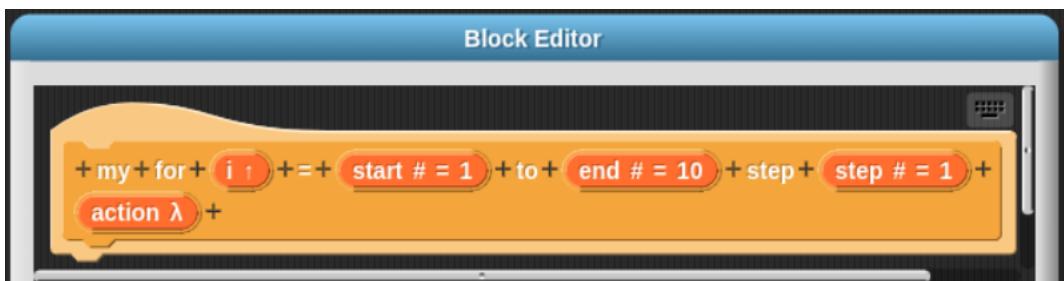
変数 action を設定します。



オプションとして、 Command (C-shape) 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。

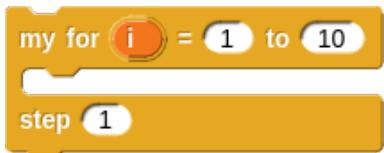
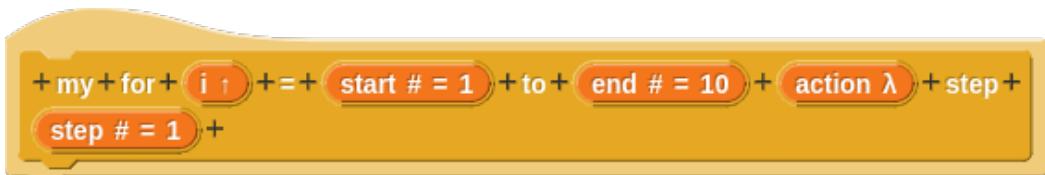
ブロック定義の本体として、実験していた for ループのスクリプトを持ってきます。action で指定されたスクリプトを実行するブロックは run です。

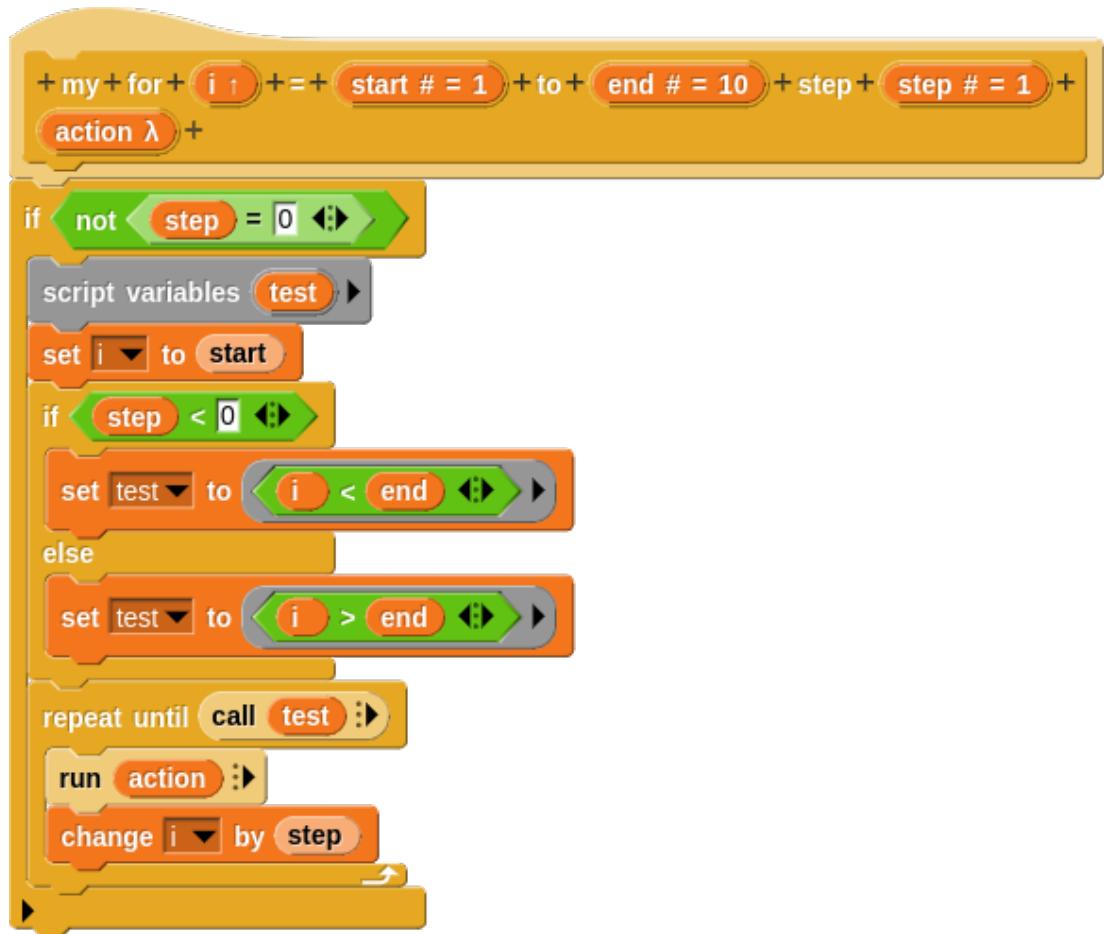
run action  で実験用のスクリプト say i for 1 secs を入れ替えます。

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。

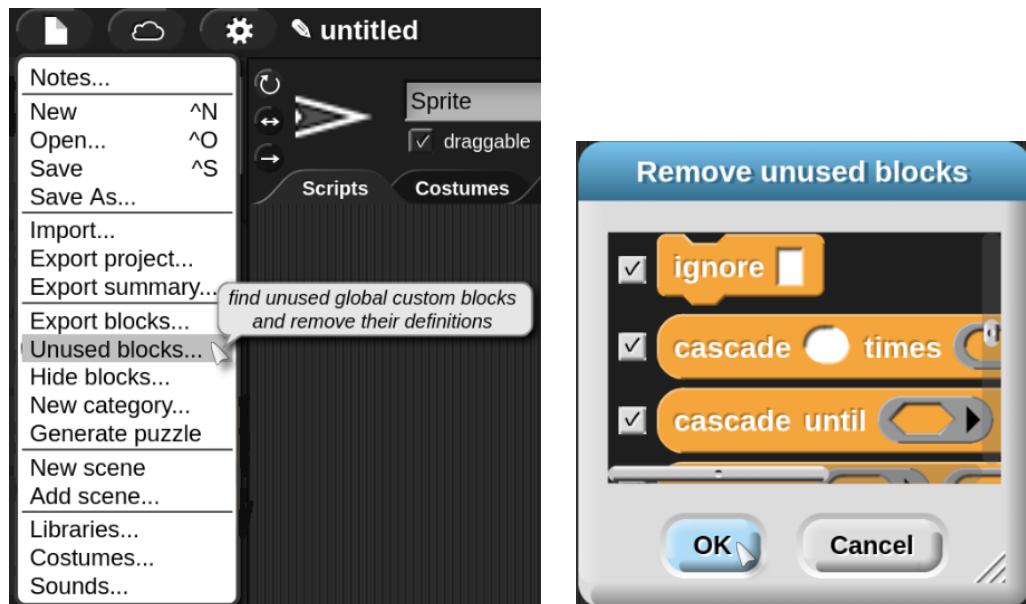


プロトタイプで step に関する部分 (step + step # = 1) を action の後ろに持ってくると次のようにすることができます。

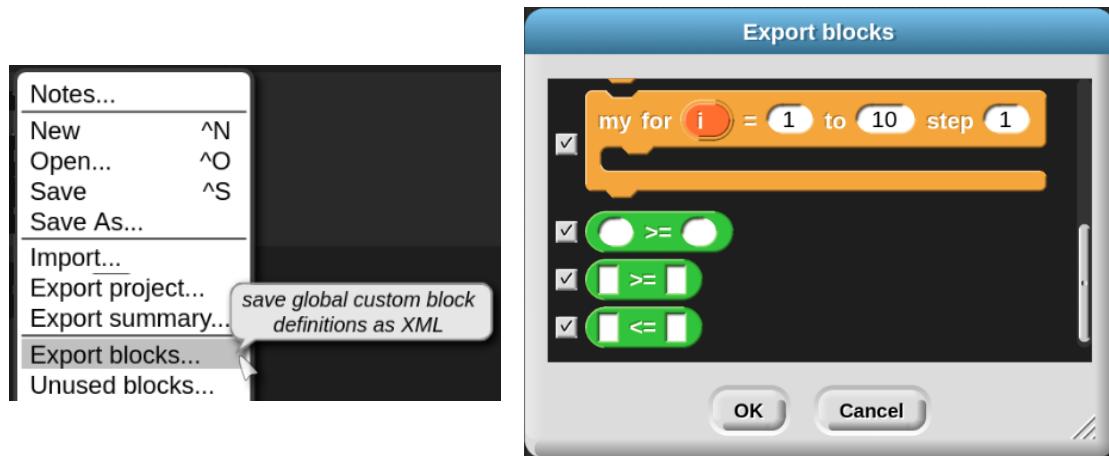




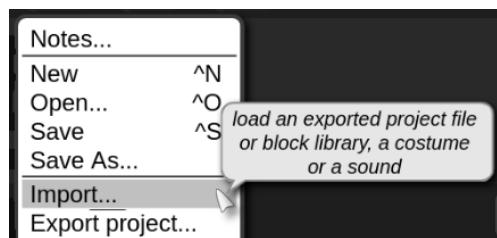
ところで、Libraries... からブロック定義をインポートしてプロジェクトを作成した場合に、普通に保存すると未使用のブロック定義も含んだファイルになります。プロジェクトを公開する場合は、次のようにして未使用のブロック定義を削除してファイルの容量を小さくしたほうがいいかもしれません。



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



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



6.4 block variables

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

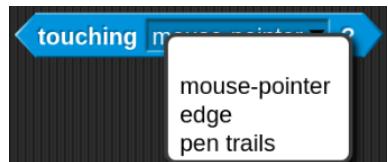
スクリプト変数は script variables ブロック以降有効になりますが、block variables はその定義ブロック内だけで有効な変数になります。(block variables を使用するとマウスのクリックに反応しなくなり、ブラウザのリセットが必要になることがあります。実行前にプログラムの保存を行い、うまく行かない時は script variables を使用して下さい。)



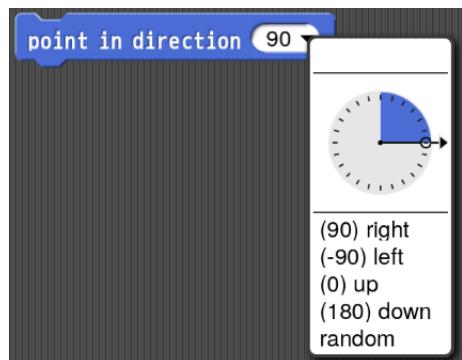
7 ブロック定義について

7.1 プルダウン入力

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



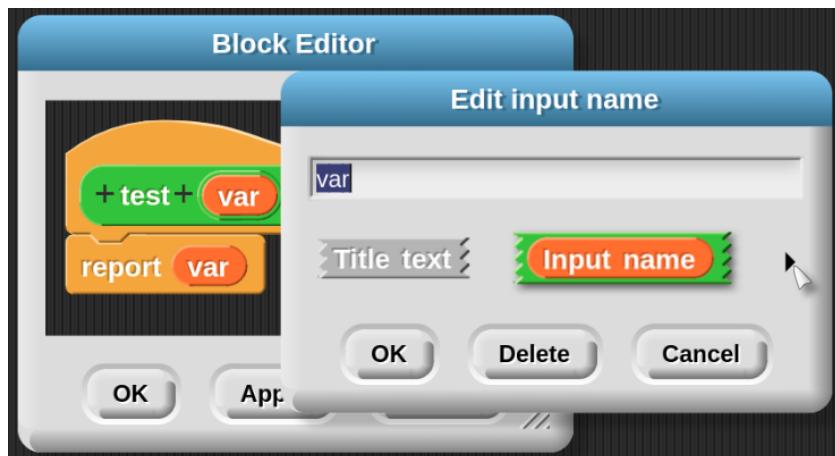
入力スロットに値をキーボードから入力することはできなくなっています。これは、後で出てくる read-only のオプションが指定されているためです。



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

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

説明のために、test というブロックを作成します。



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

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



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



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



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



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



次のように、「 ~ 」チルダだけの行はセパレーター（水平線）になります。



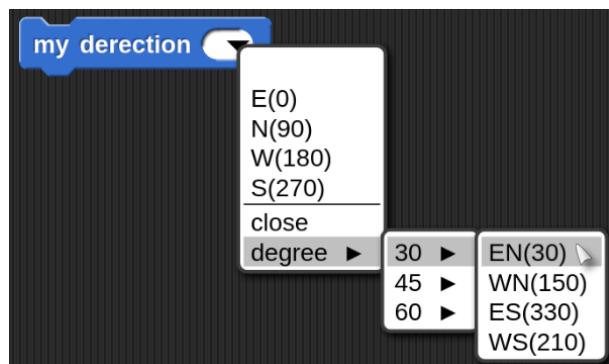
項目の前に、「 §_ 」セクション記号とアンダースコア（アンダーバー）を置くと、シフトキーを押しながらクリックしないとプルダウンメニューに表示されなくなります。因みに、§ 記号は Windows OS の場合だと、Alt キーを押しながらテンキーから（通常の数字キーではなく）0167 と入力、Linux OS の場合だと、[Ctrl]+[Shift]+[u] を押してから、a7 を入力して [Shift+Space] で出すことができます。a7 は 0167 の十六進数コードです。OS を問わず、日本語入力モードで「せくしょん」または「きごう」と入力して変換して出すこともできます。

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

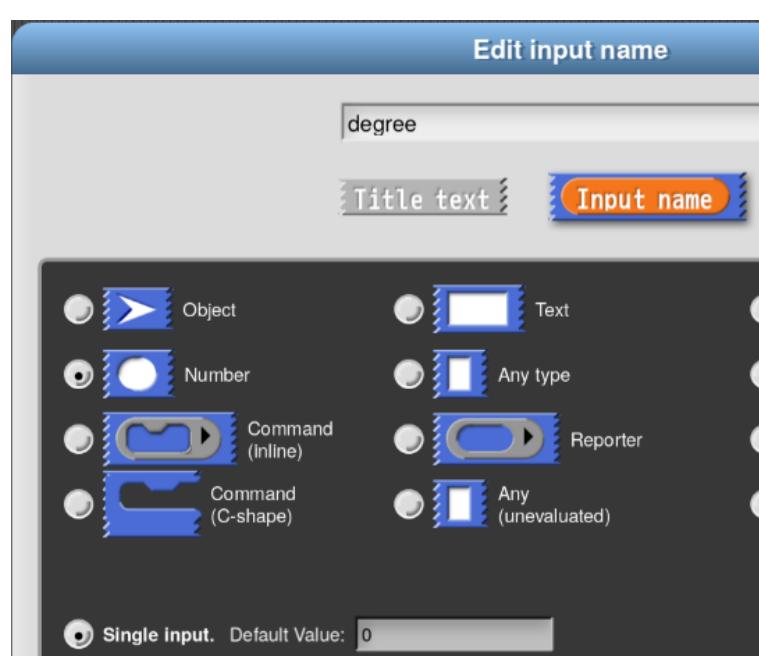
```

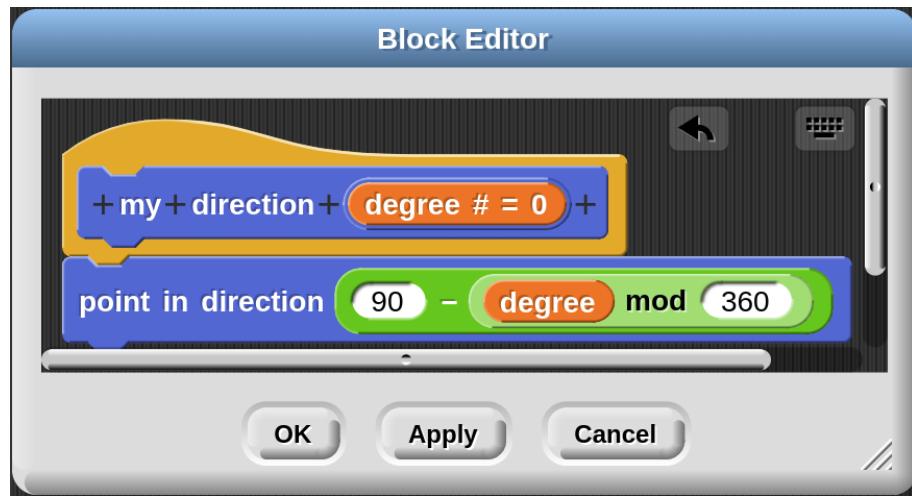
E(0)=0
N(90)=90
W(180)=180
S(270)=270
~
degree={
30={
EN(30)=30
WN(150)=150
ES(330)=330
WS(210)=210
}
45={
EN(45)=45
WN(135)=135
ES(315)=315
WS(225)=225
}
60={
EN(60)=60
WN(120)=120
ES(300)=300
WS(240)=240
}
}

```

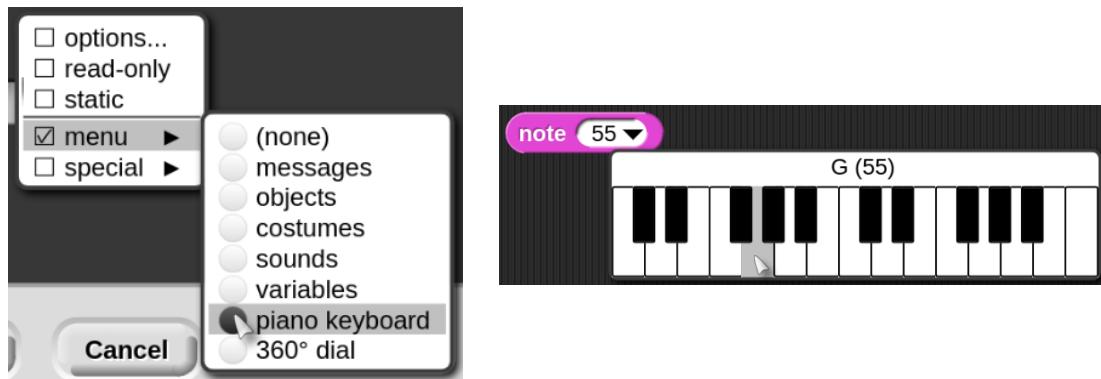


degree から、EN, WN, ES, WS のサブサブメニューを作っています。あくまで機能説明の作例ですので、あまり意味のあるものではありません。





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



7.2 Title Text とシンボル

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



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

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

そこからお目当てのシンボルを選択します。 turtle を選んでみます。

すると、入力欄に **\$turtle** がセットされます。

OK して Apply すると、 **test > █** になります。

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

シンボルメニューの最後に、「new line」があります。Title text にこれ **\$nl** を設定すると

そこで改行されます。また、シンボルじゃなくても、文字列の頭に「\$」を付けるとシンボルのように倍率と色の指定ができます。反面、シンボルと同じ文字列は使用できないということですが。

定義は、こうなります。



7.3 Input name オプションについて

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

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

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

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

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

| | | | | | | | |
|-------|-----------------|-------|----------------|-----|---------|-----|--------|
| a = 1 | default value | a ... | multiple input | a ↑ | upvar | a # | number |
| a λ | procedure types | a : | list | a ? | Boolean | a ≫ | object |

7.3.1 Reporter 型

Object を選択して次のような定義になると、 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。変数、スプライト、コスチューム、サウンドなどオブジェクトのドロップ入力を想定するものです。

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



Default Value を指定すると、



既定値を設定することができます。

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



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



リストの入力を求めていることをアピールするのが、List です。



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



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

実際には call ブロックを使ってリングブロックの値を求めます。



以前は `a` にリングが付いていないとエラーになったのですが、現バージョンでは `test a` と表示されるようです。

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



【 Object についての補足 】

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



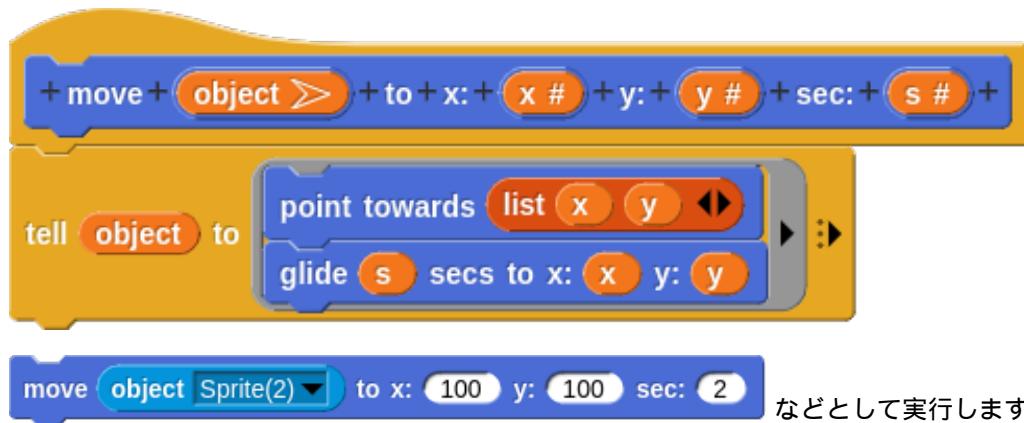
この入力スロットにオブジェクトを示す変数などをドロップすることもできます。 Sensing パレットにある Object リポーターブロックの入力メニューを開くと tell ブロックと同様に指定可能なオブジェクトを選択することができます。



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



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



`move [object Sprite(2)] to x: 100 y: 100 sec: 2` などとして実行します。

7.3.2 Predicate 型

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

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

Boolean です。



Predicate です。



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



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



7.3.3 Command 型

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

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

c 言語では、

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

のようにすると、0、1、2、3、4 と表示するプログラムになります。

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

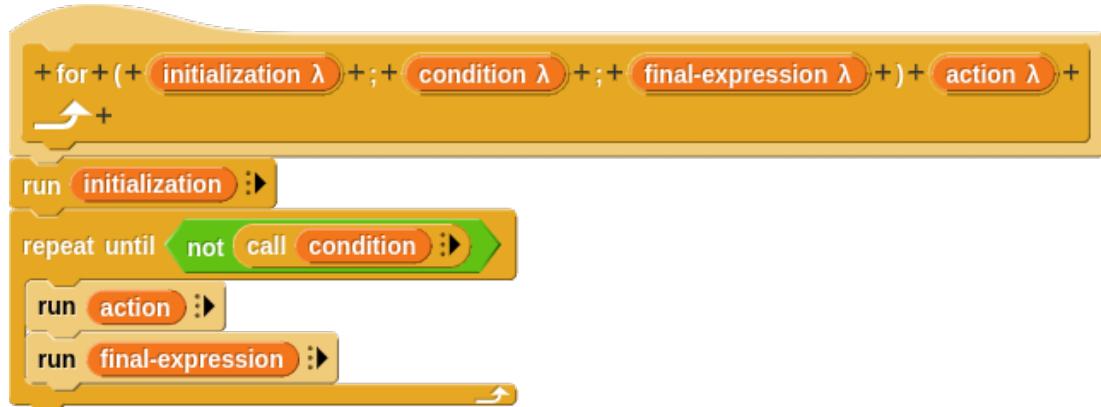
`i = 0` の部分が初期設定で、初めに一回だけ実行されます。

`i < 5` の部分が実行を続けるかどうかのテストをします。

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

{ } の内部がループの本体です。

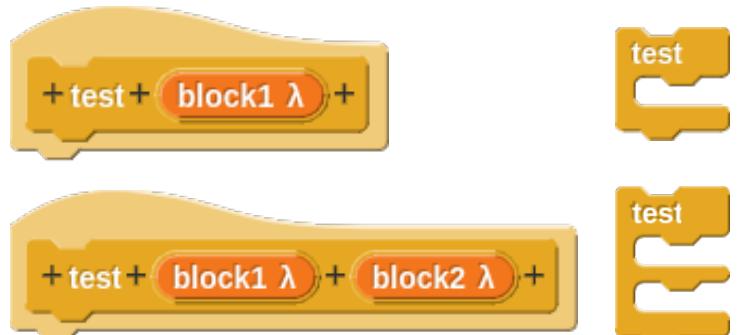
以下が定義です。プロトタイプの末尾に `text` で Loop を入れて矢印を表記しています。定義自体は `run` や `call` に丸投げするだけなので my for よりもシンプルですね。



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



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



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

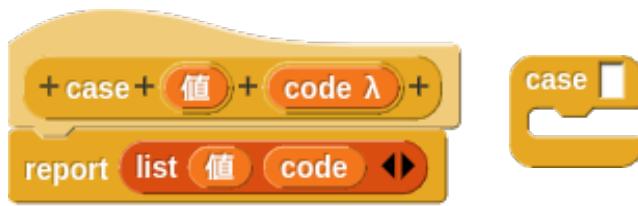


変数 block2 の前にも Boolean(unevaluated) の変数を入れ、block3 を追加すると変わった if else ブロックができます。



C 言語などで利用できる switch case ブロックを作ってみます。

switch の入力スロットで調べる変数を指定して、case ブロックで指定した値と一致した場合にその処理をします。case ブロックは追加できるようにし、default 処理も指定できます。
case ブロックは単に指定された値と実行ブロックの対をリストとしてリポートするだけです。



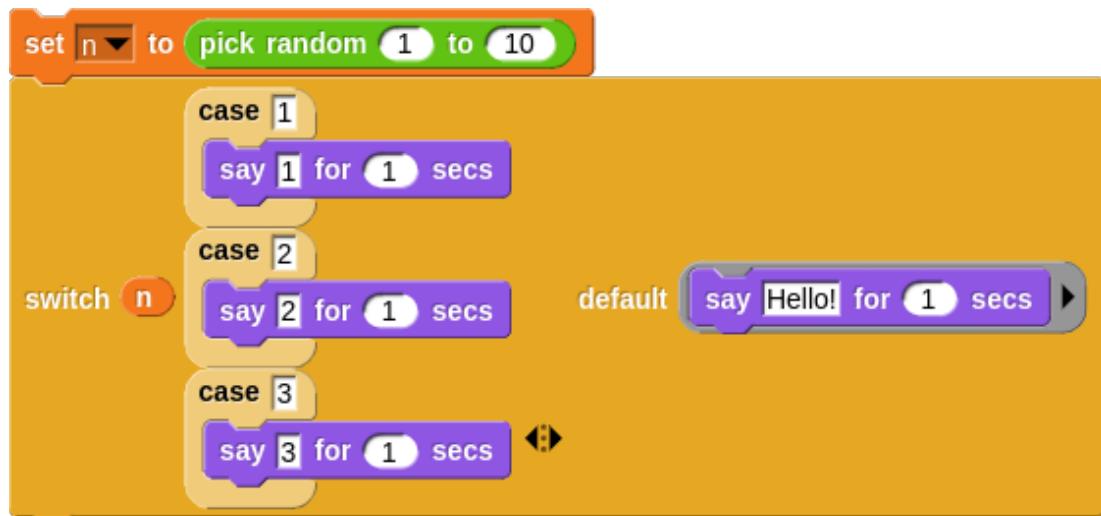
case ブロックは Reporter 型です。
code は Command(C-shape) 型
です。

switch の定義ブロックです。cases 変数は Command(inline) かつ Multiple inputs(value is list of inputs) を使って追加できるようにしています。これには上記のように、値と実行ブロックの対のリストが入っているので、switch で指定した値と case の値がマッチしたら指定されたブロックを実行します。マッチするものがなかったら default で指定されたブロックを実行します。



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

default は無指定、空のままでも大丈夫です。



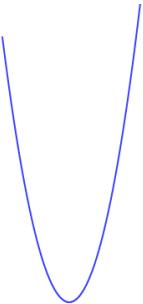
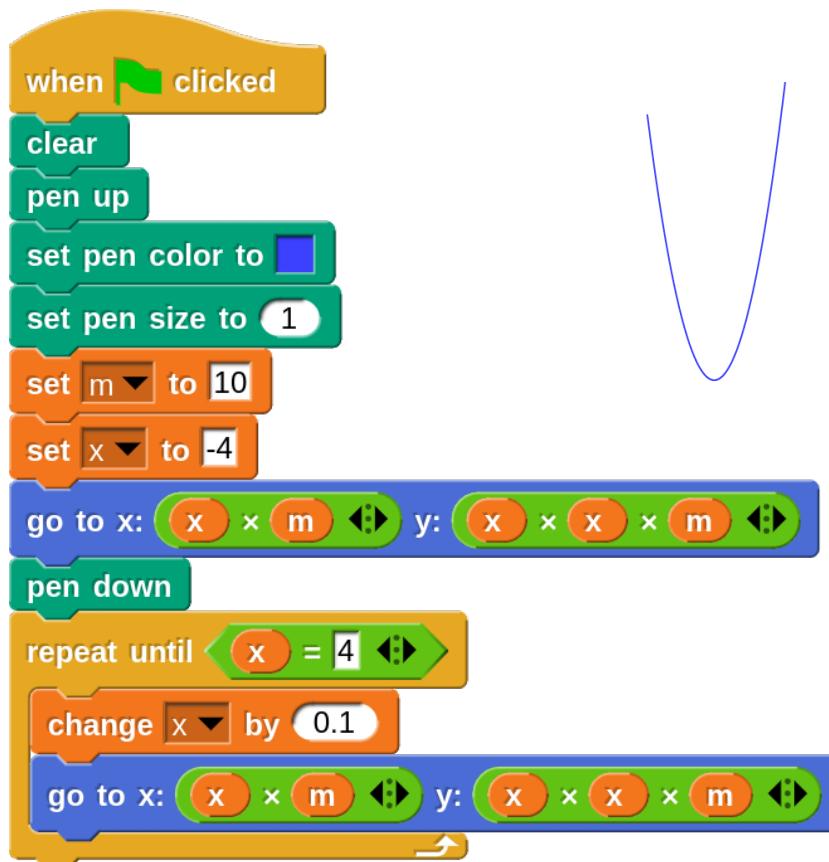
8 その他

8.1 デバッグ

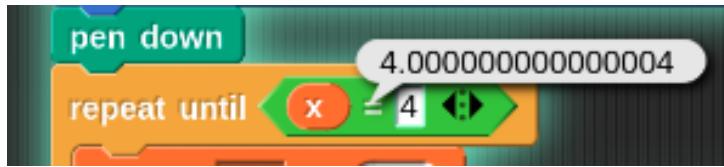
Snap! にはデバッグの機能が用意されています。

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

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



 を repeat until 内の最後に追加してから、スクリプトを直接クリックして実行させて下さい。そうすると、 $x \geq 3.9$ の時点で一時停止します。  のように、スライダーを左端にしてステップ実行モードにします。  をクリックして一ブロックずつ実行しながら x の値を確認して下さい。



このように、 x は 4 にはならないので無限ループになります。

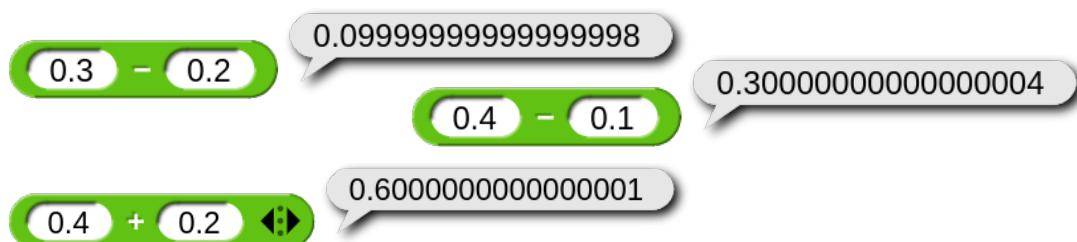
repeat until の終了条件を $x \geq 4$ にします。

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

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

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

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



ボタンをクリックすると、デバッグモード のオンオフができます。

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

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

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

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

8.2 = と identical

二つの値が同じかどうかを調べる ですが、

true や true

true を同値としては扱いたくない場合もあります。

その場合に使えるのが、 is [] identical to [] ? です。

is [] identical to [] ? false

is [] identical to [] ? false

上記は両入力スロットの値についてテストしましたが、リストに対する場合はリストの在り処が同じかどうかのテストをするようです。次は要素値は同じですが、別々に存在するリストです。

is [] identical to [] ? false

set [x] to [list [1][1]] set [a] to [x] x と a は同じリストを参照しています。

is [a] identical to [x] ? true

set [x] to [list [1][2]] is [a] identical to [x] ? false

リストの要素を変える場合は元のリストを修正するのではなく、新しくリストを作成してそれを指すようにするようです。a は元のリストを指したままなので false になります。

リストに関しても値の同値性だけをチェックするブロックを作ってみます。

「identical」ではなく「==」としてみました。

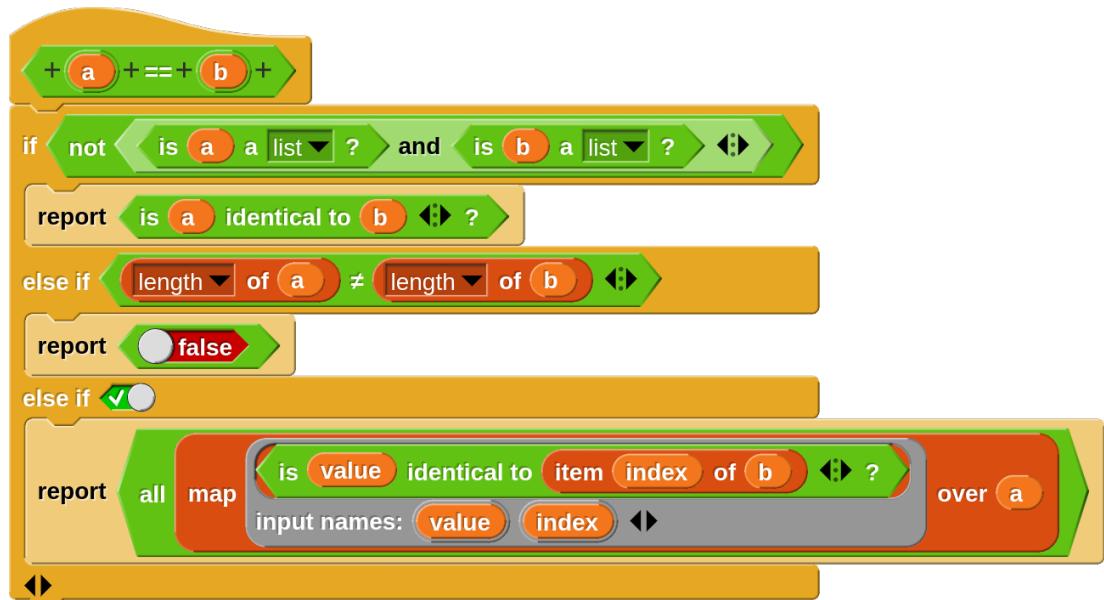
1 == 1.0 false a == A false

list [a][b][c] == list [a][b][c] true

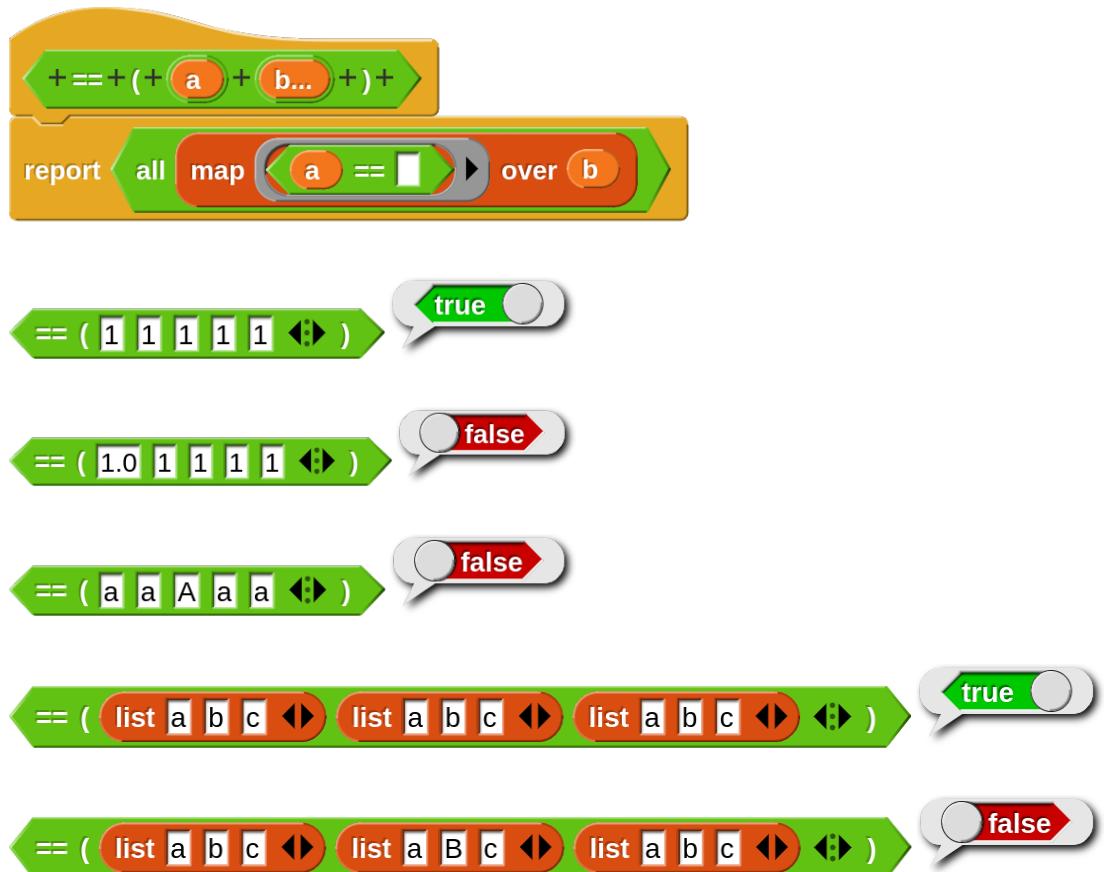
list [a][b][c] == list [a][B][c] false

list [1][2][3] == list [1.0][2][3] false

スクリプトです。a と b は Any type 型です。map を使うとすべての要素をテストすることになりますが速いです。なお、all は and の右端の三角マークの部分に map をドロップしたものです（82 ページ参照）



`is () identical to ()` では 入力スロットを追加できます。 を使って次のようにすると、同じことができます。b を Multiple inputs (value is list of inputs) にします。



8.3 連想配列、辞書

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

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

The image shows a Scratch script consisting of five blocks:

- set table [list] to []
- insert [list 1 one] at [1] of [table]
- insert [list 7 seven] at [1] of [table]
- insert [list 5 five] at [1] of [table]
- insert [list 3 three] at [1] of [table]

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

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

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

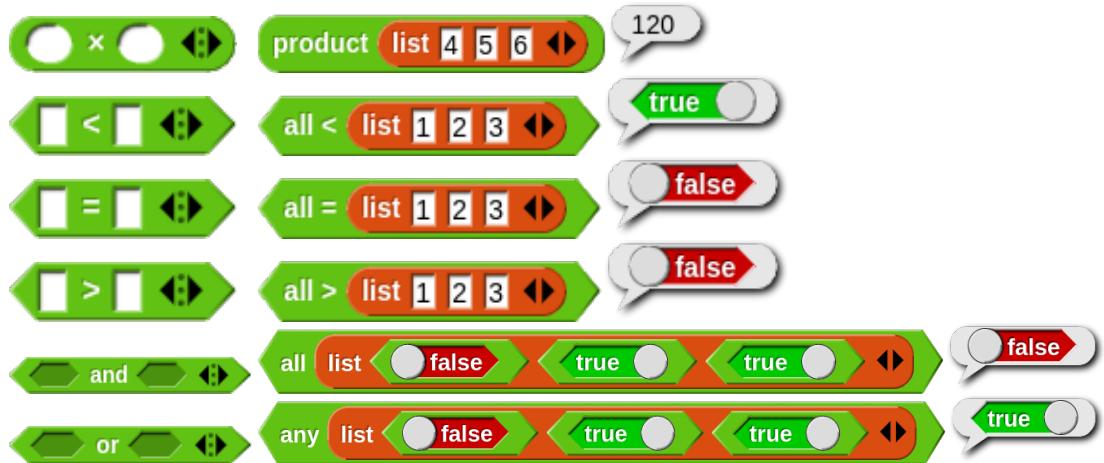
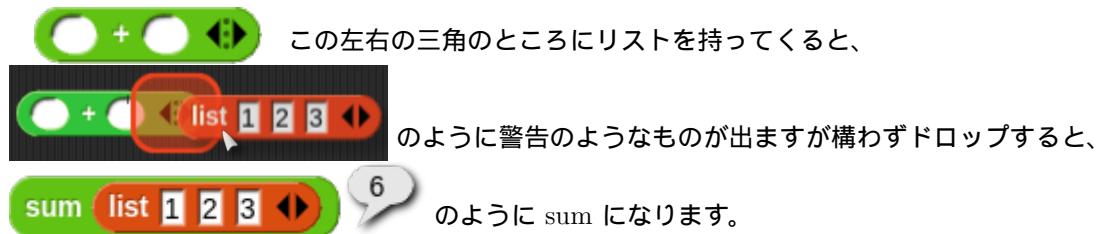
The image shows a Scratch script with the following components:

- A yellow "report" hat block.
- A "pipe" block with the condition "find first item" and the expression "item [1] of [目] = [key]".
- An "if" block with the condition "value = []" and the "then" branch containing a "report" block.
- The "report" block contains:
 - "item [2] of [value]" with "input names: value"
 - A speech bubble saying "three".
- Two "look up" blocks:
 - "look up [3] table" with a speech bubble saying "three".
 - "look up [6] table" with a speech bubble saying " ".

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

8.4 入力スロットへのリスト指定

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



xy 座標を次のようにリストにした場合、



8.5 ask

ask what's your name? and wait

ブロックの入力スロットにリストを入れてやると、リストで指定したメニューからクリックで項目を選択することができます。

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

メニュー1
メニュー2
メニュー3

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

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

タイトル
メニュー1
メニュー2
メニュー3

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

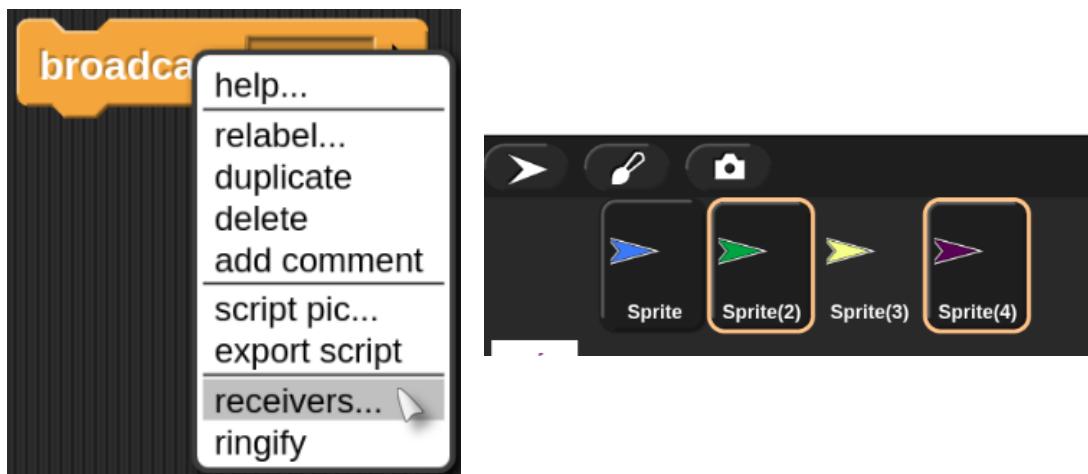
メニュー1 ▶ メニュー1.1
メニュー2 ▶ メニュー1.2

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

8.6 broadcast の検索オプション

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

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



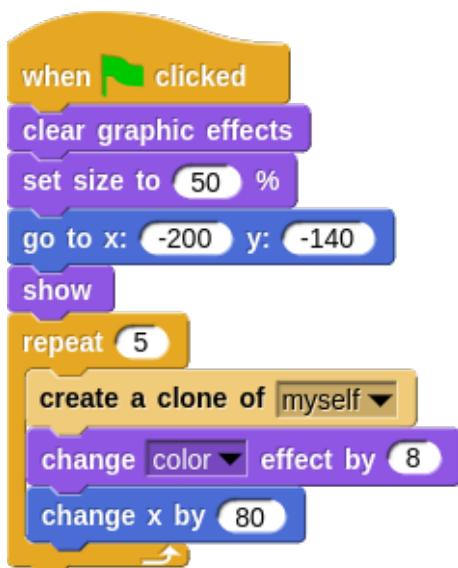
8.7 クローン

Snap! には、テンポラリクローンとパーマネントクローンがあります。

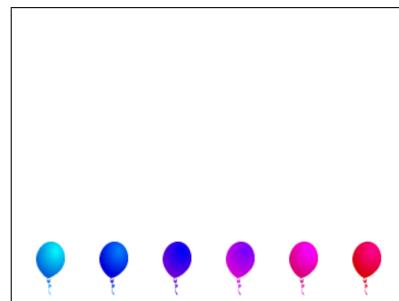
8.7.1 テンポラリクローン

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

次のようにして風船のクローンを作成してみます。



このスクリプトでは [show] により、作成された 5 個のクローンが左から表示され、一番右にクローンではない本体も表示されるので合計 6 個の風船が出現します。

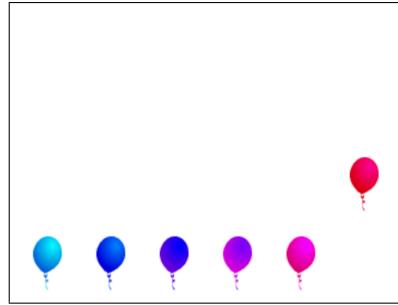


```

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

```

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



```

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

```

表示は [when I start as a clone] 内で行います。

```

when I start as a clone
  show

```

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

```

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

```

```

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

```

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

```

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

```

when I start as a clone

show

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

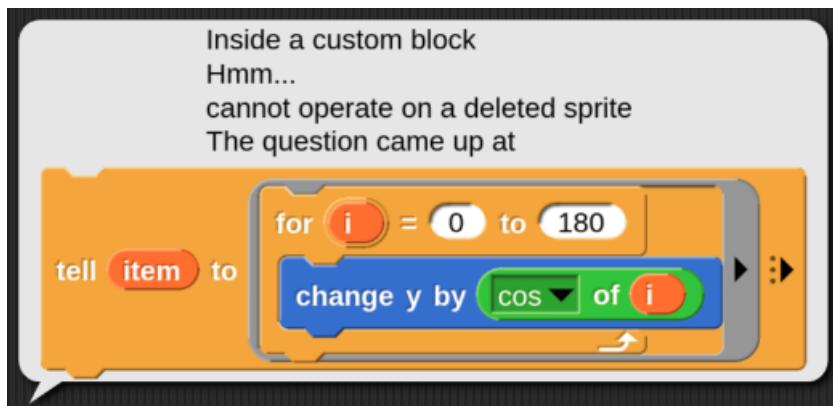
```

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

```

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

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



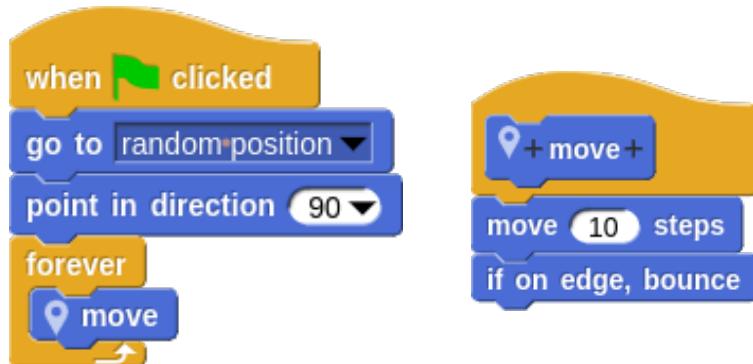
変数やリストに格納されたクローンが delete されているかどうかをテストするためのブロックは定義されていないようです。これに対処する一つの方法として、スプライト変数  にクローン番号を記憶させて、そのインデックスによって削除されたこと false を balloon リストに記録します。(クローン 1 は 1、クローン 2 は 2、... の値の変数 id を持ります。) id 番号をリスト内の位置にしているので、リストから要素を削除することはできません。



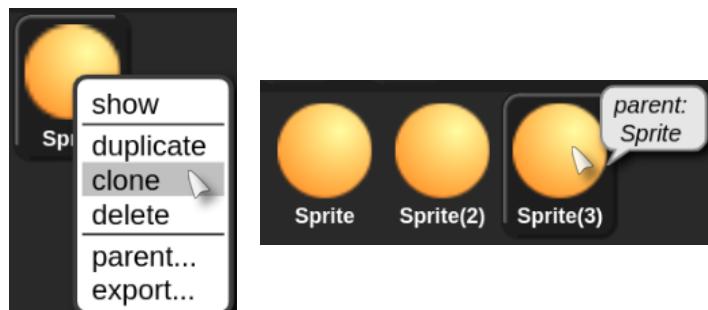
8.7.2 パーマネントクローン

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

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



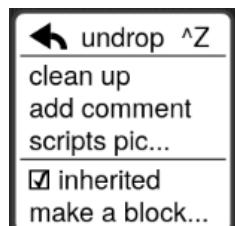
スプライトコラルで `Sprite` を右クリックして、クローンを二つ作成します。作成されたクローンのところにマウスカーソルを置くと、`parent:` 親が `Sprite` であることを表示します。



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

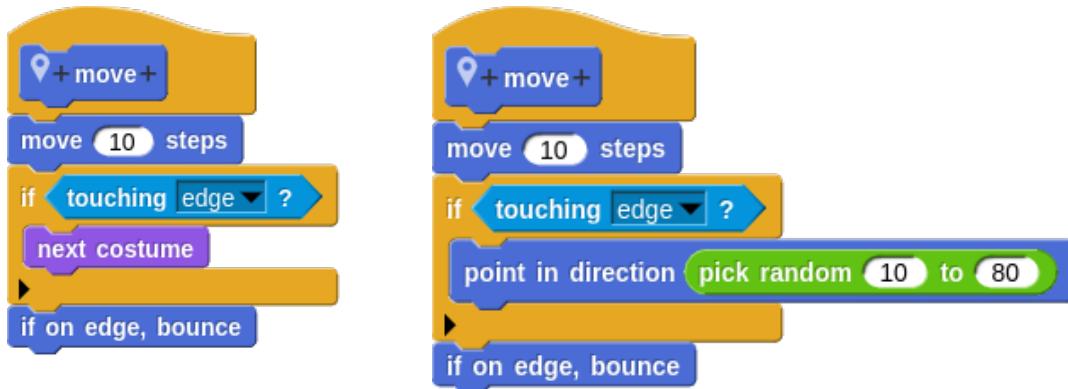
スクリプトエリアで右クリックすると右図のように `inherited` がオンになっています。この状態だと子側も親側のスクリプトと同じになります。

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

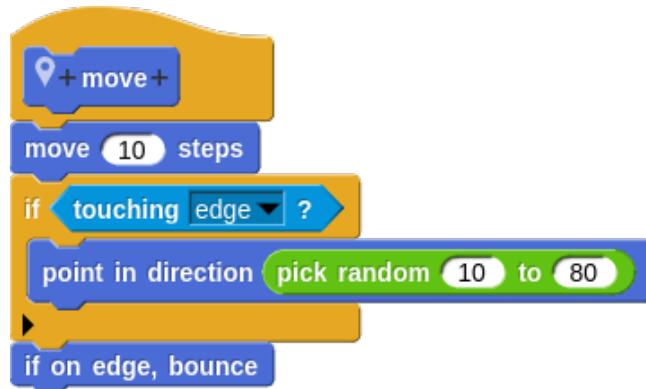


クローンを作成した時に親側のユーザー定義ブロックも子側にコピーされています。しかし、この `move` ブロックは `for this sprite only` で作成されているので、親側で変更しても子側は変わりません。子側で変更しても `inherited` の状態は変わりません。次のように子側の `move` 定義を変更してください。各スプライトで同じ名前の `move` ブロックですが、それぞれ違った動作をさせることができます。

Sprite(2)



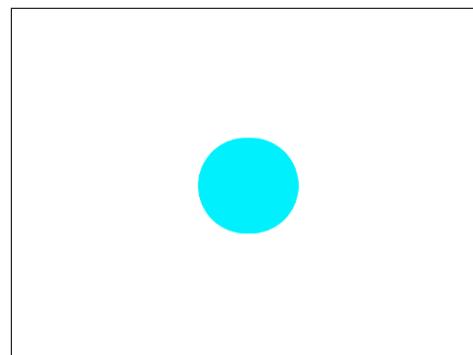
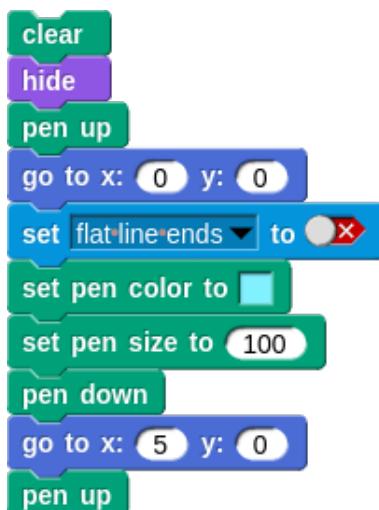
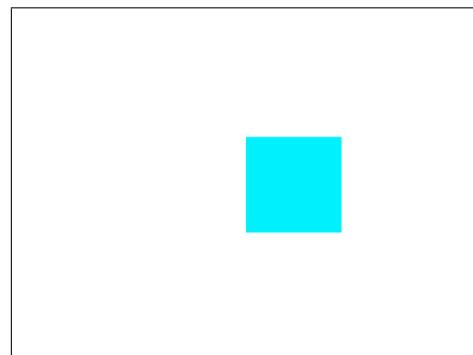
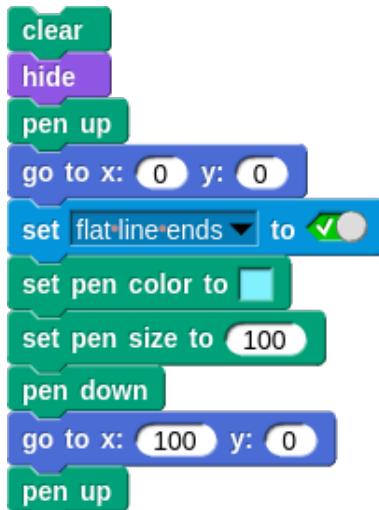
Sprite(3)

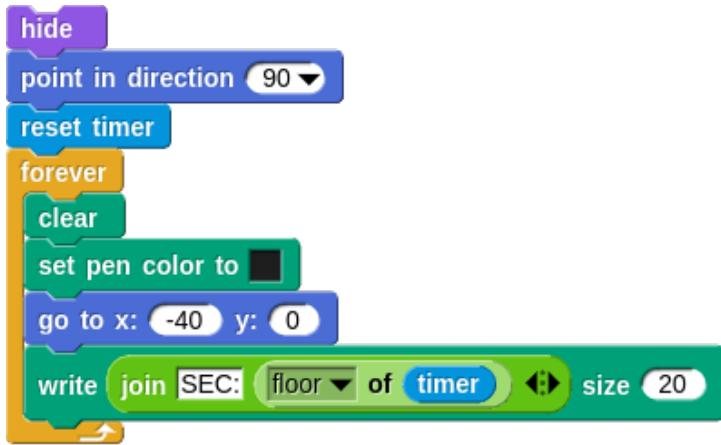


8.8 flat line ends

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

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

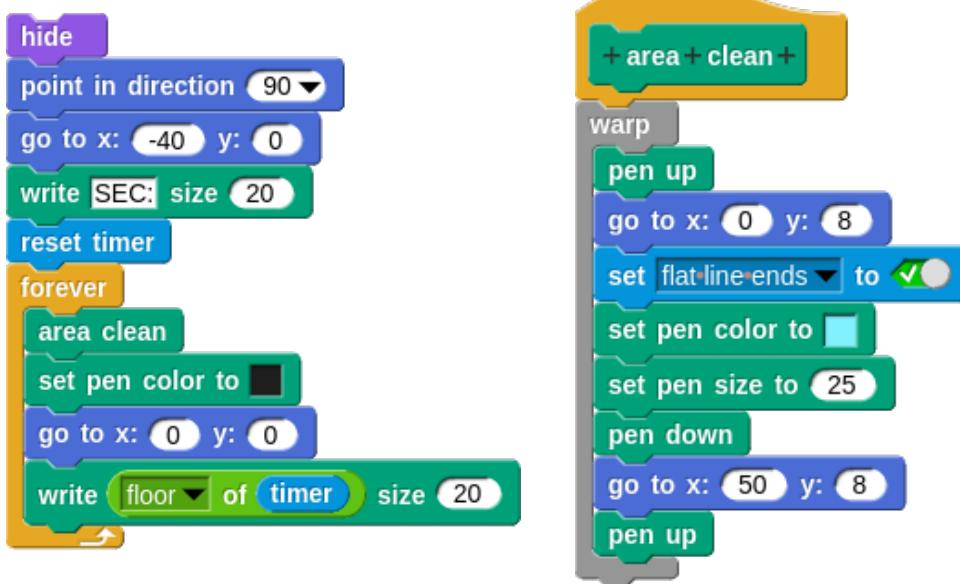




ステージにデータを表示する場合に普通は変数ウォッチャーを使いますが、Penを使って左のようになります。

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

データの部分だけを消すやり方です。



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

8.9 角度

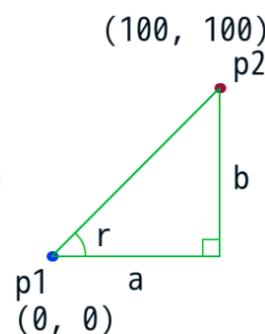
スプライトがある方向に向けたい場合、**point towards** Sprite(2) のようにすれば他のスプライトの方向に向けられます。ある座標に向けるならば **point towards** list Px Py のように指定できます。

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

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

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

atan of p2y - p1y / p2x - p1x = 45

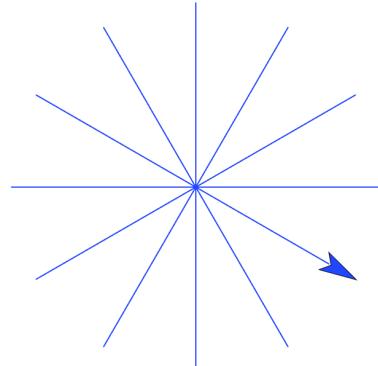


atan を使用すると象限によって向きが逆になるので atan2 を使用します。

```

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

```

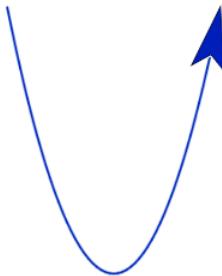


方程式のグラフを描く場合、微分によって接線の傾きが求められるのでスプライトの角度に利用することができます。

```

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

```



```

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

```

100 / 0 Infinity

atan of 100 / 0 90

atan2 100 / 0 90

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

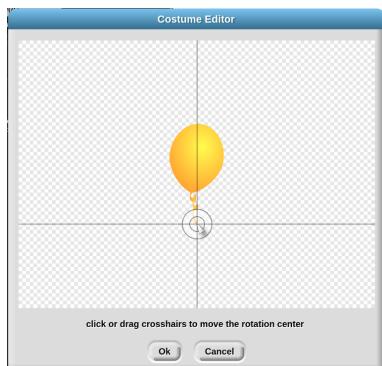
8.10 anchor アンカー

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



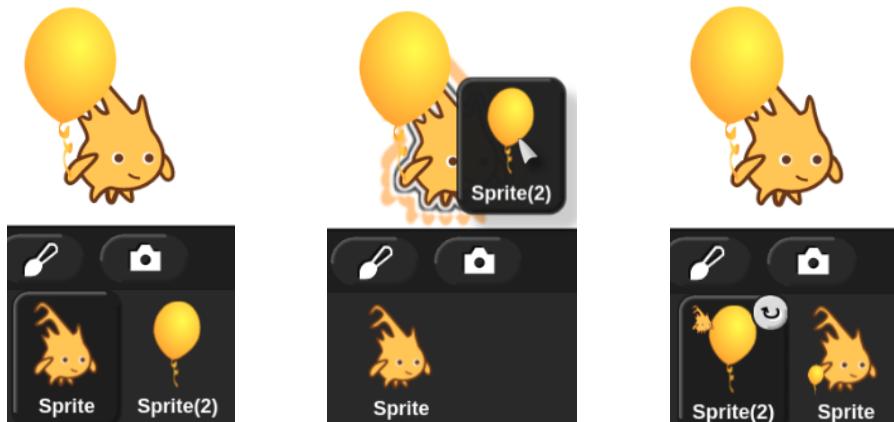
alonzo というコスチュームの手に風船を持たせてみます。風船は alonzo に持たれてはいますが、独自の動きをします。

alonzo と balloon のコスチュームをそれぞれのスプライトに設定してください。



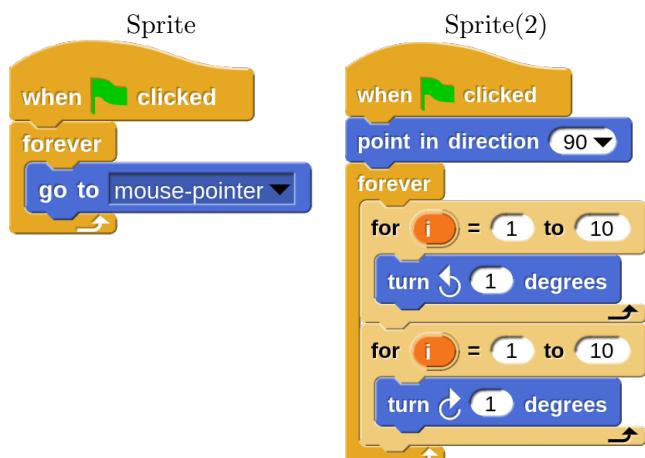
balloon をエディターで開いて、動きや回転の中心を糸の端にします。

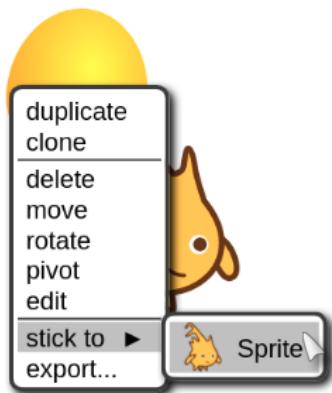
アンカー操作には二つのやり方があります。コラールにある風船のスプライトをステージ上の alonzo のスプライトのところに持っていくとハロがでます。ドロップするとコラールのスプライトの表示が変化します。



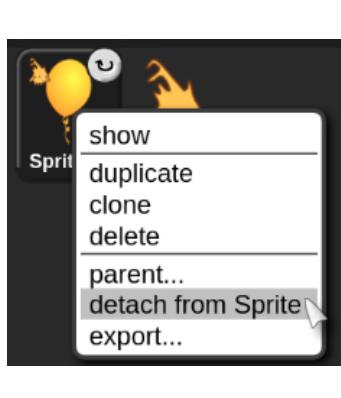
テストスクリプトです。

alonzo は動く風船を持ちながらマウスポインターに付いて行きます。





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



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

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

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

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

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

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

変数を使って演算をしてそれを返すこともできます。

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

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

```

call [JavaScript function v]
  [list <-->]
  {
    return list;
  }
with inputs [numbers from 3 to 1 <-->]

```

1 3
2 2
3 1
+ length: 3

JavaScript の配列ソートを利用しようとするとエラーになります。

```

call [JavaScript function v]
  [list <-->]
  {
    list.sort(function(a,b){
      return a-b});
    return list;
  }
with inputs [numbers from 3 to 1 <-->]

```

TypeError
list.sort is not a function

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

```

call [JavaScript function v]
  [list <-->]
  {
    var l = list.toArray();
    l.sort(function(a,b){
      return a-b});
    return l;
  }
with inputs [numbers from 3 to 1 <-->]

```

1 1
2 2
3 3
+ length: 3

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

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

```

call [JavaScript function v]
  [list <-->]
  {
    list.toArray().sort(function(a,b){
      return b-a});
    return list;
  }
with inputs [numbers from 3 to 1 <-->]

```

1 3
2 2
3 1
+ length: 3

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

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

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

主要なビット演算として、論理積 (AND)、論理和 (OR)、排他的論理和 (XOR) があります。2 つの 1 ビットの数 `n1`, `n2` が取り得る値 0, 1 に対してのそれぞれのビット演算の結果を示し

ます。

| n1 | n2 | AND | OR | XOR |
|----|----|-----|----|-----|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

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

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

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



`1 AND 0` と `1 AND 1` の部分を “`&`” にすれば OR ブロック、
“`^`” にすれば XOR ブロックになります。

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



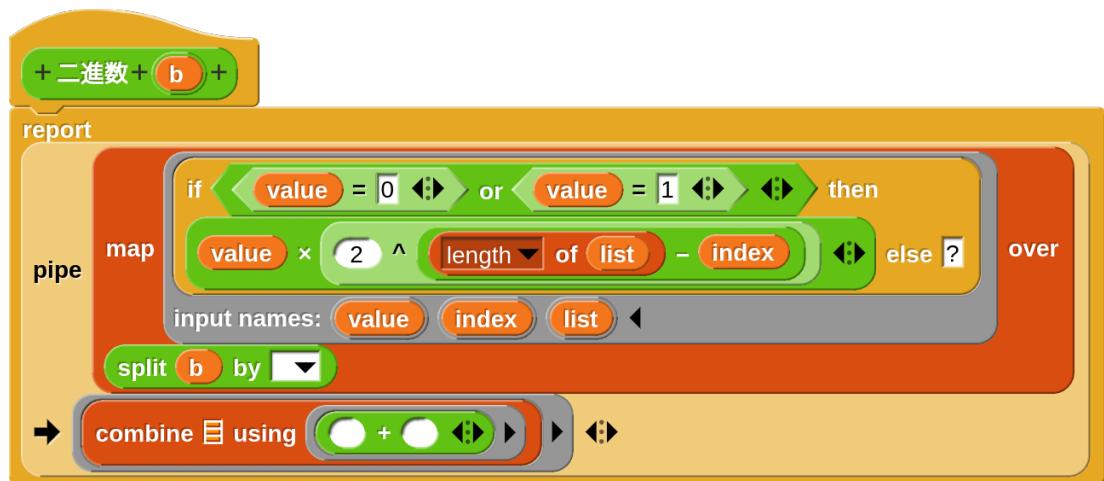
`NOT 0` の NOT が `-1` というのは分かりづらいかもしれません。数値の桁数を 4 ビット (ニブル) に限定してみます。すると表せる数は 0000 ~ 1111 (十進数では 0 ~ 15) になります。0000 の NOT なので 1111 になります。二進数では、最上位ビットをサインビット (符号ビット) として使用することで負数も扱えるようにしています。その場合、1111 は最上位ビットが 1 なので負数です。 $1111 + 0001$ を行うと桁上がりをしていて 4 ビットの範囲では 0000 になります。つまり、1111 は 1 に加えると 0 になる数である `-1` ということになります。ニブルで表せる符号付き数は 1000 ~ 0111、十進数の -8 ~ 7 になります。整数を 64 ビットで表すならば、0 と `-1` はそれぞれ 0 と 1 が 64 個並んだものになります。

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



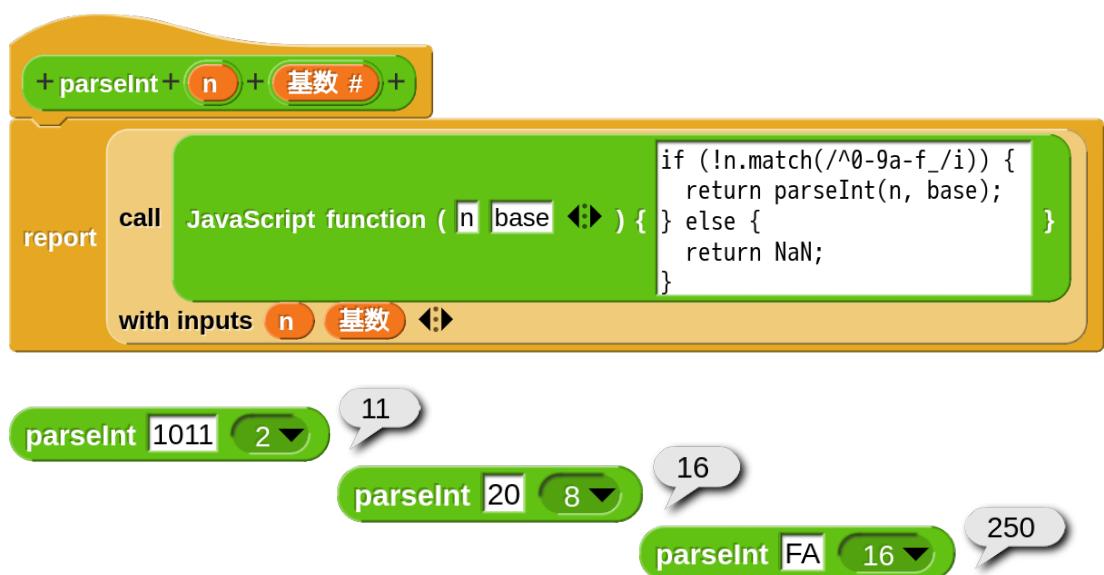
二進数で数値を指定するリポーターブロックを作成してみます。b の入力のタイプは text です。
split で指定するのは空文字です。(デフォルトの空白を削除します。)

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

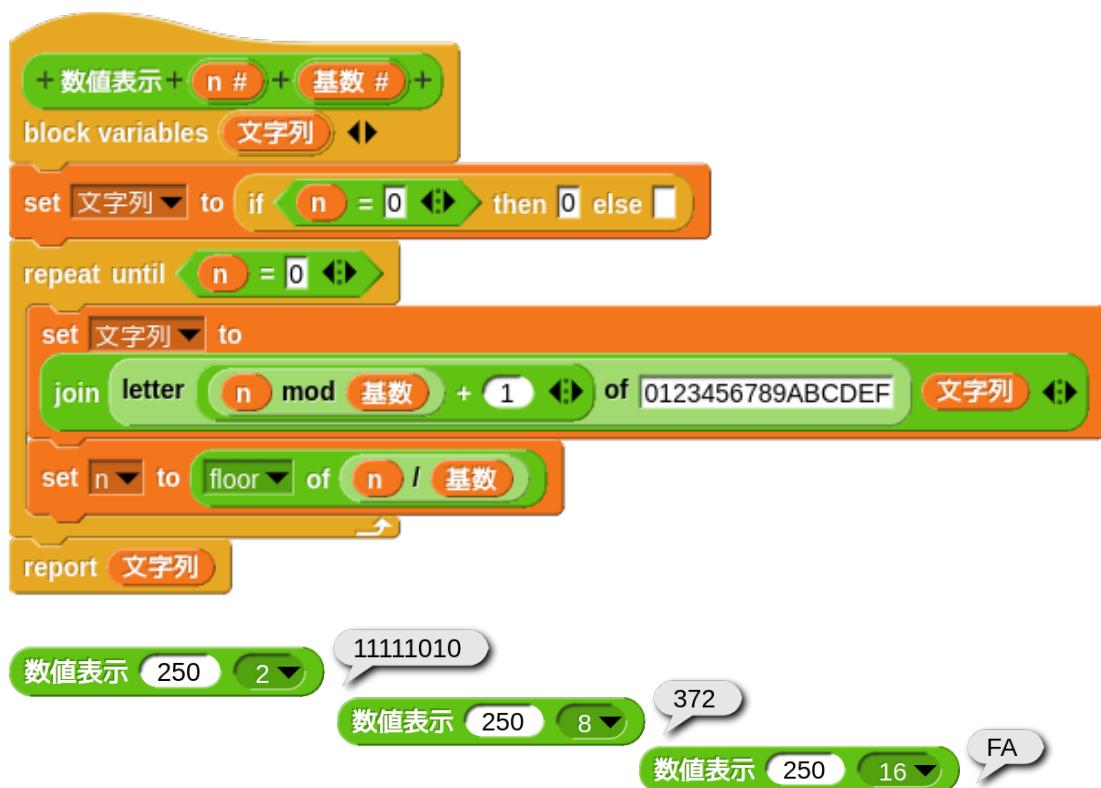


二進数では 0 と 1 しか使ないので、それ以外は「?」に置き換えてエラーになるようにします。

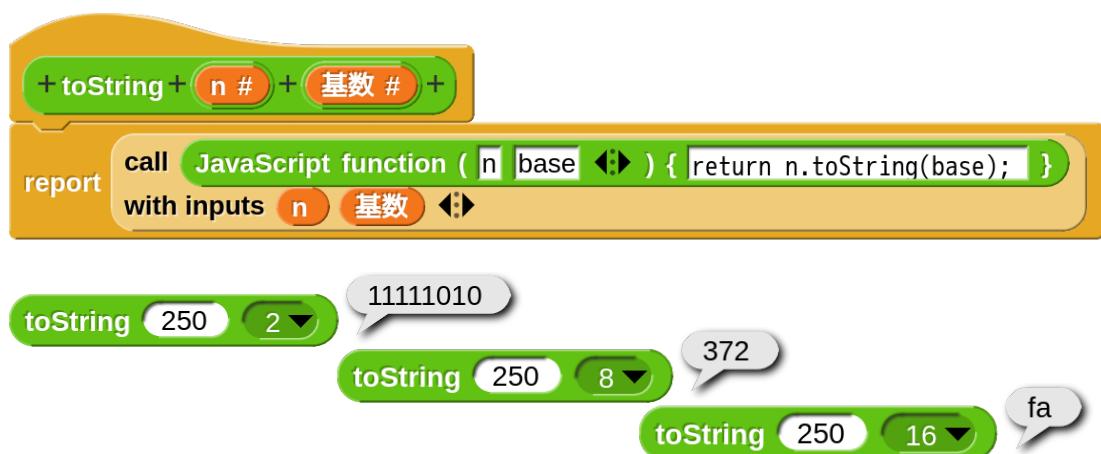
JavaScript には文字列を整数に変換する `parseInt` 関数があります。2 番目の引数で基数を指定できます。2 で二進数、16 で 16 進数の文字列からの変換になります。なお、「`base`」と「基数」が合っていませんが、JavaScript の引数は名前ではなく `inputs` のスロット位置に対応した値が設定されます。



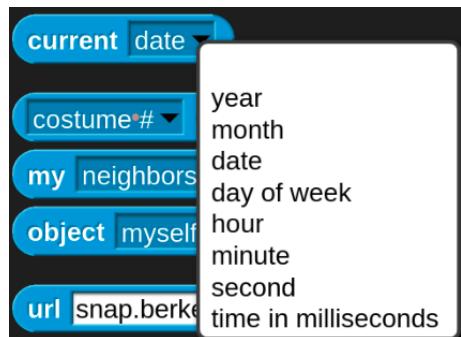
こうなると、数値を二進数表示するブロックも必要です。どうせなら基底を指定して文字列にする定義ブロックにしてみます。ただし、対応しているのは 2, 8, 10, 16 進数などです。



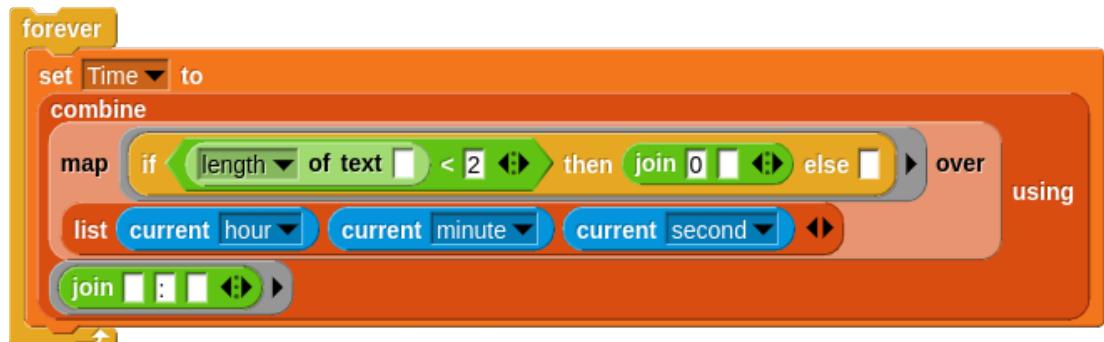
JavaScript では `toString` がそれをしてくれます。



8.12 時計



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



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

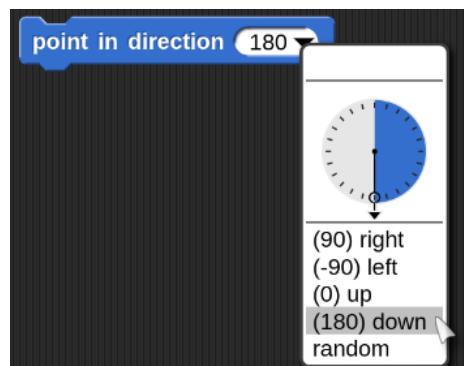
秒数は、**current second** で求められます。

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



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

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



秒針の角度です。(60 秒で一回転)



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

set minute ▾ to UTC / 60 mod 60

分針の角度です。(60 分で一回転)

point in direction minute × 360 / 60

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

set hour ▾ to UTC / 60 × 60 mod 24

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

時針の角度です。(12 時間で一回転)

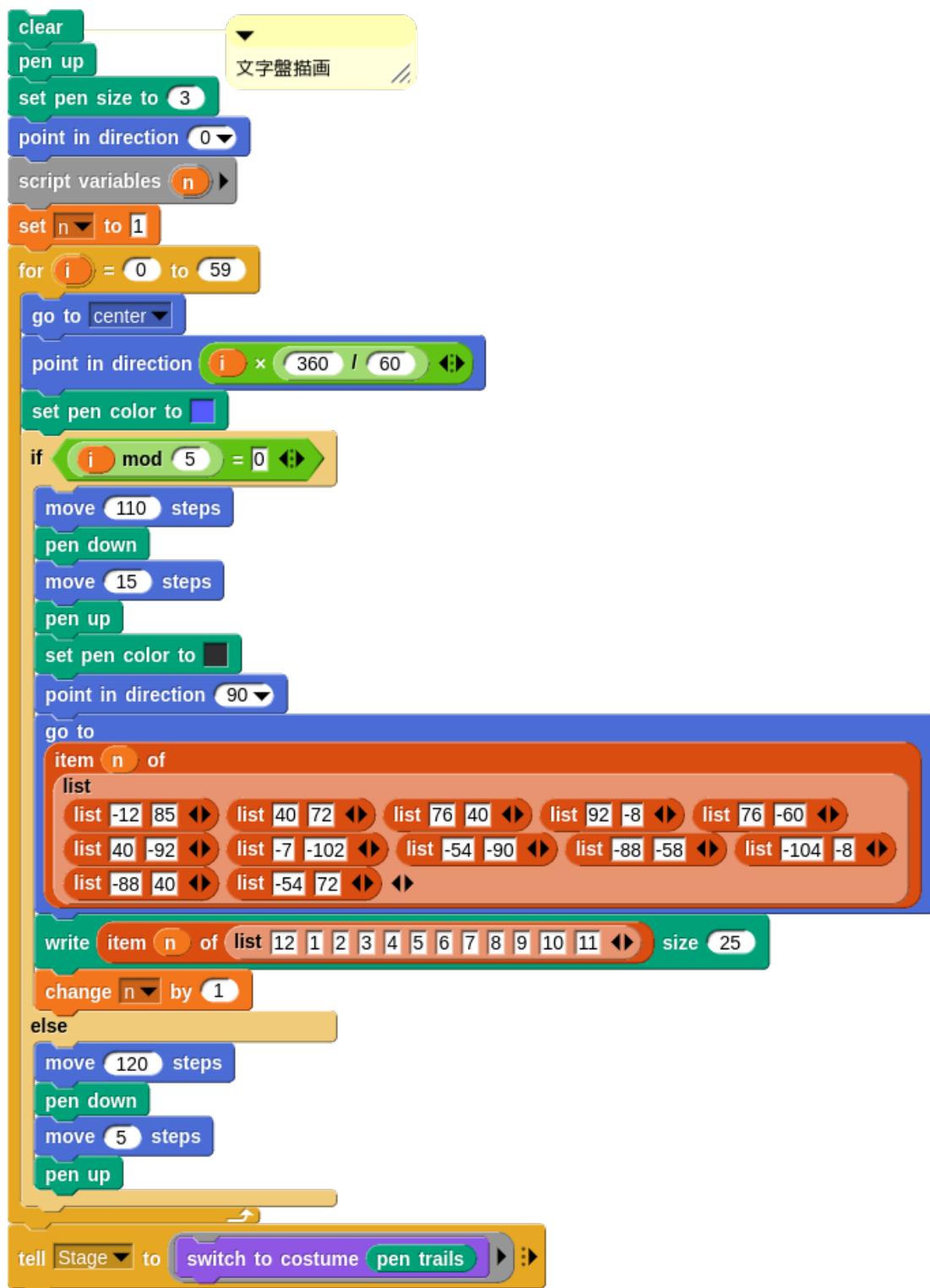
point in direction hour × 360 / 12

ペンで描く場合はこの逆の順序で描けば実際の時計と同じ重なりになります。

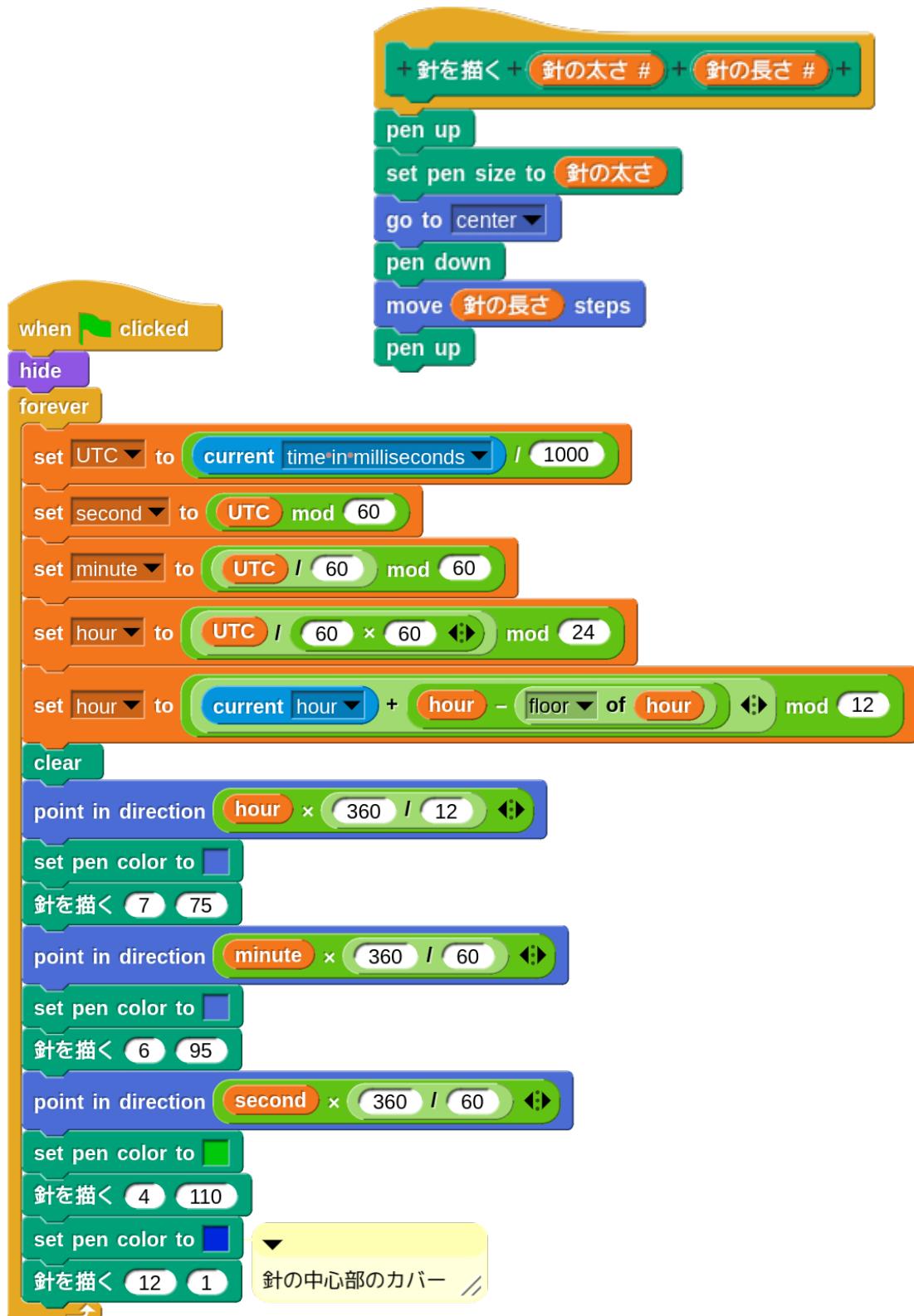
Scratch や Snap! の角度指定で画面上方向が 0 なのは時計やメーターの針の角度を指定しやすくするためかもしれません。

次のページから文字盤を描画し、ペン描画による時計のスクリプトを示します。

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

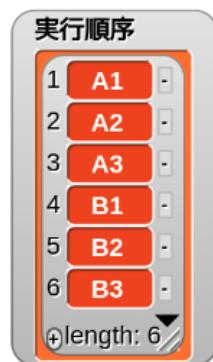
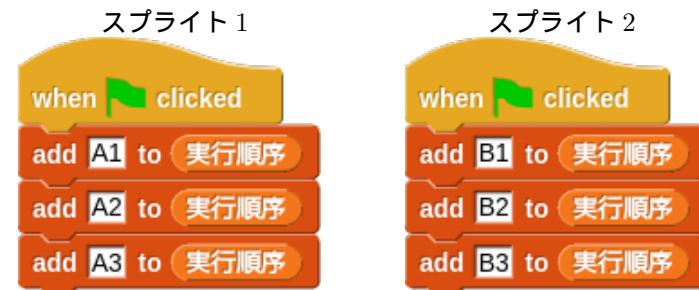


針を描くための定義ブロックです。



8.13 並列実行について

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

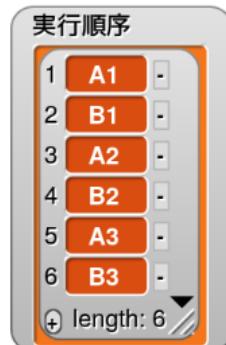


set [実行順序 ▾] to [list ▶] を実行してから  をクリックすると、左のようにリストに実行順序が表示されます。一ブロックずつ交互に実行されるのではなく、一方はもう一方のスクリプトの実行が終わるまで待たれます。実行の順番はスプライト、スクリプトが作成された順番になるようです。

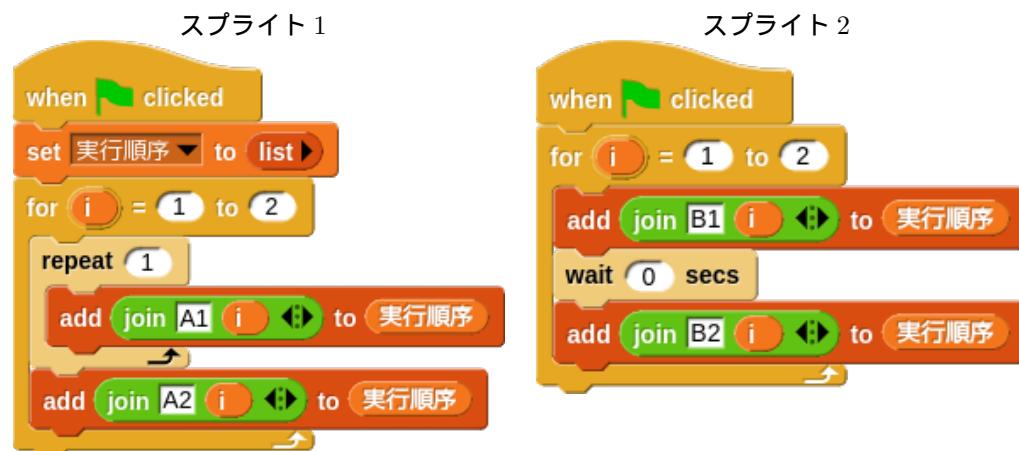
スプライト内に複数の **when green flag clicked** のスクリプトがあれば、基本的にそれがすべて完了してから他のスプライトのスクリプトの実行に移ります。他のスクリプトへの処理移行のタイミングは繰り返し処理、つまり C 型ブロック内の末端に到達した時にも起こります。



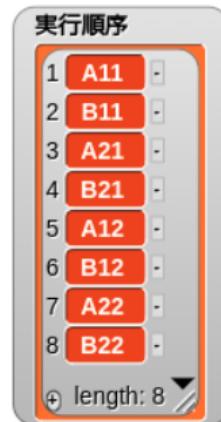
for の C 型ブロックにより、 A? と B? が交互にリストに加えられました。（? は 1 ~ 3 の数値を表します。）



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



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



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



8.14 行列の積

8.14.1 inner product

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

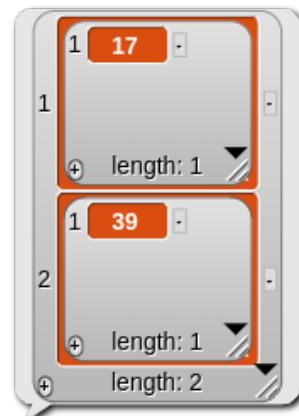
2行2列の行列同士では次のような計算方法になります。

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

a側の行列の列数とb側の行列の行数は同じ必要があります。

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

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



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

| | | |
|---|----|----|
| 1 | A | B |
| 1 | 40 | 46 |



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



にする必要があります。

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

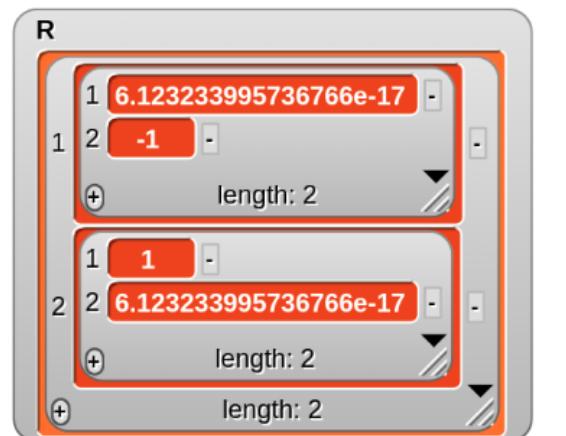
$$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

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

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



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

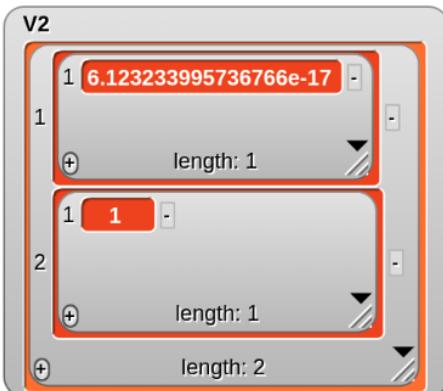


$$\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$$

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

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





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

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

$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$ の変換結果が $\begin{pmatrix} x' \\ y' \end{pmatrix}$ となり、 $\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix}$ の変換結果が

$\begin{pmatrix} x_1' & x_2' \\ y_1' & y_2' \end{pmatrix}$ となります。

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

転置行列を使うと、普通の (x, y) 座標の指定の仕方ができます

(x, y) 座標のリストで与えられた図形を一筆書きで描くブロックです。

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



三角形 を描くスクリプトです。

```

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

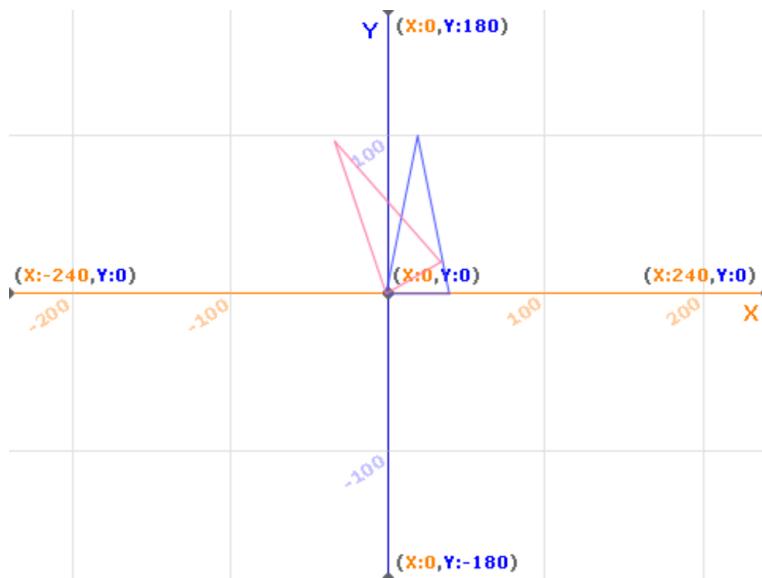
```

それを半時計回りに 30 度回転させるスクリプトです。

```

set [θ v] to [30]
set [R v] to [list [list [cos [v] of [θ]] [neg [v] of [sin [v] of [θ]]] <--> [list [sin [v] of [θ]] [cos [v] of [θ]]] <--> <-->]]
set [V2 v] to [inner product [R] [[+ [x] [y] <--> [x * y]] <--> [columns [v] of [V0]]]]
set pen color to [magenta]
draw [columns [v] of [V2]]

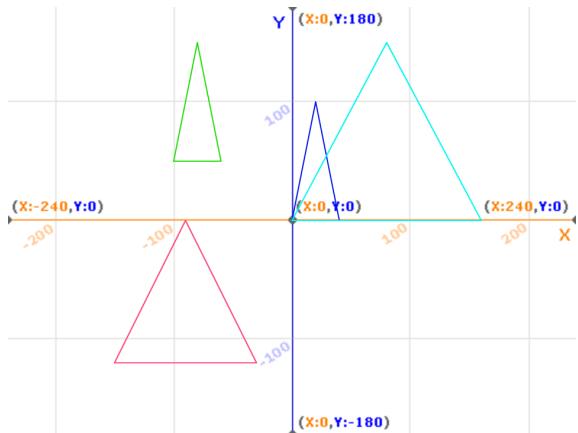
```



[参考文献]

『Python ではじめる数学の冒険
プログラミングで図解する代数、幾何学、三角関数』
Peter Farrell 著 鈴木幸敏 訳 オライリー・ジャパン 刊

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



x 軸方向に 4 倍、y 軸方向に 1.5 倍します。

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

```

set V2 to
  inner product list [4 0] list [0 1.5]
  columns of V0
  +
  *
  columns of V2

```

```

set pen color to blue
draw columns of V2

```

平行移動の場合は、補助的に要素を加える必要があります。

x 軸方向に -100、y 軸方向に 50 平行移動します。

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

```

set V2 to
  inner product list [1 0 -100] list [0 1 50] list [0 0 1]
  +
  *
  columns of map [append [list [1]] over V0]

```

```

set pen color to blue
draw columns of V2

```

三番目の補助的要素は無視してくれるようです。

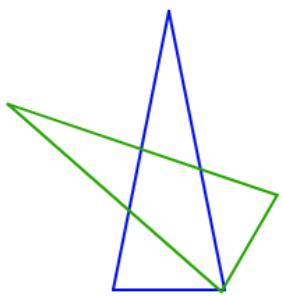
Snap! ではリスト全体に対する演算ができるので、次のように拡大縮小 (x 軸方向に 3 倍、y 軸方向に 1.2 倍) と平行移動 (-150, -120) が一度にできます。

```

set pen color to red
draw V0 * list [3 1.2] + list [-150 -120]

```

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



(G_x, G_y) を V_0 の重心にし、原点に平行移動し、回転させ、元の位置に戻します。
と言っても、座標値を変換しているだけであちこち移動させているわけではありませんが。

```

set Gx to
item 1 of item 1 of V0 + item 1 of item 2 of V0 + item 1 of item 3 of V0 / 3
set Gy to
item 2 of item 1 of V0 + item 2 of item 2 of V0 + item 2 of item 3 of V0 / 3
set θ to 60
set R to [list cos of θ neg of sin of θ
          list sin of θ cos of θ]
pipe V0 - [list Gx Gy] →
  columns of [ ] →
  inner product R [ + [ ] × [ ] ] →
  columns of [ ] →
  [ ] + [list Gx Gy] →
  [ ]
set pen color to green
draw V2

```

8.14.2 outer product

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

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

この記号のようなものはギリシャ文字の ノイオタです。

次のようにすると、九九の表の一部ができます。

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

outer product l 3 o. x l 3

計算方法を表示させてみます。

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

outer product l 3 o. join x l 3

単位行列なども作成できます。

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

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

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

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

[参考文献]

『基礎からの APL 解説と例題例解』

西川利男/日本アイビー・エム 共著 サイエンスハウス 刊

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

set N ▾ to L 5

1 ~ 5 の整数のリスト N を作り、その中の素数を求めてみます。

まず、そのリストの各要素に対して、そのリストの各要素で割った余りが 0 のものを調べます。
(111 ページ、outer product の計算方法で × の代わりに mode になります。)

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

pipe N → outer product mod = 0 / 目

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

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

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

combine in rows (reduce by column vectors)

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

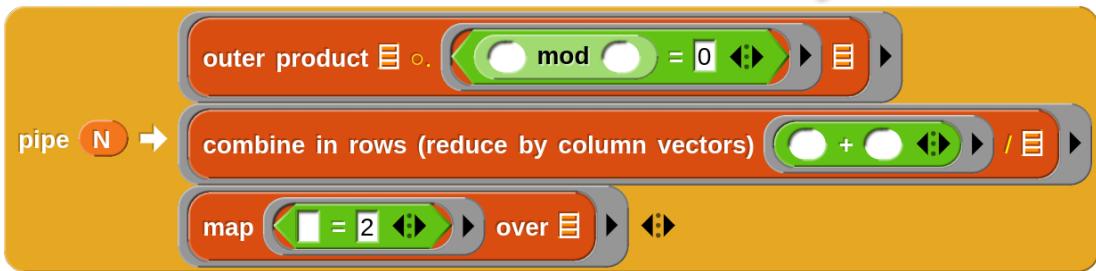
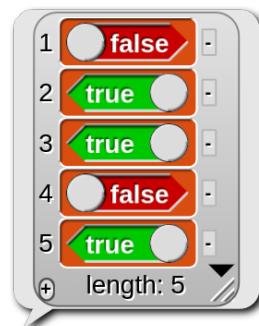
| | | |
|---|-----------|---|
| 1 | 1 | - |
| 2 | 2 | - |
| 3 | 2 | - |
| 4 | 3 | - |
| 5 | 2 | - |
| + | length: 5 | ▼ |

pipe N →

outer product mod = 0 / 目

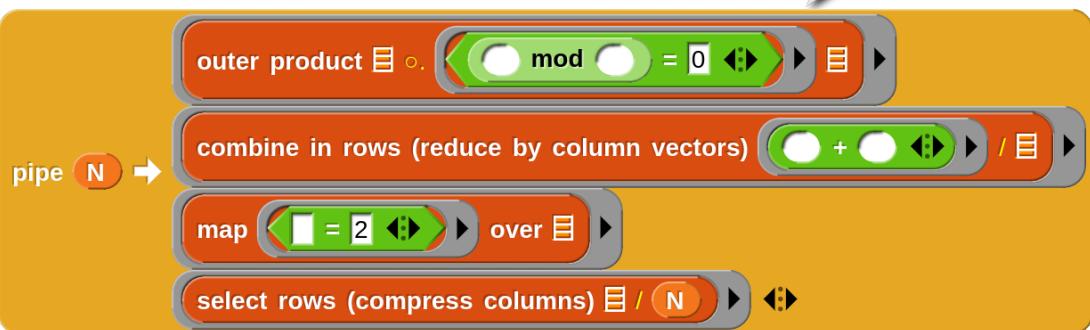
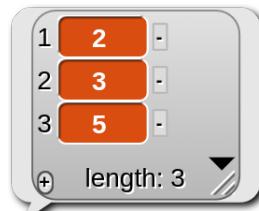
combine in rows (reduce by column vectors)

行中の true の要素の合計が 2 のものを map を使って求めます。



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

select rows (compress columns) 〇 / N



次のようにまとめることもできます。

select rows (compress columns)

map 〇 = 〇 〇 over

combine in rows (reduce by column vectors) 〇 + 〇 〇 / 〇 / N

outer product N 〇 mod 〇 = 〇 〇 N

9 再帰呼び出し

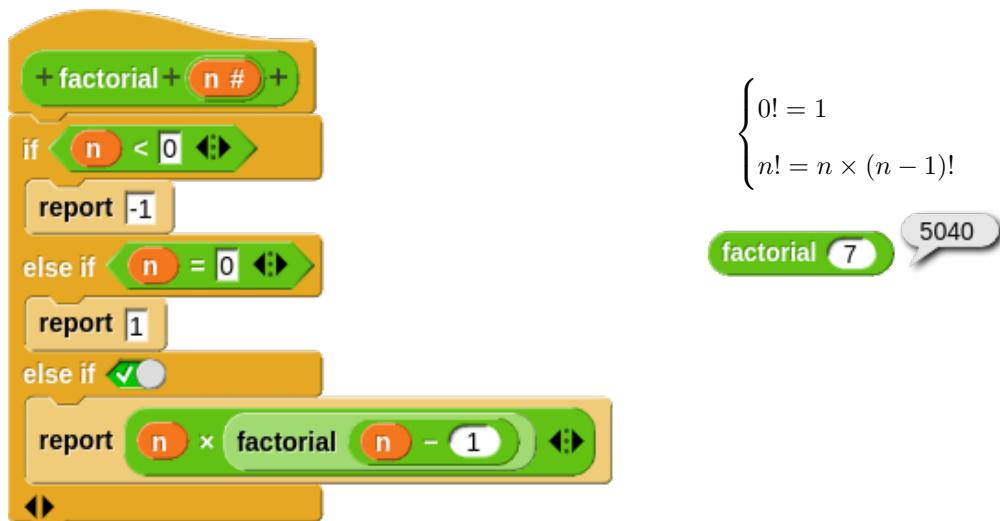
再帰、再帰呼び出し（recursive call）は作成したブロックの中で自分自身を呼び出す（実行する）ものです。関数型プログラミングでは再帰呼び出しが繰り返し処理の手法になっています。

9.1 再帰呼び出しの例

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

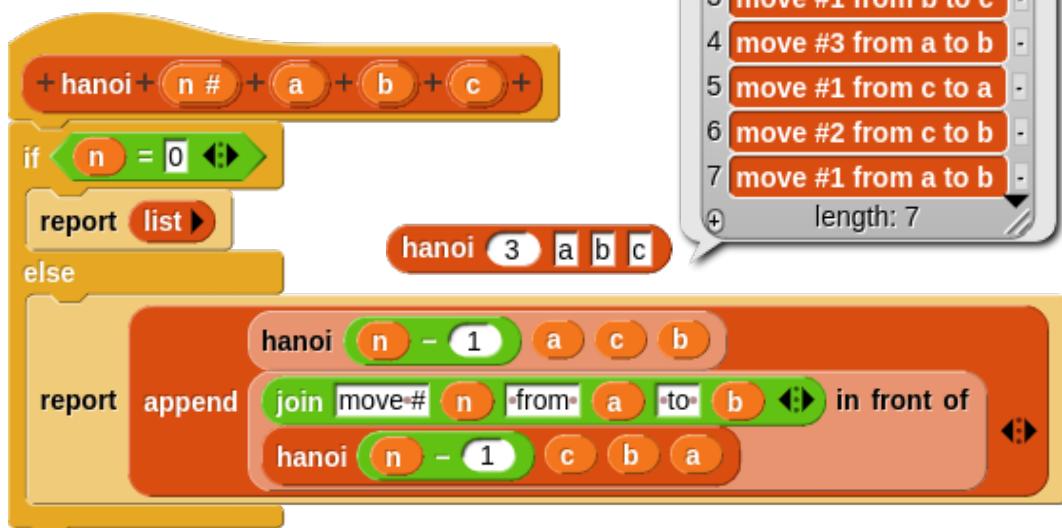
9.1.1 階乗

factorial 階乗は再帰呼び出しの例としてよく使用されます。



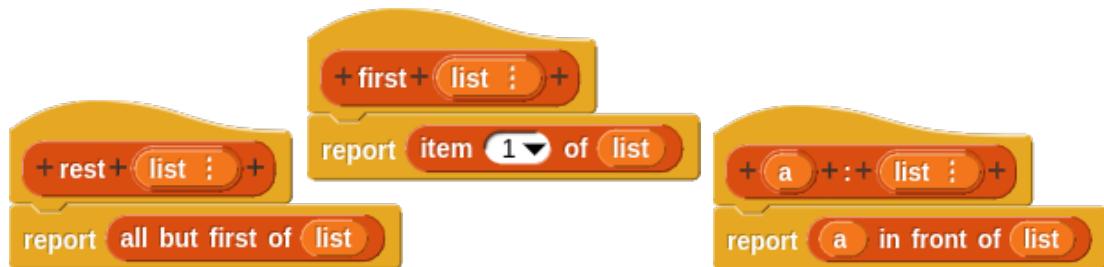
9.1.2 ハノイの塔

ハノイの塔のパズルも再帰呼び出しの例としてよく使用されます。



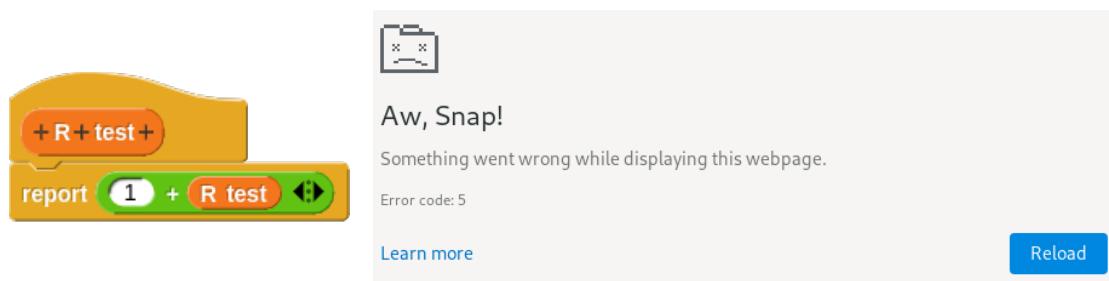
9.2 再帰呼び出しの使用

スクリプトの表示面積を小さくするために、3つのブロックを定義して使用します。



9.2.1 繰り返し

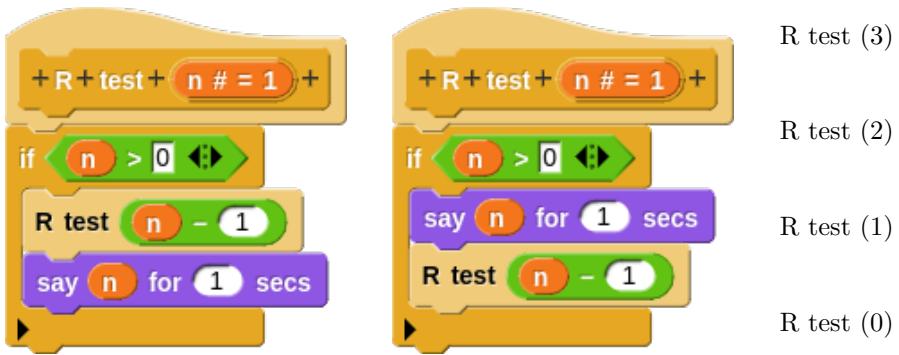
これはただ自分自身を呼び出すものです。(エラーになるので実行しないでください)



定義ブロックはどこかのスクリプトから呼び出されて実行されるわけですが、実行終了後に呼び出し元に帰る必要があります。呼び出し元のスクリプトの実行を継続するための情報を保存しておき、それを取り出してそこに復帰します。上記のスクリプトでは自分自身への無限呼び出しになり、情報を保存するための場所がなくなってしまいます。ただの無限ループというだけの問題ではありません。したがって、再帰呼び出しでは再帰呼び出しを終了するためのスクリプトが必要です。指定の回数繰り返す処理ならば、回数が 0 が終了条件です。リスト操作では、空リストが終了条件になります。リスト操作で Reporter 型のブロックを作成する場合、数値型では 0 を、リスト型では空リストを、Predicate 型のブロックを作成する場合は false を原則としてリポートして再帰を終了させます。空リストの場合は、list でも同じことです。

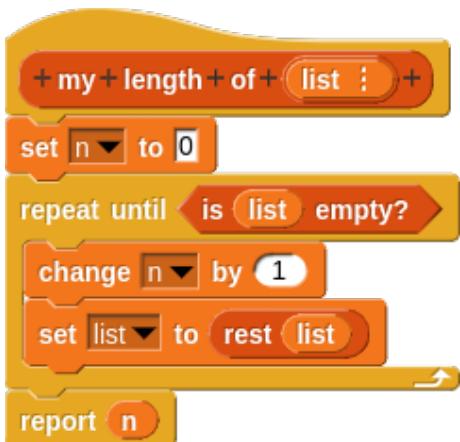


次は指定した回数だけ何かを行う定義ブロックです。n = 0 が終了条件になります。n の値を減らしながら以下の矢印(→)のように自分自身を呼び出していきます。この定義ブロックは 0 以下だと何もしないで呼び出し元に戻ります。呼び出し元に戻るを繰り返し、一番最初の呼び出し元に戻ったら終了です。再帰呼び出しブロックの位置により n の値はカウントアップにもカウントダウンにもなります。

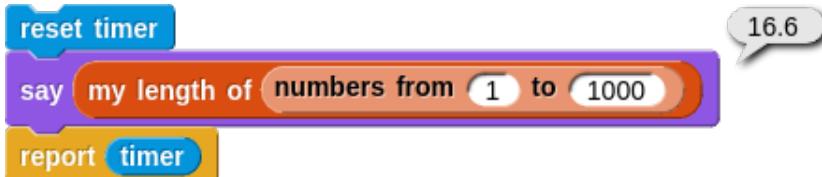


9.2.2 my length

リストの要素数を求める length ブロックを repeat until ブロックを使って作ってみます。要素数が 0 になるまで要素を一つずつ削除しながらカウントすることで求めます。

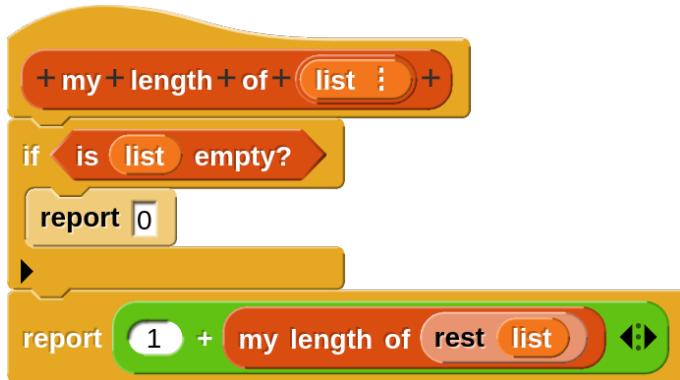


処理にかかる時間を表示します。



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

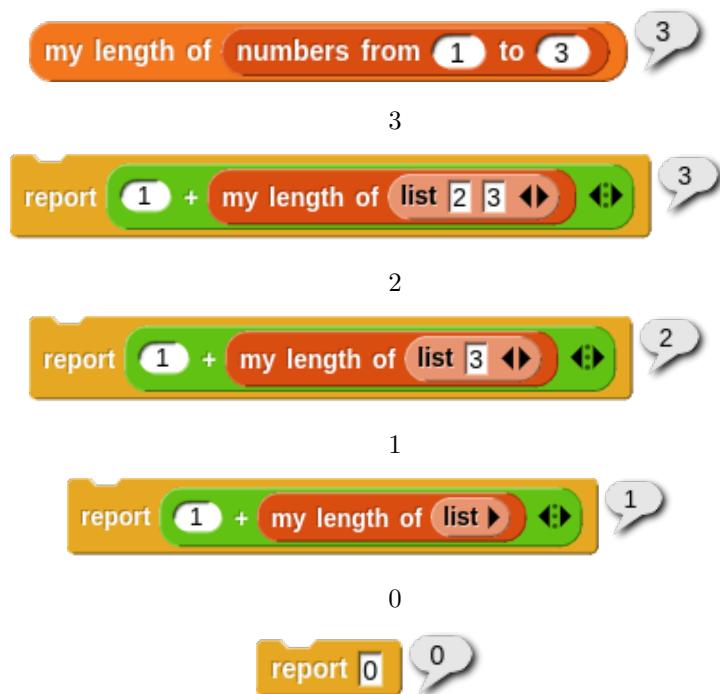
[リストの先頭要素を処理する。2 番目以降のリストを引数として自分自身を呼び出す。] ということをリストが空になるか条件が成立するまで、処理を行うのが再帰処理の基本です。この場合の処理は、リストの要素の値は使用せずにリポートされた値に 1 を加えているだけですが。



処理にかかる時間を表示します。

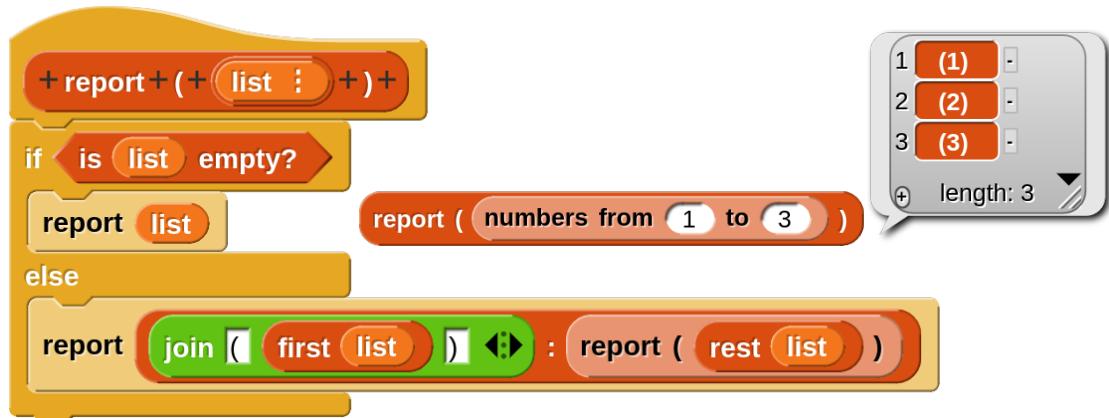


実行される様子を見てみます。　表示順に再帰呼び出しが実行され、0 と値が確定すると、表示順に返された値に 1 を加えて呼び出し元に値を返していきます。最終的に値は 3 になります。



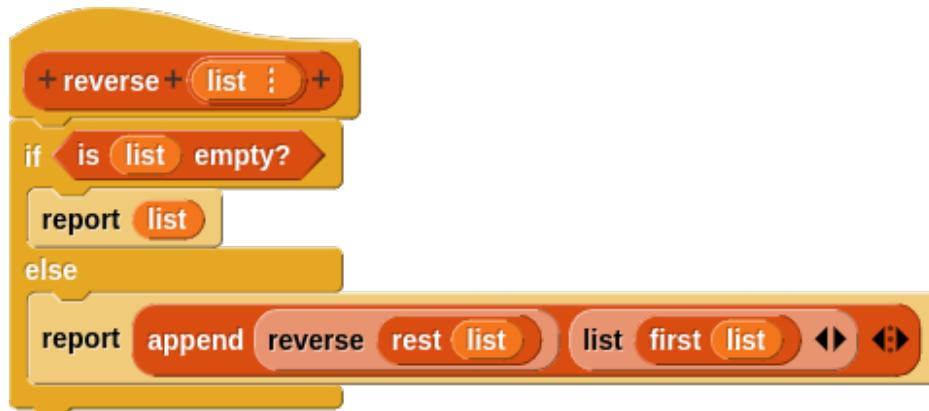
9.2.3 リストをリポート

リストの先頭の要素に対して操作し、その残りのリストに対して再帰呼び出しをした結果を加えることをリストが空になるまで行います。mapを行ったような結果になります。



リストの先頭の要素から操作したもの順に加えているのでこのようになりました。

残りのリスト操作の結果に先頭の要素を加えるようにすると、逆順リストが得られます。



appendはリスト同士を一つにするものなので、先頭の要素をlist [] :: 中に入れてリストにしないとエラーになります。

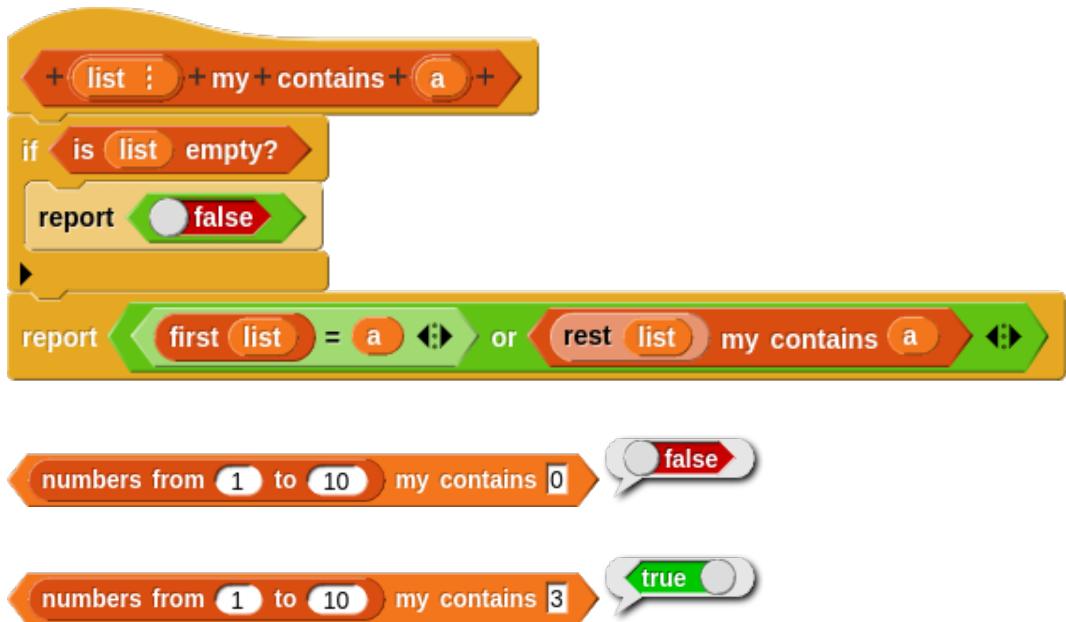


| | | | | |
|-----------------|------|---|-----|--------------|
| reverse (1 2 3) | [L2] | + | (1) | L3 = (3 2 1) |
| reverse (2 3) | [L1] | + | (2) | L2 = (3 2) |
| reverse (3) | [L0] | + | (3) | L1 = (3) |
| reverse () | | | () | L0 = () |

9.2.4 my contains

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

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



9.2.5 unique

unique of **目** という、リストから重複を取り除く機能を再帰呼び出しで定義してみます。



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

因みに、 を にして、 を引数として実行すると 100 までの素数列が求められます。(25 ページ参照)

9.2.6 リスト要素の巡回

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



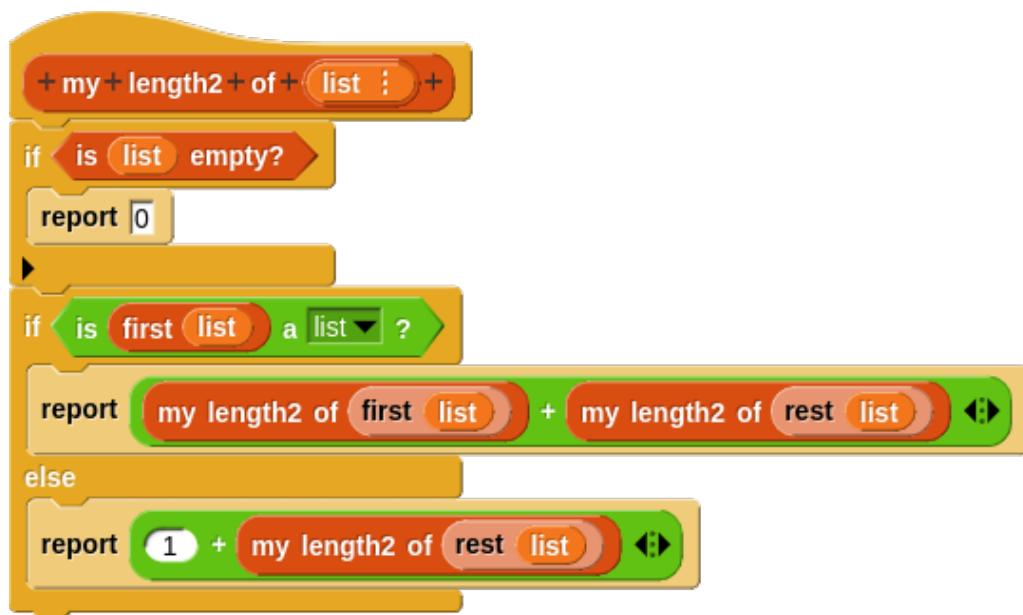
これは my length も同様です。



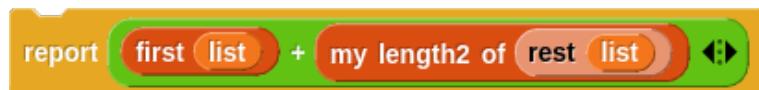
再帰を使って内部の要素に対してもアクセスしてみます。

処理の内容は次のようにになります。

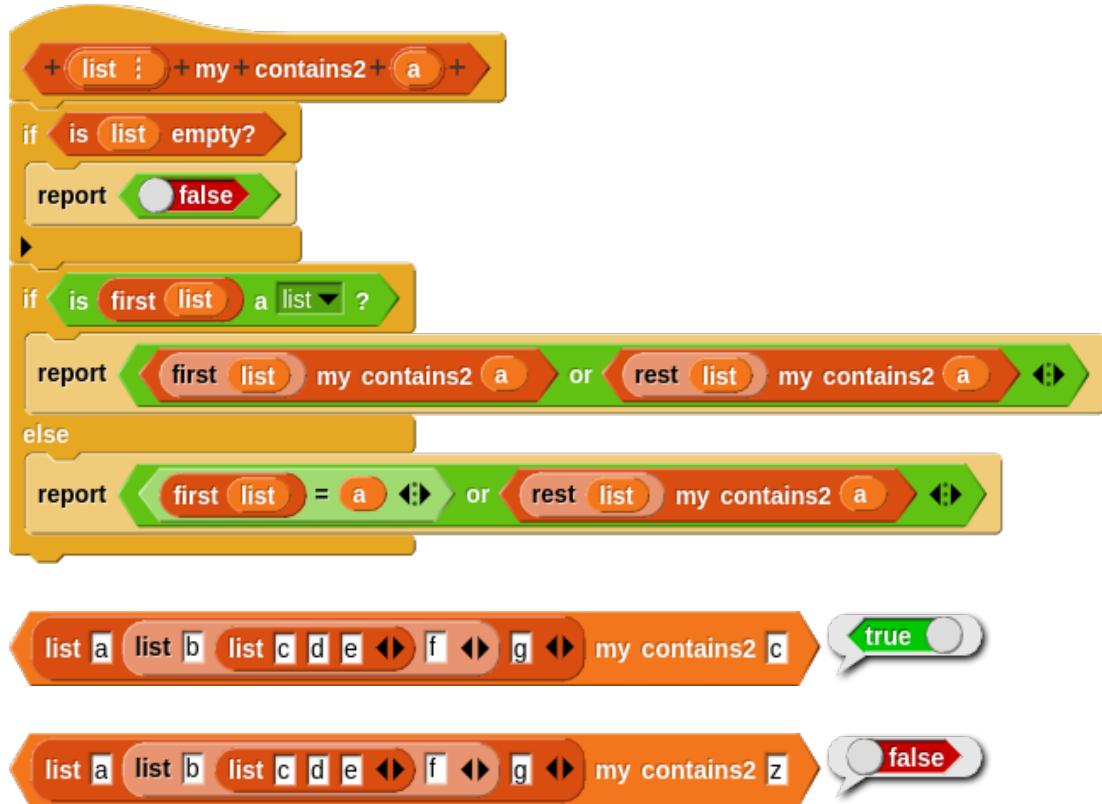
- もしリストが空ならば 0 をリポートする。
- もし先頭の要素がリストならば、そのリストに my length2 をしたものと残りに対して my length2 をしたものを加える。
- そうじゃなかったら、残りに対して my length2 をしたものに 1 を加える。



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



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



「and」ブロックでは左側のスロットから順にテストして、false ならば残りのスロットのテストは行いません。それに対して、「or」ブロックでは左側のスロットから順にテストして、true ならば残りのスロットのテストは行いません。



そのため、リストの先頭からテストしていくて、指定の要素が見つかればリストの残りはテストせずにそこで true を返して終了になります。



9.2.7 指定の要素に対する delete と replace

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

| 3 | A | B |
|---|---|---|
| 1 | 3 | |
| 2 | 1 | 2 |
| 3 | 9 | |

```
keep items not [ = 2 ] from [ list 2 3 list 1 2 [ 9 2 ] ]
```

my length2 を応用した再帰版 delete です。

```
+ delete + [ a ] + of + [ list : ] +
if [ is [ list ] empty? ]
  report [ list ]
else if [ is [ first [ list ] a [ list ] ] ? ]
  report [ delete [ a ] of [ first [ list ] ] : delete [ a ] of [ rest [ list ] ] ]
else if [ first [ list ] = [ a ] ] [ ]
  report [ delete [ a ] of [ rest [ list ] ] ]
else if [ checked? ]
  report [ first [ list ] : delete [ a ] of [ rest [ list ] ] ]
end
```

| | | |
|---|---|-----------|
| 1 | 3 | - |
| 1 | 1 | - |
| 2 | | |
| 3 | | length: 1 |
| 3 | 9 | - |
| + | | length: 3 |

```
delete [ 2 ] of [ list 2 3 list 1 2 [ 9 2 ] ]
```

リストを含まないリストの要素の入れ替えは map を使えばできますが、内部のリストまで 2 を 7 に入れ替えることはできません。

| | A | B |
|---|---|---|
| 1 | 7 | |
| 2 | 3 | |
| 3 | 1 | 2 |
| 4 | 9 | |
| 5 | 7 | |

```
map if [ = 2 ] then [ 7 ] else [ ] over [ list [ 2 3 ] list [ 1 2 ] [ 9 2 ] ]
```

指定の要素だった場合、それをコピーしないでリストの残りを続行すれば delete になりますが、置き換える要素をコピーすれば replace になります。つまり、次の下から二番目の report ブロックで new をくっつけるかどうかで replace か delete になります。

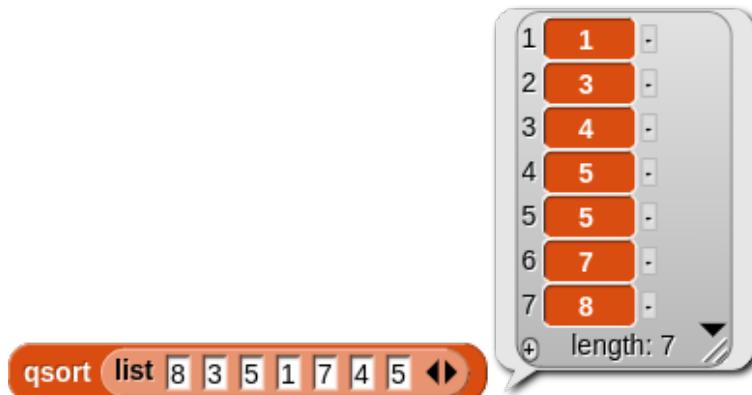
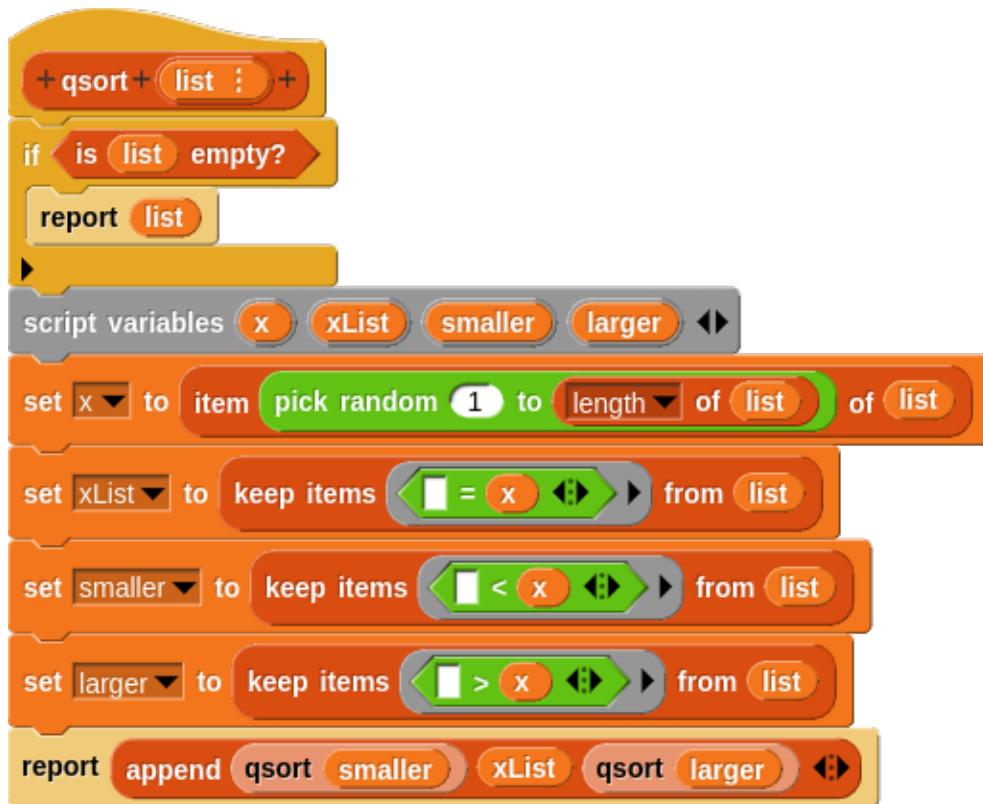
```
+ replace + [ old ] + with + [ new ] + of + [ list : ] +
if [ is [ list ] empty? ]
report [ list ]
else if [ is [ first [ list ] ] a [ list ]? ]
report
  [ replace [ old ] with [ new ] of [ first [ list ] ] : replace [ old ] with [ new ] of [ rest [ list ] ] ]
else if [ first [ list ] = [ old ] ]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

| | A | B |
|---|---|---|
| 1 | 7 | |
| 2 | 3 | |
| 3 | 1 | 7 |
| 4 | 9 | |
| 5 | 7 | |

```
replace [ 2 ] with [ 7 ] of [ list [ 2 3 ] list [ 1 2 ] [ 9 2 ] ]
```

9.2.8 クイックソート（整列 / 並べ替え）

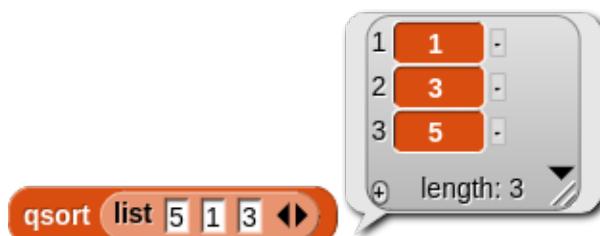
クイックソートのアルゴリズムは有名でよく題材として扱われています。リストの中から任意の値を選び、それよりも小さい値のグループ、その値、大きい値のグループに振り分ければ選択された値の位置付けができます。この操作を小さい値のグループ、大きい値のグループに対して再帰的に繰り返していくば最終的に並べ替えが完了します。任意の値の選び方として random ブロックを使いましたが、先頭の値でも構いません。Snap! には keep ブロックがあるので、リスト要素の入れ替えなどに煩わされず、アルゴリズムをそのまま表したように割と分かりやすいスクリプトが作れます。



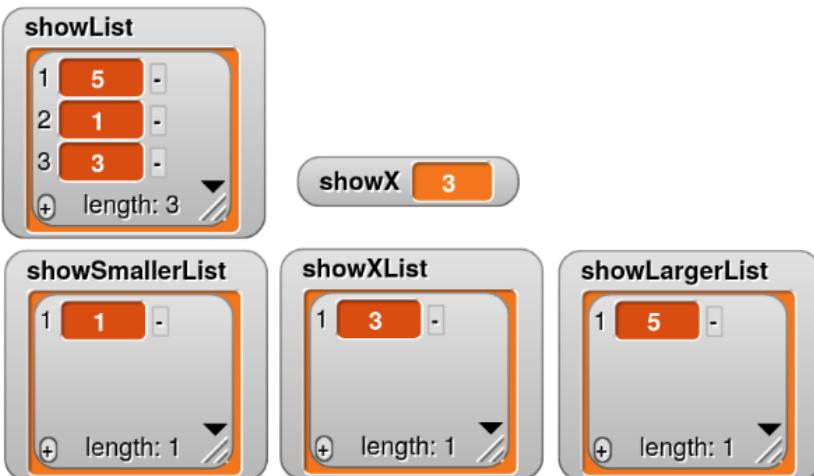
次のように showList, showX, showXList, showSmallerList, showLargerList のグローバル変数を作成し、リストを通して値を表示すると操作の様子が見られます。



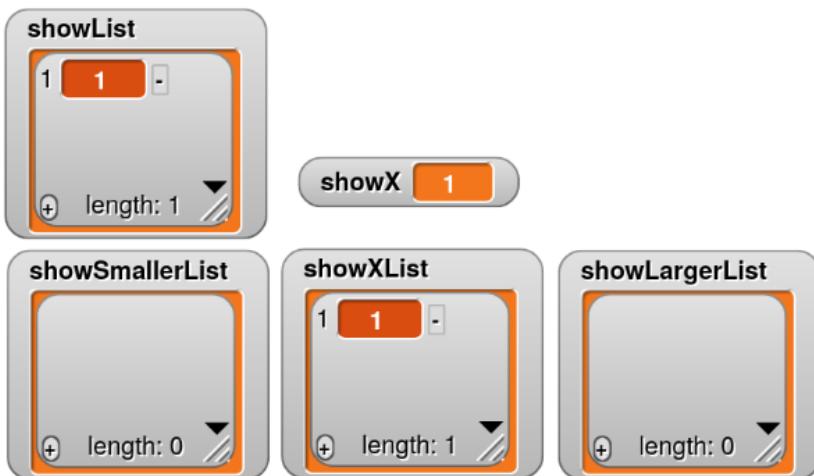
リストの値表示のたびに pause all になります。 をクリックして続行してください。



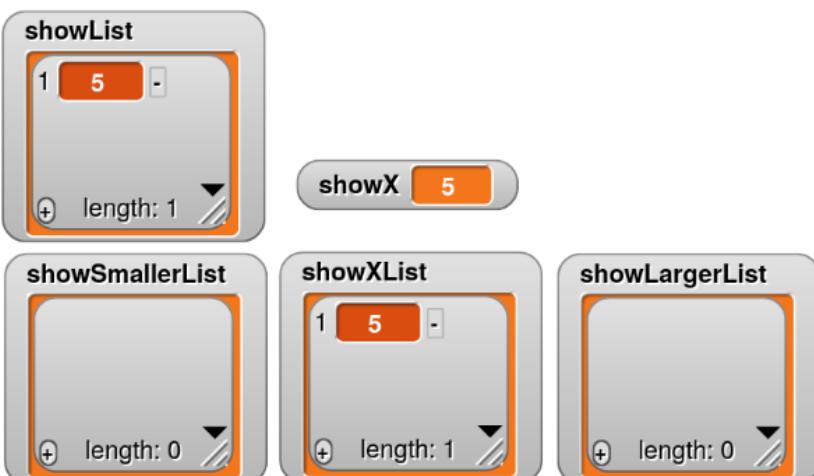
最初に選択された値が 3 だった場合です。



3 より小さい値のグループ処理



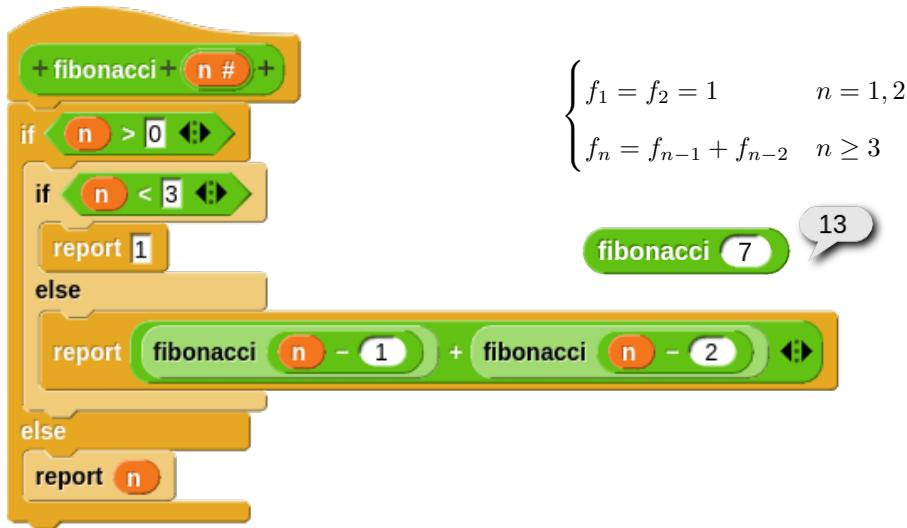
3 より大きい値のグループ処理



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

9.2.9 フィボナッチ数列

再帰の例としてよく示されるフィボナッチ数列です。

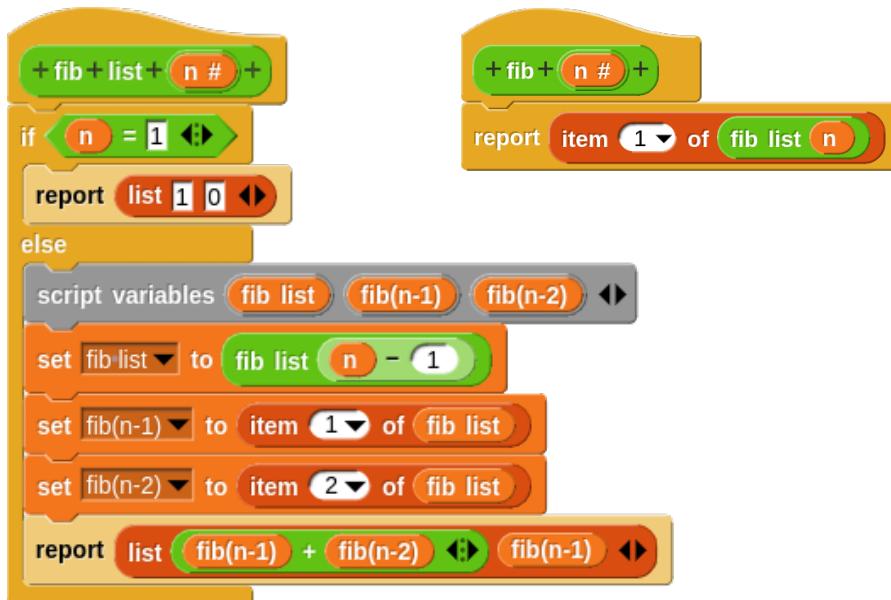


これは $(n - 1)$ と $(n - 2)$ を引数にして 2 度再帰呼び出ししています。そのため、 n が大きくなると時間がかかるてしまいます。

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

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

$\text{fib}(n)$ と $\text{fib}(n-1)$ の値をリストにしてリポートするブロックを使用するバージョンです。再帰呼び出しをする $\text{fib list}()$ がそれを行う定義ブロックです。



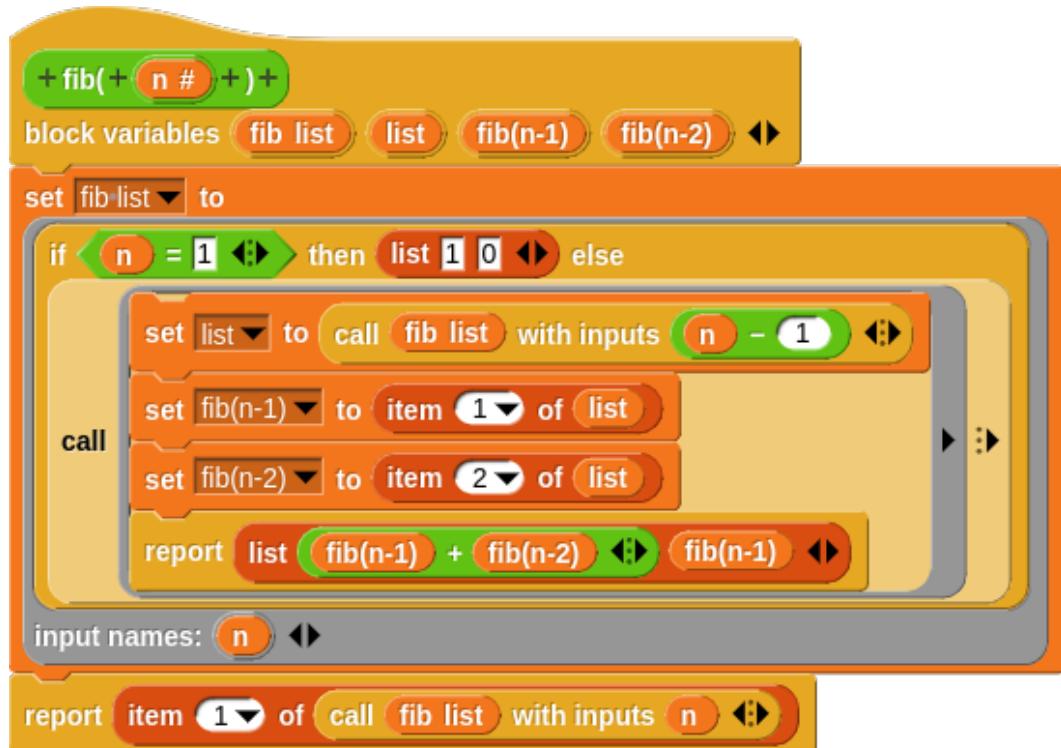
右の表は n の値に対する fib と fib list の値です。(は前の項のという意味です。) fib の値は、fib list がリポートする値であるリストの 1 番目の値になります。その 2 番目の値は $\text{fib}(n - 1)$ の値です。これによって処理済みの fib 値を渡していきます。

| n | fib | fib list |
|---|-----|------------------------|
| 1 | 1 | list(1, 0) |
| 2 | 1 | list(1, 1) (1 + 0, 1) |
| 3 | 2 | list(2, 1) (1 + 1, 1) |
| 4 | 3 | list(3, 2) (2 + 1, 2) |
| 5 | 5 | list(5, 3) (3 + 2, 3) |
| 6 | 8 | list(8, 5) (5 + 3, 5) |

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



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



[参考文献]

『プログラミング in OCaml

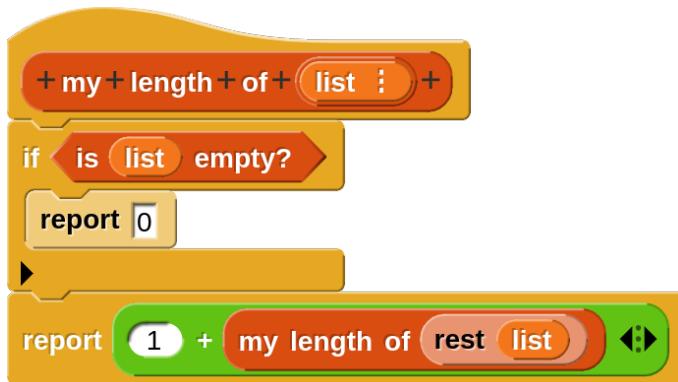
～関数型プログラミングの基礎から GUI 構築まで～』

五十嵐淳 著

技術評論社 刊

9.2.10 末尾再帰

116 ページで my length を扱いました。



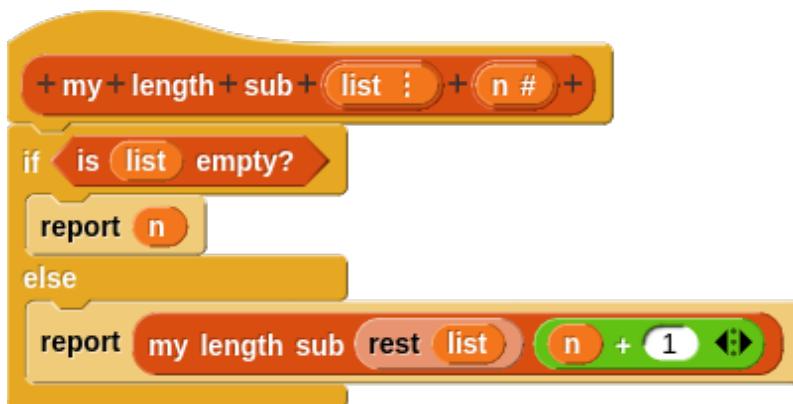
この定義ブロックでは `report [1 + my length of rest list]` で、計算式に再帰呼出しが含まれていて、再帰呼出しによる値が確定しないと計算することができません。リストが空になると確定した 0 の値を使って順々に計算した値を戻しながら一番最初のところまで戻って、ようやく値 3 を得ることになります。my length of (1 2 3) を L(1 2 3) と表してみます。

$L(1 \ 2 \ 3)$

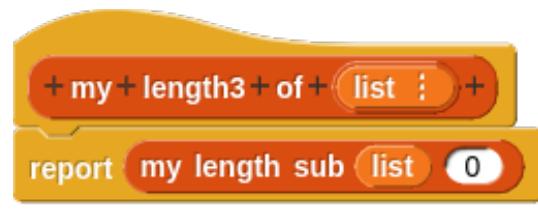
$$\begin{aligned}
& 1 + L(2 \ 3) \\
& 1 + L(3) \\
& 1 + L() \\
& 1 + 0 \\
& 1 + 1 \\
& 1 + 2 \\
& 3
\end{aligned}$$

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

引数 ($n + 1$) を渡していくことでカウントしています。



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



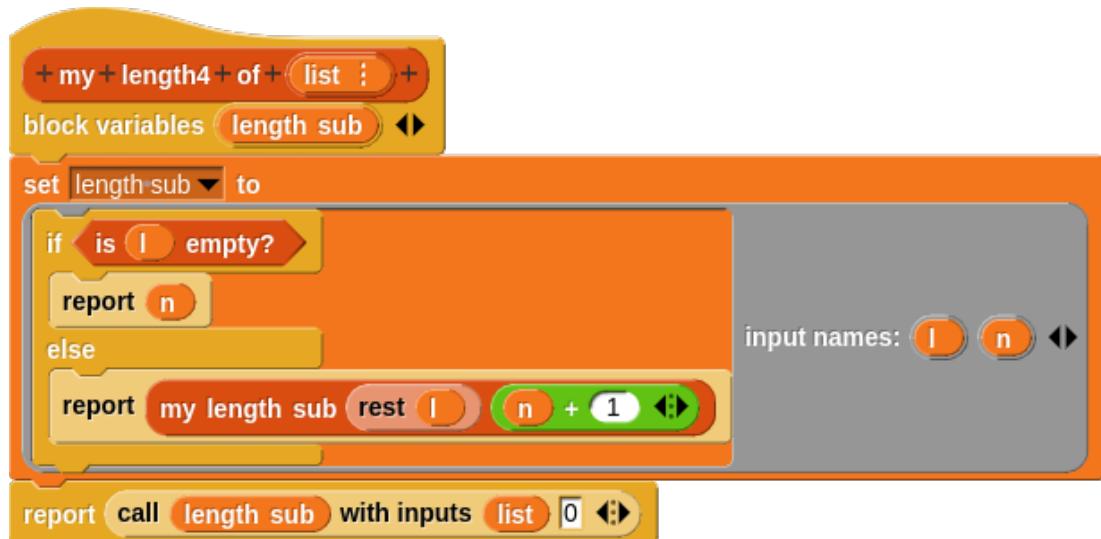
my length3 of list 1 2 3 3 となります。

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

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

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

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

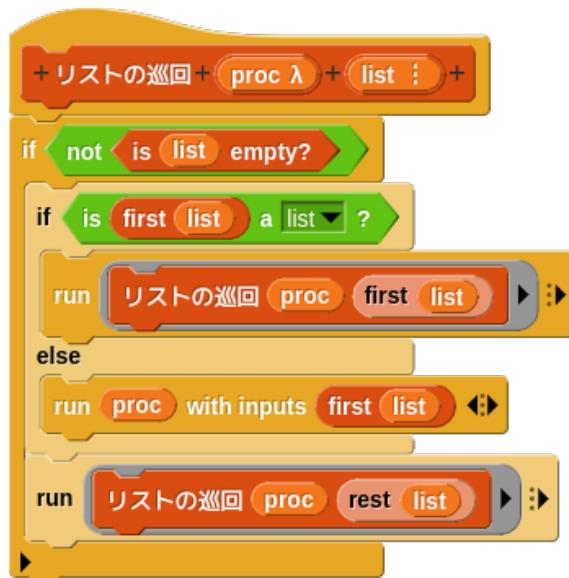


10 高階関数

関数の引数や戻り値に関数を指定できるものを高階関数と言います。リスト操作に使用する map, keep, find, combine ブロックは入力スロットにリングがあり、引数にスクリプトブロックを指定する高階関数型のブロックです。Snap! のユーザー定義ブロックでは高階関数型のものを作成することができます。入力スロットでスクリプトブロックを受け取ることができますし、スクリプトブロックを ringify リングで囲ってやればそのスクリプトブロック自体を戻り値としてリポートすることができます。

10.1 操作を指定するリストの巡回

120 ページで「リスト要素の巡回」を扱いました。引数で操作を指定すれば同じ定義ブロックでいろいろな用途に使えます。定義ブロック自体は Command 型で、引数 proc は Command(inline) 型です。proc は引数を一つ使用します。



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



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



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

リポーター版です。引数 proc は Reporter 型です。



command 版と同じようにできますが、Reporter 型なので report を入れています。
空きのスロットにはリストの要素が入ります。



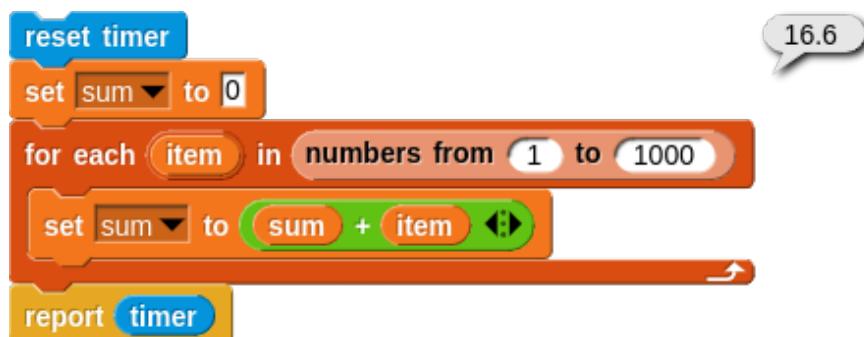
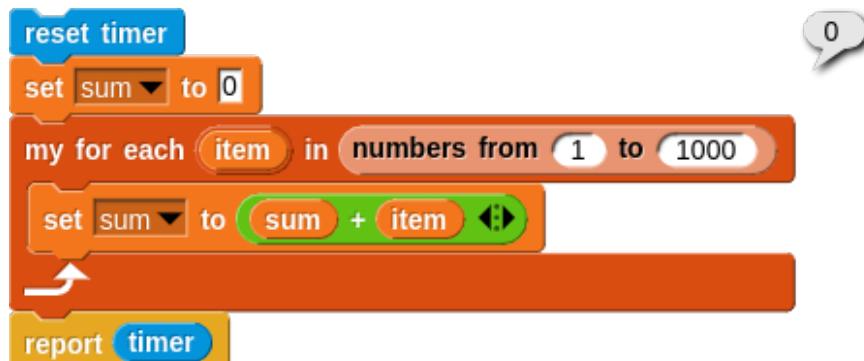
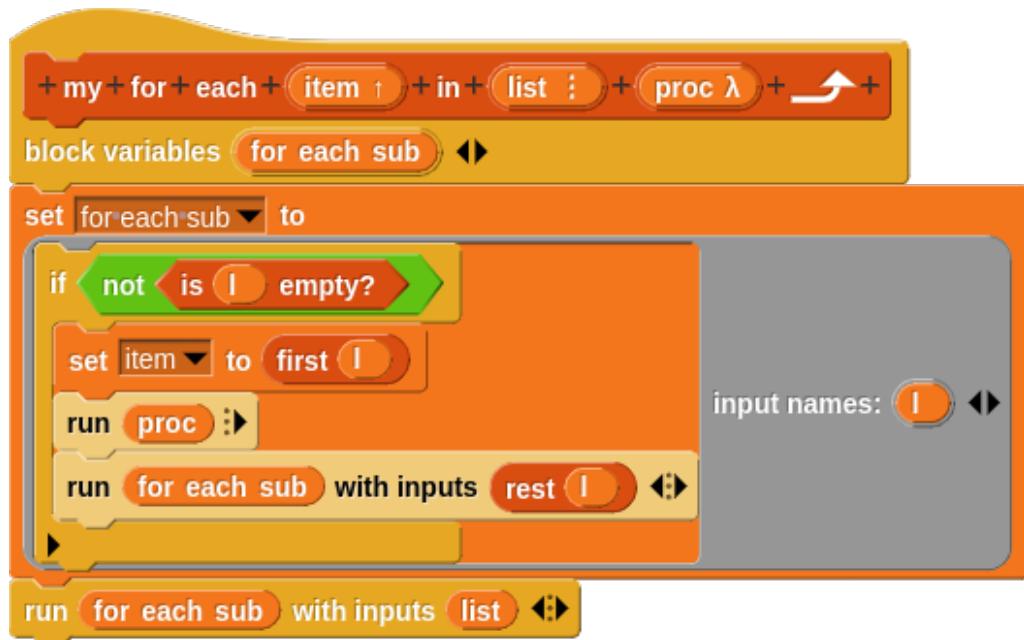
10.2 my for each (item) in (list)



→ を再帰処理で行うと高速にできます。

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

proc は Command (c-shape) で作成します。



10.3 foldl, foldr

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

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

foldl から見ていきます。



操作内容は、
0 + 1 + 2 + 3 のようになります。

フォーマルパラメータを使用すると次のようにになります。



フォーマルパラメータの順序は右下の表のようになります。(#1, #2 から名前の変更をしています)

The Scratch script shows the internal logic of foldl. It starts with a + foldl + proc λ + init + list : + sequence, followed by an if is list empty? branch. If true, it reports init. If false, it enters a loop where it reports foldl proc call proc with inputs init first list :: rest list. To the right, a table shows the state of variables over time:

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

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



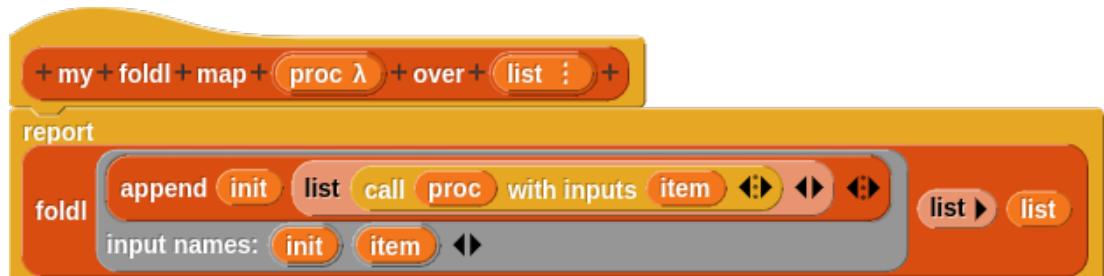
次のようにすると最大値を求めることができます。



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



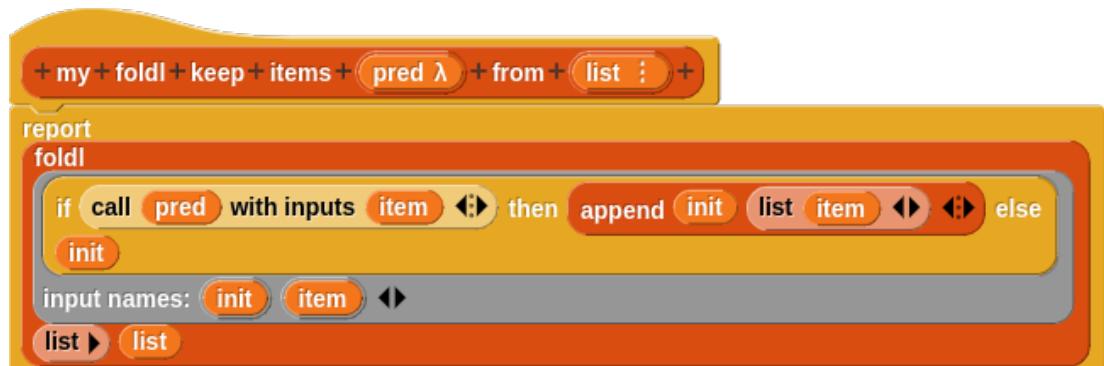
foldl 版です。



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



foldl 版です。



foldr です。

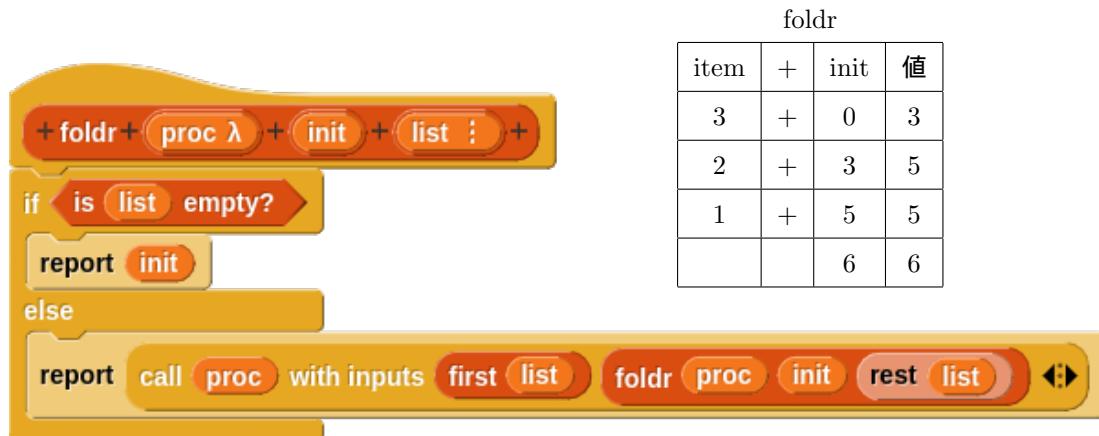


操作内容は、
のようになります。

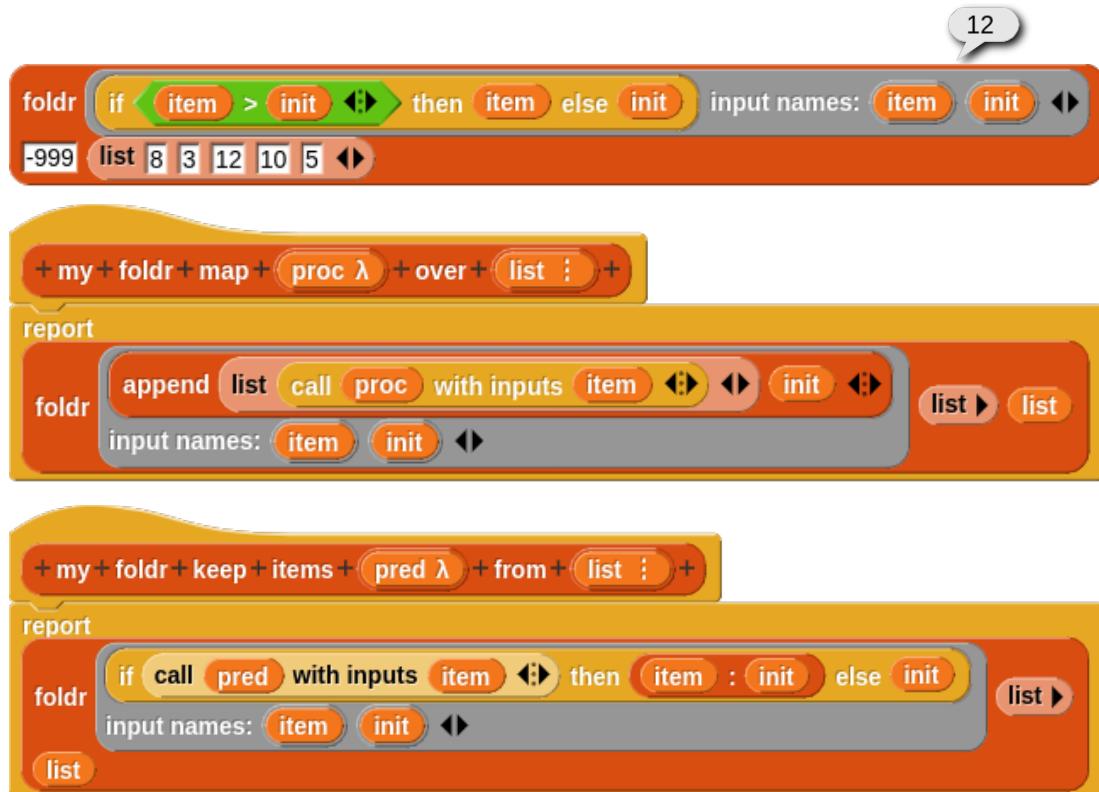
フォーマルパラメータを使用すると次のようにになります。



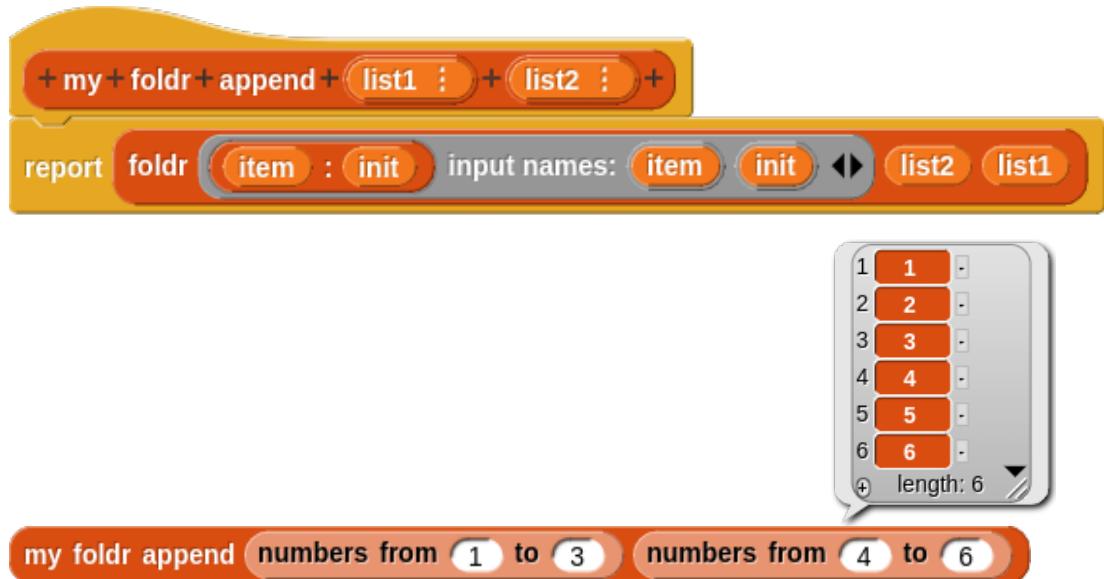
フォーマルパラメータの順序は右下の表のようになります。foldl とは違う item, init の順になります。



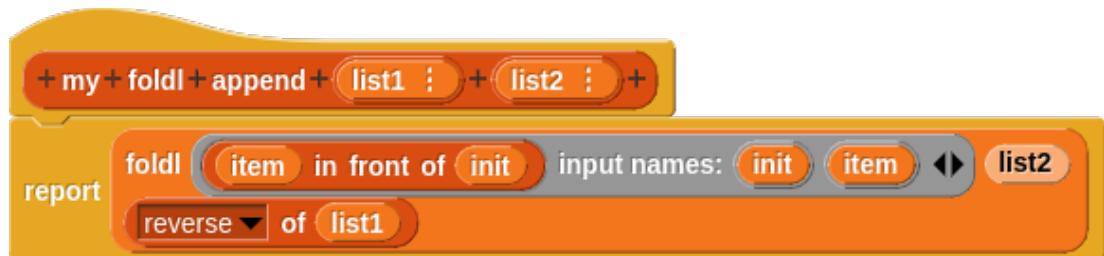
foldr を使っても foldl と同じようなことができます。



foldr がリストを右側から操作することを利用した append です。



foldl で作成する場合には、リストを逆順にする操作が必要になります。



再帰呼出しを使ってリスト操作をすれば効率の良いプログラムになりますが、再帰を意識したプログラム作成はなかなかたいへんです。fold を使用してなんでも作成できるわけではありませんが、自動的に再帰呼出しを使った処理してくれます。Rust 言語にもイテレータに fold メソッドが用意されているように有用な機能です。

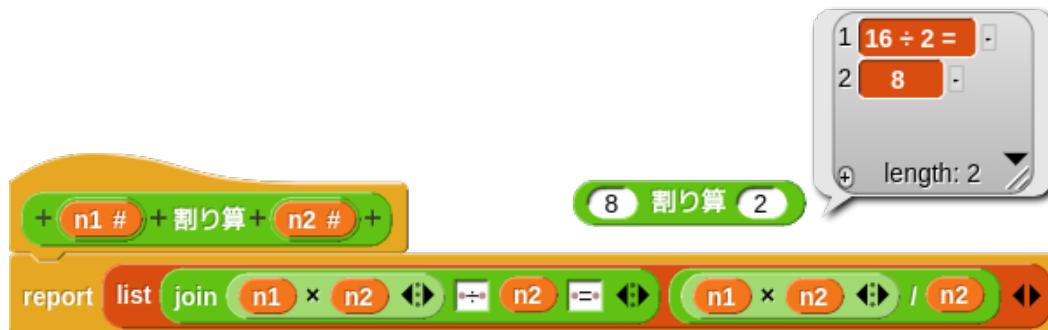
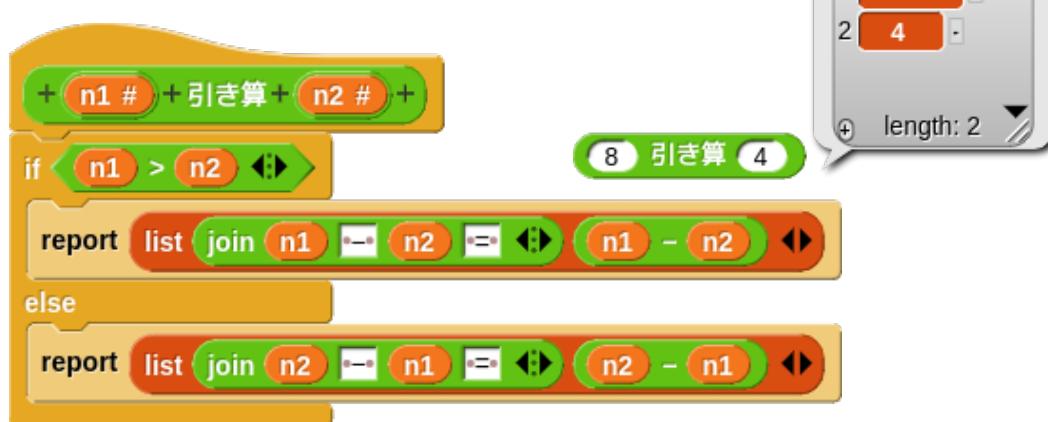
10.4 関数を返す関数

ブロックが ringify されたブロックをリポートする例を示してみます。高階関数で言うところの関数を返す関数ということです。仕組みを説明するためのものなのであまり意味はありませんが。

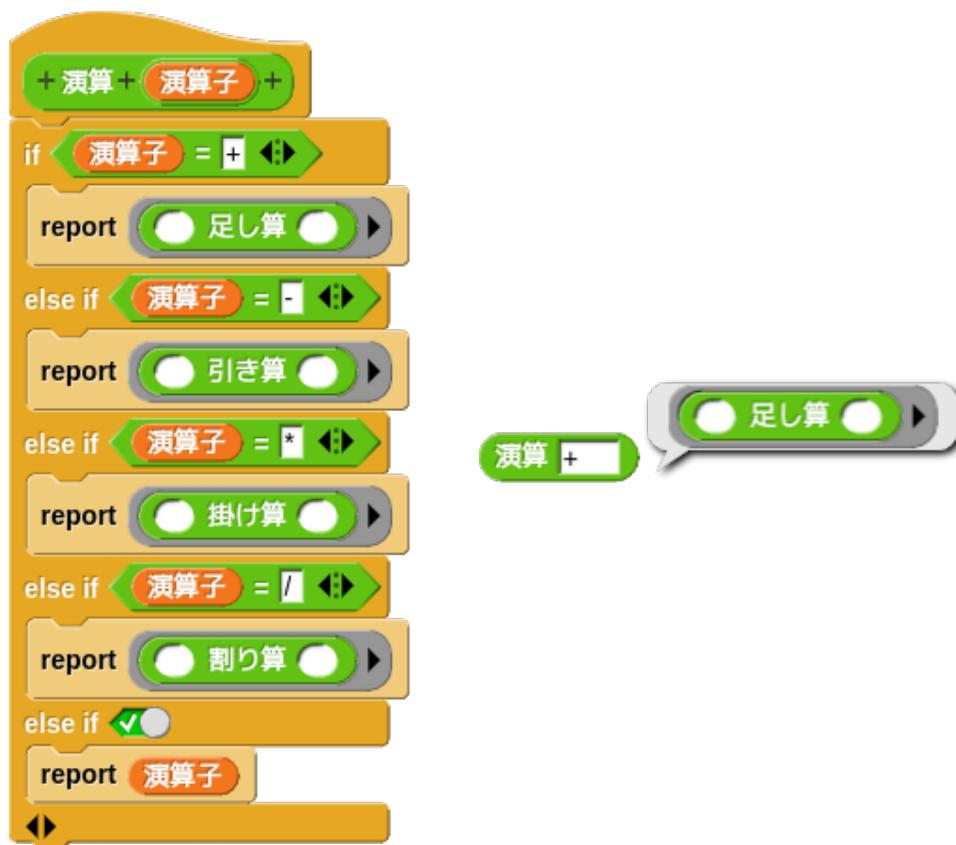
まずは基本となる 2 つの引数を取り、足し算の式と答えをリポートするものです。



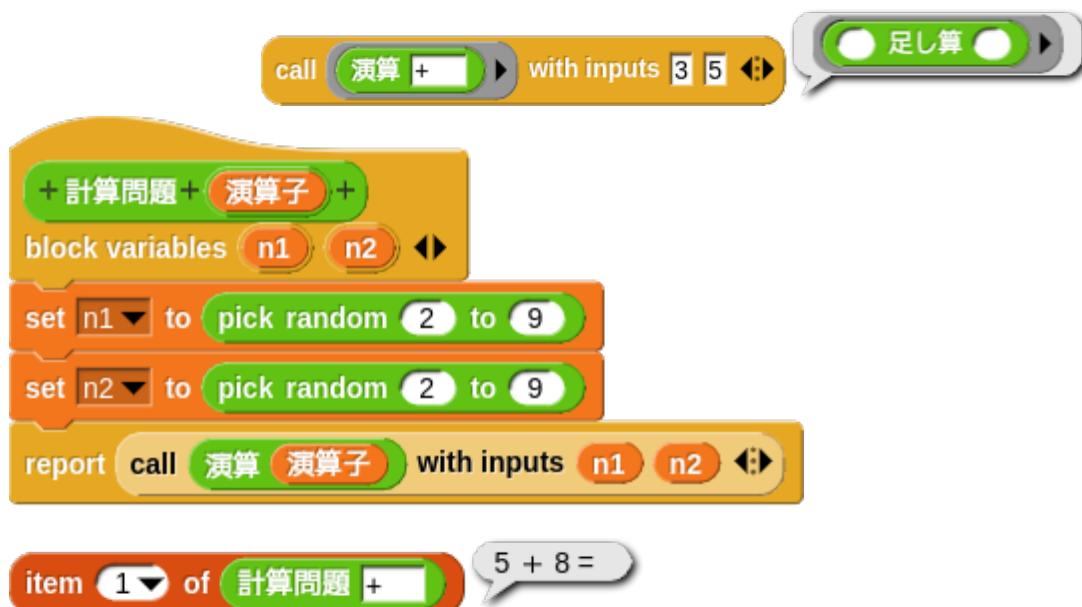
同じように引き算、掛け算、割り算のブロックです。引き算と割り算には引数の扱いに工夫が必要です。



指定された演算子によってこれらのブロック自体を返す定義ブロックです。不明な演算子はそのまま返します。「演算」のブロックが関数を返す関数にあたります。



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



10.5 カリー化

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

Haskell などでは「関数は一つしか引数を取らない」という言語仕様上重要な手法のようです。Snap! では入力スロットを増やして複数個の入力を受け付けることができたり、リストを受け取るのであまり必要ないですが、高階関数の例として示してみます。

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



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



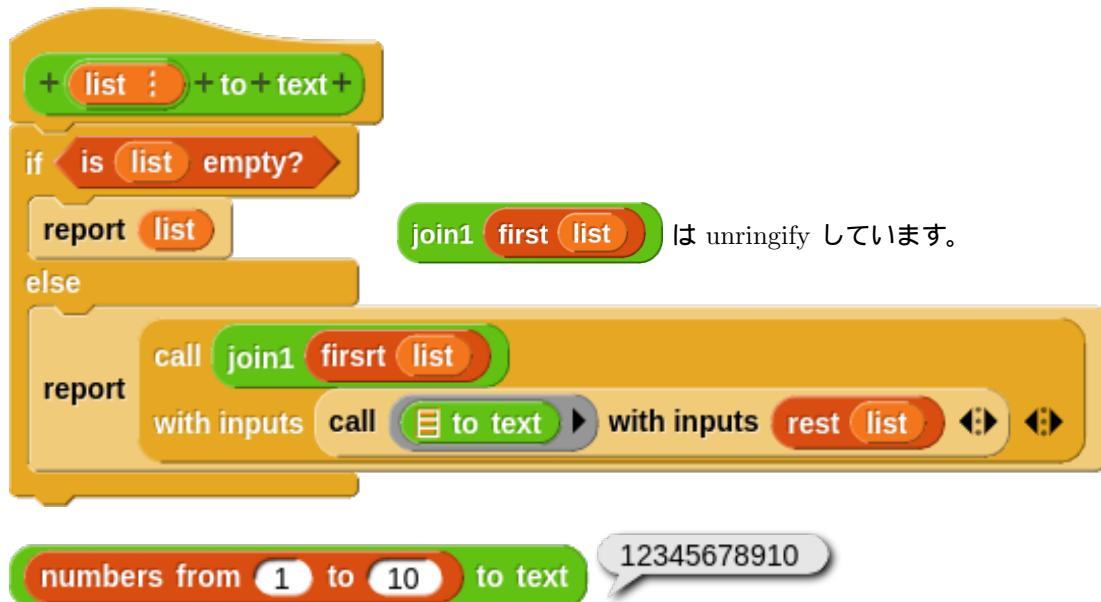
スクリプトブロックを実行するには `call` を使いますが、このままリング付きの `call` で実行するとブロック自体が表示されるだけになってしまいます。`join1` がリング付きのスクリプトブロックを返すものだからです。



`call` に入力した `join1` ブロックを `unringify` してください。



`join1` を使ってリストの要素を文字列にしてみます。



10.6 Class のような使い方

他のプログラミング言語では、Class という型を作り、変数や操作（メソッド）を設定します。その設計図のようなものを基に作られた実態にメッセージという形で指令を送って結果を得ます。Snap! には表立って Class のような仕組みはありませんが、高階関数を使うことで似たようなことをすることができます。

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

預貯金通帳を定義ブロックで作ってみます。

預貯金額の変数や通帳で行う操作のスクリプトを通帳自身に持たせます。

通帳を最初に作ると、残高用の変数を作成して、残高をリポートする操作を設定するスクリプトです。

(Default Value: 0 として、作成時の入金も可にします。)
この型を基に以下のようにして通帳を作ります。

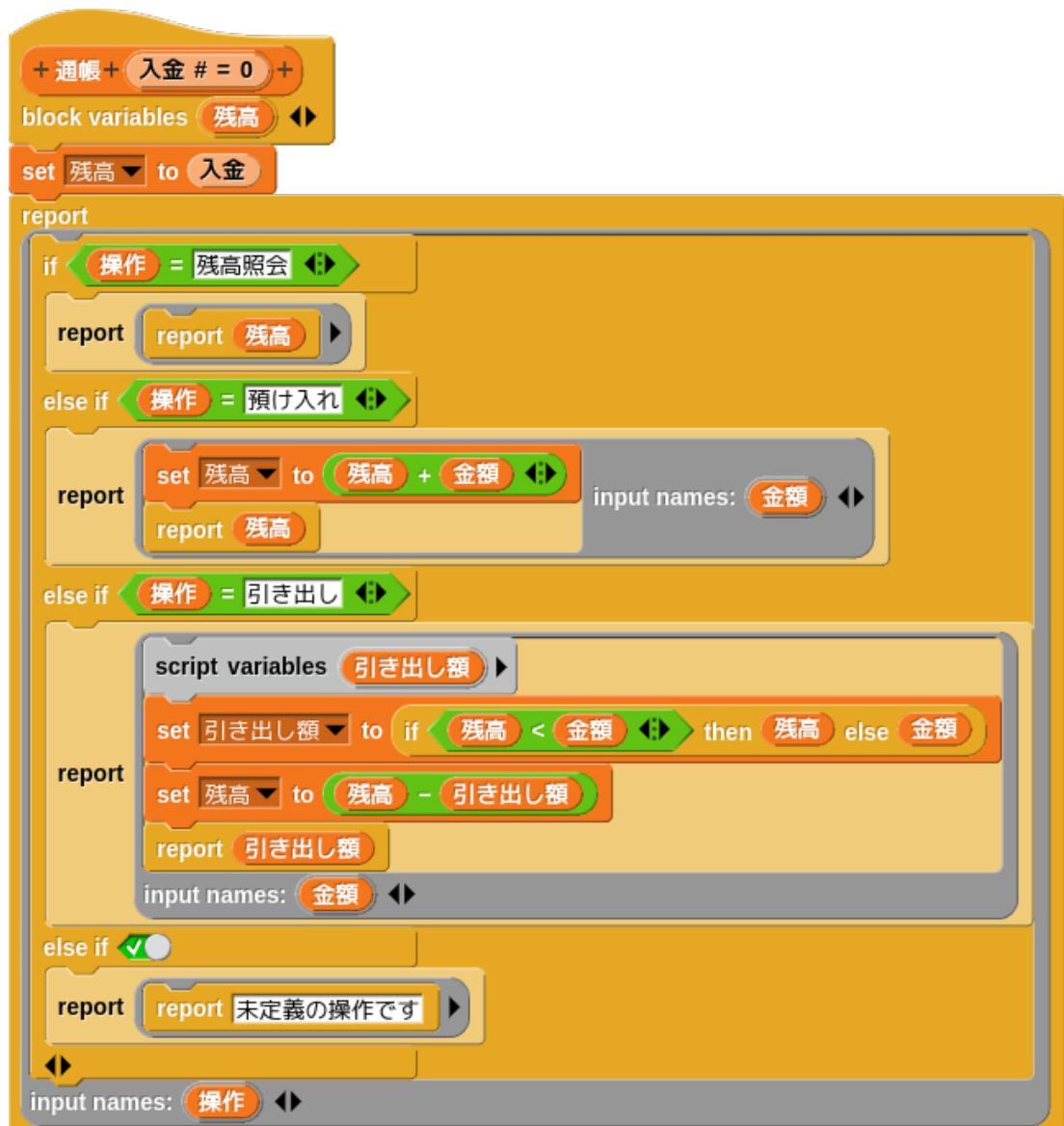


「太郎の通帳」を表示させてみます。変数「残高」は「太郎の通帳」が削除されるまで有効です。ローカル変数ですので「花子の通帳」の変数「残高」とは別物です。



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

通帳への操作として「残高照会」「預け入れ」「引き出し」ができるようにしてみます。「預け入れ」は引数として指定された金額を残高に加算し残高を返します。「引き出し」の場合は、引き出し額が残高を上回る時は残高を引き出し額とし、引き出せた金額を返します。借金はできません。負数の入金には対処していません。



定義を変更したので set ブロックの実行が必要です。

set 太郎の通帳 ▾ to 通帳 0

「残高照会」「預け入れ」「引き出し」が「操作」のリクエストであるメッセージになります。

金額を指定するために call が二段構えになっています。内側の call ブロックは unringify してください。



call call 太郎の通帳 with inputs 預け入れ <::> with inputs 10000 <::> 10000

call call 太郎の通帳 with inputs 引き出し <::> with inputs 1000 <::> 1000

call call 太郎の通帳 with inputs 残高照会 <::> 9000

使いやすいように ATM ブロックを定義します。「通帳名」は Object、「操作」は Text, options... と read-only を選択しています。(63 ページ参照)



残高照会の場合の金額入力は参照しません。

ATM 太郎の通帳 残高照会 <::> 9000

ATM 太郎の通帳 預け入れ <::> 10000 19000

ATM 太郎の通帳 引き出し <::> 3000 3000

ATM 太郎の通帳 残高照会 <::> 16000

set 花子の通帳 <::> to 通帳 10000

ATM 花子の通帳 引き出し <::> 50000 10000

ATM 花子の通帳 預け入れ <::> 20000 20000

ATM 花子の通帳 残高照会 <::> 20000

索引

Σ of (), 31
: プロック, 115
all, 80, 82
all but first of, 20
all<, 82
all=, 82
all>, 82
any, 82
atan, 91
atan2, 91
block variables, 47, 62
call, 39
case, 76
columns, 107
columns of (), 29
combine, 26, 97, 99
composition, 50
Costumes, 9
csv, 6, 13, 34
csv of (), 32
delete, 122
dimensions of (), 28
distribution of (), 30
draggable?, 9
factorial, 114
find first item, 26, 81
first プロック, 115
flatten of (), 29
identical, 79
import, 9, 34, 50
inner product, 105
input list:, 39
Iteration, 50
JavaScript, 94
JavaScript extensions, 5
join, 27
json, 6, 13, 34
json of (), 32
keep, 25, 119, 122, 124, 135
Language, 5
letter, 98
Libraries, 9
lines of (), 32
list view, 18
map, 23, 24, 80, 97, 123, 135
New, 9
object, 69, 71
Open, 9
outer product, 111
pen trails, 35
pipe, 45, 97
product, 82
rank of (), 28
relabel, 11
replace, 123
reshape, 22
rest プロック, 115
reverse of (), 31
ringify, 35, 54
rotation style, 8
run, 38
Save, 9
Save As, 9
scene, 9
script pic, 11
shuffled of (), 31
sorted of (), 30

split, 27
sum, 82
switch, 76

table view, 18
text of (), 31
transient, 14
turbo mode, 10

unique, 119
uniques of (), 30
unringify, 35
UTC, 99

with inputs, 38

Zoom, 5

オフライン版, 5

階乗, 26, 114
回転行列, 106
角度, 91
カスタムブロック, 7, 46
カリー化, 140

協定世界時, 99
行列の積, 105

クイックソート, 124
グローバル変数, 13, 14

高階関数, 131
再帰呼び出し, 114
座標変換, 106

辞書, 81
ジュークボックス, 7

スクリプトエリア, 7
スクリプト変数, 16
ステージエリア, 6
ステップ実行, 8, 49, 77
スプライトコラル, 7

スライト変数, 15
ゼブラカラーリング, 11

ソート, 95
素数, 25, 112, 119

ターボモード, 10
大域変数, 14
単位行列, 111

定義ブロックのインポート, 62
定義ブロックのエクスポート, 62
デバッグ, 8, 77
転置行列, 29, 107

時計, 99

二進数, 95
日本語化, 5

排他的論理和, 95
配列, 22, 94

バックグラウンド, 7
ハノイの塔, 114
パレットエリア, 7
八口, 12

ビット演算, 95
評価, 71, 73

フィボナッチ数列, 23, 127
フォーマルパラメータ, 23, 40

プリミティブブロック, 7
ブロック内変数, 47, 62
プロトタイプ, 47, 49, 67, 69

並列実行, 103
変数ウォッチャー, 6, 14, 18, 19, 34

末尾再帰, 129
無名関数, 40
ラムダ関数, 40

リスト要素の巡回, 120, 131

リング, 23, 35, 39, 43, 54, 70, 71, 73

ループ変数, 17

連想配列, 81

ローカル変数, 15

ロケーションピンアイコン, 16

論理積, 95

論理和, 95

ワードローブエリア, 7

ワープ, 10