ゲームを通してプログラミングの基礎を学ぼう（後編）
― Racket で学ぶプログラミング―

基般

対馬かなえ（国立情報学研究所）

Racket で始める楽しいゲームプログラミング

本稿は本誌 9 月号の記事の続きになります．ビットくんというキャラクタが車を避けながら家に帰る「ビットくん帰宅ゲーム」を完成させながら，プログラミングの基礎を学んでいきます．必要なものは，Windows または MacOSX がインストールされているパソコンとサンプルプログラム☆1，そしてプログラミングへの興味だけです．まだ Racket をインストールしていない場合には，9 月号の前編を参考にインストールしてください．

前編では，Racket プログラミングの始め方を解説し，プログラムのちょっとした変更で実行結果が変わることを体験し，ゲームの世界を作っている構造体というものを学んでいきました．

後編ではゲームを改良して，

- ビットくんが上下にも動けるようにする
- ビット君が車にぶつかったらゲームオーバーに，家に帰れたらゲームクリアにする

を実装していきます．その実装を通じて，複雑な条件分岐と関数の書き方を学んでいきましょう．その後，「自分ならでは」のゲームを作っていくためのコースの入り口までご案内します．

歩いて，止まって，一休み ～「条件分岐」で動きを変える

前編から改良している「ビットくん帰宅ゲーム」には，まだ「ビットくんが上下に移動できない」という問題が残っています．「←」「→」キーで左右に動かせますから，「↑」「↓」で上下にも動かせるようにしたいですね．どうすればよいでしょうか？

もし 9 月号の「まねておぼえるプログラミングのいろは（後編）」を読んでいたら，「ビットくんのツイートのうち『いいね！』がないものは，半透明にして見えにくくする」で使った「条件分岐」を思い出してみましょう．条件分岐とは「～な場合には……をする」のように条件が成り立つかによって実行するかどうかを変えるものです．ただし今回は，すべきことがやや複雑です．「『いいね！』がない場合に……をする」では，条件を 1 つだけ考えればよかったのですが，今回は押されたキーが「←」「→」「↑」「↓」の 4 つである場合に対して，4 つの条件分岐を考えなくてはなりません．このように，複数の条件に対する分岐を扱うためには，どうすればよいのでしょうか？

実はこのような分岐はサンプルプログラムで使われています．実際にプログラムを見てみましょう．後半部分に move-on-key という関数があります．この関数は，

```
(define (move-on-key world key)
  (cond
   [(key=? key "right")
    (bitmove world KEY-DISTANCE 0)]
   [(key=? key "left")
    (bitmove world (- KEY-DISTANCE) 0)]
   ( 中略 )
   [else world]))
```

という形になっています．2 行目の cond 以下が，たくさんの条件を扱うことのできる，複雑な条件分岐に関する部分です．最初の条件分岐（3 ～ 4 行目）

```
[(key=? key "right")
 (bitmove world KEY-DISTANCE 0)]
```

から，この部分で何が行われているのかを見てみましょう．(key=? key "right") の部分が条件です．最初の key=? は，その次にくる 2 つが同じキーであるかどうかを判定しています．押されたキー key が right

☆1 http://researchmap.jp/mutrf6bh8-2014482/#_2014482 からサンプルコードをダウンロードして解凍してください．

と同じもの，つまり「→」が押されている場合には，次の (bitmove world KEY-DISTANCE 0) が実行されます．同様に，2 番目の条件分岐（5 ～ 6 行目）

```
[(key=? key "left")
 (bitmove world (- KEY-DISTANCE) 0)]
```

は，「←」キーが押された場合に実行されます．これらの条件分岐から分かるように，この関数 move-on-key では「このキーが押された場合には，この処理を行う」ということが記述されています．どれにも当てはまらない場合は，最後の行の，

```
[else world]
```

が実行されます．

複雑な条件分岐は，条件と当てはまった場合に実行する処理を羅列しておき，上から順に「当てはまるかどうか」をチェックし，当てはまったところで実行します．どれにも当てはまらなかった場合は，最後の [else ……] に書いてある処理が実行されます．簡単な仕組みですが，強力です．

では，条件が当てはまった場合に，何をしているのか見てみましょう．最初の条件分岐（3 ～ 4 行目），

```
[(key=? key "right")
 (bitmove world KEY-DISTANCE 0)]
```

は，「→」キーが押された場合に，(bitmove world KEY-DISTANCE 0) を実行します．bitmove はビットくんを動かす関数で，world，KEY-DISTANCE，0 の 3 つの引数を受け取っています．ビットくんを動かしているのは，2 つ目と 3 つ目の引数です．2 つ目の引数は x 座標の変化，3 つ目の引数は y 座標の変化を意味しています．「→」が押された場合，KEY-DISTANCE の値だけ x 軸のプラス方向に動き，y 方向には動きません．同様に，次の「←」キーが押された場合には，KEY-DISTANCE の値だけ x 軸のマイナス方向に動き，y 方向には動きません．ここまで理解すれば，ビットくんを上下に動かす場合のプログラムが書けそうです．さきほどの条件分岐の（中略）の前に以下のプログラム，

```
[(key=? key "up")
 (bitmove world 0 KEY-DISTANCE)]
[(key=? key "down")
 (bitmove world 0 (- KEY-DISTANCE))]
```

を追加してみましょう．「↑」キーを押すと y 方向にプラス，「↓」キーを押すと y 方向にマイナスしているので良さそうですね．では，実行してみましょう．どうなるでしょうか？　ビットくんは，「↑」キーを押すと下に，「↓」キーを押すと上に動いてしまいます．これは，Racket の y 座標が上から下に向かって伸びていて，下の方が大きい値になっているからです．よって，プラスマイナスを逆に修正する必要があります．修正すると，move-on-key 関数の内容は

```
(define (move-on-key world key)
 (cond
  [(key=? key "right")
   (bitmove world KEY-DISTANCE 0)]
  [(key=? key "left")
   (bitmove world (- KEY-DISTANCE) 0)]
  [(key=? key "up")
   (bitmove world 0 (- KEY-DISTANCE))]
  [(key=? key "down")
   (bitmove world 0 KEY-DISTANCE)]
  (後略)))
```

となります．これで，ビットくんが上下に動くようになりました（図 -1）．でも，車にぶつかってもゲームオーバーになりませんし，家にたどり着いてもクリアになりません．「ビットくん帰宅ゲーム」は，まだ未完成です．

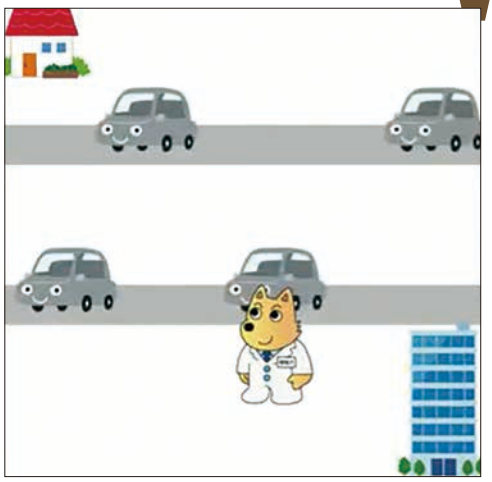

図 -1　ビットくんが上下に動いた画面

「ゲームのクリア」と「ゲームオーバー」がなくては～関数のプログラミング

「車にぶつかったらゲームオーバー」「家にたどり着い

たらクリア」を追加し，「終わり」のあるゲームにするには，どうすればよいでしょうか？

「ビットくん帰宅ゲーム」には，まだ終了判定がありません．そこで，終了を判定するための新しい関数，現在の world を受け取って終了させるかどうか判定する関数 stop-event を定義します．この関数は末尾の big-bang の直前に書きましょう．プログラム中のほかの関数を見ると，関数を定義するには

```
(define (関数名 受け取る引数名 ..)
 (実行する内容))
```

と書けばよいことが分かります．stop-event 関数の受け取る引数は，world です．world の内容によって，終了させるかどうかを判定し，終了条件を満たしていれば true（真），満たしていなかったら false（偽）を返すことにします．さらに先ほどの複雑な条件分岐 cond を使って場合分けをします．まず，場合分けは「車とビットくんが近ければゲームオーバー」だけを考えることにしましょう．ここまでを日本語の部分も含めて，プログラムの形にすると

```
(define (stop-event world)
  (cond
    [(車の位置とビットくんの位置が
      近ければ) true]
    [else false]))
```

となります．

このうち１つ目の条件分岐の日本語部分をプログラムに直せば，この関数は完成です．「車の位置」と「ビットくんの位置」を，構造体からの要素の取り出し（(world-◯◯ world):◯◯には要素の名前が入ります）を使って書き換えてみましょう．すると以下のようになります．

```
[((world-carlist world)と
  (world-bitplace world)が
  近ければ) true]
```

次に，位置が近いかどうかを判定する必要があります．実は，このサンプルプログラムにはすでに nearlist? という関数が定義されています．この関数は，１つ目の引数のリストの要素それぞれ（たとえば４台ある車の位置）が２つ目の引数と近いかどうかを判定し，近ければ true（真），近くなければ false（偽）を返す関数です．この nearlist? 関数を使って，上の日本語部分をプログラムに書き直すと，stop-event 関数は

```
(define (stop-event world)
  (cond
    [(nearlist?
      (world-carlist world)
      (world-bitplace world)) true]
    [else false]))
```

となります．しかし，まだ，ビットくんが車にぶつかっても，ゲームオーバーにはなりません．

stop-event 関数が「終了条件だ」と判定したらゲームが終わるようになるためには，プログラムの世界を作っている big-bang 部分に stop-event 関数を追加し，「この関数が終了条件を示すんだよ」と教える必要があります．「stop-event 関数が true と答えたら終了」は（stop-when stop-event）となります．big-bang の７行目の末尾に，これを追加すると，以下のようになります．

```
(big-bang (make-world
            100 START-POSN
            (中略) cargray)
           (to-draw redraw)
           (stop-when stop-event)
           (以下略))
```

プログラムを実行し，少し可哀想ですが，ビットくんを車にぶつけてみると，プログラムの実行が止まります．これで「ゲームオーバー」が作れました．

同じように，無事に家にたどり着いた場合も作ってみましょう．「ビットくん帰宅」の条件は，

```
[(家とビットくんの位置が近ければ) true]
```

と書くことができます．「ビットくんの位置」をプログラムに置き換えると

```
[(家と (world-bitplace world)が
  近ければ) true]
```

となります．家の座標は前編でも使用した posn という構造体を使って作りましょう．家の座標は（50,50）なので，

```
[((make-posn 50 50)と
  (world-bitplace world)が近ければ) true]
```

ということになります．ここで２つの座標が近いかどうかを判定する関数が必要になります．実はすでに

near? という関数が定義してあります．これは 1 つ目の引数と 2 つ目の引数の座標の距離を計算して近かったら true（真）を返し，近くなければ false（偽）を返す関数です．この near? 関数を使って，日本語をプログラムに置き換えると

```
[(near? (make-posn 50 50)
        (world-bitplace world)) true]
```

となります．これで，「ビットくん帰宅」の判定ができました．実行し，ビットくんに車を避けさせて無事帰宅させると，ビットくんが「おやすみ」の画像になり，プログラムの実行が止まることを確認できます．

しかしこれでは，ゲームクリアとゲームオーバーのどちらも，「プログラムの実行が止まる」ですので区別がつきません．これらを区別するために，無事にゴールしてゲームクリアとなった場合には，いつもどおりの表示に加えてプログラムの先頭で定義している「おつかれさまでした」という画像 otsukare を表示しましょう．そこでゲーム終了時に画像を表示する関数 last-pic を新しく定義します．この関数は world を受け取って，画像を表示します．プログラムを書く場所は big-bang の直前にしましょう．さきほどの終了判定関数 stop-event と同じように，複雑な条件分岐を使います．プログラムの形は，下記のようになります．

```
(define (last-pic world)
  (cond
    [(家の座標とビットくんの位置が近ければ)
     (画像 otsukareを表示)]
    [else ゴールしていないので,
          いつもどおりの表示]))
```

ゴールしていない場合に，いつもどおりの表示をするには，redraw 関数を利用して (redraw world) を実行します．家とビットくんが近いかどうか，つまりゴールしているかどうかの判定は，stop-event の「帰宅したので終了」条件と同様に，

```
[(near? (make-posn 50 50)
        (world-bitplace world))
 (画像 otsukareを表示)]
```

と書くことができます．このとき，いつもどおりの表示に加え，プログラムの先頭で定義している「おつかれさまでした」という画像 otsukare を表示します．これには画像を表示する関数 place-image を利用します．

図 -2　終了時の画像が表示された画面

この place-image 関数は，画像の名前・配置する x 座標・配置する y 座標・その下に表示するもの，の 4 つの引数を持ちます．ここでは，画像の名前は otsukare，表示する場所はゲームウィンドウの真ん中辺りにしたいので x 座標は 300，y 座標も 300，下に表示したいものはいつもどおりの表示 (redraw world)，ということにします．これらを加えると，last-pic 関数は

```
(define (last-pic world)
  (cond
    [(near? (world-bitplace world)
            (make-posn 50 50))
     (place-image
      otsukare 300 300 (redraw world))]
    [else (redraw world)]))
```

と完成します．終了時に last-pic 関数が実行されるためには，big-bang で終了時に実行するよう記述を加える必要があります．先ほど変更した (stop-when stop-event) の記述の最後に，last-pic を加えると以下のようになります．

```
(big-bang (make-world
           100 START-POSN
           (中略) cargray)
          (to-draw redraw)
          (stop-when stop-event last-pic)
          (以下略))
```

実行してみると，ビットくんが無事帰宅した場合，「おつかれさまでした」という画像が表示されるようになりました（図 -2）．

「自分のゲーム」を自在に作るために，次の一歩は？

最後に，「ビットくん帰宅ゲーム」をさらに改良したり，Racketで自分の作ってみたいゲームを自在に作ったりするために，必要な知識や情報を紹介します．

ゲームの世界に必要なイベントは，big-bangに登録する必要があります．今回登録したイベントには，

- ゲーム中にどう表示するかを決める`to-draw`
- どんなときにゲームを終了するかを決める`stop-when`
- 時間経過でどう`world`を変化させるかを決める`on-tick`
- キーが押されたときにどう`world`を変えるかを決める`on-key`

がありました．「ビットくん帰宅ゲーム」ではそれぞれ，`to-draw`に`redraw`，`stop-when`に`stop-event`と`last-pic`，`on-tick`に`move-on-tick`，`on-key`に`move-on-key`関数を登録しました．

Racketの情報源としてお勧めできるのは，RacketのWebサイトにあるリファレンス（https://docs.racket-lang.org/）です．英語ですが，すべての情報が載っています．たとえば，ここで「big-bang」を検索し，今回使用した「universe」のティーチパックに該当する部分を選ぶと，マウスのポインタを載せたときの`on-mouse`，押していたキーを放したときの`on-release`など，この記事で使用しなかった数多くのイベントを登録できることが分かります．それらのイベントを使うと，もっと面白いゲームが作れるかもしれません．

Racketでプログラミングを続けてみたいと思った人は，まず「ビットくん帰宅ゲーム」をもう少し改良してみましょう．以下にいくつか改良案を挙げておきますが，自分の思う改良を行うのも良いでしょう．

- ビットくんや家，建物の画像を違うものにする
- 背景を追加する
- ゲームオーバーのときにもゲームオーバーを示す画像を表示する
- チェックポイントを作り，通過していないとゴールと見なさないように変更する（たとえば鍵を拾わないと家に入れないなど）

1つ目は，画像に名前をつけている個所の画像を差し替えればできます．2つ目は，プログラム中の道路等の表示をしている`haikei`に背景画像の表示を追加しましょう．3つ目は，本稿でのゲームのクリアと同様に，`last-pic`を書き換えて新たな条件分岐を追加しましょう．4つ目は，チェックポイントを通過しているかどうかの情報が必要になります．そのため，`world`構造体の定義を変更する必要があります．また，それに伴って`world`を使用している部分の書き換えが必要になります．よって，ほかに比べてかなり難易度が高いでしょう．変更した上で`stop-event`などを書き換えましょう．

さらに，`world`の定義を変更し，登録されている関数を書き換えることで，あなた独自の新しいゲームを作ることもできます．Racketでは，さまざまなゲームを作ることができます．「落ちてくるものを拾うゲーム（得点がアップするお宝もあり）」「動くものをクリックで消していくゲーム」「ボールを跳ね返すゲーム」「ブロックをくっつけて崩すゲーム」などがRacketで実装できます．アイディア次第で今までにないような面白いゲームを作ることができるでしょう．

本稿では，プログラミング入門として，前編・後編に分けて「ビットくん帰宅ゲーム」を通して，プログラミングで使われる要素を紹介しました．プログラミングは突き詰めていくと実に奥が深いのですが，意外と単純なものの組合せや繰り返しでできているものです．本稿で，プログラミングに親しみを感じ，「続きを自分で作ってみよう」とか，「ほかのプログラミング言語も学んでみよう」と思っていただければ幸いです．

（2016年7月5日受付）

対馬かなえ（正会員）■ k_tsushima@nii.ac.jp
2013年お茶の水女子大学大学院人間文化創成科学研究科理学専攻博士後期課程修了．2015年より国立情報学研究所アーキテクチャ科学研究系助教．関数型言語，型推論，デバッグ手法等に興味を持つ．