2009 年度夏学期「情報」期末試験問題
解答と解説

制　作　khRules

注意　この文書は制作中のものです（最終更新日: 2009 年 9 月 30 日）。

共通問題 2

問題

ボーリングの点数を計算する方法を考える。

ボーリングはボールを投げて 10 本のピンを倒す競技で、10 個のフレームという単位でボールを投げる。1 つのフレームは 1 投または 2 投よりなる。1 投目で 10 本全てが倒れたら、そのフレームをストライクと言い、2 投目のボールは投げない。1 投目で倒れたのが 9 本以下なら、残っているピンに対して 2 投目のボールを投げる。そこで残っているすべてのピンが倒れたら、そのフレームをスペアと言い、1 本以上残ったらオープンと言う。

フレームごとの点数計算の規則は次のようであり、ゲームの点数はフレームの点数を累計して得られる。

1. ストライクの得点は、10 に次の 2 回の投球で倒れたピンの本数を加えたものである。
2. スペアの得点は、10 に次の 1 回の投球で倒れたピンの本数を加えたものである。
3. オープンの得点は、そのフレームの 2 回の投球で倒れたピンの本数である。

さらに、最後の 10 フレーム目に特殊な処理が必要だが、ここではそれは考えないことにする。

さて、1 回の投球で倒れたピンの数が入力として入ってくるたびに、これまでの点数の合計を行う計算を考える。入力されたピンの数は、P という変数に入ってくるものとし、点数の合計は T という変数に入れていくものとする。

P: 入力されたピンの数

T: これまでの点数の合計（初期値は 0）

さらに、計算を実行するために、次のような変数を用意する。

S: 直前のフレームの状態を示す（初期値は 0）。

S=0　オープン

S=1　スペア

S=2　ストライクでかつその前がストライクではない

S=3　ストライクでかつその前もストライク（いわゆるダブル）

R: 現在が 2 投目のとき、1 投目で倒れずに残ったピンの数

I: 現在が 1 投目か 2 投目かを表す（初期値は 1）

I=1　1 投目

I=2　2 投目

点数計算の規則は、ストライク、スペア、オープンに対してその後の投球による加算方法を述べているが、計算の際は現在の投球で倒れたピンの数を基に、前のフレームがオープン、スペア、ストライク、ダブルのいずれかで、点数合計値 T を変更していくという方法をとることにする。その手順を次のように考える。

計算手順

```
if I=1 then
    if P=10 then
        A: それに応じて T、S、I を変更する
    else （すなわち 0≦P<10 なら）
        B: それに応じて T、R、I を変更する
    endif
else （すなわち I=2 なら）
    C: それに応じて T、S、I を変更する
endif
```

ここで上の A の部分は次のように書くことができる。

A の計算手順

```
if S=0 then
    T ← T+P
    S ← 2
else if S=1 then
    T ← T+2P
    S ← 2
else if S=2 then
    T ← T+2P
    S ← 3
else （すなわち S=3 なら）
    T ← T+3P
    S ← 3
endif
I ← 1
```

ここで、「if （条件 1） then （文 1） else if （条件 2） then （文 2） else if … else （文 n） endif」は、条件によってn個に場合分けする操作を表す。すなわち、（条件 1）が成立したら（文 1）を実行し、（条件 1）が成立せずに（条件 2）が成立したら（文 2）を実行し、以下同様にして、最後に（条件 1）から（条件$n-1$）までの全てが成立しなかったら、（文 n）を実行する。

この計算手順に対し、次のようなデータが順に P に与えられたとする。

[7, 3, 8, 0, 10, 10, 6, 4, 7, 1]

このとき、T には次のような値が対応して計算される。

[7, 10, 26, 26, 36, 56, 74, 82, 96, 97]

倒れたピンの数が入力されるたびに合計を計算しているので、通常のボーリングのスコアシートとは途中の表示が異なることに注意しなさい。

問題

A の計算手順を参考にして、

(1) B の計算手順を書きなさい。

(2) C の計算手順を書きなさい。

これは難しすぎないか？

この問題は今回出された問題の中で最も難しく、多くの方が苦戦されたのではないでしょうか？

プログラミングの問題はよく出されているのですが、今まではほとんどまたは完全に出来上がっているプログラムを読むことに焦点が当てられていました。しかし、今回はプログラムを書くことが目的です。

自然言語の外国語（例えば、日本人にとっての英語、中国語、フランス語、ドイツ語など）でも、読むことよりも書くことの方が遥かに難しいと考えられます。なぜなら、読む場合は、文法や語法といった規則をマスターしていなくても、単語の意味からある程度文章の意味を推測することができるのですが、書く場合は文法や語法といった規則を内面化する（完全に覚え、かつ、もはや自分自身の一部であるかのように難なく利用できるようになる）ことが求められるからです。

プログラムを書くことはどうせ来学期（冬学期）の授業「情報科学」で取り扱うのですから、今学期（夏学期）の授業「情報」としてはプログラムを読むことにとどめるべきだったのではないかと思います。

ボーリングを見直す

今回の題材はボーリング（bowling）[1]の得点計算です。

ボーリングという競技を知らない人はまずいないと思われますが、念のため簡単に説明して

[1] もちろん、地面の発掘作業（boring）のことではありません。

おきましょう。ボーリングはスイカやメロンほどの大きさがあるボールを投げて転がし、遠くに並べられたピンを倒していく競技です。普通、ピンは一度に 10 本が用意され、倒す数が多ければ多いほど高得点が得られます。競技は普通 10 個のフレームで区切られており、新しいフレームが始まるごとに、(その直前のフレームで倒れずに) 残っているピンが捨てられ、新しいピンが補充（普通は 10 本）されます。1 フレームにボールを投げられる回数は基本的に 2 回までで、その 2 回のうちに（本当はできるだけ 1 回で）ピン（普通は 10 本）を全て倒すことがプレイヤーの目的です。成績を記録する帳簿であるスコアシートにはピンを倒した数が逐次記録され、ピンを倒した数を基に得点が計算され、それが成績になります。

得点は通常倒れたピンの数がそのまま得点になる（ピン 1 本あたり 1 点）のですが、フレーム中に全てのピンを倒す（立っているピンが一本もない状態にする）とより高い点数が得られる機会が与えられます。新しいフレームが始まってから 1 投目で全てのピンを倒すとストライクとなり、そのフレームは 2 投目を迎えることなく終了し、そこから次の連続した 2 回の投球 (違うフレーム中にあります) で倒したピンの数も（ストライクが出た）フレームの得点として加算されます。この加算で使用された倒したピンの数は、そこでピンを倒したときのフレームの得点にも（通常通り）なりますので、結果としてその部分の点数が二重に加算されることになります。特に、2 回連続でストライクを出した直後の投球については、倒れたピンの数が三重に点数として加算されることになります。

一方、各フレームの 1 投目で全てのピンを倒せなかった場合は、立っているピンをそのままにした状態で 2 投目が始まり、その投球で残っている全てのピンを倒した場合はスペアーとなり、そこから次の投球（1 回だけ）で倒したピンの数も（スペアーが出た）フレームの得点として加算されます。一度に 10 本倒しても、それがフレーム中で 2 投目の場合にはストライクではなくスペアーになることに注意してください。

ゲーム中最終フレームだけが特殊で、このフレームでは最大で 3 回まで投球を行うことができます。ストライクやスペアーが出てもそのフレームが終了せず、新たにピンが補充されます。ただし、最終フレームでストライクやスペアーが出ても、その後に倒したピンの数が重複して点数に加算されることはありません（同じフレーム内だから）。また、最終フレームの 1〜2 投目で全てのピンを倒せなかった場合は、3 投目を迎えることなくそのフレームは終了します。

なお、ボーリングにはガター（フレーム中の 1 投目で一本もピンを倒せなかった場合)、ファウル（反則を行った場合）など他の言葉もありますが、これらは得点の計算には関与しません (倒れたピンの数が 0 本と見なすだけです)。

一度に置かれるピンの数は普通通り 10 本として、得点計算の例として、問題文中で取り上げられているように、倒れたピンの数が

7、3、8、0、10、10、6、4、7、1

の場合について考えてみましょう。

最初の数字「7」が第 1 フレームの 1 投目で倒れたピンの数です。

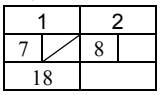
10 本のピン全てが倒れたわけではありませんので、それに続く投球は先ほどの投球と同じフレーム中にあります。つまり、その投球は第 1 フレームの 2 投目になります（倒れたピンは捨てられますが、残ったピンはそのままの状態になります。まだ新しいピンは補充されません)。

このようにして、第 1 フレームは 1 投目から順に 7 本、3 本のピンが倒れたことになり、合計で 10 本倒れていますので、ピンを全て倒したことになり、スペアーと呼ばれることになります。そこで、ボーリングのスコアシートの第 1 フレームには

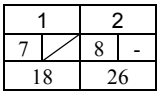
と記入されます。スペアーではその直後の投球で倒れたピンの数もフレームの得点として加算されますので、第 1 フレームの得点は次の投球を行うまで決定できません。これで第 1 フレームは終了し、第 2 フレームが開始します。

第 2 フレームの 1 投目ではピンが 8 本倒れたことになっていますので、スコアシートの第 2 フレームには

と記入されます、これで、第 1 フレームで出たスペアーの効果は終了になりますので、第 1 フレームの得点が決定されます。第 1 フレームの得点は、第 1 フレームで倒れたピンの数 10 本に、第 2 フレームの 1 投目で倒れたピンの数 8 本を加えた 18 点です。

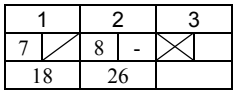


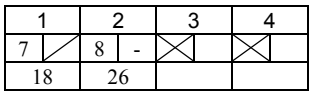
第 2 フレームの 2 投目ではピンが 0 本倒れた（一本も倒れなかった）ことになっています。第 2 フレームで 2 回投げたので、第 2 フレームは終了しますが、第 2 フレームでは合計で 8 本のピンが倒れ、全てのピンを倒せなかったために、この時点で第 2 フレームの得点が決定されます。第 2 フレームの得点は第 2 フレームで倒れたピンの数 8 本と同じ 8 点です。

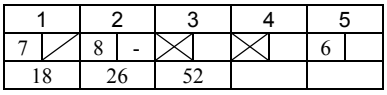


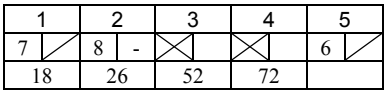
第 2 フレームの 1 投目で倒れたピンの数が第 1 フレームと第 2 フレームの両方の得点として加算されていますが、これは第 1 フレームでスペアーが出たために、倒れたピンの数を得点として二重に加算することになっているからです。

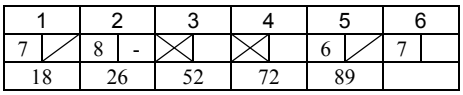
以後同様にゲームを進めていくと、スコアシートは以下のように得点が記入されます。各フレームの 1 投目で 10 本のピンが倒れた場合はストライクとなり、そのフレームは 2 投目を迎えることなく終了され、さらに、そのフレームの得点は、その直後 2 回の投球が終わって始めて決定されることに注意してください。また、スコアシートにおいて、「G」は各フレームの 1 投目でピンが一本も倒れなかったこと（ガター)、「-」は各フレームの 2 投目以降でピンが一本も倒れなかったこと、「×」はストライク、「／」はスペアーを表します。

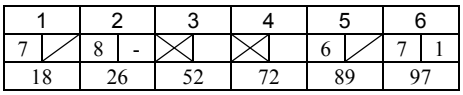












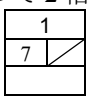
既に述べたように、通常、得点はフレームごとに記入されることになっており、フレームごとに得点が確定して初めて得点の表示が更新されるようになっています。例えば、ストライクやスペアーを出した瞬間には、得点の表示は更新されません（例えば、現在の得点が 70 点で、直前の 2 回の投球にストライクやスペアーがない状態でストライクを出しても、得点の表示は 80 点にはならず、70 点のままです)。

しかし、実際には、一球ごとに得点の計算が可能なのです。基本的にはピンを一本倒すごとに 1 点が得られるのですが、ストライクを出したときには、次の連続した 2 回の投球で倒れたピンの数が重複して得点として加算され、スペアーを出したときには、その直後の投球で倒れたピンの数が重複して得点として加算されるのでした。ということは、ストライクを出すと、次の連続した 2 回の投球では得点の倍率が 1 だけ上がり、スペアーを出すと、その直後の投球では得点の倍率が 1 だけ上がる、と言い換えることもできます。単にストライクやスペアーを出した場合は、次の 1 回（スペアーの場合）または 2 回（ストライクの場合）の投球では得点が 2 倍になりますが、2 回連続でストライクを出した場合は、その直後の投球では得点が 3 倍になります (2 つ前の投球におけるストライクによって得点の倍率が 1 上がり、1 つ前の投球におけるストライクによってもまた得点の倍率が 1 上がるため)。ずっと連続でストライクを出していけば、それ以降の投球でもずっと得点を 3 倍にしていくことができます。ただし、最終フレームではその次のフレームがありませんので、最終フレームでストライクやスペアーを出しても、その次の投球における得点の倍率を上げることはありません（ただし、最終フレームでは、まだ投球の残り回数がある場合に限り、そのフレームを終了することなく、新たにピンが補充されますので、ストライクやスペアーを出さない場合に比べて高い得点を得る機会が与えられることに代わりはありません)。

この方法で、倒したピンの数は問題文通り（7、3、8、0、10、10、6、4、7、1 本）として、1 回の投球ごとに得点を計算してみると、次のようになります。最初は得点の倍率は 1 であり、その状態で 7 本のピンを倒しましたので、7 点が得られます。ストライクでもスペアーでもありませんから、次の投球では得点の倍率は上がりません。

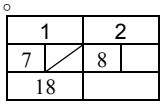
フレーム番号	フレーム中の投球番号	倒したピンの数	得点	備考
1	1	7	7 × 1 = 7	
合計			7	

次に 3 本のピンを倒しましたので、3 点が得られます。ここではスペアーになっていますので、次の投球では得点の倍率が 1 だけ上がって 2 倍になります。



フレーム番号	フレーム中の投球番号	倒したピンの数	得点	備考
1	1	7	7 × 1 = 7	
1	2	3	3 × 1 = 3	スペアー
合計			10	

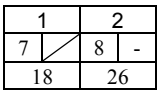
次に 8 本のピンを倒しましたが、直前の投球ではスペアーが出ていることから、得点は 2 倍になり、8 点の 2 倍で 16 点が得られます。



フレーム番号	フレーム中の投球番号	倒したピンの数	得点	備考
1	1	7	7 × 1 = 7	
1	2	3	3 × 1 = 3	スペアー
2	1	8	8 × 2 = 16	
合計			26	

フレームごとの計算では、ここで第 1 フレームの得点が 18 点に確定されています。ここで倒した 8 本のピンの得点は、第 1 フレームに加算された分は既に表示されていますが、第 2 フレームに加算された分はまだ表示されていないことに注意してください。また、ストライクやスペアーが出ておらず、先ほどのスペアーは今回の投球で得点倍率向上の効果が切れていますので、次の投球での得点の倍率は通常通り 1 倍です。

次にピンが一本も倒れませんでした（0 本倒しました）ので、得点は一点も得られません（0 点が得られます)。



フレーム番号	フレーム中の投球番号	倒したピンの数	得点	備考
1	1	7	7 × 1 = 7	
1	2	3	3 × 1 = 3	スペアー
2	1	8	8 × 2 = 16	
2	2	0	0 × 2 = 0	
合計			26	

ここで第 2 フレームの得点が 8 点に確定され、合計は 26 点と表示されますが、一球ごとに得点を計算した場合と結果が一致しています。

これ以降は同様にすることで、以下のような結果が得られます。2 回連続でストライクを出した直後の投球では得点の倍率が 3 になっていますので、6 本倒れた分の得点は 6 点の 3 倍で 18 点になっています。

1	2	3	4	5	6
7 /	8 -	X	X	6 /	7 1
18	26	52	72	89	97

フレーム番号	フレーム中の投球番号	倒したピンの数	得点	備考
1	1	7	7 × 1 = 7	
1	2	3	3 × 1 = 3	スペアー
2	1	8	8 × 2 = 16	
2	2	0	0 × 2 = 0	
3	1	10	10 × 1 = 10	ストライク
4	1	10	10 × 2 = 20	ストライク（ダブル）
5	1	6	6 × 3 = 18	
5	2	4	4 × 2 = 8	スペアー
6	1	7	7 × 2 = 14	
6	2	1	1 × 1 = 1	
合計			97	

フレームごとに計算した場合と一回の投球ごとに計算した場合とで得点が一致しており、一回の投球ごとに得点を計算することもできることが分かります。次にボーリングで遊ぶときは、実際に一回の投球ごとに得点を計算して、実際の得点に一致することを確認してみてください（ただし、既に述べたように、最終フレームでストライクやスペアーを出しても、その次の投球における得点の倍率は上がらないことに注意してください）。

いかにしてプログラムを書くか？

今回の問題は、まさに上記のような得点の計算をプログラムとして書くということになります。プログラムの一部は既に与えられていますから、まずは与えられたプログラムを読んでみましょう。ちなみに、**プログラムでは原則として上（先頭の行）から順に処理が実行されます**。今回は、プログラムのメインの部分が以下のようになっています（問題文の「計算手順」を参照してください）。

```
if I=1 then
   if P=10 then
      A: それに応じてT、S、Iを変更する
   else （すなわち0≦P<10なら）
      B: それに応じてT、R、Iを変更する
   endif
else （すなわちI=2なら）
   C: それに応じてT、S、Iを変更する
endif
```

このままでは分かりづらいのなら、普段慣れ親しんでいる言語（例えば、日本人なら日本語）に翻訳してみましょう。「if」はこれから条件を指定し、その条件に当てはまるかどうかに基づいて場合分け（条件分岐）を行うことを表します。

例えば「if I=1 then」は「もし I=1 なら」、あるいは、「もし変数 I の値が 1 に一致するのなら」と直訳できますが、これだけではまだ意味が分かりやすいとは言えないかもしれません。ここで、問題文によると、変数 I は「現在が 1 投目か 2 投目かを表す」となっており、フレーム中の投球の番号を表していることが分かります。すると、「if I=1 then」は意訳すれば、「もし、この投球がフレーム中で 1 番目のものなら」となります。これで少しは分かりやすくなったでしょうか？

ちなみに、「then」の直後には、条件（先ほどの例では「I=1」、すなわち、「変数 I の値が 1 に一致する」ことです）に当てはまる場合の処理を表す文を書き、「else」の直後には、条件に当てはまらない場合の処理を表す文を書きます。else はそれ自身（「else」）より前で、まだ他の「else」に対応づけられていない「if」のうち直近のものに対応することに注意してください。例えば、「if I=1 then」に対応する「else」は「もし、この投球がフレーム中で 1 番目のもの」でない場合を表すことになります。しかし、ボーリングでは、フレーム中での投球は 1 番目か 2 番目かのいずれかですから、単に「もし、この投球がフレーム中で 1 番目のものでなければ」と翻訳するよりは、「もし、この投球がフレーム中で 2 番目のものなら」と翻訳する方が分かりやすいでしょう。

「endif」はそこで場合分け（条件分岐）が終了していることを表します。どの場合分けが終了するのかについては、endif はそれ自身（「endif」）より前で、まだ他の「endif」に対応づけられていない「if」のうち直近のものにかかります。つまり、直前の場合分け（条件分岐）を終了させることを意味します。ここでは「条件分岐終了」と直訳しておくことにして、どの場合分け（条件分岐）を終了させるのかを具体的に見るのは後にしましょう。

他の文も同じように翻訳してみましょう。「if P=10 then」は直訳すると「もし変数 P の値が 10 に一致するのなら」になりますが、問題文では変数 P は「入力されたピンの数」とあり、一回の投球によって倒れたピンの数を表していることが分かります（「入力された」というのは、倒れたピンの数に基づいてボーリングの得点計算システムに入力されたデータという意味です）ので、「もし倒れたピンの数が 10 本なら」と意訳することもできます。さらに、この文は「if I=1 then」、すなわち「もし、この投球がフレーム中で 1 番目のものなら」に当てはまる部分に書かれており、フレーム中 1 投目で 10 本のピンを倒すことはストライクを意味しますので、「if P=10 then」に対する更なる意訳として「もし、この投球でストライクが出たら」とすることもできます。こちらの方が分かりやすいのではないでしょうか？

以上のようにしてプログラムを翻訳した例を以下に示します。

条件分岐 1-A: もし、この投球がフレーム中で 1 番目のものなら
  条件分岐 2-A: もし、この投球でストライクが出たら
    処理 A を実行する
  条件分岐 2-B: もし、この投球でストライクが出なかったら

処理 B を実行する
条件分岐 2 終了
条件分岐 1-B: もし、この投球がフレーム中で 2 番目のものなら
処理 C を実行する
条件分岐 1 終了

さて、「else」や「endif」はそれ自身より前で、まだ対応付けられていない直近の「if」にかかると説明しましたが、例えば、元のプログラムで 4 行目の「else」は 2 行目の「if」にかかります。1 行目にも「if」がありますが、2 行目の方が近いので、1 行目の「if」にはかかりません。同様にして、6 行目の「endif」は 2 行目の「if」にかかります。1 行目にも「if」がありますが、2 行目の方が近いので、1 行目の「if」にはかかりません。一方、7 行目の「else」は 1 行目の「if」にかかります。2 行目の「if」の方が 7 行目の「else」に近いのですが、2 行目の「if」は既に 4 行目の「else」と対応づけられていますので、7 行目の「else」は 2 行目の「if」にはかかりません。同様にして、9 行目の「endif」は 1 行目の「if」にかかります。2 行目の「if」の方が 9 行目の「endif」に近いのですが、2 行目の「if」は既に 6 行目の「endif」と対応づけられていますので、9 行目の「endif」は 2 行目の「if」にはかかりません。

この説明を読んで、「if」、「else」と「endif」の対応がややこしいと感じられた読者も多いことでしょう。そこで、通常は、分かりやすさのために、プログラムに何らかの印を付けておきます。場合分け（条件分岐）に番号や標識（ラベル）（label）をつけるという手もありますが、通常は、「if」-「else」-「endif」部（「else」はないこともあります）で一つのブロックと見なし、そのブロックの内部では**字下げ（インデント）**（indentation）を行う、すなわち、行の先頭にいくらかの空白を挿入するという方法を採用します。実際、元のプログラムを見ると、1、7、9 行目は同じ字下げレベル（行の先頭に挿入されている空白の幅が同じ）になっており、これらが互いに関係づけられていることが一目で分かるようになっています。1～9 行目は「if」-「else」-「endif」という一つのブロックを構成しており、その内部にある 2～6 行目と 8 行目には字下げが施されています（7 行目の「else」は 1 行目の「if」にかかりますので、字下げを施さないでおきます（1 行目の「if」と同じ字下げレベルにします））。同様にして、2～6 行目もまた「if」-「else」-「endif」という一つのブロックを構成しており、その内部にある 3、5 行目には（既に 1～9 行目のブロックの内部にあるものとして字下げされているのに）さらに字下げが施されています。

多くのプログラミング言語では、行の先頭にある空白を無視するようになっており[2]、そのため、以下のように字下げを施さなくても正しいプログラムとして動作させることができることになっています[3]。

```
if I=1 then
if P=10 then
A: それに応じてT、S、Iを変更する
else （すなわち0≦P<10なら）
```

[2] 例えば、FORTRAN や Python は行の先頭にある空白を意味のあるものと見なしていますので、プログラマーが勝手に省略することはできません

[3] とは言っても、今扱っているのは疑似コードであって、実際にコンピューターで動かすことができるようなものではありません。

```
B: それに応じてT、R、Iを変更する
endif
else （すなわちI=2なら）
C: それに応じてT、S、Iを変更する
endif
```

しかし、これではプログラムが見づらくなってしまいます（「else」や「endif」がどの「if」にかかるのかを、プログラマーが素早くかつ正確に判定することができなくなってしまいます）ので普通は推奨されません。他にも、**注釈（コメント）**（comment）など、プログラムの動作には影響を及ぼさない要素がありますが、それらはコンピューターには意味のないものでも、プログラマーには有意義なものとして利用されることが多いのです。

プログラムのメイン部分の翻訳が終わったところで、次は、処理 A を見てみましょう（問題文の「A の計算手順」を参照してください)。

```
if S=0 then
    T ← T+P
    S ← 2
else if S=1 then
    T ← T+2P
    S ← 2
else if S=2 then
    T ← T+2P
    S ← 3
else （すなわちS=3なら）
    T ← T+3P
    S ← 3
endif
I ← 1
```

「else if」は問題文にあるように、場合分け（条件分岐）で複数の場合を設定するために使用されます。今回は、「else if」を 1 個書くごとに「endif」を 1 個追加する必要はありません。その場合分けの部分を翻訳してみた結果を以下に示します。

条件分岐 3-A: もし、直前のフレームがオープン（S=0）なら

T ← T+P

S ← 2

条件分岐 3-B: もし、直前のフレームがスペアー（S=1）なら

T ← T+2P

S ← 2

条件分岐 3-C: もし、直前のフレームがストライクでかつその前のフレームがストライクではない（S=2）なら

T ← T+2P

S ← 3

条件分岐 3-D: （else）もし、直前のフレームがストライクでかつその前のフレームもストライク（S=3）なら

T ← T+3P

S ← 3

条件分岐 3 終了

I ← 1

さて、「←」は変数への値の**代入**（assignment）を表すもので、「←」の右側にあるものを計算し、その結果を「←」の左側にある変数に代入するというものです。代入は、単に変数を箱などの容れものと見なした場合に、その中に何かを入れることにすぎません。数学では等号「=」が代入の意で用いられることがありますが、このプログラムでは等号「=」は専ら「左右が等しい（かどうか）」を表す記号であって、代入は「=」ではなく「←」を用いるようになっています。実は、一つの記号に複数の意味を持たせない方が、文の解釈を単純にすることができ、与えられたプログラムを実行するプログラム（例えば、インタープリターやコンパイラー）を制作する人にとっては楽なのです。具体例を挙げると、プログラミング言語として非常に有名な C [4]は代入演算子（記号）を「=」、等比較演算子を「==」（「=」を 2 つつなげます）として明確に区別しており、C を元に作られた C++、C#、D、Java なども同様になっています。東京大学が冬学期に開講する授業「情報科学」で使用されるプログラミング言語・Ruby でも、代入演算子（記号）を「=」、等比較演算子を「==」または「===」（「=」を 3 つつなげます）として明確に区別しています（「==」と「===」は、本当は意味が異なるのですが、ここでは取り上げません）。

「←」の用法で例を挙げると、「T ← T+P」では、まず「←」の右側にある「T + P」を計算します。「T + P」は変数 T の値に変数 P の値を加算することを意味します。その加算の結果を、「←」の左側にある「T」、すなわち、変数 T に代入します。従って、「T ← T+P」を直訳すると、「T に T + P を代入する」、あるいは、「変数 T の値に変数 P の値を加算し、その結果を変数 T に代入する」となります。値の代入が行われれば、代入先の変数の内容は変化します。変数 T が「←」の左右両方に出現していますが、必ず「←」の右側の計算が終わってから値の代入が行われますので、特に問題はありません。「T ← T+P」は結局のところ、「変数 T の値を変数 P の値の分だけ増加させる」という意味になります。こちらの方が分かりやすいですね。ちなみに、それを意訳するなら、「合計点数を倒れたピンの数の分だけ増加させる」となりましょう。

同様に、「T ← T+2P」は直訳すると「T に T + 2P を代入する」、あるいは、「変数 P の値を 2 倍し、変数 T の値にそれ（変数 P の値の 2 倍）を加算し、その結果を変数 T に代入する」となり、意訳すれば「変数 T の値を、変数 P の値を 2 倍した分だけ増加させる」、さらに意訳すれば、「合計点数を、倒れたピンの数を 2 倍した分だけ増加させる」となります。「2P」は「変数 P の値を 2 倍したもの」という意味で、ここでは数学のようにかけ算（乗算）の記号を省略していますが、実際の多くのプログラミング言語ではかけ算（乗算）の演算子をアスタリスク「*」と定めており、省略はできないことに注意してください。

以上のようにして「A の計算手順」を翻訳した結果を以下に示します。

条件分岐 3-A: もし、直前のフレームがオープン（S=0）なら

合計点数を倒れたピンの数の分だけ増加させる

直前のフレームをストライクでかつその前のフレームがストライクではないとする

条件分岐 3-B: もし、直前のフレームがスペアー（S=1）なら

[4] これは略称ではなく、れっきとした正式名称（C Programming Language）です。

合計点数を、倒れたピンの数を 2 倍した分だけ増加させる

直前のフレームをストライクでかつその前のフレームがストライクではないとする

条件分岐 3-C: もし、直前のフレームがストライクでかつその前のフレームがストライクではない（S=2）なら

合計点数を、倒れたピンの数を 2 倍した分だけ増加させる

直前のフレームをストライクでかつその前のフレームもストライクとする

条件分岐 3-D: （else）もし、直前のフレームがストライクでかつその前のフレームもストライク（S=3）なら

合計点数を、倒れたピンの数を 3 倍した分だけ増加させる

直前のフレームをストライクでかつその前のフレームもストライクとする

条件分岐 3 終了

次の投球をフレーム中の 1 番目のものとする

ところで、上記の文章、すなわち処理 A は、結局のところ、何をするものでしょうか？　処理 A は現在の投球がストライクを出している場合に、得点計算と、次の投球における得点の倍率の決定を行うものです。

現在の投球における得点の倍率は、それよりも前の投球に依存する（現在の投球には依存しません）のですが、得点の倍率を変化させるのはストライクとスペアーであり、それらが影響を及ぼすのは、せいぜいその後にある 2 回の投球まで（ストライクは次の 2 回の投球に、スペアーは次の 1 回の投球に、それぞれ影響を及ぼします）ですから、直前の 2 回の投球がどのようなもの（ストライクなのか、スペアーなのか、そのいずれでもないのか）であったのかが分かれば、得点の倍率が分かります（本当は現在のフレームがゲーム中最終フレームなのかそうでないのかも得点の倍率に影響を及ぼすのですが、今回の問題では取り扱わないことになっています)。そこで求められた得点の倍率を基にして、現在の投球の得点を決定するのです。例えば、直前のフレームがオープン（S=0）の場合とは、直前の 2 回の投球ではストライクやスペアーを出していないということですから、この場合には得点の倍率は通常通りの 1 になり、倒れたピンの数の分がそのまま得点になります。一方、直前のフレームがスペアー（S=1）の場合とは､直前の投球ではスペアーを出しているということです(ボーリングのルールによれば、1 回前の投球でスペアーを出しているのなら､2 回前の投球ではストライクもスペアーも出していないはずです）から、この場合には得点の倍率は通常よりも 1 だけ上った 2 になり、倒れたピンの数を 2 倍にしたものが得点になります。直前のフレームがストライクでかつその前のフレームがストライクではない（S=2）場合も得点の倍率は通常よりも 1 だけ上った 2 になり、倒れたピンの数を 2 倍にしたものが得点になります。一方、直前のフレームがストライクでかつその前のフレームもストライク（S=3）場合は、得点の倍率は直前の 2 つのストライクによってそれぞれ 1 だけ上げられて 3 になり､倒れたピンの数を 3 倍にしたものが得点になります。

得点を計算したら、次の投球における得点の倍率が正しく求まるように、フレームの状態を変えておかなければなりません。処理 A は現在の投球でストライクを出した場合のものですか

ら、直前の 1 回の投球でストライクを出しているのか（S=2 または S=3）そうでないのか（S=0 または S=1）によって場合分けをすればよいことになります。直前の 1 回の投球でストライクを出しているのならば、次の投球では「（次の投球にとって）その直前の 2 回の投球でいずれもストライクを出している」（S=3）ことになり、得点の倍率は 3 になりますが、直前の 1 回の投球でストライクを出しているのならば、次の投球では「（次の投球にとって）1 回前の投球ではストライクを出しているが、2 回前の投球ではストライクを出していない」（S=2）ことになり、得点の倍率は 2 になります。

以上の説明に合うように作られたのが、問題文に掲載された処理 A（「A の計算手順」）なのです。なお、処理 A の最後では「I ← 1」、すなわち、明示的に次の投球をフレーム中の 1 番目のものとしていますが、そもそも処理 A に入るには現在の投球がフレーム中 1 番目（I=1）でなければなりませんので、わざわざ「I ← 1」として（既に値 1 が格納されている）変数 I に値 1 を代入する必要はありません（「I ← 1」を省略することができます）。問題文に「I ← 1」と書かれているのは、処理 B や処理 C は受験者が書くことになるし、その際には処理 A を参考にすることになっていますので、変数 I の値の変化（更新）も考慮しなさいという出題者からのメッセージを込めるためなのでしょう。

処理 A をまとめれば、「フレーム中 1 投目にストライクを出した場合における、得点の計算とフレームの状態（S）の変化」ということになります。

これで、与えられたプログラムの説明は終わりです。私たち（解答者）が行うべきことは、フレーム中 1 投目にストライクを出した場合の処理 A を基にして、フレーム中 1 投目にストライクを出さなかった場合の処理 B（得点の計算、残ったピンの数の計算、投球のフレーム中における番号の更新）と、フレーム中 2 投目に関する処理 C（得点の計算、フレームの状態の更新、投球のフレーム中における番号の更新）を書くことです。

まずは処理 B。処理 B は処理 A と同じ、フレーム中 1 投目についての処理ですから、その投球における得点の倍率は処理 A と処理 B とでは変化しないはずです。そこで、処理 A のうち、場合分け（条件分岐）と得点の計算の部分をそのまま抜き出しておきます。

プログラム（暫定）

```
if S=0 then
    T ← T+P
else if S=1 then
    T ← T+2P
else if S=2 then
    T ← T+2P
else （すなわちS=3なら）
    T ← T+3P
endif
```

プログラム（暫定）の翻訳

（説明の都合上条件分岐の番号を変えてありますが、プログラムそのものには影響はありません）

条件分岐 4-A: もし、直前のフレームがオープン（S=0）なら

合計点数を倒れたピンの数の分だけ増加させる

条件分岐 4-B: もし、直前のフレームがスペアー（S=1）なら

合計点数を、倒れたピンの数を 2 倍した分だけ増加させる

条件分岐 4-C: もし、直前のフレームがストライクでかつその前のフレームがストライクではない（S=2）なら

合計点数を、倒れたピンの数を 2 倍した分だけ増加させる

条件分岐 4-D: （else）もし、直前のフレームがストライクでかつその前のフレームもストライク（S=3）なら

合計点数を、倒れたピンの数を 3 倍した分だけ増加させる

条件分岐 4 終了

処理 B が処理 A と異なるのは、処理 A はストライクを出した場合についてのものですから、フレーム中 2 投目に向けて準備をする必要がありませんが、処理 B はストライクを出していない場合についてのものですから、フレーム中 2 投目に向けて準備をする必要があります。まず、フレーム中 2 投目で何本のピンを倒せばスペアーになるのかを計算する必要があります。なぜなら、フレーム中 2 投目でスペアーが出るかどうかによって、その直後の投球における得点の倍率が変化する（スペアーが出ていれば 2 倍、そうでなければ 1 倍）からです。何本のピンを倒せばスペアーになるのか、それは、残ったピンの本数に一致します。スペアーを出すために倒すべきピンの数、および、残ったピンの本数は、直前のフレームの状態（S）には依存しません。そのため、スペアーを出すために倒すべきピンの数は直前のフレームの状態についての場合分け（条件分岐）（「プログラム（暫定）の翻訳」で言うところの「条件分岐 4」）の内側に書かず、外側に書きます。今回は、スペアーを出すために倒すべきピンの数を求める処理は、その場合分け（条件分岐）の外側ならば、その場合分け（条件分岐）の前に書いても、後に書いてもよいでしょう。そして、その内容は、一度に並べられるピンの総数である 10 から倒したピンの数を引き算するようなものであれば結構です。すなわち、

```
R ← 10-P
```

です。もし、この行を場合分け（条件分岐）の前に書くのなら、プログラムは

```
R ← 10-P
if S=0 then
   T ← T+P
else if S=1 then
   T ← T+2P
else if S=2 then
   T ← T+2P
else （すなわちS=3なら）
   T ← T+3P
endif
```

となりますし、そうではなく、場合分け（条件分岐）の後に書くのなら、プログラムは

```
if S=0 then
   T ← T+P
else if S=1 then
   T ← T+2P
else if S=2 then
   T ← T+2P
else （すなわちS=3なら）
   T ← T+3P
```

```
endif
R ← 10-P
```

となります。

今回はどちらでも正解ですし、どちらを選択するのかは好みの問題です。なぜどちらでも正解なのかと言うと、それは、得点の計算（場合分け（条件分岐）がある部分）と、スペアーを出すために倒すべきピンの数の計算は互いに依存関係にはないからです。変数 P が共通で用いられていますが、いずれの処理においても読み出しにしか用いられていませんので、処理の順番が変わったところで結果は変わりません。処理の順番を変えてはいけないのは、例えば、ある変数を共通に使い、一方の処理は読み出しに用い、他方の処理は書き込みに用いる場合です。例えば、初期値として変数 A が 3、変数 B が 6 だとすると、

```
B ← A+1
C ← B+2
```

という処理を行うと、1 行目によって変数 B には値 4 が代入され、2 行目によって変数 C には値 6 が代入されますが、処理の順番を入れ替えて

```
C ← B+2
B ← A+1
```

とすると、1 行目によって変数 C には値 8 が代入され、2 行目によって変数 B には値 4 が代入されることになり、異なる結果になってしまいます。

本当は、処理の順番を入れ替えても処理結果が変化しない場合は、分かりやすい方を採用するのですが、今回はどちらでも分かりやすさに差はないでしょうから、その意味でどちらでもよく、どちらを選択するのかは好みの問題だと述べることにします。実際、プログラミングの正解は一つだけということは滅多になく、同じ処理を行わせる場合でも人によって書くプログラムの具体的な内容は異なるものです。そういうわけで、どちらでもよいのですが、ここでは場合分け（条件分岐）の後に書くことにします。

プログラム（暫定）

```
if S=0 then
   T ← T+P
else if S=1 then
   T ← T+2P
else if S=2 then
   T ← T+2P
else （すなわちS=3なら）
   T ← T+3P
endif
R ← 10-P
```

プログラム（暫定）の翻訳

条件分岐 4-A: もし、直前のフレームがオープン（S=0）なら

　合計点数を倒れたピンの数の分だけ増加させる

条件分岐 4-B: もし、直前のフレームがスペアー（S=1）なら

　合計点数を、倒れたピンの数を 2 倍した分だけ増加させる

条件分岐 4-C: もし、直前のフレームがストライクでかつその前のフレームがストライクではない（S=2）なら

合計点数を、倒れたピンの数を 2 倍した分だけ増加させる

条件分岐 4-D: （else）もし、直前のフレームがストライクでかつその前のフレームもストライク（S=3）なら

合計点数を、倒れたピンの数を 3 倍した分だけ増加させる

条件分岐 4 終了

スペアーを出すために倒すべきピンの数（倒れずに残ったピンの数）は 10 から倒したピンの数を引いたものとする

処理 B はこれで終わりではありません。最後に、次の投球がフレーム中何番目であるのかを更新する必要があります。なぜなら、フレーム中 2 投目は直前の投球がストライクでもスペアーでもないことが分かっているために、直前の投球がストライクなのかスペアーなのか分からないフレーム中 1 投目と比べて、得点の倍率の求め方が異なるからです。このための処理は、

```
I ← 2
```

と書くだけでよいのです。これも、場合分け（「プログラム（暫定）の翻訳」で言うところの「条件分岐 4」）の外側ならどこでもよいのですが、ここでは最後に書くことにします。

プログラム

```
if S=0 then
   T ← T+P
else if S=1 then
   T ← T+2P
else if S=2 then
   T ← T+2P
else （すなわちS=3なら）
   T ← T+3P
endif
R ← 10-P
I ← 2
```

プログラムの翻訳

条件分岐 4-A: もし、直前のフレームがオープン（S=0）なら

合計点数を倒れたピンの数の分だけ増加させる

条件分岐 4-B: もし、直前のフレームがスペアー（S=1）なら

合計点数を、倒れたピンの数を 2 倍した分だけ増加させる

条件分岐 4-C: もし、直前のフレームがストライクでかつその前のフレームがストライクではない（S=2）なら

合計点数を、倒れたピンの数を 2 倍した分だけ増加させる

条件分岐 4-D: （else）もし、直前のフレームがストライクでかつその前のフレームもストライク（S=3）なら

合計点数を、倒れたピンの数を 3 倍した分だけ増加させる

条件分岐 4 終了

スペアーを出すために倒すべきピンの数（倒れずに残ったピンの数）は 10 から倒したピンの数を引いたものとする

次の投球をフレーム中の 2 番目のものとする

今回は、問題文によれば、処理 B では「T、R、I を変更する」となっており、直前のフレームの状態を表す変数 S については手を加えないことになっていますので、これで処理 B は終わりです。

処理 B

```
if S=0 then
   T ← T+P
else if S=1 then
   T ← T+2P
else if S=2 then
   T ← T+2P
else （すなわちS=3なら）
   T ← T+3P
endif
R ← 10-P
I ← 2
```

ところで、場合分け（条件分岐）において、S=1 の場合と S=2 の場合では完全に処理の内容が同じであり、変数 S の値は 0、1、2、3 のいずれかでしかありえないことを利用して、場合分けの方法を少しだけ変更して、

```
if S=0 then
   T ← T+P
else if S=3 then
   T ← T+3P
else （すなわちS=1またはS=2なら）
   T ← T+2P
endif
R ← 10-P
I ← 2
```

とすれば、場合を 1 つ（2 行）減らすことができます。行数を減らせば、それだけプログラムを見通しやすくなります（例えば、プログラムの編集画面で、他の行を参照するためにスクロールする必要が少なくなります）が、この工夫によって、変数 S の内容が順番には並ばなくなり（0 の次がいきなり 3 になっており、その後が 1、2 になっている）、それを好ましくないと思う人もいることでしょう。この工夫によって 2 行しか減らないのであれば、無理してその工夫を適用する必要がないと考えることもできます。これも好みの問題でしょう（どちらでも正解です）。

本当は、「S=1」の場合と「S=2」の場合を、**論理和**（～または～）を用いて「S=1 or S=2」に結合すれば、同じように行数を減らすことができたのです。

```
if S=0 then
   T ← T+P
else if S=1 or S=2 then
   T ← T+2P
else （すなわちS=3なら）
   T ← T+3P
endif
R ← 10-P
I ← 2
```

しかし、今回の擬似コード（実際にコンピューターに実行されることはなく、単にプログラマーが案を練るために書かれることがあるもの）には論理和の機能が用意されていないため、論

理和を使用することはできません。もっとも、変数 S の値が 0、1、2、3 のいずれでしかありえないことを利用して、「S=1 or S=2」の代わりに「1≦S≦2」とすれば、「1≦S≦2」に似たような例が問題文中にありますのでよいでしょう。

処理 B

```
if S=0 then
   T ← T+P
else if 1≦S≦2 then
   T ← T+2P
else （すなわちS=3なら）
   T ← T+3P
endif
R ← 10-P
I ← 2
```

次は処理 C。処理 C はフレーム中の 1 投目でストライクを出さず、2 投目が発生したときに使用されます。現在の投球の直前（ここではすなわち、フレーム中の 1 投目）ではストライクが出ていないことが分かっていますから、得点の倍率に関与しうるのは、さらにその前の投球（直前のフレーム）がストライクであるかどうかだけです。前のフレームでスペアーが出ているかどうかは関係ありません。なぜなら、スペアーによって得点の倍率が上がるのは、そのスペアーが出た投球の次の投球のみであり、それはフレーム中の 1 投目なので、フレーム中の 2 投目までは及ばないからです。また、直前よりも前のフレームでストライクやスペアーが出ているかどうかににについては全く関与しません。なぜなら、そのようなストライクやスペアーも、どんなに長くてもせいぜい現在のフレーム中の 1 投目までしか得点の倍率が上がる効果が及ばないからです。

そのため、直前のフレームの状態を表す変数 S については、値が 0 の場合と値が 1 の場合はいずれも直前のフレームでストライクが出ていないことで共通のため、これら 2 つの場合では互いに得点の計算処理が同じであることが期待され、値が 2 の場合と値が 3 の場合はいずれも直前のフレームでストライクが出ていることで共通のため、これら 2 つの場合では互いに得点の計算処理が同じであることが期待されます。直前のフレームでストライクが出ていない場合（S=0 または S=1）は、ストライクやスペアーによる得点の倍率の向上が現在のフレーム中 2 投目の得点に及ぶことはありませんので、得点の倍率は通常通りの 1 です。一方、直前のフレームでストライクが出ている場合（S=2 または S=3）は、直前のフレームで出たストライクによる得点の倍率の向上だけが現在のフレーム中 2 投目の得点に及びますので、得点の倍率は通常より 1 だけ上がった 2 です。というわけで、得点の計算だけを記したプログラムは以下のようになります。

プログラム（暫定）

```
if S=0 then
   T ← T+P
else if S=1 then
   T ← T+P
else if S=2 then
   T ← T+2P
else （すなわちS=3なら）
```

```
    T ← T+2P
endif
```

プログラム（暫定）の翻訳

（説明の都合上条件分岐の番号を変えてありますが､プログラムそのものには影響はありません）

条件分岐 5-A: もし、直前のフレームがオープン（S=0）なら

合計点数を倒れたピンの数の分だけ増加させる

条件分岐 5-B: もし、直前のフレームがスペアー（S=1）なら

合計点数を倒れたピンの数の分だけ増加させる

条件分岐 5-C: もし、直前のフレームがストライクでかつその前のフレームがストライクではない（S=2）なら

合計点数を、倒れたピンの数を 2 倍した分だけ増加させる

条件分岐 5-D: （else）もし、直前のフレームがストライクでかつその前のフレームもストライク（S=3）なら

合計点数を、倒れたピンの数を 2 倍した分だけ増加させる

条件分岐 5 終了

ただ、場合分け（条件分岐）のうち後ろの方は処理の内容が完全に同じですので、else に追いやってまとめてしまうこともできますし､今回の場合はその方が見やすくてよいでしょう(もっとも、この工夫を用いなくても正解でしょう)。

プログラム（暫定）

```
if S=0 then
    T ← T+P
else if S=1 then
    T ← T+P
else （すなわちS=2またはS=3なら）
    T ← T+2P
endif
```

プログラム（暫定）の翻訳

条件分岐 5-A: もし、直前のフレームがオープン（S=0）なら

合計点数を倒れたピンの数の分だけ増加させる

条件分岐 5-B: もし、直前のフレームがスペアー（S=1）なら

合計点数を倒れたピンの数の分だけ増加させる

条件分岐 5-C: （else）もし、直前のフレームがストライク（S=2 または S=3）なら

合計点数を、倒れたピンの数を 2 倍した分だけ増加させる

条件分岐 5 終了

あるいは、変数 S の値は 0、1、2、3 のいずれかでしかありえないことを利用し、S=0 の場合と S=1 の場合も「S≦1」でひとまとめにすることもできます（この工夫を用いなくても正解で

しょう)。

プログラム（暫定）

```
if S≦1 then
   T ← T+P
else （すなわちS=2またはS=3なら）
   T ← T+2P
endif
```

プログラム（暫定）の翻訳

条件分岐 5-A: もし、直前のフレームがオープン（S=0）またはスペアー（S=1）なら

合計点数を倒れたピンの数の分だけ増加させる

条件分岐 5-B: （else）もし、直前のフレームがストライク（S=2 または S=3）なら

合計点数を、倒れたピンの数を 2 倍した分だけ増加させる

条件分岐 5 終了

現在のフレーム中 2 投目が終了すれば、これで現在のフレームが確実に終了しますので、直前のフレームの状態を表す変数 S の値を更新しなければなりません（なぜなら、直前のフレームの状態が得点の倍率に影響を及ぼすからです)。ところが、変数 S の値がどうなるのかは、少なくとも現在のフレーム中 2 投目でスペアーが出たかどうかにかかっています。

ここで､更新前の変数 S の値も更新後の変数 S の値に影響を及ぼすのかを考えてみましょう。なぜこのようなことを考えるかというと、この如何によっては、変数 S の値を更新する処理を場合分け（条件分岐）の内部に置かなければならないのか外部に置いた方がよいのかが変化するからです。更新前の変数 S の値が更新後の変数 S の値に影響を及ぼす場合は、変数 S の値を更新する処理を場合分け（条件分岐）の内部に置かなければなりませんが、そうでなければ、変数 S の値を更新する処理を場合分け（条件分岐）の外部に置くことができます（場合分け（条件分岐）の内部に置くこともできるのですが、場合の個数の分だけ同じ処理を書かなければならないので疲れるだけです)。

ところが、フレーム中 2 投目があるということは、現在のフレームではストライクが出ていません。それは、次のフレームにとっては「直前のフレームではストライクが出ていない」ということになります。また、現在のフレームでストライクが出ていませんので、直前のフレームでストライクが出たかどうかは、次のフレーム以降における得点の倍率には関係がありません。なぜなら､次のフレームからすれば､「直前のフレームはオープン（S=0）かスペアー（S=1）のいずれか」でしかないからです。そのため、変数 S の値を更新する処理を更新前の変数 S の値に応じて場合分け（条件分岐）をする必要はなく､変数 S の値を更新する処理は場合分け（条件分岐）の外部に置くことができます。

変数 S の値を更新する処理の具体的な内容についてですが、これは、倒したピンの数が、スペアーを出すために倒すべきピンの数（フレーム中 1 投目で倒れずに残ったピンの数）に一致

していれば、それは残ったピンを全て倒してスペアーを出したことを意味しますし、それとは逆に、倒したピンの数が、スペアーを出すために倒すべきピンの数（フレーム中 1 投目で倒れずに残ったピンの数）よりも少なければ、それは残ったピンを全て倒しきれず、スペアーすら出なかったことを意味します。従いまして、倒したピンの数がスペアーを出すために倒すべきピンの数（フレーム中 1 投目で倒れずに残ったピンの数）以上かどうかで場合分け（条件分岐）をすればよいでしょう。そのための処理の具体的な内容は以下のようになります。

```
if P≧R then
   S ← 1
else （すなわちP<Rなら）
   S ← 0
```

もちろん、以下のように、残ったピンを全て倒しきれなかった場合を最初に持ってきても結果は変わりません（従って、これも正解です）。

```
if P<R then
   S ← 0
else （すなわちP≧Rなら）
   S ← 1
```

ところで、倒したピンの数（P）がスペアーを出すために倒すべきピンの数（フレーム中 1 投目で倒れずに残ったピンの数）（R）よりも上回る（P>R）ことはありえませんので、以下のように、「P≧R」を「P=R」に、「P<R」を「P≠R」に置き換えても結果は変化しないため、これらも正解です。

```
if P=R then
   S ← 1
else （すなわちP<Rなら）
   S ← 0
```

または

```
if P≠R then
   S ← 0
else （すなわちP=Rなら）
   S ← 1
```

最後に、フレーム中 2 投目が終われば、次は新しいフレーム中 1 投目が来ますので、その投球がフレーム中何番目なのかもそのように更新する必要があります。

```
I ← 1
```

この処理も、直前のフレームの状態（S）には依存しませんので、場合分け（条件分岐）の外部に置けばよいでしょう。ここでは処理 C の最後に置いておきます。

以上により、処理 C の具体的な内容は、例えば以下のようになります。

処理 C

```
if S≦1 then
   T ← T+P
else （すなわちS=2またはS=3なら）
   T ← T+2P
endif
if P≧R then
   S ← 1
else （すなわちP<Rなら）
   S ← 0
I ← 1
```

もちろん、これと一字一句同じでなくても、これと同じ結果が得られるものであれば正解で

しょう。

解答例を以下にまとめて示します。

処理 B

```
if S=0 then
    T ← T+P
else if 1≦S≦2 then
    T ← T+2P
else （すなわちS=3なら）
    T ← T+3P
endif
R ← 10-P
I ← 2
```

処理 C

```
if S≦1 then
    T ← T+P
else （すなわちS=2またはS=3なら）
    T ← T+2P
endif
if P≧R then
    S ← 1
else （すなわちP<Rなら）
    S ← 0
I ← 1
```

おそらく、いきなりプログラムを書くのは難しいでしょうから、問題文には「A の計算手順を参考にして」という句を加えたのでしょう。確かに、処理 A、処理 B と処理 C は場合分け（条件分岐）の仕方がほとんど同じです。解答例では場合の個数ができるだけ少なくなるようにしましたが、処理 A とできるだけ同じように場合分けをするのなら、処理 B と処理 C は以下のようになりましょう。

処理 B

```
if S=0 then
    T ← T+P
else if S=1 then
    T ← T+2P
else if S=2 then
    T ← T+2P
else （すなわちS=3なら）
    T ← T+3P
endif
R ← 10-P
I ← 2
```

処理 C

```
if S=0 then
    T ← T+P
else if S=1 then
    T ← T+P
else if S=2 then
    T ← T+2P
else （すなわちS=3なら）
    T ← T+2P
```

```
endif
if P≧R then
   S ← 1
else （すなわちP<Rなら）
   S ← 0
I ← 1
```

ただ、このような真似事は、やはりプログラミングに精通していないと難しいでしょう。よく、「よくやり方を見て真似すれば初心者でもできるはずだ」のようなことを言う人を見かけますが、実際には、慣れていないと（工夫して発展させることはもちろんのこと）真似をすることさえも難しいのです。既に慣れている人で、「多くの人にとっては慣れていないと真似することさえも難しい」という現状を知らない人は残念ながら少なくない（むしろ大多数か）ようです。その意味で、この問題は（わずか 4 ヶ月で情報理論を勉強する人にとっては）難しすぎると考えられます。

実は、このような問題は、私のようにプログラミングに慣れている人にとっても難しいのです。私のようなプログラマーはプログラムを作るときにはコンピューターを用いて作業するのが普通で、何か誤りを犯せば、コンピューター（のソフトウェア）がすぐに指摘してくれるのもまた普通です。従って、いきなり文法的にも語法的にも、その他の規則としても正しいプログラムを書かなければならないということはなく、コンピューターが何かエラーメッセージや警告メッセージ[5]などを表示したら、そのメッセージを参考にしてプログラムを修正すればよいだけの話です。しかし、そのようなコンピューターを、試験会場では頼りにできないのです。私たちは試験会場で、一発で正しいプログラムを書かなければならないのです。熟練したプログラマーでもコンピューターに文法などの規則のチェックを頼っている人は少なくありません[6]ので、その点ではつらいところです。ここで、ざっといくつかのプログラミングの資格試験を見てみたのですが、多くが多肢選択解答形式（選択肢の中から正しいと思われるものを選ぶ形式）でしたし、記述式も時々ありましたが、その問題のうち多くはコンピューターを用いて作業することになっていました。それは単に採点が面倒だからというよりも、熟練したプログラマーでも誤りを避けられず、コンピューターの助けなしにはその誤りをくまなく見つけ出すのも困難だからでしょう。

「情報科学」の予習——Ruby とは？

さて、東京大学の冬学期には「情報科学」というプログラミングの授業が開かれますので、せっかくですから、その「情報科学」で使用されるプログラミング言語・Ruby を使用して、今回の問題で試用されたプログラムを実際に作ってみましょう。

Ruby は小規模なプログラムの開発に適していると言われているプログラミング言語です。プログラミング言語は実に数千個あると言われていますが、プログラミング言語がどれだけ有名か（あるいは、どれだけ実際に使用されているか）を示す統計データの一例として、世界の熟

[5] 「警告」と言うと、それに従わないと命を落とすなどの物々しい響きがありますが、これは英語「warning」の（あまり好ましくない）訳であり、実際には（警告されたことはできるだけ直した方がよいものの）あまり深刻なものではありません。

[6] 実は 10 年以上プログラミングを行ってきた私も、50 行程度のプログラムですら、一度もエラー（コンパイル時エラーや実行時エラーなど）を出すことなく完成することは滅多にないのです。

練したプログラマーがどのプログラミング言語を使用しているのかを示している TIOBE Programming Community Index for September 2009（Web ページ）からプログラミング言語の使用割合を引用してみます（ただし、用途の異なるプログラミング言語が混ざっていることに注意する必要があります)。

順位	プログラミング言語の名前	シェア
1 位	Java	19.383%
2 位	C	16.861%
3 位	PHP	10.156%
4 位	C++	9.988%
5 位	(Visual) Basic	9.196%
6 位	Perl	4.528%
7 位	C#	4.186%
8 位	Python	3.930%
9 位	JavaScript	2.995%
10 位	Ruby	2.377%
11 位	Delphi	1.972%
12 位	Pascal	0.961%
13 位	Lisp / Scheme	0.842%
14 位	PL / SQL	0.819%
15 位	SAS	0.781%
16 位	ABAP	0.705%
17 位	D	0.588%
18 位	Objective-C	0.585%
19 位	Lua	0.507%
20 位	MATLAB	0.506%

プログラミング言語といっても、その用途はさまざまです。例えば、オフィススイート製品である Microsoft Office（制作・販売: Microsoft）やデザイナー・印刷業者用スイート製品である Adobe Creative Suite（制作・販売: Adobe Systems）などのような、一般人に販売するような大規模なスタンドアロン型（単独で動かすという意味）ソフトウェアは、C、C++、Java、Visual Basic で開発されていることが多いようです。実際、Microsoft Office や Adobe Creative Suite は C++で開発されているようです［要出典］[7]。一方、普段のちょっとした仕事を自動化するのには、主に Perl、Python や Ruby、そして、各アプリケーションソフトウェアのマクロ言語（例えば、Microsoft Office などで使用できる Visual Basic for Applications）が使用されています。インターネットで使用するソフトウェアには、主に Java、JavaScript、Perl、PHP、Python、Ruby が使用されています。ちなみに、SAS は統計解析を行うためのソフトウェアの名前、MATLAB は数値計算を行うためのソフトウェアです。

Ruby はここ数年利用者が急増しているプログラミング言語ですが、もはや有名な領域に達したかというと少々微妙なところです。Ruby に似たプログラミング言語のひとつに Python があります。実際、Ruby は誕生時に Python の影響を強く受けています。Python は Ruby よりも有名であり、異なる用途にも目を向ければ、C、C++や Java の方が遥かに有名です。にもかかわらず、東京大学は 2006 年以降に「情報科学」で使用するプログラミング言語としては Ruby を選択しました。その理由はさまざま考えられましょうが、私は、そのもっとも大きな理由は、Ruby が日本生まれだということにあると思います。

コンピューターを長く使用されている方はご存知だと思いますが、コンピューターに関連の

[7] 出典は…忘れました。

あるもののうち中心を担っているものはほとんどが海外、特にアメリカ合衆国の生まれです。例えば、パソコン（個人用コンピューター）の頭脳とも言える CPU は多くが Intel 製か Advanced Micro Devices（AMD）製ですが、Intel も AMD もアメリカ合衆国の企業です。基本システム（operating system（OS））の代表としては Apple の Mac OS、Microsoft の Windows、Sun Microsystems の Solaris、オープンソースソフトウェア[8]の Linux が挙げられますが、いずれもアメリカ合衆国生まれです。ワープロや表計算の機能を含むオフィススイートも、今や Microsoft Office が代表格で、最近ではオープンソースソフトウェアの OpenOffice.org が注目を集めていますが、Microsoft Office はアメリカ生まれ、OpenOffice.org は生まれこそドイツですが、今は主にアメリカ合衆国育ちです[9]。ワープロソフト（富士通の OASYS や日本電気（NEC）の文豪などのワープロ専用機を除きます）では、昔は日本製で有名なものにジャストシステムの一太郎や管理工学研究所の松がありましたが、松は終了し、一太郎は一応生き残っているものの、かなりマイナーな存在になってしまいました。パッと思いつくもののほとんどがアメリカ合衆国か、あるいは台湾などの物価の比較的安い国で作られており、日本の製品はあまり思い浮かばれないのです。

そのような中で例外的な存在なのが家庭用ゲーム市場です。（他の種類の）コンピューターとは異なり、こちらは日本の製品が世界的な規模で販売されています。ハードウェアでは、任天堂のファミリーコンピューター（通称: ファミコン）（英題: Nintendo Entertainment System、NES）に始まり、今日ではソニーコンピューターエンターテインメントのプレイステーション 2 やプレイステーション 3、任天堂のWii（ウィー）は日本の製品でありながらもアメリカ合衆国を含めて世界規模でメジャーな存在であり、日本の製品でないものと言えば、今日では Microsoft の Xbox（エックスボックス）360 くらいしかありません。ソフトウェアでも、任天堂のスーパーマリオ（英題: Super Mario）シリーズ、カプコンのバイオハザード（英題: Resident Evil）シリーズ、スクウェア・エニックスのファイナルファンタジー（英題: Final Fantasy）シリーズ、ポケモンのポケットモンスター（英題: Pokémon）シリーズ、コナミのメタルギア（英題: Metal Gear）シリーズなどは日本の製品でありながら、海外でも高い人気を誇っています。GameSpot[10]、GameSpy[11]や IGN[12]などの海外のゲーム批評 Web サイトでも、日本の製品がよく登場し、高評価を得るものも少なくありません。世界でもよく売れている製品は、今日ではかなり大規模なものにまで成長しましたし、経済的にも日本を潤してくれるものになっています。

ところが、そのようなものが、コンピューターの業界にはほとんどありません。しかし、その兆しとなりそうなものがいくつかあり、そのひとつが Ruby だと考えられます。Ruby は 10 年以上前ではきわめてマイナーなプログラミング言語で、Python には大きく水をあけられていました。しかし、Ruby を使用して制作された Web アプリケーションフレームワーク・Ruby on Rails が登場してからは Ruby の知名度が急激に上がるようになりました。そうなると、教育で

[8] 内部仕様が公開されており、内部仕様も含めてある程度自由に使用できるソフトウェアのことです。
[9] 元々OpenOffice.org は「StarOffice」という名前で、ドイツの StarVision から販売されていましたが、Sun Microsystems が StarVision を買収し、StarOffice のソースコード（内部仕様）を公開してしまいました。
[10] http://www.gamespot.com/
[11] http://www.gamespy.com/
[12] http://www.ign.com/

も Ruby を導入して、早くから Ruby を知ってもらえば、Ruby を使用するプログラマーが増え、Ruby は日本が世界に誇れるプログラミング言語のひとつになる、というのが、授業で Ruby を採用した理由ではないかと思います。実際、人間は言語に限らず、多くのものについて最初に触れたものを半永久的に好み、それに基づいて思考する（刷り込まれる）傾向にあるようですから、私はそのようなことは理由として十分にあると考えています。

ちなみに、私もプログラマーなのですが、使用した言語は、最初は日本電気（NEC）の PC-9800 シリーズに搭載されていた N88-BASIC(86)[13]というもので、その後 C、Java（学習が中途半端な感じもしますが）を通って、現在では主に C#でプログラミングをしています。Python の方は、CG 制作ソフトウェア・Shade（制作・販売: イーフロンティア）で自動処理に利用できますので、一度は使用してみたいと思ったのですが、Ruby については、困ったことに利用する積極的な理由がないのです。ですから、授業で Ruby を扱うと知ったときには少々困りました。もちろん、さまざまなプログラミング言語に触れれば新しい発見が生まれるかもしれませんが、まだある程度の規模のソフトウェアを完成させたことがない身としては、とりあえず一つの言語をある程度極めたいと思っているのです。…そうは思っても、取り扱われるプログラミング言語が変わるわけではありませんので、しばらく Ruby にお付き合いすることにしましょう。

Ruby の導入

これから Ruby を用いてプログラミングを行うのですが、お使いのシステムによっては、Ruby が最初から使えたり、そうでなかったりします。基本システムの一つに、Macintosh に最初から導入（インストール）されている Mac OS（制作・販売: Apple）がありますが、現在の Mac OS X には Ruby でプログラミングを行うためのソフトウェア（**ソフトウェア開発キット**（software development kit、SDK））が最初から導入されており、すぐに使うことができますが、Windows には Ruby の SDK が入っていませんので、Ruby の SDK を別途導入する必要があります。

Ruby の SDK には何種類かありますが、Windows 用の Ruby の SDK として有名なものには以下のようなものがあります。

- ActiveScriptRuby[14]

  全自動で Ruby の SDK を導入してくれますので、最も楽でしょう。通常はこちらをお勧めするところですが、Ruby の最新版であるバージョン 1.9.1 に対応したものがまだ用意されていません（2009 年 9 月現在）。

- Ruby-mswin32[15]

  こちらは Ruby の最新版であるバージョン 1.9.1 に対応したものが用意されています（2009 年 9 月現在）。ただし、導入作業としては、対応するパッケージ（ZIP 書庫ファイル）をダウンロードしてそれを解凍した後、プログラムの実行時に Ruby の実行ファイルがある場所を参照して、Ruby の実行ファイルが見つかるようにするために、

[13] 1980 年代に流行したもので、今となってはかなり古いのですが、なぜか私の家には古いコンピューターがいくつかあったのでした。
[14] http://arton.hp.infoseek.co.jp/indexj.html
[15] http://www.garbagecollect.jp/ruby/mswin32/ja/

Ruby の実行ファイルが格納されているディレクトリー（ファイルフォルダー）を環境変数 PATH に登録する必要があります。環境変数 PATH の設定方法については、Windows 95／98／Me では、Windows が導入されているディスクドライブ（多くの場合は DOS/V 機で、そこでは C:になっているはずです。日本電気（NEC）の PC-9800 シリーズの場合は A:になっていることでしょう）のルートディレクトリーにあるファイル「AUTOEXEC.BAT」で設定することになっています。一方、Windows 2000／xp／Vista／7 では、コントロール パネルを開き、「システム」を選択して「システムのプロパティ」を開いた後、タブ「詳細設定」を選択し、ボタン「環境変数」を押すことで設定を変更できます。

もしかしたら妙に感じられたかもしれませんが、ここではバージョン番号の表記が「○.△.□」（例　「1.8.7」、「1.9.1」）のように、小数点が 2 個使用されています。バージョン番号には、いくつかの英数字とそれらを区切る小数点やハイフンなどで構成されているものが少なくありません（例　1.0.0.1-2)。このようなバージョン番号には次のような意味があります。最初の英数字はメジャーバージョン番号と呼ばれているもので、ソフトウェアに大きな変更（機能向上など）が生じた場合に変更するものです。通常は 1 ずつ増えていきます（0→1→2→3→…）。このメジャーバージョン番号は 0 で始まることがあれば 1 で始まることもあります。メジャーバージョン番号が 0 のソフトウェアは未熟な段階にあると捉えられる傾向にありますが、バージョン番号の付け方はソフトウェアを制作・製作する人（プログラマーやパブリッシャーなど）の勝手ですので、意味が必ず固定しているわけではありません。一方、メジャーバージョン番号よりも後にある英数字はマイナーバージョン番号であり、ソフトウェアに比較的小さな変更（欠陥の訂正（バグフィックス）など）が生じた場合に変更されるものです。マイナーバージョン番号が複数置かれる場合も少なくありません（例えば、「1.0.0.1-2」のうち、マイナーバージョン番号は、「0.0.1-2」を構成する「0」、「0」、「1」、「2」であると考えられます）が、先に置かれる英数字ほど、ソフトウェアの比較的大きな変更と関係しています。しかし、マイナーバージョン番号もまた付け方はソフトウェアを制作・製作する人の勝手ですので、意味が必ず固定しているわけではありません。また、一つのマイナーバージョン番号が 2 桁以上になることもあります。例えば、バージョン「0.9」（零点九）の次を「1.0」（一点零）ではなく「0.10」（零点十）としたり、「0.99」（零点九十九）の次を「1.00」ではなく「0.100」（零点百）としたりすることも少なくありません。このような場合は、例えば、「0.10」は「0.11」の前のバージョンであることはあっても、「0.2」の前のバージョンではありません。このようなバージョン番号の付け方も、やはりソフトウェアを制作・製作する人の勝手ですので、誰でも同じというわけではありません。

ここでは、導入が比較的簡単な ActiveScriptRuby について説明します。

ActiveScriptRuby の導入

注意　ActiveScriptRuby は Windows 用です。他のシステムで Ruby を使用する場合は、他のパッケージを使用してください。

まずは、インターネットで ActiveScriptRuby を入手します。ActiveScriptRuby の Web ページは以下の場所にあります（2009 年 9 月現在）。

http://arton.hp.infoseek.co.jp/indexj.html

ActiveScriptRuby の Web ページは他にも

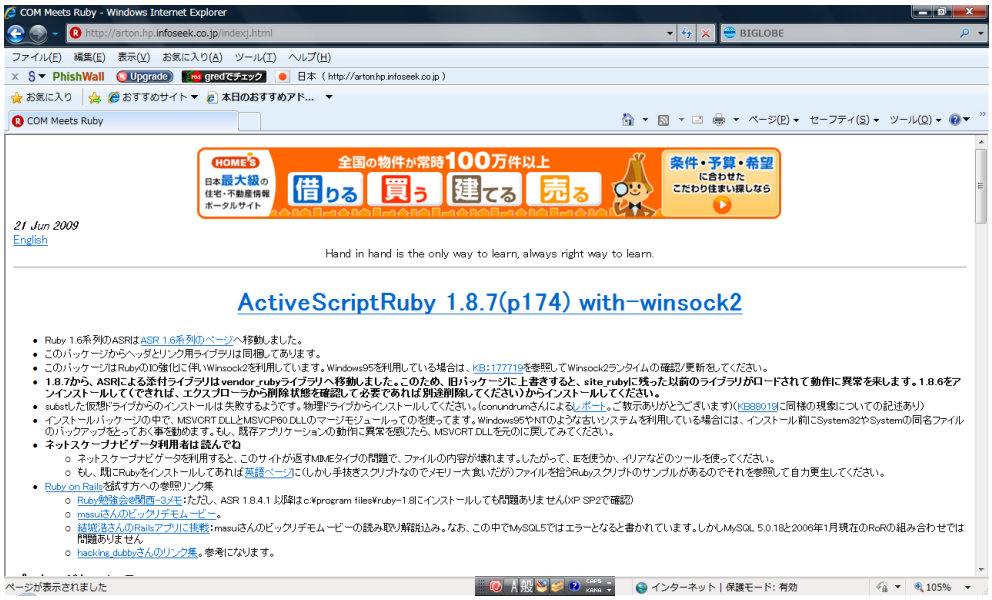
http://www.geocities.co.jp/SiliconValley-PaloAlto/9251/ruby/

がありますが、こちらは古いので使わないようにしましょう。Google などの Web 検索エンジンを使用したり、Ruby の公式 Web ページ[16]の「ダウンロード」 - 「各環境用バイナリ」に掲載されているリンクを使用したりすると、後者の古い Web ページに誘導されやすくなってしまいますので、ご注意ください。新しい方はバージョン 1.8.7 になっています（2009 年 9 月現在）。

さて、Web ページ

http://arton.hp.infoseek.co.jp/indexj.html

を開くと、以下のような画面が表示されます。



（fig_WebPageOfActiveScriptRuby_1.tif）

そこに表示されている説明書きをよく読んで、ActiveScriptRuby を使用するためには何が必要なのかを確認しておきましょう…と言いたいところですが、難しくてよく分からない方もいらっしゃるでしょうから、ここでは以下に簡単にまとめておきます。

まずは、何はともあれ、Windows は Service Pack を適用するなどして、できるだけ新しくしておきましょう。

Windows Me、2000、xp、Vista、7 またはそれ以降の Windows では、ActiveScriptRuby を使用するために別のソフトウェアやデータが必要になることはありません。ActiveScriptRuby 本体[17]をダウンロードして、これを開けば OK です。そこで、この部分は一部読み飛ばしても構いません。

---

[16] 「オブジェクト指向スクリプト言語 Ruby」 http://www.ruby-lang.org/ja/

[17] http://arton.hp.infoseek.co.jp/ActiveRuby.msi

一方、Windows 95、Windows 98（Second Edition を含む）、NT 4.0 では、ActiveScriptRuby を使用するために別のソフトウェアを用意する必要があります。例えば、ActiveScriptRuby の導入に Windows Installer を使用することになっていますので、ActiveScriptRuby を導入する前に、Windows Installer を入手して導入する必要があります。Windows Installer は下記の Web ページから入手できます（2009 年 9 月現在）。対応している範囲内で、できるだけ新しい（バージョン番号が大きいもの）ものを導入しましょう（Windows Me、2000、xp、Vista、7 では、ActiveScriptRuby を使用するだけなら、Windows Installer のダウンロード・導入は不要です）。

バージョン 2.0（Windows 95／98／Me）

http://www.microsoft.com/downloads/details.aspx?FamilyID=cebbacd8-c094-4255-b702-de3bb768148f

バージョン 2.0（Windows NT 4.0／2000）

http://www.microsoft.com/downloads/details.aspx?FamilyID=4b6140f9-2d36-4977-8fa1-6f8a0f5dca8f

バージョン 3.1（Windows 2000／xp）

http://www.microsoft.com/downloads/details.aspx?FamilyID=889482fc-5f56-4a38-b838-de776fd4138c

バージョン 4.5（Windows xp／Vista）

http://www.microsoft.com/downloads/details.aspx?FamilyID=5a58b56f-60b6-4412-95b9-54d056d6f9f4

Windows Installer をダウンロードしたら、ダウンロードしたものを実行してください。

また、Windows 95 をお使いの場合はさらに、Winsock 2.0 が導入されているかどうかを確認する必要があります。Web ページ

http://support.microsoft.com/kb/177719/ja

を参考にして、

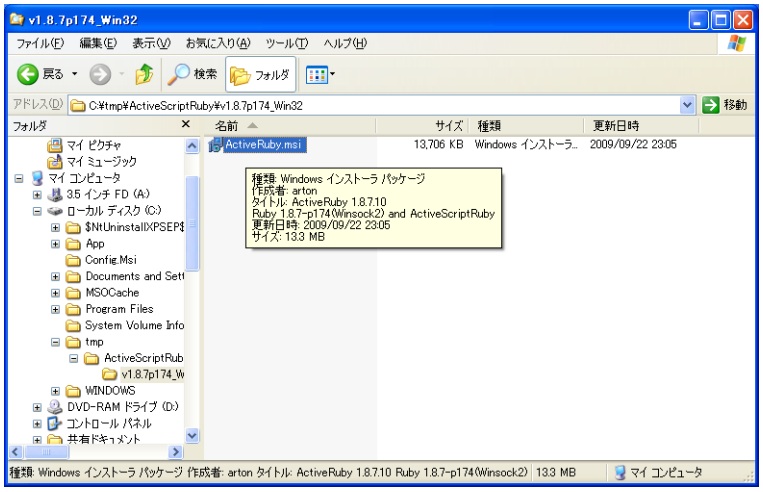
http://download.microsoft.com/download/0/e/0/0e05231b-6bd1-4def-a216-c656fbd22b4e/w95ws2setup.exe

をダウンロードし、実行してください。

ここからは全 Windows 共通（Windows Me、2000、xp、Vista、7 を含む）です。ActiveScriptRuby の本体

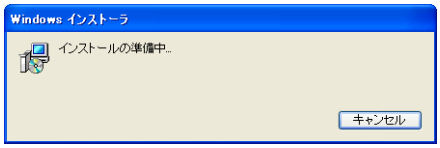
http://arton.hp.infoseek.co.jp/ActiveRuby.msi

をダウンロードし（ActiveScriptRuby の Web ページの最初にあるリンクをクリックしても結構です）、それを実行してください。



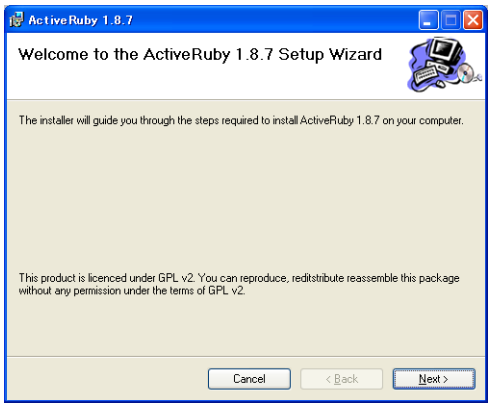
（fig_InstallingActiveRuby187_1.tif）

すると、以下のような画面が表示されます（表示される画面はお使いの基本システム（バージョンを含む）によって若干異なります）。



（fig_InstallingActiveRuby187_2.tif）

この状態でしばらく待つと、さらに以下のような画面が表示されます。



（fig_InstallingActiveRuby187_3.tif）

この画面には以下のようなメッセージが表示されています。

Welcome to the ActiveRuby 1.8.7 Setup Wizard

The installer will guide you through the steps required to install ActiveRuby 1.8.7 on your computer.

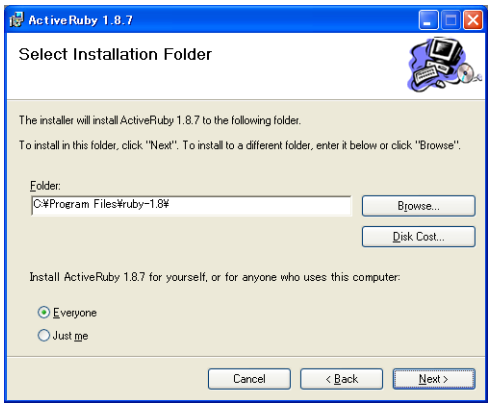
This product is licenced under GPL v2. You can reproduce, redistribute reassemble this package without any permission under the terms of GPL v2.

（日本語訳　ActiveRuby 1.8.7 セットアップウィザードへようこそ。このインストーラー（ActiveRuby 1.8.7 セットアップウィザード）はお使いのコンピューターに ActiveRuby 1.8.7

を導入するために必要な手順を示します。この製品は GPL（GNU General Public License） 第 2 版の下に使用許諾されています。GPL 第 2 版の条項に従えば、（提供者からの明示的な）許可を得ることなく、この製品を複製、再頒布、再構成することができます。）

メッセージが英語で表示されていることに戸惑われるかもしれません。残念なことに、コンピューターの世界では翻訳がかなり遅れているものが多く、英語のままで提供されるソフトウェアや文書が少なくありません。ただ、コンピューターで使用される英語は、語句こそ専門用語が多いので難しく感じられるかもしれませんが、文法は単純なことが多いですので、実は（例えば、英語で書かれた小説を読むよりも）とっつきやすいかもしれません。

画面右下には「Next >」と書かれたボタンがあります。このボタンをクリックすると次に進むことができます。このボタンをマウスでクリックしましょう。



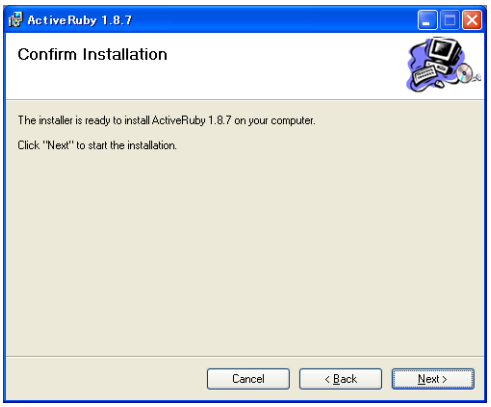
（fig_InstallingActiveRuby187_4.tif）

ここでは主に、ActiveScriptRuby の導入先、すなわち、ActiveScriptRuby の動作に必要なファイルをどこにコピーするのかを指定します[18]。初期状態では、インストール先は「C:¥Program Files¥ruby-1.8」となっていますが、通常は特に変更する必要はありません（ディレクトリーが存在しなくても、このインストーラーが自動的に作成してくれます）。他にも、ここでは、ActiveScriptRuby の利用者をこのコンピューターを利用する全員にまで拡げる（「Everyone」）のか、それとも、現在そのコンピューターを利用している自分自身に限定する（「Just me」）のかを指定することもできますが、これも、初期状態の「Everyone」のままで普通は問題ありません。設定が終わったら、右下にあるボタン「Next >」をクリックします。

なお、実際に導入作業が始まる前なら、ボタン「< Back」をクリックすることで前の手順に戻ることができます。また、ボタン「Cancel」をクリックすると、ActiveScriptRuby を導入せずにこのインストーラーを終了させることができます。

ボタン「Next >」をクリックしたとして、次は以下のような画面が表示されます。

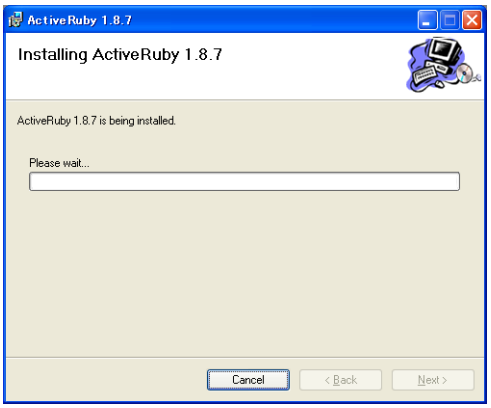
[18] ただし、一部のファイルは Windows のシステムディレクトリーなど別の場所にコピーされることがあります。



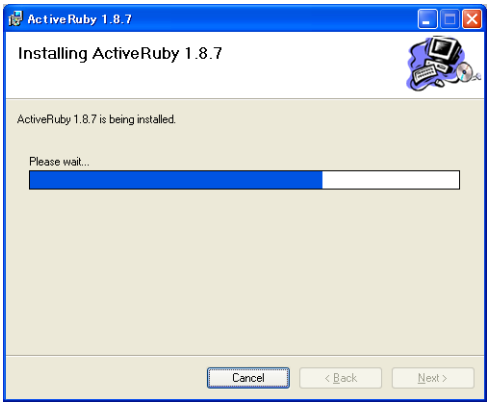
（fig_InstallingActiveRuby187_5.tif）

これは、ActiveScriptRuby の導入に必要な設定が完了し、実際の導入作業に入る準備ができたことを意味します。ボタン「Next >」をクリックすると、導入作業が開始されます。ここでは、ボタン「Next >」をクリックしましょう。

　すると、以下のような画面が表示されます（時間の経過とともに表示される画面が多少変化します）。



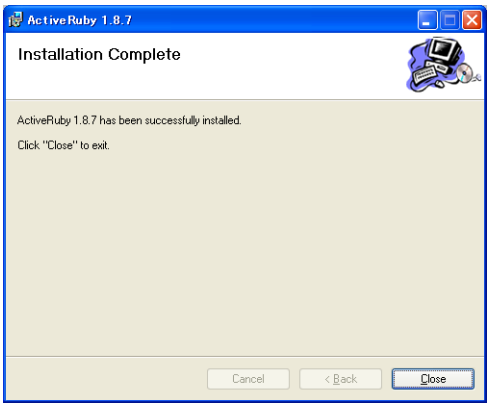
（fig_InstallingActiveRuby187_6.tif）



（fig_InstallingActiveRuby187_7.tif）

真ん中にあるバーは導入作業の進捗（しんちょく）状況を示すものです。作業が進むほど、色の着いたバーが左から右へと伸びていきます。

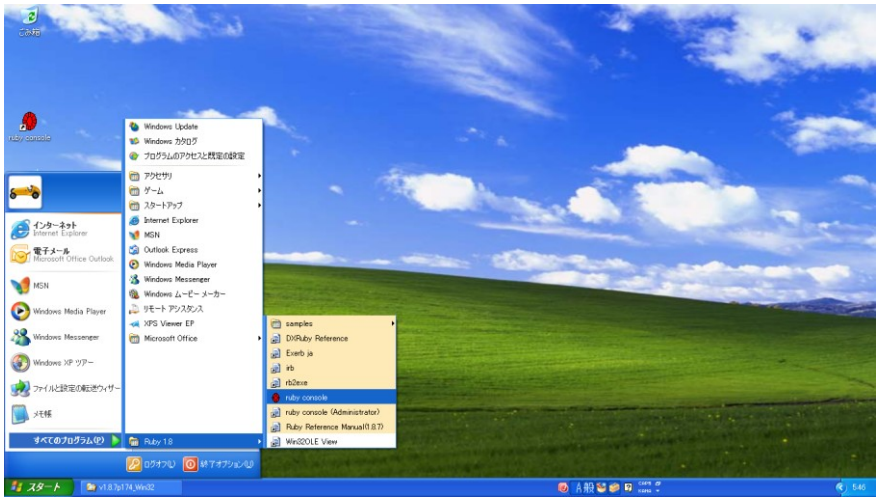
　しばらく時間が経過すると、以下のように、導入作業が完了したことを知らせる画面が表示されます。



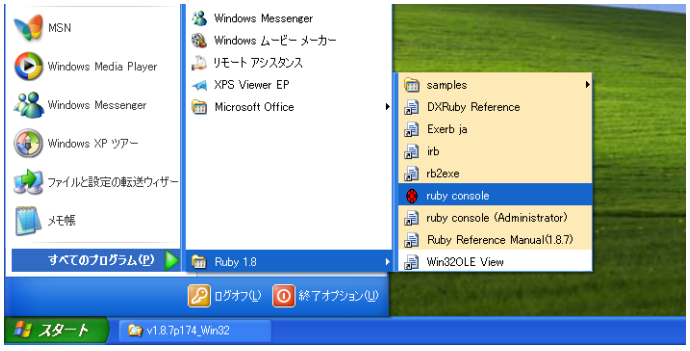
（fig_InstallingActiveRuby187_8.tif）

メッセージどおりに、右下にあるボタン「Close」をクリックします。

ActiveScriptRuby の導入が完了したところで、試しに ActiveScriptRuby を使用してみましょう。通常なら画面左下にあるスタートメニュー（「スタート」と書かれているか、Windows ロゴが描かれています）を開き、（「すべてのプログラム」にマウスカーソルを合わせるなどして）プログラム一覧を表示して、「Ruby 1.8」 - 「ruby console」を選択します。

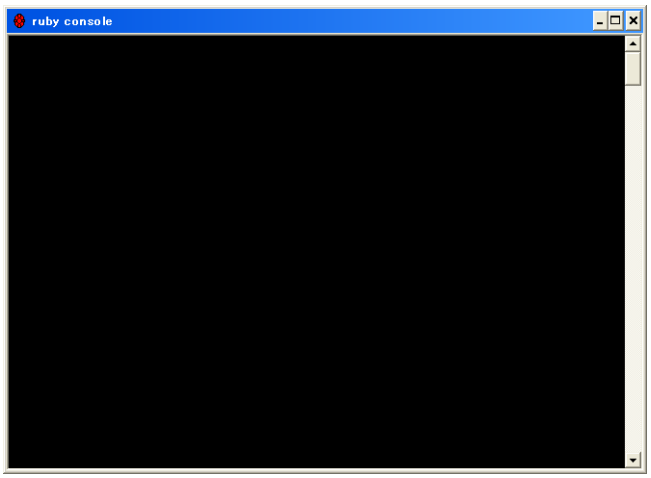


（fig_InstallingActiveRuby187_9.tif）



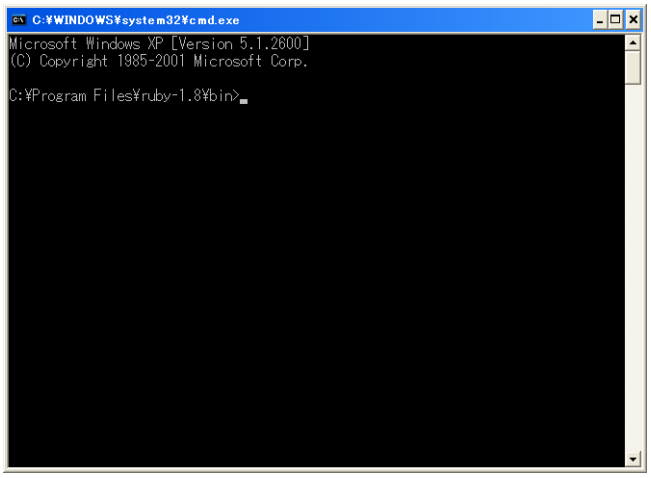
（fig_InstallingActiveRuby187_10.tif）

すると、以下の画面が表示されます。



（fig_InstallingActiveRuby187_11.tif）

この状態では何も操作ができませんが、数秒後に、以下の画面に変化します。



（fig_InstallingActiveRuby187_12.tif）

この状態になると準備完了です。

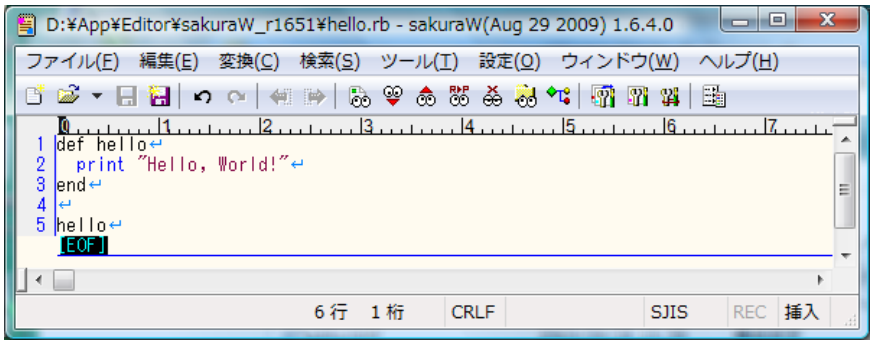
これで、とりあえずプログラミングに最低限必要なものは揃ったはずです。プログラミングをもっと便利にしてくれるものもあるのですが、それは後で述べることにします。

由緒正しい「Hello World」

SDK を導入して、プログラミングの環境ができあがったら、まず行うことはプログラム「Hello World」を作るものだと相場が決まっているようです。「Hello World」はコンピューターの画面に「Hello, World!」を表示するだけのプログラムです。…ただそれだけのプログラムです。

文章を入力するのに、おそらく多くの方はワードプロセッサー（ワープロ）（特に Microsoft Office Word）を使用されているかと思いますが、プログラミングでプログラムを書くときには、普通ワードプロセッサーは使用せずに、メモ帳（Notepad）やテキストエディットなどのテキストエディターを使用します。テキストエディターは原稿などの文章入力に特化したもので、今

日のワードプロセッサーのように、文字にフォントなどを設定して装飾したり、図や表を挿入したりするような機能を持っていません[19]。その代わり、大量の文章を書いても処理が遅くなることが少なく、処理能力の低いコンピューター上でも快適に動作します。テキストエディターの中には、キーワードに色をつけて見やすくしてくれるものもあります。



（fig_ColoringKeywords_1.tif）

ワードプロセッサーでもプログラムが書けないわけではありません（ただし、文書を保存する際に、ワードプロセッサーの標準形式ではなく、テキストファイル形式を選択しなければなりません）が、プログラミングに不要な機能（例えば、文字の装飾機能、レイアウト機能、文章校正機能（この機能は日常の文章の校正はしてくれますが、プログラムの校正はしてくれません）など）が多く、動作が（テキストエディターに比べて）遅いため、快適な作業にはそれなりに高性能なコンピューターが要求されます。そこで、ここではプログラムを書くときには、基本的にテキストエディターを使用します。

適当なテキストエディター（Windows ならメモ帳でも OK です）を用いて、以下のような内容を持つファイルを作成してください。

```
print "Hello, World!"
```

最初ですので、自分の手で入力してみることをお勧めします。**注意としては、プログラムを書くときには、（表示する文字列に日本語の文字が含まれる場合などを除いて）日本語入力を無効にしてください。**日本語入力を有効にしたまま文字入力を行うと、その結果できたプログラムは動作しないかもしれませんが、その説明は後で行います。日本語入力を入れたり切ったりするには、Windows ではキーボードで Alt キーを押しながら半角／全角キー（または漢字キー）を押します[20]。画面内にはおそらく、現在の文字入力の状態を示す IME バーが表示されているでしょうが、以下のような状態ならば、日本語入力は無効になっています。

（fig_MsImeOff_1.tif）

逆に、以下のような状態ならば、日本語入力は有効になっています。

---

[19] もっとも、ワードプロセッサーも当初（1970 年代末から 1980 年代前半にかけて）は専ら原稿入力のためのもの（日本語入力システムが内蔵されていたということ以外は、今で言う簡単なテキストエディター程度の機能しかありませんでした）であって、装飾やレイアウトなどは高価な（安くても数百万円、高ければ数千万円する）組版システムの役目でした。それが、高いコストをかけて高価な組版システムを所有する業者に依頼するという無駄なことをせずに、印刷などを全て社内でまかないたいという要求が高まるにつれて、1980 年代後半から 1990 年代にかけて、ワードプロセッサーに文字の装飾機能やレイアウト機能が備わっていくことになりました。

[20] 設定によっては、半角／全角キーだけを押すことで日本語入力を入れたり切ったりすることができます。

（fig_MsImeOn_1.tif）

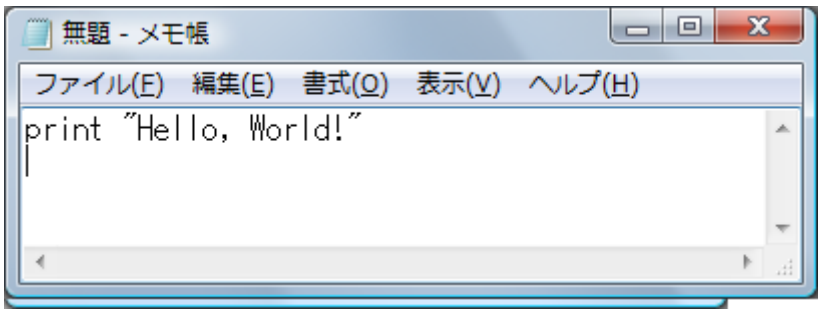
この場合は、Alt キーを押しながら半角／全角キーを押すことで、日本語入力を無効にしてください。

「print」の直後には必ず 1 つ以上（通常は 1 つ）の半角（欧文用）空白を挿入します。「Hello, World!」を二重引用符でくくってありますが、これが**文字列**（string）です[21]。ただし、くくるために使用される二重引用符は本式の「“」、「”」ではなく、簡易的な「"」（よく見ると、2 本の短い棒が曲がらずに垂直に伸びています）でなければならない（つまり、「“Hello, World!”」ではなく、「"Hello, World!"」とする）ことに注意してください。もっとも、テキストエディターを用い、キーボードで Shift キーを押しながら 2 キーを押せば、自然に簡易的な二重引用符が入力されるはずです[22]。

参考までに、

```
print "Hello, World!"
```

をテキストエディター・メモ帳に書き込んだときの画面写真を以下に掲載します。



（fig_HelloWorld_1.tif）

このような内容を持つファイルの名前をとりあえず「hello.rb」としておきましょう（ファイルの名前は何でもよいのですが、説明の都合上そうします。ただし、ファイルの名前の末尾は「.rb」にすることをお勧めします。これは拡張子（extension）と言うもので、ファイルの種類を示すものです）。このファイル「hello.rb」がまさに Ruby で書かれたプログラムなのです。

この後、**コンソール端末**（console terminal）（Windows なら、Windows 95／98／Me では「MS-DOS プロンプト」、Windows 2000／xp／Vista／7 では「コマンド プロンプト」、Windows 7「Windows PowerShell」[23]が使用できます）を開きます。これらは、スタートメニューのプログラム一覧（例えば「すべてのプログラム」の中）にある「アクセサリ」を開くと出てくるでしょう。

注意　ActiveScriptRuby 1.8.7 をインストールした場合は、コンソール端末としては、スタートメニューのプログラム一覧（例えば「すべてのプログラム」の中）にある「Ruby 1.8」 - 「ruby console」を使用してください。

[21] Ruby では、簡易的なアポストロフィー「'」でくくられる文字列を使用することもできますが、表記の方法が若干異なることがあります。ここではそのことについては深入りしません。

[22] ちなみに、Microsoft Office Word だと、オートコレクト機能により、キーボードで Shift キーを押しながら 2 キーを押しても、「"」の代わりに「“」や「”」が出現することがあります。

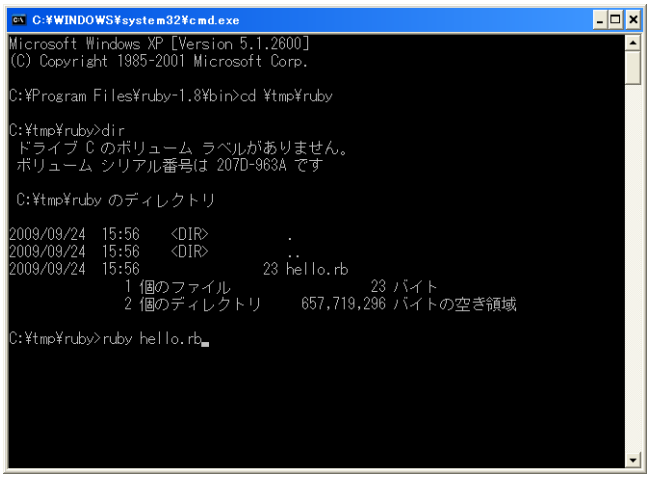
[23] Windows PowerShell は Windows xp／Vista でも、別途からダウンロードすることで使用できます。

そして、ファイル「hello.rb」が格納されているディレクトリーを「現在のディレクトリー」(current directory)に設定します。例えば、ファイル「hello.rb」がディレクトリー「C:¥tmp¥ruby」に存在するのなら、

```
cd ¥tmp¥ruby
```

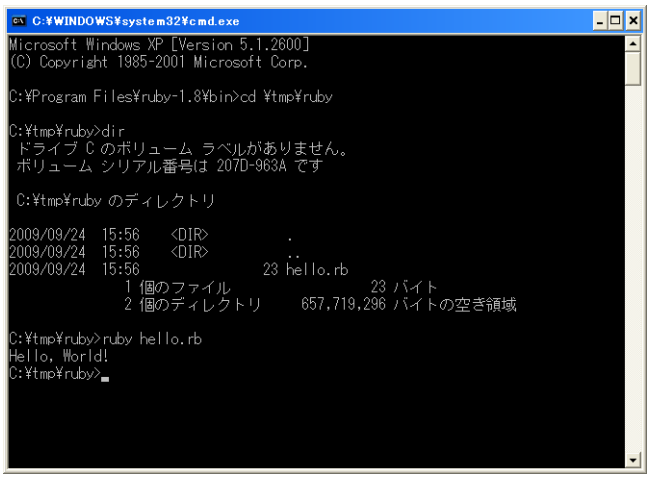
と入力します[24]。現在のディレクトリーの設定が終わったら、以下のように入力します。

```
ruby hello.rb
```



(fig_InstallingActiveRuby187_16.tif)

そうすれば、何の問題もなければ「Hello, World!」が表示され、そうでなければ何らかのエラーメッセージなどが出ます。以下の写真は成功した場合のものです。



(fig_InstallingActiveRuby187_17.tif)

エラーメッセージには、例えば以下のようなものがあります。

(Windows 95／98／Me の MS-DOS プロンプト)

コマンドまたはファイル名が違います.

[24] Windows PowerShell の場合、現在のディレクトリーを変更する命令は、本当は「Set-Location」であって、「Set-Location C:¥tmp¥ruby」と入力するのが正式ですが、本文に書かれているように「cd」を使用することもできます。

（Windows 2000／xp／Vista／7 のコマンド プロンプト）

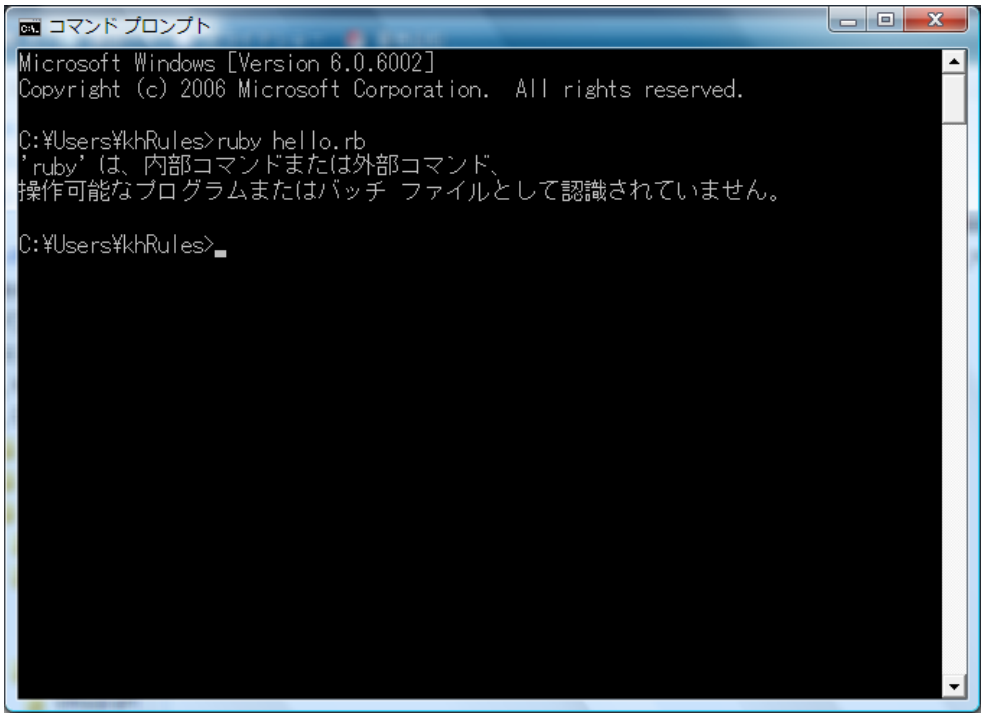
'○○○' は、内部コマンドまたは外部コマンド、
操作可能なプログラムまたはバッチ ファイルとして認識されていません。

（「○○○」は使用したコマンドの名前です。）

（Windows 7 の Windows PowerShell）

用語 '○○○' は、コマンドレット、関数、操作可能なプログラム、またはスクリプト ファイルとして認識されません。用語を確認し、再試行してください。
発生場所 行:△△△ 文字:×××
+ ○○○ <<<<

（「○○○」は使用したコマンドの名前です。）



（fig_CommandNotFound_1.tif）

これらは、Ruby で書かれたプログラムを読み取って実行するためのソフトウェアを呼び出そうとしたら、そのソフトウェアが存在しなかったことを意味します。その原因としては、例えば以下のようなものが挙げられます。

- まだ Ruby の SDK を導入していない。

  Ruby の SDK の導入作業を行ったが、正しく完了していない（例えば、環境変数 PATH の設定を行っていなかったり、設定を行ったが内容が間違っていたりする）。

  注意　ActiveScriptRuby を使用している場合は、通常の「MS-DOS プロンプト」、「コマンド プロンプト」や「Windows PowerShell」ではなく、ActiveScriptRuby のインストーラーによって作成された「ruby console」（スタートメニューのプログラム一覧（例えば「すべてのプログラム」の中）にある「Ruby 1.8」 - 「ruby console」）を使用してください。

- コマンドの綴りを間違えている（例えば、「ruby hello.rb」のうち先頭の部分「ruby」が「ryby」とか「rubt」など別のものになっている）。

エラーメッセージは他にも以下のようなものがあります。

```
ruby: No such file or directory -- hello.rb (LoadError)
```

（日本語訳　ファイルまたはディレクトリーhello.rb が見つかりません。）

これは、とりあえず Ruby で書かれたプログラムを読み取って実行するためのソフトウェアを呼び出すことには成功したものの、Ruby で書かれたプログラム（ここではファイル「hello.rb」のことです）が見つからなかったことを表しています。先ほど保存した、Ruby で書かれたプログラムのファイル名が一字一句違わず「hello.rb」になっているか（例えば、「hello.rb.txt」になっていないか。Windows のメモ帳を使うと、勝手にファイル名の最後に「.txt」を付けられることがあります。このようなこともありますので、Windows エクスプローラーでは拡張子（ファイル名のうちピリオド「.」以降の文字列）を表示させることをお勧めします)、そして、コマンド「ruby hello.rb」で「ruby」の直後が「hello.rb」（先ほど保存したプログラムのファイル名と一字一句同じ）であるかを確認する必要があります。

エラーメッセージの別の例を取り上げてみましょう。

```
hello.rb:1:in `<main>': undefined method `prunt' for main:Object (NoMethodError)
```

（日本語訳　メソッド prunt は未定義です。）

これは、とりあえず Ruby で書かれたプログラムを読み取って実行するためのソフトウェアを呼び出すことには成功したものの、Ruby で書かれたプログラム（ここではファイル「hello.rb」のことです）の内容に間違いがあることを表しています。ここでは、ファイル「hello.rb」の中身のうち、「print」が「prunt」という誤った綴りになっているために発生しています。この場合は、エラーメッセージを参考にして、正しい綴り（ここでは「print」）に訂正します。

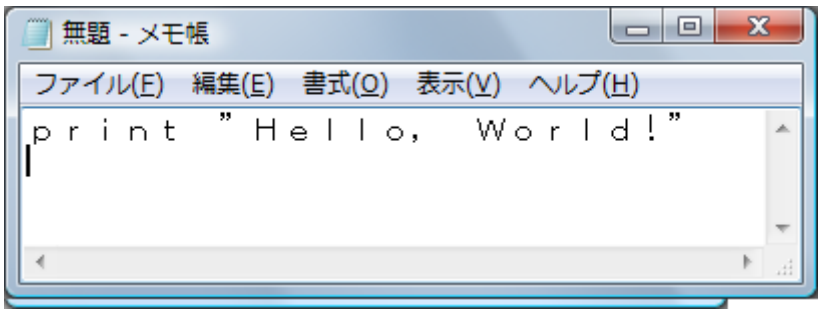
さらに別のエラーメッセージを挙げます。

```
hello.rb:1: invalid multibyte char (US-ASCII)
```

（日本語訳　エンコーディング US-ASCII としては不正なマルチバイト文字です。）

これは、とりあえず Ruby で書かれたプログラムを読み取って実行するためのソフトウェアを呼び出すことには成功したものの、マルチバイト文字（…と言っても分かりづらいと思いますが、ここでは和字（日本語の文字）だということにしましょう）が入るべきではないところに入っていることを表します。これは、例えばプログラム（ファイル「hello.rb」）の中身が以下のように、異なる種類の文字を使用していると発生します。

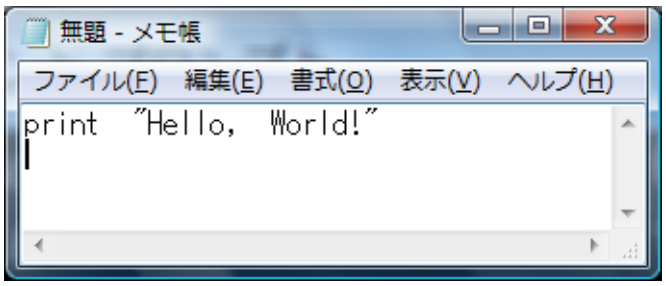
```
ｐｒｉｎｔ　”Ｈｅｌｌｏ，　Ｗｏｒｌｄ！”
```



（fig_HelloWorld_2.tif）

（英数字が全角（和文用）になっている）

```
print  "Hello,  World!"
```

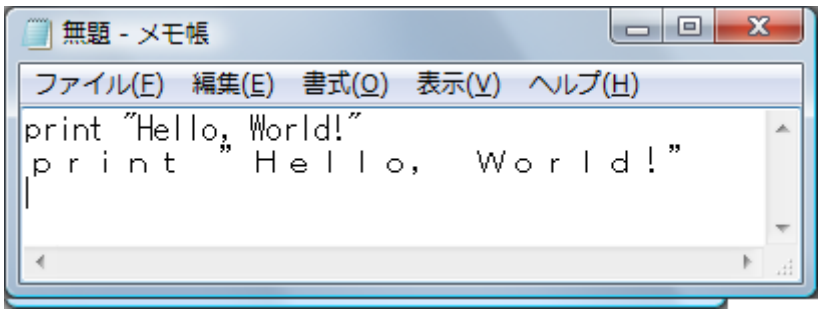


（fig_HelloWorld_8.tif）

（空白が全角（和文用）になっている）

異なる種類の文字は日本語入力が有効になっていると入ることがあります。異なる種類の文字はよく見ると、文字の形が若干異なっています。

「print "Hello, World!"」と「ｐｒｉｎｔ　”Ｈｅｌｌｏ，　Ｗｏｒｌｄ！”」

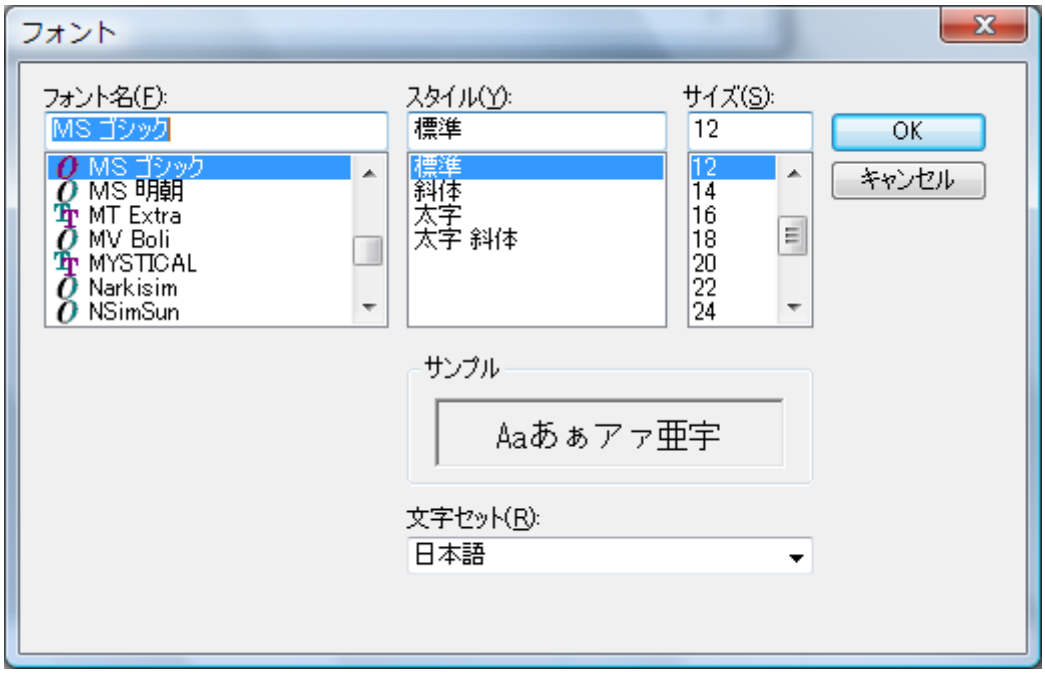


（fig_HelloWorld_3.tif）

上記の 2 つの文字列は、私たち人間はどちらでも「print "Hello, World!"」と読めそうですが、コンピューターは異なる文字と見なします（より厳密には、**コンピューターは文字の形を気にせず、文字コード（文字に割り当てられた番号）のみで同じ文字かどうかを判定します**。日本語入力を無効にして入力した「print」と、日本語入力を有効にして入力した「ｐｒｉｎｔ」は互いに異なる文字コードで構成されており、そのため異なる文字として判定されるのです）。

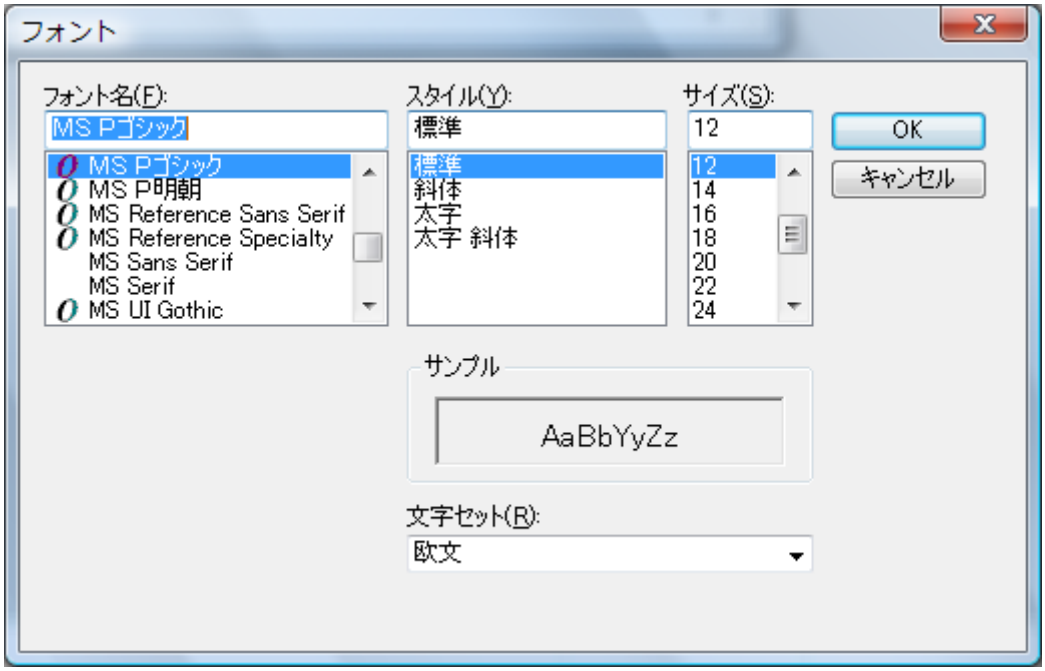
…とは言っても、普通の編集作業で文字コードをいちいち表示するのはあまりに煩わしすぎるでしょう。そこで、私たちは普通、プログラミングで普通に使用できる半角（欧文用）文字なのか、全角（和文用）文字なのかを、文字の形や幅で区別します。

先ほどの図からも分かるように、全角（和文用）文字の方が半角（欧文用）文字よりも幅が広い傾向にあります。上図では、全角（和文用）文字は半角（欧文用）文字と比べて幅が 2 倍になっていますが、実際の幅はフォントに依存します。上図では等幅フォントの一つである「MS ゴシック」を使用していますが、文字ごとに異なる幅をとるプロポーショナルフォントの一つである「MS P ゴシック」に変えてみましょう。Windows のメモ帳では、メニューバーの「書式(O)」→「フォント(F)…」を選択すると、以下の画面が出現します。



（fig_ChooseFontDialogBox_1.tif）

ここで、「フォント名」を「ＭＳ　ゴシック」から「ＭＳ　Ｐゴシック」に変更します。



（fig_ChooseFontDialogBox_3.tif）

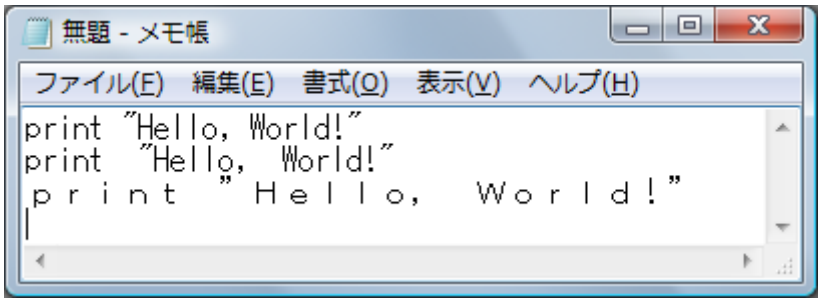
この状態でボタン「OK」をクリックすると、「MS P ゴシック」というフォントでプログラムが表示されるようになります。

ここでは比較のために、メモ帳には以下の 3 つの文を書いておきます。1 番目は全部半角（欧文用）文字で入力したもの、2 番目は空白のみ全角（和文用）文字で他は半角（欧文用）文字

で入力したもの、3 番目は全部全角（和文用）文字で入力したものです。既に見たように、1 番目が今回のプログラムとしては正しい文、残りの 2 つは間違った文でした。

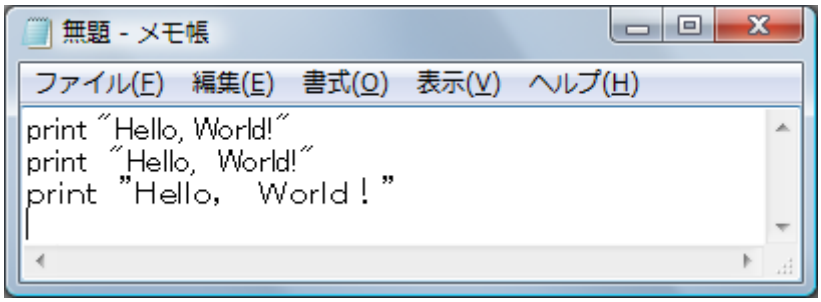
```
print "Hello, World!"
print  "Hello,  World!"
ｐｒｉｎｔ　”Ｈｅｌｌｏ，　Ｗｏｒｌｄ！”
```

すると、フォントを「MS ゴシック」に設定した場合は以下のようになります。



（fig_HelloWorld_6.tif）

一方、フォントを「MS P ゴシック」に設定した場合は以下のようになります。



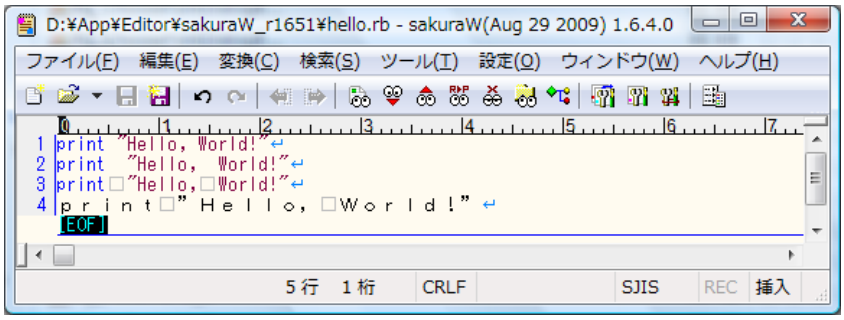
（fig_HelloWorld_7.tif）

プロポーショナルフォントである「MS P ゴシック」の方が、3 つの見た目の差が小さくなっていますが、それでも見分けられないほどではありません。

フォントを何にするのかは個人の自由です（フォントの情報がプログラムとして保存されるわけではありませんので、プログラムの実行には一切影響を与えません）が、プログラミングに慣れていないうちは、半角（欧文用）か全角（和文用）かの区別を行いやすい、「MS ゴシック」や「Osaka-等幅」などの等幅フォントを使用されることをお勧めします。

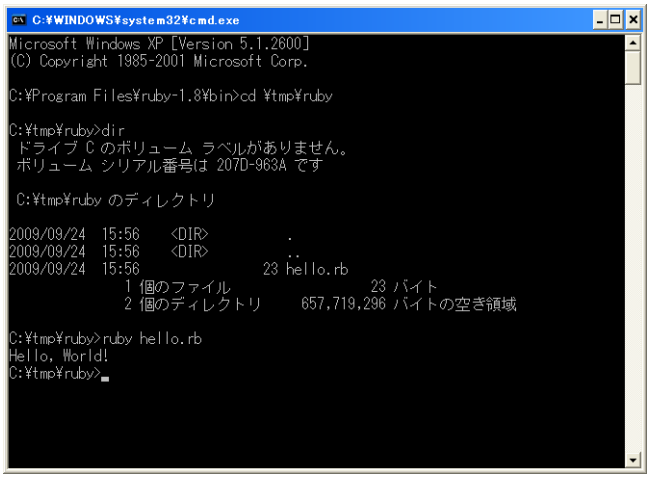
いずれにしても、半角（欧文用）か全角（和文用）かの区別がしづらいのは空白です。空白は目には見えないのですが、幅は異なります。通常、全角（和文用）空白は、半角（欧文用）空白よりも幅が広くなっています。カーソル（キャレット）を移動させ、一度にカーソルがどの程度動くのかを見るとよいでしょう。また、プログラムのソースファイルを編集するために使用するソフトウェア（テキストエディターなど）によっては、全角（和文用）空白と半角（欧文用）空白を区別して表示する機能を備えていることがあります。残念ながら、Windows に標準で添付されているメモ帳やワードパッド（Wordpad）にはそのような機能はありませんが、市販のソフトウェアである MIFES（制作・販売: メガソフト）や WZ EDITOR（制作・販売: WZ

ソフトウェア）など、オンラインソフトウェアであるサクラエディタ[25]、TeraPad（制作: 寺尾 進）[26]や秀丸エディタ（制作・販売: サイトー企画）[27]などは全角（和文用）空白と半角（欧文用）空白を区別して表示する機能を備えていますので、それらのテキストエディターを利用するのもよいでしょう。



（fig_DisplayingFullWidthSpaces_1.tif）

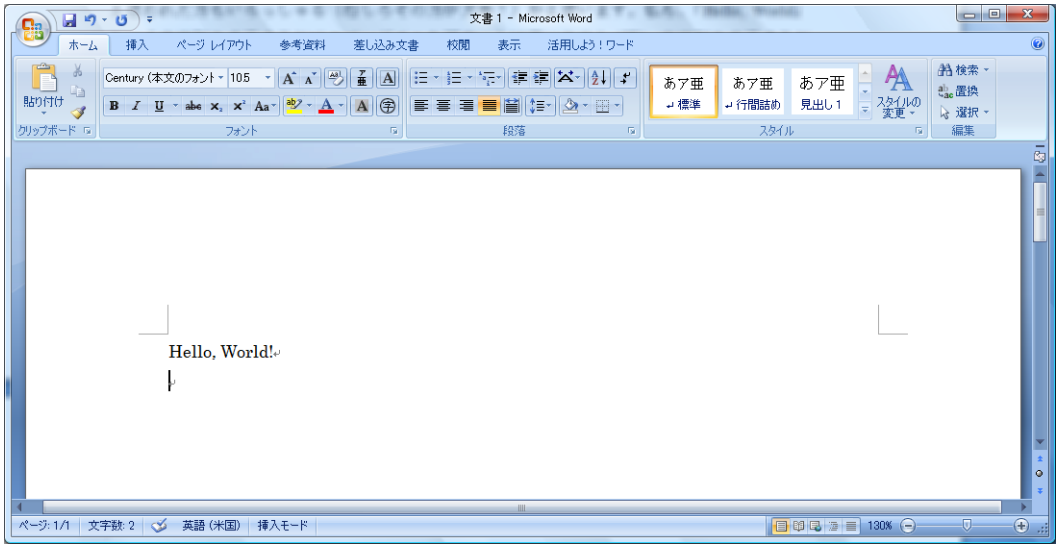
さてさて、以下の写真のように**うまく「Hello, World!」という文字列が表示されたら、おめでとうございます、あなたはプログラミングの第一歩を踏みしめることに成功しました！**



（fig_InstallingActiveRuby187_17.tif）

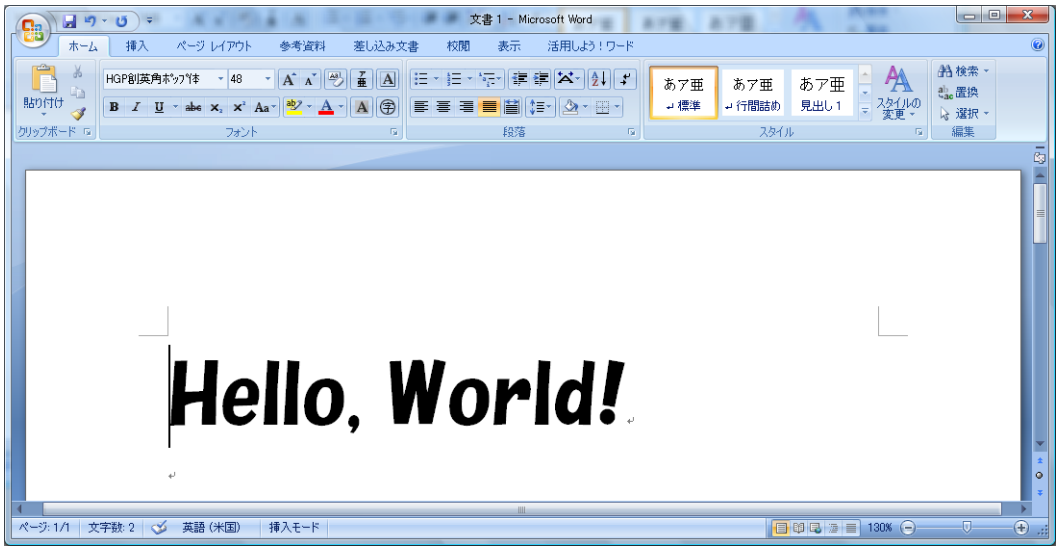
…ここで、**「何が『おめでとうございます』だ。こんなのちっとも面白くないではないか！」**と思われた方もいらっしゃる（むしろその方が大半？）かと思います。私も、「Hello, World」という文字列を表示するだけのプログラムを面白いとは思いません（笑）。文字列を表示するだけなら、ワードパッドなどのエディターや Microsoft Office Word などのワープロソフトを起動して、キーボードを用いて直接「Hello, World!」と入力すれば済むはずです。

[25] http://sakura_editor.at.infoseek.co.jp/
[26] http://www5f.biglobe.ne.jp/t-susumu/library/tpad.html
[27] http://hide.maruo.co.jp/software/hidemaru.html



（fig_HelloWorldOnMsWord_1.tif）

また、この場合はさらに、フォント（書体、文字の大きさなど）を変更し、文字の見た目を変化させることだって可能です。こちらの方が面白さの点ではまだマシだと思いませんか？



（fig_HelloWorldOnMsWord_2.tif）

それにもかかわらず、「Hello World」はプログラミングの入門では頻繁に取り上げられます。これはプログラマーの趣味が普通の人と大きく異なるからでしょうか？　それもあるかも知れませんが、最大の理由は、**「何かを表示させることでプログラムがどのような状態にあるのかが分かる」**という事実が学べることにあります。

「Hello, World!」という文字列が表示されることはプログラムを正しく作って動かすことができたことの証明になりますし、そのような文字列が表示されずに「No such file or directory」のようなエラーメッセージが表示されても、そのような表示を参考にしてとるべき行動を決定することが可能です（例えば、「No such file or directory」というエラーメッセージが表示されたのなら、プログラムの実行に必要なファイルの名前や置き場所が間違っていないかどうかを確認

して訂正するなどの行動をとります)。もし何も表示させなければ、プログラムが正しく実行されたのかどうか、正しく実行されていなければ、どこがおかしいのかを知ることができないでしょう。例えば、電化製品の多くには電源ランプがあり、電源が入っている間は電源ランプが点灯することになっています。もし電源ランプがなければ、電化製品が動かないときに、それは単に通電されていない（例えば、プラグがコンセントから抜けている）だけなのか、それとも電化製品の内部に異状があるのかを区別しづらくなってしまいます。プログラミングでも同じです。表示を多いに活用しましょう。

…とは言っても、やはり「Hello World」が面白く感じられないことはあると思います。パソコンにワープロソフトすら標準では付属されていなかった 20 年以上前（1980 年代またはそれ以前）なら、単に文字列を表示するだけのプログラムを作って動かすだけでも感動ものだったのでしょうが、ワープロソフトが用意されていることはもちろんのこと、コンピューターゲームなどで美しいコンピューターグラフィックス（computer graphics、CG）を見ることが当たり前になってしまった今日では、単に文字列を表示するだけというのは確かにつまらないかもしれません。

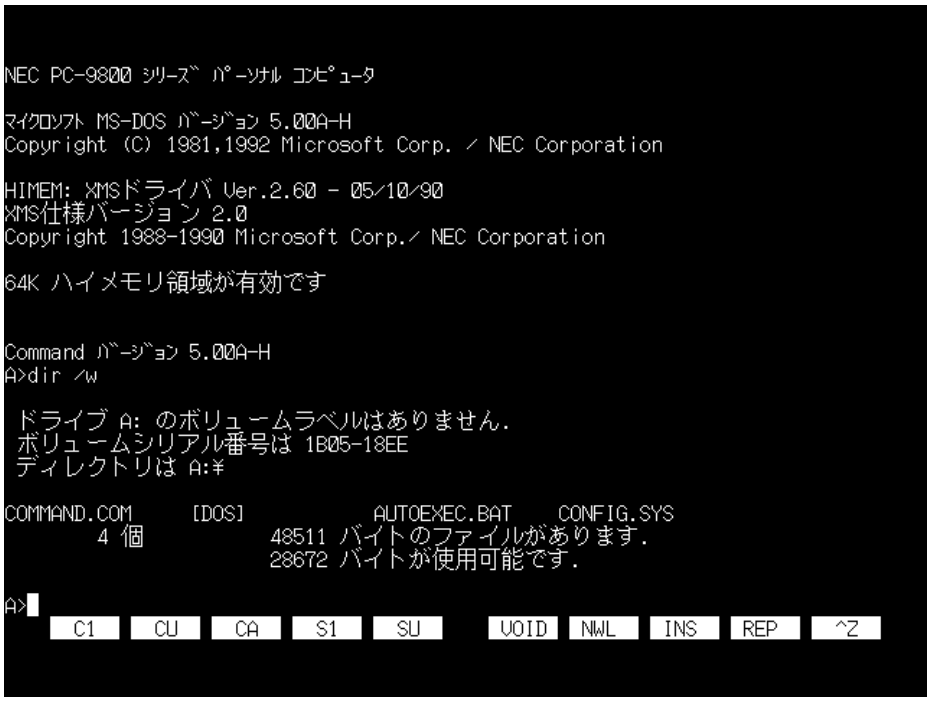
しかし、残念なことに、今日市場に出回っているソフトウェア（例えば、ワープロソフトとか、コンピューターゲームソフト）のまねをするのは大変難しいことなのです。昔だったら、1〜2 人でそれなりのソフトウェアを作ることも可能でしたが、今日のソフトウェアは非常に大規模なものになりがちであり、商品として売り出すソフトウェアを制作するのに数十人、数百人という人材、数ヶ月、数年という期間と数億円、数十億円の費用をかけることは珍しくなくなりました[28]。商品として売り出さなければ、個人が制作したソフトウェアも多数あるのですが、そのようなソフトウェアは小さいものにとどまっており、商品として売り出されるようなものに正面から対抗できるようなものではありません。そのような状況ですので、今日よく見たり触ったりするような興味深いソフトウェアをプログラミングの入門で取り扱うのはほとんど無理なのです。その意味では、やはり「Hello World」のような単純なプログラムはプログラミングの入門としてふさわしいのですが、プログラミングの入門に使え、かつ、もっと興味深いようなプログラムはないものでしょうか？

プログラミングをもっと便利にするには？

ところで、「MS-DOS プロンプト」、「コマンド プロンプト」や「Windows PowerShell」などといったコンソール端末を使ったことがあったり、操作に慣れていたりする方はおそらくかなり少ないと思われます。1980 年代以前はコンソール端末を用い、キーボードでの文字入力によってコンピューターに命令するという CUI（character-based user interface）はごく当たり前でした。

例として、Windows の前身とも言える MS-DOS の画面写真を以下に掲載します。

[28] 例えば、有名なコンピューターロールプレイングゲーム（RPG）の一つである「ファイナルファンタジー」のうちいくつかは、制作に数十億円がかかっています。例えば、1997 年 1 月にプレイステーション用ソフトウェアとして発売された「ファイナルファンタジーVII」は約 200 人によって約 15 億円をかけて制作されました（出典: http://ffx.sakura.ne.jp/ff7.htm）。



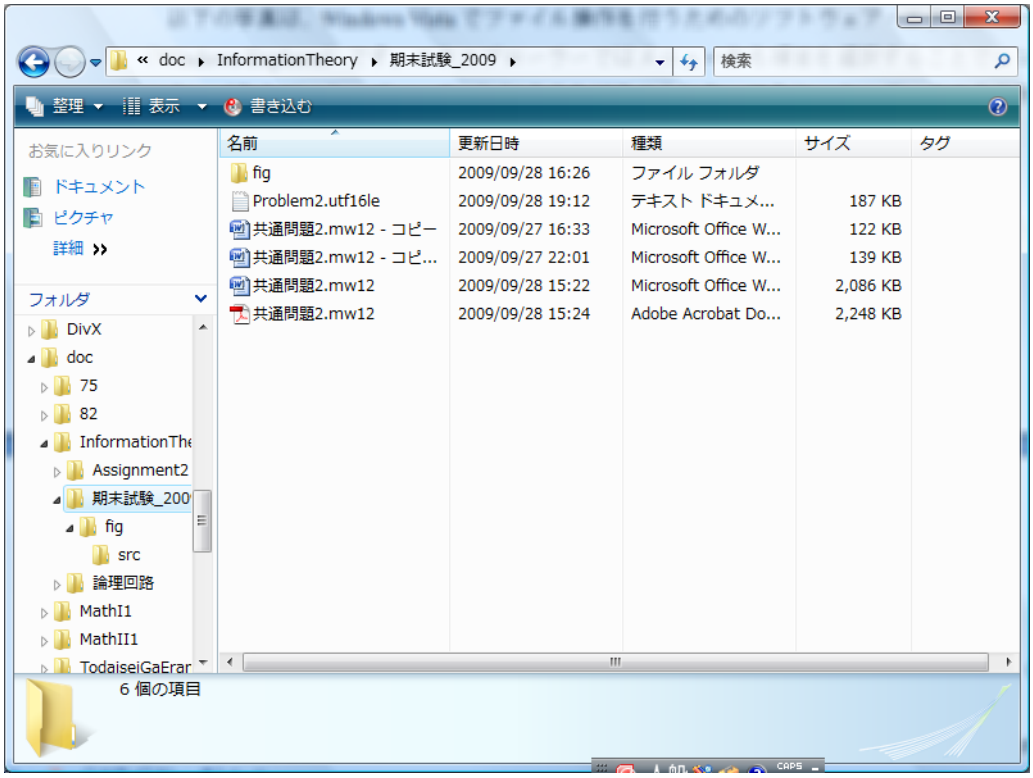
（fig_Pc98MsDos_1.tif）

「A>」という表示はプロンプト（prompt）と呼ばれており、そこに文字列で命令を入力すると、その命令に従ってコンピューターが動作します。上の写真では、ファイルを一覧表示する「dir」という命令を使用しています（「/w」というのは、1 行に 5 個のファイル名を表示するというオプションです）[29]。

しかし、1990 年代以降は、画面にグラフィックス（アイコン、画像や写真など）が表示され、それを見ながら直感的に指示するという GUI（graphic-based user interface）が当たり前になっていきました。

以下の写真は、Windows Vista でファイル操作を行うためのソフトウェア・エクスプローラー（Windows Explorer）です。エクスプローラーではメニューから項目を選択することでコンピューター（およびエクスプローラー）に命令を送ることになっており、キーボードで文字列を入力して、それをもって命令とすることはありません。

[29] 実は、MS-DOS にも、「MS-DOS シェル」など、マウスを使用して直感的にファイル操作ができるものもありましたが、MS-DOS が使用されていた間は、やはりキーボードで入力して命令を送る方法が一般的でした。



（fig_WindowsExplorer_1.tif）

このようなことがあって、コンソール端末の使用方法を覚えなければコンピューター（あるいはパソコン）が使えないということはなくなりました。

CUI 方式では、コンピューターへの命令はキーボードでの文字入力によって行われますが、一文字でも入力し間違えると、コンピューターは「善意的に」解釈したり推測したりせずに、

```
コマンドまたはファイル名が違います.

'〇〇〇' は、内部コマンドまたは外部コマンド、
操作可能なプログラムまたはバッチ ファイルとして認識されていません。

用語 '〇〇〇' は、コマンドレット、関数、操作可能なプログラム、またはスクリプト ファイルとして認識されません。用語を確認し、再試行してください。
発生場所 行:△△△ 文字:×××
+ 〇〇〇 <<<<

command not found
```

などと表示して、命令を拒否します。これは初心者にはなかなかつらいことでした。それに比べれば、画面を見ながら直感的に操作できる GUI の方が遥かに優れていると思われることでしょう。

ところが、依然として CUI を好む「上級者」は少なくありません。その理由は、キーボードでの文字入力による命令が素早く行えることにあります。そのような命令はおよそ数～数十文字（たいていは多くても 50 文字程度でしょう）で構成されていますが、コンピューターに慣れている人なら、その程度の文字数を僅か数秒で入力することができます。あるいは、CUI ベースのシステムは複数の命令を集めてまとめて実行するような仕組みを備えていることも多く、そうすれば、一つの処理が終わるごとにその都度命令を送ることなく、作業の終わりまで自動

的に行ってくれます。考えてみれば、プログラミングも多くの場合は文字入力でコンピューターに命令を送るのですから、方法としては似たようなものだと言えます。

一方、GUI は画面を見ながら操作する関係で、一つの処理が終わるごとに何らかのメッセージを出して、利用者に次の命令を求めることが多く、予め行うべきことが明確に分かっていても、一つの処理が終わるごとに命令を出さなければならないので、実は面倒ということもあります。

もっとも、これらの特徴は一概には言えなく、例えば、CUI ベースのシステムでは、長いファイル名など長い文字列を入力しなければならない場合は、キーをたくさん叩かなければなりませんので、やはり面倒な場合もあります。その場合は、文字列を完全に入力しなくても対象を選択できるような GUI ベースのシステムの方が楽な操作ができるということも考えられます。もっとも、CUI ベースのシステムでも、今では多くの場合は文字列の入力を補完する機能を備えていることがあります。例えば、Windows の「コマンド プロンプト」や Unix の「bash」では、ファイル名を途中まで入力してから Tab キーを押すと、ファイル名の残りの文字列を自動的に埋めてくれます[30]。これだと楽ですね。

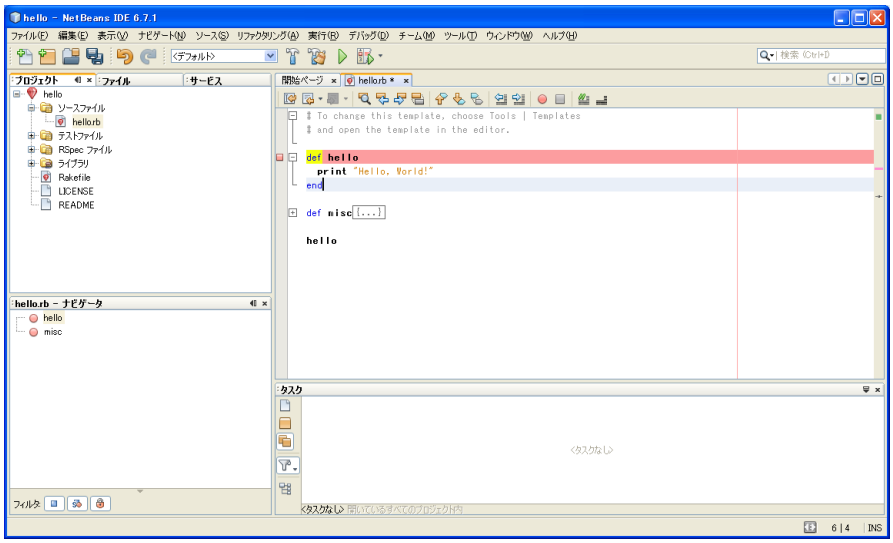
とは言え、プログラミングの環境に GUI を導入して直感的に操作できるようにする方法もあります。さすがに、プログラミングそのものに文字入力は欠かせませんが、プログラムの実行などの雑用を、キーボードを用いた文字入力による命令出しではなく、マウスなどを用いた直感的な操作によって実現することはできます。例えば、**統合開発環境**（integrated development environment、IDE）はプログラムを作成、編集するための機能だけでなく、プログラムの作成支援（必要な文字列を補完する、自動的に簡単な説明を表示して、次に何を行うべきなのかを示す、読まなくてもよい部分を表示しないようにする、など）やプログラム・デバッグの実行機能（より正確には、プログラムの実行を担当するソフトウェアを呼び出しますが、実行経過をより分かりやすく示すようにすることもできます）なども提供してくれます。

統合開発環境には Eclipse（制作: IBM）[31]や NetBeans（制作: Sun Microsystems）[32]などがあります。以下の写真は、NetBeans 6.7.1 を用い、Ruby の簡単なプログラムを書いて実行させているところです。プログラム中でキーワードを種類ごとに着色させているだけでなく、処理のまとまりや、あるキーワードに対応する別のキーワードを明示したり、見る必要のない処理のまとまりを折りたたんで隠したり、デバッグという機能を使用してプログラムの実行中に実行箇所や状態などを逐次表示したりすることができます。

[30] 途中まで入力された文字列を先頭に持つファイル名が複数ある場合は、Windows の「コマンド プロンプト」はとりあえずファイル名の全部を表示し、Tab キーを押すごとに別のファイル名を候補として出すようになっているが、Unix の「bash」は一意に定まる（複数の候補として分かれない）部分しか表示しません。

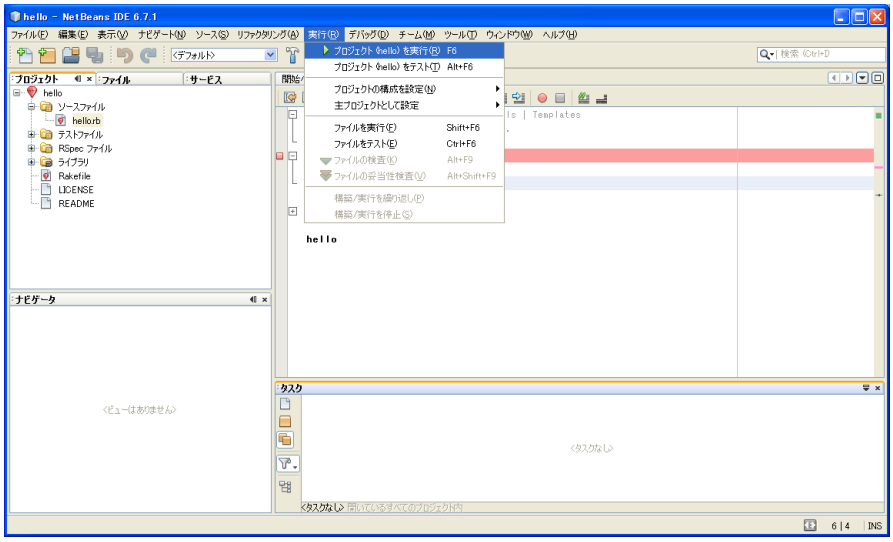
[31] http://www.eclipse.org/

[32] http://www.netbeans.org/index_ja.html



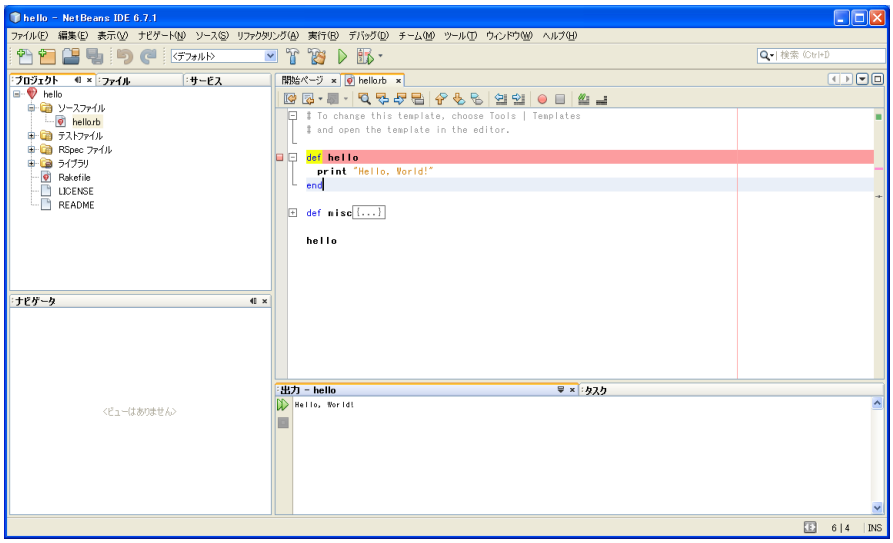
(fig_NetBeans671_1.tif)

上の写真は簡単なプログラムを書いたところです。def - end 構文は一つの処理のまとまりを表していますが、上の写真における「def hello」～「end」ように、色をつけて分かりやすく表示してくれます。また、その下にある「def misc {...}」は、本当は中身があるのですが、折りたたんで表示しないようにしてあります。



(fig_NetBeans671_2.tif)

上の写真は先ほど書いたプログラムをこれから実行しようとしているところです。コンソール端末を使用する場合のようにキーボードで命令（例えば、「ruby hello.rb」）を入力する必要はありません。



（fig_NetBeans671_3.tif）

上の写真はプログラムの実行が完了したところです。画面下側に「出力 - hello」という項目があり、その中に「Hello, World!」という文字列が表示されているのが分かります。

統合開発環境はプログラミングに必須というわけではありませんが、初心者にとってはむしろ取っ付きやすい環境を作ってくれるでしょう。また、上級者でも、大規模なプログラムを作るときには、統合開発環境によってプログラミングが便利になることも少なくありません。

本当は統合開発環境の導入方法なども説明したいところですが、時間の都合で割愛させていただきます。まぁ、大学の教養課程の授業程度なら統合開発環境がなくても大丈夫でしょう（たぶん…）。

とりあえず共通問題 2 の答えを Ruby で書く

以下に共通問題 2 の解答を再掲します。

処理 B

```
if S=0 then
    T ← T+P
else if S=1 then
    T ← T+2P
else if S=2 then
    T ← T+2P
else （すなわちS=3なら）
    T ← T+3P
endif
R ← 10-P
I ← 2
```

処理 C

```
if S=0 then
    T ← T+P
else if S=1 then
    T ← T+P
else if S=2 then
```

```
            T ← T+2P
        else （すなわちS=3なら）
            T ← T+2P
        endif
        if P≧R then
            S ← 1
        else （すなわちP<Rなら）
            S ← 0
        I ← 1
```

これらを問題文にある「A の計算手順」と共に、問題文にある「計算手順」に埋め込むと、以下のようになります。

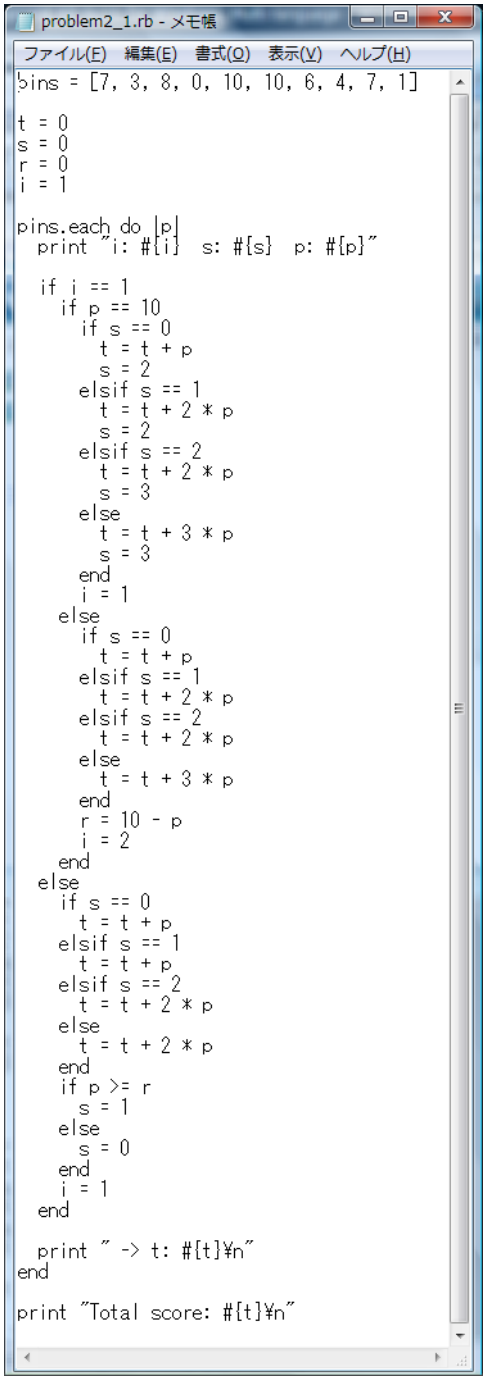
```
    if I=1 then
        if P=10 then
            if S=0 then
                T ← T+P
                S ← 2
            else if S=1 then
                T ← T+2P
                S ← 2
            else if S=2 then
                T ← T+2P
                S ← 3
            else （すなわちS=3なら）
                T ← T+3P
                S ← 3
            endif
            I ← 1
        else （すなわち0≦P<10なら）
            if S=0 then
                T ← T+P
            else if S=1 then
                T ← T+2P
            else if S=2 then
                T ← T+2P
            else （すなわちS=3なら）
                T ← T+3P
            endif
            R ← 10-P
            I ← 2
        endif
    else （すなわちI=2なら）
        if S=0 then
            T ← T+P
        else if S=1 then
            T ← T+P
        else if S=2 then
            T ← T+2P
        else （すなわちS=3なら）
            T ← T+2P
        endif
        if P≧R then
            S ← 1
        else （すなわちP<Rなら）
            S ← 0
        I ← 1
    endif
```

これをできるだけ忠実に Ruby で書き直し、得点計算の途中経過を表示するようにしたものが以下のプログラムです。

プログラム例 1

```
pins = [7, 3, 8, 0, 10, 10, 6, 4, 7, 1]

t = 0
s = 0
r = 0
i = 1

pins.each do |p|
  print "i: #{i}  s: #{s}  p: #{p}"

  if i == 1
    if p == 10
      if s == 0
        t = t + p
        s = 2
      elsif s == 1
        t = t + 2 * p
        s = 2
      elsif s == 2
        t = t + 2 * p
        s = 3
      else
        t = t + 3 * p
        s = 3
      end
      i = 1
    else
      if s == 0
        t = t + p
      elsif s == 1
        t = t + 2 * p
      elsif s == 2
        t = t + 2 * p
      else
        t = t + 3 * p
      end
      r = 10 - p
      i = 2
    end
  else
    if s == 0
      t = t + p
    elsif s == 1
      t = t + p
    elsif s == 2
      t = t + 2 * p
    else
      t = t + 2 * p
    end
    if p >= r
      s = 1
    else
      s = 0
    end
    i = 1
  end

  print " -> t: #{t}¥n"
end

print "Total score: #{t}¥n"
```

プログラム例 1 を書いたところの写真



（fig_Problem2_1.tif）

このプログラム例 1 について説明しましょう。第 1 行で、倒したピンの数をまとめて配列変数 pins に格納しています。配列（array）は、複数の値を一つにまとめたものです。そして、第 8 行（空行も一緒に数えています）にある「pins.each do |p|」（「|p|」で「p」を囲んでいる「|」は数字の一「1」、英大文字のアイ「I」や英小文字のエル「l」ではなく、垂直バーです。この垂直バー「|」は一般的な日本語キーボードでは、Shift キーを押しながら¥キーを押すことで入力できます）で、配列変数 pins に格納されている値を一つずつ取り出して変数 p に代入し、その

それぞれの値について、第 9 行以降の処理を実行します（結果として、配列変数 pins に格納されている値の個数の分だけ、第 9 行以降の処理は繰り返されることになります)。

ここで処理が繰り返されるのは、第 59 行にある「end」の直前まで、すなわち、第 9～58 行（ちなみに、第 58 行は「print " -> t: #{t}¥n"」です）ですが、これらの行には、先頭に 2 つの半角（欧文用）空白を挿入することで字下げを行っています。Ruby の規則として字下げは必須ではない（字下げがなくても、プログラム中に記されている語句によって、繰り返しの対象となる範囲は明確に定められます）[33]のですが、**字下げを行うと、プログラマー（人間）が見ても繰り返し行われる処理のまとまりなどが分かりやすくなりますので、普通は字下げを行います**。仮に字下げを行わなかった場合のプログラムを以下に示します。

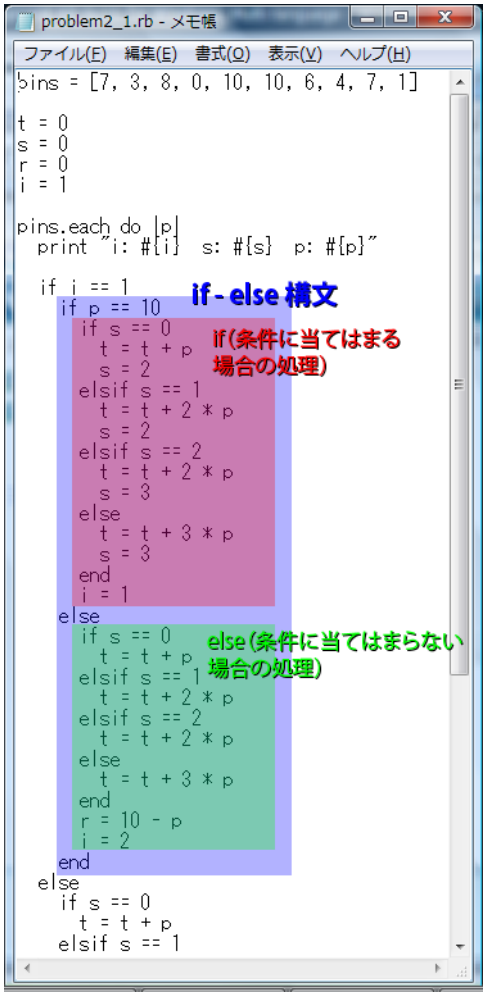
プログラム例 2

```
pins = [7, 3, 8, 0, 10, 10, 6, 4, 7, 1]

t = 0
s = 0
r = 0
i = 1

pins.each do |p|
print "i: #{i}  s: #{s}  p: #{p}"

if i == 1
if p == 10
if s == 0
t = t + p
s = 2
elsif s == 1
t = t + 2 * p
s = 2
elsif s == 2
t = t + 2 * p
s = 3
else
t = t + 3 * p
s = 3
end
i = 1
else
if s == 0
t = t + p
elsif s == 1
t = t + 2 * p
elsif s == 2
t = t + 2 * p
else
t = t + 3 * p
end
r = 10 - p
i = 2
end
else
if s == 0
t = t + p
elsif s == 1
t = t + p
```

[33] FORTLAN や Python など、字下げを必須とするプログラミング言語もあります。

```
elsif s == 2
t = t + 2 * p
else
t = t + 2 * p
end
if p >= r
s = 1
else
s = 0
end
i = 1
end

print " -> t: #{t}¥n"
end

print "Total score: #{t}¥n"
```

字下げを施さなくてもプログラムは正しく実行されます。しかし、処理のまとまりは、「do - end」、「if - （elsif） - （else） - end」（丸括弧「(」、「)」でくくられた部分は省略可能）など様々なものによって作られるのですが、字下げが施されていないプログラムはプログラマーが見て簡単にどこからどこまでが処理の一つのまとまりなのかが分かるようになっていません（簡単には分かりませんよね？）。しかし、字下げを施せば、字下げの幅が最初と比べて小さくなっていない連続した行が処理の一つのまとまりであることが簡単に分かります。



（fig_Problem2_1_3.tif）

処理のまとまりがより明確になるように色を着けてみましたが、これは説明の都合で写真編集を行った結果であって、実際の表示では上の写真のように色が着くわけではありません。

人間（プログラマー）にとって見やすい、読みやすいプログラムを作ることは、プログラミングでは事実上必須です。小さいプログラムなら読みづらくても何とかなるのですが、数万行、数百万行といった大規模なプログラムでは、様々な部分を読んだり編集したりしなければならないため、読みづらい箇所があると作業が非常に大変になります。このことは実際に大規模なプログラミングに携わると実感することができます。

例えば、プログラムの中に潜む誤りを探して訂正しなければならないことはよくあります。しかし、そのプログラムが読みづらいものだと、その誤りがどこにあるのかを探したり、その誤りをどのように訂正したりすればよいのかを判断するのが難しくなってしまいます。事実上修正不能になることも少なくありませんし、その場合は、誤りを訂正するのをあきらめ、プログラムを一から作り直さなければならないかもしれません。それは非常に苦労することですが、その苦労はプログラムの機能向上など有益なものにはほとんど結びつきません。

他の仕事や作業でもそうなのですが、がむしゃらにがんばるほど――日本のように国によってはがんばるのが美徳になっているようですが――状態はよくなるどころかむしろ悪くなることが少なくなく[34]、逆に、成功したときほど、実は難なくスマートに物事が進んだ結果であることも多いのです。そこで、ここでは**プログラミングの成功の秘訣は「ずぼら」である**とでも述べておきましょう。**後々下手にがんばらないようにするために、今工夫する姿勢が大切です。**

プログラムの解説に戻りましょう。第 9 行の「print "i: #{i}　s: #{s}　p: #{p}"」は、変数 i、s、p の値を表示するものです。**変数の値を表示することで、プログラムの実行中にプログラムがどのような状態になっているのか（正しく動いているのかなど）を確認することができます。**何かを表示させることで状態を知るというのは、先に取り上げた由緒正しいプログラム「Hello World」と同じです。

第 10 行は空行なので飛ばして、第 11 行（「if i == 1」）以降は if などを用いた様々な場合分け（条件分岐）を行っています。この部分は元の問題文に記された計算手順に似ていますので、ここでは異なる部分を列挙したいと思います。

まずは、**値を比較して同じであるかどうかを判定するための演算子が、Ruby では「=」ではなく「==」になっています。**数学では等号「=」は代入、恒等式（変数にどのような値を代入しても成立するという関係を表す）や方程式（変数に特定の値を代入すれば成立するという関係を表す）の表現に漁されており、一つの記号が複数の意味を持っています。プログラミングでそのようにすることも不可能ではないのです[35]が、一つの記号に複数の意味を持たせると、意味の解釈が難しくなってしまいますので、通常は（プログラミング言語や、それに基づくイ

[34] 例えば、外国語の作文の試験で、がんばって書いたのに点数がもらえなかった経験はありませんか？　がんばって書いた文は、実は下手な文であることが多いのです。そもそも言語は苦労して使うものではありません。言語を使うだけで苦労していたら、言語を利用した上での活動が何もできないでしょう。

[35] 例えば、BASIC というプログラミング言語（Visual Basic を含む）では、変数に値を代入するための演算子と、値を比較して同じであるかどうかを判定するための演算子はいずれも「=」です。

ンタープリター・コンパイラーを作る人の都合で）異なる意味をもつものには異なる記号を割り当てることも多いのです。**Ruby では「=」は専ら代入**（元の問題文の計算手順における「←」に同じ）**を意味し、等号を 2 つ並べた「==」は両辺の値を比較して同じであるかどうかを判定するために使用されます**[36]。

実は、Ruby にはさらに、等号を 3 つ並べた「===」という演算子もあります。「===」という演算子は見た目からして奇怪に思われるかもしれませんが、これでも工夫の末生み出されたものなのです。それは、プログラミング言語の歴史を勉強するとより明確にわかりましょうが、それをここで述べるとまた長くなりそうですので、ここでは述べないでおきます。

他に異なる部分と言えば、変数に値を代入するための演算子が「←」ではなく「=」になっていること、場合分け（条件分岐）での「else if」が「elsif」になっていること、「endif」が「end」になっていること、**かけ算（乗算）を表現するには、単に定数や変数名を並べる**（例えば、「2p」）**だけではダメで、かけ算のための演算子「*」**（アスタリスクです。**「×」や中黒「・」ではありません**）**を挟まなければならない**こと、**左辺が右辺以上であるかどうかを判定する演算子は数学で親しまれている「≥」や「≧」ではなく、1 つの不等号と 1 つの等号を並べて書いた「>=」である**ことが挙げられます（**「>=」における文字の順番を入れ替えた「=>」はダメです。**これは別の意味を持ちます）。

細かいことを言えば、**引き算（減算）のための演算子は本式のマイナス記号「−」ではなく、ハイフンマイナス「-」です**。ハイフンマイナスとは、見た目がハイフンであって普通はハイフンとして使用されるものの、やむを得ずマイナス記号としても用いられることがあるものです。ハイフンマイナスは、コンピューターの処理能力が貧弱で、あまり多種多様な文字を扱うことができなかったときに、似たような形の文字を包摂したことによってできたものです。今でも、日本語入力機能を無効にした状態でキーボードから直接入力することができます（0 キーの右隣にあるはずです）が、本式にはハイフンマイナスはハイフンではありえても、マイナス記号ではありえないことに注意してください。従って、プログラミングではハイフンマイナスをマイナス記号として用いることは（それが言語上の規則なので）やむを得ませんが、その他の場合、例えばワープロなどで印刷用の文書を作る場合には、ハイフンマイナスをマイナス記号として用いるのは（本式のマイナス記号が利用できない環境を除いて）推奨されません。

似たようなことは、プログラミングでは掛け算（乗算）を表していることが多いアスタリスク「*」についても言えます。また、今回は登場しませんが、**割り算（除算）を行うための演算子はスラッシュ「/」です**。数式では、割り算（除算）を行う演算子には（国によって異なりますが）スラッシュ「/」、コロン「:」（「比」を表すものです）、「÷」（日本ではこれが一般的です）がありますが、Ruby も含めて多くのプログラミング言語では、割り算（除算）のための演算子はスラッシュ「/」一択です。

ちなみに、先ほど取り上げたプログラム（プログラム例 1）では、

```
if s == 0
  t = t + p
  s = 2
elsif s == 1
```

[36] この「=」と「==」は、実は多くのプログラミング言語で採用されています。例えば、C、C++、C#、D、Java。

```
    t = t + 2 * p
    s = 2
  elsif s == 2
    t = t + 2 * p
    s = 3
  else
    t = t + 3 * p
    s = 3
  end
```

のように、変数 s の値で場合分け（条件分岐）を行う if - （elsif） - （else） - end 構文の中で変数 s の値を変更していますが、変数 s の値を変更しているからといって、その瞬間に別の場合（別の条件が当てはまる場所）にプログラムの実行が飛んでいくということはありません。あくまで、変数 s の値の判定はプログラムの実行が if - （elsif） - （else） - end 構文に到達したときに限られます。

例えば、変数 s の値が 0 の場合には、「if s == 0」と「elsif s == 1」の間にある「t = t + p」、「s = 2」という処理が実行されますが、「s = 2」によって変数 s の値が 2 になったからといって、その瞬間に、変数 s の値が 2 の場合の処理「t = t + 2 * p」、「s = 3」（「elsif s == 2」と「else」の間にあります）が実行されることはありません。変数 s の値が 2 に変更されたのが場合分け（条件分岐）の結果に反映されるのは、次に変数 s に関する if - （elsif） - （else） - end 構文に到達するときです。

やや脱線してしまいましたので、本題に戻りましょう。先ほどのプログラム（プログラム例 1）をファイルに保存します。ここではファイルの名前を「problem2_1.rb」としましょう[37]。そして、「コマンド プロンプト」などのコンソール端末で「ruby problem2_1.rb」（「problem2_1.rb」は先ほど保存したファイルの名前です。ファイルの名前が異なる場合は読み替えてください）と入力して、以下のように表示されれば成功です。

```
i: 1  s: 0  p: 7 -> t: 7
i: 2  s: 0  p: 3 -> t: 10
i: 1  s: 1  p: 8 -> t: 26
i: 2  s: 1  p: 0 -> t: 26
i: 1  s: 0  p: 10 -> t: 36
i: 1  s: 2  p: 10 -> t: 56
i: 1  s: 3  p: 6 -> t: 74
i: 2  s: 3  p: 4 -> t: 82
i: 1  s: 1  p: 7 -> t: 96
i: 2  s: 1  p: 1 -> t: 97
Total score: 97
```

もし上記のように表示されなかった場合は、プログラムのどこかに誤りがあることでしょう。

例えば、プログラムの第 59 行（ちなみに、第 58 行は「print " -> t: #{t}¥n"」です）の「end」を取り除いてしまうと、以下のようなエラーメッセージが表示されます。

```
problem2_1.rb:61: syntax error, unexpected $end, expecting kEND
```

（日本語訳　problem2_1.rb 第 61 行: 統語エラーです。kEND が期待されているところで予期しない$end が出現しました。）

これは、プログラムの第 61 行でプログラムの終わりを迎えたものの、本当は「end」というキ

[37] MS-DOS など古い基本システムをお使いの場合は、ファイル名として「problem2_1.rb」を与えることはできないかもしれません（主にファイル名の長さの制限）。その場合は、ファイルの名前を別のもの（例えば、「prob2_1.rb」）に変え、これ以降の説明におけるファイル名の部分を読み替える必要があります。もっとも、今日（2009 年現在）主に用いられている基本システムで「problem2_1.rb」というファイル名を受け付けないようなものは皆無だと思いますが。

ーワード（またはそれに相当する、処理のまとまりの終わりを示すもの）が必要だったという意味のエラーメッセージです。

ところが、第 61 行は「print "Total score: #{t}¥n"」であり、この部分には誤りはありません。第 61 行とは別の場所に「end」を挿入しなければならないのですが、残念ながら、**プログラムの構造上の誤り**（今回のように、「do - end」、「if - （elsif） - （else） - end」の「end」を入れるのを忘れてしまったなど）**については、誤りがあるということは分かっても、その場所を的確に判定するのは困難である傾向にあります**。単なる誤植（例えば、「print」を「prunt」としてしまった）はすぐに発見できますが、構造の問題については、表面的な解決策（とりあえず、「do - end」、「if - （elsif） - （else） - end」などといったものの釣り合いをとるだけ）は複数考えられますので、正しい解決策をなかなかズバリ言い当てられないのです。

このような誤りについてはがんばって探すしかないのですが、がんばるのは大変です。そこでお勧めしたいのは、**いきなり大きなプログラムを作らずに、小さなプログラムから始めて、作業が一段落したら試しにプログラムを実行してみて、それまでの作業内容が正しいかどうかを確認することをできるだけ行う**ことです。そうすれば、プログラムの誤りが発生しても、その誤りを特定するのが比較的容易になることでしょう。

プログラムを工夫してみよう

さて、期待通りにプログラムが動けば、そのプログラムは正しいということになります。しかし、**工夫することで、より短く、読みやすいプログラムにすることができます**。このことは、全体が数万行またはそれ以上にもなる大規模なプログラミングでは特に重要になります。

工夫の一つは、**処理のまとまりに名前を付ける**ことです。以下のプログラムをご覧ください。

プログラム例 3

```
pins = [7, 3, 8, 0, 10, 10, 6, 4, 7, 1]

$t = 0
$s = 0
$r = 0
$i = 1

def proc_a(p)
  if $s == 0
    $t = $t + p
    $s = 2
  elsif $s == 1
    $t = $t + 2 * p
    $s = 2
  elsif $s == 2
    $t = $t + 2 * p
    $s = 3
  else
    $t = $t + 3 * p
    $s = 3
  end
  $i = 1
end

def proc_b(p)
```

```
  if $s == 0
    $t = $t + p
  elsif $s == 1
    $t = $t + 2 * p
  elsif $s == 2
    $t = $t + 2 * p
  else
    $t = $t + 3 * p
  end
  $r = 10 - p
  $i = 2
end

def proc_c(p)
  if $s == 0
    $t = $t + p
  elsif $s == 1
    $t = $t + p
  elsif $s == 2
    $t = $t + 2 * p
  else
    $t = $t + 2 * p
  end
  if p >= $r
    $s = 1
  else
    $s = 0
  end
  $i = 1
end

pins.each do |p|
  print "i: #{$i}  s: #{$s}  p: #{p}"

  if $i == 1
    if p == 10
      proc_a(p)
    else
      proc_b(p)
    end
  else
    proc_c(p)
  end

  print " -> t: #{$t}¥n"
end

print "Total score: #{$t}¥n"
```

ここでは、元の問題文の処理 A（「A の計算手順」）に「proc_a」、処理 B（「B の計算手順」）に「proc_b」、処理 C（「C の計算手順」）に「proc_c」という名前（識別子（identifier））を付け、本筋（第 57 行（「pins.each do |p|」）から第 73 行（「print "Total score: #{$t}¥n"」）まで）にはそれらの名前だけを記しておき（第 62，64，67 行）、その具体的な内容を本筋の部分から切り離しています。このような処理のまとまりを**関数**（function）、**手続き**（procedure）とか**メソッド**（method）などと言います。どの用語が採用されるのかは主に習慣の問題です[38]。

[38] ただし、プログラミング言語によっては、これらの用語が異なる意味をもつものとして用いられることがあります。例えば、Pascal というプログラミング言語では、処理のまとまりのうち、数学の関数（三角関数、指数関数、あるいはより一般に「$f(x)$」などで代表されるもの）のように値を返すものを「関数」（function）、値を返さないものを「手続き」（procedure）と言います。これは、Pascal の利用者には数学者が多く、数学では値を返すというのが関数としては必須の性質の一つであるのに、プログラミングでは値を返さないものに「関数」と呼ぶのが不自然に思われて嫌だった

ここで使用した、処理のまとまりの名前に含まれている「proc」というのは、「手続き」という意味を持つ英語「procedure」を省略したものです。変数や、関数、手続き、メソッドにはある程度（規則の範囲内で）好きな名前をつけられます。コンピューターからすれば、名前は単にものを（他のものと）区別するためのものに過ぎません[39]ので、同じ使い方をする限りでは、変数や、関数、手続き、メソッドの名前を変えても、プログラムの実行結果は何ら変化しません[40]。

変数や、関数、手続き、メソッドに好きな名前をつけられると言っても、**名前をあまり長くしすぎると、プログラムの行が画面の幅より長くなり、画面をスクロールさせないと行を全部見ることができなくなってしまい、面倒になってしまいます**。かと言って、**一時的にしか用いないものを除いて、何でも「a」とか「b」といった意味のない短い名前にすると、その変数や関数、手続き、メソッドがどのような意味を持って、何に用いられるのかがプログラマーにとって分かりづらくなってしまいます**[41]。

そこで、**適度に語句を略記する**という方法がよく用いられます。よく用いられる（またはその可能性の高い）略語を以下に示します（ただし、略語によっては複数の解釈が認められて、紛らわしくなる可能性がありますので注意してください）。これらの略語は今すぐに覚えなければならないものではありませんが、頻繁に見たり使ったりするようであれば意識してみてください。

略語	意味
abs	絶対値（absolute value）
app	アプリケーションソフトウェア（application software）
arg	引数（argument）
attr	属性（attribute）
bak、bk	バックアップ（backup）
bin	二進法の、バイナリー（binary）
btn	ボタン（button）
buf	バッファー（buffer）
cd	コンパクトディスク（compact disc[42]）
	現在のディレクトリー（current directory）
c、ch、char	文字（character）
cmd	命令（command）
cmp	比較（comparison）
cnt	個数（count）
ctrl	操作（control）
dec	十進法の（decimal）
	減少（decrement）
del	削除（deletion）
def	定義（definition）
desc	記述、説明（description）
dest	～先（destination）
dic	辞書（dictionary）
diff	差分（difference）
dir	ディレクトリー（directory）

からなのでしょう。

[39] だから、変数や、関数、手続き、メソッドなどの名前はしばしば「**識別子**」（identifier）と呼ばれるのです。

[40] 実は、**リフレクション**（reflection）（プログラムの実行中に、プログラムが自らの構造を読んだり書き換えたりすること）という機能を使うと、プログラム中に使用された名前（識別子）を取得したり操作したりすることもできます。

[41] 今回のプログラムも本当は、合計得点を格納する変数の名前を「t」ではなく、「total_score」にする方がよいかもしれませんが、それは場合によります。変数の意味があまりにも簡単に分かる場合、または、変数に特別な意味がない場合は、1～2 文字の名前で済ませることも少なくありません。

[42] 「compact disk」ではありません。

略語	意味
doc	文書（document）
dup	複製（duplication）
enc	エンコーディング（encoding）
env	環境（environment）
eof	ファイルの終端（end of file）
err	誤謬、エラー（error）
ex、exp、expr	式（expression）
ex、ext	拡張（extension）
	外部の（external）
exp	（べき乗などの）指数（exponent）
fd	ファイル記述子（file descriptor）
	フロッピーディスク（floppy disk）
fmt	形式（format）
func	関数（function）
hex、hexa	十六進法の（hexadecimal）
i、int	整数（integer）
inc	増加（increment）
img	画像（image）
init	最初の（initial）
	初期化（initialization）
ins	挿入（insertion）
kb	キーボード（keyboard）
	キロビット（kilobits）
	キロバイト（kilobytes）
lang	言語（language）
len	長さ（length）
lib	ライブラリー（library）
mem	メモリー（memory）
num	数、番号（number）
obj	オブジェクト（object）
oct	八進法の（octal）
op	演算子（operator）
p、ptr	ポインター（pointer）
para、param	媒介変数（parameter）
pos	位置（position）
prev	直前の（previous）
proc	手続き（procedure）
	プロセス（process）
pt	点（point）
rand、rnd	乱数（random value）
rect	長方形、矩形（rectangle）
ref	参照（reference）
regex、regexp	正規表現（regular expression）
s、str	文字列（string）
scr	画面（screen）
src	源、～元（source）
std	標準（standard）
tbl	表（table）
temp、tmp	一時的な（temporary）
val	値（value）
var	変数（variable）

傾向としては、変数よりも関数、手続きやメソッドの方が、略語があまり用いられないような気がします（たぶん…）。

ところで、これまで変数や、関数、手続き、メソッドの名前は英語またはそれを基にしたものを用いてきましたが、英語が母国語でない方は母国語（例えば、中国語、日本語や韓国語）で名前を書きたいと思われるかもしれません。例えば、日本語を使うと以下のような感じになります（ここではメソッド「proc_a」を例に出して書き換えています）。

```
def 処理A(p)
```

```
    if $直前フレームの状態 == 0
      $合計得点 = $合計得点 + 倒したピンの数
      $直前フレームの状態 = 2
    elsif $直前フレームの状態 == 1
      $合計得点 = $合計得点 + 2 * 倒したピンの数
      $直前フレームの状態 = 2
    elsif $直前フレームの状態 == 2
      $合計得点 = $合計得点 + 2 * 倒したピンの数
      $直前フレームの状態 = 3
    else
      $合計得点 = $合計得点 + 3 * 倒したピンの数
      $直前フレームの状態 = 3
    end
    $フレーム中の投球の番号 = 1
  end
```

しかし、それはプログラミング言語やプログラミングの環境によっては（技術的に）できないことがあります。また、（技術的に）できたとしても、習慣の問題で、普通は、名前は英語で書くことになっています。他の人が書いたプログラムで、プログラムの実行に直接的な影響を及ぼさない文字列やコメントを除き、英語以外の言語で使用される文字が出てくることは（技術的に可能だとしても）ほとんどありません。

また、名前に複数の単語を使用したいときは、単語をアンダースコア「_」で区切ったり、各単語の最初の文字を英大文字（capital letter）に変えたり（名前の最初の文字だけ英小文字のままにするという流儀もあります）します。例えば、「合計得点」を意味する「total score」を格納する変数の名前は「total_score」、「TotalScore」、「totalScore」などが考えられます。

複数の単語を使用するときにどのように名前を構成するのかは習慣によります。Ruby の場合、変数やメソッドの名前は英小文字でアンダースコア「_」区切り（例　num_of_files（number of files の意味で））、定数は英大文字でアンダースコア「_」区切り（例　NUM_OF_BITS_PER_BYTE（number of bits per byte の意味で））、クラス（説明するとそれなりに長くなりますが、ここでは、変数、定数やメソッドをひとまとめにし、一連の機能を提供するためのもの、と考えてください）の名前は各単語の先頭の文字を英大文字でそれ以外は英小文字（アンダースコア「_」は使用しない）とするのが習慣です。この習慣に従わなくてもプログラムの実行には影響しないのですが、習慣に従うと他の人との意思伝達が若干容易になりますので、通常は習慣に従われることをお勧めします[43]。

ところで、メソッドで「proc_a」、「proc_b」、「proc_c」という名前はそれぞれ「処理 A」、「処理 B」、「処理 C」をほぼ直訳したものです。本来なら、関数、手続きやメソッドの名前は変数の場合と同様に、内容を端的に表すものにしておけば、（複雑であることも少なくない中身をわざわざ見なくても）一目で何をするのかが分かりますので都合が良いのですが、今回の場合、そのようにしようとすると、例えば、「処理 A」は「ストライクを出した場合における、合計得点と直前のフレームの状態を変更する」処理ですから、

```
change_total_score_and_previous_frame_state_if_strike_is_scored
```

のような長々しい名前になってしまいがちです。略語を駆使しても、せいぜい

```
change_score_and_prev_frm_stat_if_strike
```

[43] もっとも、この習慣が実はただの都市伝説である（実際にはあまり用いられていない）可能性もありますので、実際に Ruby で書かれたさまざまなプログラムを読まれることをお勧めします。もちろん、人によって習慣の使い方は異なることも少なくありません。都合が悪ければ、一部の習慣を捨てることもあります。

（「frame」は「frm」に、「state」は「stat」（「statistics」（統計結果）と紛らわしい？）に略してみました。）

ぐらいにしかならないでしょう。そこで、やむを得ず「proc_a」などというつまらない名前にしました（コメントという、プログラムの実行に関与しない説明を付けることもできますが、それについては後述します）。

ちなみに、メソッドの名前の隣にある「(p)」は**引数**（argument）を表すもので、メソッドの外部からメソッドの内部に値を渡すために使用されます。なぜなら、これがないと、**原則として、メソッドの内部はメソッドの外部から値を取り出すことができない**からです。

例外は、変数の名前の先頭が「$」になっている**グローバル変数**（global variable）で、これは、プログラム中のどこからでもアクセス（内容を読み書きすること）可能です。本当は、**グローバル変数も、注意して取り扱わないと、変数とメソッドの依存関係が複雑になり、プログラムを編集しづらくなる**のですが、今回はプログラムとしては単純ですので、使用させていただきました。要するに、これは程度の問題です。

いずれにせよ、**メソッドなどを用いて、処理のまとまりを適当に区切って名前を付けると、部分的には何をするのかが分かり、構造的に理解しやすくなります**。

もっとプログラムを工夫してみよう

工夫は他にもあります。例えば、今回のプログラムでは、

```
if $s == 0
  （何らかの処理）
elsif $s == 1
  （何らかの処理）
elsif $s == 2
  （何らかの処理）
else
  （何らかの処理）
end
```

のような場合分け（条件分岐）が随所に登場しますが、このように、場合が多数あるような場合分け（条件分岐）はプログラムにすると長くなりがちですが、これも工夫次第で短く表現することができます。

例えば、先ほどのプログラム（プログラム例 3）におけるメソッド「proc_b」は

```
def proc_b(p)
  if $s == 0
    $t = $t + p
  elsif $s == 1
    $t = $t + 2 * p
  elsif $s == 2
    $t = $t + 2 * p
  else
    $t = $t + 3 * p
  end
  $r = 10 - p
  $i = 2
end
```

となっていますが、変数$s（これは名前が「$」で始まっていますので、グローバル変数なのでした）が 1 の場合と 2 の場合とではいずれも処理の内容が

```
$t = $t + 2 * p
```

で全く同じです。

このような場合は、以下のように、場合の条件を工夫することで統合できます。

```
def proc_b(p)
  if $s == 0
    $t = $t + p
  elsif ($s == 1) || ($s == 2)
    $t = $t + 2 * p
  else
    $t = $t + 3 * p
  end
  $r = 10 - p
  $i = 2
end
```

ここでは、「||」という演算子を使用しています。「||」は難しい言葉で「**論理和**」を意味しますが、これは要するに「～または～」(「～」には条件が入ります）を意味するものです。例えば、上記のプログラムで使用した「($s == 1) || ($s == 2)」は「変数$s の値が 1 に等しい、または、変数$s の値が 2 に等しい」という意味です。

余談ですが、「($s == 1) || ($s == 2)」の括弧は、実はなくてもよく、「$s == 1 || $s == 2」とも書くこともできますが、ここでは分かりやすさのために、括弧を挿入しました[44]。

このように、条件が「$s == 1」と「$s == 2」で分かれていたのをまとめて「($s == 1) || ($s == 2)」とすることで、同じ処理をまとめることができました。このようにすることで、プログラムが短くなって読みやすくなるだけでなく、プログラムを修正する際にも、修正箇所が減って、修正し忘れ（誤りの一種）を減らすことができます。

しかし、実は、「($s == 1) || ($s == 2)」をさらに分かりやすく書き換える方法が Ruby にはあります。それは、次のようにメソッド「between?」を用いるものです。

```
def proc_b(p)
  if $s == 0
    $t = $t + p
  elsif $s.between?(1, 2)
    $t = $t + 2 * p
  else
    $t = $t + 3 * p
  end
  $r = 10 - p
  $i = 2
end
```

メソッド「between?」は、2 つの引数をとり、それとは別に与えられた数が、引数に指定された 2 つの数のいずれかと同じ、または、その 2 つの数の間にあるかどうかを判定するものです。例えば、「a.between?(b, c)」は$b \leq a \leq c$という条件が満たされる場合、またその場合に限りに真（当てはまる）となります。

---

[44] 実は、これは「**演算子の優先順位**」に関わるものです。括弧でくくられた部分は先に**評価される**（evaluated）（これは少々難しい語句でしょうが、ここではとりあえず「効果が発動する」と思っておいてもよいでしょう）のですが、括弧がなくても、演算子「==」よりも演算子「||」の方が先に評価されるため、評価の順序は変わらないというわけです。ちなみに、Ruby には、「||」と同様に「or」という演算子もありますが、これは「||」とは演算子の優先順位が異なるだけです（「or」は「==」よりも後に評価されるために、「$s == 1 or $s == 2」は「$s == (1 or $s) == 2」（これは何を意味するのか!?）と結果が同じになり、「($s == 1) or ($s == 2)」とは結果が異なります）。しかし、他のプログラミング言語（特に C など）では「||」だけが論理和（～または～）を表すものだったことから、「or」よりも「||」が用いられることが多いようです。また、演算子の優先順位をきちんと全て覚えている人は、実は実務でプログラミングを行っている人ですらあまりいないと思います。だからこそ、括弧を用いて評価の優先順位を明示するのには意味があるのです。

ここでは、「$s.between?(1, 2)」と書かれた部分があり、これは「変数$s の値が 1 以上 2 以下（1 や 2 も含まれます）であれば真（当てはまる）とし、そうでなければ偽（当てはまらない）とする」ことを意味します。また、「$s.between?(1, 2)」は「($s >= 1) && ($s <= 2)」と書いた場合と同じ結果が得られます。ちなみに、「($s >= 1) && ($s <= 2)」は「($s == 1) || ($s == 2)」とは異なる条件のはずですが、今は変数$s は整数以外をとらないことになっていますので、どちらの条件も「変数$s の値は 1 または 2 である」と読み替えることができます。

今回のプログラムで用いられている「&&」という演算子は難しい言葉で「**論理積**」を意味しますが、これは要するに「～または～」（「～」には条件が入ります）を意味するものです。例えば、上記のプログラムで使用した「($s >= 1) && ($s <= 2)」は「変数$s の値が 1 以上で、かつ、変数$s の値が 2 以下である」という意味です。「||」の場合と同様に、括弧をはずして「$s >= 1 && $s <= 2」と書くこともできるのですが、括弧をつけた方が見やすいと思われます[45]。

ところで、「変数$s の値は 1 以上 2 以下である」を表現する際に、数学では$1 \leq s \leq 2$と書くのが普通ですが、Ruby を含めて多くのプログラミング言語では、数学の「≦」を「<=」と書くことを考慮に入れても、**「1 <= $s <= 2」というように書くことはできません**。なぜなら、「<=」という演算子は、左右 2 つの値を比べてすぐに（3 つ目の値と比較する前に）結果を出してしまうからです（つまり、「(1 <= $s) <= 2」に同じ）。そこで、仕方なく「($s >= 1) && ($s <= 2)」と表現しなければならないプログラミング言語が多いのですが、Ruby にはメソッド「between?」が用意されていますので、「($s >= 1) && ($s <= 2)」よりは直感的に書けると思います。

また余談ですが、Ruby ではメソッドの名前に疑問符「?」を付けることができます。ただし、疑問符「?」は付けるとしても普通最後にしか付けません。Ruby では、条件に合致しているかどうかの判断を行うメソッドには名前の最後に疑問符「?」を付ける習慣があります[46]。

統合できるところは他にもあります。プログラム（プログラム例 3）におけるメソッド「proc_c」は以下のようになっています。

```
def proc_c(p)
  if $s == 0
    $t = $t + p
  elsif $s == 1
    $t = $t + p
  elsif $s == 2
    $t = $t + 2 * p
  else
    $t = $t + 2 * p
  end
  if p >= $r
    $s = 1
  else
    $s = 0
  end
  $i = 1
```

[45] やはり「||」と「or」の関係と同様に、Ruby には、「&&」とは演算子の優先順位が異なるが、それ以外は同じである「and」という演算子もあります。「and」という演算子を用いた場合は、やはり括弧を省略すると異なる結果が得られます（「and」は「==」よりも後に評価されるために、「$s >= 1 and $s <= 2」は「$s >= (1 and $s) <= 2」（これは何を意味するのか!?）と結果が同じになり、「($s >= 1) or ($s <= 2)」とは結果が異なります）。そして、実際には「&&」の方が「and」よりも多く用いられていることでしょう。

[46] 他の多くのプログラミング言語では、英語の疑問文に見られるように、関数、手続きやメソッドの名前としては動詞に相当する語句を先頭に置くことが多いのです（例　「is_between」、「completed」）。また、多くのプログラミング言語では、疑問符「?」は関数、手続きやメソッドの名前としては使用できないことに注意が必要です。

```
  end
```

このうち、

```
    if $s == 0
      $t = $t + p
    elsif $s == 1
      $t = $t + p
    elsif $s == 2
      $t = $t + 2 * p
    else
      $t = $t + 2 * p
    end
```

という部分についてですが、これを見れば分かるとおり、変数$s の値が 0 の場合と 1 の場合とでは処理の内容が完全に同じ（「$t = $t + p」）であり、変数$s の値が 2 の場合と 3 の場合とでは処理の内容が完全に同じ（「$t = $t + 2 * p」）です。

これらは結局、変数$s の値が 1 以下なのか 2 以上なのかが境目ですから、それに従った場合分け（条件分岐）に書き換えておきましょう。

```
    if $s <= 1
      $t = $t + p
    else
      $t = $t + 2 * p
    end
```

上記のように工夫した場合のプログラム全体を以下に示します。

プログラム例 4

```
pins = [7, 3, 8, 0, 10, 10, 6, 4, 7, 1]

$t = 0
$s = 0
$r = 0
$i = 1

def proc_a(p)
  if $s == 0
    $t = $t + p
    $s = 2
  elsif $s == 1
    $t = $t + 2 * p
    $s = 2
  elsif $s == 2
    $t = $t + 2 * p
    $s = 3
  else
    $t = $t + 3 * p
    $s = 3
  end
  $i = 1
end

def proc_b(p)
  if $s == 0
    $t = $t + p
  elsif $s.between?(1, 2)
    $t = $t + 2 * p
  else
    $t = $t + 3 * p
  end
  $r = 10 - p
  $i = 2
end
```

```
def proc_c(p)
  if $s <= 1
    $t = $t + p
  else
    $t = $t + 2 * p
  end
  if p >= $r
    $s = 1
  else
    $s = 0
  end
  $i = 1
end

pins.each do |p|
  print "i: #{$i}  s: #{$s}  p: #{p}"

  if $i == 1
    if p == 10
      proc_a(p)
    else
      proc_b(p)
    end
  else
    proc_c(p)
  end

  print " -> t: #{$t}¥n"
end

print "Total score: #{$t}¥n"
```

ところで、場合分け（条件分岐）の部分はよく考えてみると、変数$s の値が何なのかによるものであって、変数$s が繰り返し使用されています。この場合は、今まで行ってきたように単に if - （elsif） - （else） - end 構文を用いるよりも簡単な方法があります。それは、**case - when - end 構文**を使用するというものです。これは、「case」の直後に値を判定したい変数（または式）を置き、「when」の直後でその変数（または式）がとる値を条件として指定し、さらにその後に、その条件に当てはまる場合の処理を書くというものです。

例えば、プログラム例 3 におけるメソッド「proc_a」は以下のようになっています。

```
def proc_a(p)
  if $s == 0
    $t = $t + p
    $s = 2
  elsif $s == 1
    $t = $t + 2 * p
    $s = 2
  elsif $s == 2
    $t = $t + 2 * p
    $s = 3
  else
    $t = $t + 3 * p
    $s = 3
  end
  $i = 1
end
```

この if - （elsif） - （else） - end 構文を case - when - end 構文で書き換えると、以下のようになります（上記の最後の else は、実質的には変数$s の値が 3 の場合を指していることに注意してください)。

```
def proc_a(p)
  case $s
  when 0
    $t = $t + p
    $s = 2
  when 1
    $t = $t + 2 * p
    $s = 2
  when 2
    $t = $t + 2 * p
    $s = 3
  when 3
    $t = $t + 3 * p
    $s = 3
  end
  $i = 1
end
```

これによると、変数$s の値が 0 の場合は、「when 0」と「when 1」にはさまれている「$t = $t + p」、「$s = 2」という処理が実行されます[47]。また、変数$s の値が 1 の場合は、「when 1」と「when 2」にはさまれている「$t = $t + 2 * p」、「$s = 2」という処理が実行されます。変数$s の値が when で指定したもののいずれでもない場合のための処理を、else を用いて指定することもできます。例えば、変数$s の値は 0、1、2、3 のいずれかですから、3 は 0、1、2 のいずれでもないと考えられます。そこで、「when 3」を「else」に書き換えても結構です（以降でもそのようにします）。

```
def proc_a(p)
  case $s
  when 0
    $t = $t + p
    $s = 2
  when 1
    $t = $t + 2 * p
    $s = 2
  when 2
    $t = $t + 2 * p
    $s = 3
  else
    $t = $t + 3 * p
    $s = 3
  end
  $i = 1
end
```

また、if - （elsif） - （else） - end 構文と同様に、case - when - end 構文でも、途中で変数の値が変わったからといって、その瞬間に別の場所にプログラムの実行が飛ぶことはありません。

when には、コンマ「,」で区切って複数の値を指定することもできます。例えば、プログラム（プログラム例 3）におけるメソッド「proc_b」は以下のようになっていました。

```
def proc_b(p)
  if $s == 0
    $t = $t + p
  elsif $s == 1
    $t = $t + 2 * p
  elsif $s == 2
    $t = $t + 2 * p
  else
    $t = $t + 3 * p
```

[47] 少々難しい話ですが、case - when - end 構文では、値の判定を演算子「==」（完全一致）ではなく演算子「===」（等号「=」を 3 つ並べたもの）で行います。演算子「===」はとりあえずは「含まれる」とか「当てはまる」とでも和訳するとよいでしょう。

```
    end
    $r = 10 - p
    $i = 2
  end
```

この if - （elsif） - （else） - end 構文をそのまま case - when - end 構文で書き換えると、以下のようになります。

```
  def proc_b(p)
    case $s
    when 0
      $t = $t + p
    when 1
      $t = $t + 2 * p
    when 2
      $t = $t + 2 * p
    else
      $t = $t + 3 * p
    end
    $r = 10 - p
    $i = 2
  end
```

ところが、変数$s の値が 1 の場合（「when 1」）と値が 2 の場合（「when 2」）とでは処理の内容が完全に同じ（「$t = $t + 2 * p」）ですから、これらは、以下のように、when に 1 と 2 の両方を指定することで統合できます。

```
  def proc_b(p)
    case $s
    when 0
      $t = $t + p
    when 1, 2
      $t = $t + 2 * p
    else
      $t = $t + 3 * p
    end
    $r = 10 - p
    $i = 2
  end
```

ここでは、以下のように、コンマ「,」の直後に 1 つの半角（欧文用）空白を挿入しています。

```
  when 1, 2
```

実際には、以下のように、この半角（欧文用）空白がなくてもプログラムの実行に影響はありません。

```
  when 1,2
```

しかし、半角（欧文用）空白がある方が区切りとしては分かりやすいという理由で、半角（欧文用）空白を付ける方がよいでしょう。

また、複数の値のいずれかに当てはまるような場合の条件を書く方法としては、上記のようにコンマ「,」で複数の値を区切ることの他に、範囲指定のための演算子（**範囲式**）を使用するというものがあります。範囲と言うと、普通（プログラミングではなく、印刷用の文書を作る場合）はエヌダッシュ（en dash）「–」を使用したり、波線「～」を使用したりします（例えば、「1 から 3 まで」を略記して「1–3」とか「1～3」などと書きます）[48]が、Ruby ではそれらのいずれも使用できず、代わりに「..」や「...」を使用します。

[48] 日本語では波線「～」を用いるのが普通ですが、英語では範囲を指定するのに波線「～」は使用できません。代わりにエヌダッシュ（en dash）「-」を使用します。ちなみに、エヌ（en）とは、元々英語の小文字・エヌ「n」の横幅の長さを表すものですが、今日では半角の幅と同じ意味で用いられることが多いようです。

「..」を使用した場合は、「..」の左右にある値も範囲に含まれますが、「...」については、左にある値は範囲に含まれますが、右にある値は範囲に含まれません。例えば、「1 .. 10」は 1 から 10 までで、両端を含むもの（$1 \leq x \leq 10$）を表しますが、「1 ... 10」は右端である 10 を含みません（$1 \leq x < 10$）。

この範囲式を利用して、以下のようにプログラムを書くこともできます。

```
def proc_b(p)
  case $s
  when 0
    $t = $t + p
  when 1 .. 2
    $t = $t + 2 * p
  else
    $t = $t + 3 * p
  end
  $r = 10 - p
  $i = 2
end
```

そうすれば、変数$s の値が 1 から 2 までに含まれる場合は、「when 1 .. 2」に該当し、処理「$t = $t + 2 * p」が実行されるというわけです。

いずれにせよ、case - when - end 構文を用いると、対象となる変数（または式）（ここでは変数$s のこと）を複数回書かなくても済むようになります。この工夫を用いた場合のプログラムを以下に示します。

プログラム例 5

```
pins = [7, 3, 8, 0, 10, 10, 6, 4, 7, 1]

$t = 0
$s = 0
$r = 0
$i = 1

def proc_a(p)
  case $s
  when 0
    $t = $t + p
    $s = 2
  when 1
    $t = $t + 2 * p
    $s = 2
  when 2
    $t = $t + 2 * p
    $s = 3
  else
    $t = $t + 3 * p
    $s = 3
  end
  $i = 1
end

def proc_b(p)
  case $s
  when 0
    $t = $t + p
  when 1, 2
    $t = $t + 2 * p
  else
```

```
    $t = $t + 3 * p
  end
  $r = 10 - p
  $i = 2
end

def proc_c(p)
  case $s
  when 0, 1
    $t = $t + p
  else
    $t = $t + 2 * p
  end
  if p >= $r
    $s = 1
  else
    $s = 0
  end
  $i = 1
end

pins.each do |p|
  print "i: #{$i}  s: #{$s}  p: #{p}"

  if $i == 1
    if p == 10
      proc_a(p)
    else
      proc_b(p)
    end
  else
    proc_c(p)
  end

  print " -> t: #{$t}¥n"
end

print "Total score: #{$t}¥n"
```

もっともっとプログラムを工夫してみよう

ところが、これでもまだ序の口。さらに強烈な場合分け（条件分岐）の工夫があります。それは**配列**（array）を利用することです。

配列とは、複数の値をひとつにまとめたもののことです。単なる数の塊と言えましょう。また、**配列に番号を指定すると、配列からその番号に対応する場所にあるひとつの数を取り出すことができます。**例えば、数学では、配列に相当する数列（sequence of numbers）$\{a_n\}$を

$$\{a_n\} = \{2, 3, 5, 7, 11, 13, 17\}$$

と定義すると、「a_1」のようにnに 1 を代入すれば、数列の中で 1 番目の数である 2 が得られ、「a_2」のようにnに 2 を代入すれば、数列の中で 2 番目の数である 3 が得られます。

ところが、数列は、nに代入する数によって取り出される数が変化することから、以下のように場合分けを利用した関数として書くこともできます。

$$a_n = f(n) = \begin{cases} 2, & n = 1 \\ 3, & n = 2 \\ 5, & n = 3 \\ 7, & n = 4 \\ 11, & n = 5 \\ 13, & n = 6 \\ 17, & n = 7 \end{cases}$$

このことからも分かるように、**配列は数をまとめて扱うだけでなく、場合分け（条件分岐）にも利用できるのです**。

ここで、配列を用いて書き換える前のプログラムを見てみましょう。プログラム例 3 におけるメソッド「proc_b」は以下のようになっています。

```
def proc_b(p)
  if $s == 0
    $t = $t + p
  elsif $s == 1
    $t = $t + 2 * p
  elsif $s == 2
    $t = $t + 2 * p
  else
    $t = $t + 3 * p
  end
  $r = 10 - p
  $i = 2
end
```

このうち、場合分け（条件分岐）が用いられている

```
  if $s == 0
    $t = $t + p
  elsif $s == 1
    $t = $t + 2 * p
  elsif $s == 2
    $t = $t + 2 * p
  else
    $t = $t + 3 * p
  end
```

という部分についてですが、一般には、変数$s の値によって処理の内容が異なっています。しかし、いずれの処理も、「変数$t に変数 p の値の何倍かを加算する」という内容になっています。異なるのは、この「何倍か」という部分のみです。そこで、この「何倍か」という部分を配列で表現し、処理を一つにまとめてみましょう。

Ruby では、配列は角括弧「[」、「]」で中身をくくって表現します。ここでは、得点の倍率を表す変数 mul（「multiplier」の略）を用意し、この変数に配列を代入します。

```
mul = [1, 2, 2, 3]
```

配列から値を取り出すときには、配列の右に、配列中の位置を示す番号を角括弧「[」、「]」でくくったものを付けます。ただし、**配列中の位置を示す番号は、最初が 0 です**。これは、多くのプログラミング言語で共通です[49]。例えば、[1, 2, 2, 3]という配列では、先頭にある「1」が「0 番目」、その次にある「2」が「1 番目」、さらにその次にある「2」が「2 番目」、そして最後の「3」が「3 番目」というような感じになります。日常生活では、これらはそれぞれ「1 番目」、「2 番目」、「3 番目」、「4 番目」となるはずで、プログラミングではそれらよりも 1 だけ小さい

[49] 例外としては、Pascal では配列中の位置を示す番号は 1 から始まります。また、BASIC では、標準状態では配列中の位置を示す番号は 0 から始まりますが、1 から始まるように設定を変えることもできます。

番号で表現されるというわけです。そこで、例えば、プログラム中で「mul[0]」と書くと、変数 mul に格納されている配列の中で、プログラミングで言う「0 番目」（日常生活での「1 番目」）に位置する値、すなわち 1 が取り出されます。

このようにして配列を作ったら、その配列の中から取り出す値を決めるために、直前のフレームの状態を表す変数$s を、配列中の位置を示す番号として使用するのです。そのことを表現したのが「mul[$s]」です。変数$s の値が 0 なら、「mul[$s]」は「mul[0]」を意味し、1 が取り出されます。一方、変数$s の値が 1 なら、「mul[$s]」は「mul[1]」を意味し、変数 mul に格納されている配列の中で、プログラミングで言う「1 番目」（日常生活での「2 番目」）に位置する値、すなわち 2 が取り出されます。

これらを利用して、先ほど場合分け（条件分岐）が用いられている箇所として取り上げた部分

```
if $s == 0
  $t = $t + p
elsif $s == 1
  $t = $t + 2 * p
elsif $s == 2
  $t = $t + 2 * p
else
  $t = $t + 3 * p
end
```

を以下のように書き直すのです。

```
mul = [1, 2, 2, 3]
$t = $t + mul[$s] * p
```

何と、9 行あったものがたったの 2 行になってしまいました！ **場合分け（条件分岐）は場合の種類が多くなるとプログラムが長くなってしまう傾向にあり、プログラムを見づらくする原因になりやすいのですが、配列を利用すると場合分け（条件分岐）が非常に簡潔に表現できることもあるのです。**

少々脱線して、配列を用いた場合分け（条件分岐）の方法をもう少し取り上げて見ましょう。これから、月を英語の表現に置き換えることを考えて見ましょう。1 から 12 までの整数をひとつ与えたら、それを月と見なして、それに対応する英語の表現を得るようにします。ここでは、元となる月の番号を変数 month に格納し、それに対応する英語の表現を変数 en_month（「English month」の略）に格納し、その中身を表示することにします。

もし、配列を用いずに、場合分け（条件分岐）では平凡な case - when - end 構文を用いると、プログラムは以下のようになります（第 1 行により、変数 month の値が 1 なので、最終的に「January」と表示されるようなプログラムになっています。これを他の月に変えたい場合は、変数 month の値を変えてください）。

```
month = 1
en_month = nil

case month
when 1
  en_month = "January"
when 2
  en_month = "February"
when 3
  en_month = "March"
when 4
```

```
  en_month = "April"
when 5
  en_month = "May"
when 6
  en_month = "June"
when 7
  en_month = "July"
when 8
  en_month = "August"
when 9
  en_month = "September"
when 10
  en_month = "October"
when 11
  en_month = "November"
when 12
  en_month = "December"
end

print en_month
```

第 2 行に「en_month = nil」という行が見られますが、これはこれから en_month を使用するという宣言です。実は、最後の「print en_month」で変数 en_month を使用することになっているのですが、変数 en_month に一度も値を代入したり、その他の宣言をしなかったりすると、

```
undefined local variable or method `en_month' for main:Object (NameError)
```

（日本語訳　ローカル変数またはメソッド en_month は未定義です。）

という Ruby からのエラーメッセージが表示されてしまい、プログラムを正しく実行できなくなってしまいます。case - when - end 構文には変数 en_month に値が代入されるような処理がありますが、変数 month の値が 1 から 12 まで以外の場合はそのような代入が行われないので、「変数 en_month に値が一回も代入されない可能性がある」という意味でエラーになるわけです。そのため、やむを得ず「en_month = nil」という、一見無駄に見える行を挿入したのです。ちなみに、**「nil」は、無効、または無意味を意味する値です**。

このプログラムを、配列を利用して書き換えると以下のようになります。

```
month = 1
en_month_list = [nil, "January", "February", "March", "April", "May", "June", "July",
"August", "September", "October", "November", "December"]

print en_month_list[month]
```

変数名の「en_month」を何回も書かなくて済むようになりました。

ここで注意したいのは、既に説明したように、Ruby の配列中では位置を示す番号が 0 から始まりますので、1 月（変数 month の値が 1）に相当するものは、対応する英語の表現が格納されている変数 en_month_list の配列では、プログラミングで言う「1 番目」（日常生活での「2 番目」）に置き、2 月（変数 month の値が 2）に相当するものは、対応する英語の表現が格納されている変数 en_month_list の配列では、プログラミングで言う「2 番目」（日常生活での「3 番目」）に置き、3 月（変数 month の値が 3）に相当するものは、対応する英語の表現が格納されている変数 en_month_list の配列では、プログラミングで言う「3 番目」（日常生活での「4 番目」）に置き、というように、ひとつずらさなければならないことです。そのために、変数 en_month_list の配列の先頭には、無効、または無意味を意味する「nil」が置かれています。もちろん、この「nil」を置かなくてもよいように、変数 month の値を 1 減らして、それを配列中の位置を示す

番号として利用するという手もあります。

```
month = 1
en_month_list = ["January", "February", "March", "April", "May", "June", "July", "Aug
ust", "September", "October", "November", "December"]

print en_month_list[month - 1]
```

どちらが分かりやすいのかは人それぞれでしょう（私は前者のように、変数 month の値を変えないほうが好みです)。

実は、case - when - end 構文を用いた場合でも、以下のようにすると、変数名の「en_month」を何回も書かなくて済みます。

```
month = 1
en_month = nil

en_month = case month
  when 1 then "January"
  when 2 then "February"
  when 3 then "March"
  when 4 then "April"
  when 5 then "May"
  when 6 then "June"
  when 7 then "July"
  when 8 then "August"
  when 9 then "September"
  when 10 then "October"
  when 11 then "November"
  when 12 then "December"
end

print en_month
```

この場合は、上記のように、条件と処理の内容の間に「then」が必要になります（それまでは、あってもなくてもよいのでした)。ただ、この場合でも、条件で変数 month の値を明示しなければならないのが、配列と比較して若干面倒でしょう。でも、これもひとつの方法として覚えておくとよいでしょう。配列が使えないときには便利だと思います。

一方、月の英語の表現を数字に変換する（例えば、「January」に対して 1、「February」に対して 2、「March」に対して 3 を得ます）ときはどうでしょうか？　もし、先ほどと似たようなプログラムで書くのなら、平凡な case - when - end 構文を用いると、

```
en_month = "January"
month = nil

case en_month
when "January"
  month = 1
when "February"
  month = 2
when "March"
  month = 3
when "April"
  month = 4
when "May"
  month = 5
when "June"
  month = 6
when "July"
  month = 7
when "August"
  month = 8
```

```
when "September"
  month = 9
when "October"
  month = 10
when "November"
  month = 11
when "December"
  month = 12
end

print month
```

となります。

ところが、配列を使用しようとしても、配列中の位置を示す番号には整数しか指定できませんので、「"January"」という文字列を入れて「month_list["January"]」のようにするわけにはいきません。

しかし、配列に代わるものがあります。それは、**連想配列**（associative array）、**テーブル**（table）、**辞書**（dictionary）、**マップ**（map）などと呼ばれるものです。Ruby では連想配列を「**ハッシュ**」と呼んでいますが、実はプログラミングではあまり一般的な呼び方ではありません[50]。配列では角括弧「[」、「]」で囲まれた部分に整数を指定して取り出す値を決定していたのが、連想配列では整数以外のものを指定して取り出す値を決定することもできます。連想配列（ハッシュ）を使用したプログラム例を以下に示します。

```
en_month = "January"
month_table = {"January" => 1, "February" => 2, "March" => 3, "April" => 4, "May" => 5,
 "June" => 6, "July" => 7, "August" => 8, "September" => 9, "October" => 10, "November"
 => 11, "December" => 12}

print month_table[en_month]
```

連想配列（ハッシュ）が配列と異なるのは、連想配列（ハッシュ）を作るときには、角括弧「[」、「]」ではなく波括弧「{」、「}」でくくるということ、対応元と対応先の値を「=>」（等号「=」と大なり不等号「>」をつないでいますが、結果として矢印のような形をしています）で結ぶ（「=>」の左側が対応元の値、右側が対応先の値）ことです。

連想配列（ハッシュ）を作ったら、後は、配列と同様に、角括弧「[」、「]」に挟まれている部分に対応元の値を入れれば、その値に対応する値を取り出すことができます。例えば、上記のプログラムでは、「"January" => 1」によって「month_table["January"]」から 1 が得られ、「"February" => 2」によって「month_table["February"]」から 2 が得られ、「"March" => 3」によって「month_table["March"]」から 3 が得られます。

連想配列（ハッシュ）から取り出す値を決定するものには、配列の場合と同様に整数を使用することもできます。ということは、連想配列（ハッシュ）は配列よりも高機能だと言えます。それなら配列など不要に思われるかもしれませんが、実は、連想配列（ハッシュ）の内部構造は配列よりも複雑であり、そのために必要なメモリー量も多くなってしまいます。もし、巨大なファイルのように、格納するデータが非常に巨大であり、その中から一部を取り出すときに使用するのは連番だと予想される場合は、連想配列（ハッシュ）よりも配列を用いるのが普通

[50] なぜなら、ハッシュ（hash）とは普通、任意のデータに対して何らかの演算を施して、より小さいデータに集約することだからです。

です。

例えば、以下のプログラム（完全なものではありませんので、これをファイルに保存しても実行できません）は、ファイルの中で先頭（先頭のバイトが 0 バイト目であるとします）から 1,048,576 バイト（1 メガバイト）目にあるバイトを取り出すものです。

```
a = file_dat[1048576]
```

この場合に変数 file_dat に格納されるものとして、配列と連想配列（ハッシュ）のいずれかを選ぶのなら、配列が適切でしょう（ただし、Ruby では、実はファイルからデータを読み込む場合は文字列[51]にするのが普通です）。

かなり脱線してしましたが、本題に戻りましょう。プログラムを、配列を用いて書き換えると以下のようになります。ここでは、得点の倍率（変数 mul に格納）だけでなく、変更後の直前のフレームの状態（変数 next_s に格納）も配列にしています。プログラムが短くなったのがよく分かるでしょう。

プログラム例 6

```
pins = [7, 3, 8, 0, 10, 10, 6, 4, 7, 1]

$t = 0
$s = 0
$r = 0
$i = 1

def proc_a(p)
  mul = [1, 2, 2, 3]
  next_s = [2, 2, 3, 3]
  $t = $t + mul[$s] * p
  $s = next_s[$s]
  $i = 1
end

def proc_b(p)
  mul = [1, 2, 2, 3]
  $t = $t + mul[$s] * p
  $r = 10 - p
  $i = 2
end

def proc_c(p)
  mul = [1, 1, 2, 2]
  $t = $t + mul[$s] * p
  if p >= $r
    $s = 1
  else
    $s = 0
  end
  $i = 1
end

pins.each do |p|
  print "i: #{$i}  s: #{$s}  p: #{p}"

  if $i == 1
    if p == 10
```

[51] Ruby では、文字列は基本的にバイト列を意味します。ただし、Ruby バージョン 1.9 から、文字列は単なるバイト列ではなく、エンコーディングに関する情報を含むようになりました。

```
      proc_a(p)
    else
      proc_b(p)
    end
  else
    proc_c(p)
  end

  print " -> t: #{$t}¥n"
end

print "Total score: #{$t}¥n"
```

もっともっとも～っとプログラムを工夫してみよう

工夫はまだまだ続きます。例えば、今まで、ある変数の値を増やしたり減らしたりするには、

```
変数A = 変数A + 値B
```

などのように、いったん変数と別の値について何らかの演算を行い（上記の例における「変数A＋ 値 B」の部分）、その結果を改めて変数に代入するような文を書いていました。しかし、これをもっと短く、例えば、先ほどの例に対しては、

```
変数A += 値B
```

と書き換えることができます。

例えば、「a = a + 5」は実質的には「変数 a の値を 5 だけ増やす」という意味ですが、直訳はあくまで「変数 a の値に 5 を加算し、その結果を変数 a に代入する」です。しかし、「a = a + 5」は「a += 5」と書いても同じ結果が得られます。こちらの方が、「変数 a の値を 5 だけ増やす」という実質的な意味がよく現れているのではないでしょうか？ **実際のプログラミングでも、同じ変数名を２回書いてしまう「a = a + 5」のようなものよりも、1 回で済んでしまう「a += 5」が好まれる傾向にあります。なぜなら、後で変数の名前を変えたくなったときに、同じ変数名が２回出現している箇所があると、その２つとも間違いなく変更しなければならないのですが、うっかり１つしか変えなかったという間違いが少なくないからです。**

このように、ある変数に対して何らかの演算を行い、その結果を改めてその変数に代入することを Ruby では**自己代入**と言います。自己代入を用いるには、演算に使用し、結果の代入先となる変数に続いて、自己代入のための演算子（例 「+=」、「-=」）を書き、そして、その直後に演算に使用する値を書きます。自己代入のための演算子は原則として、演算のための演算子に等号「=」を付けたものになっています。例えば、足し算（加算）と代入を行う自己代入の演算子は「+=」ですが、これは、足し算（加算）の演算子「+」と代入の演算子「=」をつなげたものになっています。同様にして、「-=」は引き算（減算）の演算子「-」と代入の演算子「=」をつなげたもの、「*=」は掛け算（乗算）の演算子「*」と代入の演算子「=」をつなげたもの、「/=」は割り算（除算）の演算子「/」と代入の演算子「=」をつなげたものになっています。

このことを利用すると、先ほどのプログラム（プログラム例 6）における「$t = $t + mul[$s] * p」は「$t += mul[$s] * p」と書くことができます。

次の工夫に行きましょう。実は、配列の紹介における case - when - end 構文で少し取り上げたのですが、if - （elsif） - （else） - end 構文や case - when - end 構文をそのまま代入につなげ

ることができます。例えば、先ほどのプログラム（プログラム例 6）におけるメソッド「proc_c」の一部分である

```
if p >= $r
  $s = 1
else
  $s = 0
end
```

は、実質的には「変数$s には、変数 p の値が変数$r の値以上であれば 1、そうでなければ 0 を代入する」という意味を持つものですが、これを、以下のように書き換えることができます。

```
$s = if p >= $r then 1 else 0 end
```

ここでは、条件「p >= $r」の直後に「then」を置くことが必須となります。「then」を置かないと、

```
syntax error, unexpected tINTEGER, expecting keyword_then or ';' or '¥n'
```

（日本語訳　統語エラーです。Keyword_then または「;」または「¥n」が期待されているところで予期しない tINTEGER が出現しました。）

というエラーメッセージが表示されて、プログラムを正しく実行できません。

なお、

```
$s = if p >= $r then 1 else 0 end
```

という文は、さらに簡潔に

```
$s = (p >= $r) ? 1 : 0
```

と書くこともできます。ここでは演算子を構成する記号が 2 つ（疑問符「?」、コロン「:」）使用されていますが、疑問符「?」の直前には条件を置き、疑問符「?」とコロン「:」の間には条件を満たす場合の値や式を置き、コロン「:」の直後には条件を満たさない場合の値や式を置きます。疑問符「?」、コロン「:」という 2 つの記号でひとつの演算子をなしています。この演算子を**条件演算子**（conditional operator）と言い、3 つの値をとっていることから、条件演算子は**三項演算子**（ternary operator）の一種であると言えます。**通常は、場合によって代入する値を変えるようなプログラムを書くときには**、こちらのように、**条件演算子を用いた簡潔な表現の方が好まれています**。

ただし、条件と対応する値や式の組み合わせが 3 つ以上の場合に分かれる場合は、条件演算子を用いた表現よりも、if - （elsif） - （else） - end 構文や case - when - end 構文の方か好まれます。例えば、変数 b に、変数 a の値が正（0 を超える）なら文字列"positive"、負（0 未満）なら文字列"negative"、0 なら文字列"neutral"を代入するための文は、if - （elsif） - （else） - end 構文なら、

```
b = if a > 0 then "positive" elsif a < 0 then "negative" else "neutral" end
```

と書けますが、条件演算子（「?」、「:」）には一度にひとつの条件しか指定できませんので、条件演算子に対する値や式の中に、さらに条件演算子を使用しなければなりません。そのため、以下のように書かざるを得ません。

```
b = (a > 0) ? "positive" : ( (a < 0) ? "negative" : "neutral" )
```

しかし、これだと括弧の数が増えてしまいますので、文が読みづらくなってしまいます。数学では括弧の種類を変える（丸括弧「(」、「)」だけでなく、波括弧「{」、「}」や角括弧「[」、「]」も使う）こともできますが、Ruby も含めて多くのプログラミング言語では、括弧の種類によって意味が異なりますので、結局丸括弧「(」、「)」しか使用できません。

ここまでの工夫を使用した結果としてのプログラムを以下に示します。

プログラム例 7

```
pins = [7, 3, 8, 0, 10, 10, 6, 4, 7, 1]

$t = 0
$s = 0
$r = 0
$i = 1

def proc_a(p)
  mul = [1, 2, 2, 3]
  next_s = [2, 2, 3, 3]
  $t += mul[$s] * p
  $s = next_s[$s]
  $i = 1
end

def proc_b(p)
  mul = [1, 2, 2, 3]
  $t += mul[$s] * p
  $r = 10 - p
  $i = 2
end

def proc_c(p)
  mul = [1, 1, 2, 2]
  $t += mul[$s] * p
  $s = (p >= $r) ? 1 : 0
  $i = 1
end

pins.each do |p|
  print "i: #{$i}  s: #{$s}  p: #{p}"

  if $i == 1
    if p == 10
      proc_a(p)
    else
      proc_b(p)
    end
  else
    proc_c(p)
  end

  print " -> t: #{$t}¥n"
end

print "Total score: #{$t}¥n"
```

もっともっとも～っとも～～っとプログラムを工夫してみよう（いい加減にしなさい！）

もうこれでいいではないかという気がしなくもないですが、まだまだ工夫は続きます。

実は、ひとつ気になることとして、処理 A（メソッド「proc_a」）、処理 B（メソッド「proc_b」）、処理 C（メソッド「proc_c」）はいずれも、主に得点計算と直前フレームの状態の更新を担っており、その点では似ているというのがあります。何とかしてひとつに統合できないでしょうか？

面倒に思われるかもしれませんが、**わざわざ似たような処理をひとつに統合するのは、後にプログラムを変更するときに、同じような修正・変更を何回も繰り返さなくてもよいようにす**

るためです。単に新しい部分を複数作成するだけなら、テキストエディターなどのコピー機能（Windows で言うところの「コピー & ペースト」）を利用すればよいのですが、修正・変更ではそうは行きません。修正・変更する箇所が複数あると、それだけ修正・変更し忘れ、結果としてプログラムが正しく動かなくなってしまい、その原因究明に時間がかかってしまう可能性が高くなります。**後で楽をするためにも、プログラムは（極端に読みづらくなる場合は別として）できるだけ小さく保つのです。**

そこで、ここではメソッド（処理のまとまり）を再編成し、今まで、フレーム中 1 投目でストライクを出した場合（メソッド「proc_a」）、フレーム中 1 投目でストライクを出さなかった場合（メソッド「proc_b」）、フレーム中 2 投目の場合（メソッド「proc_b」）で分けていたのを、フレーム中 1 投目の場合（メソッド「proc_first」）とフレーム中 2 投目の場合（メソッド「proc_second」）で分けなおしてみました（つまり、メソッド「proc_a」と「proc_b」を統合したものになっています）。その結果のプログラムが以下のものです。

プログラム例 8

```
pins = [7, 3, 8, 0, 10, 10, 6, 4, 7, 1]

$t = 0
$s = 0
$r = 0
$i = 1

def proc_first(p)
  mul = [1, 2, 2, 3]
  next_s = [2, 2, 3, 3]
  $t += mul[$s] * p
  $s = next_s[$s] if p >= 10
  $r = 10 - p if p < 10
  $i = (p >= 10) ? 1 : 2 end
end

def proc_second(p)
  mul = [1, 1, 2, 2]
  $t += mul[$s] * p
  $s = (p >= $r) ? 1 : 0
  $i = 1
end

pins.each do |p|
  print "i: #{$i}  s: #{$s}  p: #{p}"

  if $i == 1
    proc_first(p)
  else
    proc_second(p)
  end

  print " -> t: #{$t}¥n"
end

print "Total score: #{$t}¥n"
```

ここでもまた新しい文の書き方が登場しています。それは、例えば、

```
$r = 10 - p if p < 10
```

です。式「$r = 10 - p」の直後に「if p < 10」という条件を付けています。これは、条件を満た

す場合のみ、式の効果が発動するというものです。この例の場合は、変数 p の値が 10 未満（p < 10）の場合には、変数$r に 10 から変数 p の値を引いたものを代入する（$r = 10 - p if p < 10）というもので、変数 p の値が 10 以上の場合には何もしません。

今までなら、

```
if p < 10
  $r = 10 - p
end
```

と書いていたところですが、「$r = 10 - p if p < 10」の方が若干簡潔でしょう。もっとも、元の方法であっても、

```
if p < 10 then $r = 10 - p end
```

のように、1 行で書くこともできます。「$r = 10 - p if p < 10」と比べてどちらがよいかは好みの問題かもしれませんが、Ruby のプログラマーはどちらかと言えば「$r = 10 - p if p < 10」の方をとるようです（たぶん…）。

似ている箇所の統合を推し進めるのなら、フレーム中 1 投目の場合（メソッド「proc_first」）とフレーム中 2 投目の場合（メソッド「proc_second」）だって、前者（メソッド「proc_first」）のみ変数$r に対して代入を行い、後者（メソッド「proc_second」）のみ変数$r から値を取り出しているという違いこそあれ、それ以外の、得点計算、直前フレームの状態の更新についてはやはり似ていますので、2 つのメソッド「proc_first」と「proc_second」を統合してしまいましょう。その結果を以下に示します。

プログラム例 9

```
pins = [7, 3, 8, 0, 10, 10, 6, 4, 7, 1]

$t = 0
$s = 0
$r = 0
$i = 1

def proc(p)
  mul = [nil, [1, 2, 2, 3], [1, 1, 2, 2]]
  next_s_if_strike = [2, 2, 3, 3]
  $t += mul[$i][$s] * p
  if $i == 1
    if p >= 10
      $s = next_s_if_strike[$s]
    else
      $r = 10 - p
    end
  else
    $s = (p >= $r) ? 1 : 0
  end
  $i = ( (p >= 10) || ($i == 2) ) ? 1 : 2
end

pins.each do |p|
  print "i: #{$i}  s: #{$s}  p: #{p}"
  proc(p)
  print " -> t: #{$t}¥n"
end

print "Total score: #{$t}¥n"
```

フレーム中の投球の番号やストライクを出したかどうかで大きく分けていないのは、初期のプ

ログラムであるプログラム例 1 と同じですが、プログラム例 1 と比べてずいぶんと小さくなっていることが分かります。

ここで何か新しいことをやったかと言えば、変数 mul で配列の中に配列を入れたことくらいでしょうか。変数 mul に見られるように、配列の中に配列を入れる（入れ子にする）こともできます。このように、配列が入れ子になっているようなものを**多次元配列**（multi-dimensional array）と言います。

今、変数 mul は以下のようになっています。

```
mul = [nil, [1, 2, 2, 3], [1, 1, 2, 2]]
```

まず、第 1 段階として、最も外側にある配列を分解してみると、以下のようになります。

```
mul[0] = nil
mul[1] = [1, 2, 2, 3]
mul[2] = [1, 1, 2, 2]
```

このように､最も外側にある配列を分解してみれば､既に触れたとおりの配列が現れています。mul[1]、mul[2]は結局のところただの配列ですから、さらに角括弧「[」、「]」の間に番号を入れ、それぞれの直後に付ければ、配列の中から一つの値を取り出すことができます。例えば、「mul[1][0]」と書くと、mul からプログラミングで言う「1 番目」のものを取り出して配列[1, 2, 2, 3]を得て、さらにその配列[1, 2, 2, 3]からプログラミングで言う「0 番目」のものを取り出して 1 を得ます。一方、「mul[2][3]」と書くと、mul からプログラミングで言う「2 番目」のものを取り出して配列[1, 1, 2, 2]を得て、さらにその配列[1, 1, 2, 2]からプログラミングで言う「3 番目」のものを取り出して 2 を得ます。

というわけで、変数 mul に「[番号]」のようなものを 2 つ指定することで 1 つの値を取り出すのですが、2 つの「[番号]」における番号にはそれぞれ何を指定すればよいかというと、ここでは、1 番目の番号にはフレーム中の投球の番号（1 または 2）、2 番目の番号には直前フレームの状態を表す番号（0 から 3 までの整数）を指定すればよいように作られています（変数 mul の中身が定義されています)。つまり、

```
mul[フレーム中の投球の番号][直前フレームの状態を表す番号]
```

のように書けばよいというわけです。もちろん、これは模式的なものであって、これをそのままコピーしたのではプログラムは動きませんが、フレーム中の投球の番号は変数$i に、直前フレームの状態を表す番号は変数$s に代入されていますから、プログラム中で

```
mul[$i][$s]
```

と書けば、フレーム中の投球の番号と直前フレームの状態を表す番号に応じて、得点の倍率を取り出すことができるというわけです。

ちなみに、mul[0]が nil（無効、無意味な値）になっているのは、フレーム中の投球の番号が 1 で始まっており、この値をそのまま変数 mul に格納されている配列中の位置を示す番号として使用すればよいようにするためです（そう、**Ruby も含めて多くのプログラミング言語では「1 番目」は先頭ではないのでした**)。

ひとつ、**今回のプログラムにおける場合分け（条件分岐）で注意したいことは、倒したピンの数に関する条件よりもフレーム中の投球の番号に関する条件を先に使用することです**。なぜなら、ある投球で倒したピンの数が（10 本中）10 本であっても、それがフレーム中 1 投目ならストライク、フレーム中 2 投目ならスペアーと、異なる結果が得られるからです。従って、こ

こでの場合分け（条件分岐）の基本構造は

```
if p >= 10
  if i == 1
     (何らかの処理)
  else
     (何らかの処理)
  end
else
   (何らかの処理)
end
```

ではなく、

```
if i == 1
  if p >= 10
     (何らかの処理)
  else
     (何らかの処理)
  end
else
   (何らかの処理)
end
```

でなければなりません。

別の工夫として、些細なことですが、

```
$t = 0
$s = 0
$r = 0
```

は同じ値を代入していますが、これらも以下のように統合できます。

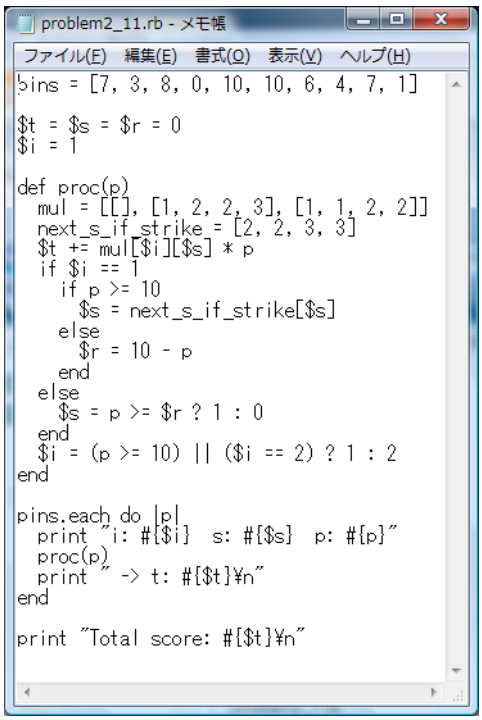
```
$t = $s = $r = 0
```

ここまでの工夫を利用した結果のプログラムを以下に示します。

プログラム例 10

```
pins = [7, 3, 8, 0, 10, 10, 6, 4, 7, 1]

$t = $s = $r = 0
$i = 1

def proc(p)
  mul = [nil, [1, 2, 2, 3], [1, 1, 2, 2]]
  next_s_if_strike = [2, 2, 3, 3]
  $t += mul[$i][$s] * p
  if $i == 1
    if p >= 10
      $s = next_s_if_strike[$s]
    else
      $r = 10 - p
    end
  else
    $s = (p >= $r) ? 1 : 0
  end
  $i = ( (p >= 10) || ($i == 2) ) ? 1 : 2
end

pins.each do |p|
  print "i: #{$i}  s: #{$s}  p: #{p}"
  proc(p)
  print " -> t: #{$t}¥n"
end

print "Total score: #{$t}¥n"
```

このプログラムは全部で 28 行あります。今日なら余裕で 1 画面に収まるでしょう。



(fig_Problem2_11.tif)

このプログラムを初期のプログラムと比較してみましょう。

プログラム例 1

```
pins = [7, 3, 8, 0, 10, 10, 6, 4, 7, 1]

t = 0
s = 0
r = 0
i = 1

pins.each do |p|
  print "i: #{i}  s: #{s}  p: #{p}"

  if i == 1
    if p == 10
      if s == 0
        t = t + p
        s = 2
      elsif s == 1
        t = t + 2 * p
        s = 2
      elsif s == 2
        t = t + 2 * p
        s = 3
      else
        t = t + 3 * p
        s = 3
      end
      i = 1
    else
      if s == 0
        t = t + p
      elsif s == 1
```

```
      t = t + 2 * p
    elsif s == 2
      t = t + 2 * p
    else
      t = t + 3 * p
    end
    r = 10 - p
    i = 2
    end
  else
    if s == 0
      t = t + p
    elsif s == 1
      t = t + p
    elsif s == 2
      t = t + 2 * p
    else
      t = t + 2 * p
    end
    if p >= r
      s = 1
    else
      s = 0
    end
    i = 1
  end

  print " -> t: #{t}¥n"
end

print "Total score: #{t}¥n"
```

このプログラムは全部で 61 行あります。何と、さまざまな工夫の末に、プログラムは半分以下にまで小さくなったのです！　これだけプログラムが小さくなっても、実行結果に変化はありません。このように、プログラムの機能（実行結果）を保ちながら、プログラムの内容を変えて構造を整理することを**リファクタリング**（code refactoring）と言います。

このように**プログラミングでは、一つの機能を実現させるための書き方は非常にさまざまありますが、その書き方によって、プログラムは長くも短くもなるのです**。最後に出来上がったプログラム例 10 は初期のものであるプログラム例 1 と比べて短くなりました。普通は、短いプログラムのほうが読みやすいのです。というのも、初期のプログラムでも 61 行しかない今回の問題ではたいしたものではないのですが、実際のプログラミングでは数百行でもまだぬるく、数千行、数万行、大規模なソフトウェアでは数百万行に到達することも珍しくありません。そのような状況では、長いプログラムは急激に読みづらくなるものですから、**プログラムを短く書くことは結構重要です**。

誤解されると困りますので、念のため述べておきますと、いきなりプログラム例 10 のような短い（スマートな？）プログラムを書きなさいと言うわけではありません。プログラム例 10 は工夫に工夫を重ねて生み出されたものであって、最初のひと筆で書けるものではありません。私でも、最初に書くのはせいぜいプログラム例 7 のようなものだと思います。いきなり整理されたプログラムを書こうと思わずに、とりあえずはプログラムを書いて、それから工夫を加えてより読みやすく分かりやすいプログラムに編集していく（リファクタリングしていく）という手もあります。それは、市販されている書籍（特にプログラミングの本というわけではなく、

岩波新書やちくま新書といった普通の書籍を想像していただければ結構です）の本文がいきなりひと筆で書かれるものではなく、最初は下書きの段階であり、工夫、検討・吟味や校正を重ねて、市販される書籍に掲載されるような最終版に近付けていくのと同じようなものです。

しかしながら、さまざまな工夫を利用していく中で、確かにプログラムは短く簡潔になったけど、逆に読みづらく分かりづらくなった、というようなものもあるかもしれません。このあたりは人によって異なるでしょう。プログラムが短ければ無条件によいというわけではありません。おそらく、Ruby の言語規則において重箱の隅をつつくようにすれば、さらに短いプログラムが書けるかもしれません[52]、そのようなことをしてもプログラミングがやりやすくなるどころか逆に難しくなる可能性が高いですので、ここでは行いません。ただ、簡単なプログラムを作るうえではやりやすいと考えていた方法が大規模なプログラムを作るうえでは通用しないということも少なくありませんので、どの方法がよいのかという見極めは決して簡単ではありません[53]。

Ruby には他にもさまざまな機能があります。その機能は非常に膨大で、覚えるのは大変かもしれませんし、なくてもプログラミングができるかもしれません。しかし、そのような機能でも利用すれば結構便利で作業がはかどるようなものも多数あります。そのような機能を使いこなせるかどうかが、実はプログラム完成品の質にも大きく影響することが少なくありません。ぜひ、さまざまな文書を読んでは、さまざまな機能を利用してみてください。それに、何もかも覚える必要はありません。取扱説明書などの文書を参照するのはいつでもできるのですから、作業に支障が出ない程度にまで覚えればよいのです[54]。

プログラムを説明しよう

ところで、プログラムを分かりやすく書く方法として重要な方法に、プログラムの説明書きを付けるというものがあります。

商品を購入すると、ほぼ全てと言ってよいほど取扱説明書が付いてきます。取扱説明書はその名の通り、商品の使い方を説明するものです。取扱説明書がなくても、商品の見た目などから使い方をある程度推測することもでき、それはそれで重要なのですが、それだけでは分からないこともあり、そのようなときには取扱説明書が役に立ちます。

例えば、取扱説明書なしで空調機（エアコン）の清掃を行うのはあまり推奨されません。なぜなら、空調機は様々な部品で構成されており、取り外し方や清掃方法は部品によって異なります。正しくない取り外し方を強引に適用すれば部品が破損するかもしれません。また、部品によっては水洗いしてもよいもの（または、水洗いせざるを得ないもの）や水洗いをしてはならないものがあり、部品の性質によって清掃の方法を変えないと、やはり部品が破損するかも

[52] 何としてでも短いプログラムを作るという遊びの例に Rubyist Magazine（制作：日本 Ruby の会）（http://jp.rubyist.net/magazine/）の「るびまゴルフ」があります。もちろん、これはお遊びなので、実用的なものではありませんが、参考にはなるかもしれません。

[53] ただ、私の経験によれば、プログラム例 10 くらいのものはよく出てくると思われますので、慣れておくとよいでしょう。

[54] その意味でも、コンピューターも取扱説明書も使用できない状態でプログラミングの筆記試験を行うのは反対なのです。

しれません[55]。

プログラミングでも同様です。もちろん、わざわざ取扱説明書などを読まなくても、プログラム本体を読めば意味が分かるようにするのは重要なことです。しかし、それにも限界がありますので、説明書きを付けることでプログラムを分かりやすくするのです。

特に、プログラミングは複数の人が共同で行うこともあるもので、その場合は特に、チームの他のメンバーに適切に意思伝達を行う必要があります。もし適切な意思伝達がなければ、チームのメンバーが別のメンバーが書いたプログラムに修正を加えようとしても、そのプログラムを読めないので作業が進まなかったり、他のメンバーが書いたプログラムを正しく理解しないまま強引にプログラムの修正作業を進めて、結果としてプログラムを破壊してしまい、作業がさらに困難になったりすることが少なくありません。

自分か書いたプログラムなら、しばらくの間は特に説明書きを付けなくても、自分の頭で覚えているためにどのような内容なのかが分かりますが、長い間プログラムに手をつけないでいると、自分が書いたもののはずなのに読めなくなってしまうこともあります。「今日の自分は明日の他人」という教訓があるくらいですから、油断は禁物です[56]。

そのようなわけで、直接はプログラムの機能向上とは関係がなさそうに見える、プログラムの説明を行うのですが、プログラムの説明を行う方法にはいくつかあります。一つは、プログラムを作る上で使用した考え方、プログラムにおける部分の大まかな意味、プログラム全体または部分の使い方などを、ワードプロセッサーなどを使用して（プログラム本体とは別の）文書にまとめるというものです。この方法を用いれば、文章だけでなく、表や図を挿入することもできますので、いくらでも分かりやすい説明を行う可能性はありますが、その作業が実は結構面倒なのが難点です。

そこで、たいてい（ほぼ全てと言ってもよいでしょう）のプログラミング言語は、プログラム中に説明書きを加えるのを「許して」います。プログラム中に加えられる説明書きは**注釈**（comment）と言われ、プログラムの実行には一切影響を与えません。

プログラムの実行には一切影響を与えないようなものを書いても意味がないと思われるかもしれませんが、決してそうではありません。もし意味がないのなら、プログラミング言語が注釈を「許す」ような仕組みを備えること自体意味がないはずだからです。そうではなく、プログラムの実行に一切影響を与えないからこそ、注釈には好きな内容を入れることができるのです。つまり、プログラマーに都合のよいことを書けるというわけです。

ただし、注釈はプログラム中に入れるものであるために、あまり注釈が長すぎると、かえってプログラムが読みづらくなってしまうかもしれません。そこで、注釈は（ワードプロセッサーなどで作成する文書よりも）短く簡潔に書くのが普通です。

それでは、どのように注釈をつければよいのかを見てみましょう。ここで、例として以下の

---

[55] 例えば、私の家にある空調機について述べると、集塵のための部品は水洗いすることになっていますが、脱臭のための部品は水洗いをせずに埃をたたいて落とすにとどめ、代わりに脱臭機能を復活させるために紫外線（日光など）を当てることになっています。

[56] もちろん、一回書いたらあとは二度と手をつけないような使い捨て型のプログラムを作るのなら、わざわざ説明書きをする必要はないでしょう。

ようなRubyで書かれたプログラムがあるとします。

```
a = 4
b = 3
c = 5

if a >= b
  if a >= c
    print a
  else
    print c
else
  if b >= c
    print b
  else
    print c
  end
end
```

このプログラムは最初の方で変数aに4、変数bに3、変数cに5を代入していますが、これは大した問題ではありません。興味があれば、これらの変数に代入する値を変えても結構です（ただし、数（整数や実数）にしてください）。

さて、これは何をするプログラムか分かりますか？　このプログラムはよくプログラミングの入門で取り上げられるものですが、このプログラムを読んだことがない限りは、何のプログラムなのかはすぐに分からないかもしれません。

結論を申し上げると、このプログラムは、a、b、cという3つの変数に格納されている数のうち最大のものを表示するものです。そこで、以下のように注釈を入れることができます。

```
a = 4
b = 3
c = 5

# a、b、cのうち最大の数を表示する。
if a >= b
  if a >= c
    print a
  else
    print c
else
  if b >= c
    print b
  else
    print c
  end
end
```

ここでは「# a、b、cのうち最大の数を表示する。」が注釈です。このように、Rubyでは、「#」（ただし、文字列に含まれているものを除きます）および行中でその「#」以降の文字列は注釈とみなされ、プログラムの実行には関与しません。以下のように、行の中で途中まではプログラムとして実行の対象になる文を書き、後は注釈で埋めることもできます（以下のものでは、「# とりあえず、使用する変数の宣言だけはしておく。」が注釈です）。

```
result = nil  # とりあえず、使用する変数の宣言だけはしておく。
src = [1, 2, 3]
```

ただし、先ほどa、b、cという3つの変数に格納されている数のうち最大のものを表示するプログラムに付けた注釈「# a、b、cのうち最大の数を表示する。」は、実はあまりよいものではありません。

例えば、以下のようにするのはどうでしょうか？

```
def max(a, b, c)
  if a >= b
    if a >= c
       return a
     else
       return c
  else
    if b >= c
      return b
    else
      return c
    end
  end
end

d = 4
e = 3
f = 5
print max(d, e, f)
```

ちなみに、「return」は、そこでメソッドを終了し、「return」の直後にある値をメソッドの結果として外部に返すことを示します。

ここでは、3 つの数の中で最大のものを表示する部分をメソッド max として定義しています（「def max(a, b, c)」）。つまり、その部分に「max」という名前を付けたのです。このような名前が付いていたら、誰だって最大値を求めるものだと思うでしょう。もし、「max」という名前が付いていながら、実は最小値を求めるものだったり、平方根を求めるものだったり、適当なファイルを作成するものだったりしたら詐欺ではないでしょうか？　つまり、名前を付けることで何をするのかが分かる処理に付いては、名前を付ければよいのであって、注釈を付ける必要はありません。

このように名前を付けるのが注釈よりも優位に立つ点は、一つは、名前はプログラムの実行にある程度関与しますので、何か問題が発生したときに、その名前を表示してくれることがあるということです。

そして、もう一つは、余計な注釈がなければ管理が面倒にならなくて済むという点です。注釈はプログラムの実行には影響を与えませんので、その管理は原則としてプログラマーが行わなければなりません。もし、もととなるプログラムを変更したら、その内容に合わせて注釈を更新しなければなりません。もし注釈を更新せずに放置したら、その注釈は嘘ということになり、そのプログラムを見る人に余計な混乱をもたらす可能性が高くなります。例えば、以下のようなプログラムが該当します。

```
a = 4
b = 3
c = 5

# a、b、cのうち最大の数を表示する。
if a <= b
  if a <= c
    print a
  else
    print c
else
  if b <= c
    print b
```

```
  else
    print c
  end
end
```

このプログラムを読むときには、真っ先に注釈「」に目が行くと思います。そのため、このプログラムは変数 a、b、c に格納されている値のうち最大のもの、つまり最大値を表示するものであると思われることでしょう。しかし、このプログラムを実行してみても分かることですが、実は、このプログラムは変数 a、b、c に格納されている値のうち最小のもの、つまり最小値を表示するものなのです！　つまり、プログラムの内容と注釈の内容が一致しないのです。

この場合は 2 通りの解釈が考えられます。一つは、(注釈を除く)プログラムの内容が正しく、注釈の内容が誤っていること（つまり、もともと最小値を表示することを意図していたということ)。もう一つは、注釈の内容が正しく、プログラムの内容が誤っていること（つまり、もともと最大値を表示することを意図していたということ)。しかし、この 2 通りの解釈のうちどちらが正しいのかは、そのプログラムを書いた人に聞いたり、プログラムの内容を調べ上げたりしなければ分からないのですから、結構面倒です。

そのようなわけで、注釈はプログラムの内容に沿ったものでなければなりませんが、そのことを守れるようにするためにも、注釈の書き過ぎには注意しなければなりません。

これと似たような問題に、文を直訳したものを注釈として入れてしまうというものがあります。例えば、以下のように注釈を付けてしまうことを言います。

```
a = 4  # 変数aに値4を代入する。
b = 3  # 変数bに値3を代入する。
c = 5  # 変数cに値5を代入する。

if a <= b  # もし、変数aの値が変数bの値以下なら
  if a <= c  # もし、変数aの値が変数cの値以下なら
    print a  # 変数aの値を表示する。
  else   # そうでなければ
    print c  # 変数cの値を表示する。
else  # そうでなければ
  if b <= c  # もし、変数bの値が変数cの値以下なら
    print b  # 変数bの値を表示する。
  else   # そうでなければ
    print c  # 変数cの値を表示する。
  end  # 終わり
end  # 終わり
```

このような注釈を入れると、一見してプログラムが読みやすく思われるかもしれません。特に初心者にとってはそのようになりますので、よくプログラミングの入門書には採用されています。プログラミングの入門書では、文の直訳になっている注釈をよく見ることができます。しかも、何も知らずに誤って注釈までプログラムに含めても、そのことはプログラムの実行結果には影響を及ぼしませんので、初心者には都合がよいことでしょう。

プログラミングの入門書において、特定のプログラミング言語の規則を説明する段階なら、注釈を利用した説明も方法の一つとして数えられることでしょう。しかし、実際のプログラミングでは、このような注釈は単に煩わしいだけです。プログラムを変更することは普通に起こりますが、そのような場合は、注釈もプログラムの内容に沿って変更しなければならないため、面倒なことになります。

よくよく考えてみれば、プログラミング言語の規則を知らないのにプログラミングを行うことはあり得ないのです。それは、例えば、フランス語の文法、語法や意味を全く知らないのにフランス語で書かれた文学作品を研究することが無謀である（なぜなら、わざわざ他の言語に翻訳されなければその文学作品を読めないのであれば、非常に面倒だからです）のと同じようなものでしょう。そのような理由からも、プログラムを見れば分かることを注釈には書かないのです。

注釈の使い方として意味があるのは、主に以下のような場合です。一つは、メソッドの使い方や注意事項を説明する場合です。例えば、以下のようなプログラムが該当します。

```
def sin(x)
  return x - (x ** 3.0) / 6.0 + (x ** 5.0) / 120.0 - (x ** 7.0) / 5040.0
end

print sin(1.0)
```

このプログラムは、sin 1（角度の単位はラジアン）を計算しようとするものです[57]。ちなみに、演算子「**」はべき乗を表し、「a ** b」はa^bを意味します（一般にプログラム中では上付き文字を使用することができません[58]し、Ruby では「a ^ b」と書いてべき乗を表すこともできません）。また、この関数は整数で計算されると（例えば、「return x - (x ** 3) / 6 + (x ** 5) / 120 - (x ** 7) / 5040」と書いてしまうと、引数 x に整数が代入された場合に、最終結果だけでなく途中経過まで全て整数で計算されてしまいます）正しい結果が得られません（小数点以下の桁が捨てられるからです）ので、わざわざ数を書く際に「3.0」というように小数点を付けることで実数と見なしてもらうようにしています。

これは名前を見れば三角関数の一種である正弦関数$\sin x$の計算を行うように見えます。しかし、その内容は、実は、正弦関数$\sin x$を、$x = 0$を中心にテイラー展開したうちの 7 次の項までしか計算しないのです。

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \cdots$$

$$\approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}$$

テイラー展開は関数の数値計算でよく用いられるのですが、どこまで計算するか（どこで打ち切るか）、関数にどのような値を代入するかによって、誤差（テイラー展開する前の関数の値との差）が変化します。今回の場合は、正弦関数$\sin x$を、$x = 0$を中心にテイラー展開していますので、関数（のx）に代入する値は 0 に近いほど、関数の値の誤差は小さく見積もれる（つまり、関数の値に対する信頼性が高い）[59]傾向にあります。

そこで、このメソッド「sin(x)」の使い方として、

```
# このメソッドは、x = 0を中心として7次までテイラー展開した結果を使用して計算します。
```

[57] 正弦関数の計算は Ruby で最初から可能であり、「Math.sin(1.0)」のようにすれば使用可能ですので、実際には、わざわざ正弦関数を計算するメソッドを定義する必要はありません。説明の都合で取り上げただけであることを予めご了承ください。

[58] 数式処理ソフトウェアの一つである Mathematica では、上付き文字などを使用して本格的に数式を書いた上で、その数式について様々な計算をさせることができますので、それは直感的で便利なのですが、そのようなことができるものはソフトウェア一般、またはプログラミング一般としては稀です。

[59] 誤差の正確な値を知ることはできません。

```
# 引数
#   x  角度（ラジアン）
```

というのを注釈として入れるのがよいでしょうし、また、使用上の注意事項として、

```
# 注意  引数xに指定する値はできるだけ絶対値の小さなものにしてください。
```

というのを注釈として入れるのもよいでしょう。つまり、以下のようにプログラムに注釈を加えることができます[60]。

```
# このメソッドは、x = 0を中心として7次までテイラー展開した結果を使用して計算します。
# 引数
#   x  角度（ラジアン）
# 注意  引数xに指定する値はできるだけ絶対値の小さなものにしてください。
def sin(x)
  return x - (x ** 3.0) / 6.0 + (x ** 5.0) / 120.0 - (x ** 7.0) / 5040.0
end

print sin(1.0)
```

このようにすれば、このメソッド「sin(x)」の使い方や性質に関する判断を誤る可能性が低くなることでしょう。

注釈の主な使い方のもう一つは、プログラミングのメモをとるというものです。そのメモの内容は、プログラムを作る上で使用した考え方、プログラム中の文や処理のまとまりの意訳（直訳ではありません）などです。

例えば、本書では今回の期末試験で取り上げられた、ボーリングの得点計算を行うプログラムを、Ruby で書き換えることを行いましたが、そのプログラム例 10 ではメソッド「proc」を以下のように定義しました。

```
def proc(p)
  mul = [nil, [1, 2, 2, 3], [1, 1, 2, 2]]
  next_s_if_strike = [2, 2, 3, 3]
  $t += mul[$i][$s] * p
  if $i == 1
    if p >= 10
      $s = next_s_if_strike[$s]
    else
      $r = 10 - p
    end
  else
    $s = (p >= $r) ? 1 : 0
  end
  $i = ( (p >= 10) || ($i == 2) ) ? 1 : 2
end
```

この中で、ストライクが出たりスペアーが出たり、そのいずれも出なかったりしたことで場合分け（条件分岐）を行う個所があります。

```
  if $i == 1
    if p >= 10
      $s = next_s_if_strike[$s]
    else
      $r = 10 - p
    end
  else
    $s = (p >= $r) ? 1 : 0
  end
```

ここで、「p >= 10」はピンが 10 本全部倒れたことを意味しますので、そのことを明記しておく

[60] ただ、このような用法なら、RDoc などの自動文書作成システムを利用するという手もありますが、ここでは割愛します。

とよいでしょう。ここでは、このプログラムを読む人から「p が 10 を超えた場合はどうなるのか」というツッコミを受けても大丈夫なように（？）、倒れたピンの数を示す p が 10 を超えるというあり得ない場合についても言及しています。

```
    if p >= 10  # ピンが全部倒れた場合（10本を超えることはないが、それを条件に含めても問題ない）
```

ところで、ピンが 10 本全部倒れたというだけでは、それがストライクを意味するのか、それともスペアーを意味するのか判定できません。これらを判定するには、その投球がフレーム中何番目なのかという情報が必要です。フレーム中 1 番目の投球（$i == 1）でピンを 10 本倒せばストライクですし、そうではなく、フレーム中 2 番目の投球（$i == 2）でピンを 10 本倒せばスペアーです。このことは見落としやすいかもしれませんので、プログラム中にそのことを注釈として明記してもよいでしょう。

```
# 単にピンが10本倒れた（p == 10）というだけではストライクかスペアーのどちらなのか分からない。
# それがフレーム中1投目（$i == 1）ならストライクであり、
# フレーム中2投目（$s == 2）ならスペアーであることに注意。
if $i == 1  # フレーム中1投目の場合
  if p >= 10  # ピンが全部倒れた場合（10本を超えることはないが、それを条件に含めても問題ない）
    $s = next_s_if_strike[$s]  # ストライクの場合における、直前フレームの状態の更新
  else # 倒れずに残ったピンがある場合
    $r = 10 - p  # 残ったピンの数を計算
  end
else  # フレーム中2投目の場合
  $s = (p >= $r) ? 1 : 0  # 倒したピンの数がフレーム中1投目で残ったピンの数に一致すればスペアー
end
```

ようやく終わり！

いかがだったでしょうか？　随分と長い説明になってしまいましたが、プログラミングの基本的な考え方がいくらかでも身につけることができたのなら、私としては幸いです。だけど、ここまで説明が長くなるとは思わなかった…（汗）。