BYOB* による計算機数学入門†

神戸大学人間発達環境学研究科
高橋　真

この資料はビジュアルプログラミング環境の BYOB(Build Your Own Blocks) を利用して計算機数学の講義で扱う分野（計算論，ホーア論理，モデル検査）の初歩を分かりやすく解説することを目的としています.

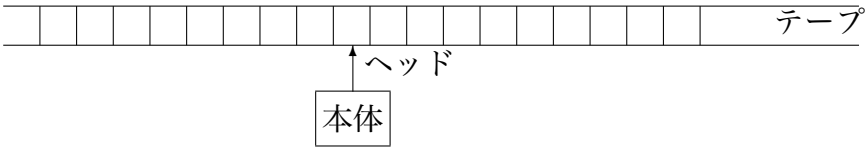
*Build Your Own Blocks http://snap.berkeley.edu/

†この講義資料は JSPS 科研費 23650507 及び 26560089 の助成を受けた研究の過程で得られたものです.

1　チューリング機械

1.1　チューリング機械とは

チューリング機械は Alan Turing[1] が 1936 年に発表した彼の論文のなかで定義した抽象的な機械で，“計算”のもつ機械的な性質に着目して考えだされた．チューリング機械は現在のコンピュータの思想的な源の一つである．



1. チューリング機械 M は，本体とテープ及びテープから記号を読んだり書いたりするヘッドからなっている．

2. テープは両側に無限に延びていて，ます目に区切られている．また各マス目にはあらかじめ定められた有限個の入力アルファベット記号（空白記号を含む）を一つだけ書くことができる．

3. 空白記号以外の記号が書かれているマス目は有限個である．

4. 本体は各瞬間に有限種類の状態のうちのいずれか 1 つの状態をとる．

5. M は各瞬間にヘッドの下にあるテープのます目に書いてある記号を読みとって本体へ送り，本体の現在の状態と送られてきたテープ記号に応じて，本体の状態を他に変えて（同じ状態のままでもよい）ヘッドが見ているマス目の記号を書き換え（書き込む記号は入力アルファベット記号の中から選ぶ．同じ記号で書き換えてもよい），ヘッドを 1 ます分左に動かすか，右に動かす．

5 で述べたような，現在の状態 s とテープ記号 a_i に対し，次の瞬間の状態 t に変え，テープ記号を a_j に書き換え，ヘッドの移動 m（m は L(左) または R(右)）を行う規則を $(s, a_i; a_k, m, t)$ と表現し，このチューリング機械の基本操作と呼ぶ．実行したい計算により，基本操作を定める．記号や状態の集合はいずれも有限であったから，基本操作の取り方も高々有限個である．

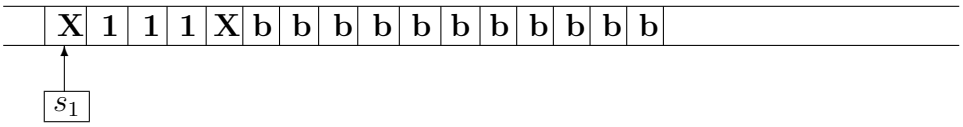
本体の動作は初期状態と呼ばれる状態からヘッドをテープ上の指定されたます目に置いて開始し，停止状態と呼ばれる状態になったとき停止する．停止状態は複数あってもよい．

[1] http://www.turing.org.uk/turing/

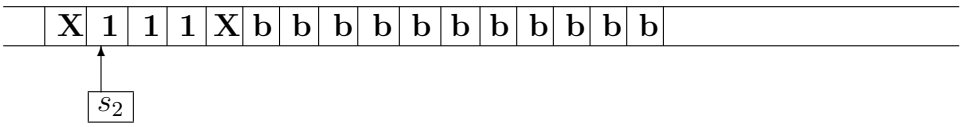
例 **1.1** テープ記号が $\mathbf{1}, \mathbf{X}, \mathbf{b}$ ($\mathbf{b}$ は空白記号) で s_1, s_2, s_3, h を状態としてもち，基本操作を次のように定めたチューリング機械を考える．但し，s_1 が初期状態で h を停止状態とする．

$(s_1, \mathbf{1}; \mathbf{1}, R, s_1), (s_1, \mathbf{X}; \mathbf{X}, R, s_2), (s_1, \mathbf{b}; \mathbf{b}, R, s_1),$
$(s_2, \mathbf{1}; \mathbf{b}, R, s_2), (s_2, \mathbf{X}; \mathbf{b}, L, s_3), (s_2, \mathbf{b}; \mathbf{b}, R, s_2),$
$(s_3, \mathbf{1}; \mathbf{1}, L, s_3), (s_3, \mathbf{X}; \mathbf{X}, R, h), (s_3, \mathbf{b}; \mathbf{b}, L, s_3)$

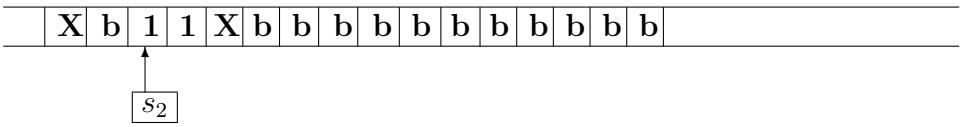
このチューリング機械に次のテープを指定されたヘッドの位置から動かしてみよう．



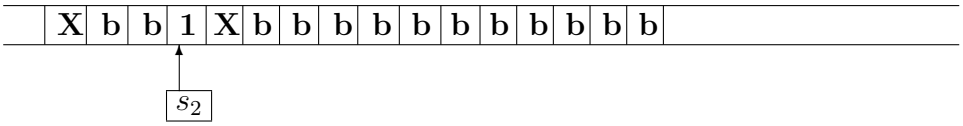
$(s_1, \mathbf{X}; \mathbf{X}, R, s_2)$ であるから，読んでいる記号 $\mathbf{X}$ はそのままで，本体の状態を s_2 に変えて，ヘッドを右に移す．

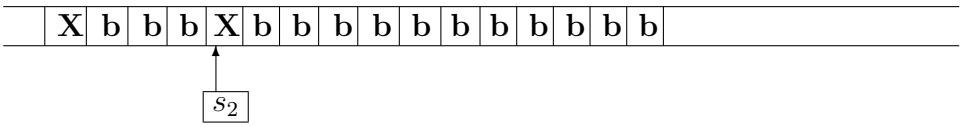


$(s_2, \mathbf{1}; \mathbf{b}, R, s_2)$ であるから，読んでいる記号 1 を $\mathbf{b}$ に書き換え，本体の状態は s_2 のままで，ヘッドを右に移す．

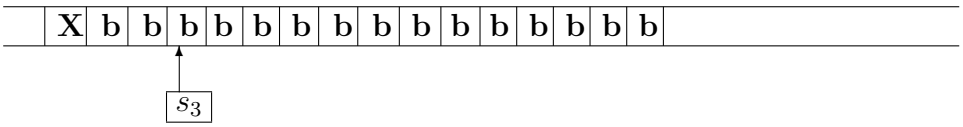


以下 2 ステップ同じように動く．

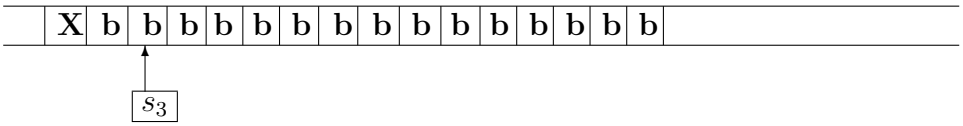




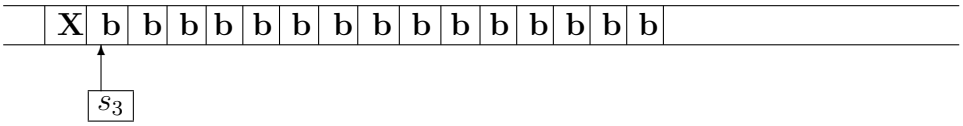
$(s_2, \mathbf{X}; \mathbf{b}, L, s_3)$ であるから，読んでいる記号 $\mathbf{X}$ を $\mathbf{b}$ に変え，本体の状態を s_3 に変えて，ヘッドを左に移す．

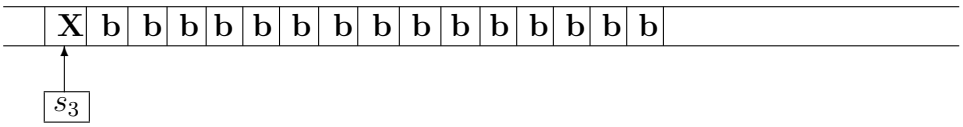


$(s_3, \mathbf{b}; \mathbf{b}, L, s_3)$ であるから，読んでいる記号 $\mathbf{b}$ はそのままで，本体の状態も s_3 のまま，ヘッドを左に移す．

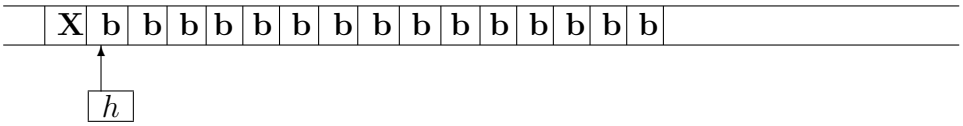


以下２ステップ同じように動く．

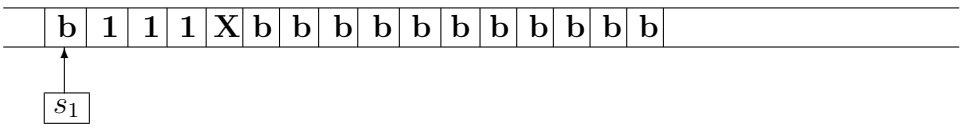




$(s_3, \mathbf{X}; \mathbf{X}, R, h)$ であるから，読んでいる記号 $\mathbf{X}$ はそのままで，本体の状態を h に変えて，ヘッドを右に移す．状態 h は停止状態であるのでチューリング機械は停止する．



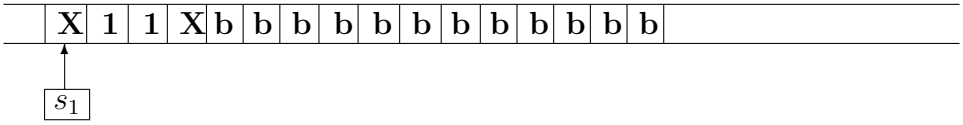
注意:チューリング機械はかならず停止するわけではない．例えば，上の例とおなじチューリング機械に次のテープを与え動作させると停止しない．



問題 1.2 停止しないことを確認せよ．

問題 **1.3** 次の基本操作を持つチューリング機械に下のテープをかけて動作を開始させるとテープ上にどんな記号を書き込んで停止するか，停止したときのヘッドの位置と停止したときにテープ上に書かれている記号を書け．但し，s_1 は初期状態，h は停止状態とする．

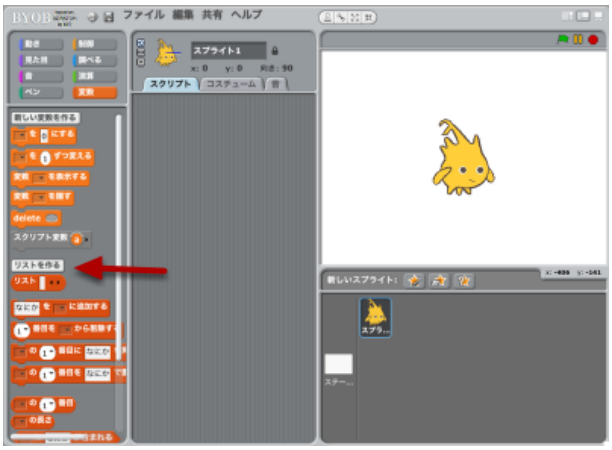
$(s_1, \mathbf{X}; \mathbf{b}, R, s_2), (s_2, \mathbf{1}; \mathbf{b}, R, s_3), (s_2, \mathbf{X}; \mathbf{X}, R, s_6),$
$(s_3, \mathbf{1}; \mathbf{1}, R, s_3), (s_3, \mathbf{X}; \mathbf{X}, R, s_3), (s_3, \mathbf{b}; \mathbf{1}, R, s_4),$
$(s_4, \mathbf{b}; \mathbf{1}, L, s_5), (s_5, \mathbf{b}; \mathbf{b}, R, s_2), (s_5, \mathbf{1}; \mathbf{1}, L, s_5),$
$(s_5, \mathbf{X}; \mathbf{X}, L, s_5), (s_6, \mathbf{b}; \mathbf{X}, L, h), (s_6, \mathbf{1}; \mathbf{1}, R, s_6)$



1.2 チューリング機械のシミュレータ作成

チューリング機械のシミュレータはいろいろ作られているが，ここでは BYOB のリストを利用したシミュレータを作成する．チューリング機械の動作を考えながら以下の手順を確認してほしい．なお，作成の詳細は BYOB で作るチューリング機械シミュレータ http://herb.h.kobe-u.ac.jp/tmsimulator/byob/index.html を参照して欲しい．また，リストを利用したシミュレータでは，簡単のため状態記号は文字一つで表すことにする．

1. BYOB を起動する．
2. 「リストを作る」で 3 つのリストを作成し，それぞれ本体，テープ，基本操作と名前を付ける．

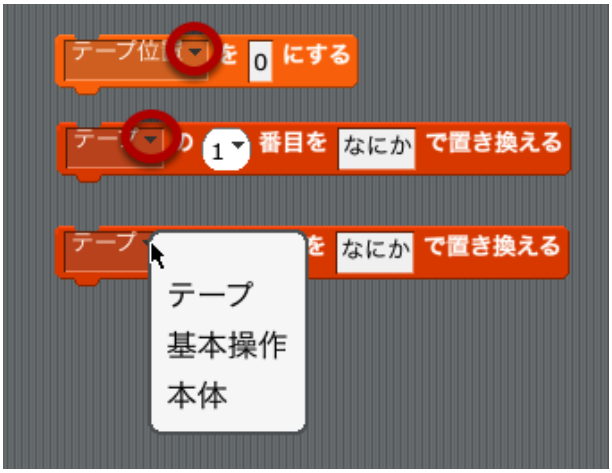


3. 本体が読んでいるテープの位置を表す変数と基本操作を探すための変数を作成する．名前はテープ位置と基本操作位置とする．変数は隠しておく．

4. 本体の 1 番目の文字が本体の状態を表すことになるが，可読性をよくするためブロックを作成して「本体の状態」で使えるようにする．ブロックの作成は変数カテゴリー内の最後尾にあるボタン「ブロックを作る」をクリックする．レポーター型を選び，「本体の状態」と名前を付ける．

5. 「report 本体の 1 番目」とすることで，ブロック「本体の状態」は本体の 1 番目を表すことになる．

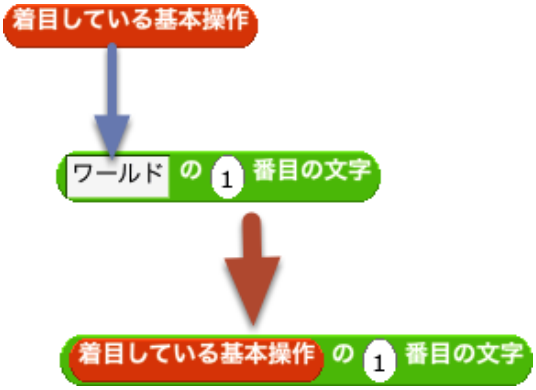
6. 変数やリストのブロックで変数名やリスト名を変えるには▼の部分をクリックすると定義済みの変数やリストに変更できる．



7. ヘッドの位置は変数「テープ位置」に記憶するので，ヘッドが見ている文字は「テープのテープ位置番目」で表すことができる．これも見やすくするためにレポータ型のブロックで「ヘッドの見ている文字」を作る．（カテゴリーはリストにする．）

8. 求める基本操作を見つけるために基本操作のリストを順番に探していくが，着目している基本操作は変数「基本操作位置」に記憶するので，着目している基本操作は「基本操作の基本操作位置番目」で表すことができる．これもわかりやすくするためレポータ型のブロックで「着目している基本操作」を作る．(カテゴリーはリストにする．)

9. 次の準備と本体のスクリプトの作成のときに同じブロックを何度も組み立てるのであらかじめよく使うブロックを作成しておいてそれを複製するようにすると便利である．



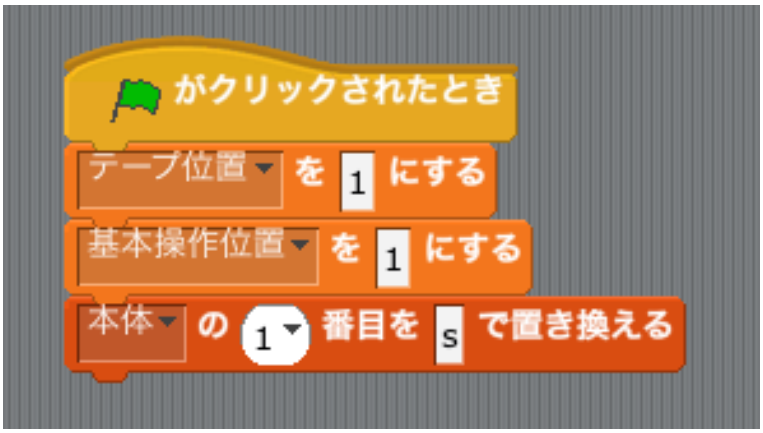
10. 本体の状態と見ているテープ記号に一致するかどうか（今着目している基本操作の１番目の文字が本体の状態と一致しかつ２番目の文字がヘッドが見ているテープの記号と一致するかどうか）判定する述語をブロックエディタで作る．

11. 「ブロックを作る」ボタンをクリックする．述語型で名前は「探している基本操作」にする．

12. report 部分にさきほど組み立てたブロックを入れる．

13. スクリプトは本体の状態と現在見ているテープ記号にマッチする基本操作を探し，見つけたらその基本操作の定義に従って動作するようにする．また，本体の状態が停止状態（ここではhを停止状態としている）になったらスクリプトをストップする．

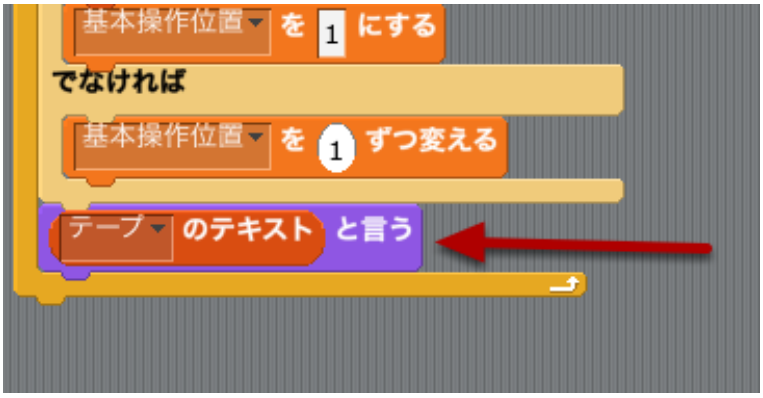
14. 最初にテープの位置と基本操作の位置を 1 にし本体の状態を初期状態にする．

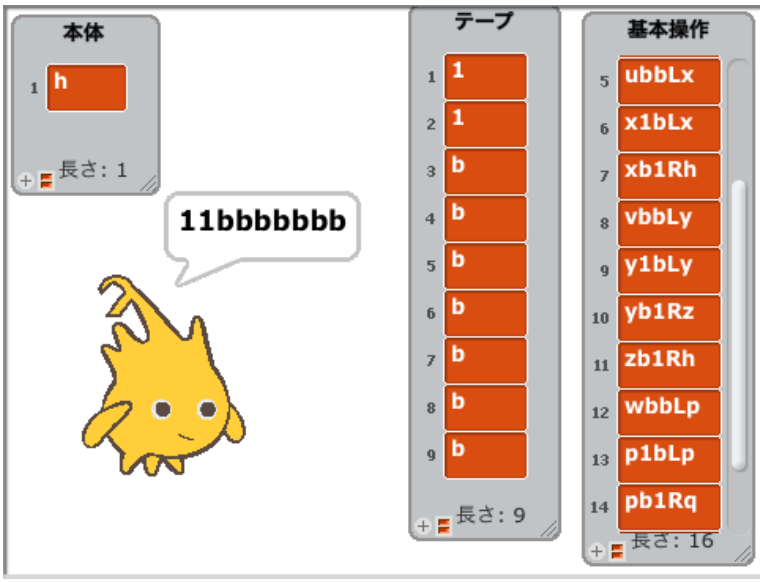


15. 本体の状態が停止状態（ここでは h）になるまでスクリプトは繰り返される．探している基本操作が現れるまで基本操作を探す．探している基本操作が見つかれば以下のことを行う．

16. 探している基本操作なので，その後に書かれている情報を読んでテープの書換，ヘッドの移動，状態の遷移を行う．テープ位置の文字を基本操作で指定される文字に書換し，テープのカーソル位置をヘッドの移動に応じて 1 増減し，最後に本体の状態を書き換えて基本操作の位置を先頭に戻す．なお，テープ位置が右端や先頭にある場合には，それぞれヘッドを右や左に動かす場合そこには空白文字があると考えるので，空白記号の b を挿入する．

17. BYOB ではリストは縦にのびるのでテープ上の記号の遷移を確認しやすいとはいえない．テキストを表示するブロックを使うとテープ表示が見やすくなる．具体的には，リストをテキスト化するブロックとテキストを言うブロックを組み合わせる．





実行例

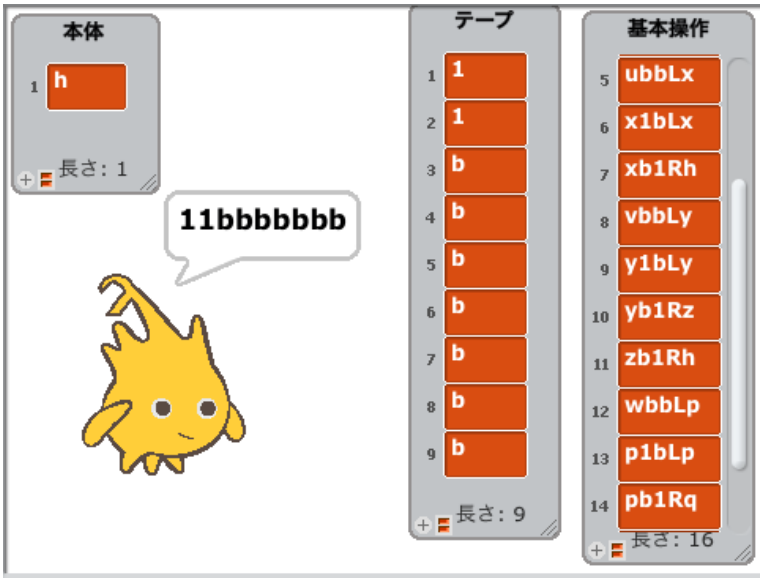
1. 次の基本操作は与えられた数を 3 で割ったときの余りを求めるものである．ただし，ここでは自然数 n を表すために 1 を n+1 個並べた文字列をとることにする．

   s1bRu
   u11Rv
   v11Rw
   w11Ru
   ubbLx
   x1bLx
   xb1Rh
   vbbLy
   y1bLy
   yb1Rz
   zb1Rh
   wbbLp
   p1bLp
   pb1Rq
   qb1Rr
   rb1Rh

   実行例はテープに 7（1 が 8 個）を書き込んでいる．

2. リストの要素はテキストファイルから読み込むことができるので，上記の基本操作をファイルに書いておいて読み込んでも良い．

3. スタートをクリックすると次のようにテープに 1 を 2 個残して状態が停止状態で停止する．これは 7 を 3 で割ったときの余りが 1 であることを表している．



問題 1.4 問題1.3で定義したチューリング機械を実際に作成したシミュレータで実行せよ．ただし，状態名は添字を使用せずに，適当な文字を与えよ．（全く異なる文字にすると間違え易いので，初期状態だけ s にして後の状態は添字の番号を状態として使うようにすると間違いが少ない．例えば，$(s_1, \mathbf{X}; \mathbf{b}, R, s_2)$ は sXbR2, $(s_2, \mathbf{1}; \mathbf{b}, R, s_3)$ は 21bR3 などどする．）

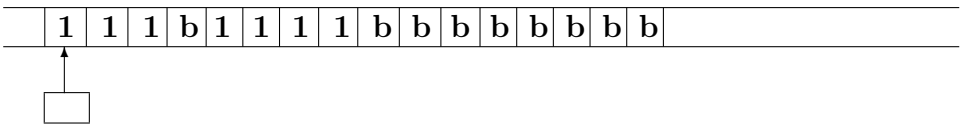
次に自然数上の関数を計算するチューリング機械を定義することを考える．$\mathbb{N}$ を自然数の集合[2]とし，φ を $\mathbb{N}$ 上の ℓ 変数関数とする．

この授業ではテープ記号に $\mathbf{1}$ があると仮定して自然数を $\mathbf{1}$ の有限列として表すことにする．具体的には自然数 x のテープ上の表現 $\overline{x}$ はテープ記号 $\mathbf{1}$ が $x+1$ 個ひきつづいたます目に書かれていることで表されるものとする[3]．M に $\overline{x_1}, \overline{x_2}, \cdots, \overline{x_\ell}$ を記入したテープをかけ，初期状態から動作を開始させると，有限時間内にテープに $\overline{\varphi(x_1, \cdots, x_\ell)}$ を書き込んで停止するとき，φ は M によって計算されると定義する．

このことをふまえて，自然数の足し算を計算するチューリング機械 M_+ を定義する．テープ記号の集合を $\{\mathbf{1}, \mathbf{b}\}$ とする．$\mathbf{b}$ は空白を表す記号とする．状態の全体は $\{h, s_1, s_2, s_3\}$ とする．

[2] 0 は自然数であるとする．
[3] x 個とすると 0 が表されないので，$x+1$ 個で表すことにする．

x と y をたしたいとき，テープ上には記号 **b** をはさんで，$\overline{x}$ すなわち $x+1$ 個の連続した **1** と $\overline{y}$ すなわち $y+1$ 個の連続した **1** を記入し，他のます目には **b** の書かれたテープが与えられるものとする．たとえば，$2+3$ を計算したいときには次のようなテープがかけられる．



計算の手続きは，テープ上の $\overline{x}\mathbf{b}\overline{y}$ に対し，$\overline{x}$ の左側から 1 を順次 $\overline{y}$ の右側に移し，全部移し終わったところで **1** をひとつ消して（自然数 x を表すのに $x+1$ 個の 1 で表すので 1 個余分になるので消して）停止する，というものにする．そのために，このチューリング機械の基本操作を次のように定める．

状態 h となったら，このチューリング機械は停止する（停止状態）．計算は，状態 s_1 でスタートすることにする（初期状態）．そこで，

$(s_1, \mathbf{1}; \mathbf{b}, R, s_2), (s_2, \mathbf{1}; \mathbf{1}, R, s_2), (s_2, \mathbf{b}; \mathbf{b}, R, s_3), (s_3, \mathbf{1}; \mathbf{1}, R, s_3)$

$(s_3, \mathbf{b}; \mathbf{1}, L, s_4), (s_4, \mathbf{1}; \mathbf{1}, L, s_4), (s_4, \mathbf{b}; \mathbf{b}, L, s_5), (s_5, \mathbf{1}; \mathbf{1}, L, s_5), (s_5, \mathbf{b}; \mathbf{b}R, s_1)$

と定めれば，$\overline{x}\mathbf{b}\overline{y}$ の $\overline{x}$ は $\overline{y}$ の右側にすっかり移され，s_1 の状態で **b** を読み込むことになる．従って

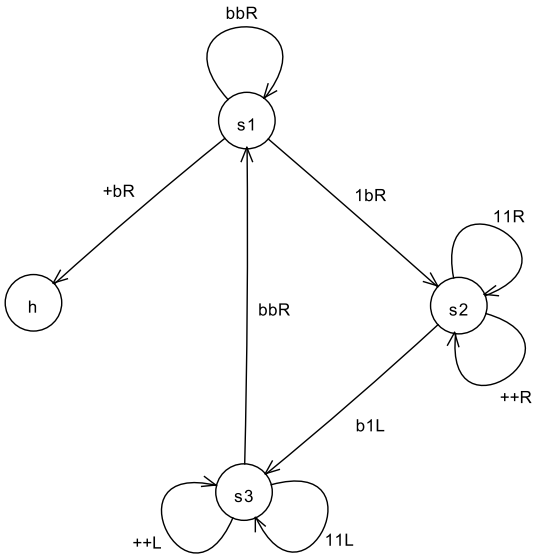
$$(s_1, \mathbf{b}; \mathbf{b}, R, s_6), (s_6, \mathbf{1}; \mathbf{b}, R, h)$$

とすればよい．

Remark 加算を行うチューリング機械として，最初の **1** を **b** に書き換えて，間の **b** を **1** に書き換え，最後に端の **1** を 1 個 **b** に書き換えて停止するというものでもかまわない．ただし，もう少し複雑なものを考える時には役にたたない方法である．

問題 **1.5** 上記の加算のチューリング機械をシミュレータで実行して動作を確認せよ．

基本操作は，しばしば，次のような図であらわされる．このような図は状態遷移図とよばれる．



問題 1.6 問題 1.3 で定義したチューリング機械の状態遷移図を書け.

n 個の自然数の組 $(x_1, \cdots, x_n)$ のテープ上の表現 $\overline{(x_1, \cdots, x_n)}$ は $\overline{x_1}\mathbf{b}\overline{x_2}\mathbf{b}\cdots\mathbf{b}\overline{x_n}$ であると約束する.

問題 1.7 自然数上の 1 変数関数 $S(x) = x + 1$ を計算するチューリング機械を作り, BYOB 上のシミュレータで動作を確認せよ.（以下の問題でも同様に確認せよ.）

問題 1.8 $\overline{(x_1, x_2, x_3)}$ が書かれたテープの **1** の左端にヘッドをセットすることにする. このとき $\overline{x_3}$ の左端の **1** で止まる（すなわち間にある **b** を 2 個読んで停止する）チューリング機械を作れ.

問題 1.9 上の問題のチューリング機械を参考にして, $\varphi(x, y, z) = y$ を計算するチューリング機械を作れ.

問題 1.10 問題 1.3 で定義したチューリング機械は, **X** の間に挟まれた 1 の 2 倍の数の 1 を書き込んで停止する. これを参考にして, $\varphi(x) = 2x$ を計算するチューリング機械を作れ.

問題 1.11 自然数上の 2 変数関数 $f(x, y)$ を次のように定義する.

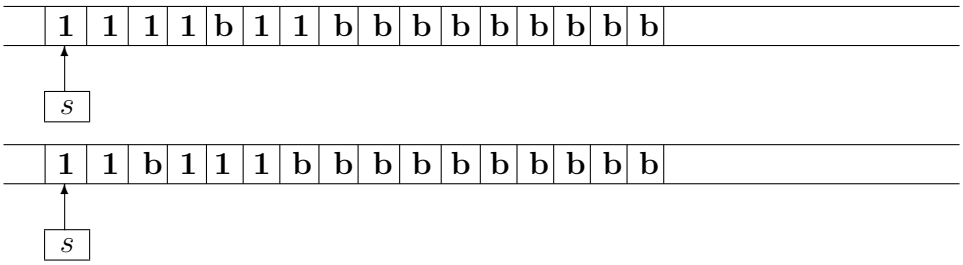
$$f(x,y) = \begin{cases} x & if \quad x \text{は偶数} \\ y & if \quad x \text{は奇数} \end{cases}$$

$f(x, y)$ を計算するチューリング機械を作れ.

問題 **1.12** 自然数上の２変数関数 $f(x,y) = x \dot{-} y$ を次のように定義する．（0 は自然数としている．）

$$x \dot{-} y = \begin{cases} x - y & if \quad x \geq y \\ 0 & if \quad x < y \end{cases}$$

$f(x,y)$ を計算するチューリング機械を作れ．作成したチューリング機械に対して，以下の二つのテープに対して動作を確かめよ．（但し，s は初期状態とする．）



ヒント：$\overline{x}$ と $\overline{y}$ から 1 つずつ **1** を消すとよいのであるが，$\overline{x}$ が先に消える場合と $\overline{y}$ が先に消える場合があるので，その判定をどうするか考える．

問題 **1.13** $\Sigma = \{0,1\}$ とする．Σ 上の有限列（0 と 1 からなる有限列）が与えられたとき，それが回文であるかどうか判定するチューリング機械を作れ．

ヒント：どうなったら回文であるとわかるか考える．長さが偶数の場合と奇数の場合があるのでその処理をどうするか考える．判定の表示の仕方はいろいろある．例えば，停止状態を２種類用意して（h_1, h_2 とする），h_1 で停止したら回文で，h_2 で停止したら回文でないように作るか，あるいはテープ上に **1** を 1 つだけ書き込んで停止したら回文で，**0** を 1 つだけ書き込んで停止したら回文ではないとすればよい．

チューリング機械の定義では非常に単純な能力しか考えていないが，実際にはチューリング機械の潜在能力は十分に大きく，テープの数を増やすなどして能力をつけ加えてたとしても，本質的な能力は全く増えないことがわかる．従って，ある問題が，コンピュータによって（原理的に）解けるか否かを議論するためには，チューリング機械の能力の限界をさぐれば十分であることがわかる．

1.3 チューリング機械の数学的定義

チューリング機械は数学的には次のように定義される．

定義 **1.14** 以下の条件を満たす体系 $M = (Q, \Gamma, \delta, s, H)$ をチューリング機械という．

1 Q は状態の有限集合

2 $H \subseteq Q$ は停止状態の集合

3 Γ はアルファベットの有限集合で空白記号 **b** を含む

4 $\delta : (Q \backslash H) \times \Gamma \longrightarrow \Gamma \times \{L, R\} \times Q$

5 $s \in Q$ は初期状態

遷移関数 δ は，グラフ[4]を考えれば，基本操作の有限集合である．(以下，δ のグラフを Δ と書き，関数 δ の代わりにしばしば用いる）δ が関数であることから，Δ に属する任意の基本操作

$$(s_1, a_1; \alpha_1, \beta_1, \gamma_1) \text{ と } (s_2, a_2; \alpha_2, \beta_2, \gamma_2)$$

に対し，$(s_1, a_1) = (s_2, a_2)$ ならば $(\alpha_1, \beta_1, \gamma_1) = (\alpha_2, \beta_2, \gamma_2)$ である．これは M の基本操作が，状態と読み込まれたテープ記号から唯一に定まることを意味している．

Remark:この性質はチューリング機械の決定性と呼ばれる．この性質をもたないもの，すなわち基本操作が，状態と読み込まれたテープ記号から唯一に定まらず，2つ以上の異なった動作をすることもありうるものもチューリング機械の仲間に加えて，これを非決定性チューリング機械と呼ぶ．非決定性チューリング機械も考慮に入れるときは，決定性を持つチューリング機械を決定性チューリング機械と呼ぶ．

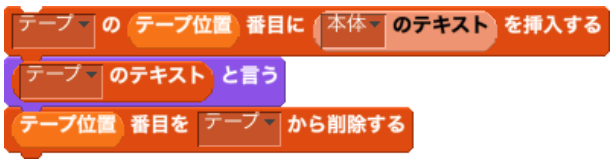
次にチューリング機械による“計算”を形式的に定義するために，チューリング機械 M の計算過程を定める．

チューリング機械においては有限回のステップの間に眺めたり書き換えたりすることのできるます目の総数は有限であるから，最初に与えられるテープ上に書かれている空白記号以外の記号が有限であったので，テープ上で空白記号以外の記号が書かれているます目は常に有限個である．従って，テープを Γ の要素の有限列として表すことができる．このことを念頭に以下のように時点表示を定義をする．

定義 1.15 Γ の要素と Q の要素を有限個並べた列で Q の要素をただ 1 個だけ含む記号列をチューリング機械 M の時点表示という．時点表示から Q の要素を取り除いた記号列を，時点表示から得られるテープ表示という．

BYOB で作成したチューリング機械シミュレータでスプライトの吹き出しに現れるのは各時点におけるテープ表示である．シミュレータのスクリプトの最後を次のように修正すると，時点表示がスプライトの吹き出しの中に現れる．

[4]関数 $f : X \to Y$ のグラフとは集合 $\{(x, y) \in X \times Y \mid y = f(x)\}$ のことである．



Γ の要素の有限列全体の集合（空列も含む）を Γ^* で表すことにすると，時点表示は $\tau q\tau'(\tau, \tau' \in \Gamma^*, q \in Q)$ と表される．

時点表示 $a_1a_2\cdots a_{i-1}qa_i\cdots a_n$ はテープ

$$a_1a_2\cdots a_n$$

に対して，チューリング機械が状態 q で a_i の書かれているます目を眺めていることだと解釈すると，時点表示によりチューリング機械の各時点での様子を表していると考えられる．この解釈のもとでは，時点表示 α, β に対し，空白記号の有限列 $\boldsymbol{b}_1, \boldsymbol{b}_2, \boldsymbol{b}_3, \boldsymbol{b}_4$ が存在して，$\boldsymbol{b}_1\alpha\boldsymbol{b}_2 = \boldsymbol{b}_3\beta\boldsymbol{b}_4$ となるとき，時点表示 α, β は同じ状況を表していると考えられるから，この二つを同一視し，$\alpha \sim \beta$ で表すことにする．（但し，記号列としては異なることに注意すること．）

定義 1.16 α, β が M の時点表示で，α の状況が M の基本操作により β にかわるとき，

$$\alpha \to_M \beta$$

と書き，M の計算状況という．すなわち，M の計算状況はつぎの 4 条件により定義される．

1 時点表示 α が $\tau_1a'sa\tau_2$ であるとき，$(s, a; a'', L, s') \in \Delta_M$ ならば
$\alpha \to_M \tau_1s'a'a''\tau_2$ は M の計算状況である．

2 時点表示 α が $sa\tau_2$ であるとき，$(s, a; a'', L, s') \in \Delta_M$ ならば
$\alpha \to_M s'ba''\tau_2$ は M の計算状況である．

3 時点表示 α が $\tau_1saa'\tau_2$ であるとき，$(s, a; a'', R, s') \in \Delta_M$ ならば
$\alpha \to_M \tau_1a''s'a'\tau_2$ は M の計算状況である．

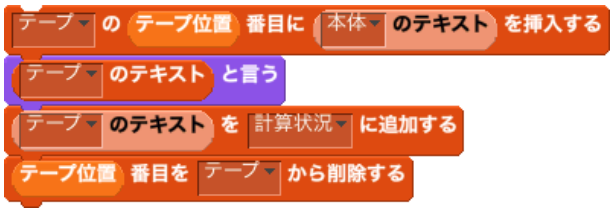
4 時点表示 α が τsa であるとき，$(s, a; a', R, s') \in \Delta_M$ ならば
$\alpha \to_M \tau a's'\mathbf{b}$ は M の計算状況である．

条件 2 と条件 4 より与えられたテープ表示の外側には空白記号が並んでいると解釈できる．混乱の恐れがない限り，$\to_M$ は単に $\to$ で表すことにする．

例 **1.17** 例 1.1 で与えたチューリング機械と与えられたテープに対し，最初の 6 ステップの計算状況は以下のようになる．

$$s_1\mathbf{X}111\mathbf{X} \to \mathbf{X}s_2111\mathbf{X} \to \mathbf{X}\mathbf{b}s_211\mathbf{X} \to \mathbf{X}\mathbf{b}\mathbf{b}s_21\mathbf{X} \to \mathbf{X}\mathbf{b}\mathbf{b}\mathbf{b}s_2\mathbf{X} \to \mathbf{X}\mathbf{b}\mathbf{b}s_3\mathbf{b}\mathbf{b} \to \mathbf{X}\mathbf{b}s_3\mathbf{b}\mathbf{b}\mathbf{b}$$

BYOB のチューリング機械シミュレータで，「計算状況」というリストを作成し次のようなブロックを初期設定の部分と最後に追加すると計算状況がリストに現れる．



例 1.1 のチューリンング機械に適用すると以下のように表示される．（ただし，初期状態は s，他の状態は添字の数字，停止状態は h で表している．）

問題 **1.18** 問題 1.3 で定義したチューリング機械と与えられたテープに対し，最初の 10 ステップの計算状況を書け．

定義 **1.19** 以下の条件を満たすような，時点表示の有限列 $\alpha_0, \cdots, \alpha_n$ をチューリング機械 $M = (Q, \Gamma, \delta, s, H)$ による計算過程という．

1 $\alpha_{i-1} \to \alpha_i \quad (i = 1, 2, \cdots, n)$

2 α_0は $s\tau$ の形をしている．

3 α_nは$\tau_1 h \tau_2 \quad (h \in H)$ の形をしている．

定義の内容から明らかなように，計算過程はテープの左端（そこから左側には空白記号しかないところ）のます目にヘッドをセットし，初期状態で M をスタートさせると，順次基本操作を繰り返して，停止状態となって停止するまでの経過を記述したものである．チューリング機械のシミュレータの計算状況のリストは計算過程そのものである．

前にも述べたが，自然数上の関数が計算されることは，この計算過程を用いて，きちんと述べることができる．

定義 1.20 $\varphi(x_1, \cdots, x_n)$ を自然数上の関数とする．チューリング機械 $M = (Q, \Gamma, \delta, s, H)$ が φ を計算するとは，任意の自然数の組 $(x_1, \cdots, x_n)$ に対して，時点表示 $s\overline{(x_1, \cdots, x_n)}$ で始まり，$h\overline{\varphi(x_1, \cdots, x_n)}(h \in H)$ に同値な時点表示で終わる計算過程が存在するときをいう．

定義 1.21 関数 φ は，φ の値を計算する適当なチューリング機械を定めることができるとき，（チューリングの意味で）計算可能であるという．

1.4 万能チューリング機械

万能チューリング機械とは，あらゆるチューリング機械の動作をまねるように組み立てられているものである．ここではチューリング機械のアルファベットの集合はすべて共通とし，例えば $\Gamma = \{b, 1, X\}$ とする．また，状態記号の集合 Q も $s_1, s_2, \cdots, s_n, \cdots$ の有限部分集合とする．

万能チューリング機械は次のように定義される：まず，任意のチューリング機械 $M = (Q, \Gamma, \delta, s, H)$ に対し，M の構造についての情報を $\Gamma = \{b, 1, X\}$ の記号列 $\ulcorner M \urcorner$ としてテープ上に表現する方法を定める．万能チューリング機械 U は，$\ulcorner M \urcorner$ と M に与える入力列 σ を見て，$\ulcorner M \urcorner$ を解読し，M の基本操作に相当する動作をし，計算を進める．最後に U は M の停止状態に相当する動作をして停止する．従って，U は M が動作中にテープに書く記号列に相当するものを同じように書いており，停止時に U のテープを見ることで M の計算結果を知ることができる．

定理 1.22 (停止問題の判定不能性) 任意のチューリング機械 M と任意の記号列 σ に対して，M が σ を入力として行う計算が停止するか否かを判定するチューリング機械は存在しない．

この定理より，任意のプログラムに任意のデータを与えて実行した時にそれが停止するかどうかを前もって判断するプログラムは原理的に存在しないことになる．

定理 1.22 の証明：任意のチューリング機械 M と任意の記号列 σ に対して，$\ulcorner M \urcorner$ と σ を入力として与えたとき，M が σ を入力として行う計算が停止するか否かを判定するチューリング機械 T_0 が存在したとする．このとき，次のようなチューリング機械 T_1 を作成する．T_1 は $\ulcorner M \urcorner$ と M の入力列 σ を入力として受け取り，T_0 と同じ動きを行う．T_0 が計算が停止すると判定したときは，T_1 は停止せずに無限に計算を続け（何を読んでも右に動くようにする），T_0 が計算が停止しないと判定したときは，T_1 は停止するようにする．

次に T_1 を用いてチューリング機械 T を次のように作成する．T は $\ulcorner M \urcorner$ を入力として受け取ると，$\ulcorner M \urcorner$ のコピーをテープ上に作成して，T_1 の基本操作をそのまま行う．T_1 は $\ulcorner M \urcorner$ と $\ulcorner M \urcorner$ を M への入力としたときの動きをシミュレートする．すなわち，M に入力 $\ulcorner M \urcorner$ を与えたときの動きを行い，上で定義したように停止するかあるいは停止せずに無限に計算を続ける．

T に自分自身のコード $\ulcorner T \urcorner$ を与えた時，T は停止するかあるいは無限に計算を続けるか考える．T に自分自身のコード $\ulcorner T \urcorner$ を与えたとき停止するとする．このとき T の定義より，T_1 は $\ulcorner T \urcorner$ と $\ulcorner T \urcorner$ を受け取って計算を行い停止する．一方，T に自分自身のコード $\ulcorner T \urcorner$ を与えたとき停止するから，T_0 は停止すると判定する．従って，T_1 の定義より，T_1 は $\ulcorner T \urcorner$ と $\ulcorner T \urcorner$ を受け取って計算を行い停止しない．これは矛盾である．

逆に，T に自分自身のコード $\ulcorner T \urcorner$ を与えたとき停止しないとすると，T_1 は $\ulcorner T \urcorner$ と $\ulcorner T \urcorner$ を受け取って計算を行い続け停止しない．一方，T に自分自身のコード $\ulcorner T \urcorner$ を与えたとき停止しないから，T_0 は停止しないと判定する．従って，T_1 の定義より，T_1 は $\ulcorner T \urcorner$ と $\ulcorner T \urcorner$ を受け取って計算を行い停止することになるが，これも矛盾である．

T に自分自身のコード $\ulcorner T \urcorner$ を与えた時，停止するとしても停止しないとしても，いずれにしても矛盾する．これは，任意のチューリング機械 M と任意の記号列 σ に対して，M が σ を入力として行う計算が停止するか否かを判定するチューリング機械 T_0 が存在すると仮定したことによる．よって，任意のチューリング機械 M と任意の記号列 σ に対して，M が σ を入力として行う計算が停止するか否かを判定するチューリング機械は存在しない．

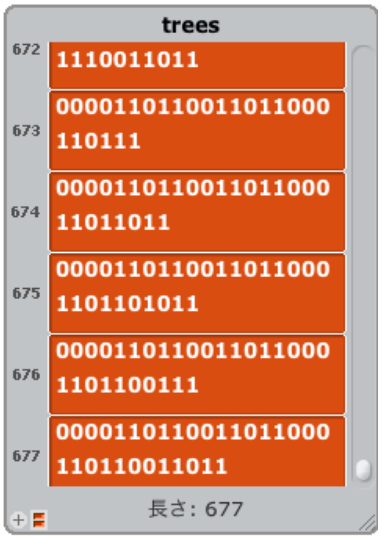
2　BYOB で計算可能な関数

“計算可能な関数”というと一番わかりやすいイメージはそれを計算するプログラムがあるということである．関数型言語や C などを使って計算可能性を議論している文献として [1, 2] などがあるが，ここでは BYOB で計算できる関数を定義する[5]．BYOB の機能はたくさんあるのでそれらを何でも利用していいとなると議論が面倒になるので，ここでは機能を制限することにする．

2.1　再帰的構造と BNF 記法

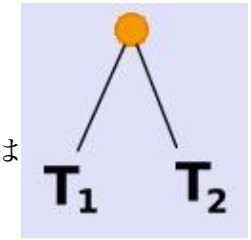
次の節で BYOB 関数ブロックを再帰的定義を用いて定義するが，その構造がやや複雑なので，最初にやさしい例で再帰的定義を説明する．
次の R1 から R3 は hoge という性質をもつ 0 と 1 からなる有限列の全体を定義するものである．
R 1　1 は hoge である．
R 2　T_1 と T_2 が hoge のとき，$0T_1T_2$ も hoge である．
R 3　以上によって，hoge とわかるもののみが hoge である．
どのような 01 有限列が hoge の性質をもつか考えてみよう．まず最初に hoge とわかるのは， R 1 より 1 である．この段階で 1 が hoge であることがわかったので，R 2 の T_1 と T_2 として，1 を適用することができる．従って，011 が hoge であることが R2 よりわかる．この段階で，1 と 011 が hoge であることがわかった．次に R 2 の T_1 と T_2 として，1 と 011 のいずれかをとることにより，新たに 01011,,00111, 0011011 が hoge であることがわかる．この段階で 5 個の 01 有限列 1,011,01011,00111, 0011011 が hoge であることがわかる．同じように続けると，次の第 3 段階では新たに $5^2 - 4 = 21$ 個の新しい hoge が生成される．この操作を続けることで，hoge が生成されていく．最後にこの操作で生成されないものは，hoge ではないことが R 3 によってわかる．
次の BYOB プログラムを実行すると，第 4 段階までの hoge をすべて生成することができる．

[5] BYOB の特徴からすると λ 計算的に計算可能性を議論するのが本来の姿かもしれないが，ここでは while プログラム的に考えることにする．

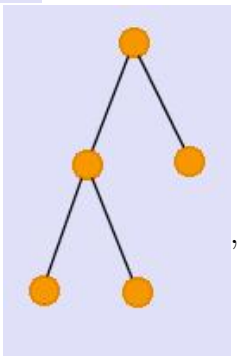
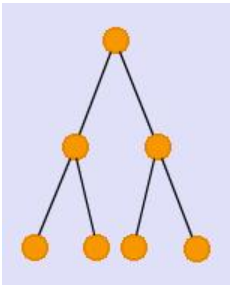


問題 **2.1** 実際に BYOB で上記のスクリプトを作成して実行せよ．なお，変数として `stage`，`length`，`pre`，`i`，`j` リストとして `trees` を作成している．`pre` の役割は，重複して同じ hoge を生成しないためである．

上の定義では，余計な先入観が入らないように hoge としたが，ここで hoge とよばれた記号列は以下のように解釈することで，全２分木 (full binary tree) とよばれる構造をもつことがわかる．

1 には  ，$0T_1T_2$ には 

という図形を対応させる．

この対応により，例えば 00111 には  , 0011011 には 

が対応する．一般に再帰的定義は R1,R2,R3 のような書き方が使われるが，プログラミング言語のように一度にたくさんの概念を定義する場合は BNF 記法あるいは BNF 記法を拡張した EBNF 記法が用いられる．例えば最初に定義した hoge は，

⟨hoge⟩::=1 | 0⟨hoge⟩⟨hoge⟩

と１行で表される．ここで『＊＊::=××』は『＊＊とは××である』と読み、縦棒『|』は『か』とか『または』と読む．従って，この通り読むと，「hoge とは 1 であるか，あるいは 0 のあとに hoge と hoge をつないだものである」となる．

左辺に現れる概念を，右辺で定義しようとするが，その中に定義しようとするものが現れている．循環論法のように見えるが，実際は上で確認したように右辺に現れた hoge は「すでに hoge とわかっているもの」と解釈しているので循環論法にならない．

問題 2.2 ２分木を次のように BNF 記法で定義する。

⟨２分木⟩::=1 | 0⟨二分木⟩| 0⟨二分木⟩⟨二分木⟩

この定義で２分木とわかるものを第２段階まで書き出せ．また，二分木を生成するプログラムを BYOB を用いて作成せよ．

2.2 BYOB 関数ブロック

以下のように同時並行的に BYOB 関数ブロックを定義する．

1. 式とはレポーター型のブロックで次のように再帰的に定義する．

ここで func は関数ブロックで後で定義される．

2. ブール式は述語型のブロックで次のように再帰的に定義する．

3. プログラムはコマンド型のブロックで次のように再帰的に定義する．それぞれ代入文，複合文，条件文，until 文とよぶ．

4. 関数ブロックはレポーター型のブロックで次のように定義する．

ブロックの作成は変数カテゴリー中にある「ブロックを作る」で行い，レポーター型のブロックとして作成する．ブロック名は自由である．また，ブロック内では仮引数用の変数として x1,x2,...,xn，出力変数として a を用い，a の値を report することにする．なお，スクリプト変数[6]は a 以外にも自由に使えるものとする．

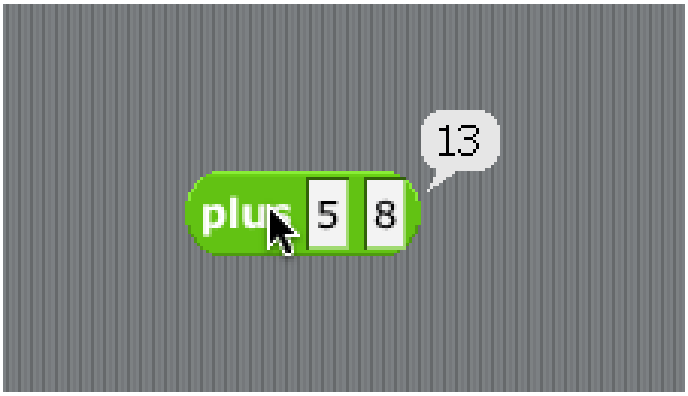
定義 2.3 $\varphi(x_1, \cdots, x_n)$ を自然数上の関数とする．関数ブロック func [ ] [ ] ... [ ] が φ を計算するとは，任意の自然数の組 $(k_1, \cdots, k_n)$ に対して，func k1 k2 ... kn が $\varphi(k_1, \cdots, k_n)$ の値を報告するときをいう．ただし，実際の BYOB では扱える数に上限があるが，ここではいくらでも大きい数が扱えるものとして考える．

定義 2.4 関数 φ は，φ の値を報告する適当な BYOB 関数ブロックを定めることができるとき，**BYOB で計算可能**であるという．また，その BYOB 関数ブロックは関数 φ を計算するという．

例 1 $\mathrm{plus}(a, b) = a + b$ は BYOB で計算可能である．

[6]BYOB ではグローバル変数もスクリプト変数もいずれも変数は常に初期値として 0 が与えられる．

とすると，例えば $(5, 8)$ に対し次のように $\mathrm{plus}(5, 8) = 13$ を報告する．



例 2 $\mathrm{mult}(a, b) = a * b$ は BYOB で計算可能である．

plus を計算する関数ブロックの定義中の x1+x2 を x1*x2 にすればよい．

上記の例は最初から式に+や*があるので，BYOB で計算可能であるのは当たり前であるが，他にどのような関数が BYOB で計算可能であるかということについては次章の帰納的関数において述べることにする．

3 帰納的関数

3.1 値の計算できる関数

ここでは，具体的に変数に値が与えられたら，その値をもとに関数の値が“計算できる”ような関数を考えていく．ここで“計算できる”というのは，あいまいな概念であり，すぐに明確に定義できるものではないが，我々の日常の言葉としての

“計算”ととりあえず考える．例えば，足し算 $f(x,y)=x+y$ は自然数 m と n が与えられたら，値 $m+n$ は“計算できる”ものとする．

数列 $\{a_n\}$ を次の漸化式で定義する．

$$\begin{cases} a_1 = 1 & \cdots(1) \\ a_{n+1} = a_n + n & \cdots(2) \end{cases}$$

これらの式により数列が定義できているという意味は，“どんな番号 N を与えても N 番目の数 a_N の値が必ず求めることができる”ので数列が決定できているというものであった．

例えば，$N=5$ とすると，(2) 式より $a_5 = a_{4+1} = a_4+4$ であり，再び (2) 式を使うと $a_5 = (a_3+3)+4$ となる．このように (2) 式の適用を続けると $a_5 = (((a_1+1)+2)+3)+4$ となり (1) 式から $a_1=1$ であるので $a_5 = (((1+1)+2)+3)+4$ となり，足し算を計算すると，$a_5 = 11$ と値を求めることができる．

数列の漸化式による定義を関数にも適用することを考えよう．例えば，$g(x)$ という関数を

$$\begin{cases} g(1) = 1 \\ g(x+1) = g(x) + x \end{cases}$$

と定義することを考えよう．最初の数列 a_n の定義と全く同じであるのでどんな自然数 n を与えても $g(n)$ の値を計算できることがわかる．

定義中の足し算を $f(m,n)$ を用いて表すと，

$$\begin{cases} g(1) = 1 \\ g(x+1) = f(g(x), x) \end{cases}$$

と表される．ここで f として足し算以外の関数をとって，$g(3)$ の値を考えてみよう．

$g(3) = g(2+1) = f(g(2),2)$ であるが，$g(2) = g(1+1) = f(g(1),1)$ で $g(1)=1$ であるから $g(2) = f(1,1)$ となり，$g(3) = f(f(1,1),2)$ となる．従って，f の値を計算することができるのなら，$g(3)$ の値も計算できることになる．

このような関数の定義を，関数の再帰的あるいは帰納的定義と呼ぶ．上に述べたことを，この言葉を用いて言い換えると，“値の計算できる関数”を用いて再帰的に定義できる関数も“値の計算できる関数”になるということである．

ここまで足し算 $f(m,n)=m+n$ は“値の計算できる関数”として仮定してきたが，この足し算をもっと簡単な関数から定義してみよう．

まず，最初に自然数の定義（ペアノの公理）を確認する．自然数の公理は，定数記号 0, 1 変数関数記号 $'$ を用いて書かれる．（変数記号はもちろん使用する．）

1. 0 は自然数である．
2. x が自然数ならば，x' は自然数である．
3. x, y が自然数で $x' = y'$ ならば $x = y$ である．
4. x が自然数ならば，$x' \neq 0$ である．
5. （数学的帰納法の原理）自然数 x に関する命題 $P(x)$ が次の 2 つの性質
   (a) $P(0)$
   (b) $P(x)$ ならば $P(x')$

   が成り立つならば，任意の自然数 x に対し，$P(x)$ が成り立つ．

最初の 2 つの条件から，自然数とわかるものは $0, 0', 0'', 0''', \cdots$ と表されるものである．x に対し，x' を x の次の自然数とよぶことにする．さらに 0 の肩に「$'$」が k 個ついたものを「k」と略記する．これにより従来通り，自然数を $0, 1, 2, 3, \cdots$ と表すことができる．

$\mathrm{succ}(x) = x'$ とすると，$\mathrm{succ}(x)$ は与えられた自然数の次の自然数を値に持つ関数である．我々は自然数を知っている（少なくとも定義は知っている）という立場で考えるので，これは明らかに“値の計算できる関数”である．この $\mathrm{succ}(x)$ を用いて，次のように足し算を再帰的に定義することができる．f が足し算であることは数学的帰納法で確かめることができる．ここでは $m+1$ ではなく，m' を用いていることに注意する．

$$\begin{cases} f(0, n) = n \\ f(m', n) = \mathrm{succ}(f(m, n)) \end{cases}$$

従って，足し算も“値の計算できる関数”として考えることができるのである．では，かけ算はどうであろうか．かけ算 $g(x, y) = xy$ は次のように足し算を用いて再帰的に定義できるので，やはり“値の計算できる関数”と考えることができる．

$$\begin{cases} g(0, n) = 0 \\ g(m', n) = f(g(m, n), n) (= g(m, n) + n) \end{cases}$$

次に $h(x,y)=(x+y)^2$ という関数を考えてみよう．この関数の値がどのように計算されるのか具体的に見てみよう．例えば，$h(2,3)$ の値は

$h(2,3)=(2+3)^2=5^2=5\times 5=25$ となる．値の求め方は，まず最初に足し算 $2+3$ を計算して（足し算は“値の計算できる関数”），$2+3=5$ を求め，次にかけ算（かけ算も“値の計算できる関数”）5×5 を計算して $5\times 5=25$ が得られる．

$h(x,y)$ を足し算 $f(x,y)$ とかけ算 $g(x,y)$ を用いて表すと $h(x,y)=g(f(x,y),f(x,y))$，すなわち $f(x,y)$ と $g(x,y)$ の合成関数として定義されるのである．もう一度，g と f を用いて計算の仕組みを考えると，$h(2,3)=g(f(2,3),f(2,3))$ で f が“値の計算できる関数”であるから値を計算すると $f(2,3)=5$ になるので，この値を代入すると $h(2,3)=g(5,5)$ となり，g も“値の計算できる関数”であるから値を計算すると $g(5,5)=25$ となる．

このことから何がわかるかというと，“値の計算できる関数”を合成してできる関数も“値の計算できる関数”と考えることができるということである．

3.2 原始帰納的関数

以上の考察をもとにして，原始帰納的関数と呼ばれる関数のグループを定義する．まず，最初に次のような 3 種類の関数を考える．

1. $\mathrm{succ}(x)=x'$
2. $\mathrm{zero}(x)=0$
3. $\mathrm{u}_i^n(x_1,\cdots,x_n)=x_i$

これらを初期関数 (initial functions) とよぶ．初期関数はどれも“値の計算できる関数”と考えることのできる関数であることに注意しよう．２つ目の関数は，どんな変数の値に対しても，常に 0 を関数の値としてとるのであるから，いつでも値は計算（何もせずに！）できる．最後の関数は，与えられた変数の値の中から指定された変数の値を関数の値としてとるのであるから，やはりこれも値を計算することができる．

定理 3.1 初期関数はすべて BYOB で計算可能である．

それぞれ関数ブロックを次のように定義すればよい．

次の２種類の操作を考える．

1. （合成の操作）

   $h_1(x_1,\cdots,x_n), h_2(x_1,\cdots,x_n),\cdots,h_m(x_1,\cdots,x_n), g(y_1,\cdots,y_m)$ が与えられたとき，これから f を次のように定義する．この操作を合成とよぶ．

$$f(x_1,\cdots,x_n) = g(h_1(x_1,\cdots,x_n), h_2(x_1,\cdots,x_n),\cdots,h_m(x_1,\cdots,x_n))$$

2. （原始帰納）

$g(x_1,\cdots,x_n)$ と $h(y,z,x_1,\cdots,x_n)$ が与えられたとき，これから f を次のように定義する．この操作を原始帰納とよぶ．

$$\begin{cases} f(0,x_1,\cdots,x_n)=g(x_1,\cdots,x_n) \\ f(k',x_1,\cdots,x_n)=h(f(k,x_1,\cdots,x_n),k,x_1,\cdots,x_n) \end{cases}$$

初期関数から合成と原始帰納の操作を有限回繰り返して得られる関数を原始帰納的関数 (primitive recursive function) とよぶ．初期関数は全域関数であるから原始帰納的関数も全域関数である．与えられた関数が原始帰納的であることを示すためには，それが初期関数からどのように作られるか示すことが求められる．原始帰納的関数は“値の計算できる関数”である初期関数から合成や原始帰納の操作を用いて定義される関数であるから，“値の計算できる関数”と考えることができる．

定義 3.2 関数列 $\phi_0,\cdots,\phi_n$ が次の条件をみたすとき，この列を ϕ の原始帰納的記述 (*primitive recursive description,prd)* とよぶ．

1 ϕ_0は初期関数である．

2 $\phi_i\,(1\le i\le n)$ は初期関数であるか，或いは$\phi_{j_1},\cdots,\phi_{j_m}\,(j_k\le i-1,k=1,2,\cdots,m)$ から合成または帰納法によって定義される．

3 $\phi_n=\phi$

ϕ が原始帰納的関数であるとは，ϕ の *prd* が存在することである．

例 1 $\mathrm{plus}(a,b)=a+b$ は prf である．

$\phi_0(x)=\mathrm{succ}(x),\phi_1(x)=\mathrm{u}_1^1(x),\phi_2(x,y,z)=\mathrm{u}_1^3(x,y,z),\phi_3(x,y,z)=\phi_0(\phi_2(x,y,z))$

$$\begin{cases} \phi_4(0,x)=\phi_1(x) \\ \phi_4(k',x)=\phi_3(\phi_4(k,x),k,x) \end{cases}$$

とすると，$\phi_0.\phi_1,\phi_2,\phi_3,\phi_4$ は plus の prd である．$\phi_4(k,x)=\mathrm{plus}(k,x)$ であることは，k に関する数学的帰納法により示すことができる．

$k=0$ のとき，$\phi_4(0,x)=\phi_1(x)=u_1^1(x)=x=0+x=\mathrm{plus}(0,x)$ であるから成り立つ．

$k=i$ のとき成り立つとすると，$k=i'$ のとき，

$\phi_4(i',x)=\phi_3(\phi_4(i,x),i,x)=\phi_0(\phi_2(\phi_4(i,x),i,x))=\phi_0(u_1^3(\phi_4(i,x),i,x))=\phi_0(\phi_4(i,x))=\phi_0(i+x)=(i+x)'=i'+x=\mathrm{plus}(i',x)$ となり成り立つ．

注意：例 1 の中で $\phi_3(x,y,z)=\phi_0(\phi_2(x,y,z))$ としているが，$\phi_0=\mathrm{succ}, \phi_2=\mathrm{u}_1^3$ であるから，$\phi_3(x,y,z)=\phi_0(x)=x'$ である．なぜ，わざわざこのような関数を用意するかというと，それは原始帰納を適用する時に，ϕ_3 は 3 変数関数でなければならないので，1 変数関数である $\phi_0(x)$ を 3 変数関数と考える必要があるため射影関数 u_1^3 を用いて 3 変数関数にしているのである．このような変形はこれから何度も出てくるので注意して欲しい．例えば，2 変数関数 $g(y,u)$ を 4 変数関数 $f(x,y,z,u)$ と考えるには，u_2^4, u_4^4 を用いて，$f(x,y,z,u)=g(u^4(x,y,z,u), u_4^4(x,y,z,u))$ とすればよい．

例 1 を見てもわかるように，ϕ の prd は，原始帰納的関数 ϕ の関数値を計算するアルゴリズムの記述と考えることができる．$\mathrm{plus}(a,b)=a+b$ が原始帰納的であることは本質的には，plus が

$$\left\{\begin{array}{l}\mathrm{plus}(0,x)=x\\ \mathrm{plus}(k',x)=\mathrm{succ}(\mathrm{plus}(x,z))\end{array}\right. \qquad \left(\left\{\begin{array}{l}0+x=x\\ k'+x=(k+x)'\end{array}\right.\right)$$

をみたすことであり，このような表現から prd を作ることが可能である．

$k'+x=(k+x)'$ に $k=0$ を代入すると，$0'+x=(0+x)'=x'$ であるから，$x'=x+1$ が成り立つ．今後は，a' は $a+1$ で表現することにする．

例 2 $\mathrm{mult}(a,b)=ab$ は prf である．

$$\left\{\begin{array}{l}\mathrm{mult}(0,x)=\mathrm{zero}(x)\\ \mathrm{mult}(k+1,x)=h(\mathrm{mult}(k,x),k,x)\end{array}\right. \qquad \left(\left\{\begin{array}{l}0\times x=0\\ (k+1)\times x=(k\times x)+x\end{array}\right.\right)$$

ここで $h(x_1,x_2,x_3)=\mathrm{plus}(\mathrm{u}_1^3(x_1,x_2,x_3), \mathrm{u}_3^3(x_1,x_2,x_3))$ である．

定理 3.3 自然数 n と関数 $h(y,z)$ が与えられたとき，これから f を次のように帰納的に定義する．

$$\left\{\begin{array}{l}f(0)=n\\ f(k+1)=h(f(k),k)\end{array}\right.$$

$h(y,z)$ が prf ならば，$f(x)$ も prf である．

∴) この帰納的定義が原始帰納の一種であることを示せば良い．まず, $g_1(x)=\mathrm{succ}^n(\mathrm{zero}(x))$, $g_2(x,y,z)=h(u_1^3(x,y,z), u_2^3(x,y,z))$ とする．g_1, g_2 ともに prf である．次に 2 変数関数 $g(x,y)$ を原始帰納を用いて次のように定義する．

$$\left\{\begin{array}{l}g(0,y)=g_1(y)\\ g(k+1,y)=g_2(g(k,y),k,y)\end{array}\right.$$

g は prf である．さらに，$g_1(y) = \mathrm{succ}^n(\mathrm{zero}(y)) = n, g_2(g(k,y),k,y) = h(g(k,y),k)$ であるから，数学的帰納法により，y の値に関わらず，$f(x) = g(x,y)$ であることがわかる．よって，f も prf である．

例 3

$$pd(a) = \begin{cases} 0 & if \quad a = 0 \\ a-1 & if \quad a > 0 \end{cases}$$

は prf である．

$$\begin{cases} pd(0) = 0 \\ pd(a+1) = a \end{cases}$$

$pd(a)$ は以下の関数ブロックにより BYOB で計算可能である[7]．

定理 3.4

1 $h_1(x_1,\cdots,x_n), h_2(x_1,\cdots,x_n),\cdots,h_m(x_1,\cdots,x_n), g(y_1,\cdots,y_m)$ が prf のとき，合成

$$f(x_1,\cdots,x_n) = g(h_1(x_1,\cdots,x_n), h_2(x_1,\cdots,x_n),\cdots,h_m(x_1,\cdots,x_n))$$

により定義される $f(x_1,\cdots,x_n)$ も prf である．

[7]引き算は使えないこととスクリプト変数は宣言と同時に初期値として 0 が与えられることに注意する．

2 $g(x_1, \cdots, x_n)$ と $h(y, z, x_1, \cdots, x_n)$ が prf のとき，原始帰納

$$\begin{cases} f(0, x_1, \cdots, x_n) = g(x_1, \cdots, x_n) \\ f(k+1, x_1, \cdots, x_n) = h(f(k, x_1, \cdots, x_n), k, x_1, \cdots, x_n) \end{cases}$$

により定義される $f(k, x_1, \cdots, x_n)$ も prf である．

上記の定理より，すでに prf とわかっている関数から，合成や原始帰納で定義される関数は prf であるから，今後は特に prd を示さずに上記の定理を用いて，prf であることを示す．また，prf はすべて BYOB で計算可能であることが，初期関数が BYOB で計算可能であることと次の定理よりわかる．

定理 3.5

1 $h_1(x_1, \cdots, x_n), h_2(x_1, \cdots, x_n), \cdots, h_m(x_1, \cdots, x_n), g(y_1, \cdots, y_m)$ が BYOB で計算可能のとき，合成

$$f(x_1, \cdots, x_n) = g(h_1(x_1, \cdots, x_n), h_2(x_1, \cdots, x_n), \cdots, h_m(x_1, \cdots, x_n))$$

により定義される $f(x_1, \cdots, x_n)$ も BYOB で計算可能である．

2 $g(x_1, \cdots, x_n)$ と $h(y, z, x_1, \cdots, x_n)$ が BYOB で計算可能のとき，原始帰納

$$\begin{cases} f(0, x_1, \cdots, x_n) = g(x_1, \cdots, x_n) \\ f(k+1, x_1, \cdots, x_n) = h(f(k, x_1, \cdots, x_n), k, x_1, \cdots, x_n) \end{cases}$$

により定義される $f(k, x_1, \cdots, x_n)$ も BYOB で計算可能である．

$k > 0$ のとき，$f(k, x_1, \cdots, x_n) = h(f(pd(k), x_1, \cdots, x_n), pd(k), x_1, \cdots, x_n)$ であるから，次のように作ればよい．

例 **4** $\mathrm{power}(a,b) = b^a$ は prf である.

$$\begin{cases} \mathrm{power}(0,b) = \mathrm{succ}(\mathrm{zero}(b)) \\ \mathrm{power}(a+1,b) = \mathrm{mult}(\mathrm{power}(a,b),b) \end{cases}$$

$\mathrm{power}(a,b)$ は BYOB で計算可能である.

例 **5** $a!$ は prf である.

$$\begin{cases} 0! = 1 \\ (a+1)! = \mathrm{mult}(a!, \mathrm{succ}(a)) \end{cases}$$

例 **6**

$$a \dot{-} b = \begin{cases} a-b & if \quad a \geq b \\ 0 & if \quad a < b \end{cases}$$

は prf である.

$$\begin{cases} a \dot{-} 0 = a \\ a \dot{-} (b+1) = pd(a \dot{-} b) \end{cases}$$

例 7 $\max(a,b), \min(a,b)$ は prf である. $\min(a,b) = b \dot{-} (b \dot{-} a)$ $\max(a,b) = (a+b) \dot{-} \min(a,b)$

例 8

$$sg(a) = \begin{cases} 1 & if \quad a > 0 \\ 0 & if \quad a = 0 \end{cases}$$

$$\overline{sg}(a) = \begin{cases} 1 & if \quad a = 0 \\ 0 & if \quad a > 0 \end{cases}$$

は prf である.

例 9 $|a-b|$ は prf である. $|a-b| = (a \dot{-} b) + (b \dot{-} a)$

問題 3.6 例 5 から例 9 までの関数は BYOB で計算可能であることを示せ.

例 10 $\phi(\overline{x},0) + \phi(\overline{x},1) + \cdots + \phi(\overline{x},z-1)$ を $\sum_{y<z} \phi(\overline{x},y)$ と書き,

$\phi(\overline{x},0)\phi(\overline{x},1)\cdots\phi(\overline{x},z-1)$ を $\prod_{y<z} \phi(\overline{x},y)$ と書く.

但し, $\sum_{y<0} \phi(\overline{x},y) = 0, \prod_{y<0} \phi(\overline{x},y) = 1$ とする.

このとき, ϕ が prf ならば, $\xi(z,\overline{x}) = \sum_{y<z} \phi(\overline{x},y)$, $\eta(z,\overline{x}) = \prod_{y<z} \phi(\overline{x},y)$ は共に prf である.

$$\begin{cases} \xi(0,\overline{x}) = 0 \\ \xi(z+1,\overline{x}) = \xi(z,\overline{x}) + \phi(\overline{x},z) \end{cases}$$

$$\begin{cases} \eta(0,\overline{x}) = 1 \\ \eta(z+1,\overline{x}) = \eta(z,\overline{x})\phi(\overline{x},z) \end{cases}$$

自然数上の述語 $P(x_1,\cdots,x_n)$ に対し, 関数 $\varphi : \mathbb{N}^n \to \{0,1\}$ で,

$$\varphi(x_1,\cdots,x_n) = 1 \Leftrightarrow P(x_1,\cdots,x_n)$$

をみたす $\varphi(x_1,\cdots,x_n)$ を述語 $P(x_1,\cdots,x_n)$ の表現関数とよぶ．$P(x_1,\cdots,x_n)$ の表現関数を φ_P で表すことにする．述語 $P(x_1,\cdots,x_n)$ の表現関数が prf であるとき，P は原始帰納的述語 (primitive recursive predicate,prp) であるとよばれる．prf は計算値を計算するアルゴリズムを持つ関数と考えられるから，prp は真偽を決定するアルゴリズムを持つ述語である．

例 11 $a=b$ は prp である．$\varphi_=(a,b)=\overline{sg}(|a-b|)$

$\because)\ \overline{sg}(|a-b|)=1 \Leftrightarrow |a-b|=0 \Leftrightarrow (a\dot{-}b)+(b\dot{-}a)=0$

$\Leftrightarrow a\dot{-}b=0$ かつ $b\dot{-}a=0 \Leftrightarrow a\leqq b$ かつ $b\leqq a \Leftrightarrow a=b$

例 12 $a\leqq b$ は prp である．$\varphi_\leq(a,b)=\overline{sg}(a\dot{-}b)$

例 13 $a<b$ は prp である．$\varphi_<(a,b)=sg(b\dot{-}a)$

定理 3.7 $P(x_1,\cdots,x_n)$ が *prp* で $\xi_1(y_1,\cdots,y_m),\xi_2(y_1,\cdots,y_m),\cdots,\xi_n(y_1,\cdots,y_m)$ が *prf* のとき，

$$Q(y_1,\cdots,y_m)\equiv P(\xi_1(y_1,\cdots,y_m),\xi_2(y_1,\cdots,y_m),\cdots,\xi_n(y_1,\cdots,y_m))$$

は *prp* である．

$\because)\ \varphi_Q(y_1,\cdots,y_m)=\varphi_P(\xi_1(y_1,\cdots,y_m),\xi_2(y_1,\cdots,y_m),\cdots,\xi_n(y_1,\cdots,y_m))$

例 3.8 $x+y\leqq z$ は *prp* である．

定理 3.9 P,Q が *prp* のとき，$\neg P, P\wedge Q, P\vee Q, P\Longrightarrow Q$ も *prp* である．

$\because)\varphi_{\neg P}=\overline{sg}(\varphi_P),\varphi_{P\wedge Q}=\varphi_P\varphi_Q,\varphi_{P\vee Q}=sg(\varphi_P+\varphi_Q),\varphi_{P\Rightarrow Q}=\varphi_{(\neg P)\vee Q)}$

定理 3.10 $P(x_1,\cdots,x_n,y)$ が *prp* のとき，$(\forall y)_{y<z}P(x_1,\cdots,x_n,y)$, $(\forall y)_{y\leqq z}P(x_1,\cdots,x_n,y)$, $(\exists y)_{y<z}P(x_1,\cdots,x_n,y)$, $(\exists y)_{y\leqq z}P(x_1,\cdots,x_n,y)$ も *prp* である．

$\because)\ \varphi_{(\forall y)_{y<z}P(x_1,\cdots,x_n,y)}=\prod_{y<z}\varphi_P(\overline{x},y),\ \varphi_{(\exists y)_{y<z}P(x_1,\cdots,x_n,y)}=sg(\sum_{y<z}\varphi_P(\overline{x},y)),$

$\varphi_{(\forall y)_{y\leqq z}P(x_1,\cdots,x_n,y)}=\varphi_{(\forall y)_{y<z+1}P(x_1,\cdots,x_n,y)},\ \varphi_{(\exists y)_{y\leqq z}P(x_1,\cdots,x_n,y)}=\varphi_{(\exists y)_{y<z+1}P(x_1,\cdots,x_n,y)}$

例 3.11 $a|b$ (a は b を割り切る) は *prp* である．$(a|b\Leftrightarrow(\exists x)_{x\leqq b}(ax=b))$

問題 3.12 $Pr(a)$ (a は素数である) は *prp* であることを示せ．

有界 μ-作用素 (bounded μ-operator)

$$\mu y_{y<z} P(x_1, \cdots, x_n, y) = \begin{cases} \min\{y |\ P(x_1, \cdots, x_n, y)\} & if (\exists y)_{y<z} P(x_1, \cdots, x_n, y) \\ z & otherwise \end{cases}$$

定理 3.13 $P(x_1, \cdots, x_n, y)$ が prp のとき，$\mu y_{y<z} P(x_1, \cdots, x_n, y)$ は prf である．

例 3.14 P_i ($(i+1)$ 番目の素数をとる関数) は prf である．

問題 3.15 $f(x_1, x_2, z) = \mu y_{y<z}(x_1 | z \wedge x_2 | z)$ とする．$f(4, 6, 25), f(4, 6, 10)$ の値を求めよ．

3.3 アッカーマン関数

“値の計算できる関数”はすべて原始帰納的関数と考えることはできるのであろうか．答えは否である．

次のように定義される 2 変数の関数 $A(i, x)$ を考えよう．($A(i, x)$ はアッカーマン関数と呼ばれる．)

$$\begin{cases} A(0, x) = x + 1 \\ \begin{cases} A(i+1, 0) = A(i, 1) \\ A(i+1, x+1) = A(i, A(i+1, x)) \end{cases} \end{cases}$$

例えば $A(1, 2)$ を計算してみよう．

$A(1, 2) = A(0{+}1, 1{+}1) = A(0, A(0{+}1, 1)) = A(0, A(0{+}1, 0{+}1)) = A(0, A(0, A(0{+}1, 0))) = A(0, A(0, A(0, 1))) = A(0, A(0, 1 + 1)) = A(0, 2 + 1) = 3 + 1 = 4$

となる．

問題 3.16 $A(2, 3)$ を計算せよ．ただし，途中で $A(1, 2) = 4$ は用いて良いが，それ以外は定義に従って計算せよ．

他の場合も上の定義式に従って計算することができるから，このアッカーマン関数 $A(i, x)$ も“値の計算できる関数”と考えることができるが，この $A(i, x)$ は原始帰納的関数ではないことが証明できる [6]．

問題 3.17 アッカーマン関数を計算する BYOB 関数ブロックを作成せよ．

以上のことから“値の計算できる関数”というのは原始帰納的関数よりももっと大きい関数のグループであることになる．$A(i,x)$ は２重帰納法と呼ばれる原始帰納の拡張したもので定義されているので，原始帰納的関数を定義するときに原始帰納のかわりに２重帰納法を使うことにするとどうなるであろうか．それを例えば，２重帰納的関数とよぶことにする．すると，やはり３重帰納法で定義された関数で２重帰納的関数にはならないものが存在する．従って，原始帰納の部分を拡張していっても，“値の計算できる関数”というものをとらえることはできないことがわかる．すなわち，“値の計算できる関数”をとらえるためには合成や原始帰納のほかに別の関数を生み出す仕組みが必要になる．次にこのことを考えよう．

3.4 帰納的関数

$f(x,y)$ を値の計算できる関数とする．次のようにして１変数の関数 $g(x)$ を定義する．

$$g(x) = \text{“}f(x,y) = 0 \text{ をみたす最小の } y \text{ の値”}$$

この $g(x)$ は，x に対し $f(x,y) = 0$ をみたす y が存在しなければ値が定義できないので部分関数になるが，この関数がすべての自然数に対して定義されている場合は“値の計算できる関数”であることを調べてみよう．例えば $g(3)$ の値は次のように計算できる．

まず，$f(x,y)$ の x に３を代入し，y に０から順番に値を代入して $f(3,0), f(3,1), f(3,2)\cdots$ を計算していく．$f(x,y)$ は“値の計算できる関数”であるから，これは実行できる．このとき，もし $f(3,n) = 0$ となる n が存在するならば，$f(3,0), f(3,1), f(3,2)\cdots$ を計算していくとかならずどこかで値が０になるので，一番最初に値が０となる y が $g(x)$ の値になって計算が終了する．従って，上のように定義された関数 $g(x)$ も“値の計算できる関数”とみなすことができる．

原始帰納的関数を定義する初期関数と操作（合成，帰納法）にさらに以下の関数の定義の方法をつけ加える．

$n+1$ 変数の関数 $\psi(x_1,\cdots,x_n,y)$ が

$$\forall x_1 \forall x_2 \cdots \forall x_n \exists y[\psi(x_1,\cdots,x_n,y) = 1]$$

をみたすとき，このような ψ から次のように新しい関数 ϕ を作る．

$$\phi(x_1,\cdots,x_n) = \mu y[\psi(x_1,\cdots,x_n,y) = 1]$$

仮定より，$\phi(x_1, \cdots, x_n)$ は全域的である．

原始帰納的記述の定義と同様に，この方法も許して作られる関数列を帰納的記述と呼ぶ．すなわち，関数列 $\phi_0, \cdots, \phi_n$ が次の条件をみたすとき，この列を ϕ の帰納的記述（ recursive description) と呼ぶ．

1. ϕ_0は初期関数である．
2. $\phi_i\,(1 \leq i \leq n)$ は初期関数であるか，或いは$\phi_{j_1}, \cdots, \phi_{j_m}\,(j_k \leq i-1, k = 1, 2, \cdots, m)$ から合成，帰納法または μ-作用素を用いて定義される．（但し，μ-作用素を用いて定義するときは，適用の条件が満たされているとする．）
3. $\phi_n = \phi$

そして ϕ の帰納的記述が存在するとき ϕ を帰納的関数と呼ぶ．

上の操作は，合成や帰納法を適用することに比べると，かなり超越的な感じがある．条件

$$\forall x_1 \forall x_2 \cdots \forall x_n \exists y[\psi(x_1, \cdots, x_n, y) = 1]$$

をみたすか否かを一般に判定するアルゴリズムが存在するとは限らないからである．しかし，何らかの方法でこの条件が成立することがわかっていて，ψ が帰納的ならば，y の値を $0, 1, \cdots$ と動かして最初にψの値が 1 になるときの y の値を求めればよいから，確かに関数値を計算することができる操作になっている．

原始帰納的ではなかったアッカーマン関数も帰納的関数であることが証明できる．

命題 3.18 $A(i, x)$ は帰納的である．

“値の計算できる関数”と考えられる関数で帰納的関数でない例は見つかっていない．クリーニは“値を計算するアルゴリズムをもつ関数とは，帰納的関数のことである”と考えた．チューリングは“値を計算するアルゴリズムをもつ関数とは，チューリング機械で計算可能な関数のことである”と考えたが，実は両者は同値であることがわかった．

定理　チューリング機械で計算可能な関数は帰納的関数であり，帰納的関数はチューリング機械で計算可能である．

同様に

定理　帰納的関数は BYOB で計算可能な関数であり，BYOB で計算可能な関数は帰納的関数である．

チューリング機械で計算可能な関数や帰納的関数の他にいろいろな関数のグループが“値を計算するアルゴリズムをもつ関数”の候補としてあげられたが，それらはすべて同値であった．そこでチャーチは次のように仮説を提唱し，それは今ではチャーチ－チューリングの提唱とよばれ数学や計算機科学の分野で受け入れられている．

チャーチーチューリングの提唱
値を計算するアルゴリズムをもつ関数とは，帰納的関数のことである

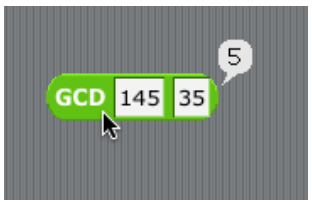
4　BYOB プログラムの正当性

プログラムを作成するときに気をつけることというと、まず思いつくのは間違いのないプログラムを作成することであろう．間違いのないプログラムというときには、何をもって間違いがないと判断するのであろうか．通常は、何か目的をもってプログラムを作り、プログラムを動作させて必要な情報を手にいれるのである．従って、プログラムを動かしたときに、止まらなかったり[8]或いは止まっても目的と違う結果が出てきたのでは困り、そのようなプログラムは“正しい”プログラムとは呼べない．プログラムが正しいとか間違っているとか言うときには、このように前もってプログラムに求めるものが決まっている．それをまとめたものを仕様 (specification) と呼ぶ．言い替えるとプログラムの正しさとは仕様に対して決まるのである[9]．

『プログラムの正しさは仕様に対して決まる』とはどのようなことか次の BYOB のプログラムを例にとって考えよう．

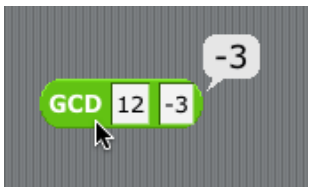
このプログラムは、正整数 x と y の最大公約数を求める関数ブロックである．変数 x と y の値を与えてブロックをクリックすると、変数 a の値が報告されそれが x と y の最大公約数になる．

[8] ここでは止まらないことが求められるプログラムは除外している．
[9] この章の内容は [7] を参考にしている．



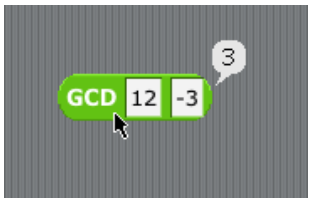
問題 4.1 入力 $x = 145, y = 35$ に対し出力は 5 であることを変数表を書いて確かめよ.

しかし、これで本当に最大公約数が求められているのであろうか. 適当な値を x と y に与えると、確かに a の値は最大公約数になっているが“本当に”任意の値に対して最大公約数を求めているのか. 変数 x と y に負の整数を与えたとき、上のプログラムはどういう結果を出力するだろうか.



上のプログラムは整数 x と y の最大公約数を求めるプログラムとすると間違いである. 少なくとも一方が 0 ではない整数 x と y の最大公約数を求める正しいプログラムは

としなければならない．



最初のプログラムは、正整数に対して最大公約数を求めるという仕様に対しては正しいが、すべての整数に対して最大公約数を求めるという仕様に対しては正しくないのである．

BYOB プログラムで関数ブロックを入れて議論すると複雑になるので、この授業では次のような BYOB のプログラムに対しての“正しさ”について考察する．

1. 式とはレポーター型のブロックで次のように再帰的に定義する．

2. ブール式は述語型のブロックで次のように再帰的に定義する．

3. プログラムはコマンド型のブロックで次のように再帰的に定義する．それぞれ代入文，複合文，条件文，until 文とよぶ．

定数や変数の定義がないが，任意の自然数を定数として扱え，変数は整数を値にもつものでBYOBで定義できるものとする．特に変数の値が整数であることに注意が必要な場合を除き，今後変数の値が整数であることは特に言及しない。

以上のBYOBプログラムの構成的な定義に従うと、BYOBプログラムが“正しい”ということを証明方法として，構造に関する帰納法を適用することが考えられる．しかし、上で述べたようにプログラムが正しいとか正しくないというのは、仕様抜きでは考えられない．

そこでホーア論理とよばれるプログラムの正しさを検証するための論理学的方法が考えられた．ホーア論理では表明つきプログラムというものを使って、プログラムが仕様を満たすということを表現する．

A, B を何らかの条件を表わす（論理）式、P をプログラムとする．

$$\langle A\rangle\ P\ \langle B\rangle \quad \{A\}\ P\ \{B\}$$

のように表現されたものを表明つきプログラム、A を事前表明、B を事後表明とよぶ．事前表明事後表明をあわせて単に表明とよぶ．P の前後に表明をつけることで P に対する仕様を表現しているのである．仕様のわからないプログラムに対しては“正しい”とか“正しくない”ということは何も言えなかったが、表明つきプログラムに対しては定義可能である．$\langle A\rangle\ P\ \langle B\rangle$ 、 $\{A\}\ P\ \{B\}$ が正しいということをそれぞれ次のように定義する．

$\langle A\rangle\ P\ \langle B\rangle$ が正しいとは、A が成り立っているときにプログラム P を実行すると、その実行は停止し、さらに停止後 B が成り立つときをいう．

$\{A\}\ P\ \{B\}$ が正しいとは、A が成り立っているときにプログラム P を実行するとき、もし実行が停止するならば停止後 B が成り立つときをいう．（従って、P の実行が停止しない場合は$\{A\}\ P\ \{B\}$ は正しいと考える．）

前者にはプログラムの停止性が含まれているが､後者には含まれていない．$\langle A\rangle\ P\ \langle B\rangle$ を完全正当性、$\{A\}\ P\ \{B\}$ を部分正当性とよぶ．

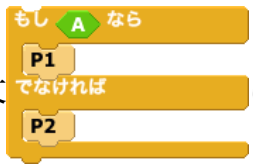
以上のように表明付きBYOBプログラム $\{A\}P\{B\}$ を定義することにより，それらが正しいことをプログラムの構成に関する帰納法で証明することが可能になる．具体的には、代入文の公理から複合文、条件文、until文の推論規則及び帰結の推論規則を用いて証明をする．（BYOBプログラムのホーア論理の健全性：証明可能なものは正しい）

BYOBのプログラムをそのまま記載すると見にくいため，今後次のようにBYOBのプログラムを略記することにする．

代入文 

は $x := t$，　複合文 

は $\mathrm{P}_1;\mathrm{P}_2$，

条件文 

は **if** A **then** P_1 **else** P_2 **fi**,

until 文 

は **until** A **do** P_1 **od**

で表すことにし，場合により BYOB プログラムを記載することにする．
また，

は $A \wedge B$, 

は $A \vee B$, 

は $\neg A$

で表し，表明の表現にも同じ論理記号 $\wedge, \vee, \neg$ を用いることにする．

問題 **4.2** 次の BYOB プログラムを上で述べた略記法により記述せよ。

5　代入文の公理と帰結の規則

例 **5.1** $\{y > 2\}\ x := y + 1\ \{x > 3\}$は正しい．

代入文 $x := y+1$ の実行前に変数 y の値が2より大きいとすると，代入文 $x := y+1$ の実行により変数 x の値は $y+1$ の値になるから，実行後の x の値は $2+1=3$ より大きくなる．従って，この表明付きプログラム$\{y > 2\}\ x := y+1\ \{x > 3\}$は正しい．

例 **5.2** $\{x > 5\}\ \ x := x+1\ \ \{x > 6\}$は正しい．

代入文 $x := x+1$ の実行前に変数 x の値が5より大きいとすると，代入文 $x := x+1$ の実行により変数 x の値は1増加するから，実行後の x の値は6より大きくなる．従って，この表明付きプログラム$\{x > 5\}\ \ x := x+1\ \ \{x > 6\}$は正しい．

$R[x \to t]$ を表明 $R(x)$ の中にある変数 x を式 t ですべて置き換えて得られる表明とする．例えば, $R(x) \equiv [x > x^2+6]$ で $t \equiv x+1$ のとき, $R[x \to t] \equiv [x+1 > (x+1)^2+6]$ である．このような表記法の下で，$\{R[x \to t]\}\ \ x := t\ \ \{R(x)\}$ は常に正しい表明つきプログラムである．この理由を考えてみる．代入文 $x := t$ の実行前に $R[x \to t]$ が成り立つとする．すなわち, 変数 x の代わりに t を考えているので, 表明 $R(x)$ 中の x の値を t の値にすると成り立つことになる．$x := t$ を実行すると，変数 x の値は t の値になるので, 実行後に $R(x)$ が成り立つことになる．従って, $\{R[x \to t]\}\ \ x := t\ \ \{R(x)\}$ は常に正しい表明付きプログラムとなる。$\{R[x \to t]\}\ \ x := t\ \ \{R(x)\}$ の形の表明付きプログラムを代入文の公理とよぶことにする..

注意 $\{R(x)\}\ \ x := t\ \ \{R[x \to t]\}$ は正しい表明とはいえない．

例 **5.3** $\{x < 5\}\ \ x := x+1\ \ \{x+1 < 5\}$ は正しくない．(x の値が4のときを考える.)

代入文の公理は，代入文 $x := t$ に関し，任意の事後表明 R に対しそれに適切な事前表明として $R[x \to t]$ がとれることを示している．もちろん，部分正当性の定義から $A \to R[x \to t]$ が正しい表明ならば，$\{A\}\ \ x := t\ \ \{R\}$ も正しい表明つきプログラムである．

例 **5.4**

- $\{x+1 > 1\}\ \ x := x+1\ \ \{x > 1\}$
- $\{x^2 > x^2 y\}\ \ x := x^2\ \ \{x > xy\}$

問題 **5.5** 次の代入文と事後表明に対し，代入文の公理となるように事前表明を定めよ．(式を計算せずに形式的に求める.)

	代入文	事後表明
(a)	$x := x + 7$	$x + y > 20$
(b)	$x := x - 1$	$x^2 + 2x = 3$
(c)	$x := x - 1$	$(x + 1)(x - 1) = 0$
(d)	$y := x + y$	$y = x$
(e)	$y := x + y$	$y = x + y$

一般に$\{A\}\ P\ \{B\}$ が正しい表明つきプログラムで$C \to A, B \to D$ がともに正しい表明ならば，$\{C\}\ P\ \{D\}$ は正しい表明つきプログラムである．このことを次のような図式で表し，帰結の推論規則とよぶ．

帰結の推論規則

$$\frac{\{A\}\ P\ \{B\}}{\{C\}\ P\ \{D\}} \qquad \begin{array}{l}(\text{ただし } C \to A, B \to D \text{ がと}\\ \text{もに真のとき})\end{array}$$

この図式は，$C \to A$,$B \to D$ がともに真のときに，$\{A\}\ P\ \{B\}$ が正しければ，$\{C\}\ P\ \{D\}$ も正しいことを表す．今後，このようにいくつかの表明付きプログラムから別の表明付きプログラムを導く推論規則を定義するが，そこに現れる図式中の横棒は$\therefore$と考えれば良い．また帰結の推論規則を用いるとき，$C \to A$,$B \to D$ が自明ではないときは，それが成り立つ理由を推論規則の横に記載することにする．

例 **5.6** $\{x^2 > 2 \wedge x \in \mathbb{N}\}\ x := x - 1\ \{x > 0\}$は正しい表明付きプログラムであることを，代入文の公理と帰結の推論規則を用いて示せ．

$$\frac{\{x - 1 > 0\}\ x := x - 1\ \{x > 0\}}{\{x^2 > 2 \wedge x \in \mathbb{N}\}\ x := x - 1\ \{x > 0\}} \qquad \begin{array}{l}(\because x^2 > 2 \text{ より，} x \neq 0, 1. \text{ よって，} x \in \mathbb{N} \text{ よ}\\ \text{り } x > 1 \text{ であるから，} x - 1 > 0 \text{ である．})\end{array}$$

上式の$\{x - 1 > 0\}\ x := x - 1\ \{x > 0\}$は代入文の公理であるから正しい表明つきプログラムで，下式の$\{x^2 > 2 \wedge x \in \mathbb{N}\}\ x := x - 1\ \{x > 0\}$は帰結の規則により導かれている．よって，$\{x^2 > 2 \wedge x \in \mathbb{N}\}\ x := x - 1\ \{x > 0\}$も正しい表明付きプログラムである．

問題 **5.7** 問題5.5で求めた事前表明を簡単な事前表明に書き換え，それを事前表明にもつ表明付きプログラムの正しさを代入文の公理と帰結の推論規則を用いて示せ．

6　複合文、条件文、until 文の部分正当性

6.1　複合文の部分正当性

次のような表明付きの複合文を考える．

$$\{a = 2 \wedge b = 3\}\ \ c := a;\ a := b;\ b := c\ \ \{a = 3 \wedge b = 2\}$$

複合文の実行は前から逐次的に行われるからこれは正しい表明付きプログラムである．我々は表明つきプログラムの正しさをその部分プログラムの正しさから導き出そうと考えている．したがって複合文においても、その部分プログラムの正しさから全体の正しさを導けるように考えたい．上記の例でそのことをみていこう．上の表明付きプログラムでそれぞれの代入文の実行前後で正しくなる表明を考えてみよう．まず、最初の代入文 $c := a$ では変数 c に変数 a の値が代入されるから、

$$\{a = 2 \wedge b = 3\}\ c := a\ \{c = 2 \wedge b = 3\}$$

は正しい表明つきプログラムである．さらに、これは代入文の公理の形をしていることに注意をしよう．同じように、

$$\{c = 2 \wedge b = 3\}\ a := b\ \{c = 2 \wedge a = 3\},\ \{c = 2 \wedge a = 3\}\ b := c\ \{b = 2 \wedge a = 3\}$$

も正しい表明つきプログラムである．最後の表明つきプログラムにおいて $b = 2 \wedge a = 3 \rightarrow a = 3 \wedge b = 2$ が常に成り立つから、帰結の推論規則より事後表明を $a = 3 \wedge b = 2$ に変更してできる表明つきプログラム

$$\{c = 2 \wedge a = 3\}\ b := c\ \{a = 3 \wedge b = 2\}$$

も正しい表明つきプログラムである．これを逆に考えるとまず代入文の公理と帰結の推論規則から、

$$\begin{array}{l}\{a = 2 \wedge b = 3\}\ c := a\ \{c = 2 \wedge b = 3\}\\ \{c = 2 \wedge b = 3\}\ a := b\ \{c = 2 \wedge a = 3\}\\ \{c = 2 \wedge a = 3\}\ b := c\ \{a = 3 \wedge b = 2\}\end{array}$$

が正しいことがわかり、このことから最初の表明つきプログラム

$$\{a = 2 \wedge b = 3\}\ \ c := a;\ a := b;\ b := c\ \ \{a = 3 \wedge b = 2\}$$

の正しさが判断できるようになる．このように表明付きの複合文

$$\{A\}\ \ P_1; P_2; \cdots; P_n\ \ \{B\}$$

が与えられたときにその正しさを証明するには、新たに $n-1$ 個の表明 $S_1, S_2, \cdots S_{n-1}$ を適切に選んで

$$\{A\}\ P_1\ \{S_1\},\{S_1\}\ P_2\ \{S_2\},\cdots\ ,\{S_{n-1}\}\ P_n\ \{B\}$$

という n 個の表明つきプログラムの正しさを示すことができれば $\{A\}\ \ P_1; P_2; \cdots ; P_n\ \ \{B\}$ の正しさを証明できることになる．このことをつぎのような推論規則で表すことにする．

$$\frac{\{A\}\ P_1\ \{S_1\}\ \{S_1\}\ P_2\ \{S_2\}\cdots\{S_{n-1}\}\ P_n\ \{B\}}{\{A\}\ \ P_1; P_2; \cdots ; P_n\ \ \{B\}}$$

例題 6.1 $\{A\}\ \ x := x+1; y := x-y\ \ \{x < y\}$ が表明付きプログラムとして正しくなるように事前表明 A を定めよ．

証明：代入文の公理より，$\{x < x-y\}\ y := x-y\ \{x < y\}$ は正しい表明付きプログラムで，同様に，$\{x+1 < x+1-y\}\ x := x+1\ \{x < x-y\}$ も正しい表明付きプログラムである．従って，A として $x{+}1 < x{+}1{-}y$ をとると，$\{A\}\ \ x := x+1; y := x-y\ \ \{x < y\}$ は正しい表明付きプログラムとなる．なお帰結の推論規則を用いて，A として，$y < 0$ をとってもよい．（こちらの方が適切ではある．） □

問題 6.2 次の複合文と事後表明に対し，表明付きプログラムとして正しくなるように事前表明を定めよ．

	複合文	事後表明
(a)	$x := x-1; x := x^2$	$x > 0$
(b)	$x := x-y; y := x+y$	$x = 1 \wedge y = 2$
(c)	$x := x^2; y := x+y$	$x = 1 \wedge y = 2$

例題 6.3 $\{(x+y)^2 > x\}\ \ x := x+y; y := x-y; x := x^2\ \ \{x > y\}$ は正しい表明つきプログラムであることを代入文の公理、帰結の推論規則及び複合文の推論規則を用いて示せ．

証明：$A \equiv (x+y)^2 > x, S_1 \equiv x^2 > x-y, S_2 \equiv x^2 > y, B \equiv x > y, C \equiv (x+y)^2 > (x+y)-y$ とおく．$C \equiv S_1[x+y/x], S_1 \equiv S_2[x-y/y], S_2 \equiv B[x^2/x]$ である．よって、

$$\frac{\dfrac{\{C\}\ x := x+y\ \{S_1\}}{\{A\}\ x := x+y\ \{S_1\}}\ \{S_1\}\ y := x-y\ \{S_2\}\ \{S_2\}\ x := x^2\ \{B\}}{\{A\}\ \ x := x+y; y := x-y; x := x^2\ \ \{B\}}$$

であるから与えられた表明つきプログラムは正しい[10]． □

[10] このような図式を証明図 (proof figure) とよぶ．

上の証明では代入文の公理に形をあわせるために帰結の推論規則が用いられたことに注意しよう.

問題 6.4 $\{xy+1>0\}\ \ x := xy; y := x+1; x := x+1\ \ \{x+y>0\}$ は正しい表明つきプログラムであることを代入文の公理、帰結の推論規則及び複合文の推論規則を用いて示せ.

6.2 条件文の部分正当性

次の表明付きの条件文を考える.

$$\{A\}\ \textbf{if}\ \alpha\ \textbf{then}\ P_1\ \textbf{else}\ P_2\ \textbf{fi}\ \{B\}$$

この条件文の実行はブール式αが真であるか偽であるかにより決まる. αが真のときはP_1が実行されαが偽のときはP_2が実行される. 従って、上の表明つきプログラムが正しいものであるためには、Aが成り立っているとき、P_1, P_2のいずれの実行においてもBが成り立つことが必要である. すなわち、$\{A\wedge\alpha\}\ P_1\ \{B\}$,$\{A\wedge\neg\alpha\}\ P_2\ \{B\}$が正しい表明つきプログラムのとき、$\{A\}$ **if** α **then** P_1 **else** P_2 **fi** $\{B\}$ が正しいと判断できる. このことを次のような推論規則を用いて表す.

$$\frac{\{A\wedge\alpha\}\ P_1\ \{B\}\quad \{A\wedge\neg\alpha\}\ P_2\ \{B\}}{\{A\}\ \textbf{if}\ \alpha\ \textbf{then}\ P_1\ \textbf{else}\ P_2\ \textbf{fi}\ \{B\}}$$

例題 6.5 $\{x=a\wedge y=b\}$ **if** $x\geq y$ **then** $z := x$ **else** $z := y$ **fi** $\{z=\max\{a,b\}\}$ は正しい表明つきプログラムであることを代入文の公理、帰結の推論規則及び条件文の推論規則を用いて示せ.

証明：$z=\max\{a,b\}$ を A で表す.

$$\frac{\dfrac{\{A[z\to x]\}\ z := x\ \{A\}}{\{x=a\wedge y=b\wedge x\geq y\}\ z := x\ \{A\}}\quad \dfrac{\{A[z\to y]\}\ z := y\ \{A\}}{\{x=a\wedge y=b\wedge\neg(x\geq y)\}\ z := y\ \{A\}}}{\{x=a\wedge y=b\}\ \textbf{if}\ x\geq y\ \textbf{then}\ z := x\ \textbf{else}\ z := y\ \textbf{fi}\ \{A\}}$$

$x=a\wedge y=b\wedge x\geq y$ ならば $\max\{a,b\}=a=x$ であるから，$A[x/z]$ が成り立つ．また，$x=a\wedge y=b\wedge\neg(x\geq y)$ ならば $\max\{a,b\}=b=y$ であるから，$A[y/z]$ が成り立つ．よって，最初の二つの推論は帰結の推論規則の条件をみたす．従って，与えられた表明つきプログラムは公理と推論規則で導かれるので，正しい表明つきプログラムである. □

問題 **6.6** 次の表明つきプログラム正しい表明つきプログラムであることを推論規則を用いて示せ.

1. $\{x > 5\}$ **if** $x \leq y$ **then** $x := x$ **else** $c := x; x := y; y := c$ **fi** $\{x \leq y\}$
2. $\{x = a\}$ **if** $x < 0$ **then** $x := -x$ **else** $x := x$ **fi** $\{x = |a|\}$
3. $\{y$ は自然数 $\wedge\, y > 0 \wedge z \cdot x^y = a\}$
   if $odd(y)$ **then** $z := z \cdot x; y := y - 1$ **else** $x := x^2; y := y/2$ **fi**
   $\{y$ は自然数 $\wedge\, y \geq 0 \wedge z \cdot x^y = a\}$
   ただし、$odd(y)$ は y が奇数であるかどうか判定する述語である.

6.3 BYOB プログラムと帰納的関数

前節の最後の問題で条件文に $odd(y)$ という述語を用いたが, このような述語を用いることは BYOB プログラムの範囲を逸脱しないのだろうか. 条件文に用いられるブール式に用いることのできる論理記号は $\wedge, \vee, \rightarrow, \neg$ のみで, $odd(y) \equiv (\exists x)_{x<y}[y = 2x+1]$ であるから[11], このままでは我々の定義した BYOB プログラムには使用できないように思える. しかし, そうではなく実は任意の帰納的関数 $f(x_1, \cdots, x_n)$ を用いて代入 $y := f(x_1, \cdots, x_n)$ を行ったり, 任意の帰納的述語 $R(x_1, \cdots, x_n)$ を条件文のブール式として使用しても我々の BYOB プログラムの範疇からは逸脱しないことが示される. それは以下の定理によって保証される.

定理 **6.7** 任意の帰納的関数は BYOB プログラムにより計算できる.

この定理より $f(x_1, \cdots, x_n)$ が帰納的関数のとき$\{true\}\ P\ \{y = f(x_1, \cdots, x_n)\}$ を正しくする BYOB プログラム P が存在するので, プログラム中で $z := f(x_1, \cdots, x_n)$ となっているところを $P; z := y$ でおきかえると, もともとの BYOB プログラムの文法で書かれたプログラムになる. また, 条件文で帰納的述語 $R(x_1, \cdots, x_n)$ を用いて, **if** R **then** P_1 **else** P_2 **fi** と使われている場合は, $R(x_1, \cdots, x_n) \Leftrightarrow f(x_1, \cdots, x_n) = 1$ となる帰納的関数 f が存在するので, この f を計算する BYOB プログラム P を用いて, (f の値が変数 y に入っているとすると), P; **if** $y = 1$ **then** P_1 **else** P_2 **fi** とするとやはりこれも, もともとの BYOB プログラムの文法で書かれたプログラムになる. 以上のことから, 代入文や条件文, until 文に帰納的な関数や述語を用いてもそれに対応する BYOB プログラムが存在するので, 自由に使用してよいということになる.

[11] 従って, $odd(y)$ は原始帰納的述語である.

6.4 until 文の部分正当性

次の表明つきプログラムを考える.

$$\{a = 0 \wedge b = 3\}\ \textbf{until}\ b = 0\ \textbf{do}\ a := a + 1; b := b - 1\ \textbf{od}\ \{a = 3 \wedge b = 0\}$$

$a = 0, b = 3$ で上の **until** 文を実行すると、until 文のループが回る間に変数 a, b は

ループ回数	0	1	2	3
a	0	1	2	3
b	3	2	1	0

となって確かに $a = 3, b = 0$ で止まるから、上の表明つきプログラムは（完全正当性の意味でも）正しい.

我々は表明つきプログラムの正しさをその部分プログラムの正しさから導き出そうと考えている. したがって until 文においても、until 文中の **do** $\cdots$ **od** の間にあるプログラムの正しさから、全体の正しさを導くような規則が欲しい. つまり

$$\frac{\{A\}\ P\ \{B\}}{\{A'\}\ \textbf{until}\ \alpha\ \textbf{do}\ P\ \textbf{od}\ \{B'\}}$$

となるような推論規則がほしい.

従って、until 文のループの実行中にループの回る回数に無関係の条件があって、それが常に成り立っている状況が必要である.

このとき、A, B に求められるものは何であろうか. まず until 文で P が実行されるのは α が偽のときのみであるから、A が成り立つときには α も偽であるような条件でなければならない. 従って A は“$\neg\alpha \wedge A_0$”のような表明でなければならない (あるいは意味がない).

またループが 1 回実行されたとき、次にブール式 α の値を評価して偽になったときには、再び P を実行したいのであるから、B から A が導けなければならない. すなわち、少なくとも

$$\{\neg\alpha \wedge A_0\}\ P\ \{A_0\}$$

が正しいことが求められる.

このような A_0 はループの実行中に常に成立しているものなので、ループ不変表明と呼ばれる.

それでは上の例でこのような A_0 に対応するものは何であろうか.

ループ回数	0	1	2	3
a	0	1	2	3
b	3	2	1	0

ループの実行中に常に成立している関係を考えると $a+b=3 \wedge 0 \leq a \leq 3 \wedge 0 \leq b \leq 3$ である．a, b は整数であると仮定すると、$a+b=3 \wedge 0 \leq a \leq 3 \wedge 0 \leq b \leq 3 \Leftrightarrow a+b=3 \wedge b \geq 0$ である．このとき、

$$\{b>0 \wedge (a+b=3 \wedge b \geq 0)\}\ a:=a+1; b:=b-1\{a+b=3 \wedge b \geq 0\}$$

は正しい．従って $a+b=3 \wedge b \geq 0$ はこの until 文のループ不変表明である[12]．それでは、until 文の推論規則の上式を

$$\{\neg\alpha \wedge A_0\}\ P\ \{A_0\}$$

にしたとき、それからどのような表明つきの until 文が正しいといえるか．until 文を実行する前に A_0 が偽の場合は、部分正当性の意味から A_0 を事前条件に持つ表明つきプログラムはすべて正しい．A_0 が真のとき、もし α の値が真ならば until 文はなにもせずに終了するから、事後条件は事前条件から論理的に導かれるものならばよい．従って A_0 が真で、α が偽であるときが問題で、このときは上の式から α の値が偽である限りループが実行され、ループの停止後は α が真になる（このとき A_0 は真のままである）．そこで until 文の推論規則として、

$$\frac{\{\neg\alpha \wedge A_0\}\ P\ \{A_0\}}{\{A_0\}\ \textbf{until}\ \alpha\ \textbf{do}\ P\ \textbf{od}\ \{\alpha \wedge A_0\}}$$

を考えると確かに上式が正しければ、下式も正しくなる．$A \equiv a, b \in \mathbb{Z} \wedge a+b=3 \wedge b \geq 0$ として、上の例に当てはめると

$$\frac{\{b \neq 0 \wedge A\}\ a:=a+1; b:=b-1\ \{A\}}{\{A\}\ \textbf{until}\ b=0\ \textbf{do}\ a:=a+1; b:=b-1\ \textbf{od}\ \{b=0 \wedge A\}}$$

となる．従ってこの推論規則を使って最初の表明つきプログラムの正しさを示すに

[12] $a+b=3$ だけでもループ不変表明であるが、後で見るように全体での推論を考えると $a+b=3 \wedge b \geq 0$ にする必要がある．

は次のような証明図を作成すればよい．

$$\dfrac{\dfrac{\dfrac{\dfrac{\left\{\begin{matrix}(a+1)+(b-1)=3\\ b-1\geq 0\end{matrix}\right\}\ a:=a+1\ \left\{\begin{matrix}a+(b-1)=3\\ b-1\geq 0\end{matrix}\right\}}{\left\{\begin{matrix}b\neq 0\\ A\end{matrix}\right\}\ a:=a+1\ \left\{\begin{matrix}a+(b-1)=3\\ b-1\geq 0\end{matrix}\right\}\qquad \left\{\begin{matrix}a+(b-1)=3\\ b-1\geq 0\end{matrix}\right\}\ b:=b-1\ \{A\}}}{\{b\neq 0\wedge A\}\ a:=a+1;b:=b-1\ \{A\}}}{\{A\}\ \textbf{until}\ b=0\ \textbf{do}\ a:=a+1;b:=b-1\ \textbf{od}\ \{b=0\wedge A\}}}{\left\{\begin{matrix}a=0\\ b=3\end{matrix}\right\}\ \textbf{until}\ b=0\ \textbf{do}\ a:=a+1;b:=b-1\ \textbf{od}\ \left\{\begin{matrix}a=3\\ b=0\end{matrix}\right\}}$$

ここで，最上式の二つの表明つきプログラムはどちらも代入文の公理をみたし，帰結の規則を使用している 2 箇所は適用条件をみたしている．例えば，
$b\neq 0\wedge A\rightarrow\ (a+1)+(b-1)=3\wedge b-1\geq 0$ は次のように示される．
$b\neq 0, a+b=3, b\geq 0$ とする．$a+b=(a+1)+(b-1)$ より，$(a+1)+(b-1)=3$ である．b は整数で，$b\neq 0, b\geq 0$ であるから，$b-1\geq 0$ である．

問題 6.8

1. 次の表明つき BYOB プログラムは与えられた整数 n と x の積を求めるプログラムである．

   $\{i, n\in\mathbb{Z}\wedge i=0\wedge p=0\wedge n\geq 0\}\ \textbf{until}\ i=n\ \textbf{do}\ i:=i+1; p:=p+x\ \textbf{od}\ \ \{p=n\cdot x\}$

   (a) $n=5$ の場合に変数 i と p がどのように変化していくか次の変数表に値を書いて正しい答えが得られることを確かめよ．また，$p=i\cdot x\wedge i\leqq n$ がループ不変表明になることを確かめよ．($\top$ は真であることを表す．)

ループ回数	0	1	2	3	4	5
i	0					
p	0					
$p=i\cdot x$	$\top$					
$i\leqq n$	$\top$					

   (b) この表明付きプログラムの正しさを推論規則を用いて示せ．

2. 次の表明つき BYOB プログラムは与えられた数 a と b に対し，a を b で割ったときの商 $[a/b]$ と余り $a\ \textbf{mod}\ b$ を求めるプログラムである．上の問題を参考にして適切なループ不変表明を定め，この表明付きプログラムの正しさを推論規則を用いて示せ．

$\{a \geq 0 \wedge b > 0 \wedge q = 0 \wedge r = a\}$
until $r < b$ **do** $q := q + 1; r := r - b$ **od**
$\{q = [a/b] \wedge r = a \ \mathbf{mod}\ b\}$

6.5 帰結規則の使い方

ここまで複合文、条件文、until 文の推論規則を説明していくつか例をみてきた．その例をみるとわかるが表明つきプログラムが与えられたときそれを部分に分割して推論規則を適用していくステップは帰結の推論規則を除くと唯一に決まっている．すなわち、複合文には複合文の推論規則、条件文には条件文の推論規則そして until 文には until 文の推論規則が適用される．ただし、適用できるといっても until 文の推論規則を適用するためには、事前表明と事後表明の形に制限があるので、一般にはそのままでは適用できない．また、代入文の公理にも事前表明と事後表明の形に制限がある．それらを調整するために帰結の推論規則が使用される．

6.6 公理と推論規則のまとめ

ここまで考察してきた公理と推論規則をまとめると次のようになる．

代入文の公理

$$\{R[x \to t]\} \ \ x := t \ \ \{R\}$$

帰結の推論規則

$$\frac{\{A\}\ P\ \{B\}}{\{C\}\ P\ \{D\}} \qquad \text{(ただし } C \to A, B \to D \text{ がともに真のとき)}$$

複合文の推論規則

$$\frac{\{A\}\ P_1\ \{S_1\}\ \{S_1\}\ P_2\ \{S_2\} \cdots \{S_{n-1}\}\ P_n\ \{B\}}{\{A\} \ \ P_1; P_2; \cdots; P_n \ \ \{B\}}$$

条件文の推論規則

$$\frac{\{A \wedge \alpha\}\ P_1\ \{B\} \quad \{A \wedge \neg\alpha\}\ P_2\ \{B\}}{\{A\}\ \mathbf{if}\ \alpha\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2\ \mathbf{fi}\ \{B\}}$$

until 文の推論規則

$$\frac{\{\neg\alpha \wedge A\}\ P\ \{A\}}{\{A\}\ \mathbf{until}\ \alpha\ \mathbf{do}\ P\ \mathbf{od}\ \{\alpha \wedge A\}}$$

7　完全正当性

前章において部分正当性の意味で正しい表明つきプログラム導くための公理と推論規則について考察した．ここでは完全正当性について考えてみよう．部分正当性と完全正当性に違いはプログラムの停止性の保証があるかどうかである．部分正当性では停止性を保証する必要がなかったが，完全正当性ではプログラムの正しさばかりでなく停止性についても考慮しないといけない．代入文は必ず停止する．また前章で考察した部分正当性に関する推論規則のなかで until 文の推論規則を除くと，推論規則の上式が停止するプログラムならば下式は必ず停止することがわかる．一方 until 文の推論規則では，上式が停止するプログラムであっても下式が停止するとは限らない．例えば部分正当性の until 文に関する推論規則をそのまま適用して次のような完全正当性に関する推論を考えてみよう．

$$\frac{\langle 1 = 1 \wedge x > a \rangle\ x := x + 1\ \langle x > a \rangle}{\langle x > a \rangle\ \mathbf{until}\ 1 \neq 1\ \mathbf{do}\ x := x + 1\ \mathbf{od}\ \langle 1 = 1 \wedge x > a \rangle}$$

明らかに上式は停止するが，下式は停止しない．

従って、部分正当性に関する until 文の推論規則を完全正当性に移し替えた

$$\frac{\langle \neg\alpha \wedge A \rangle\ P\ \langle A \rangle}{\langle A \rangle\ \mathbf{until}\ \alpha\ \mathbf{do}\ P\ \mathbf{od}\ \langle \alpha \wedge A \rangle}$$

は上式の条件だけでは下式の停止性を保証できないため誤った推論であることになる．それではこの下式が（完全正当性の意味で）正しい表明つきプログラムであるためには上式の表明がどのような条件をみたしていればよいであろうか．ここで以前にとりあげた例をもう一度みてみよう．

$$\frac{\{b \neq 0 \wedge A\}\ \mathbf{begin}\ a := a + 1; b := b - 1\ \mathbf{end}\ \ \{A\}}{\{A\}\ \mathbf{until}\ b = 0\ \mathbf{do}\ a := a + 1; b := b - 1\ \mathbf{od}\ \{b = 0 \wedge A\}}$$

この例では下式は b の値がどんな値であっても停止することがわかる．それは until 文の **do** 以下を実行するごとに b の値が 1 減るためにループを b 回くり返すと b の値が 0 になることがわかるからである．ここでは b がカウンターの役目をしていてループが回るごとに b の値がへり、やがて b の値が 0 になった時点で実行が終了するのである．

従って、一般の場合にもこの b に相当するカウンターがあってループが回るたびに減少してやがて 0 になるそのようなものがとれれば、停止性を示すことができる．そこで整数を値にもつ式 T で次の性質をみたすものを考える．

1. T はループの繰り返しで減少する．すなわち、$\langle \neg\alpha \wedge A \wedge v = T \rangle\ P\ \langle A \wedge\ T < v \rangle$ が正しい．

2. ループの繰り返しの間は T の値は正である．すなわち、$\neg\alpha \wedge A \rightarrow T > 0$ である．

もし、与えられた表明付きの until 文に対してこのような T を見つけることができれば、その until 文の停止性を示すことができる．このような式 T を上界関数[13]とよぶ．そこで次のような推論規則を考える．

$$\frac{\langle \neg\alpha \wedge A \wedge v = T \rangle\ P\ \langle A \wedge\ T < v \rangle}{\langle A \rangle\ \textbf{until}\ \alpha\ \textbf{do}\ P\ \textbf{od}\ \langle \alpha \wedge A \rangle}$$

(ただし、T は整数を値にもつ式で $\neg\alpha \wedge A \rightarrow T > 0$ をみたす)

この推論規則では、確かに上式が正しい表明つきプログラムならば下式も正しい表明つきプログラムである．これを用いることで多くの表明つきプログラムの正当性を示すことができる．例えば問題 6.8 において次の表明つきプログラムの正しさを証明したが、これの完全正当性版を考えてみよう．

$$\{i = 0 \wedge p = 0 \wedge n \geq 0\}\ \textbf{until}\ i = n\ \textbf{do}\ i := i + 1; p := p + x\ \textbf{od}\ \ \{p = n \cdot x\}$$

$T := n - i$ とおくと、この T が上界関数であることをわかるので、上記の推論規則より完全正当性の意味でも正しいことがわかる．

問題 **7.1**

$$\langle i < n \wedge p = i \cdot x \wedge v = n - i \rangle\ i := i + 1; p := p + x\ \langle i \leqq n \wedge p = i \cdot x \wedge n - i < v \rangle$$

が正しいことを示せ．

完全正当性に関する公理と推論規則をまとめると次のようになる．

完全正当性に関する公理と推論規則

代入文の公理

$$\langle R[t/x] \rangle\ \ x := t\ \ \langle R \rangle$$

帰結の推論規則

$$\frac{\langle A \rangle\ P\ \langle B \rangle}{\langle C \rangle\ P\ \langle D \rangle}$$

(ただし $C \rightarrow A$,$B \rightarrow D$ がともに真のとき)

[13] 上界関数を用いたが，一般の場合には整礎集合上の順序を用いる必要がある．

複合文の推論規則

$$\frac{\langle A\rangle\ P_1\ \langle S_1\rangle\ \langle S_1\rangle\ P_2\ \langle S_2\rangle \cdots \langle S_{n-1}\rangle\ P_n\ \langle B\rangle}{\langle A\rangle\ \ P_1; P_2; \cdots ; P_n\ \ \langle B\rangle}$$

条件文の推論規則

$$\frac{\langle A \wedge \alpha\rangle\ P_1\ \langle B\rangle \quad \langle A \wedge \neg\alpha\rangle\ P_2\ \langle B\rangle}{\langle A\rangle\ \textbf{if}\ \alpha\ \textbf{then}\ P_1\ \textbf{else}\ P_2\ \textbf{fi}\ \langle B\rangle}$$

until 文の推論規則

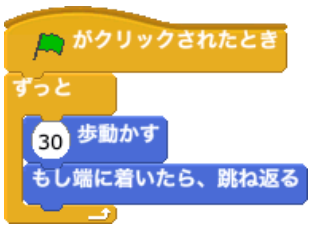
$$\frac{\langle \neg\alpha \wedge A \wedge v = T\rangle\ P\ \langle A \wedge\ T < v\rangle}{\langle A\rangle\ \textbf{until}\ \alpha\ \textbf{do}\ P\ \textbf{od}\ \langle \alpha \wedge A\rangle}$$

(ただし、T は整数を値にもつ式で
$\neg\alpha \wedge A \rightarrow T > 0$ をみたす)

8　並行性

8.1　並行に動作するプログラム

BYOB では一つのスプライトに対して複数のスクリプトを作成することが可能であり，またスプライトも複数あるとさらに多くのスクリプトが作成される．これらは一度に動作することが可能であり，その場合複数のスプリプトが同時に動作していることになる．このようにシステムが複数のプロセス（スクリプト）から構成され，それらが同時に動作するようなシステムを並行システムという．たとえば，BYOB で次のスクリプトをもつ２つのスプライトを作成して緑旗をクリックすると二つのスプライトは同時に動きだす．



二つのスクリプトに共通の資源（変数など）がない場合は並行に動作しても問題はおきないが，共有資源があると単体で動作する時には発生しなかった問題がおきることがある．例えば，次のようなスクリプトを作成する．

変数 var の値を 0 にして，これを実行すると変数 var の値は 10 で停止する．さらに，このスクリプトを複製して二つ同時に動かすと変数 var の値は 20 になり停止する．

このスクリプトを繰り返しブロックを用いて次のように変更する．

追加したものは単にブロック内を 1 回実行するだけなので，もとのスクリプトと実質的な内容は同じで変数 var の値は 10 で停止する．従って，これをそのまま動かしても特に問題はおきない．しかし，このスクリプトを複製して同時に二つのスクリプトを動かすとどうなるか．それぞれのス

クリプトは変数 var の値を 1 増やす操作をそれぞれ 10 回行うものであるから，最初のスクリプトと同じで変数 var の値が 20 になることが期待されるが，実際に動かしてみると変数 var の値は 10 で停止してしまう．単体では同じ結果を生むのに，複製して二つ同時に動かすと異なる結果を生む．これが並行に動作するシステムに潜む大きな問題を端的に表している．

なぜこのようなことが起きてしまうのか．これは後者の場合二つのスプライトの命令がインターリーブ (interleave) して交互に実行されるためである．つぎのように，いまどこを実行しているかわかるように各命令の間にリストへのテキスト出力を行うようにしてみると確かに交互にスクリプトが動作していることが確認できる．

(a) スクリプト　　(b) リスト

図 1: インターリーブして交互に動作する

上で確認したようにプロセスが複数互いに変数を共有したりメッセージを交換して動作する場合は単体では起こりえなかった問題が出現する．最初にあげた例のような場合はプログラムを作成する際に問題を見つけることが可能であるが，もっと複雑なプログラムになると単に動作を追うだけではわからない問題が潜む可能性がある．これが個人的に使うプログラムであれば不具合があっても自分の問題で済むが，社会において重要な役割を担うシステムでは，常に高い信頼性が求められる．

図 2: 実行順序

そのような重要なシステムでは，設計の段階からシステムの動作に問題が無いことを検証しておく必要がある．

この講義では，このような並行システムにおける問題をモデル検査ツール Spin を利用して検証する仕組みについて学ぶ．モデル検査とは「状態空間の網羅的な探索によって、当該システムが与えられた仕様を満たすか否かを判定する方法の総称」（中島 [14]）である．

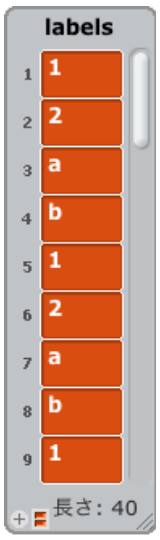
8.2 BYOB のスクリプトの実行順序について

並行性に潜む問題を BYOB のスクリプトを例にあげながら学んでいくが，その前に BYOB のスクリプトの命令がどのような順序で実行されるか確認をしておく．BYOB で複数のスクリプトが同時に動作する場合，それぞれのスクリプトが順に実行されるが，スクリプト内に制御タイルで最後に⏎があるものは制御がこのタイルの最後まで来たときは，いったん次のスクリプトに実行が移る．例えば，上であげた図 1 の (a) のスクリプトを考える．

緑旗がクリックされると最初に右側のスクリプトが動き，「1 回繰り返す」を終えた時点で左のスクリプトが動作し「1 回繰り返す」を行う．再び右側のスクリプトが動き次の「1 回繰り返す」を実行して，左のスクリプトに実行が移り「1 回繰り

返す」を行う．また，右のスクリプトに実行が移ると「10 回繰り返す」タイルの最後に⮒があるので，左に実行が移る左も同様に「10 回繰り返す」タイルの最後に⮒があるので，そのまま右へ移る．右のスクリプトは「10 回繰り返す」の先頭にもどり同じことを繰り返す．このことは最初に確認したように labels リストの内容をみることでも確認できる．この実行順序はスクリプト内に自分で作成したブロックがある場合も，そのブロック内の定義中に最後が⮒であるタイルを含む場合は同様にその時点で制御が他のスクリプトに移動する．例えば，上で作成したスクリプトを用いて，次のようにブロック left, right を作成してみる．ただし，上部のアトミックのチェックは入れないでおく．

これを用いて，次のようにスクリプトを作る．



このスクリプトを実行すると最初と全く同じ結果になる．

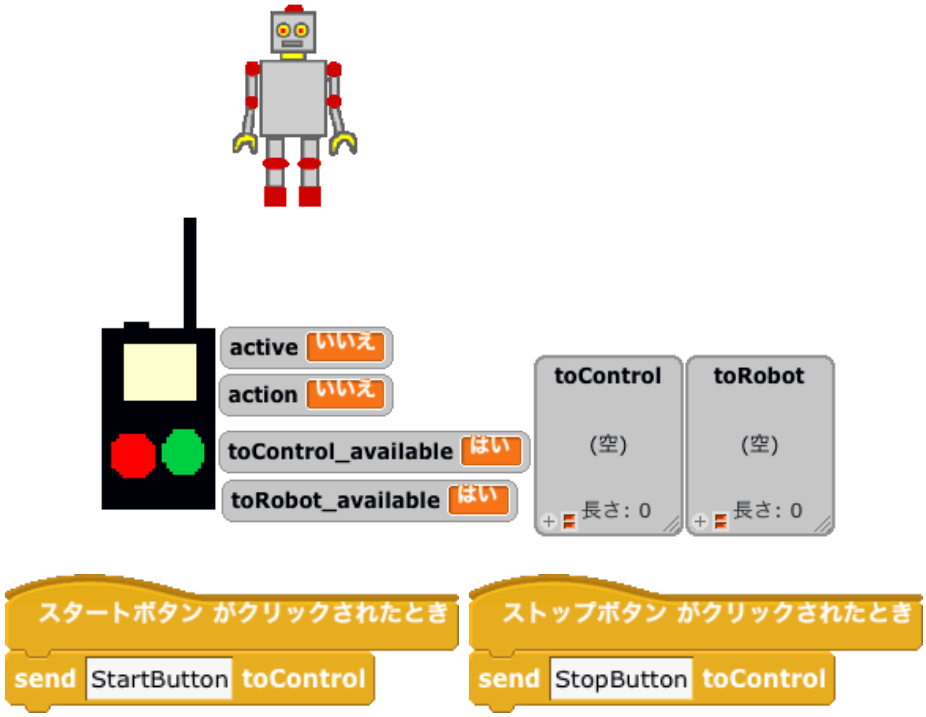
しかし，ブロックがアトミック指定されている場合はこのブロックの実行中に他のスクリプトが動くことはない．実際に上の left と right のブロック定義で上部のアトミック指定にチェックを入れてからスクリプトを実行すると，リスト labels は右図のようになり，interleave がおきていないことがわかる．また，「待つ」タイルである場合は，指定された時間が経過するまでは他のスクリプトが動作する．

8.3　意図しない結果になるプログラムの例

単純なスクリプトであれば並行に動作していても実行順序の予想はつくが，長いスクリプトがある場合やメッセージのやり取りをする場合などは実際にどのように実行が経過していくかは予想がつかず意図しない結果になる場合もある．

次のプログラムは，リモコンを用いてロボット操作を行うものである[14]．4つのスプライトと5つのスクリプトからできている．

プログラムの詳細は12.3で解説するが，ここではロボットとリモコンの間を高々長さ1のふたつのリスト toRobot と toControl を使用してデータの送受信を行う前提でプログラムが書かれている．



[14] この例は [19] で [14] の検証例を下に作成したものをこの講義資料用に改変したものである．

プログラムは次のように動くことを意図している．

- ユーザがスタートボタンを押すと，toControl に StartButton が送られ，リモコンは toControl から StartButton を読み込むと toRobot に Start を送る．
- ロボットは toRobot から Start を読み込むと動き始める．
- ユーザがストップボタンを押すと toControl に StopButton が送られ，リモコンはそれを受けて toRobot に Stop を送る．
- ロボットは Stop を受け取ると動作を停止し，toControl に Terminate を送る．
- リモコンは Terminate を受け取ると待機状態に戻り，ユーザがボタンを押すのを待つ．

スタートボタンとストップボタンを通常の使い方でクリックする限りでは正常に動作するため特に問題はないように思えるが，ロボットが動いているときに，ストップボタンを連打するとデッドロックをおこして動かなくなる。なぜ，このようなことになるかは 12.3 でモデル検査ソフトの Spin を用いて分析する．

9　有限 1 次元セルオートマトン

セルオートマトンは，チューリング機械と同様に両側に無限にのびたテープをもち，各セルは近傍（例えば両隣と自分）のセルの値から次の値を計算していく．例えば，テープ記号として $0, 1$ を使い，次の時刻のセルの値の計算を両隣のセルの値から次の規則

(左，自分，右)	000	001	010	011	100	101	110	111
次の値	0	0	1	0	1	0	1	0

により決めるとする．このとき，テープ上に 1 がひとつだけあるテープをこの規則を用いて計算すると，（両側に無限にある 0 は無視して記載すると）$010 \to 1, 100 \to 1, 000 \to 0$ より，100000 は 110000 となる．これを繰り返すと
100000
110000
011000
001100
000110
...

と変化していく．

両側無限でなく有限の長さのテープにすると端点の規則（境界条件）をどうするかが問題になるが，端と端をつないでリング状に考えたり（周期境界条件），端と同じ状態を仮想的に隣に考えたり（反射境界条件），特定の値に固定して考えたり（固定境界条件）する．

ここでは，マス目が 10 個のテープで周期境界条件により規則

000	001	010	011	100	101	110	111
0	1	1	0	1	0	0	1

を適用した例をあげる．

1000000000
1100000001
1010000010
1011000110
1000101000
...

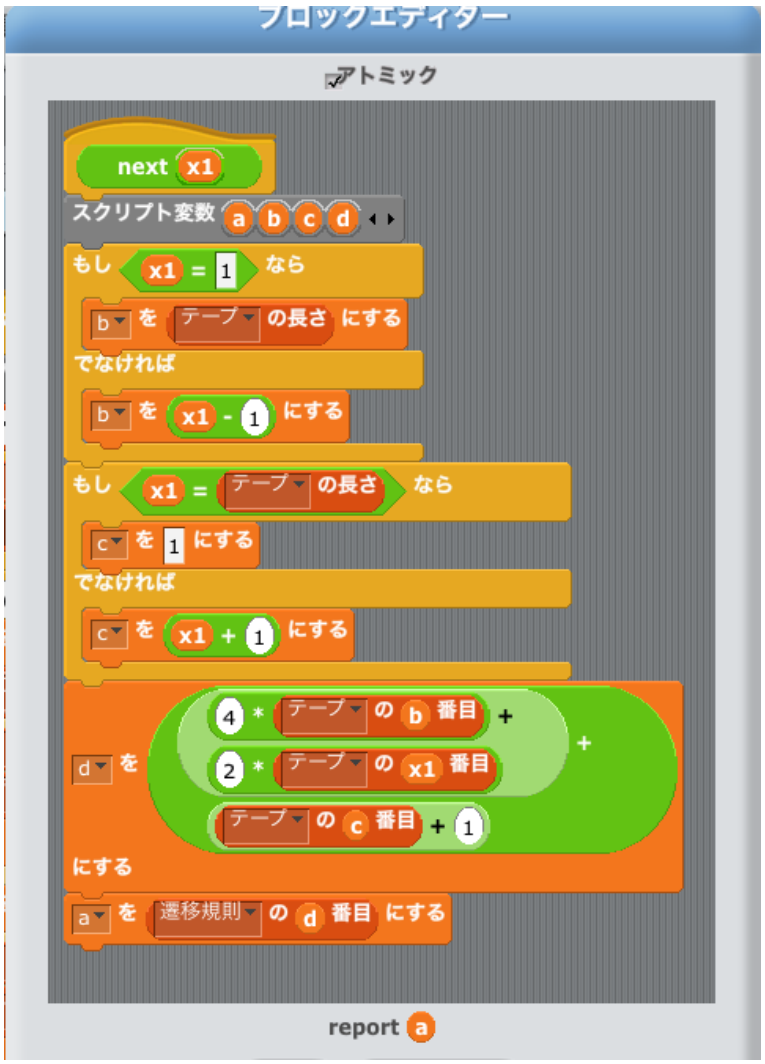
となる．

問題 9.1 上の遷移を実際に計算して確認せよ．

手計算では大変なので，上記の設定でセルオートマトンの時間発展を表示するプログラムを BYOB で作成してみよう．

まず，遷移規則，テープ，（次の時刻のテープ内容を記録する）仮テープ，時間発展のリストを作成する．遷移規則は $000, 001, 010, 011, 100, 101, 110, 111$ の順でリストに値を入れて定義することにする．それぞれを 2 進表示と考えて 10 進表示にすると $0, 1, 2, 3, 4, 5, 6, 7$ となるので，それぞれに対応する値は 2 進表示から 10 進表示を求めて 1 足して遷移規則の対応する値を調べることで得られる．例えば $(100)_{10} = 4$ であるから，遷移規則の 5 番目に 100 の対応する値が書かれていることになる．

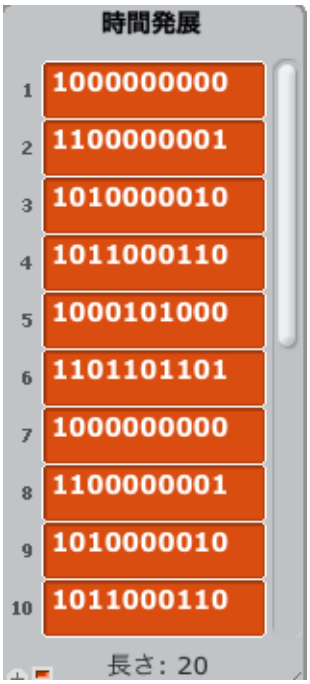
テープの i 番目のセルの次の時刻の値を計算する関数ブロック next は次のように定義できる．周期境界条件で考えるため最初に両端の処理を行っていることに注意する．



これを利用して次のようにスクリプトを作成する.

時間発展の回数を 20 回に固定しているが，自由に変更したい場合は，変数を定義してそれを利用するようにしたらよい．

さきほどの例を実際に実行してみると次のように時間発展が得られる．



問題 9.2 上記の BYOB スクリプトを作成して，遷移規則や初期値（長さも含めて）をいろいろ変更してセルがどのように変化するか調べよ．

ここで次のような問題を考える．与えられた遷移規則とテープの初期値に対して，テープのセルがすべて 1 になることがあるかどうか調べる．例えばテープの長さが 10

のときはテープの内容は高々 $2^{10} = 1024$ しかないので，1024 回書き換えを実行すると途中の異なる時刻に同一のセル内容が必ず現れる．時刻 t_1 のテープの内容 $C(t_1)$ と時刻 t_2 のテープ内容 $C(t_2)$ が同じとすると，$t_1 < t_2$ ならば，t_2 以降は t_1 から t_2 までの変化を繰り返すことになる．従って，1024 回書き換えを実行して，テープのセルがすべて 1 になることがなければ，それ以降も決して到達しないことがわかる．上の例では，6 回書き換えで初期値の 1000000000 に戻っている．従って，それ以降同じことの繰り返しになるので 1000000000 から 1111111111 に遷移することがないことがわかる．

ここまでは決定的なセルオートマトンを考えてきたが，ここで非決定的なセルオートマトンを考えてみる．例えば，上の例では，$101 \to 0$ であったが，101 は 0 でも 1 でもどちらに変えても良いことにする．このような非決定的な規則は次のように記述することにする．

000	001	010	011	100	101	110	111
0	1	1	0	1	0/1	0	1

こうすると，与えられた初期値に対して可能な遷移は 101 が出現する箇所で 2 通り考えられるため，20 回遷移を行うとその可能性は爆発的に増えることになる．さてこのように定義された非決定的セルオートマトンにおいて与えられた初期設定からテープのセルがすべて 1 になることがあるかどうか調べるにはどうしたらよいか．

非決定的なプログラムを BYOB で実行することはできないため，乱数を使ってシミュレートすることにする．遷移規則の決めるスクリプトの一部をを次のように変更する．

これで時間発展の回数を 1000 回ほどにしてシミュレートしてみると 1111111111 になる場合があることがわかる．

問題 **9.3** 実際に BYOB のスクリプトを動かしてこのことを確かめよ．

では，110 → 0 を同じように変える場合はどうなるか．こちらも同じようにシミュレートしてもなかなか 1111111111 にはならない．

問題 **9.4** 実際に BYOB のスクリプトを動かしてこのことを確かめよ．

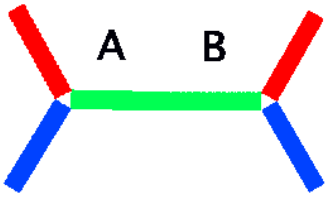
上の例のような場合テストだけでは，本当に 1111111111 に到達しないのかどうかはわからない．このようなときにモデル検査ツールを利用すると全数探索（もちろん資源に限りがあるのでいつでもできるわけではないが）を行って本当に 1111111111 に到達する可能性はないのかどうかを調べることができる．

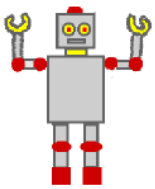
10　かんたんなモデル検査

10.1　状態遷移図による検査

ここでは簡単なシステムに対しシステムが与えられた性質をみたすかどうか，状態遷移図を実際に書いてみてしらべることにする．

例 **10.1** 次の様なシステムを考える．A と B はそれぞれ緑，赤，青の羽根を持ち，互いに連携しながら回転し，A と B が接触する色によりロボットが動いたり停止する．





システムの動作条件

1. A は最初，B と緑色の羽根で接触する位置にいて，順に右回りに 120 度ずつ回転する．ただし，B の赤の羽根と接触する場合，次の瞬間に回転しないこともある．

2. B は最初，A と緑色の羽根で接触する位置にいて，順に左回りに 120 度ずつ回転する．ただし，A と B が同じ色の羽根で接触するときは，次の瞬間には回転しない．

3. ロボットは最初停止していて，A と B がともに青で接触すると動き始め，A と B がともに赤で接触する時に停止する．それ以外は動作は変化しない．

このシステムを BYOB で実際に作成すると，例えば次のようなスクリプトになる．変数 stop はブール変数[15]としている．

[15]値として真偽値「はい」「いいえ」をもつ変数をブール変数とよぶ．BYOB では変数に区別はないので単に「はい」「いいえ」を代入するだけである．

ロボットのスクリプト

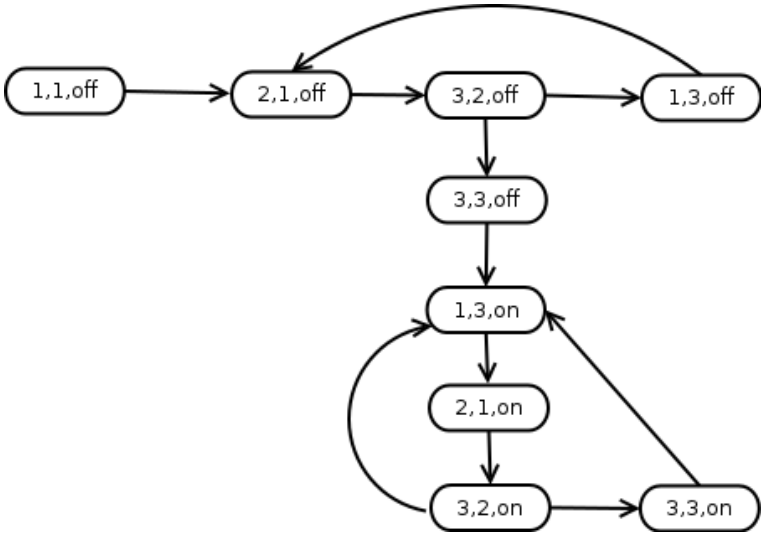
A の動作条件の非決定的な動きは記述できないので，A のスクリプトでは乱数を用いている．

A のスクリプト

B のスクリプト

問題 10.2 上のスクリプトで「1 回繰り返す」のブロックを外すと，意図した動作をしない．理由を考えよ．

このシステムの状態遷移図を書き，次の性質が成り立つかどうか確認する．
検査項目
・ロボットが停止していても，いつか動き出す．
・一旦ロボットが動き出すと，ずっと動いたままである．
このシステムの状態を，(接触地点の A の羽根の色，接触地点の B の羽根の色，ロボットの on/off) の 3 つ組で表すことにすると，状態遷移図は次のように表される．ただし，緑，赤，青をそれぞれ 1,2,3 で表している



従って，(1,1,off)→(2,1,off)→(3,2,off)→(1,3,off)→(2,1,off)→ · · · と繰り返すとロボットは停止したままであるので，最初の検査項目の「ロボットが停止していても，いつか動き出す」は間違いであることがわかる[16]．一方，一度ロボットが動き出すと，その後はロボットは on のままであるため，検査項目「一旦ロボットが動き出すと，ずっと動いたままである」は正しいことになる．

問題 10.3 例 10.1 で A が右回転ではなく左回転をするとき，検査項目の真偽はどうなるか答えよ．

問題 10.4 例 10.1 で，動作条件を次のように変更するとき検査項目の真偽はどうなるか答えよ．

[16]最初の検査項目の反例は確率的なプログラムで動かし続けると非常に小さな確率になるので，上であげた BYOB のスクリプトを実行するとロボットは常に動き出す．BYOB のスクリプトで乱数を取っている部分を外部のセンサー値，例えば光センサーの値でどちらかをとることにすると光センサーの置かれた場所によってはいつまでたってもロボットが動き出さない可能性がある．

1. A は最初，B と緑色の羽根で接触する位置にいて，順に右回りに 120 度ずつ回転する．ただし，B の赤の羽根と接触する場合，次の瞬間に回転しないこともある．

2. B は最初，A と緑色の羽根で接触する位置にいて，順に左回りに 120 度ずつ回転する．ただし，A と B が青色の羽根で接触するときは，次の瞬間に逆回転することもある．

3. ロボットは最初停止していて，A と B がともに青で接触すると動き始め，A と B がともに赤で接触する時に停止する．それ以外は動作は変化しない．

10.2 時相論理について

システムの状態は時間とともに変化するため，システムに関わる問題を考察する場合時間に関わる推論が必要となる．モデル検査では時間とともに変化するシステムが望ましい性質をもつかどうか検査する．この時間とともに変化する性質を記述するために時相論理式を用いる．時相論理では，$\wedge, \vee, \rightarrow \neg$ などの論理結合子とともに，$\Box$ (いつでも)，$\Diamond$ (いつか)、$\circ$ (次に) などの様相結合子を用いて論理式[17]を記述する．

例えば，「ロボットが停止していても，いつか動き出す」は，

$$\Box(\text{robot==off} \rightarrow \Diamond(\text{robot==on}))$$

と表すことができ，「一旦ロボットが動き出すと，ずっと動いたままである」は，

$$\Box(\text{robot==on} \rightarrow \Box(\text{robot==on}))$$

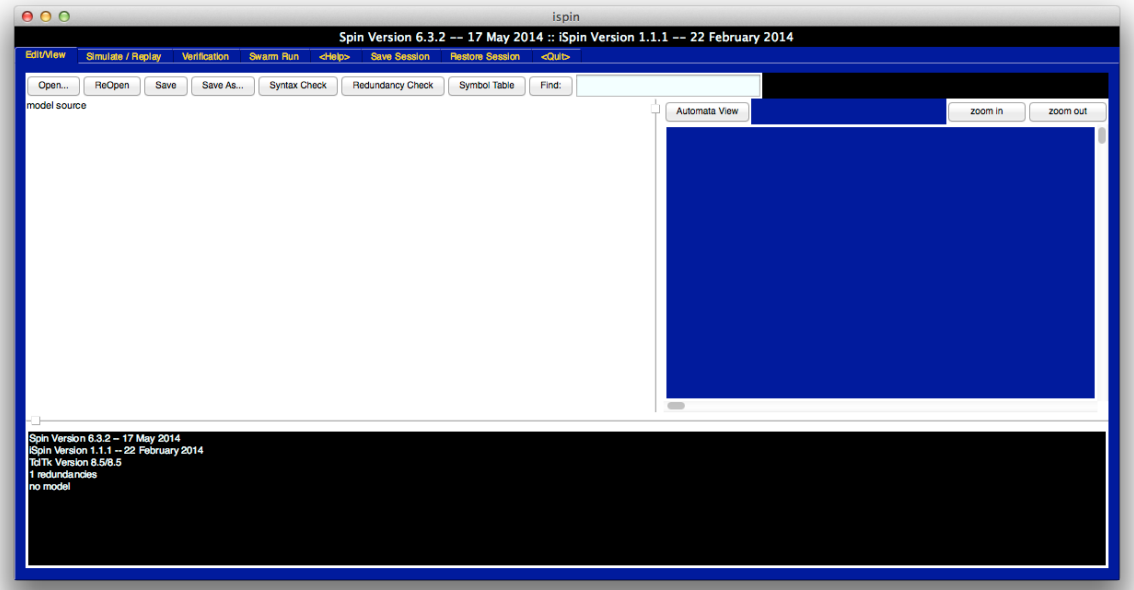
と表すことができる．

11 SPIN のかんたんな使い方

SPIN(Simple Promela INterpreter) のインストールについては，
`http://herb.h.kobe-u.ac.jp/spin_install_document.html` を参考にせよ．ここでは ispin を用いて説明する．

ispin を実行すると次のウインドウが出る．

[17] この授業で扱う時相論理は LTL とよばれる線形時相論理であるが，この授業の範囲では様相結合子の意味だけわかれば十分であるので詳しくは述べない．詳しくは参考文献を参照して欲しい．



左側の model source の部分に Promela(Process Meta Language)（C 言語に近い）でプロセスの動作を記述する．外部エディターでソースプログラムを書いて，ispin で読み込むこともできる．このテキストだけで Promela の説明をするのは困難であるので，さらに SPIN や Promela について知りたい人は [11], [17] などを参照して欲しい．

11.1 プロセスの記述

11.1.1 変数

データ型は，`int`，`bool`，`bit`，`byte`，`short` が使える．変数の宣言は C と似ている．例えば，int 型の変数 x,y を宣言するときは

```
int x,y;
```

と宣言する．変数はできるだけ宣言と同時に初期化する．これは記述したシステムの状態数をできるだけ減らすためである．初期化する場合は，

```
int x=1,y=0;
```

とする．（初期化しなければ，spin は自動的に初期値として 0 または false を入れるため不要な状態が増える．）

次に C にはないものであるが，特定の文字列を値にもつ変数を宣言するために mtype がある．例えば，left と right を値にとる変数 move を作りたい場合は，

```
mtype ={left,right};
mtype move=left;
```

とする．

11.1.2 プロセスの記述その 1

並行システムでは複数のプロセスが並行的に動作するが，それぞれのプロセスを表現するために Promela では次のように 1 つのプロセスを表現する．なお，active をつけることで，プログラムの実行と同時にプロセスが動き始める．（active を付けずに，他のプロセスから明示的に動作を開始させる方法もある．）

```
active proctype P(){

/* ここにプロセスを記述 */

}
```

C と同様に，/*と*/で囲まれた部分はコメントになる．

11.1.3 do 文

並行システム等で停止せずに動き続けることが求められる場合は，プロセスの記述もそのように行う．繰り返しを表現するのはおもに do 文である．do 文は，do と od にはさんで記述し，break 文で強制的に停止しない限り動き続ける．do 文は次のように記述する．else はあってもなくても構わない．条件 1 から条件 n のいずれも成り立たない場合にのみ実行される．

```
do
:: 条件 1->文; 文; . . . ; 文
:: 条件 2->文; 文; . . . ; 文
. . .
```

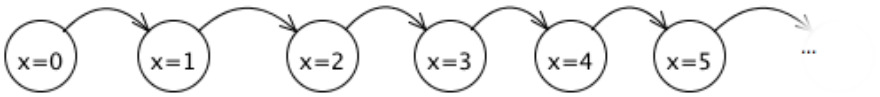
```
:: 条件 n->文; 文; . . . ; 文
:: else ->文; 文; . . . ; 文
od
```

条件は排他的である必要はなく，同時に複数の条件を満たす場合があってもよい．その場合は非決定的にどれかのプロセスが実行される．また，条件はなくても構わない．->は条件をわかりやすく記述するためのもので実際はセミコロン; と同じ働きである．; は Promela では区切りの文字なので，最後は書かなくて構わない．

例として次のプロセスを考える．

```
int x=0;

active proctype P(){
do
:: x=x+1
od
}
```

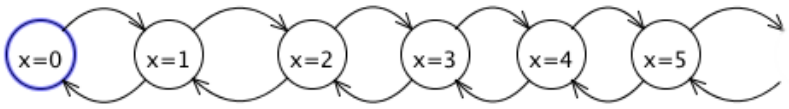
このプロセスは，変数 x の値を増加させるだけある．状態遷移図で表すと次の様なプロセスを表す．



もう少し複雑な例を作ると，

```
int x=0;

active proctype P(){
do
:: true->x=x+1
:: x>0->x=x-1
od
}
```

true 文は常に真である文である．これを状態遷移図で表すと次のようになる．



11.1.4 if 文

if 文も do 文に似た構文である．

```
if
:: 条件 1->文; 文; . . . ; 文
:: 条件 2->文; 文; . . . ; 文
...
:: 条件 n->文; 文; . . . ; 文
:: else ->文; 文; . . . ; 文
fi
```

条件が成り立つときに，その後の文が実行される．

11.2 性質のチェック方法

Promela で記述したシステムに対して，与えられた性質をもつかどうかチェックをすることができる．最初は一番かんたんな assert 文を利用したチェック方法について説明し，その後 LTL 論理式で表現された性質のチェック方法について説明する．

11.2.1 assert 文

assertion 文は，次のように書く．

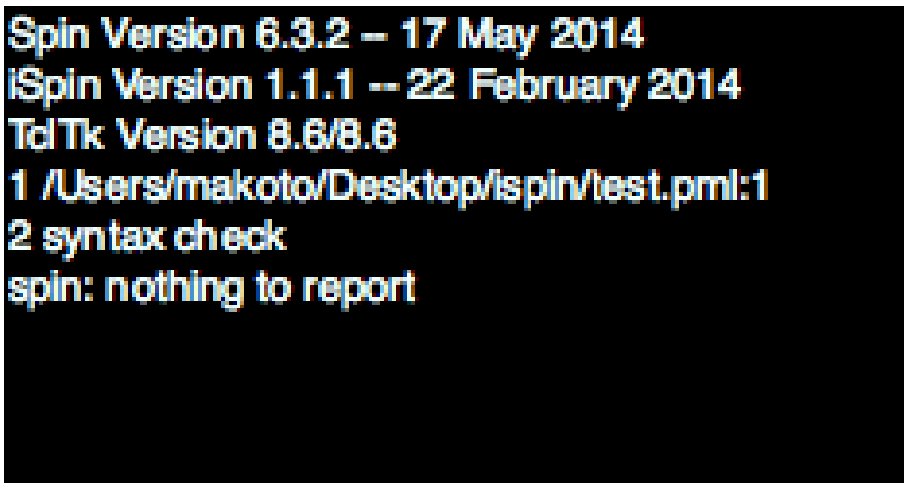
```
assert{ブール式}
```

ブール式を評価して false になると assertion error を報告する．その時点で成り立って欲しい性質を記述することで，確かに問題が起きないことの検査ができる．

例えば，次のような Promela プログラムを作成して ispin で読み込む．

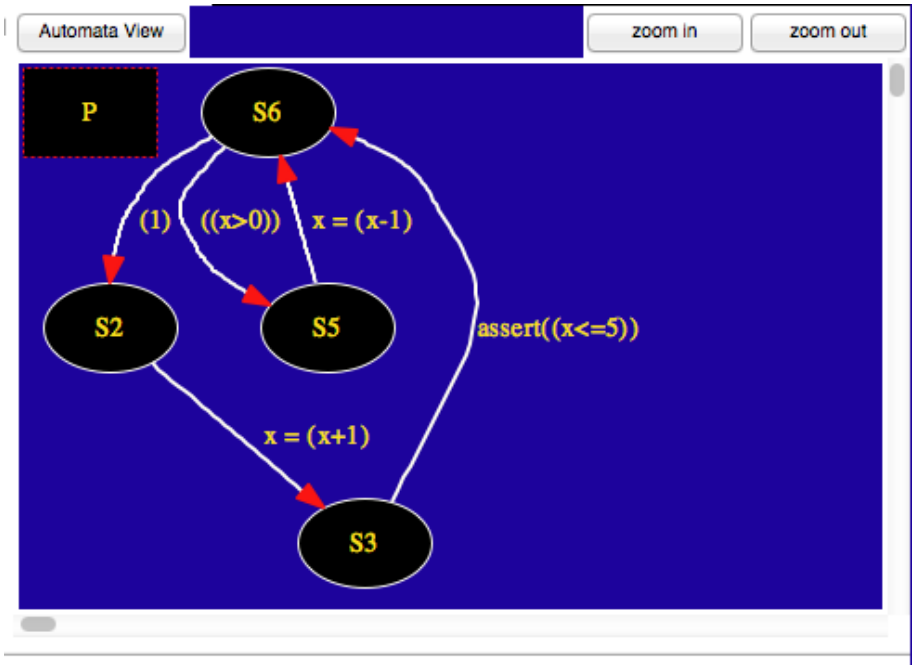
```
int x=0;

active proctype P(){
do
:: true->x=x+1;assert{x <=5}
:: x>0->x=x-1
od;
}
```

ispin の Edit/View タブの Open をクリックして，保存したファイルを開く．最初に，Syntax Check を実行する．ウインドウ下部の黒い部分に，spin: nothing to report と出れば文法エラーはない．（ファイルを編集したときは必ずこのチェックを忘れないようにする．）

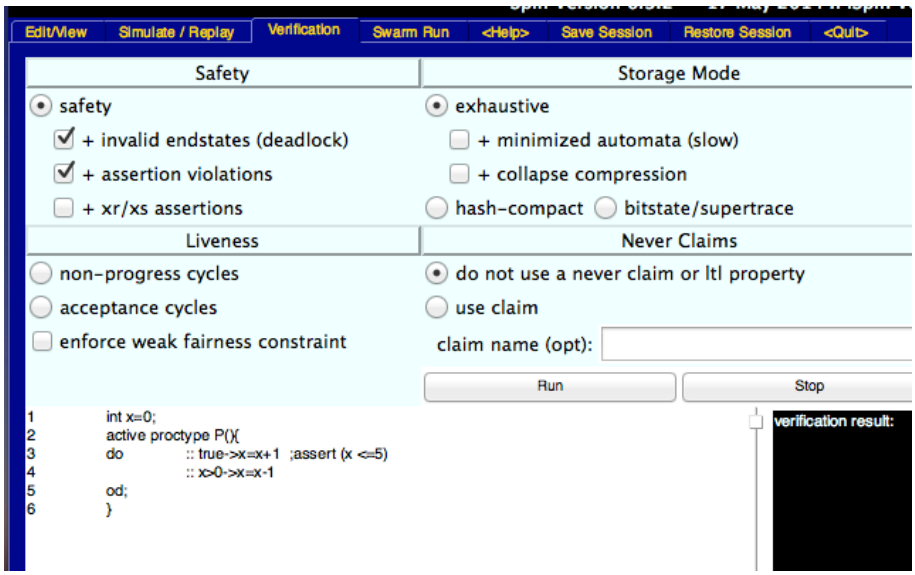


Dot[18]がインストールされている場合は，同じウインドウで Automata view をクリックすると状態遷移図が作成され確認できる．

[18] http://www.graphviz.org より Graphviz をインストールすると Dot が自動的にインストールされる．

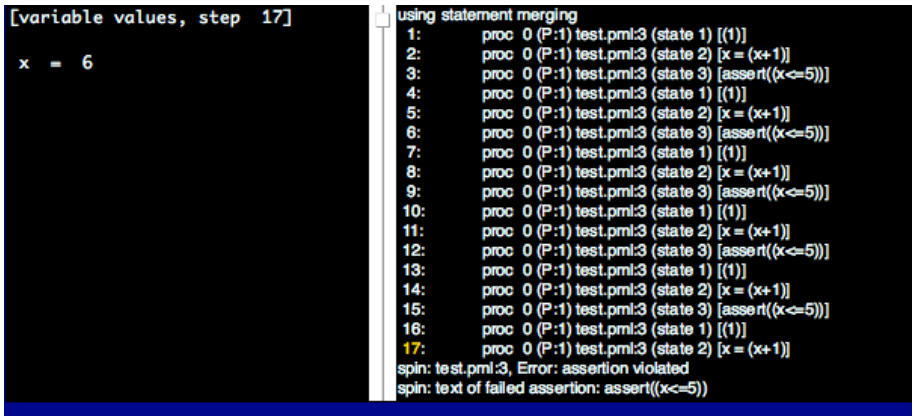


次に，Verification タブをクリックする．デフォルトでは，Safety にチェックがあり，invalid endstates(dead lock) と assertion violation にチェックがあるが，もし，そうなっていなければ Safety にチェックを入れて，invalid endstates(dead lock) と assertion violation にチェックを入れる．



確認できたら下の Run ボタンをクリックすると，検証結果がでる．今回は，

```
pan:1:  assertion violated (x<=5) (at depth 17)
```
となり，assertion が成り立たない場合があることがわかる．どのような場合に assertion が成り立たないか調べるには，Simulate/Replay タブ内の Run を，Guided にチェックが付いているのを確認してクリックする．



Rewind をクリックすると，実行ステップが先頭にもどる．Step Forward をクリックしていくとエラーが出てくるまでのステップを表示する．spin は判断した条件なども 1 ステップで表示するので，プロセスの経過が読みにくいが，今回は variable values ウインドウを見ると，`x=x+1` を実行し続けることで `x` の値が 6 になり，assertion が成り立たなくなったことがわかる．

11.2.2 LTL 論理式

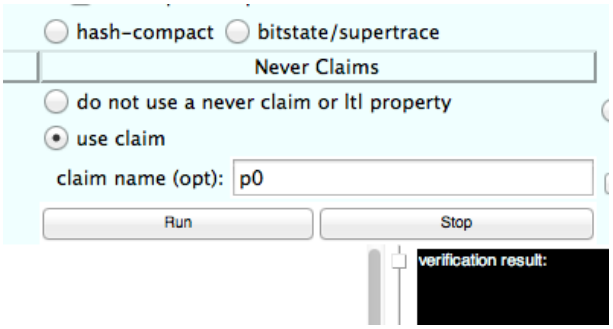
上の assert 文と同じプログラム（assert 文は除く）に対して，LTL 論理式で検査を行うことにする．検査論理式は □(`x<=5`)，すなわち，いつでも `x <=5` がなりたつこととする．これを検査するには，プログラムに以下の文を追記する．

```
ltl p0 {[](x<=5)}
```
LTL 論理式を記述する文法は

```
ltl [名前] {LTL 論理式 }
```
である．名前は LTL 論理式を区別するためにつけるがなくても良い．付けない場合は自動的に `ltl_0` のように spin が自動的に付ける．なお，ltl 文はプロセス記述の外側に書かなくてはならない．Promela では，様相結合子として，□, ◇ の他に U(Until),W(weak Until),V(dual of U) を使うことができる．詳しくは Promela の Man-Pages and Semantics Definition[18] を参照してほしい．

LTL 論理式を検査するには，Verification タブをクリックして，Never claims のところの use claim にチェックを入れて，検証する式の名前，例えば p0 を入力する．

検証式が一つしか書いていない場合や先頭の検証式を検証する場合は空欄でも構わない．



Run をクリックすると検証が始まる．検証式に反する実行経路がある場合，`errors:1` のようにエラーの存在を表示する．今の場合は assertion 文でチェックしたように，反例があるのでエラーがあることを表示して停止する．

```
spin -a  test.pml
ltl p0: [] ((x<=5))
gcc -DMEMLIM=1024 -O2 -DXUSAFE -w -o pan pan.c
./pan -m10000  -a -N p0
Pid: 21349
warning: only one claim defined, -N ignored
pan:1: assertion violated  !( !((x<=5))) (at depth 24)
pan: wrote test.pml.trail

(Spin Version 6.3.2 -- 17 May 2014)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim         + (p0)
        assertion violations + (if within scope of claim)
        acceptance   cycles  + (fairness disabled)
        invalid end states - (disabled by never claim)

State-vector 28 byte, depth reached 24, errors: 1
      13 states, stored
       0 states, matched
      13 transitions (= stored+matched)
       0 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
   0.001   equivalent memory usage for states (stored*(State-vector + overhead))
   0.291   actual memory usage for states
 128.000   memory used for hash table (-w24)
   0.534   memory used for DFS stack (-m10000)
 128.730   total actual memory usage



pan: elapsed time 0.01 seconds
To replay the error-trail, goto Simulate/Replay and select "Run"
```

反例に至る実行経路は assert 文のときと同様に確認できる．

例 **11.1** 例 10.1 では，システムが同期して変化するがこのようなシステムの場合は，変数の更新を同時にする必要がある．例えば，例 10.1 は次のように Promela で記述できる．

```
mtype = {green, red, blue, on, off};
mtype a=green, b=green;
mtype robot=off;
mtype Na, Nb, Nrobot;

active proctype robo(){
    do
    ::true ->
        if        /*  次の瞬間の A の状態の決定  */
        :: (b==red) -> Na = a
        :: (a==green) -> Na = red
        :: (a==red) -> Na = blue
        :: (a==blue) -> Na = green
        fi;

        if        /*  次の瞬間の B の状態の決定  */
        :: (a==b) -> Nb = b
        :: else ->
            if
            :: (b==green) -> Nb = red
            :: (b==red) -> Nb = blue
            :: (b==blue) -> Nb = green
            fi
        fi;

        if        /*  次の瞬間の robot の動作決定  */
        :: (a==blue && b==blue) -> Nrobot = on
        :: (a==red && b==red) -> Nrobot = off
        :: else           -> Nrobot = robot
        fi;
/* 値の更新*/
         a = Na;
         b = Nb;
         robot = Nrobot
    od
```

問題 11.2 上の Promela プログラムに対し，

$$\Box(\text{robot}==\text{off} \rightarrow \Diamond(\text{robot}==\text{on}))$$

及び

$$\Box(\text{robot}==\text{on} \rightarrow \Box(\text{robot}==\text{on}))$$

を Spin で検査せよ．

問題 11.3 ６つのマス目のなかに５枚の数字が入ったコマがあり，一つの空きスペースを利用して，コマを動かすことができる．

このとき，何回かコマを動かして指定された形にできるか Spin を利用して調べよ．

1. 

を 

にできるか．（すなわち 1 と 2 は固定して，3,4,5 を右回りに回せるか？）できる場合はその手順を示せ．

2. 任意の２つのコマを入れ替えることはできるか．

例えば， 

を 

にできるか[19]．

（すなわち 3 と 4 と 5 は固定して，1 と 2 を交換できるか？）

問題 11.4 次の非決定的な有限セルオートマトンに対し，テープ 1000000000 をかけて動かす．ただし，境界条件は周期境界条件とする．このとき，1111111111 となる可能性があるかどうか Spin を用いて調べよ．

000	001	010	011	100	101	110	111
0	1	1	0	1	0	0/1	1

ヒント：同じ様な記述が続くので inline 記法を用いて，次のようにマクロ定義を利用するとよい．

```
inline trans(x,y,z,u){
if
::(x==0&&y==0&&z==0)||(x==0&&y==1&&z==1)||
            (x==1&&y==0&&z==1)||(x==1&&y==1&&z==0)->u=0
::(x==0&&y==0&&z==1)||(x==0&&y==1&&z==0)||
            (x==1&&y==0&&z==0)||(x==1&&y==1&&z==0)||(x==1&&y==1&&z==1)->u=1
fi;
}
```

12　複数プロセス

12.1　複数プロセスの記述

ここまでの Promela プログラムの例は，例 10.1 の Promela 記述も含め，すべて単一のプロセス内で状態の分岐がおきるものであったが，一般的なリアクティブシステムでは複数のプロセスが相互作用しながら動作を続けて行く．従って，今後は複

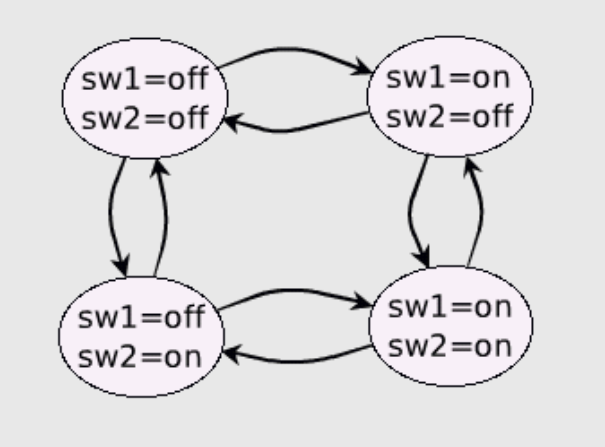
[19] これができると，他の場合もできることが少し考えるとわかる．

数のプロセスが同時に動作しているシステムの記述とそこでおきる問題について考察していく.

例 **12.1** ふたつのスイッチ sw1 と sw2 があり，単にそれらが独立に on と off を繰り返すシステムは次のように記述できる.

```
mtype ={on,off};
mtype sw1=off, sw2=off;

active proctype P1(){
do
:: sw1==off-> sw1=on
:: sw1==on -> sw1=off
od
}

active proctype P2(){
do
:: sw2==off-> sw2=on
:: sw2==on -> sw2=off
od
}
```

Spin は複数のプロセスに対して，合成したプロセスを作成して検証する．例えば，この例では，spin 内部では次のような状態遷移図をもつプロセスが構成される.



この例は二つのプロセスが独立に動いているので，互いに相手に影響を及ぼすことはないが，次は同時に動作することで影響するプロセスの例である．

例 **12.2** `int x;`

```
proctype Proc1() {
  x = 1;
  x = x + x;
  assert(x == 2);
}

proctype Proc2() {
  x = 0;
}

init {
  run Proc1();
  run Proc2();
}
```

ここでは，プロセス定義には `active` を付けない変わりに，`init` プロセスを定義して，`Proc1` と `Proc2` を順に走らせている．一見すると，`Proc1` が動作を終えてから，`Proc2` を動作させているように思えるが，プロセスは独自の実行速度をもち，また文の実行も `Proc1` と `Proc2` がインターリーブ（間に互いに挟まって実行が進む）するため，`assert` 文がなりたたない場合があることを spin は報告する．

同時に動くことを想定している場合は、このように意図せぬ動作がおきることに注意をする必要がある．上の例では、`x=1;x=x+x` の間に他のプロセスにより、x の値が書き変わることを防ぐ必要がある．Promela ではそのような対処をするための仕組みとして、`atomic` がある．一連のプログラムを `atomic` でくくると、その内部のプログラムを実行中は他のプロセスによるインターリーブを許さないことができる．これは BYOB で関数ブロックを定義してアトミックの指定をするのと同じである．

問題 **12.3** 次のシステムを検証せよ．

```
int x;

proctype Proc1() {
  atomic{x = 1;
```

```
  x = x + x;
  assert(x == 2)
  }
}

proctype Proc2() {
  x = 0;
}

init {
  run Proc1();
  run Proc2();
}
```

12.2 排他制御

上の例のように，何かの変数を変えたりするときに他のプロセスに邪魔をされたくない場所のことを危険部分（Critical Section）とよび，その危険部分に入るプロセスを制御する仕組みを相互排除とよぶ．相互排除を実現するための仕組みはいろいろあるが，デッドロック（互いに動作を牽制し合い誰も動けなくなる）がおきないこと，応答性（入ろうとしたら必ずいつか入れる）などを保障することが重要である．次にあげるのは相互排除のためのアルゴリズムの一つであるピーターソンのアルゴリズムである．なお，この Promela 記述で，try とか cs とあるのはラベルで，プログラムそのものには影響はない．検証する時にプログラムの位置を示すのに利用する．

```
bool wantP=false,wantQ=false;
byte last=1;

active proctype P(){

do
:: wantP=true;
   last=1;
 try: (wantQ==false)||(last==2);
 cs:   wantP=false
```

```
od
}

active proctype Q(){

do
:: wantQ=true;
   last=2;
 try: (wantP==false)||(last==1);
 cs:   wantQ=false
 od
}
```

二つのプロセス P,Q はそれぞれ危険部分 (cs のラベルがある部分) に入りたい場合は，それぞれ wantP と wantQ を true にする．次に last 変数を P は 1 に Q は 2 にする．これは危険部分を直前に自分が使っていて相手が待っている可能性があるので，その場合に相手に先に入らせるための仕組みである．ラベル try の場所では．相手が危険部分に入ることを表明していないか，相手が自分に先に入れるよう last 変数が自分用に代わっているかを判断して，どちらかが真のときは危険部分に入り，自分の要求を偽にして，もとにもどる．(本来であれば，危険分で何らかの仕事をしたり，危険部分に入ろうとする以前に何らかの仕事をしているはずであるが，その部分はすべて隠して本質的な部分のみをこのアルゴリズムは表現している． 上で，ラベルは検証する時にプログラムの位置を示すのに利用すると書いたが，例えば，P@cs という述語は，プロセス P がラベル cs の位置にいれば真で，他の位置にいれば偽になる．従って，♢(P@cs && Q@cs) は，実行途中で，P も Q も cs の位置にいることがあれば真になるので，この検証式の否定!♢(P@cs && Q@cs) が正しいかどうかを検証して，もし反例があがればピーターソンのアルゴリズムは正しくないことになる．

問題 **12.4** ピーターソンのアルゴリズムが正しいことを検証せよ．

問題 **12.5** BYOB でピーターソンのアルゴリズムを実装してその動きを確認せよ．

12.3 具体的なプログラムの検証

最後に 8.3 であげたプログラムに対して，Spin を用いて検証を行う．

12.3.1　BYOB スクリプトの解説

検証を行う前にプログラムの詳細を解説する．プログラムで使用するスプライトは，リモコン，ロボット，スタートボタン，ストップボタンでそれぞれに対して次のようにスクリプトを作成する．

まず最初に（デッドロックに陥った時に初期設定を容易にするために）緑旗をクリックしたら変数やリストを初期化してリモコンとロボットの動作体制をとるため次のようにスクリプトを作成する．

ここで initialize ブロックはつぎのように作成する．

変数の toControl_availabe と toRobot_available はそれぞれリスト toControl,toRobot が使用できるかどうかを表すためのブール変数である．リストには高々 1 つのデータしか登録できないという仕様のために，ここが危険領域になる．そこでピーターソンのアルゴリズムと同様にこのような変数を用意して管理する．また，変数の active はリモコンからロボットに Start を送信したことを表すためのブール変数で，変数 action はロボットが動いていることを表すためのブール変数である．

スタートボタンとストップボタンのスクリプトはつぎのようにする．

ここで send-messgage タイルは次のように作成する．

toControl リストが使用可能になるまで待って，メッセージ（StartButton/StopButton）を送る．メッセージが相手に読み込まれるまでの間，toControl に他のメッセージが送られてくることを阻止するために available 変数を「いいえ」にしている．

リモコンのスクリプトは次のようにする．

リモコンはメッセージを受信すると何らかの動作を行うようにしている．toControl receive msg ブロックはつぎのような述語型のブロックとして作成しているため，「toControl receive msg を待つ」は「toControl_availabe を待つ」と同じことである．メッセージのやり取りを意識してこの

待機状態（ロボットに動作指令を出していない状態）のときは，toControl にメッセージが送られてくるのを待ち，メッセージが StartButton ならば，ロボット（正確には toRobot リス

ト）に Start を送る．待機状態で Ston-
button を読み込むとき無視すること
にしている．Start を送信後は待機状態から動作状態に移行し，次のメッセージの受信を待つ．動作状態で toControl から StopButton を読み込むと，toRobot に Stop を送る．動作状態で StartButton を読み込むときは無視をする．toControl から Terminate（これはロボットは動作を停止したことをリモコンに伝えるために送るメッセージ）を読み込んだら待機状態に戻る．

ロボットのスクリプトは次のようにする．

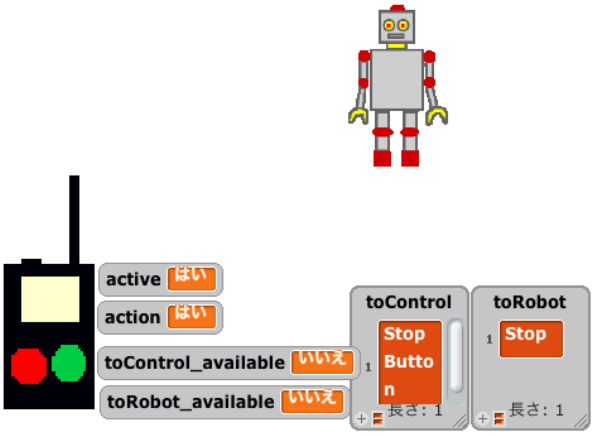
ロボットは，待機状態ではリモコンから toRobot へのメッセージ受信を待ち，toRobot に Start が送られて来たら動作状態に移行する．待機状態で Stop が送ら

れて来た場合は無視する．動作状態に移行後は，リモコンから toRobot に Stop が送られてくるまで動き続ける．動作状態で Start が送られて来た場合は無視する．Stop が送られて来たら動作を停止し，リモコン（正確には toControl に）Terminate を送信して待機状態に戻る．

以上がプログラムの全体である．

このプログラムを緑旗のクリックから開始し，スタートボタン／ストップボタンをクリックしてみても特に問題はないように思えるが，次のようなスクリプトを作成して，ロボットの動作状態で，左矢印キーを押し続けてみる．

すると次のように toContro と toRobot にメッセージが残ったままデッドロックに落ちいる．



12.3.2 Spin による解析

上で作成した BYOB プログラムを Spin を用いて解析してみる．このプログラムを Promela で表現すると例えば次のようにできる．

```
mtype = {StopButton, Terminate, StartButton, Start, Stop};
mtype msg;
chan toControl =[1] of {mtype};
chan toRobot = [1] of {mtype};
```

```
active proctype PushStart ()
{
end:
 do
 :: toControl!StartButton
 od
 }

active proctype PushStop ()
{
end:
 do
 :: toControl!StopButton
 od
 }

 active proctype Controller()
 {
 Inactive: end:
 do
 ::atomic{ toControl?msg ->
 if
 :: (msg==StartButton) -> toRobot!Start ; goto Active
 :: (msg==StopButton)
 fi
 }
 od;

 Active:
 do
 ::atomic{ toControl?msg ->
 if
 :: (msg==StopButton) -> toRobot!Stop
 :: (msg==Terminate) -> goto Inactive
```

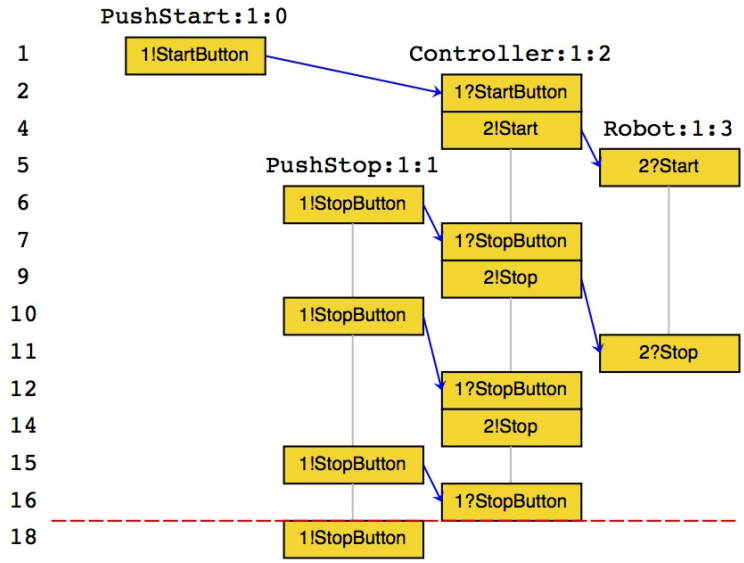
```
:: (msg==StartButton)
fi
}
od
}

active proctype Robot()
{
Inactive: end:
do
:: atomic{toRobot?Start -> goto Action}
:: toRobot?Stop
od;

Action:
do
:: atomic {toRobot?Stop -> toControl!Terminate; goto Inactive}
od
}
```

これを Spin にかけて検証を行うと，

```
spin -a  robot.pml
gcc -DMEMLIM=1024 -O2 -DXUSAFE -DSAFETY -DNOCLAIM -w -o pan pan.c
./pan -m10000
Pid: 68772
pan:1: invalid end state (at depth 44)
pan: wrote robot.pml.trail

(Spin Version 6.3.2 -- 17 May 2014)
Warning: Search not completed
+ Partial Order Reduction

Full statespace search for:
never claim          - (not selected)
assertion violations +
```

```
cycle checks        - (disabled by -DSAFETY)
invalid end states +

State-vector 60 byte, depth reached 45, errors: 1
       46 states, stored
       21 states, matched
       67 transitions (= stored+matched)
       36 atomic steps
hash conflicts:         0 (resolved)
```

となりエラーを報告する．エラーがおきるまでの過程を（shorttrail のオプションにチェックを入れて）Simulate で確認すると，



となり，ロボットが動作中にリモコンから Stop が送信され，それを受けて動作を終了しリモコンに Terminate を送信しようとしたときに，ストップボタンが先に押されて toControl に StopButton が送られたため，リモコンは toRobot に Stop を送信しようとする．さらに，そのタイミングであらに StopButton が押されたため toControl には StopButton が送られている．一方，ロボットは Terminate の送信待ちで toRobot を受け取る状態にないため，リモコンは Stop 送信で待ち続けるため Terminate の受信に移行できないというデッドロックがおきていることがわかる．

よって，このデッドロックを解消するにはリモコンが動作状態のときに StopButton を受け取ったら，ロボットから Terminate が送られてくるまでストップボタンが押

されても無視する必要があることがわかる．例えば，次のようにリモコンのスクリプトを書き換えるとこのデッドロックは解消される．

この例であげたプログラムは実行時のバグが比較的にわかりやすいものであるが，大規模なプロブラグでは潜在的に潜むバグがテストなどではあらわれないことが多い．そのような場合でも Spin などのモデル検査ツールを利用することでシステム設計に潜むバグを見つけ出すことが可能である．

参考文献

[1] 計算モデル論入門ーチューリング機械からラムダ計算へ , 井田哲雄・浜名誠共著 , サイエンス社

[2] C 言語による計算の理論, 鹿島亮著, サイエンス社

[3] 計算論への入門-オートマトン・言語理論・チューリング機械 , エフィーム・キンバー／カール・スミス著　筧捷彦 監修　杉原崇憲 訳 , ピアソンエデュケーション

[4] 計算論ー計算可能性とラムダ計算，高橋正子著，近代科学社

[5] 計算論, 廣瀬健著, 朝倉書店

[6] 計算の基礎理論, 細井勉著, 教育出版

[7] プログラム検証論 , 林晋著 , 共立出版

[8] コンピュータサイエンス入門, 論理とプログラム意味論, 田辺誠・中島玲二・長谷川真人著, 岩波書店

[9] プログラム仕様記述論, 荒木啓二郎・張漢明著, オーム社

[10] Principles of Model Checking, C. Baier and J.P. Katonen, The MIT Press

[11] SPIN モデル検査入門 , Mordechai Ben-Ari 著　中島 震　監訳 , オーム社

[12] Logic in Computer Science-Modelling and Reasoning about Systems, M. Huth and M. Ryan,2nd edution, Cambridge University Press

[13] コンピュータサイエンス入門　論理とプログラム意味論 , 田辺誠・中島玲二・長谷川真人著 , 岩波書店

[14] SPIN モデル検査 検証モデリング技法, 中島 震著, 近代科学社

[15] SPIN モデル検査入門, 中島震・谷津弘一・野中哲・足立太郎著, オーム社

[16] 4 日で学ぶモデル検査, 産業技術総合研究所システム検証研究センター, NTS

[17] Spin Online References, `http://spinroot.com/spin/Man/index.html`

[18] Promela:Man-Pages and Semantics Definition, `http://spinroot.com/spin/Man/promela.html`

[19] ビジュアルプログラミング環境を用いたシステム検証の教育について，山上和哉，修士論文，神戸大学大学院人間発達環境学研究科，2013

平成 27 年 12 月 17 日作成
平成 28 年 4 月 30 日修正
平成 28 年 6 月 22 日修正