Snap!(BYOB) による計算論入門 *

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 Snap!(Build Your Own Blocks) 4.0[*1]を利用して計算論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.

* 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.

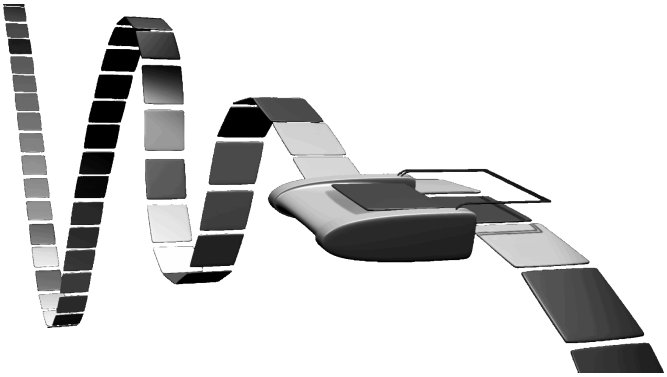
[*1] http://snap.berkeley.edu/

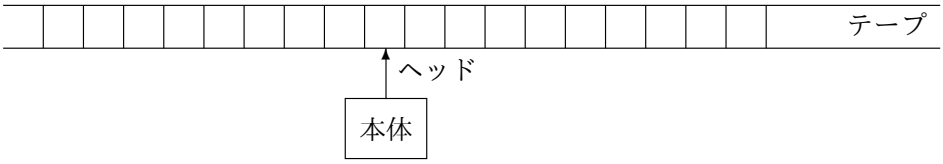
目次

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[*2] 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．下の絵は Wikipedia の Turing machine の項からの転載である．





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．

[*2] http://www.turing.org.uk/turing/

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，次の瞬間に状態を 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よぶ．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るから，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．

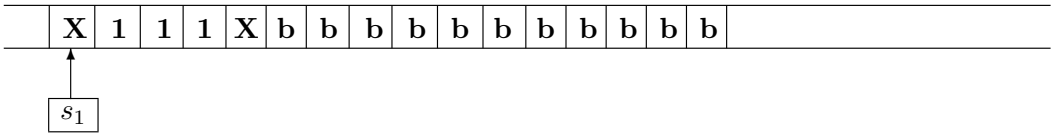
例 1.1　テープ記号が $\mathbf{1}, \mathbf{X}, \mathbf{b}$ ($\mathbf{b}$ は空白記号) で s_1, s_2, s_3, h を状態としてもち，基本操作を次のように定めたチューリング機械を考える．但し，s_1 が初期状態で h を停止状態とする．

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

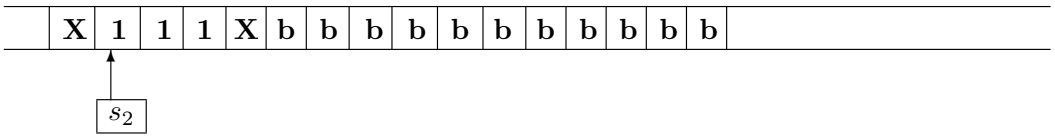
$(s_2, \mathbf{1}; \mathbf{b}, R, s_2), (s_2, \mathbf{X}; \mathbf{b}, L, s_3), (s_2, \mathbf{b}; \mathbf{b}, R, s_2),$

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

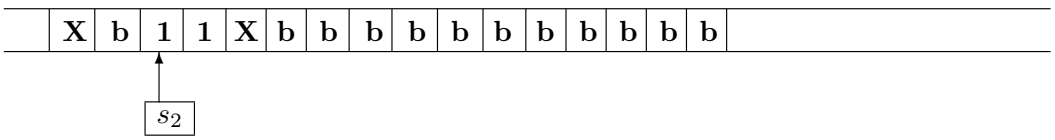
このチューリング機械に次のテープを指定されたヘッドの位置から動かしてみよう．



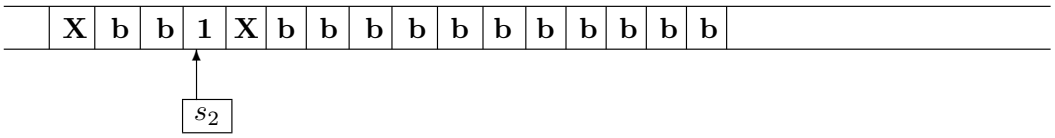
$(s_1, \mathbf{X}; \mathbf{X}; R, s_2)$ であるから，読んでいる記号 $\mathbf{X}$ はそのままで，本体の状態を s_2 に変えて，ヘッドを右に移す．

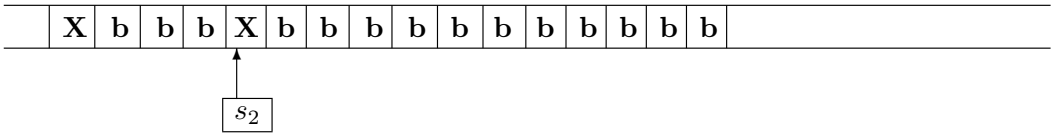


$(s_2, \mathbf{1}; \mathbf{b}, R, s_2)$ であるから，読んでいる記号 1 を $\mathbf{b}$ に書き換え，本体の状態は s_2 のままで，ヘッドを右に移す．

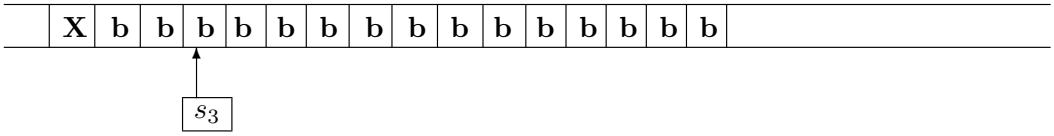


以下２ステップ同じように動く．

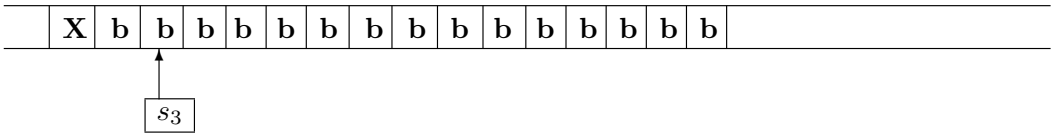




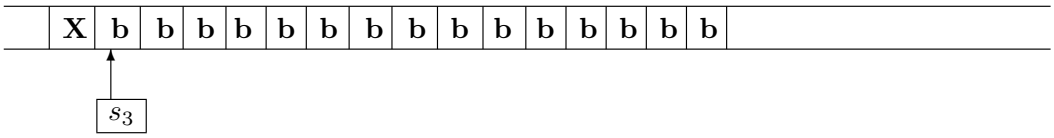
$(s_2, \mathbf{X}; \mathbf{b}, L, s_3)$ であるから，読んでいる記号 $\mathbf{X}$ を $\mathbf{b}$ に変え，本体の状態を s_3 に変えて，ヘッドを左に移す．

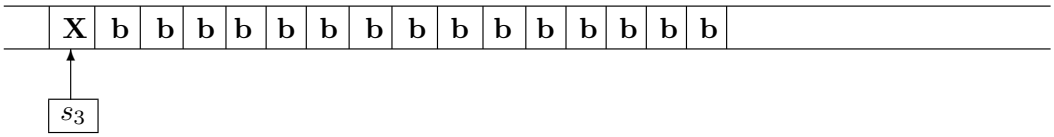


$(s_3, \mathbf{b}; \mathbf{b}, L, s_3)$ であるから，読んでいる記号 $\mathbf{b}$ はそのままで，本体の状態も s_3 のまま，ヘッドを左に移す．

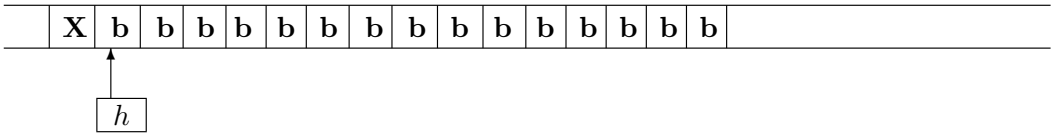


以下２ステップ同じように動く．

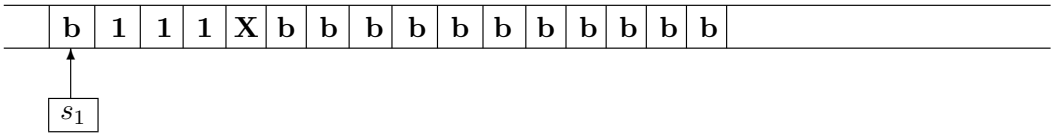




$(s_3, \mathbf{X}; \mathbf{X}, R, h)$ であるから，読んでいる記号 $\mathbf{X}$ はそのままで，本体の状態を h に変えて，ヘッドを右に移す．状態 h は停止状態であるのでチューリング機械は停止する．



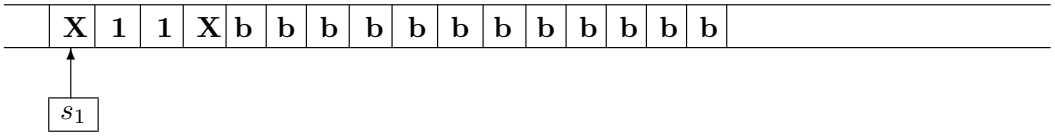
注意:チューリング機械はかならず停止するわけではない．例えば，上の例とおなじチューリング機械に次のテープを与え動作させると停止しない．



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 Snap![*3]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 Snap!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/snap/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. Snap!を起動する．
2. 本体，テープ，基本操作という名前のリストを作成する．
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．レポーターを選び，「本体の状態」と名前を付ける．「本体の 1 番目の値を返す」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．
5. 本体の状態と見ているテープ記号に一致するかどうか（今読んでいる基本操作の 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で作る．

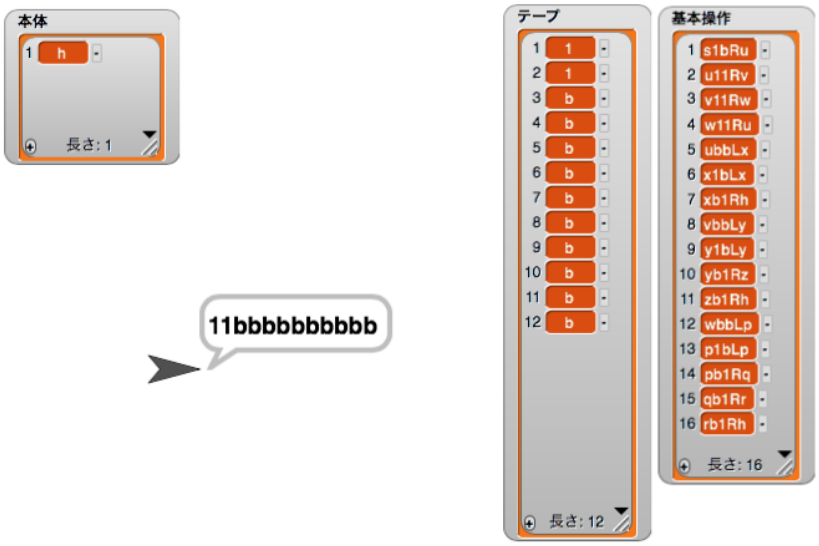
[*3] http://snap.berkeley.edu/

6. 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．

（a）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．

（b）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でなければ次の基本操作に進む．探している基本操作ならば以下のことを行う．

（c）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．



問題 1.4　問題**??**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．）

1.3　自然数上の関数を計算するチューリング機械

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[*4]とし，φ を $\mathbb{N}$ 上の ℓ 変数関数とする．

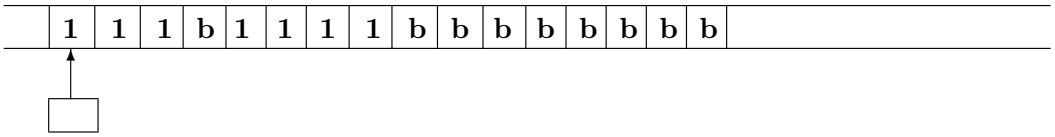
この授業ではテープ記号に $\mathbf{1}$ があると仮定して自然数を $\mathbf{1}$ の有限列として表すことにする．具体的には自然数 x のテープ上の表現 $\overline{x}$ はテープ記号 $\mathbf{1}$ が $x+1$ 個ひきつづいたます目に書かれ

[*4] 0 は自然数であるとする．

ていることで表されるものとする[*5]．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．

x と y をたしたいとき，テープ上には記号 $\mathbf{b}$ をはさんで，$\overline{x}$ すなわち $x+1$ 個の連続した $\mathbf{1}$ と $\overline{y}$ すなわち $y+1$ 個の連続した $\mathbf{1}$ を記入し，他のます目には $\mathbf{b}$ 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 $\mathbf{1}$ 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 $\mathbf{b}$ を読み込むことになる．従って

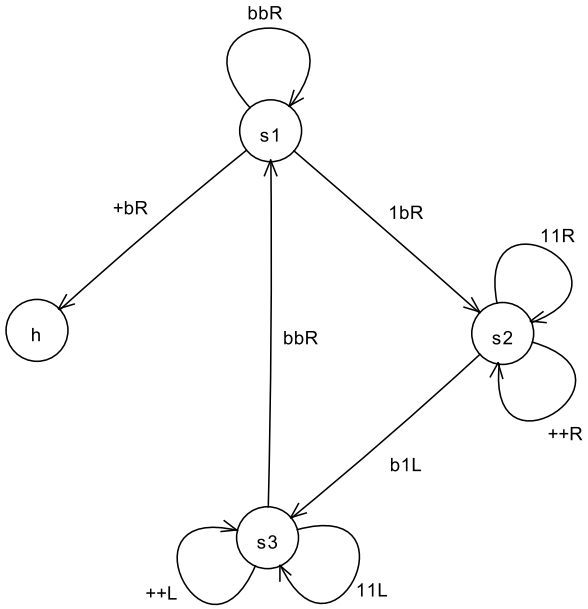
$$(s_1, \mathbf{b}; \mathbf{b}, R, s_6), (s_6, \mathbf{1}; \mathbf{b}, R, h)$$

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 $\mathbf{1}$ を $\mathbf{b}$ に書き換えて，間の $\mathbf{b}$ を $\mathbf{1}$ に書き換え，最後に $\mathbf{1}$ を 1 個 $\mathbf{b}$ 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．

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．

[*5] x 個でもよいのだが，それでは 0 が表されないので，$x+1$ 個で表すことにする．



問題 1.5　問題**??**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 1.6　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，Snap!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 1.7　$\overline{(x_1, x_2, x_3)}$ が書かれたテープの $\mathbf{1}$ 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 $\mathbf{1}$ で止まる（すなわち間にある $\mathbf{b}$ を 2 個読んで停止する）チューリング機械を作れ．

問題 1.8　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 1.9　問題**??**で定義したチューリング機械は，$\mathbf{X}$ 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 1.10　自然数上の 2 変数関数 $f(x, y)$ を次のように定義する．

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

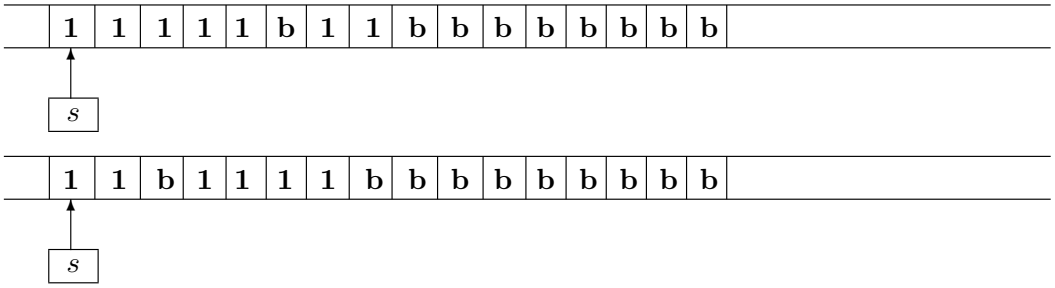
$f(x, y)$ を計算するチューリング機械を作れ．

問題 1.11　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 1.12　$\Sigma = \{0, 1\}$ 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.

1.4　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 1.13　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[*6]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．

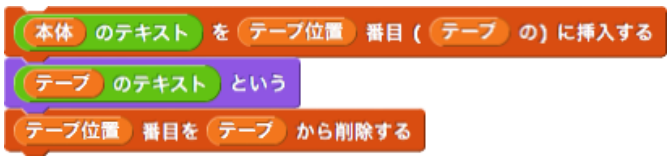
Remark:この性質はチューリング機械の決定性と呼ばれる．この性質をもたないもの，すなわち基本操作が，状態と読み込まれたテープ記号から唯一に定まらず，２つ以上の異なった動作をすることもありうるものもチューリング機械の仲間に加えて，これを非決定性チューリング機械とよぶ．非決定性チューリング機械も考慮に入れるときは，決定性を持つチューリング機械を決定性チューリング機械とよぶ．

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 1.14　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．

Snap!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．



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 $a_1 a_2 \cdots a_{i-1} q a_i \cdots a_n$ はテープ

$$a_1 a_2 \cdots a_n$$

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

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 1.15　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．

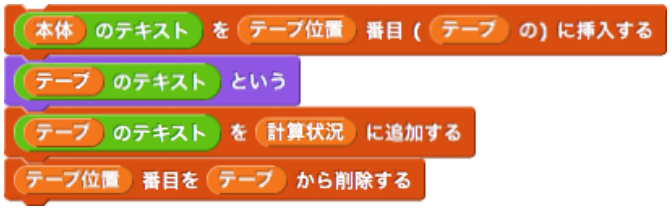
1　時点表示 α が $\tau_1 a' s a \tau_2$ であるとき，$(s, a; a'', L, s') \in \Delta_M$ ならば
　$\alpha \to_M \tau_1 s' a' a'' \tau_2$ は M の計算状況である．
2　時点表示 α が $sa\tau_2$ であるとき，$(s, a; a'', L, s') \in \Delta_M$ ならば
　$\alpha \to_M s' b a'' \tau_2$ は M の計算状況である．
3　時点表示 α が $\tau_1 s a a' \tau_2$ であるとき，$(s, a; a'', R, s') \in \Delta_M$ ならば
　$\alpha \to_M \tau_1 a'' s' a' \tau_2$ は M の計算状況である．
4　時点表示 α が $\tau s a$ 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 1.16　例**??**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．

$$s_1\mathbf{X}111\mathbf{X} \to \mathbf{X}s_2111\mathbf{X} \to \mathbf{Xb}s_211\mathbf{X} \to \mathbf{Xbb}s_21\mathbf{X} \to \mathbf{Xbbb}s_2\mathbf{X} \to \mathbf{Xbb}s_3\mathbf{bb} \to \mathbf{Xb}s_3\mathbf{bbb}$$

Snap!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 1.17　問題**??**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 1.18　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 1.19　$\varphi(x_1, \cdots, x_n)$ 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 1.20　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．

1.5 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 1.21（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??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「T」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　Snap!で計算可能な関数

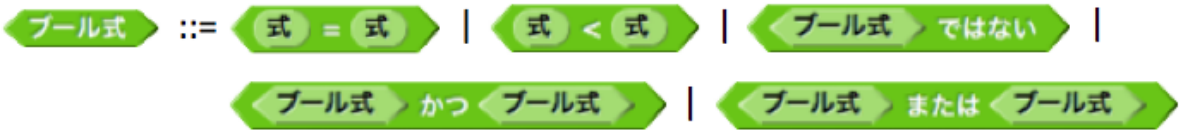
“計算可能な関数”というと一番わかりやすいイメージはそれを計算するプログラムがあるということである．関数型言語や C などを使って計算可能性を議論している文献として [?, ?] などがあるが，ここでは Snap!で計算できる関数を定義する．Snap!の機能はたくさんあるのでそれらを何でも利用していいとなると議論が面倒になるので，ここでは機能を制限することにする．

2.1　Snap!関数ブロック

以下のように同時並行的に Snap!関数ブロックを定義する．なお，定義は BNF 記法[*7]風に記述する．

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

[*7] バッカス記法とも呼ばれる．『○○::=××』は『○○とは××である』と読み、縦棒『　|』は『か』とか『または』と読む．

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．

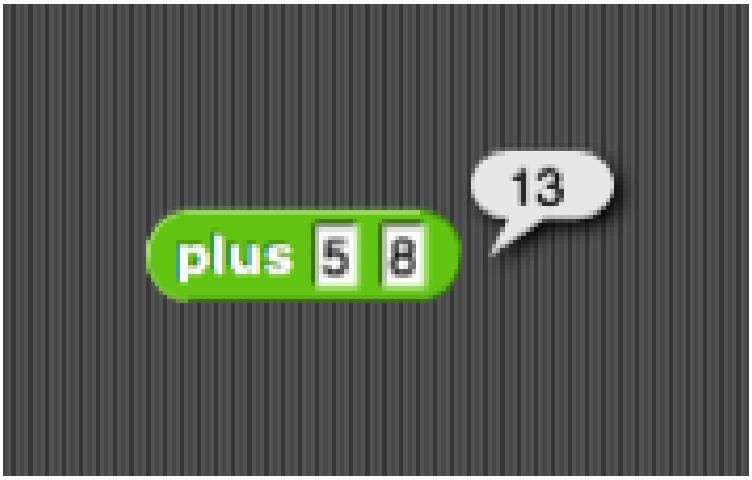
ブロックの作成は変数カテゴリー中にある「ブロックを作る」で行い，レポーター型のブロックとして作成する．ブロック名は自由である．また，ブロック内では仮引数用の変数として x1,x2,...,xn，出力変数として a を用い，a の値を返すことにする．

定義 2.1　$\varphi(x_1, \cdots, x_n)$ 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．

定義 2.2　関数 φ は，φ の値を報告する適当な Snap!関数ブロックを定めることができるとき，**Snap!で計算可能**であるという．また，その Snap!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 Snap!で計算可能である．

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 Snap!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，計算できるのは当たり前であるが，他にどのような関数が Snap!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よぶ．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 Snap!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 3.3　$\mathrm{plus}(a,b)=a+b$ 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**??**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**??**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

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

をみたすことであり，このような表現から prd を作ることは容易である．

$k'+x=(k+x)'$ 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 3.4　$\mathrm{mult}(a,b)=ab$ は prf である．

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

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 3.5　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．

$$\begin{cases}f(0)=n\\ f(k+1)=h(f(k),k)\end{cases}$$

$h(y,z)$ が prf ならば，$f(x)$ も prf である．

$\because$) 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．

$$\begin{cases} g(0,y) = g_1(y) \\ g(k+1,y) = g_2(g(k,y),k,y) \end{cases}$$

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 3.6

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

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 Snap!で計算可能である．（引き算は使えないことに注意する．）

定理 3.7

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．

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．

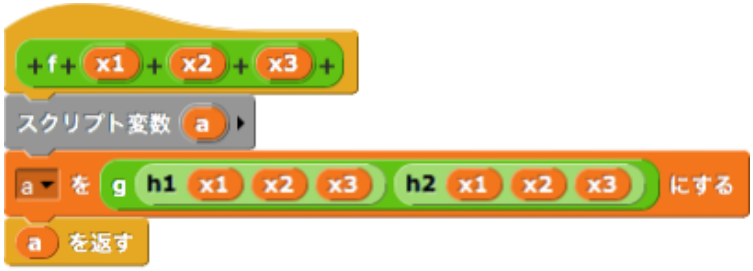
上記の定理より，すでに prf とわかっている関数から，合成や原始帰納で定義される関数は prf であるから，今後は特に prd を示さずに上記の定理を用いて，prf であることを示す．また，prf はすべて Snap!で計算可能であることが，初期関数が Snap!で計算可能であることと次の定理よりわかる．

定理 3.8

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 Snap!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 Snap!で計算可能である．

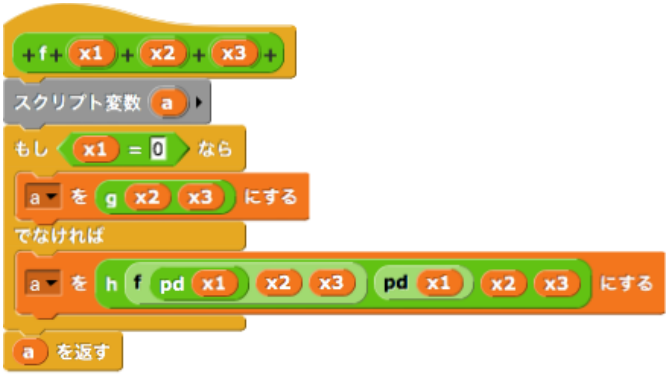


2　$g(x_1,\cdots,x_n)$ と $h(y,z,x_1,\cdots,x_n)$ が Snap!で計算可能のとき，原始帰納

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

により定義される $f(k,x_1,\cdots,x_n)$ も Snap!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．



prf であることを確認するには，原始帰納の形で書く必要があるが，Snap!の関数ブロックを定義する場合は，このように pd を使った形で考える必要があるので注意する．

例 3.9　$\mathrm{power}(a,b) = b^a$（ただし，$0^0 = 1$） 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 Snap!で計算可能である．

例 3.10　$a!$ は prf である．

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

例 3.11

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

は prf である．

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

例 3.12　$\max(a,b), \min(a,b)$ は prf である．

$\min(a,b) = b \dot{-} (b \dot{-} a)$ $\max(a,b) = (a+b) \dot{-} \min(a,b)$

例 3.13

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

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

は prf である．

例 3.14　$|a-b|$ は prf である．

$|a-b| = (a \dot{-} b) + (b \dot{-} a)$

問題 3.15　例**??**から例**??**までの関数は Snap!で計算可能であることを示せ.

例 3.16

$$\mathrm{rem}(a,b)=\begin{cases} r & if \quad a=bq+r(0\leq r<b) \\ a & if \quad b=0 \end{cases}$$

は prf である.

$$\begin{cases} \mathrm{rem}(0,b)=0 \\ \mathrm{rem}(a+1,b)=(\mathrm{rem}(a,b)+1)sg(b\dot{-}(\mathrm{rem}(a,b)+1))+(\mathrm{rem}(a,b)+1)\overline{sg}(b) \end{cases}$$

例 3.17

$$[a/b]=\begin{cases} q & if \quad a=bq+r\ (0\leq r<b) \\ 0 & if \quad b=0 \end{cases}$$

は prf である.

$$\begin{cases} [0/b]=0 \\ [a+1/b]=([a/b]+\overline{sg}(b\dot{-}(\mathrm{rem}(a,b)+1)))sg(b) \end{cases}$$

例 3.18　$\phi(\overline{x},0)+\phi(\overline{x},1)+\cdots+\phi(\overline{x},z-1)$ 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 3.19　$a=b$ 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 3.20　$a\leqq b$ は prp である. $\varphi_\leq(a,b)=\overline{sg}(a\dot{-}b)$

例 3.21　$a < b$ は prp である．$\varphi_<(a,b) = sg(b \dot{-} a)$

定理 3.22　$P(x_1, \cdots, x_n)$ 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 3.23　$x + y \leqq z$ は prp である．

定理 3.24　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 3.25　$P(x_1, \cdots, x_n, y)$ 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 3.26　$a|b$ (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 3.27　$Pr(a)$ (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 3.28　$P(x_1, \cdots, x_n, y)$ が prp のとき，$\mu y_{y<z} P(x_1, \cdots, x_n, y)$ は prf である．

例 3.29　P_i ($(i+1)$ 番目の素数をとる関数) は prf である．

問題 3.30　$f(x_1, x_2, z) = \mu y_{y<z}(x_1|y \wedge x_2|y)$ 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．)

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

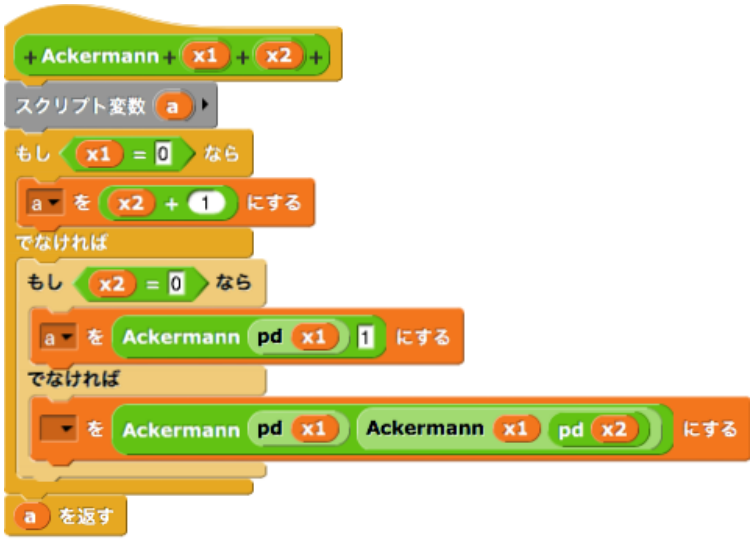
例えば $A(1,2)$ を計算してみよう．

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

となる．他の場合も上の定義式に従って計算することができる．[*8]

問題 3.31　$A(2,3)$ 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．

アッカーマン関数は Snap!で計算可能である．実際，次のように関数ブロックを定義すれば良い．



従って，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．従って，“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．

$$g(x) = \text{“}f(x,y) = 1 \text{ をみたす最小の } y \text{ の値”}$$

[*8] 但し，計算の手間は大変で，例えば $A(2,3) = 9$ を手計算でするのも一仕事である．

この $g(x)$ は，x に対し $f(x,y)=1$ 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 $f(3,n)=1$ 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 1 になるので，一番最初に値が 1 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．

$g(x)$ を定義する際に，“・・・をみたす最小の y”という表現を用いたが，これを μ 作用素で表すことにする．有界 μ 作用素の有界性を取り除いたものである．$g(x)=\mu y[f(x,y)=1]$ とすると，与えられた x に対して，$f(x,y)=1$ となる y が存在しなければ，$g(x)$ の値は未定義となる．従って，$g(x)$ が部分関数となる場合もありうることになるが，$f(x,y)$ が条件 $\forall x\exists y[f(x,y)=1]$ をみたすならば，$g(x)$ は全域的な関数となる．このことを踏まえて以下のように帰納的関数を定義する．

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よぶ．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) とよぶ.

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**とよぶ.

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 3.32　$A(i, x)$ 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 Snap!で計算可能な関数であり，Snap!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の提唱（定立）とよばれ多くの人から受け入れられている．

チャーチの提唱（定立）

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

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, 廣瀬健著, 朝倉書店

[5] 計算の基礎理論, 細井勉著, 教育出版