チューリング機械開発システムの作成 (1)

宮　下　英　明

目　次

1　チューリング機械の存在論

1.1 存在論の問題化

チューリング機械の理論では，チューリング機械の存在論は閑却してよい主題である。しかし，チューリング機械を実現しようとするときには，これの存在論が問題化する。

1.2 解釈としての“チューリング機械”

“チューリング機械”は解釈である。それは，

平成3年3月29日受理

(1) 機械を，テープとヘッドと内部状態でなるものとし，
(2) それの動作を，“状態推移図／動作表”に従うものとする
解釈である。言い換えると，

“《一つの状態推移図／動作表を規則としてこれに従って動作する機械》のように解釈された機械を，この状態推移図／動作表によって定義されるチューリング機械と呼ぶ”

ということである。この意味で，

《一つの状態推移図／動作表を規則としてこれに従って動作するようにつくられた機械》

は，この状態推移図／動作表によって定義されるチューリング機械である。

特に，“チューリング機械”は“テープ，ヘッド，内部状態，そして状態推移図／動作表で定義される抽象的な機械”――〈仮想機械〉――ではない。

実際，“仮想機械”といえども，機械である。それは機械として仮想できるものでなければならない。そして，“テープとヘッドと内部状態の三つだけでなる機械”は，仮想できないのである。機械として仮想できるためには，これらの要素は他の部品によって結合され，連絡していなければならない。また，起動システムを閑却して機械を仮想することも，不可能である。

“状態推移図／動作表で定義される機械”――“状態推移図／動作表としての機械”――の仮想も不可能である。実際，われわれはそのような様を想像し得ない。

“チューリング機械”の概念には，“仮想機械”という含意はない。“チューリング機械”と解釈されたものがチューリング機械であり，チューリング機械は実在の機械であり得る。そしてまた，チューリング機械を仮想することもできる――仮想したそのチューリング機械は，仮想機械である。

1.3 潜在性と顕在化

“一つの動作表はチューリング機械を定義している”と考えるとき，われわれは，チューリング機械を潜在性として導入していることになる。〈機械〉は，テープが装着されて起動されたのち，テープ記号に対する反応として，顕在化する。これが〈動作〉である。

一般に，動作する以前の〈機械〉を対象化しようとするとき，対象化の形態は，〈機械＝潜在性〉となる。“動作以前”は，“潜在性”であるしかない。動作表によるチューリング機械の定義は，このようなものである。それは，動作以前の機械を対象化する方法に則っている。

この方法は，つぎの一般的方法の枠内にある：

《現象を生成的現象と見なし，“現象以前”を生成の潜在性として対象化する。明示的には，〈生成規則〉として対象化する。》(注)

動作表の意義は，“生成規則”である。われわれは，運動生成の潜在性として，そして明示的には運動生成規則として，チューリング機械を導入したのである。

（註）　例えば，“言語現象以前”の言語を，文法（文生成規則）として対象化する。

1.4 “起動”の概念の欠如

チューリング機械には，“起動(スイッチオン)”の概念がない。

テープへの“入力”によって起動すると考えることはできない。何故なら，何も書かれていないテープもまた，“入力テープ”になるからである。

テープの“装着”を起動と考えることはできよう。しかし，《テープ装着のつぎにくるものとしての“起動”の概念が，はじめから欠如している》と見る方が，自然である。

“テープの装着”の概念化は，同時に，テープが装着されていないチューリング機械”の概念化である。“テープの装着されていないチューリング機械”は，動作表のことではない。われ

われはチューリング機械を〈身体〉と考える。そして，動作表は身体の解釈であり，身体そのものではない。

チューリング機械の起動を，われわれはテープが既に装着されている時点でのものとして考える。この起動は，身体の状態（“内部状態”ではない）の一変——一つの状態から別の一つの状態への変化——として考えねばならない。コンピュータで言うと，電気回路としてのそれのオフからオンへの状態変化である。チューリング機械には，この意味の“起動”の概念が欠如しているのである。

チューリング機械に“起動”の概念が欠如していることは，理論の上の問題にはならない。“起動”は，この理論では閑却してよい主題である。しかし，チューリング機械を実現しようとするときには，“起動”を導入しなければならなくなる。

1.5 “テープ”の意味

テープの意味は，“入力に使うテープ”（入力装置）ではない。それはコンピュータのメモリ・ボードに相当する。

“テープ入力”とは，起動時のテープに対する一つの特別な読み方（思いを込めた読み方）である。これに対しては，“メモリへの直接書込み”の読みの方が当たっていることになるが，いずれにしても，チューリング機械の主題においてわれわれは“テープへの書込み”の概念を必要としない。実際われわれは，テープを所与として扱うことになる。

1.6 テープ記号

われわれは，言い回しの簡単のため，

(1) テープの“無記号”を表わす記号

(2) テープ記号

をあわせて“テープ記号”と呼ぶことにする。

なお以下では，$\_$ を“無記号”の記号とする。

1.7 基底動作

いま，チューリング機械を一つ固定して考え，これの“基底動作”の概念をつぎのように定義する。即ち，動作の集合 M において条件：

(1) 任意の動作を，M の要素の組み合わせで実現することができる；

(2) M のどの要素も，M の他の要素を組み合わせるやり方ではつくることができない．

が満たされているとき，M に属する各動作を基底動作と呼ぶ。

このとき，各テープ記号 x に対する

（x）ヘッドが指しているコマに，記号 x を書き込む

動作は，基底動作である。そしてこれの他に基底動作となるものは，つぎの三つである：

（r）ヘッドを右に一コマ移動する；

（l）ヘッドを左に一コマ移動する；

（e）ヘッドが指しているコマに記号が書かれていれば，これを消去する．

1.8 チューリング機械リアライザ

“チューリング機械”に対して，“チューリング機械をその上に現出させることのできる機械”の概念が立つ。これを“チューリング機械リアライザ”と呼んでおく。“チューリング機械リアライザ”は，“チューリング機械”のメタ概念と見なせる。

“チューリング機械リアライザ”も，“チューリング機械”と同様，解釈である。一つの機械に対して，チューリング機械か否か，チューリング機械リアライザか否かが，決まるのではない。

なお，チューリング機械リアライザをチューリング機械として作成できたとき，それは“汎用チューリング機械”というものになる。

2 チューリング機械の表現

2.1 〈チューリング機械＝潜在性〉の表現

2.1.1 〈チューリング機械＝潜在性〉の関数表現

いま，つぎの集合を考える：

(1)　“テープ記号の全体”と読まれる集合S
(2)　“内部状態の全体”と読まれる集合Q
(3)　“(基底) 動作の全体”と読まれる集合A

〈チューリング機械＝潜在性〉は,“内部状態”,“ヘッド (の動作)”,“テープ (記号)”の語を用いて〈物理的機械〉のように述べられるが,理論的には,上のS, Q, Aの集合に関する関数として定式化できる——そしてこのときの関数には,つぎのようなものが考えられる:

(1)　読み:
“(内部状態, 記号) ⟼ (動作, 内部状態)”の関数:

$$Q \times S \longrightarrow A \times Q$$

(2)　読み:
“内部状態⟼(記号⟼(動作, 内部状態))”の関数

$$Q \longrightarrow (A \times Q)^S$$

(3)　読み:
“内部状態⟼(動作, 記号⟼内部状態)”の関数:

$$Q \longrightarrow A \times Q^S$$

(1)はクラシカルな定式化である。
(2)は(1)に準ずるものであり,
《記号に応じて(動作, 内部状態)に分岐し,
動作して, つぎの内部状態にいく》
ということで, switch-do-goto タイプ——略して, SD タイプ——と言える。
(3)は,
《動作してから, 記号に応じて内部状態に分岐する》
ということで, do-switch-goto タイプ——略して, DS タイプ——と言える。

われわれは, 以下, 〈チューリング機械＝潜在性〉の表現関数としては(2)と(3)の二つを専ら考えていくことにする。

2.1.2 動作表

〈チューリング機械＝関数〉の一つの表現式が“動作表”である。そこで, 機械に対する“チューリング機械”の解釈の一つは, それの動作を一つの動作表に従うものと見なすことである。

〈チューリング機械＝関数〉にSD, DSの二つのタイプが考えられることに対応して, 動作図にもSD, DSの二タイプが考えられる。

クラシカルな動作表はSDタイプのもので, つぎのようになる:

	s_1 ···· s_k
⋮	••••
q_i	a_1q_1 ···· a_kq_k
⋮	••••

($s_1, \cdots, s_k$: テープ記号の全体
q_n : 内部状態　a_n : 基底動作)

これに対するdsタイプの動作表を, われわれは, つぎのように書くことにする:

		s_1 ···· s_k
⋮	⋮	••••
q_i	a	q_1 ···· q_k
⋮	⋮	••••

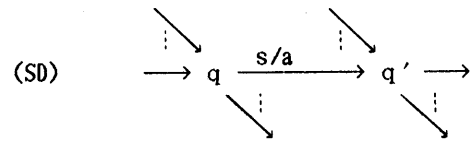
(a: 基底動作)

2.1.3 状態推移図

〈チューリング機械＝関数〉の表現としては, 動作表の他に“状態推移図”が一般的である。

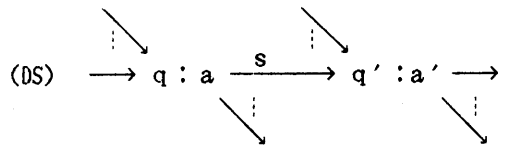
関数の二つのタイプSD, DSに対応して, 状態推移図にもSD, DSの二タイプが考えられる。

クラシカルな状態推移図は, SDタイプのものであり, 一つの状態推移が, 内部状態q, q', テープ記号s, 動作aに対する図式:



で示される。

これに対し, DSタイプの動作表には, つぎの形の状態推移図が応ずることになる:



2.1.4 動作推移図

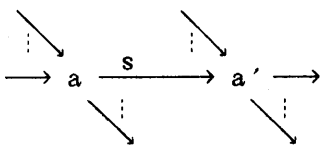
状態推移図における〈内部状態〉の意義は，“ネットワークのノード”である——これに尽きる。そして“異なる内部状態”が“異なるノード”として示されているところの状態推移図においては，〈内部状態〉の表記は本質的に過剰なことである。

実際，〈内部状態〉の表示は，無しで済ませることができる。それは，〈内部状態〉をノードとする状態推移図を，〈動作〉をノードとする“動作推移図”に書き換えるという形で，実現できる。そして状態推移図において考えられたタイプ分け——SD と DS ——は，動作推移図においては解消する。

DS タイプの状態推移図を動作推移図に変える規則は簡単であり，それは

《ネットワークのノードになっている“q : a”を，“a”に換える》

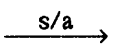
である。§2.1.3の図（DS）は，つぎの図に変わる：



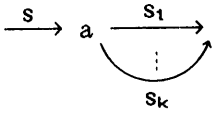
SD タイプの状態推移図に対しては，つぎのようにする：

(1) 何もしない動作 null を導入して，全ての内部状態を null に書き換える。

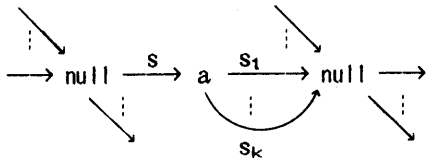
(2) 表記：



を，

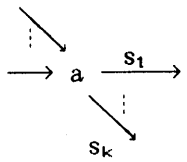


に書き換える——ここで，s_1……，s_k はテープ記号の全体。§2.1.3の図（SD）は，つぎの図に変わる：



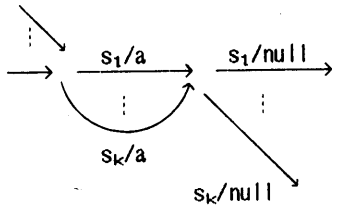
このように，SD，DS のどちらのタイプの状態推移図も動作推移図に変え得るわけであるが，逆に，動作推移図から SD と DS の両方のタイプの状態推移図を復元することも可能である。

実際，つぎのようにする。即ち，動作推移図でのノード：



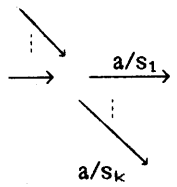
（a は動作，s はテープ記号）に対し，

$(1)_{SD}$ SD タイプの状態推移図の復元では，これを一旦



に書き換える。

$(1)_{DS}$ DS タイプの状態推移図の復元では，これを一旦



に書き換える。

(2) 空となったノード全体に，互いに異なる記号を書き込み，これらを〈内部状態〉と読む。

2.2 〈チューリング機械＝状態推移〉の表現

“働いているチューリング機械”は，ヘッドの位置（＝テープのコマ），内部状態，テープの上の記号列の三つ組――簡単に，“状態”と呼ぶことにする――の変化である。

いま，テープを固定して考え，“テープのコマ全体”の集合をNとする。このとき，チューリング機械のヘッドの基底動作は，

“ヘッドのいまの位置 $\longmapsto$
　（ヘッドが上書きするテープ記号，
　ヘッドの移動先）”

の読みで関数：

$$N \longrightarrow S \times N$$

と見なせる。よって，動作全体の集合Aは，$(S \times N)^N$である。

また，逐次変化するテープ記号列の各々は，“テープのコマ
　$\longmapsto$ そこに書かれている記号”
の読みで，関数：

$$N \longrightarrow S$$

と見なせる。そしてこのとき，テープ記号列全体の集合はS^Nである。

テープ記号列t，テープのコマn，テープ記号sの組（t, n, s）に
〈tのコマnに書かれているテープ記号を
sに書き換えて得られるテープ記号列〉
を対応させる関数を，Cとする。

さて，現象としてのチューリング機械は，状態推移：

（いまの　　（つぎの
ヘッドの位置，　　ヘッドの位置，
内部状態，　$\longrightarrow$　内部状態，
テープ記号列）　　テープ記号列）

を初期状態から最終状態（停止状態）まで順に追っていくことで，集合$N \times Q \times S^N$の要素の有限列――即ち，自然数の系$\mathbb{N}$の或る切片$K = [0, e]$から集合$N \times Q \times S^N$への関数――として表現できることになる。

この状態推移は，〈チューリング機械＝潜在性〉の表現関数を用いて，つぎの形の表現を得る（ここで，動作を関数：$N \longrightarrow S \times N$と定めたことに留意すること）：

(1) SDタイプの

$$F : Q \longrightarrow (A \times Q)^S$$

に対し

(n, q, t)
$\longrightarrow$ (pr_2 ((pr_1 (F (q) (T (n)))) (n)),
　pr_2 (F (q) (T (n))),
　C (t, n,
　　pr_1 ((pr_1(F(q)(T(n))))(n))
　)

(2) DSタイプの

$$F : Q \longrightarrow (A \times Q)^S$$

に対し

(n, q, t)
$\longrightarrow$ (pr_2 ((pr_1 (F (q))) (n)),
　(pr_2 (F (q))) (
　　T (pr_2 ((pr_1 (F (q))) (n)))
　　),
　C (t, n, pr_1 ((pr_1 (F (q))) (n))
　)

実際，初期状態として一つの状態（n_0, q_0, t_0）を一旦与えれば，上の規則にしたがって，状態推移の連鎖が自動的に生成される。これが，

《チューリング機械にテープをセットし，
初期条件を与えて起動すれば，チューリン
グ機械は動作表に従って自動的に動く》

ということの定式化である。

3　意味論

テープの上の記号列は，無意味である。

例えば，二進数の掛け算を行なうチューリング機械の実行していることが，二進数の掛け算と読まれる必然性はない。“二進数の掛け算”は，あくまでも，われわれの意味づけによる。――一方，チューリング機械は，記号列の変形に“アルゴリズム”の意味づけをしている。

4　チューリング機械定義プログラム

4.1 動作表に対する“プログラム”の解釈

動作表は，“内部状態”を“アドレス”と読み直すことで，そのまま“実行プログラム”と見なすことができる。――但し，チューリング機械リアライザが受容する実行プログラム（その意味は“チューリング機械定義プログラム”）として。

例えば，三つの内部状態0，1，2と二つのテープ記号 _，1の上のSDタイプ動作表：

	_	1
0	r0	l1
1	r1	l2
2	**	**

は，つぎのプログラムに解釈される：

```
0 switch {
        case _ ;
          r ;    goto 0 ;
        case 1 ;
          1 ;    goto 1 ;
  }
1 switch {
        case _ ;
          r ;    goto 1 ;
        case 1 ;
          1 ;    goto 2 ;
  }
2 halt ;
```

逆に，このような形式の動作プログラムで動く機械は，“チューリング機械”となる。

4.2 チューリング機械定義プログラムの二形式

状態推移図および動作推移図に対しては，“チューリング機械の設計図”としての使用が考えられる。即ち，

《状態推移図／動作推移図をチューリング機械定義プログラムに翻訳する》

という使い方である。

このとき，状態推移図から自ずと導かれるプログラム形式と，動作推移図から自ずと導かれるプログラム形式は，異なる。

実際，状態推移図の翻訳では〈内部状態〉にプログラムの〈アドレス（ラベル）〉が対応し，〈状態推移〉に〈アドレス（ラベル）へのジャンプ〉が対応する。結局，そのプログラムは，アセンブリプログラムやBASICプログラムのようになる。

一方，動作推移図に直結するプログラム形式は，〈アドレス（ラベル）〉無しで済ませるという制約から，《モジュールによる構成》となり，Cプログラムのような“構造化された”プログラムをもたらす。

5　チューリング機械の開発

5.1 “プログラミング”の観点からの“チューリング機械開発”の発想

動作表に対して“チューリング機械定義プログラム”の見方がされるようになるとき，“チューリング機械の開発”が“プログラミング”の観点から発想されるようになる。

5.2 機械語－高級言語，コンパイル

動作表を直接書くことをチューリング機械のプログラミングとするとき，意図通りに動作するチューリング機械をプログラミングすることは，至難の業である。その作業は非常に煩瑣で，見通しの効かないものになる。

動作表を〈プログラム〉として見るとは，それを〈チューリング機械リアライザが受容する実行プログラム〉として見るということであった。そこでいま，動作表の記述を，チューリング機械リアライザが直接受容できる言語――“機械語”――による記述と見て，動作表作成の困難は機械語に因ると考えてみよう。即ち，“動作表の記述言語”を《リアライザには受容し易いが，われわれには扱い難い機械語》として対象化し，これに，《われわれに扱いやすい言語》と

して“高級言語”を対置する。

そして，つぎのことを主題化する：

《チューリング機械の記述を高級言語で行ない，この記述を機械語に翻訳（コンパイル）することで，動作表を得る》

5.3 動作推移図とDSタイプの選択

チューリング機械を開発する場合，状態推移図を使うよりは動作推移図を使う方が有利である。

実際，〈アドレス（ラベル）へのジャンプ〉をたどる形でプログラムを書く／読むよりは〈処理〉をたどる形でプログラムを書く／読む方が，われわれの生理に合っている。前者の場合，プログラムはスパゲティプログラムになるが，このようなプログラムはわれわれの生理に合わない。

チューリング機械は，最終的に動作表で表現することになるが，動作表には既にSDとDSの二つのタイプが考えられている。したがって，どちらのタイプを採用すべきかがこのとき問題になる。そして，“状態推移図よりも動作推移図の方が有利”としたいまの場合，DSタイプを選択する方が有利となる。

何故なら，動作表は同タイプの状態推移図の直接の翻訳としてつくられるが，状態推移図への動作推移図の書き直しでは，既に見たように，状態推移図のタイプがDSである場合の方が簡単になるからである（§2.1.4）。

5.4 チューリング機械開発システムの作成

“チューリング機械開発システムの作成”として，具体的につぎのことを行なう：

(1) 動作推移図のダイレクトな翻訳としてチューリング機械定義プログラムを書くことを可能にする言語を作成する。

この言語は，動作表を“チューリング機械リアライザの受容する実行プログラム”と見なし，その記述を“機械語”による記述と見なすとき，機械語に対する高級言語として位置付けられるものである。その意味で，この節では，この言語を単に“高級言語”と呼ぶことにする。

(2) 高級言語で書かれたチューリング機械プログラム（ソースプログラム）からDSタイプの動作表を生成するツール——コンパイラ——を作成する。但し，モジュール別開発を可能にするため，コンパイラはリロケータブル・オブジェクトモジュールを生成するものとし，リロケータブル・オブジェクトモジュールのリンカをコンパイラと併せて作成する。

(3) DSタイプの動作表を，二つの形式で実現する。一つは，チューリング機械リアライザに合った仕様のもの（BIN形式），そしてもう一つは，われわれに合った——書きやすい／読みやすい——仕様のもの（TBL形式）である。

コンパイラ＆リンカは，BIN形式とTBL形式の両方の動作表を生成する（但し，TBL形式の動作表の生成は，オプション）仕様とする。

(4) BIN／TBL形式のDSタイプ動作表を受容するチューリング機械リアライザを作成する。

(5) DSタイプ動作表のTBL形式に対応して，SDタイプ動作表のTBL形式を定める。

(6) DSタイプのBIN動作表およびTBL動作表とSDタイプのTBL動作表の間のコンバータを作成する。

5.5 チューリング機械の開発手順

“チューリング機械開発システム”を使用したチューリング機械の開発は，以下のような流れになる。

(1) 高級言語で，所期のチューリング機械の定義プログラム——ソースプログラム——を書く。

(2) ソースプログラムをコンパイラにかけてリロケータブル・オブジェクトモジュールを得る。

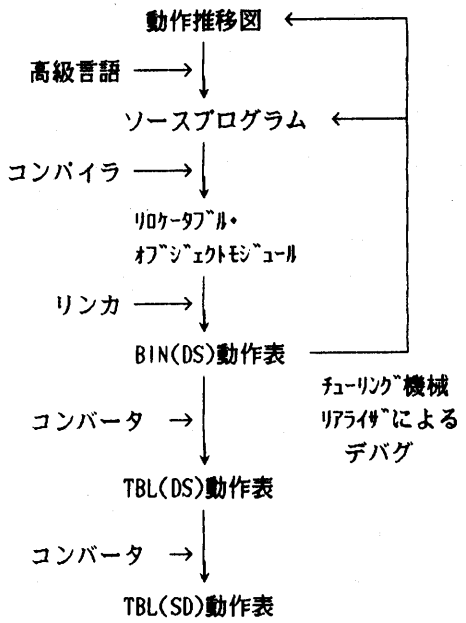
(3) リロケータブル・オブジェクトモジュールをリンカにかけて，BIN形式のDSタイプ動

作表を得る。

(4) チューリング機械リアライザに動作表とテープを与え，リアライザの上にデモンストレートされるチューリング機械の動作を見て，プログラムを——あるいはさらに動作推移図に遡ってこれを——デバグする。

(5) リンカのオプションによって，あるいは，BIN 形式の DS タイプ動作表をコンバータにかけることによって，TBL 形式の DS タイプ動作表を得る。

(6) コンバータによって，DS タイプの BIN／TBL 動作表から SD タイプの TBL 動作表を得る。



6　高級言語

6.1 高級言語の構成

われわれは，以下のものを高級言語の要素として考える：

(1) プリプロセッサ指示語

(2) チューリング機械連結手続き記号

(3) スキップ記号（注釈記号）

(4) 基底チューリング機械の名

(5) モジュールとしてのチューリング機械の名（モジュール名）

ここで，基底動作に対応するチューリング機械を，基底チューリング機械と呼ぶ。

基底チューリング機械以外のチューリング機械は，既成のチューリング機械を連結するという形でつくられるが，このことは，

《モジュールの中で基底チューリング機械，あるいは別のモジュールを呼び出す》

という形で記述できる。チューリング機械連結手続き記号は，この記述のためのもの，即ち，チューリング機械の連結の仕方を記述するための記号である。

6.2 高級言語 M

チューリング機械記述用高級言語として，われわれは以下に述べる仕様の言語を，“M 言語”——“Mini C 言語”——の名で導入する。

6.2.1 基底チューリング機械の名

右移動，左移動，記号消去の基底チューリング機械の名を，それぞれそれぞれ r，l，e とし，予約語とする。

テープの“無記号”を表わす記号を _ と書き，_ を除くテープ記号 x に対し，x を書く基底チューリング機械を x そのもので表わす。

6.2.2 プリプロセッサ指示語

プリプロセッサ指示語は，

#symbol

#define

の二つである。

#symbol は，テープ記号の指定に使用する。即ち，_ を除くテープ記号（各一文字）を

#symbol　$s_1 s_2 \cdots\cdots s_n$

（s_i は 1 文字）

の書式によって指定する。空白は不可——特に，指定文字間に空白があってはならない。空白，タブ，改行あるいはスキップ記号に出会うまで

の文字列が，テープ記号の列挙と判断される。

#defineは，C言語のものと同じである。

6.2.3 連結手続き記号

チューリング機械連結手続き記号には，つぎのものがある：

(1) {, }, (,)
(2) ;
(3) if( ), elseif( ), else
(4) while, break
(5) for( )
(6) return, exit

以下に述べることの外は，C言語の文法に準ずる：

(1) whileは while{……} の形で用い，Cでの while(1){……} と同じ。
(2) if( ), elseif( )の( )には，テープ記号が入る。ヘッドが指すコマにこの記号があるとき，真。
(3) for( )の( )には，ループ繰り返しの回数が入る。
(4) exit;は，mainモジュールを強制終了させるコマンド。

6.2.4 スキップ記号

C言語のスキップ記号/*, */をM言語でも採用する。即ち，/*と*/ではさんだ記号列は読み込まれない。特に，注釈が書ける。

ネストした書き方も許される。

6.2.5 非公式記号

基底機械fに対する

```
for(k){
        f;
}
```

の簡略表記としてf ^ k;を許す。但し，M言語では非公式のものとする。

実際この表記は，つぎのように公式に実現できる：

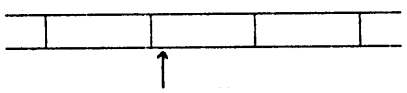
```
#define  (x)^(n)    for(n){x}

main
 {
      .....
       (f;)^(n)
      .....
 }
```

6.3 Mプログラムの例

ここで，Mプログラムの例として，テープ記号が“無記号”記号の_と1のみであるときの二進数の積を求めるプログラムを示す。

テープ記号が_と1のみであるとき，ヘッドの位置を如何に特定するかが問題になる。ここでは，“フィールド”の考えを導入することで，この問題をクリアする。即ち，フィールド幅を固定し，ヘッドの最初の位置をもとにテープ上に区画を想定する。そしてこの区画を約束として，記号の処理を考えるのである：



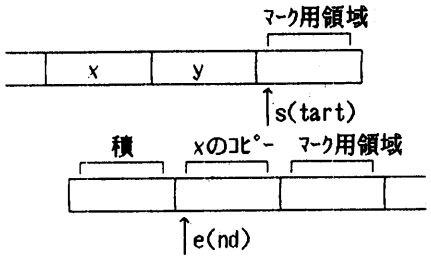
ここでは，一区画を1バイトとする。さらに，二進数データは先頭のビットを空けた7ビットとする——先頭の1ビットは，フラグあるいはマークとしての使用を予定して空けておく。

以下のプログラムに現われるpは，《1を書き込む》基底機械である。M言語の文法に従い，最終的に1（イチ）で置換すべきものである。——1（イチ）とl（エル）が紛れやすいので，1をpで表わしている。

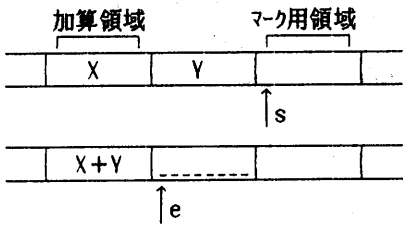
なお，プログラムは，構成的明証性を優先した結果，オーバヘッドの大きいものになっている。

```
/*****************************************
    各機械の仕様
mul
  ヘッドの左の二つの区画に記された二進数の
```

積を求める。
ヘッドの右の四つの区画が，ワーキングエリアとして使用される：



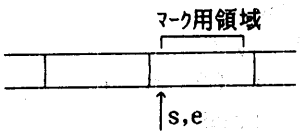
add
ヘッドの左の二つの区画に記された二進数の積を求める。



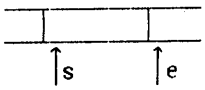
ante : ANTEcedent
ヘッドの左に書かれている数を，それの前者の数に書き換える．
最後にテープ記号を書き換えた位置が，ヘッドの最終位置．

succ : SUCCessor
ヘッドの左に書かれている数を，それの後者の数に書き換える．
最後にテープ記号を書き換えた位置が，ヘッドの最終位置．

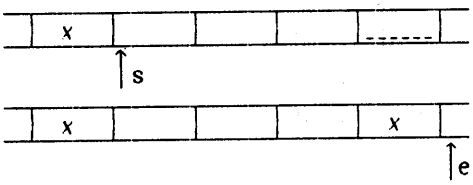
isnil
ヘッドの左の一区画がすべて空白かどうかを調べる。
ヘッドの最終位置は，最初の位置と同じ．
ヘッドの左の一区画がすべて空白ならば，ヘッドの最終位置に 1 を立てる．そうでなければ，空白．



cr : Clear Right
ヘッドの位置を左端とする一区画の記号を消去する．
ヘッドの最初の位置の右一区画先が，ヘッドの最終位置



kr4 : Copy Right4区画先
ヘッドの左 1 区画を，右 4 区画先の区画にコピーする
ヘッドをコピー先区画の右に置いて終了



lnb : Left Not Blank
記号（非空白記号）に出会うまで左移動
**********/

```
#symbol 1
#define U     8
#define U2   16
#define U3   24
#define U4   32

main
 {
    mul;
}

mul
 {
    while {
```

```
        isnil ;
        if ( _ ) {
            r ^ U2 ; cr ; l ^ U ;
            break;
        }
        l ^ U ; kr4 ; l ^ U3 ;
        ante ;
        r ^ U ; lnb ; r ^ U3 ;
        add ;
        l ^ U2 ;
    }
}

add
 {
    while {
        isnil ;
        if ( _ ) {
            l ^ U ;
            break ;
        }
        ante ;
        r ^ U ; lnb ; l ^ U ;
        succ ;
        r ^ U2 ; lnb ;
    }
}

ante {
    while {
        l ;
        if (1) {
            e ;
            break ;
        }
        p ;
    }
}

succ
 {
    while {
        l ;
        if ( _ ) {
            p ;
            break ;
        }
        e ;
    }
}

isnil
 {
    p ;             /*  終了判定用の    */
                    /*  マークを設定    */
    while {
        l;
        if ( ! _ ) {
            r ^ U ;
            if ( _ ) {
                lnb ;
            }
            break ;
        }
        r ^ U ;
            if(1) {
                e ;
                break ;
            }
        l ^ U ;
    }
}

cr
 {
    for (U) {
          e ; r ;
    }
}
```

```
kr4
 {
        r ^ U3 ; p ; r ^ U ; l ;
                        /*  終了判定用の   */
                        /*  マークを設定   */
    while {
        l ^ U4 ;
        if(1) {
            r ^ U4 ;
            if(1) {  /*  いまが         */
                     /*  ラストだった   */
                break ;
            }
            p ;
        }
        else {
            r ^ U4 ;
            if(1) {  /*  いまが         */
                     /*  ラストだった   */
                e ;
                break ;
            }
        }
        l ;
    }
  r ^ U ;
}

lnb
 {
    while {
        l ;
        if ( ! _ ) {
            break ;
    }
}
```

7　コンパイル，リンク

7.1 ソースファイルの作成

高級言語によるチューリング機械の記述――部品とするチューリング機械の連結手順の記述――を，テクストファイルの作成として行なう。このファイルが，コンパイルに関するソースファイルとなる。

7.2 分割コンパイル & リンク

ソースファイルから動作表ファイルを生成する方法として，われわれは以下のような〈分割コンパイル & リンク〉方式をとることにする：

(1)　分割コンパイル

ソースファイルの中の各モジュールに対し，

(1.1) 構造木（パース・トリー）をつくり，

(1.2) そしてこれを，リロケータブル・オブジェクトモジュールへと翻訳する(註)；

(2)　リンク

(2.1) メインモジュールの先頭から出発して，モジュール呼び出しの連鎖のとおりに，再帰的にオブジェクトモジュールをつないでいく；

(2.2) この再帰的処理は，〈分岐処理〉か〈基底チューリング機械の呼び出し〉にまで遡行する；この段階で動作表の一つの行が確定される；

(2.3) 以上の再帰的処理が終了するとき，所期の動作表が得られている。

（註）われわれは，リロケータブル・オブジェクトモジュールを一旦ファイル（オブジェクトファイル）におとすことにする。但し，ファイルにおとすかどうかということは，“分割コンパイル & リンク”の概念にとって本質的なことではない。

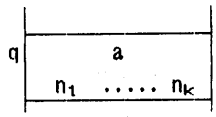
7.3 オブジェクトモジュールの仕様

リロケータブル・オブジェクトモジュールは，モジュールとしての機械の表現であるが，それ

は機械のDSタイプ動作表の行

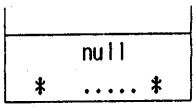
		s_1 ····· s_k
q	a	n_1 ····· n_k

(s_i：テープ記号，a：動作，q，n_i：内部状態)の表現になるセル

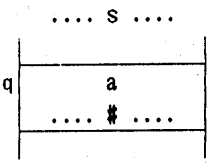


を動作表の通りに連ねたものとする。

われわれの仕様では，オブジェクトモジュールの末尾は，何もしない動作nullに対する



となり，テープ記号sに対するexitへの分岐は，



のように書かれる。

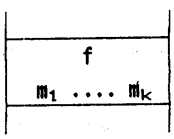
7.4 リンク

リンクは，モジュール(機械)mainのオブジェクトモジュールの最初のアドレスから出発して，モジュール（機械）の中からのモジュール（機械）呼び出しの連鎖の通りに，オブジェクトモジュールをつないでいく。

それのイメージ——実際の作業内容（§14）ではない——は，以下のようになる。

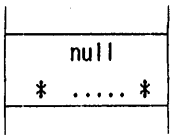
モジュールfの呼び出しに対して，

(1) fを呼び出しているセル



の下に，fを実現しているセル連結を挿入する。このとき，アドレス（内部状態名）に重複が生じないように，fのセルのアドレスをつけ直す。

(2) fの末尾のセル：



の＊を，順にm_1，……，m_kに書き換える。

(3) fを呼び出しているセルの“f”を“null”に書き換え，m_1，……，m_kすべてをfの先頭セルのアドレスに書き換える。

8　チューリング機械リアライザ

8.1 実行プログラムとしてのチューリング機械リアライザ

われわれは，一つの実行プログラムによって，コンピュータ上にチューリング機械を実現する。そしてこの実行プログラムを，〈チューリング機械リアライザ〉と位置づける。(この実行プログラムが動いているコンピュータ全体も，チューリング機械リアライザである。)

チューリング機械のこの実現は，“コンピュータによるチューリング機械のシミュレーション”ではない。実際，“チューリング機械”というものが存在しているわけではない——“チューリング機械”は解釈であり，“チューリング機械”と解釈されたものがチューリング機械である。

8.2 チューリング機械リアライザの機能

ここでは，チューリング機械リアライザの機能をつぎのように考える：

(1) チューリング機械の設定を，動作表ファイルの読み込みという形で受け付ける。

(2) テープ（メモリ）の設定を，テープファイルの読み込み，あるいはテープ記号列の直接入力の受容という形で，受け付ける。

(1)と(2)が揃った段階で，起動する前のコンピュータに相当するチューリング機械が実現

されたことになる。

(3) キー入力に反応して，チューリング機械を起動する。いままでは見えていなかったチューリング機械が，〈テープ入力に対するチューリング機械の反応〉という形で顕在化する。