数学教育におけるコンピュータ教育(2)
——チューリング機械開発システムの作成——

宮下英明*・松沢正規**

目　次

＊宮下　英明　　金沢大学教育学部
＊＊松沢　正規　　金沢市高尾台中学校

11 高級言語

11.1 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番号

(3) スキップ番号（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。

11.2 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。

11.2.1 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。

11.2.2 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 $s_1 s_2 \cdots s_n$

（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。

11.2.3 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。

11.2.4 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。

11.2.5 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。

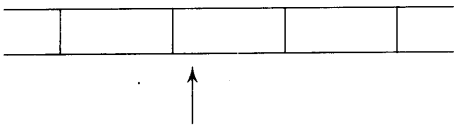
実際この表記は，つぎのように公式で実現できる：

```
#define  (x)^(n)  for(n){x}
main
 {
    .......
    (f;)^(n)
    .......
 }
```

11.3 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：

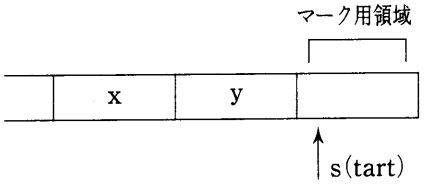


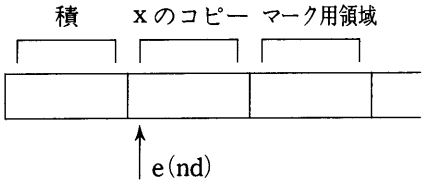
ここでは，一区画を8バイトとする。さらに，二進数データは先頭のビットを空けた7ビットとする——先頭のビットは，フラグあるいはマークとしての使用を予定して空けておく。

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。

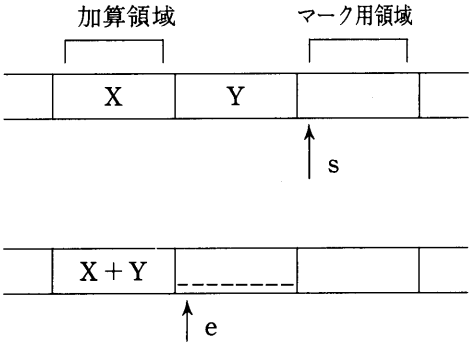
```
／＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊
        各機械の仕様
mul
    ヘッドの左の二つの区画に記された二進数の積を求める。
    ヘッドの右の四つの区画が，ワーキングエリアとして使用される：
```





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。

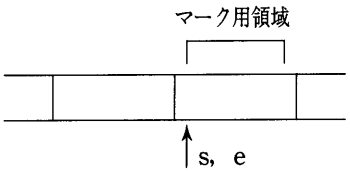
最後にテープ記号を書き換えた位置が，ヘッドの最終位置。

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。

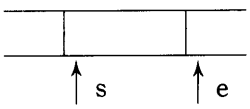
ヘッドの左の一区画がすべて空白ならば，ヘッドの最終位置に1を立てる。そうでなければ，空白。



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。

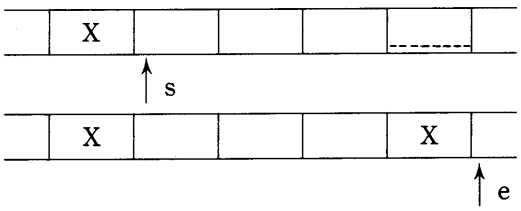
ヘッドの最初の位置の右一区画先が，ヘッドの最終位置



kr 4： Copy Right 4区画先

ヘッドの左一画を，右4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 Sot 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 ; kr 4 ; 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( l ) {
            e ;
            break ;
        }
        p ;
    }
}

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

isnsl
 {
    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 ;
    }
}

kr 4
 {
    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 ;
    }
}
```

12 コンパイル，リンク

12.1 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。

12.2 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）つくり，

(1.2) そしてこれを，リロケータブル・オブジェクトト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。

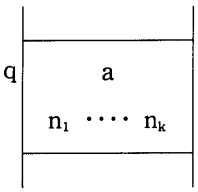
12.3 オブジェクトモジュールの仕様

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

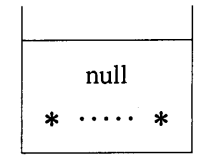
（s_l：テープ記号，a：動作，q, n_l：内部状態）

の表現になるセル

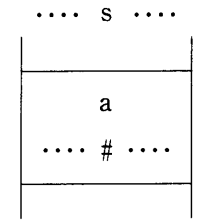


を動作表の通りに連ねたものとする。

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。

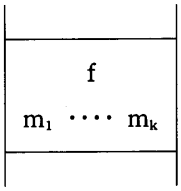
12.4 リンク

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。

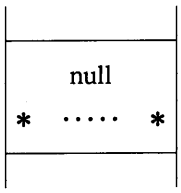
モジュールfの呼び出しに対して，

(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。

13 コンパイラ

われわれは，コンパイラとして

TC. EXE

を用意する。TCは，拡張子Mのソースファイル（Mファイル）からそれと同名で拡張子OBJのオブジェクトファイル（OBJファイル）を生成する。

13.1 ソース，オブジェクトファイルの仕様

13.1.1 ソースファイル

ソースファイルは，拡張子がMのテクストファイル（Mファイル）である。

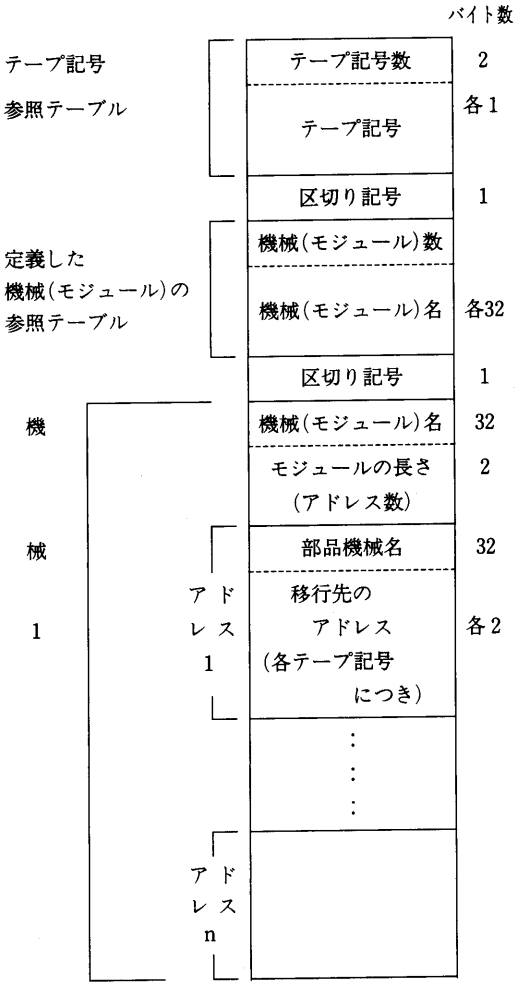
Mプログラムは，プリプロセッサ行を定めるのに空白，タブ，改行コードが意味をもつということを除けば，フリーフォーマットである。即ち，これの読み込みにおいては，空白，タブ，改行コードはスキップされる。

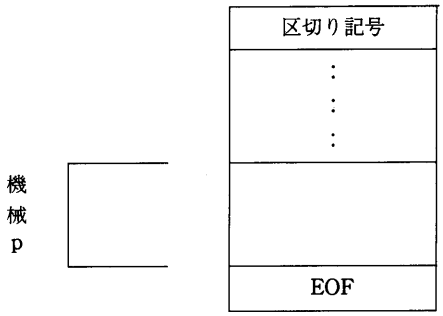
モジュール(機械)名は，32文字まで許される。それより長い名は，32文字でカットする(エラーとはしない)。

一つのMソースプログラムには，複数のモジュール(機械)を記述できる。

13.1.2 オブジェクトファイル

オブジェクトファイルは，拡張子がOBJのバイナリファイル（OBJファイル）である。その仕様を，つぎのように定める：





但し，exitに対応する#を－2，モジュールの終了に対応する＊を－1，とそれぞれ定める。

13.2 コンパイラTC

TC起動の書式は，

TC 〈Mファイル名〉

である。〈Mファイル名〉では，拡張子を省略できる。

TCは，以下のことを行なう：

(1) Mファイルを読み込みモードで開き，つぎに同名のOBJファイルを書き込みモードで開く。

(2) Mプログラムの#symbol行を読み，〈記号参照テーブル〉の形でOBJファイルに書き込む。

(3) Mファイルに記述されている各モジュールにf対し，以下のことを繰り返す：

(3.1) fを読み込む。

(3.2) fのパース・トリーtを作成する。

(3.3) tをもとにfのリロケータブル・オブジェクトモジュールに作成し，OBJファイルに書き込む。

(3.4) fに対するこの作業において新たに確保したメモリを開放する。

(4) MファイルのOBJファイルを閉じる。

13.3 コンパイル作業の詳細

Mソースファイルに記述されたモジュールからリロケータブル・オブジェクトモジュールを生成する方法を，つぎのMプログラムを例にして述べる：

```
#symbol  1   /*記号は__と1のみ*/
 a
 {
    if(__)
        b:
    if(1) {
        c;
        while {
            if(__)
                break;
            while {
                if(1)
                    exit;
                d;
                if(1)
                    break;
            }
            e;
        }
    }
    else
        f;
    g;
 }
```

13.3.1 整　形

コンパイラは，先ず，ソースプログラムに対して整形の処理をする。即ち，“何もしない”チューリング機械nullを，つぎの規則に従って挿入する：

(1) if, while, break, return, exitの前に挿入する；

(2) 二つの連続した } の間に挿入する；

(3) モジュールを閉じる } の前に挿入する；

(4) 略記号；（これはnull；の略記号と見なされる）をnull；に換える。

nullは，コンパイル・リンクの処理のルーチンを簡単にするために導入されるダミーのチューリング機械である。

例のMプログラムは，つぎのようになる：

```
a
 {
    null;                          0
    if(__) {
        b;                         1
    }
    null;                          2
    if(1) {
        c;                         3
        while {
            null;                  4
            if(__)
                null;              5
                break;
            }
            null;                  6
            while {
                null;              7
                if(1) {
                    null;          8
                    exit;
                }
                d;                 9
                null;              10
                if(1) {
                    null;          11
                    break;
                }
                null;              12
            }
            e;                     13
        }
        null;                      14
    }
    else {
        f;                         15
    }
    g;                             16
    null;                          17
}
```

なお，コンパイラは整形と併行して，

(1) 括弧の対応，

(2) 位置的に予約語であるべき語の綴り，

(3) elseif，elseがこれに対応するifをもっているかどうか

のチェックも行なう。

13.3.2 パース

ソースプログラムを整形すると，コンパイラはつぎにソースプログラムの文法解析(パース)に入る。これは，ソースプログラムの構造木の作成で終わる。

パースは，つぎの二つの作業を併行して行なう形で進められる。

トークンへの分解

構造木の作成

この作業は，ソースプログラムの頭から順になされる。

13.3.2.1　トークンへの分解

トークンの画定は，つぎの規則による：

《{，　}，；のうしろで区切る》

例では，トークンへの分解はつぎのようになる：

a {	1	}	21
null ;	2	d ;	22
if(_) {	3	null ;	23
b ;	4	if(1) {	24
}	5	null ;	25
null ;	6	break ;	26
if(1) {	7	}	27
c ;	8	null ;	28
while {	9	}	29
null ;	10	e ;	30
if(_) {	11	}	31
null ;	12	null ;	32
break ;	13	}	33
}	14	else {	34
null ;	15	f ;	35
while {	16	}	36
null ;	17	g ;	37
if(1) {	18	null ;	38
null ;	19	}	39
exit ;	20		

13.3.2.2　構造木の作成

構造木は，トークンを一つ画定する毎につぎの処理をしていくことで，最終的に得られる：

(1)　トークン構造体をメモリに一つ確保する

(2)　この構造体に，トークンの情報を書き込む

(3)　ここまでに作成されている構造木に，この構造体を規則にしたがってリンクする。

13.3.2.3　トークン構造体への書き込み

トークン構造体は，つぎのように定義される：

id　(char)

name　⟶ char 型配列

left　⟶ トークン構造体

right　⟶ トークン構造体

idには，トークンのアイデンティティ：

if, elseif, else, while, break, exit, return, MAIN, MACHINE, NIL

を書き込む。ここで，

(1)　MAINは，本モジュール(以下，“メインモジュール”と呼ぶ) のアイデンティティ

(2)　MACHINEは基底動作および本モジュールが呼び出しているモジュール (以下，単に“モジュール”と呼ぶ) のアイデンティティ，

(3)　NILは } のアイデンティティ。

nameには，

(1)　トークンが基底動作あるいはモジュールのときは，それの名

(2)　if, elseifならば，その引き数

を書き込む。

13.3.2.4　トークン構造体のリンク

トークン構造体のリンク処理は，再帰的になされる。この処理では，新しいトークン構造体Sに対し，Sをリンクするターゲットのトークン構造体Tが，その都度指定されている。

リンクの規則は，つぎのようになる：

(1)　Tのトークンが if, elseif, else, while で，Sのトークンがこの文の中にあるときは，Tの leftにSをつなげる。

(2)　(1)以外の場合は，Tのrightに Sをつなげる。

ターゲット更新の規則は，つぎのようになる：

(1)　Sのトークンが if, elseif, else, while のと

きは，いまのネストでターゲットをSに更新するとともに，再帰のネストを一つ進めた処理を初期ターゲットSで始める。

(2) SのトークンがNILのときは，再帰のネストを一つ戻す。

(3) (1), (2)以外の場合は，ターゲットをSに更新する。

例では，図1に示した構造木が生成される。

13.3.3 オブジェクトモジュールの作成

ソースプログラムをパースしてこれの構造木を作成すると，つぎはオブジェクトモジュール作成の段階になる。作業は，

(1) 構造木の先頭のトークンをターゲットとすることから出発し，

(2) ターゲットのトークン構造体を参照して，規則の定める処理を行なう

という形で進められる。結果として，所期のオブジェクトモジュールが得られる。

13.3.3.1 MT構造体

オブジェクトモジュールの作成にあたり，先ず，動作表の“一行”（モジュールの“一項”）：

〈内部状態〉

〈機械〉

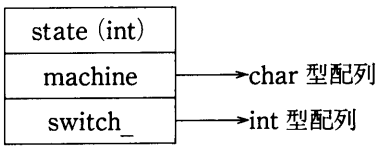
〈テープ記号 s_1 に対し推移する内部状態〉

.....

〈テープ記号 s_k に対し推移する内部状態〉

(s_1, ····, s_k は，テープ記号の全体)

を書き込むMT（module-term）構造体を定義する：



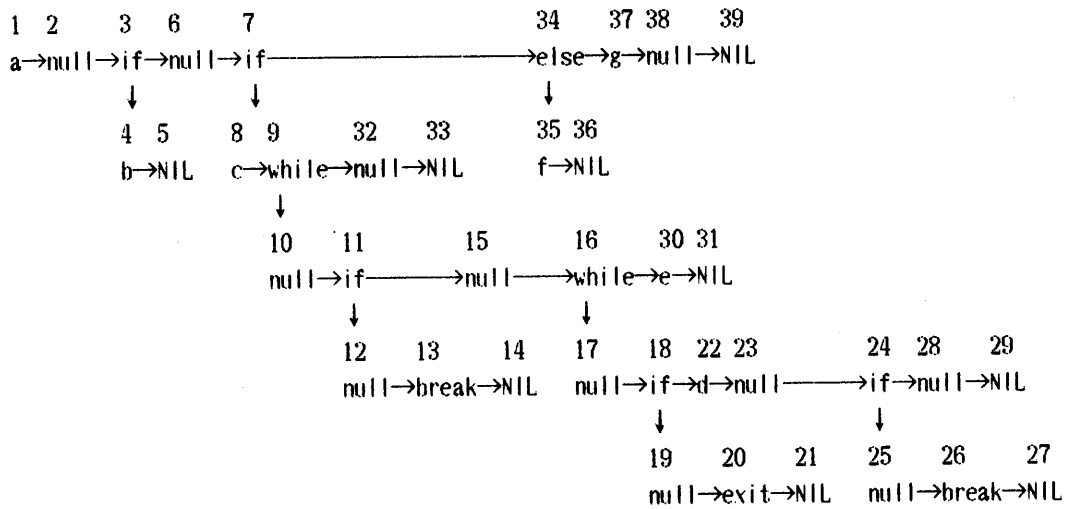


図 1

即ち,

(1) stateには,〈内部状態〉の記号(アドレス)としての整数が書かれる。

(2) machineは,

〈stateに書かれている内部状態に対応する機械名〉

を書いた配列をポイントする。

(3) switch_は,

〈stateに書かれている内部状態においてmachineに書かれている動作をした後,各テープ記号に対して推移する内部状態(整数値)〉

を書いた配列をポイントする。

13.3.3.2 道 具

この作業のために,スタック:

(1) ネスト情報保存スタック

(2) ペンディングアドレス保存スタック

(3) ループ開始アドレス保存スタック

(4) ループブレークアドレス保存スタック

(5) IF引数スタック

およびフラグ:

(1) ループブレークフラグ

(2) IFブレークフラグ

そして,新しく確保したMT構造体のstateに書き込むべき整数を保存する変数:

new_state (初期値0)

を,道具として用意する。

13.3.3.3 処理規則

処理規則はつぎのようになる。即ち,ターゲットのトークンのidが

(1) MAINならば,

(1.1) ネスト情報保存スタックにMAINをプッシュする。

(1.2) rightがポイントするトークンをターゲットトークンとする。

(2) MACHINEならば,

(2.1.) MT構造体をメモリに1つ確保する。以下,new_state=nであるとして,

(2.2) stateに,nを書く。

(2.3) nameに,ターゲットトークンのnameをコピーする。

(2.4) switch_配列を,RETURNを表わす値(ここでは−1)で埋める(初期化)。(以下,この値が入っている箇所を“未記入”と言い表わす)。

(2.5) ループブレイクフラグまたはIFブレイクフラグが立っているとき,

(2.5.1) ペンディングアドレス保存スタックにプッシュされているすべてのアドレスをポップする;

(2.5.2) ポッズされた各アドレスmに対し,mをstate値とするMT構造体のswitch_配列の未記入箇所すべてに,nを書き込む。

(2.5) そうでないとき(これは,直前のトークンがMACHINEかifの場合である)は,ペンディングアドレス保存スタックのラストインをポップし,このアドレスをstateとするMT構造体に対して,以下の処置をする:

(2.5.1) IF引数スタックにテープ記号(番号)が入っているときは,これらをすべてポップし,各々のテープ記号について,これと対応するswitch_配列の箇所にnを書き込む。

(2.5.2) IF引数スタックが空のときは,

switch_配列を n で埋める。

(2.6) ペンディングアドレス保存スタックに n をプッシュする。

(2.7) new_state を n＋1 にする。

(2.8) right がポイントするトークンをターゲットトークンとする。

(3) break ならば,

(3.1) ペンディングアドレス保存スタックからアドレスを一つポップし，ループブレークアドレス保存スタックにこれをプッシュする。

(3.2) right がポイントするトークンをターゲットトークンとする。

(4) exit ならば,

(4.1) ペンディングアドレス保存スタックからアドレスを一つポップし，これが state 値になっている MT 構造体に対し，switch_配列の未記入箇所すべてに EXIT を表わす値（ここでは－2）を書き込む。

(4.2) right がポイントするトークンをターゲットトークンとする。

(5) return ならば,

(5.1) ペンディングアドレス保存スタックからアドレスを一つポップする。

（switch_配列は，初期化において RETURN を表わす値で埋めたので，いまポップしたアドレスを state 値とする MT 構造体に対する処理は既に済んでいることになる。）

(5.2) right がポイントするトークンをターゲットトークンとする。

(6) if ならば,

(6.1) ネスト情報保存スタックに IF をプッシュする。

(6.2) ターゲットトークンの name に書き込まれている IF 引数を，全て IF 引数スタックにプッシュする。

(6.3) ペンディングアドレス保存スタックのラストインを一つの自動変数 if_start_address に保存し，ネストを一つに進めて，left がポイントするトークンをターゲットトークンとする再帰処理に入る。

(6.4) 再帰処理から戻ったら,

(6.4.1) IF ブレイクフラグを立てる。

(6.4.2) if_start_address をペンディングアドレス保存スタックにプッシュする。

(6.4.3) ターゲットトークンを，right がポイントするトークンにする。

(7) elseif ならば,

(7.1) ネスト情報保存スタックに IF をプッシュする。

(7.2) ペンディングアドレス保存スタックのラストインを参照し，if_start_address に保存する。（この elseif に対応する if を right でポイントするトークンを x とするとき，x に対して確保された MT 構造体の state 値が，いまのアドレスに当たる。）

(7.3) このとき IF ブレイクフラグが立っていることになるが（(6.4.1) 参照），これを下ろす。

(7.4) ターゲットトークンの name に書か込まれている ELSEIF の引数を，全て IF 引数スタックにプッシュする。

(7.5) ペンディングアドレス保存スタックのラストインを自動変数 if_start_address に保存し，ネストを一つ進めて，left がポイントするトークンをターゲットトークンとする再帰処理に入る。

(7.6) 再帰処理から戻ったら，

(7.6.1) IF ブレイクフラグを立てる。

(7.6.1) if start address をペンディングアドレス保存スタックにプッシュする。

(7.6.3) ターゲットトークンを，right がポイントするトークンにする。

(8) else ならば，

(8.1) ネスト情報保存スタックに IF をプッシュする。

(8.2) このとき IF ブレイクフラグが立っていることになるが，これを下ろす。

(8.3) ネストを一つ進めて，left がポイントするトークンをターゲットトークンとする再帰処理に入る。

(8.4) 再帰処理から戻ったら、IF ブレイクフラグを立てる。

(8.5) ターゲットトークンを，right がポイントするトークンにする。

(9) while ならば，

(9.1) ネスト情報保存スタックに while をプッシュする。

(9.2) ループ開始アドレス保存スタックに，ペンディングアドレス保存スタックのラストインをコピーする。

(9.3) ネストを一つ進めて，left がポイントするトークンをターゲットトークンとする再帰処理に入る。

(9.4) 再帰処理から戻ったら，ループブレイクフラグを立てる。

(9.5) ターゲットトークンを，right がポイントするトークンにする。

(10) NIL ならば，

ネスト情報保存スタックから内容をひとつポップする。それが

(10.1) MAIN なら，

作業終了

(10.2) if ならば，

一つ前のネストに戻る。

(10.3) while ならば，

(10.3.1) ループ開始アドレス保存スタックからアドレスを一つポップする。これを q とする。

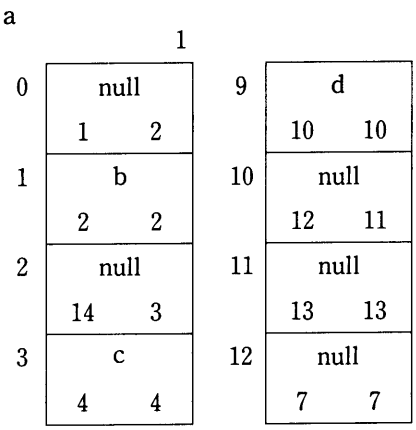
(10.3.2) ペンディングアドレス保存スタックに入っている全てのアドレスをポップし，各アドレスについて，これを state 値とする MT 構造体の switch_配列の未記入部分全てに q を書き込む。

(10.3.3) ループブレイク保存スタックに入っているアドレスを，＞q である限りポップし，全てペンディングアドレス保存スタックにプッシュする。

(10.3.4) 1 つ前のネストに戻る。

13.3.3.4 結　果

例の構造木からは，つぎのリロケータブル・オブジェクトモジュールが生成される：



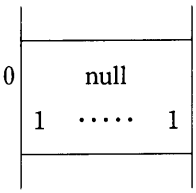
4	null	5 6	13	e	4 4
5	null	13 13	14	null	16 16
6	null	7 7	15	f	16 16
7	null	9 8	16	g	17 17
8	null	# #	17	null	* *

ここで#は，exitに対応して移行先アドレスが無い状態であり，＊はこのモジュールの終了ということで，移行先アドレスが無い状態である。

13.3.4 圧　縮

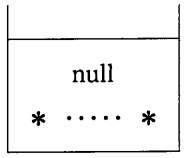
オブジェクトモジュールが得られると，つぎにこれの圧縮の処理に入る。即ち，これからnullのセルをつぎの規則によって削除する：

(1) アドレス0が

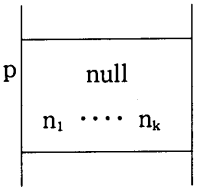


であるときは，これを削除する。

(2) 末尾の

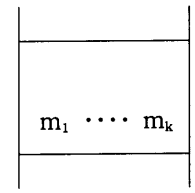


を除く全てのnullに対し，つぎの処理を行なう。即ち，



に対し，

（2-1）アドレスpを含む全ての枠



の $m_l = p$ を n_l に書き換える。

（2-2）このnullの枠を削除する。

この処理の結果，先のリロケータブル・オブジェクトモジュールは，つぎのように圧縮される：

a

	—	1			
0	null	1 3	13	e	16 #
1	b	15 3	15	f	16 16
3	c	16 #	16	g	17 17
9	d	9 13	17	null	* *

なおこの例では，d，cが呼び出されることはない。これらの圧縮は，“最適化”の主題になる。

13.3.5 オブジェクトファイルの出力

オブジェクトモジュールを，オブジェクトファイルに出力する。このとき，オブジェクトファイルの仕様を満たすよう，適当に情報を付加する。

14 リンカ TL

リンカとして

TL. EXE

を用意する。

TLは，OBJファイルから拡張子BINの動作表ファイル（BINファイル）を生成する。

14.1 TL起動の書式

TL起動の書式は，

TL [-t] 〈ファイル名〉+…+ 〈ファイル名〉，〈動作表ファイル名〉

である。ここで，

(1) -tは，TBLファイルを同時に生成することを指定するオプション。

(2) 〈ファイル名〉は，リンクするOBJファイルの名で，拡張子を省略したもの。

(3) 〈動作表ファイル名〉は，生成するＢＩＮファイルの名で，拡張子を省略したもの。

14.2 TLの作業内容

TLは，以下のことを行なう：

14.2.1 初期作業

(1) コマンドラインで指定されたすべてのOBJファイルを読み込みモードで開き，また，同じくコマンドラインで指定された動作表ファイルを，書き込みモードで開く。

(2) 先頭に指定されたOBJファイルからテープ記号参照テーブルを読み込み，残りのOBJファイルのテープ記号参照テーブルを，これと照合する。違いがあれば，エラーとし，開いていたファイルを閉じて作業を終了する。

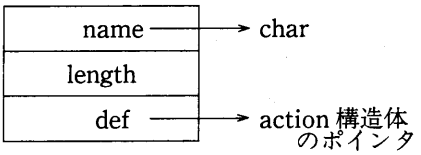
以下，テープ記号がつぎのk個であるとする：

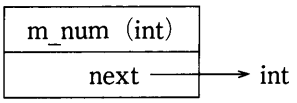
$s_0 = \_\_,\ s_1,\ \cdots\cdots,\ s_{k-1}$

14.2.2 〈作業テーブル〉の作成——ステップ1

各OBJファイルにある〈定義機械参照テーブル〉をもとに，第1階〈作業テーブル〉を以下のように作成する：

(1) machine構造体とaction構造体を





のように定義する。

(2) 基底機械の数と，null登録のための1，そして各OBJファイルで定義されている機械（モジュール）の数の合計sを求め，machine構造体のポインタs個分のメモリを，配列の形で確保する。

(3) 機械の登録として行なうことは，

(3.1) 各機械に対して，その都度machin構造体のメモリを確保する；

(3.2) nameにファイル名をコピーする；

(3.3) lengthに機械の（モジュールとしての）長さを書き込む。

このとき，メモリ確保に先立って，既に同じ機械名が読み込まれていないかを，ここまでにつくられている〈作業テーブル〉に対してチェックする。そして同じ機械名があれば，エラーとし，開いているファイルを閉じて，作業を終了する。

(4) 機械の登録は，基底機械と null の登録から始める。つぎに，各 OBJ ファイルに対し，〈定義機械参照テーブル〉から機械名を一つずつ読み込み登録する。

14.2.3 〈作業テーブル〉の作成——ステップ2

引き続き，各 OBJ ファイルから機械（モジュール）の内容を読み込みながら，〈作業用テーブル〉を以下のように作成する：

(1) 読み込んだ〈機械(モジュール)名〉と mach [p] ->name がポイントする文字列が一致するような p を特定する。

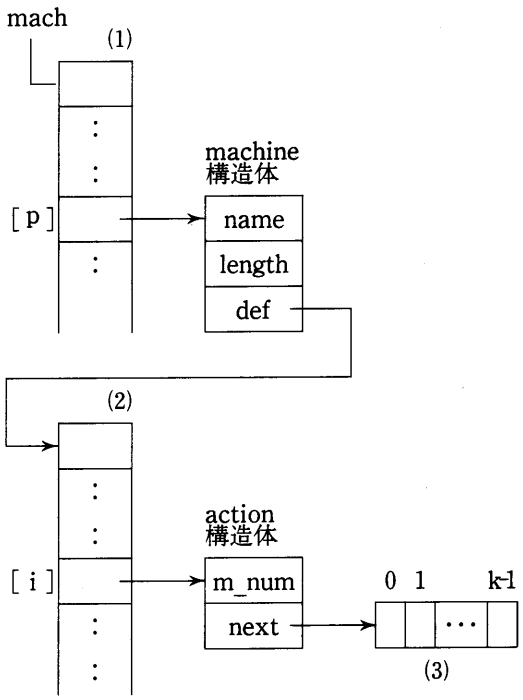
(2) 〈モジュールの長さ〉n を読み込み，action 構造体のポインタ n 個分のメモリを，配列の形で確保する。そして mach [p] ->def がこれの先頭のアドレスを指すようにする。

(3) (mach [p] ->def) [i-1] までの処理が終わっていて，〈部品機械名〉がさらに読み込めるとき，action 構造体の一つ分のメモリを確保して，(mach [p] ->def) [i] がこれの先頭のアドレスを指すようにする。さらに，整数型の配列をテープ記号数の長さだけ確保し，((mach [p] ->def) [i] ->next がこれの先頭のアドレスを指すようにする。

(4) 読み込んだ〈部品機械名〉と mach [q] ->name がポイントする文字列が一致するような q を特定する。この q を ((mach [p] ->def) [i]) ->m_num に書き込む。

(5) 引き続き〈移行先アドレス〉を読み込み，(_を含めた) k 個のテープ記号に対応する移行先アドレスを，((mach [p] ->def) [i]) ->next が指す配列に書き込む。

以上の処理によって，〈作業用テーブル〉が以下の形で実現される：



ここで，

(1) の配列の番号 p は，mach [p] ->name に保存されている機械（モジュール）名——X とする——の登録番号と解釈される。

(2) の配列の番号 i は，(mach [p] ->def) [i] がポイントする動作の，モジュール X におけるアドレスと一致する。

(3) のマスに入っている数は，モジュール X のアドレス i に示されている〈移行先アドレス〉。

last = mach [p] ->length − 1 とするとき，

(((mach[p]->def)[last])->next)[j] = −1

(j = 0, ····, k − 1)

14.2.4 リンク＝動作表作成

リンク＝動作表作成は，つぎのように行なう。即ち，mach [m] ->name が“main”を指すとき，

entry = m,

adr_off＝0　（アドレス・オフセット），
return_adr＝return
として，（再帰的）処理
write_tbl（entry，adr_off，return_adr）
に入る。ここで，write_tbl（entry，adr_off，return_adr）は，iが0からlast＝mach[entry]->lenght－1まで動くところの，つぎの処理：

(1) entryl＝（(mach［entry］->def）［i］->m_numが基底機械あるいはnullの登録番号であるとき，BINファイルに
i＋adr_off，
machine［0］
next［0］
……
machine［k－1］，
next［k－1］
を順に書き込む――ここで，

(1.1) machine［j］（j＝0，…k－1）は，
(1.1.1) entrylが基底機械の登録番号のときはmach［entryl］->nameで，
(1.1.2) nullの登録番号のときは，記号s_jを書き込む基底機械の名（j＝0のときは，e）

(1.2) next［j］（j＝0，…，k－1）は，
(1.2.1) i≠lastのとき，
next［j］＝
(((mach［entry］->def）［i］)
->next）［j］＋ad_off
(1.2.2) i＝lastのとき，
next［j］＝return_adr［j］
である。

(2) entrylが基底機械およびnullでないとき，処理：
write_tbl（entryl，adr_off＋last＋1，
((mach[entry]->def）[i]->next)
に入る。

TBLファイルの作成では，アドレスをASCII文字列で書き込むことになるが，このときには，nullとreturnの対に対応するところの最後の書き込みを，K個の＊＊の書き込みにする。

なお，リンクの図式を図2に示す。

14.3 MAKEファイルの作成

TC，TLの使用では，MAKEユーティリティを利用できる。このときのMAKEファイルは，つぎのような具合になる。：

```
#----------------------------------------
#  a. mk  (makefile)
#----------------------------------------
.m. obj：
    tc $*. m
a. obj：  a. m
x. obj：  x. m
y. obj：  y. m
a. bin：  a. obj x. obj y. obj
    tl -t  a+x+y, a
```

但し，a，x，yのどれかにmainモジュールがただ一つ存在し，a，x，yから呼び出されるモジュールは全てa，x，yのどれかに含まれるものとする。

15 チューリング機械リアライザ

15.1 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

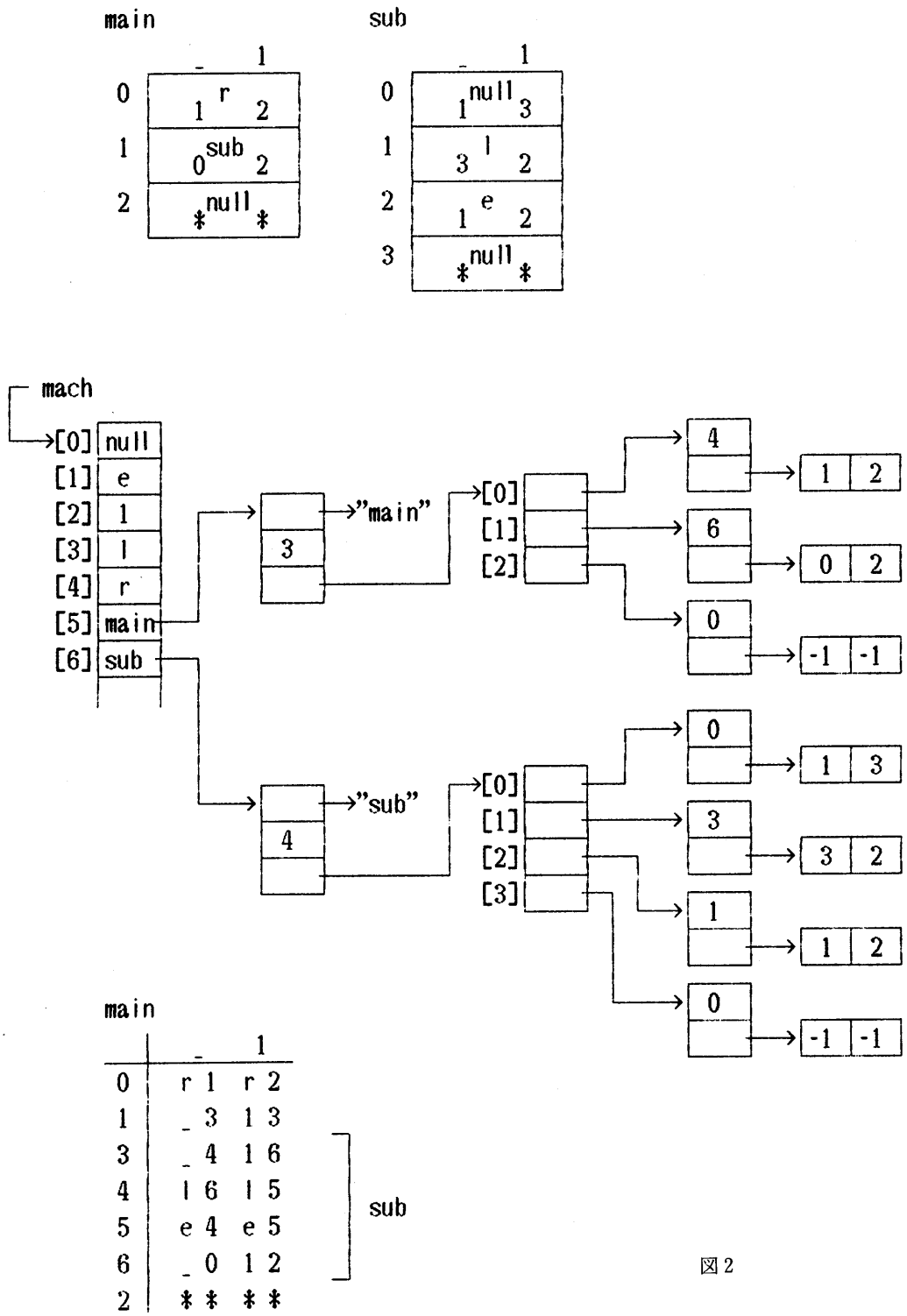

main
sub
mach
[0] null
[1] e
[2] 1
[3] l
[4] r
[5] main
[6] sub
3
4
"main"
"sub"
[0]
[1]
[2]
[3]
main
sub

図2

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。

15.2 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) チューリング機械定義ファイル(動作表ファイル)を読み込み，オフ状態でしかもテープがまだセッティングされていない段階のチューリング機械——メモリボードを装着していない電源オフのコンピュータに相当——を実現する。

(2) つぎにこのオフ状態において，“テープのセッティング”を，テープファイルの読み込み，あるいは文字列の直接入力を受けつけるという形で，実現する。これで，電源を入れ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。

15.3 チューリング機械リアライザ TU

われわれは，チューリング機械リアライザ——兼デバッガ（チューリング機械定義プログラムの）——として，実行プログラム

TU. EXE

を用意する。

TUは，BINファイル／TBLファイルを読み込み，これに記述されているチューリング機械を実現する。即ち，設定されたテープに対するこれの反応——初期状態から停止状態に至るまでのヘッドの移動およびテープ上の記号列の変化——を表示する。

デバッガとしての用途のために，トレース機能も付加している。

15.4 TU起動の書式

TUの起動の書式は，

TU [-e〈エディタ名〉]
[-f〈動作表ファイル名〉]
[-t〈テープファイル名〉]

である。

〈エディタ名〉は，TUの子プロセスとしてファイル編集するときに使用するエディタの名である。

〈動作表ファイル名〉では，拡張子を省略できる。省略されているとき，TUは拡張子がBINであるとしてファイルを検索する。見つからなければ，拡張子がTBLの名で検索する。

〈テープファイル名〉は，入力テープを定義するファイルの名である。拡張子——テープファイルの拡張子はTAP——を省略できる。

以上のファイル名は，パス名で指定する。

なお，ユーザは，バッチファイルを作成することにより，オプション指定の煩瑣を回避することができる。

例：

```
tu_. bat
    tu -eVZ -fADD -t5&8
```

TUは，指定された動作表ファイル／テープファイルが見つかれば，これを読み込む。

指定の動作表ファイル／テープファイルを見出せなかった場合，あるいは動作表ファイル／テープファイルがはじめから指定されていなかった場合，TUは初期画面において，欠けているファイルの指定を待つ。指定が得られ，かつ指定のファイルが見出せれば，これを読み込む。

TUは，最初にロードした動作表ファイルから，テープ記号を決定する。

15.5 動作表のファイル形式

15.5.1 TBLとBIN

動作表ファイル（チューリング機械定義ファイル)は，“動作表”のように見える必要はない。動作表ファイルについて肝心なことは，われわれにとってそれの可読性ではなく，チューリング機械リアライザにとってそれの可読性である。

われわれにとっての読みやすさと，リアライザにとっての読みやすさとは，同じではない。そこでわれわれは，動作表ファイルを二通りに定めることにする。

一つは，われわれにとっての可読性を優先した仕様であり，DSタイプの動作表をそのまま記述したテクストファイルである。そしてもう一つは，リアライザにとっての可読性（処理の効率性）とファイルのコンパクト性を優先した仕様のバイナリファイルである。この二種類の仕様ファイルは，それぞれ拡張子TBLとBINによって区別する。

チューリング機械リアライザTUは，この二種類のファイルを動作表ファイルとして受容できる。

BINファイルはMファイルのコンパイルによってつくられる。TBLファイルは，以下のいずれかの方法でつくられる：

(1) 直接エディタを用いてつくる；
(2) BINファイルをコンバータにかける；
(3) リンカの-tオプションで，BINファイルと同時に生成する。

TBL形式の動作表を受容できるようにしているのは，任意の動作表がコンパイル&リンクの方法で生成できるわけではなく，また，コンパイル&リンクの方法による動作表の作成が“最適化”の問題を孕んでいるからである。

15.5.2 TBLファイルの仕様

TBLファイルの仕様を，以下に述べる。なお，“行”は，改行コードで定義される“行”を意味するものとする。

リアライザは，行の先頭に書かれたーを，区切り記号として認識する。かつ，区切り記号のある行——以下，“区切り行”と呼ぶ——は，スキップして読み込まない。

区切り行は，三つ設けられる。したがって，TBLファイルの内容は，四つの領域でなる。

リアライザは，一番目と四番目の領域を，スキップして読み込まない。したがってこの領域を，表題/注釈の記述に使用できる。またこの二つの領域は，空でもよい。

第二および第三の領域の各行においては，スペースあるいはタブでなる記号列が，データの区切り記号になる。——即ち，リアライザは，スペース/タブ列をスキップして読み込まず，また，スペース/タブと異なる文字に出会うと，つ

ぎのスペース/タブに出会うまでの記号列——以下これを単に“記号列”と呼ぶ——をデータとして読み込む。

第二領域は一行でなり，リアライザはこれを，〈テープ記号〉の列挙と認識する。テープ記号は1バイトの文字でなければならず，スペース/タブ列で区切って書く。

第三領域の各行は，〈内部状態〉，〈動作〉，そしてテープ記号に対応する〈状態推移〉の記述である。

最終行の〈内部状態〉を，〈終了（停止）状態〉として固定する。これに対する〈動作〉，〈状態推移〉は，すべて＊とする。

例：

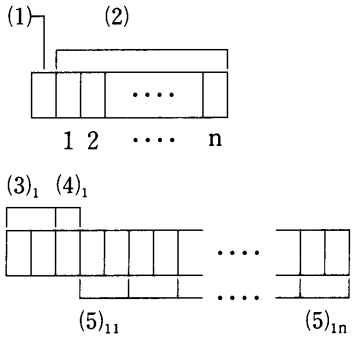
lb. tbl

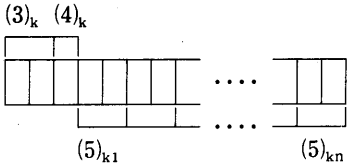
空白（“無記号”）に出会うまで左移動

```
--------------------------------
                  _     1
--------------------------------
    q0     l      q1    q0
    q1     *      *     *
--------------------------------
```

15.5.3 BIN ファイルの仕様

BIN ファイルの仕様をつぎのように定める（1マスが1バイト）：





(1) テープ記号の数：

1バイトの整数

(2) テープ記号：

テープ記号（キャラクターコード）

（但し，r，l，e以外）

$(3)_i$ i番目の内部状態（アドレスに表現）：

2バイトの整数

$(4)_i$ i番目の内部状態に対する動作：

右移動：文字r（キャラクタコード）

左移動：文字l（キャラクタコード）

消去　：文字e（キャラクタコード）

書込　：テープ記号（キャラクタコード）

$(5)_{ij}$ i番目の内部状態に対応する動作の後，j番目のテープ記号に対応して推移する先の内部状態（アドレス）：

2バイトの整数

15.5.4 TB0ファイル

SD タイプの動作表のファイル形式を，以下のように定める——ファイルの拡張子は，TB0とする。

ファイル形式は，TBL ファイルの場合に準じ，第三領域の書き方のみ異なる。

即ち，第三領域の各行において，最初の記号列は〈内部状態〉であり，そしてこれの後に〈動作—状態推移〉の列挙が続く。

最終行を除いては，各〈動作—状態推移〉は〈動作〉の1バイト文字と推移先の〈内部状態〉の記号列の連結である。

最終行の〈内部状態〉を，〈終了(停止)状態〉として固定する。これに対する〈動作—状態推移〉は，すべて記号列＊＊とする。

例：

lb. tb0

空白（“無記号”）に出会うまで左移動

```
--------------------------------
          _       1
--------------------------------
    q0      lq1     lq1
    q1      _q2     lq1
    q2      **      **
--------------------------------
```

15.6 ファイルコンバータ

われわれはユーティリティとして，BIN, TBL, TB 0 三種類のファイルの間のコンバータ

CONV. EXE

を用意する。

コンバートの方向は，つぎのオプションによって指定する：

—xy

ここで，x，yは，

BIN を表わすb

TBL を表わすt

TB0 を表わす0

のいずれかで，オプションの意味は“xからyへのコンバート”である。

BIN と TBL の間のコンバートは，データ型の変換に過ぎない。

TBL から TB0 へのコンバートは，つぎのようにする。即ち，

		s_1 ···· s_k
⋮	⋮	····
q	a	q_1 ···· q_k
⋮	⋮	····

の各行に対し

		s_1 ···· s_k
⋮	⋮	····
q	a	r ···· r
⋮	⋮	····
r	null	q_1 ···· q_k

の処置をする。そして

q	a	r ···· r

の形の行を

q	ar ···· ar

に書き換え，

q	null	q_1 ···· q_k

の形の行を

q	$s_1 q_1$ ···· $s_k p_k$

に書き換える。

逆に，TBO から TBL へのコンバートは，つぎのようにする。即ち，

	s_1 ···· s_k
⋮	····
q	$a_1 q_1$ ····$a_k q_k$
⋮	····

の各行に対し，

	s_1 ···· s_k
⋮	····
q	$s_1 r_1$ ···· $s_k r_k$
⋮	····
r_1	$a_1 q_1$ ···· $a_1 q_1$
⋮	····
r_k	$a_k q_k$ ···· $a_k p_k$

の処遇をする。そして

$$q \mid s_1 q_1 \quad \cdots\cdots \quad s_k q_k$$

の形の行を

$$q \mid \text{null} \mid q_1 \quad \cdots\cdots \quad q_k$$

に書き換え，

$$q \mid \quad ar \quad \cdots\cdots \quad ar$$

の形の行を

$$r \mid a \mid q \quad \cdots\cdots \quad q$$

に書き換える。

15.7 テープ

15.7.1 テープの書式

テープの書式は次の通りである：

(1) テープの記述では，動作表ファイルで指定されているテープ記号と矛盾しない限り，任意の文字が使える。

動作表ファイルの指定するテープ記号と矛盾するとき，TUはエラーメッセージを表示する。このとき，テープと動作表のうちエラーとして捨てられるのは，後から入力した方である。

(2) コンマは，〈ヘッド記号〉として予約されている。即ち，

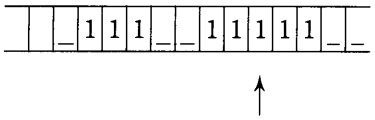
《コンマの直後の記号が，初期状態でヘッドの指している記号である》

となる。

例えば，

＿１１１＿＿１１，１１１

は，



を意味する。

(3) テープの記述に，〈ヘッド記号〉が欠けていたり，〈ヘッド記号〉が複数個あるのは，エラー。このときは，エラーメッセージが表示される。

(4) テープ記号の読み込みにおいて，空白はスキップされる。

(5) テープの記述は，テープの左端からの記述とみなされる。

(6) TUは，記号列の右端から以降を“無記号”記号(＿)が書き込まれているものと判断し，トレースの際必要に応じてこの記号を補う。

15.7.2 テープの入力

テープ入力の方法はつぎの二通りである。

(1) あらかじめ入力式を書き込んだテクストファイルを拡張子TAPでつくり，TUに読み込ませる。ファイルの指定は，TUを起動するときのコマンドラインか，TUのファンクションキーメニューによる。

(2) TUのファンクションキーメニューからテープ入力モードを選択し，直接入力する。