一般社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
SS2012-8 (2012-5)

Malbolgeの高級アセンブリ言語への配列機能の追加

安藤 聡† 酒井 正彦†† 坂部 俊樹†† 草刈 圭一朗†† 西田 直樹††

†,†† 名古屋大学 大学院情報科学研究科
〒464-8603 愛知県名古屋市千種区不老町
E-mail: †ando@sakabe.i.is.nagoya-u.ac.jp, ††{sakai,sakabe,kusakari,nishida}@is.nagoya-u.ac.jp

あらまし Malbolgeは最も難解なプログラミング言語として知られている．高級アセンブリ言語の開発によりMalbolgeプログラムの作成が可能になっているものの，プログラム中で使用できる変数の値の最大値が固定されておりゲーデルコーディングが不可能であるため，配列機能がないのは記述力不足であった．本論文ではこの問題を解決するため，高級アセンブラに用いられている実現手法を整理し，これに配列機能のための命令である領域確保命令と間接参照命令を追加する方法を提案する．
キーワード 難解プログラミング言語，Malbolge，配列機能

Introducing Array Mechanism into High-Level Assembly Language for Malbolge

Satoshi ANDO†, Masahiko SAKAI††, Toshiki SAKABE††,

Keiichirou KUSAKARI††, and Naoki NISHIDA††

† Graduate School of Information Science, Nagoya University
Furo-cho, Chikusa-ku, Nagoya-City, Aichi, 464-8603 Japan
E-mail: †ando@sakabe.i.is.nagoya-u.ac.jp, ††{sakai,sakabe,kusakari,nishida}@is.nagoya-u.ac.jp

Abstract Malbolge is known to be one of the most esoteric programming languages. Although it is possible to write programs in Malbolge by the development of a high-level assembly language, lack of facility to manage individual data in a group of data like an array in language causes problem because Godel-coding is impossible in a program due to bounded values in variables. In this paper, in order to solve this problem, we show implementation issues used in the assembler in well-organized manner and propose a method for implementing a memory allocation instruction and an indirect reference instruction for array facility into the assembler.
Key words Esoteric Programming Language, Malbolg, Array Mechanism

1. はじめに

難解プログラミング言語は意図的にその言語でのプログラミングが困難になるように設計された言語である．このような言語で書かれたプログラムは解読困難性を持つため，情報セキュリティにおいてプログラムの改ざん防止や知的財産保護に役立つと考えられている．

Malbolge [1] は難解言語の中でも特に難解として知られており，その難解性からプログラムの解読や変更だけでなく直接のプログラミングも非常に困難である．これまで，Malbolgeでは特定の文字列を出力するプログラムやエコープログラム程度しか知られていなかったが，近年，飯澤らによりプログラミング手法 [2]~[4] が提案・拡張され，不可能であると考えられてきたMalbolgeでのループプログラム作成が，難しいながらも可能になった．飯澤らの手法では，ループ耐性を持った低級アセンブリ言語の提案と，低級アセンブリプログラムからMalbolgeコードを生成する低級アセンブラの構築が行われた．また，飯澤らはMalbolgeプログラミング効率化のための高級アセンブリ言語を設計し，低級アセンブリ言語で実装したインタプリタでシミュレートさせている．しかし，配列のような機能が利用できるものはなく，プログラム中で使用できる変数の値の最大値も固定であり，実用上の問題があった．

本稿では，この問題を解決するために高級アセンブリ言語へ配列機能を追加する．配列機能の追加は，領域確保命令と間接参照命令を高級アセンブリ言語に加えることで実現した．領域確保命令の追加は，プリプロセッサを拡張し，加算命令と変数

表 1 Malbolge の命令
Table 1 Malbolge Instruction

命令	表記	説明
i	Jmp	ジャンプ. C:=mem[D].
j	MovD	D レジスタの更新. D:=mem[D].
p	Opr	演算命令. A,mem[D]:=op(A,mem[D]).
*	Rot	右ローテート. A,mem[D]:=rotr(mem[D]).
/	Input	入力. A:=getchar().
<	Output	出力. putchar(A).
o	Nop	無操作. 何も行わない.
v	Halt	終了. プログラムの実行を停止.
その他	Nop'	無操作.

表 2 op の各桁の演算
Table 2 Operation of Each Trit of 'op'

Y_i \ X_i	0	1	2
0	1	0	0
1	1	0	2
2	2	2	1

コピー命令に変換することで実現した. 間接参照命令の追加は, プリプロセッサにより変数コピー命令へ変換, 新たな命令ユニットの設計, それを用いたインタプリタの拡張を行うことにより実現した.

2. Malbolge

ここでは Malbolge の仕様について本稿で必要な部分のみを説明し, Malbolge の命令を拡張した疑似命令について紹介する. Malbolge についてのより詳細な説明は文献 [2], [3], [5] を参照されたい.

Malbolge は仮想機械上で動作する機械語であり, インタプリタによって意味が定められている. 仮想機械は三つのレジスタ (A,C,D) とメモリ (mem) を持ち, 値は三進数十桁 (10trits) で表現される. よって値は 0000000000t〜2222222222t となり, メモリのアドレス空間も mem[0]〜mem[59048] で定義される.

表 1 に Malbolge の命令をわかりやすく表した. ここで, 命令 Opr の演算子 op(X,Y) は 2 引数の各桁同士で表 2 に従って trit 演算を行う命令である. 以下にその例を示す.

op(0120120120t, 0001112222t) → 1001022212t

また, 命令 Rot の演算子 rotr を例で示すと以下のようになる.

rotr(0001112222t) → 2000111222t

疑似命令 [2] は, Malbolge の演算命令の表現をより単純化し, 引数にメモリ名を指定できるようにしたものである. 例えば, 以下の疑似命令列を考える.

```
ROT  X
OPR  Y
```

これは以下の計算を意味する.

```
A,X  :=  rotr(X)
A,Y  :=  op(A,Y)
```

また, 疑似命令の引数に定数として Con0(=0000000000t), Con1(=1111111111t), Con2(=2222222222t), Pat21(=2222222221t) を利用できる. なお文献 [2] では Input と Output に対応する疑似命令が定義されていないが, これらは Malbolge 命令から自然に定義できる.

```
ENTRY:  FLAG1+1              REV_OPR
R_CON1: ROT_UNIT CON1        FLAG2
CON1:   1111111111t             R_CON1
        REV_ROT          O_Y: OPR_UNIT Y
        FLAG1            Y:   0120120120t
           O_Y                REV_OPR
O_X:    OPR_UNIT X           END
X:      0
```

図 1 低級アセンブリプログラムの例
Fig. 1 Example Code of Low-Level Assembly Program

表 3 低級アセンブリ言語の命令
Table 3 Low-Level Assembly Language Instruction

命令	操作
ROT_UNIT *label*	A,[*label*] := rotr([*label*])
REV_ROT	ROT 命令の復元
OPR_UNIT *label*	A,[*label*] := op(A,[*label*])
REV_OPR	OPR 命令の復元
JMP_UNIT *label*	PC:= *label*
MOV_PC_UNIT *label*	PC:= *label*
REV_MOV_PC	MOV_PC 命令の復元
INPUT_UNIT	A:=getchr()
REV_INPUT	INPUT 命令の復元
OUTPUT_UNIT	putchr(A)
REV_OUTPUT	OUTPUT 命令の復元
FLAG *label*	FLAG が ON 場合 : PC:= *label*
	FLAG が OFF の場合 : 何もしない
	命令実行後には FLAG がフリップされる
FLAG+1	FLAG がフリップされる (ON↔OFF)
END	終了

3. 低級アセンブリ言語

ここでは, 強制的な動的書き換えにより Malbolge でループプログラム作成が困難という問題を解決するために設計された低級アセンブリ言語の仕様と, その実現手法について紹介する.

3.1 仕　　様

低級アセンブリ言語 [2] は Malbolge と同じメモリ空間を持つ仮想機械として定義される. レジスタは PC と A の 2 つを持ち, 値は 10trits で表現される. 図 1 に低級アセンブリプログラム例を示す. PC と A の初期値はそれぞれ ENTRY, 0 であるとする. このプログラムは以下の疑似命令列と対応している.

```
ROT  Con1
OPR  X ( 初期値 0)
ROT  Con1
OPR  Y( 初期値 11355)
```

低級アセンブリプログラムは, メモリアドレスを表すラベルを付加可能なデータの列によって定義される. 各データは変数と命令, およびフラグのいずれかであり一行で記述される. 命令やフラグの引数にはラベルを用いる. 低級アセンブリ言語の命令を表 3 にまとめた. ただし, [*label*] は *label* でラベル付けされたデータを表しており, A は A レジスタを指す.

表 3 から分かるように, JMP_UNIT を除いた各命令にはその復元命令が存在する. プログラムを記述する際には, プログラ

表 4　命令実行後の変換サイクル [6]

Table 4　Periodic of character-replacement

period	sequence
2	F J
4	* r } i
5	) f ' < 3
6	% g u o x :
9	2 P B > L O C U I 2
68	! 5 - w N 1 W 0 { G S ~ 9 [···

ム実行時にある命令が実行された後その命令の復元命令が実行されるようにする必要がある．フラグはフリップフロップの役割を果たしており，これによって実行を制御する．フラグの初期値は全て ON である．変数の値には，定数として十進数(0～59048) と三進数(0000000000t ～2222222222t) が利用でき，特別な定数として任意の定数を表す DUP が利用できる．

3.2 実　　現

低級アセンブラは命令ユニットを用いて実現される [3].

命令ユニットとは，Malbolge の命令を動的に書き換える変換の周期性(表 4) を利用したもので，任意個の SNop と周期性のある命令，Jmp という並びで構成される命令列である．SNop とは任意のアドレスで存在する，常に Nop か Nop' となる命令である．飯澤らは Jmp 以外の命令について主に周期 2 の命令ユニットを構築し，それを低級アセンブリ言語の命令としている．それらは同じ構造であるため，ここでは Opr のユニットについてのみ紹介する．

OPR_UNIT は以下の構造である．

```
          SNop
OPR_UNIT: Opr/Nop'
REV_OPR   Jmp
```

この命令ユニットの使い方を説明する．今，C レジスタがどこかの Jmp を指しており，D レジスタが以下のデータの最初の位置を指しているとする．

データ	実行される命令	書き換え
OPR_UNIT-1	Jmp (C レジスタを OPR_UNIT-1 にセット)	SNop → SNop
0	Opr (op(A,0) を A と X に格納)	Opr → Nop'
REV_OPR-1	Jmp(C レジスタを REV_OPR-1 にセット)	Nop' → Opr

まず，Jmp により C レジスタが OPR_UNIT-1 にセットされる．そして命令の書き換えが行われるが，OPR_UNIT-1 の命令は SNop であるため，書き換え後も SNop である．C，D レジスタともインクリメントされ，次は Opr が実行される．そして Opr が Nop' に書き換えられる．再び C，D レジスタがインクリメントされ，Jmp で C レジスタが REV_OPR-1 にセットされ，命令が書き換えられる．この際，仕様により Jmp は実行後も書き換えられず，ジャンプ先の命令が書き換えられる．REV_OPR-1 に置かれた Opr は周期 2 のため，この書き換えで Nop' から Opr に復元される．これらの実行終了後，C レジスタは Jmp を指しており，D レジスタは次のアドレスを指しており，同様に実行が続行する．

このようにして飯澤らは，命令ユニットとそれに対応するデータ構造を構築し，命令を復元しながら実行することでループプログラミングを可能にしている．

```
VAR_SET
VAR_X =         0000000000t
VAR_Y =         0000000001t

PROGRAM
PROGRAM_INIT:
LOOP:           INPUT VAR_X
                MOV VAR_X,VAR_Y
                INC VAR_Y
                BRANCH VAR_Y,EXIT
                OUTPUT VAR_X
                BRANCH 0,LOOP
EXIT:           STOP
```

図 2　高級アセンブリプログラムの例

Fig. 2　Example Code of High-Level Assembly Program

4. 高級アセンブリ言語

文献 [2] で飯澤らは Malbolge プログラミング効率化のための高級アセンブリ言語を定義している．しかし，その構文はユーザにわかりにくいので，本稿では飯澤らの定義した高級アセンブリ言語のシンタックスシュガーを与え，それを高級アセンブリ言語として扱い，飯澤らのものを中間コードと呼ぶ．高級アセンブリプログラムから中間コードへの変換は，我々が構築したプリプロセッサを用いて行う．また，飯澤らは中間コードをシミュレートし動作するインタプリタを構築した [2] が，インタプリタ内部の動作は公開されていなかった．そのため本節ではインタプリタの解析を行うことで明かになった事実について整理して説明する．

4.1 高級アセンブリ言語の仕様

高級アセンブリ言語は Malbolge プログラミングをより容易にすることを目的として設計した言語である．このため，値の取る範囲は 10trits である．まずサンプルプログラムを図 2 に示す．プログラムは変数宣言部と本体部の二種類からなる．変数宣言部では本体部で使用される変数の初期値の宣言を行い，その先頭には "VAR_SET" を記述する．値は末尾に "t" をつけると三進数として解釈し，何もつけなければ十進数として解釈する．本体部ではプログラム本体の記述を行い，その先頭には "PROGRAM" を記述する．プログラムはラベルを付加可能な命令列であり，ラベルはその命令の置かれたアドレスを表している．命令と引数は必ず一行で記述し，コロンの左側がラベルであり複数付与することが可能である．命令の引数は変数か数値，ラベルであり，数値で記述する場合は変数宣言部と同様の解釈をなす．

高級アセンブリ言語の命令を表 5 にまとめた．IP はプログラムカウンタ，X，Y は変数，L はラベルを表す．なお INC の引数の値が 59048 のときの演算結果は 0，DEC の引数の値が 0 のときの演算結果は 59048，ADD でオーバーフローが発生したときは演算結果の最上位 trit を切り捨てた値を Y に代入する．

4.2 プリプロセッサと中間コードのインタプリタ

前節で説明した高級アセンブリ言語はユーザに使いやすく設計された言語である．このため，プリプロセッサを用いてイン

表 5　高級アセンブリ言語の命令

Table 5　High-Level Assembly Language Instruction

命令と引数	操作	備考
INC X	X := X+1, IP:=IP+1	インクリメント
DEC X	X := X-1, IP:=IP+1	デクリメント
MOV X,Y	Y := X, IP:=IP+1	変数コピー
BRANCH X,L	if X=0 then IP:=L else IP:=IP+1	ゼロ判定分岐
INPUT X	X := input(), IP:=IP+1	入力
OUTPUT X	output(X), IP:=IP+1	出力
ADD X,Y	Y := X+Y, IP:=IP+1	加算
STOP	halt	終了

```
                                MOD_MOV
VAR_X :         0000000000      VAR_X-2
                VAR_RET         VAR_RET
VAR_Y :         0000000001      VAR_Y-2
                VAR_RET         VAR_RET
VAL_0 :         0000000000      MOD_INC
                VAR_RET         VAR_Y-2
VAL_EXIT:       EXIT            MOD_BRANCH
                VAR_RET         VAR_X-2
VAL_LOOP:       LOOP            VAL_EXIT-2
                VAR_RET         MOD_OUTPUT
                                VAR_X-2
PROGRAM_INIT:                   MOD_BRANCH
LOOP:           MOD_INPUT       VAL_0-2
                VAR_X-2         VAL_LOOP-2
                          EXIT: MOD_STOP
```

図 3　プリプロセッサの出力例(中間コード)

Fig. 3　Example Output of Preprocessor

タプリタでシミュレート出来るように中間コードに変換する必要がある.

4.2.1　プリプロセッサ

プリプロセッサの入力は高級アセンブリプログラムで，出力は中間コードである．中間コードは入力の各命令から一意に変換できるデータ列，ならびに変数データで構成される．例として，図 2 のプログラムからプリプロセッサが生成する中間コードを図 3 に示す.

プリプロセッサは，各命令の名前の前に "MOD_" を，引数である変数の後ろに "-2" を付け，以下の変換を行う.

- INC,DEC,INPUT,OUTPUT,ADD,STOP の場合

これらの命令は，命令の下の行に引数を順に一行ずつ記述する.

- BRANCH の場合

命令の下の行に引数を順に一行ずつ記述する．ただし，引数に使用するラベルを値とする新たな変数を高級プリプロセッサにより作成し，引数はその変数とする．その際，新たな変数につけるラベルは値とするラベルの前に "VAL_" をつけたものとする.

- MOV の場合

命令の下の行に第一引数を記述し，次に VAR_RET を記述．そして第二引数を記述し，最後にまた VAR_RET を記述する.

```
                SNop                    SNop
MM_UNIT:        SNop            MMOM_UNIT:  MovD (9)
                MovD (9)        REV_MMOM:   MovD (9)
REV_MM:         MovD (9)                    SNop
                Jmp                         SNop
                                            Opr (9)
                                            MovD (9)
                                            Jmp
```

図 4　低級アセンブリ言語の新たな命令

Fig. 4　New Instruction of Low-Level Assembly Language

- 変数宣言部の場合

変数名をラベルにし，初期値を記述する．そして下の行に VAR_RET を記述する．なお，値は全て三進数十桁に変換して記述する.

変換におけるいくつかの不可解な操作は，この後に説明するインタプリタの実装の都合によるものである.

4.2.2　中間コードのインタプリタ

ここでは中間コード参照し動作するインタプリタについて，文献 [2], [3], [7] で明らかとなっていない部分について以下の順に説明する.

- 低級アセンブリ言語の拡張
- インタプリタの全体図
- ライブラリ群の構築

まずインタプリタ構築に必要となる低級アセンブリ言語の拡張を説明する．飯澤らは，低級アセンブリ言語に新たに二つの命令 MM_UNIT, MMOM_UNIT とその復元命令 REV_MM, REV_MMOM を加えた．MM_UNIT と MMOM_UNIT の構成は図 4 である．命令の後ろについている括弧内の値は各命令の周期を表しており，SNop, Jmp 以外はすべて周期性を持つことから，復元を行えば繰り返しの使用が可能であることが分かる.

これらの命令の使用例を表 6 に示す．表 6 の左側に簡単な低級アセンブリプログラムを，右側にその実行トレースを示す．右の表でラベルを用いてるが，ラベルはアドレス，つまり数値として考える．C, D レジスタの初期値は実行トレースの一行目になっていると仮定する．まず最初の Jmp で C レジスタを MM_UNIT にセットする．そして，Nop で DUP を読み飛ばしたあと，一つ目の MovD で PROGRAM_INIT に置かれたアドレスにジャンプしたのち，二つ目の MovD で MOD_NOP にジャンプする．MMOM_UNIT は一つ目の MovD で PROGRAM_INIT に置かれたアドレスにジャンプしたのち，二つ目の MovD で Opr を行いたい値の二つ上にジャンプし，二つ Nop を挟んだ後に Opr を行い，VAR_RET にジャンプする.

これらの命令は低級アセンブラの拡張として捉えることが自然であるが，その実現には低級アセンブラ自体の拡張は必要でなく，低級アセンブリプログラム内にある命令ユニットを記述する部分にこれらを記述すればよい.

次にインタプリタ全体の構成について紹介する．インタプリタ全体の概略は図 5 のようになる．インタプリタはライブラリ群からなり，中間コードを参照することで動作する．ライブラリ群は主に高級アセンブリ言語の各命令と同じ機能を実現する

表 6　MM_UNIT と MMOM_UNIT の使用例

Table 6　Few-steps execution of MM_UNIT and MMOM_UNIT

adr.	低級アセンブリ プログラム例
0	DUP
1	DUP
2	VAR_X : 0000000001
3	VAR_RET
4	PROGRAM_INIT:MOD_NOP
5	VAR_X-2
6	MM_UNIT
7	DUP
8	PROGRAM_INIT
9	MOD_NOP: JMP_UNIT NEXT
10	NEXT: MMOM_UNIT
11	PROGRAM_INIT+1
12	VAR_RET: JMP_UNIT
	⋮

	実行トレース	
step.	[C]	[D]
0	Jmp	MM_UNIT
1	Nop	DUP
2	MovD	PROGRAM_UNIT
3	MovD	MOD_NOP
4	Jmp	JMP_UNIT
5	Jmp	MMOM_UNIT
6	MovD	PROGRAM_UNIT+1
7	MovD	VAR_X-2
8	Nop	DUP
9	Nop	DUP
10	Opr	0000000001
11	MovD	VAR_RET
12	Jmp	JMP_UNIT
⋮		

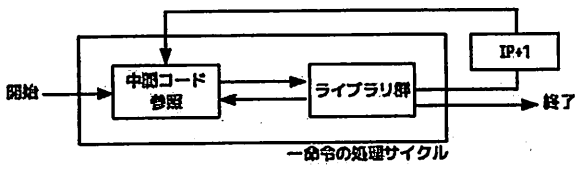

図 5　中間コードのインタプリタ

Fig. 5　Middle Code Interpreter

```
ROT Con1                OPR X
OPR Z                   ROT Con2
OPR Z                   OPR X
OPR Y                   OPR Z
OPR Y                   OPR Y
ROT Con2
```

図 6　変数コピーの疑似命令列 [2]

Fig. 6　Pseudo Instruction of Data Copy

低級アセンブリ プログラムのモジュールで構成される．

インタプリタの動きを説明する．まず中間コードを参照し，その命令と対応するライブラリ群のモジュールへ移動する．モジュールでは命令の操作を実行するが，必要に応じて中間コード中の引数などの参照を行う．この参照の際に先ほど低級アセンブリ言語の拡張で追加した二つの命令を利用する．モジュール呼び出しには MM_UNIT を用いて，引数アクセスには MMOM_UNIT を用いる．高級アセンブリ言語の一命令の処理が終了すると，プログラムカウンタ IP をインクリメントし，次の命令の処理に取りかかる．このような繰り返しをインタプリタは行う．

ライブラリ群の構築法について紹介する．飯澤らと長坂らにより，ROT と OPR をどのように組み合わせれば高級アセンブリ言語の各命令の機能が実現できるのかが明らかにされている [2] [7]．よって，それらを元に低級アセンブリ言語でコーディングをすることで，ライブラリ群の各モジュールが構築されている．例として変数コピー MOV X,Y の ROT と OPR の並びを疑似命令列で図 6 に示す．変数 X，Y，Z の初期値は任意である．

表 7　新たな命令の変換

Table 7　Transformation of New Instruction

ALLOC X,Y	STOREI X	LOADI X
MOD_MOV	MOD_MOV	MOD_MOV
VAR_ALLOC-2	X-2	IN_DIRECT
VAR_RET	VAR_RET	VAR_RET
X-2	IN_DIRECT	X-2
VAR_RET	VAR_RET	VAR_RET
MOD_ADD		
Y-2		
VAR_ALLOC-2		
MOD_ADD		
Y-2		
VAR_ALLOC-2		

5.　高級アセンブリ言語への配列機能の追加

前節までに説明した Malbolge のための言語のいずれにおいても，プログラム中で使用できる変数の値の最大値は固定であり，かつ配列のような機能も存在せず，実用上で問題がある．

そこで，高級アセンブリ言語へ配列機能を追加することで，この問題を解決する．配列機能の追加は，領域確保命令 ALLOC と間接参照命令 STOREI，LOADI を高級アセンブリ言語に追加することで実現する．各命令の仕様は以下である．なお，VAR_ALLOC とはメモリ中で使用可能な範囲の先頭アドレスを格納した特殊な変数であり，INDEX はインデックスレジスタである．

- ALLOC X,Y

X:=VAR_ALLOC,VAR_ALLOC:=VAR_ALLOC+Y,IP:=IP+1

- STOREI X

[INDEX]:=X,IP:=IP+1

- LOADI X

X:=[INDEX],IP:=IP+1

これら命令の追加はプリプロセッサの拡張，特殊な命令ユニット INDIRECT_UNIT の構築，インタプリタの拡張により実現する．

5.1　プリプロセッサの拡張

4.2.1 で紹介したプリプロセッサを拡張する．拡張したプリプロセッサの入力は ALLOC，STOREI，LOADI を記述可能な高級アセンブリプログラムで，出力は中間コードである．各命令の変換は表 7 のように行う．

変換について説明する．ALLOC は加算と変数コピーに分けて操作を行うため，それぞれの命令に対応した中間コードに変換する．まず第一引数に VAR_ALLOC の値をコピーする．この変数はメモリ中で使用可能な範囲の先頭アドレスを格納した特殊な変数であり，その初期値は現在プリプロセッサを実行する毎にあらかじめ適切な値の設定が必要である．次に VAR_ALLOC に第二引数の値を二度加える．これはメモリ中の使用可能な範囲の先頭アドレスが領域確保により変更されるためであり，二倍の領域を確保する理由はインタプリタの拡張で説明する．

STOREI，LOADI の操作は，表 5 で示した変数コピー命令の操作と非常に似ている．そこで変数コピー命令を利用することでこれらの命令を実現する．そのためプリプロセッサはこれ

らの命令を引数が特殊な変数コピーの中間コードに変換する．STOREIの第二引数，LOADIの第一引数がIN_DIRECTとなっており，固定である．IN_DIRECTはこの後説明する間接参照モジュールを利用するための記述である．変数コピー命令に関する詳しい説明は文献[2]を参照されたい．

5.2 中間コードのインタプリタの拡張

4.2.2で紹介したインタプリタのライブラリ群の拡張を行う．拡張内容はライブラリ群の変数コピーモジュールのサブモジュールとしての間接参照モジュールの追加である．前節で紹介したIN_DIRECTはこのモジュール内に記述されたラベルであり，拡張したモジュールへの入り口だと考えれば良い．

間接参照モジュールで行う操作は，間接なOpr命令を行うことである．図6で示した疑似命令の順にRotとOprを行えば，XをYにコピーできる．つまり間接参照命令を実現する上で行う必要があるのは，引数として任意のアドレスをとり，そのアドレスへとMovDを用いてジャンプしたのちにOprを行い，再びインタプリタ部に戻ってくるような命令ユニットを構築することである．その命令ユニットの引数となる変数をINDEXレジスタとして扱う．

5.2.1 INDIRECT_UNITの設計

間接参照モジュールで行う操作の肝となる命令ユニットINDIRECT_UNITの設計を行う．この命令ユニットがどのような動きをするものかは上で述べたが，この命令ユニットを構築する上で鍵となるのは如何にしてインタプリタ部に戻ってくるかである．この問題は飯澤らのプログラミング手法におけるメモリ上の二つのデータ構造により解決できる．

一つ目は，低級アセンブリプログラムを展開した場所より後ろのメモリには，81，29443という値が交互に格納されていることである．これは，プログラムをメモリに展開する際に余った領域には$[m] := op([m-1],[m-2])$という処理を行うMalbolgeの仕様を利用し，飯澤らが意図的に構築したものである．配列に使用するメモリ領域を低級アセンブリプログラムの後ろにすることで，これらの値を利用してMovDでDレジスタを81にセットする．このため，配列に使用するのは29443が格納されていたアドレスのみであり，この仕様から配列の大きさの二倍の領域を確保せねばならない．

二つ目は，メモリの81番地付近にデータモジュール[2]と呼ばれるデータ群が格納されていることである．これは低級アセンブリプログラムをMalbolgeのメモリ上に構築するために用いられるデータ群であるが，そのデータ群の中に高級アセンブリインタプリタのエントリーポイントのアドレスが存在する．この値を利用してDレジスタをインタプリタ上に戻す．

これらの事実を利用し設計したINDIRECT_UNITが図7である．データモジュールなどの固定のデータ構造をうまく利用するために，MovDとOprの間に挿むSNopの数を考慮し，MovDとOprが短い周期性を持つことを条件として設計した結果，このような構造となった．

このINDIRECT_UNITを間接参照モジュール内でINDEXレジスタを引数にして用いることにより，間接的なOprが可能となり，結果，間接参照の機能を実現することが出来た．

```
                SNop            UNIT2_5:  SNop
INDIRECT_UNIT:  MovD(6)                   SNop
UNIT2_1:        SNop                      SNop
                SNop                      SNop
                SNop                      SNop
                Opr(9)                    SNop
UNIT2_2:        SNop                      SNop
                SNop                      MovD(4)
UNIT2_3_2:      MovD(5)           UNIT2_6:  MovD(5)
UNIT2_4:        MovD(9)           UNIT2_7:  Jmp
```

図7 INDIRECT_UNIT

6. まとめ

高級アセンブリ言語に領域確保命令，間接参照命令を追加することにより，配列機能の使用を可能にした．本稿の結果を元に，長坂らの示したMalbolgeの弱チューリング完全性[7]を完全なものにできるのではないかと考えている．それは長坂らの示した弱チューリング完全性が，変数の値，および変数と命令の個数にある上限を仮定した性質であるためである．

Malbolgeプログラムをソフトウェア保護の目的で利用する場合，本稿で紹介した作成手法で作成されたことが分かってしまうと解析困難性が低下してしまう恐れがある．特に飯澤らの高級アセンブリ言語で実装したプログラムの逆コンパイルが可能であることがすでに明らかになっている[8]．そこで逆コンパイルを困難にするために，現在のインタプリタを用いた手法ではなく，高級アセンブリプログラムから直接低級アセンブリプログラムを生成するコンパイラを作成するなどして，より難読性を高めることが必要であると考えている．

謝辞 本研究は一部，科研費 #22650003 の助成を受けている．

文献

[1] Ben Olmstead: "Malbolge: Programming from Hell", http://www.boulet fermat.com/danny/malbolge/, 1998.
[2] 飯澤恒: 難解言語 Malbolge に基づくプログラム難読化に関する研究，名古屋大学修士論文，2006.
[3] 飯澤恒，坂部俊樹，酒井正彦，草刈圭一朗，西田直樹: 難読プログラミング言語 Malbolge におけるプログラム構成手法，信学技報，電子情報通信学会，Vol.105，No. 129，pp. 25-30，2005.
[4] 安藤聡，酒井正彦，坂部俊樹，草刈圭一朗，西田直樹: Malbolge の高級アセンブリ言語への加算命令の追加，日本ソフトウェア科学会第 28 回大会講演論文集，No. 5A-3，12 pages，那覇，September 2011.
[5] Malbolge，Wikipedia, http://en.wikipedia.org/wiki/Malbolge.
[6] Lou Scheffer: Introduction to Malbolge, http://www.lscheffer.com/malbolge.html.
[7] 長坂哲，酒井正彦，坂部俊樹，草刈圭一朗，西田直樹: 難解言語 Malbolge のチューリング完全性について，信学技報，電子情報通信学会，Vol. 110，No. 227，pp. 55–60，2010.
[8] 菅優也： 難読言語 Malbolge の逆コンパイル困難性に関する研究，高知工科大学卒業論文，2011.