

Malbolge 低級アセンブラにおけるコード配置アドレスの決定法

加藤 起騎[†] 酒井 正彦^{††} 坂部 俊樹^{††} 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: [†]tatsuki@sakabe.i.is.nagoya-u.ac.jp, ^{††}{sakai,sakabe,nishida}@is.nagoya-u.ac.jp

あらまし Malbolge は最も難解でプログラミングが困難な言語として知られている。近年, Malbolge 低級アセンブラが開発され, Malbolge プログラムの生成が可能になってきている。しかしながら, これまでの低級アセンブラでは, コード配置アドレスをユーザにより試行錯誤により定める必要があった。これは, 配置コードを生成するプログラム長が与えた配置アドレスにより著しく変化することと, メモリ不足から生成プログラムと配置アドレスを離して置くことができないことが原因であった。本研究ではこの問題の解決のために, メモリ空間を拡大した Malbolge20 のための改良版低級アセンブラを提案する。改良版低級アセンブラでは, コード長は増加するもののコード長が配置アドレスに依存しない手法を用いることで, コード配置アドレスを自動化できた。

キーワード 難解プログラミング言語, Malbolge

Deciding Code Allocation on Malbolge Low-Level Assembler

Tatsuki KATO[†], Masahiko SAKAI^{††}, Toshiki SAKABE^{††}, and Naoki NISHIDA^{††}

[†] Graduate School of Information Science, Nagoya University

Furo-cho, Chikusa-ku, Nagoya-City, Aichi, 464-8603 Japan

E-mail: [†]tatsuki@sakabe.i.is.nagoya-u.ac.jp, ^{††}{sakai,sakabe,nishida}@is.nagoya-u.ac.jp

Abstract Malbolge is known as one of the most esoteric and hard-to-programming languages. Recently a low-level assembler (LA-assembler) has been developed, which makes it possible to produce a class of Malbolge programs. In the current LA-assembler, however, users must give an appropriate address for code allocation by trial and error. This is because the lengths of programs for generating codes are unexpectedly changed depending on code allocation addresses, and it is impossible to clearly separate the allocated code and the programs for generating codes from lack of memory. In this paper, we propose an LA-assembler for Malbolge20, which is a memory-space extended Malbolge, in order to solve this problem. The LA-assembler generates codes whose lengths do not depend on code allocation addresses although the generated codes are longer than ones generated by the previous assembler. Thus a function for automatic address allocation is easily implemented.

Key words Esoteric Programming Language, Malbolge

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, 数ある難解言語の中でも特に難解として知られており, プログラムの解析はもちろんのこと, プログラムの作成すら非常に困難である。これまでに, 飯澤らにより

Malbolge のための中間言語として低級アセンブリ言語が設計され, そのプログラムから Malbolge プログラムを生成する低級アセンブラが構築されている [7]。低級アセンブラを用いたプログラミングにおいては, ターゲット言語である Malbolge のメモリ空間の制約によりプログラムの生成にしばしば失敗していた。著者らはこの問題を解決するために Malbolge のメモリ空間を拡大した Malbolge20 及び Malbolge20 用低級アセンブラを構築した [8]。Malbolge20 用低級アセンブラの構築によりメモリ不足は解決されたが, プログラミングは依然困難なままであった。これは, 低級アセンブラが行うべき処理のうちで

コード配置アドレスの決定は自動で行えず、ユーザの試行錯誤が必要であったためである。

本稿ではこの問題の解決のために、Malbolge20 用低級アセンブラを、コード配置アドレスの決定が行えるように改良する。本稿の構成は次の通りである。2 節では、準備として Malbolge20 について説明する。3 節で Malbolge20 用低級アセンブラの問題点とその解決方法を明らかにし、4 節でまとめを述べる。

2. 準備

本節では、準備として言語 Malbolge20 を簡単に述べる。詳細については文献 [8] を参照されたい。

Malbolge20 プログラムは、印字可能文字 (ASCII 値: 33~126) の列であり、その長さは 3^{20} 文字未満でなければならない。プログラムの各文字 c はすべて、ASCII コードで 33 から 126 の間の印字可能文字であり、かつ、 k を加えた後に変換表 xlat1 で変換して $i,j,p,*,/,<,v,o$ となるもののみが許される。ここで、 k は文字 c が置かれた位置であり、変換表 xlat1 は印字可能文字領域上の順列である。

Malbolge20 の意味は仮想機械により定められている。仮想機械は 3 つのレジスタ (アキュムレータ A、コードポインタ C、データポインタ D) とメモリ (mem) を持ち、値及びメモリ空間は三進数 20 桁 (20trits)、即ち三進数の表記で 00000000000000000000t から 22222222222222222222t の範囲である。ここで、0012t のように末尾に t をつけることで、それが三進数表記であることを表す。

ソースコード上の 1 文字はメモリ上の 1 ワードに対応し、0 番地から順に ASCII コードが (xlat1 による変換なしに) 格納される。プログラムが書き込まれなかった残りのメモリ m は、あとで述べる演算 op を用いて、 $op(mem[m-1], mem[m-2])$ の値に、フィボナッチ列のように初期化される。メモリには命令しかロードできないため、Malbolge20 のメモリの初期値には著しい制約があり、プログラミングが困難である理由の一つになっている。

プログラムはレジスタ値を全て 0 に初期化の後に実行され、C レジスタが指す命令を表 1 に従って実行し、命令実行後に変換表 xlat2 を用いて C レジスタが指すメモリ値を変えこれにより、実行後に変化する。一つの命令を実行すると、C と D にそれぞれ 1 を加算した後に次の命令の実行に移る。

表 1 に Malbolge20 の各命令についての説明とそのニーモニック表記を示す。これ以降で Malbolge20 の命令は、便宜的に表 1 の表記で **Jump** などのように表す。

ここで、 $[i]$ はアドレス i のメモリ値を表し、また、 $[i] := j$ はアドレス i のメモリに値 j を格納することを表す。

Opr 命令の演算 $op(X, Y)$ は、三進数で表された 2 つの引数の各桁同士を表 2 に従って計算を行う。以下に op 演算の例を示す。

```
op ( 01201201200001112222t,
    00011122220120120120t )
= 10010222121120020210t
```

Rot 命令の演算子 **rotr** は 20trits の値に対して以下のように

表 1 Malbolge20 の命令

xlat1 による値	表記	説明
i	Jump	ジャンプ。C:=mem[D].
j	Mov_d	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).
v	Halt	終了。プログラムの実行を停止.
o	Nop	無操作。何も行わない.

表 2 op の各桁の演算

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

各桁を右にシフトする。

$rotr(00\cdots\cdots222t) = 200\cdots\cdots22t$

変換表 xlat2 による実行後のコードの書換えのため、あるコードを実行した時、ほとんどの場合実行後にそのコードは全く別の値に書き換えられてしまい、Malbolge20 ではコードの反復実行が非常に困難になっている^(注1)。変換表 xlat2 は xlat1 による変換と同様に文字の値とその位置に依存した表示可能文字上の順列であり、任意の表示可能文字についてこの変換を繰り返し行うことで元の文字に戻すことが可能である。この xlat2 の変換サイクルを表 3 に示す [2]。どの変換サイクルに属するかは、変換される文字のみに依存して決まる。Malbolge20 の

表 3 xlat2 の変換サイクル

周期=2	F → J → F
周期=4	* → r → } → i → *
周期=5	) → f → ‘ → < → 3 →)
周期=6	% → g → u → o → x → : → %
周期=9	2 → P → B → > → L → O → C → U → I → 2
周期=68	! → 5 → - → w → N → ... → & → q → D → !

命令に該当しない印字可能な文字はプログラムとしてロードすることは不可能であるが、**Nop** として動作することは可能である。任意のアドレスに対して、xlat2 の書換えに対して安定な、すなわち、常に **Nop** として働くことのできる文字が存在している。これを **SNop** と呼ぶ。

3. Malbolge20 プログラミングにおける問題点と解決策

本節では、Malbolge20 用低級アセンブラ [8] を用いた Malbolge20 プログラミングにおける問題点と解決策を述べる。Malbolge20 用低級アセンブラは Malbolge20 用低級アセンブ

(注1) : **Jump** 命令の場合のみ、実行後に C レジスタが指す値はジャンプ先のアドレスであるため実行した **Jump** は変換されない。

リ言語から Malbolge20 プログラムを作成するツールである [8]. 以下では Malbolge20 用低級アセンブリ言語とそのプログラム、並びに、Malbolge20 用低級アセンブラのことを、それぞれ単に低級アセンブリ言語、低級アセンブリプログラム、低級アセンブラという。Malbolge20 プログラミングにおける問題点としては、低級アセンブリプログラムを記述する際に発生する問題と、低級アセンブラを使用する際に発生する問題の大きく分けて 2 つの問題がある。

3.1 低級アセンブリプログラム記述上の問題点

まず、低級アセンブリプログラムを記述する際の問題点は、命令ユニットと呼ばれる Malbolge20 コード群の記述の煩雑さにある。まず、これまでの低級アセンブリ言語の概要と命令ユニットについて説明する。

DATA	# データ部	# ROT の命令ユニット
ENTRY:		300:58 : SNop
ROT.CON2:	ROT CON2	ROT_UNIT:
CON2:	2222222222222222222t	\$:59 : Rot
	REV ROT	REV_ROT: # 復元命令用ラベル
OPR.X:	OPR X	\$:60 : Jmp
X:	00000000000000000001t	
	REV OPR	# OPR の命令ユニット
	IF FLAG1	\$:81 : SNop
	BRANCH OPR.X	OPR_UNIT:
	END	\$:82 : Opr
		REV_OPR:
CODE	# コード部	\$:83 : Jmp
# 終了命令		
END:	# ラベル付加	# フラグ 1(初期状態 ON)
300:09	: HALT	301:59 : SNop
		FLAG1: # 周期 2
		\$:60 : Mov.d
		\$:61 : Jmp

(右側に続く)

図 1 低級アセンブリプログラムの例

低級アセンブリプログラムは、図 1 のプログラム例のように、データ部とコード部の 2 部で構成される。データ部にはプログラム本体のデータ列を記述し、コード部にはデータ部のプログラムを実行するために必要な Malbolge20 コードを直接記述する。なお、命令ユニット [7] もコード部に記述する。

“DATA” で始まるデータ部には、基本的に表 4 に示す低級アセンブリ命令もしくはデータの系列を記述する。これらの前にはラベルをつけてもよい。なお、DUP を、値が指定できないデータとして用いることができる。

一方、“CODE” で始まるコード部には、Malbolge20 のコードを直接記述する。ここに記述するコードには主に、低級アセンブリプログラムの動作に必要な命令ユニットを記述する。命令ユニットはデータ部に記述された命令を実行するための Malbolge20 のコードである。Malbolge 及び Malbolge20 では、Jmp 以外の命令はその実行後に変換表 xlat2 にしたがって別の値に書き換えられるため、特定の命令を反復実行することが困難であるとされていた。この問題を解決するために考案されたものが低級アセンブリ言語であり、その基本的なアイデアが命令ユニットである。データ部に記述される低級アセンブリ

表 4 低級アセンブリ言語の命令

命令	動作
ROT label	A,[label] := rotr([label]), D := label+1
REV ROT	ROT の復元命令, D:=D+1
OPR label	A,[label] := op([label]), D := label+1
REV OPR	OPR の復元命令, D:=D+1
SKIP label	D:=label
JMP label	D:=[label]
REV JMP	JMP の復元命令, D:=D+1
OUTPUT	putchr(A); D:=D+1
REV OUTPUT	OUTPUT の復元命令, D:=D+2
INPUT	A:=getchr(); D:=D+1
REV INPUT	INPUT の復元命令, D:=D+2
IF FLAG_name	フラグが ON の時, D:=[label]
BRANCH label	フラグが OFF の時, D:=D+2 実行後にフラグのサイクル値がインクリメントされる
NEXT FLAG_name	フラグのサイクル値をインクリメント, D:=D+1

- ※ ROT, OPR, SKIP においては、 $D < \text{label} \leq D+n$ (n は定数) を満たす必要がある。
- ※ ROT, OPR, OUTPUT, INPUT, JMP 命令の後にはそれぞれに対応した復元命令を実行する必要がある。
- ※ 命令の引数として用いられるデータラベルの記述は、必ずその命令よりも後で定義されている必要がある。

言語の命令は Malbolge20 の命令ではなく、命令ユニットの置かれているアドレスが置かれているだけである^(注2)。

まず例を用いて命令ユニットの動きを理解しよう。図 2 の左側が OPR の命令ユニットであり、いくつかの Malbolge20 の命令が置かれている。また、右側はデータ部のデータであり、図 1 データ部の低級アセンブリ命令 OPR から始まる 4 行に対応している。先に述べたように、この OPR 命令は命令ユニットのアドレス 81 がデータとして置かれている。C レジスタ値が 90, D

adr	label	命令
81	—	SNop
82	OPR_UNIT	Opr
83	REV_OPR	Jmp
⋮	⋮	⋮
90		Jmp

C→

adr	ラベル	データ
100	OPR.X	81(=OPR_UNIT-1)
101	X	1
102	—	82(=REV_OPR-1)
103	—	108(=FLAG1-1)

D→

図 2 命令ユニットの実行

レジスタ値が 100 であるとする、以下のように実行される。

- (1) 90 番地の Jmp が実行され C レジスタが 81 に設定される。
- (2) C,D をインクリメントし次のステップに移る。X に対して 82 番地の Opr 命令が実行され、その結果が A レジスタと X に格納される。
- (3) C,D をインクリメントし次のステップに移る。83 番地の Jmp 命令が実行され、C レジスタが 82 に設定される。

(注2)：低級アセンブリ言語の命令が置かれる場所をデータ部と呼ぶのはこのためである。

- (4) C,D をインクリメントし次のステップに移る．83 番地の **Jump** 命令により C レジスタが 108 に設定される．

このようにして，データ部の低級アセンブリ命令が，コード部の **Malbolge20** 命令によって実現される．ここで，実行された **Malbolge20** 命令は以下のように別の命令に書換えられる．(1) の **Jump** の実行では，C レジスタの指す 81 番地が **xlata2** により書き換えられるが，ここには **SNop** が置かれているため実質的な変化はない．(2) の **Opr** の実行では，82 番地の **Opr** 命令が他の文字に書き換えられる．(3) の **Jump** の実行では，(2) で書き換えられた命令が再び書換えられる．ここで，82 番地に，周期 2 の **Opr** 命令 (**Opr/Nop** と記述する) を予め置いておくことで，**Opr** の復元に成功する．

以上で述べた例の方法においては，低級アセンブリの命令が扱うデータが命令の直下に置かれている必要があり，制限が強すぎる．これに対しては，命令ユニットの上部に k 個の **SNop** 命令をあらかじめ準備しておくことで，下方に $k+1$ 番地までのデータにアクセスできるように制限を緩められる．例えば図 3(a) の低級アセンブリ命令列を考えると，一行目の **ROT** の引数である **CON2** は 5 番地下に置かれている．この場合は，“**ROT**

<pre> ROT CON2 LABEL1: INPUT DUP REV INPUT OPR CON2 CON2 : 2222222222222222222t IF FLAG1 BRANCH LABEL1 </pre>	<pre> SNop SNop SNop SNop ROT_UNIT: Rot/Nop REV_ROT: Jump </pre>
(a) 低級アセンブリ命令列	(b) 命令ユニット

図 3 離れたデータへのアクセス

CON2” に対して **ROT** ユニットの **Rot** 命令より 4 番地上の **SNop** を実行するようなアドレスを置いておく．これにより **SNop** が 4 回実行され，実際に命令ユニット内の **Rot** の実行がなされる際にはちょうど D レジスタが **CON2** の位置に到達する．このように命令ユニットの呼び出しの場所に応じて，低級アセンブリプログラマが必要数の **SNop** を準備しておく必要があった．

もう一度，図 3 のプログラムに戻ろう．(a) の 1 行目の **ROT** により 2 行目から 5 行目の低級アセンブリ命令は実行されない．このため，本質的に繰り返しが不要なプログラムであっても，2 行目からのコードを実行するためには **LABEL1** まで戻ってくる必要が生ずる．そのために，低級アセンブリプログラムに欠かせない命令ユニットであるフラグが用いられる．フラグ命令ユニットは命令ユニット内で **Malbolge20** の **Mov_d** 命令が置かれていること以外は他の命令ユニットと同じ構造をしている．

	SNop
FLAG1:	Mov_d/Nop
REV_FLAG1:	Jump

図 3(a) の “**IF FLAG1**” は，フラグユニットの **FLAG1** に対応するアドレスが，“**BRANCH LABEL1**” には **LABEL1** のアドレスが

置かれるため，フラグユニット内の **Mov_d** 命令が実行されると D レジスタに **LABEL1** が入り，低級アセンブリ命令上では **LABEL1** にジャンプすることができる．ここで，フラグユニット内の **Mov_d** 命令は **Nop** に置き換えられるため，二度目の実行では **LABEL1** へのジャンプは行われず次の命令に実行を移すことが可能になる．すなわち，フリップフロップの状態に基づくジャンプ制御を行っているのとらえることができる．ここで，**Mov_d** などの命令の周期 i はそれが置かれるアドレスに依存して定まるため，表 3 中の $i = 2, 4, 5, 6, 9, 68$ の周期が可能である．例えば，**Mov_d** は最初の実行以外は **Nop** として動作し，5 回の実行により **Mov_d** に復元されるようなものが存在する．

このように，定数回の繰り返しを行う低級アセンブリプログラムを作成する上で複数のフラグを駆使する必要があるが，プログラマがこれをコード部に準備しなくてはならない．そのためには，適切なフラグとして動作する周期の **Mov_d** を得るための，コードを置くアドレスなどの知識が必要である．

以上の議論で得られた低級アセンブリプログラム作成上の問題点をまとめると以下になる．

- (a) 命令ユニットをプログラマが準備してコード部に記述する必要がある．
- (b) 命令ユニットはどのアドレスに置くかによって周期が変わるため，適切な番地をプログラマが知っている必要がある．
- (c) 命令ユニットの上部には，低級アセンブリプログラムの書き方に依存した，適切な数の **SNop** を置く必要がある．
- (d) 特にフラグについては，2 以外の周期も含む複数のフラグが利用される．

これらの煩雑さを回避するために，低級アセンブラがこれらの必要な命令ユニットを自動的に準備できるようにする．すなわち，低級アセンブラが低級アセンブリプログラムを解析し，使用されている命令に応じて必要な命令ユニットを準備するように変更する．

まず (a) と (b) に対処するため，表 5 の命令ユニットをコード部への宣言なしに使用できるようにした．ここで，アドレスの 94 に対する剰余を下位番地と呼ぶ．(c) への対処として，データ部に記述されたアセンブリプログラムから得られる，各命令ユニットに対して必要な **SNop** 命令数を用意する．表 5 中においては，値が自動的に決定されるという意味で * と記述してある．

(d) に関しては，複数のフラグが必要であることから，宣言部を追加し，そこで名前，周期，初期状態を指定することでフラグの命令ユニットが生成されるように変更した．ここで，周期は $i = 2, 4, 5, 6, 9$ の中から指定可能であり，初期状態は 0 から $i-1$ の値が指定可能であり，状態 0 はフラグによるジャンプが起こる状態に対応する．フラグの命令ユニットの一覧を表 6 に示す．ここで，**Mov_d(i, j)/Nop** は周期 i で初期状態 j のフラグを表す．表 6 中の m と n の値は表 7 から定められる．例えば，周期 5 で初期状態 2 のフラグの場合には $m = 60$ ， $n = 32$ が求められる．

3.2 低級アセンブラの利用時の問題点

ここでは低級アセンブラの利用時の問題点とその解決につい

表 5 改良版低級アセンブラが用意する命令ユニット

	下位番地	ラベル	命令
HALT ユニット	0	END	Halt
	24	—	SNop
	25	OUTPUT	Output/Nop
OUTPUT ユニット	26	—	Jmp
	*	—	SNop
	⋮	⋮	⋮
SKIP ユニット	39	—	SNop
	40	SKIP	Jmp
	46	—	SNop
INPUT ユニット	47	INPUT	Input/Nop
	48	—	Jmp
	*	—	SNop
ROT ユニット	⋮	⋮	⋮
	59	ROT	Rot/Nop
	60	—	Jmp
JMP ユニット	*	—	SNop
	⋮	⋮	⋮
	64	JMP	Mov_d/Nop
OPR ユニット	65	—	Jmp
	*	—	SNop
	⋮	⋮	⋮
OPR ユニット	86	OPR	Opr/Nop
	87	—	Jmp

表 6 設定されるフラグ用命令ユニット

	下位番地	ラベル	命令
FLAG ユニット	$n-1$	—	SNop
	n	指定したラベル名	Mov_d(i, j)/Nop($=m$)
	$n+1$	—	Jmp

表 7 各周期の各状態を実現する ASCII 値 m と下位番地 n

		状態 j における m										n (下位番地)
		0	1	2	3	4	5	6	7	8		
周期 i	2	70	74									64
	4	42	114	125	105							92
	5	102	96	60	51	41						32
	6	37	103	117	111	120	58					3
	9	50	80	66	62	76	79	67	85	73		84

て述べる。

まず低級アセンブラの動作方法を概説する。低級アセンブラは低級アセンブリ言語のプログラムを入力として受け取り、以下の手順で Malbolge20 プログラムを出力する。

- (i) アセンブリプログラムを表現するメモリイメージの作成
- (ii) メモリイメージを実際にメモリに展開する Malbolge20 プログラムの作成

(i) では、低級アセンブリプログラムを読み込み、与えられた低級アセンブリプログラムのコード部とデータ部を Malbolge20 メモリへの配置を表すメモリイメージを作成する。ところが、Mlabolge20 のメモリ初期値の制約から、(i) で生成されたメモリイメージ中の多くの値は Malbolge20 命令として直接記述出来ない。このため (ii) で、実行によりメモリイメージファイル

通りのメモリ状態を作成する Malbolge20 を生成する。イメージ作成後に初期の実行開始アドレスにジャンプするようにしておくことで、与えられた低級アセンブリプログラムを実現する Malbolge20 プログラムが生成される。図 4 に、メモリの利用配置図を示す。この中で、(B) と (C) が実行される Malbolge20 のプログラム部分であり、(A) と (D) が実行途中で生成される部分である。

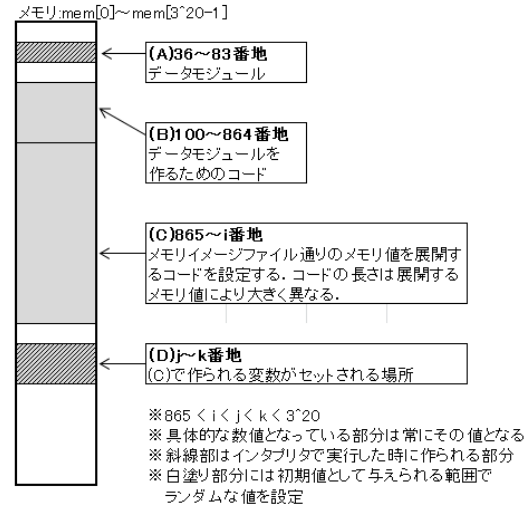


図 4 低級アセンブラによって作成される Malbolge20 プログラム

(ii) はさらに以下の手順に分割して実現されている。

- (a) データモジュール作成コードの生成
- (b) 特定の値を設定するコードを生成
- (c) 初期化用末尾調整

(a) では、図 4 における (B) の場所に、任意のメモリ値の作成のために必要なデータモジュールを (A) に作成するコードを生成する。データモジュールは飯澤らによって経験的に発見 [7] されたものであり、データモジュール作成コードは入力のアセンブリプログラムによらず一定である。

(b) では、図 4 における (C) の場所に、(i) で得られたメモリイメージ通りにメモリの値を (D) に設定するコードを生成する。以下では、(C) の場所に生成されるコードのことをメモリ値設定コードと呼ぶ。

(c) では、Malbolge20 特有の残りメモリの初期化処理が行われた際、生成される Malbolge20 プログラムの前提となる初期値となるように末尾 2 文字を設定する。

低級アセンブラを利用する際の問題点は、ステップ (i) でメモリイメージファイルを作成する際に必要な (D) の番地 j, k を指定する必要があることである。(C) のメモリ値設定のためのコード長は j, k の値によって大きく変化するためメモリ値設定コードの長さを予測することは困難であり、本プログラミング手法を熟知したユーザであっても試行錯誤が必要であった。これはプログラミングを困難にする大きな要因の一つとなっていた。

この問題の解決のために、低級アセンブラが用いているメモリ値設定コードの生成手法、すなわち、1trit ずつ目的の値に設定していく手法 [7] について解析する。この手法は、変数 VAR の最下位桁の値が 1 のときに、以下の計算により VAR の最下位桁以外の値を変化させることなく最下位桁の値を更新できることを利用している。ただし、最下位を 0 にしたい場合には $i=1$ 、1 にしたい場合には $i=2$ 、2 にしたい場合には $i=0$ とする。

$$\text{VAR}' := \text{op}(22222222it, \text{VAR}) \quad (1)$$

$$\text{VAR} := \text{op}(222222222t, \text{VAR}') \quad (2)$$

例えば 0120120120t を作成する場合は、1111111111t が格納された VAR に対し $1111111111t \rightarrow 1111111110t \rightarrow 1111111120t \rightarrow 1111110120t \rightarrow \dots \rightarrow 0120120120t$ というように、(1) と (2) の計算、ならびに、右ローテート計算により、1trit の値の作成と左シフトを繰り返すことで実現される。

このように 1trit ずつ設定していく手法では、ほとんどの計算が (1) と (2) を繰り返すだけであるため、メモリ値設定コード長は設定するメモリ数に比例して増加する。実際に解析した結果、1 つの 20trits のデータを展開するために生成されるコード長が平均して 800 程度であることが分かった。したがって、(設定するデータ数) $\times$ (800 + α) をメモリ値設定コード長の予測値とした。ここで α は平均値より少し多めに設定するための定数である。低級アセンブラがステップ (ii-a) で作成するデータモジュール作成コードの長さ (図 4(B) の範囲) は固定で 864 番地まで使用するため、以下のようにして図 4 の i, j, k の値を決定することでコード配置アドレスを自動的に設定する。

$$i := 864 + (\text{設定するデータ数}) \times (800 + \alpha)$$

$$j := i + 1 \text{ 以上の値}$$

$$k := j + (\text{設定するデータ数})$$

4. ま と め

本稿では、低級アセンブリプログラムの記述における煩雑さを、命令ユニットを自動設定するように修正することで解消した。また、低級アセンブラを用いたプログラミングの際に必要なユーザの試行錯誤によるコード配置アドレスの決定を低級アセンブラが行うように変更し、プログラミングの効率を向上させた。なお、本稿で示した問題点の解決を行った改良版低級アセンブラは、Web インタフェースを <http://www.trs.cm.is.nagoya-u.ac.jp/Malbolge/> にて公開予定である。

改良版低級アセンブラによりユーザの試行錯誤は必要なくなるものの、低級アセンブリ言語には通常のプログラミング言語が持つような演算命令がなく、Malbolge20 特有の演算を行う命令のみであるため、改良版低級アセンブラでのプログラミングには依然困難が伴う。この問題は従来版にも存在し、解決策として低級アセンブリプログラム作成のための高級アセンブリ言語が飯澤らによって構築され [7]、安藤らによって使いやすさ

が改良されている [3] [4] [5]。しかしながら、高級アセンブリプログラムはライブラリ方式によりインタプリタに近い形で実現されているため、適切な時点で実行を停止させてメモリの内容を見ることで高級アセンブリプログラムを復元することは不可能とは言えない。安全性を高めるためには、高級アセンブリプログラムを低級アセンブリプログラムに変換するほうが望ましいと考えられる。本稿で述べたような低級アセンブリプログラム環境の改善は、この目的のために役立つと考えられる。

低級アセンブラには、変数や A レジスタの値にしたがって分岐する機能は用意されていない。そのため、高級アセンブリ言語のインタプリタの実現では、依然としてコード部にトリッキーな Malbolge コードを記述している。これらを洗練した形で低級アセンブリ言語に取り込むことは重要な今後の課題である。

謝辞 本研究は、一部 JSPS 科研費 26540027 の補助を受けている。

文 献

- [1] Ben Olmstead : “Malbolge: Programming from Hell”, <http://www.bouletfermat.com/danny/malbolge/>, 1998.
- [2] Lou Scheffer: *Introduction to Malbolge*, <http://www.lscheffer.com/malbolge.html>.
- [3] 安藤聡, 酒井正彦, 坂部俊樹, 草刈圭一郎, 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, 2011.
- [4] 安藤聡, 酒井正彦, 坂部俊樹, 草刈圭一郎, 西田直樹, Malbolge の高級アセンブリ言語への配列機能の追加, 信学技報, 電子情報通信学会, vol.112, No. 23, pp. 43–48, 2012.
- [5] 安藤聡, 難解言語 Malbolge のプログラム作成のための中間言語の拡張と開発支援, 名古屋大学修士論文, 2013.
- [6] 飯澤恒, 坂部俊樹, 酒井正彦, 草刈圭一郎, 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.
- [7] 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.
- [8] 加藤起騎, 酒井正彦, 坂部俊樹, 草刈圭一郎, 西田直樹, Malbolge のワード長の拡大とそのプログラミング支援ツール, 信学技報, 電子情報通信学会, Vol.113, No. 159, pp. 73–78, 2013.
- [9] 長坂哲, 難解言語 Malbolge の弱チューリング完全性とプログラミング環境, 名古屋大学修士論文, 2011.