日本ソフトウェア科学会第 28 回大会 (2011 年度) 講演論文集

Malbolgeの高級アセンブリ言語への加算命令の追加

安藤 聡　酒井 正彦　坂部 俊樹　草刈 圭一朗　西田 直樹

Malbolge は最も 難解なプログラミング言語の一つとして知られており，その難解性から Malbolge を用いてプログラミングを行うこと自体がプログラム難読化になり 得ることが飯澤らによって提案された．その中で，Malbolge プログラミングをより 効率よく 行う ための高級アセンブリ 言語が設計されたが，高級アセンブリ 言語の算術命令にはインクリメントとデクリメントしか存在せず，加算を行う にはインクリメント を繰り 返し 用いる必要があった．本論文では，高級アセンブリ 言語への加算命令の追加を行う．これを処理するために，Malbolge 向きの加算アルゴリ ズムが何かを考え，そのアルゴリ ズムを低級アセンブリ 言語で実装する．

1　はじめに

作成したプログラムを不特定多数のユーザーに配布する場合，プログラムの改ざん防止やアルゴリ ズムなどの知的財産の保護を目的として，プログラムの内部を解析困難にするよう に求められる 場合がある．そのよう な場合には，プログラムを解析が困難になるよう に変換する 手法が有用であると 考えられる．このよう な同一言語上のプログラムの等価変換は，これまで**プログラム難読化**と 呼ばれている [1].

これに対し，難解プログラミング言語を用いてプログラミングを行うことがプログラム難読化の手法の一つになりえることが飯澤らにより 提案された [2]. 難解プログラミング言語(以降，難解言語) とは，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り，このよう な言語で書かれたプログラムは解析困難性をもつため，ソフトウェアプロテクショ ンに関して有益であると 考えられている．文献 [2] では，難解言語の中でも 特に難解として知られている **Malbolge** [3] が用いられている．

```
(=<‘$9]7<5YXz7wT.3,+O/o’K%$H"’˜D|#z@b=‘{ˆLx8%$X
mrkpohm-kNi;gsedcba‘_ˆ]\[ZYXWVUTSRQPONMLKJIHGFE
DCBA@?>=<;:9876543s+O<oLm
```

図 1　HEllO WORld 出力プログラム

本研究の目的は，飯澤らの成果を基に Malbolge プログラミングをさらに容易にすることである．

Malbolge は，その難解性からプログラムの解析だけでなくプログラムの作成さえ非常に困難であり，“人類が設計しうる最も 邪悪な言語” などど 称される．図 1 に，Malbolge プログラムのサンプルとして “HEllO WORld” を出力するプログラムソース [4] を示す． Malbolge プログラミングを困難にしている理由としては，次の三点が主に挙げられる．

- プログラムロード 時のメモリ の初期値に著しい制約がある
- 命令の強制的な動的書き換えにより，コードの反復実行が困難である
- 演算用・制御用とも に極めて限定的な命令しか持たない

これらの各問題点に対し 飯澤らは解決策を提案し，それに基づき Malbolge のための言語として，Malbolge の演算命令を拡張した疑似命令，ループプログラム

Introducing Addition Instruction into High-Level Assembly Language for Malbolge.

Satoshi Ando　Masahiko Sakai　Toshiki Sakabe　Keiichirou Kusakari　Naoki Nishida, 名古屋大学 大学院情報科学研究科, Graduate School of Information Science, Nagoya University.

が作成可能な低級アセンブリ言語，ならびに，低級アセンブリプログラムを用いて基本モジュール方式で実現される高級アセンブリ言語を設計した．しかし，これらの言語の中で最も命令が豊かである高級アセンブリ言語においても，算術命令にはインクリメントとデクリメントしか存在しておらず，これまで加算を行うためにはインクリメントを繰り返し用いらなければならなかった．

本論文では，高級アセンブリ言語へ加算命令を追加し，次のように実現した．

- 低級アセンブリ言語での加算モジュールの作成．これは飯澤らの提案したプログラミング手法に従い，以下の順に行った
  - **Step1** Malbolge 向きの加算アルゴリズムの検討
  - **Step2** Step1 のアルゴリズムの疑似命令列での実現
  - **Step3** Step2 の疑似命令列の低級アセンブリプログラムへのコーディング
- 加算モジュールを用いた加算命令の実現

本論文の構成もこの過程の順に従い，第 2 節で Malbolge と飯澤らが設計した言語ついて述べたのちに，第 3 節で Step1 について，第 4 節で上記の Step2 で用いる，Malbolge プログラミングで重要となる二引数三値関数を実現する疑似命令列の探索手法について，第 5 節で Step2，Step3 について述べる．そして第 6 節で作成した加算モジュールを用いた加算命令の実現について述べ，最後に第 7 節で本論文で行ったこと・明らかになったことをまとめる．

2 Malbolge とその関連する言語について

本節では Malbolge の仕様についてその重要な部分を説明する．また飯澤らが設計した Malbolge のための言語である，疑似命令，低級アセンブリ言語，高級アセンブリ言語のそれぞれについて概説する．

2.1 Malbolge

Malbolge [3] は仮想機械上で動作する機械語として定義され，開発者が C 言語で実装したインタプリタによってその意味が定められている．

```
const char xlat1[] =
       "+b(29e*j1VMEKLyC})8&m#~W>qxdRp0w"
       "krUo[D7,XTcA\"lI.v%{gJh4G\\-=O@5"
       "‘_3i<?Z’;FNQuY]szf$!BS/|t:Pn6^Ha";
```

図 2　変換表 xlat1

```
const char xlat2[] =
       "5z]&gqtyfr$(we4{WP)H-Zn,[%\\3dL+"
       "Q;>U!pJS72FhOA1C"B6v^=I_O/8|jsb9"
       "m<.TVac‘uY*MK’X~xDl}REokN:#?G\"i@";
```

図 3　変換表 xlat2

Malbolge プログラムは 59048 文字未満の印字可能文字（ ASCII 値:33 ～126）の列であり，スペースと改行は無視される．また，各 p 文字目における印字可能文字 c は，変換表 `xlat1[(c-33+p)%94)]` による解釈で i，j，p，*，/，<，v，o の 8 文字のいずれかに該当せねばならず，それ以外の場合はロード時にエラーとなる．ここで `xlat1` は図 2 のように定義されている． また実行ステップ中に実行された命令 c は，変換表 `xlat2[c-33]` によって置換され別の文字（ 値）に書き換えられる．ここで `xlat2` は図 3 のように定義されている．

Malbolge の仮想機械はメモリ（ mem）と 3 つのレジスタ（ コードポインタ C，データポインタ D，アキュムレータ A）を持ち，1 ワードは三進数十桁（ 10trits）であるためメモリ領域も 10trits で表現される．このため値の範囲は，三進数表現で 0000000000t～2222222222t，十進数表現では 0～59048（ $= 3^{10} - 1$）となる．ここで，三進数表現の末尾に t を用いているのは十進数と区別するためである．

表 1 に Malbolge 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．これ以降では，Malbolge の命令を便宜的にニーモニック表記で記述する．ここで，mem[p] は，アドレス p のメモリ値を表している． `OPR` 命令の演算 op(X,Y) は，三進数で表された二引数 $X = x_1x_2\cdots x_{10}$ と $Y = y_1y_2\cdots y_{10}$ の各桁 x_i，y_i 同士で表 2 に従って計算を行う．このように三進数で表された二引数の各桁である表に基づき計算を行う関数を**二引数三値関数**と呼ぶ．文献 [2] において，op 関数を合成することによって任意の二引数三

表 1　Malbolge の命令とその説明

命令	説明とニーモニック表記
i	ジャンプ．C:=mem[D].'JMP'
j	D レジスタの更新．D:=mem[D].'MOV_D'
p	演算命令．A,mem[D]:=op(A,mem[D]).'OPR'
*	右ローテート．A,mem[D]:=rotr(mem[D]).'ROT'
/	入力．A:=getchar().'INPUT'
<	出力．putchar(A).'OUTPUT'
o	無操作．何も行わない．'NOP'
v	終了．プログラムの実行を停止．'HALT'

表 2　op(X,Y) の各桁の演算

		x_i		
		0	1	2
y_i	0	1	0	0
	1	1	0	2
	2	2	2	1

表 3　疑似命令

命令	操作
ROT X	A,X := rotr(X)
OPR X	A,X := op(A,X)
INPUT	A := getchar()
OUTPUT	putchar(A)

値関数を実現できることが示されている．以下に op 演算の例を示す．

```
op(0120120120t, 0001112222t) = 1001022212t
```

ROT 命令の演算子 rotr は引数の 10trits の値に対して右ローテートを行う．以下に例を示す．

```
rotr(0001112222t) = 2000111222t
```

2.2　疑似命令

疑似命令[2] は，Malbolge の演算命令や入出力命令の表現をより わかり やすくし，引数にメモリ名を指定できるようにしたものである．命令列の実行は逐次実行のみとし，反復実行や条件分岐は考えない．

表 3 に疑似命令とその操作について示す．ここで，疑似命令列の引数に特別な変数として $CON0$(初期値: 0000000000t)，$CON1$(初期値: 1111111111t)，$CON2$(初期値: 2222222222t)，$PAT20$(初期値: 2222222220t) を利用できるものとする．

図 4 に例として，入力された値を X に代入し，その値を出力する疑似命令列を示す．ここで，各変数 X，Y の初期値は任意の値とする．

```
ROT CON1          INPUT
OPR Y             OPR Y
OPR Y             OPR X
OPR X             OUTPUT
OPR X
```

図 4　疑似命令列の例

表 4　命令・フラグとその操作

命令	操作
ROT *label*	A,[*label*] := rotr([*label*])
REV_ROT	ROT 命令の復元
OPR *label*	A,[*label*] := op(A,[*label*])
REV_OPR	OPR 命令の復元
JMP *label*	PC:= [*label*]
MOV_PC *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*	命令が MOV_PC である場合 (ON):MOV_PC と同様 命令が NOP である場合 (OFF):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)

2.3　低級アセンブリ言語

低級アセンブリ言語[2] は，強制的な命令の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言語である．

低級アセンブリ言語は 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．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 4 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 レジスタである．

表 4 から分かるように，JMP 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

```
L1:       REV_MOV_PC     # (ラベル):(データ)
          JMP  L2
ENTRY:    ROT CON1       # エントリーラベル
CON1:     1111111111t    # 三進数表現の定数
          REV_ROT
L2:       OPR Y
Y:        0              # 十進数表現の定数
          REV_OPR
          FLAG2+1
          FLAG2  L3
          FLAG1  L2
L3:       OPR X
X:        0
          REV_OPR
          FLAG2+1
          FLAG2  L6
          FLAG1  L3
L4:       INPUT_UNIT
          DUP
          REV_INPUT
          FLAG2+1
          MOV_PC L1
L6:       OUTPUT_UNIT
          DUP
          REV_OUTPUT
RETURN:   END
```

図 5　図 4 の低級アセンブリプログラム

には，プログラ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．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．

図 5 に，図 4 の疑似命令列を低級アセンブリプログラムに変換した例を示す．ただし，ここで，“#” から改行までをコメントとし，“ENTRY” を PC の初期値を表すエントリーラベルとする．

2.4　高級アセンブリ言語

高級アセンブリ言語 [2] 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された言語である．このため，値の取る範囲は 10trits である．

高級アセンブリプログラムは，アドレスを表す省略可能なラベルがついた命令「（ ラベル名）:（ 命令）（ 引数）」の列で定義される．特に “PROGRAM INIT” というラベルはエントリーポイントを表す．命令の引数には変数かラベルを用いる．例として図 6 に高級アセンブリプログラムの例を示す．これは，標準入力により入力された文字を標準出力するプログラムであり，EOF（ =59048）が入力されるまで繰り返し実行される．ここでプログラム中の変数 $Zero$ は 0 を表している．

```
PROGRAM_INIT:
Loop:     INPUT Var_X
          MOV Var_X,Var_Y
          INC Var_Y
          BRANCH Var_Y,Var_Exit
          OUTPUT Var_X
          BRANCH Zero,Var_Loop
Exit:     STOP
```

図 6　高級アセンブリプログラムの例

表 5　命令と操作

命令	操作
INC $label$	$[label]$:= $[label]+1$
DEC $label$	$[label]$:= $[label]-1$
MOV $label1$,$label2$	$[label2]$:= $[label1]$
BRANCH $label1$,$label2$	$[label1]=0$ の時,IP:=$[label2]$ それ以外の場合何もしない
INPUT $label$	$[label]$:=getchr()
OUTPUT $label$	putchr($[label]$)
MASK $label1$,$label2$	$[label2]$:=mask($[lable1]$,$[label2]$)
ROTATE $label$	$[label]$:=rotr($[label]$)
STOP	実行の停止

表 6　関数 mask(X,Y) の表

		x_i		
		0	1	2
y_i	0	0	0	0
	1	2	0	1
	2	0	1	2

高級アセンブリプログラム内の記述で使用できる命令を表 5 にまとめた． INC 命令の引数の値が 59048 のときの演算結果は 0，DEC 命令の引数の値が 0 のときの演算結果は 59048 とする．MASK 命令で使用する関数 mask(X,Y) は表 6 に従い計算を行う 二引数三値関数である．IP については第 2.5 節で説明する．

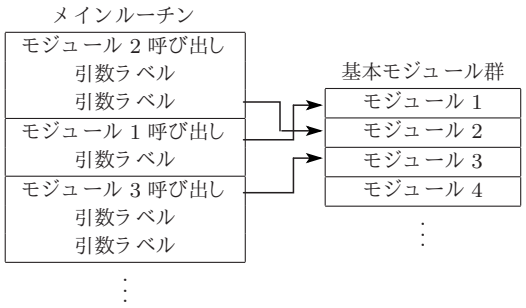

図 7 基本モジュール方式

2.5 高級アセンブリプログラムの実現

高級アセンブリプログラムは，基本モジュール方式 [6] を利用して低級アセンブリプログラムで実現されている [2] [5]．これは，ある仮想マシンを Malbolge 上に実装しており，その仮想マシンはプログラムカウンタ IP とレジスタ REG0，REG1 を持つ．レジスタや IP は低級アセンブリプログラムの特定の変数であるため，低級アセンブリ言語のレジスタ（ A，PC）とは異なる．この低級アセンブリプログラムを**高級アセンブリ言語の実行系**と呼ぶ．

高級アセンブリプログラムの実現に用いた**基本モジュール方式**とは，プログラミングを容易にすることを目的として定義されたもので，基本モジュールと呼ばれるある機能を持たせた命令列と，それらをサブルーチンコールのように呼び出すメインルーチンで構成される（ 図 7）．高級アセンブリプログラムはこのメインルーチンにあたり，高級アセンブリ言語の各命令と同じ機能を持つ低級アセンブリプログラムを基本モジュールとして呼び出している．

2.6 言語間の関係

これまでに述べてきた言語について，高級アセンブリプログラムから低級アセンブリプログラムに変換する手法 [2] [5] と低級アセンブリプログラムから Malbolge プログラムに変換する手法 [2] が提案されている．後者の変換を行うアセンブラ [2] が存在する．ただしこの**低級アセンブラ**は，入力プログラムに合わせてエントリーラベルのアドレス値の指定やデータのメモリ領域の指定を低級アセンブラのソースコードに記述する必要がある．疑似命令列から低級アセンブリプログラムへの変換は実現されておらず，ノウハウを用いて手変換する必要がある．

表 7 sum(X,Y) の表

		x_i		
		0	1	2
y_i	0	0	1	2
	1	1	2	0
	2	2	0	1

3 Malbolge 向けの加算アルゴリズム

これまで Malbolge における加算はインクリメントを用いてのみ実現されていた．しかし $n_1 < n_2$ の加算 $n_1 + n_2$ を考えたときに，その実現方法ではインクリメントを n_1 回繰り替えさなければならず，n_1 が非常に大きな数であった場合効率が悪い．

そこで，本節では Malbolge で効率よく加算を実現するためのアルゴリズムを検討する．Malbolge の仕様より，ここで扱う値はすべて 10trits で表現できる値とする．

3.1 問題点と解決策

これまで Malbolge で加算が実現されていなかった理由に，利用可能な演算命令が限定されていることが挙げられる．第 2.1 節でも述べたように Malbolge の演算命令は `OPR` と `ROT` の二つのみであり，`OPR` は表 2 の op 演算を桁毎に行う命令で，`ROT` は右ローテート命令である．これらの命令を用いて実現する演算は，すべての桁を平等に扱うもの以外非常に困難であり，どのように加算の桁上げを行うかが問題であった．

その解決策として，二つの二引数三値関数を用いてそれぞれが桁を特別扱いしない加算を用いる．

3.2 加算アルゴリズム

二つの二引数三値関数を用いた加算アルゴリズムを述べる．ここで二つの関数とは，入力の二数に対し桁上げを考えない加算を行う関数 sum（ 表 7）と，入力の二数の加算に関する桁上げのみの計算を行う関数 carry（ 表 8）である．

表 8 carry(X,Y) の表

		x_i		
		0	1	2
y_i	0	0	0	0
	1	0	0	1
	2	0	1	1

入力 10trits で表される二つの値

出力 入力の二数を足し合わせた数

アルゴリズム

1. 入力の二数を変数 X，Y に入れる
2. X，Y に対して関数 sum の演算を行う
3. X，Y に対して関数 carry の演算を行う
4. 2 の演算結果を X にコピーする
5. 3 の演算結果を左に 1trit シフトし，最下位 trit を 0 クリアの後，Y にコピーする
6. 2 から 5 を 10 回繰り返す
7. X を出力する

アルゴリズムの実行例を表 9 に示す．表の値はアルゴリズムの繰り返し開始時の X，Y の値と，その X，Y に対して関数 sum と関数 carry の演算を行った結果の値である．この例は 11355（ =0120120120t）と 1131（ =0001112220t）の加算を行っており，計算結果は 10 ステップ目の sum(X,Y) の値で 12486（ =0122010110t）となり加算として正しい結果となっている．この例では 3 ステップが終了した時点で carry(X,Y) の値が 0 になっており，それ以降のステップは X に 0 を足す操作の繰り返しとなっている．

アルゴリズムの仕様として，桁上げによってオーバーフローが発生した場合，桁上げを計算し左に 1trit ずらした後に最下位 tirt を 0 クリアしているので，オーバーフロー分の最上位 trit が切り捨てられた値が出力となる．

またアルゴリズムが一定時間で停止する理由として，このアルゴリズムで扱う値は 10tirts で表現できる数であるので，どれだけ桁上げが発生したとしても最大 10 回で収まる．そのためアルゴリズムの繰り返しを 10 回にすることにより必ず計算は終了するので，インクリメントのように計算量が増えることはない．例として，最も桁上がりが発生する例の一つである 59048（ =2222222222t）と 2（ =0000000002t）の加算には，10 ステップが必要である．正しい加算結果は 59050（ =10000000001t）であるが，アルゴリズムの仕様に沿って計算結果はオーバーフローを切り捨てた値 1（ =0000000001t）である．

表 9 加算アルゴリズムの実行例 1

ステップ	X	Y
	sum(X,Y)	carry(X,Y)
1	0120120120t	0001112220t
	0121202010t	0000010110t
2	0121202010t	0000101100t
	0121000110t	0000101000t
3	0121000110t	0001010000t
	0122010110t	0000000000t
4	0122010110t	0000000000t
	0122010110t	0000000000t
⋮	⋮	⋮
10	0122010110t	0000000000t
	0122010110t	0000000000t

4 二引数三値関数を実現する疑似命令列の探索手法

第 3 節で提案したアルゴリズムを疑似命令列で実現する上で，アルゴリズムで用いる二つの二引数三値関数 sum，carry を実現する疑似命令列を見つける必要がある．

そこで本節では，単純な三値関数である二引数三値関数を実現する疑似命令列の探索法について述べる．ここで，ある二引数三値関数 f を疑似命令列 I が実現するとは，入力を X，Y とし出力を A レジスタとしたとき，疑似命令列 I を実行した直後の A レジスタの値が $f(X,Y)$ となることである．

第 4.1 節で任意の二引数三値関数が疑似命令列で実現できることを示す．その後に，第 4.2 節で二引数三値関数をより少ない命令数で実現する疑似命令列を探索するアルゴリズムを示し，第 4.3 節でそれを実装したプログラムの評価を行う．

4.1 二引数三値関数の疑似命令列での実現

ここでは，任意の二引数三値関数を疑似命令列で実現できることを示す．

第 2 節で紹介したように，`OPR` 命令に用いられる

表 10　インクリメント用の関数 inc(X,Y)

		x_i		
		0	1	2
y_i	0	0	1	2
	1	1	2	0
	2	1	2	0

op 関数を合成することにより任意の二引数三値関数が実現できることがすでに文献[2]で示されている．また，その調査に用いられた op 関数合成プログラムを利用することで，特定の二引数三値関数を実現する op 関数の合成を知ることができる．例えば，表 10 に対し op 関数合成プログラムを用いると，以下の式で表 10 を実現できることが分かる．

```
op(op(op(X,CON0),op(CON2,op(Y,CON0))),op(CON2,X))
```

このような式をシステマティックに疑似命令列に変換することが出来れば，任意の二引数三値関数を疑似命令列で実現できることがいえる．

この変換で問題となってくるのが，OPR 命令が A レジスタの値と変数 X の値について op 演算を行い，その結果を A レジスタと X に代入していることである．この破壊代入とも呼べる操作により OPR 命令の引数である X は上書きされてしまい，X の値を繰り返し用いることができない．また OPR 命令は op 演算の第一引数に A レジスタをとっているので，上記の式のように直接変数を第一引数に指定することができない．

これらの問題の解決策として，次の手法を提案する．

- op 関数合成プログラムによって得られた式で複数の op 演算の第二引数が同じ変数であった場合あらかじめ必要な数だけその変数の値を別の変数にコピーしておく
- op 演算の第一引数が変数であった場合はその変数の値を A レジスタへロードする操作を行う

変数コピーと A レジスタへのロードは，それぞれ図 8，図 9 の疑似命令列で行うことができる． ここで，変数コピーの疑似命令列で Z は作業用変数であり，X がコピー元の変数，Y がコピー先の変数である．変数 Y，Z の初期値は任意である．A レジスタへのロードの疑似命令列は，変数 X の値が A レジスタへロードされる．

```
ROT CON1
OPR Z
OPR Z
OPR Y
OPR Y
ROT CON2
```

```
OPR X
ROT CON2
OPR X
OPR Z
OPR Y
```

```
ROT CON2
OPR X
ROT CON2
OPR X
```

図 8　変数コピー　　図 9　A レジスタへのロード

```
ROT CON2
OPR Y
ROT CON2
OPR Y
OPR S
ROT CON2
OPR S
ROT CON2
```

```
OPR X
ROT CON2
OPR X
OPR CONO
OPR S
ROT CON2
OPR X
ROT CON2
```

```
OPR S
ROT CON2
OPR S
OPR X
```

図 10　表 10 を実現する疑似命令列

```
ROT CON2
OPR Y
OPR S
ROT CON2
OPR S
```

```
ROT CON2
OPR X
OPR CONO
OPR S
OPR X
```

図 11　表 10 を実現するより短い疑似命令列

提案手法により，op 関数合成プログラムで得られる式をすべてシステマティックに疑似命令列に変換することができる．提案手法を用いて表 10 を実現する式を疑似命令列に変換したものを図 10 に示す．ここで変数 S の初期値は CON0 とする．

以上により，任意の二引数三値関数を疑似命令列で実現できることがいえた．

しかし，この手法で見つけられる疑似命令列は必ずしも最短のものとは限らない．たとえ op 関数合成プログラムにより得られる式が最短であったとしても，特定の二引数三値関数を実現する op 関数の合成は複数存在する．実際，表 10 は図 11 の疑似命令列でも実現できる．ここで変数 S の初期値は CON0 とする．Malbolge のメモリは固定であるため，できるだけメモリを節約することを考えなければならず，二引数三値関数を疑似命令列で実現する場合にはより少ない命令数のものを用いることが望ましい．

4.2 疑似命令列探索のアルゴリズム

任意の二引数三値関数をより短い命令数で実現する疑似命令列を網羅的に探索するアルゴリズムを述べる．

まず次を定義する．

INS(I) {ROT $CON0$, ROT $CON1$, ROT $CON2$, OPR X, OPR Y, OPR A_i | $0 \leq i \leq I$ }
ただし，変数 X，Y は入力で，変数 A_i の初期値は $CON0$，$CON1$，$CON2$ のいずれか

疑似命令 $x \in$INS(I)

疑似命令列 $\alpha \in$(INS(I))*

疑似命令列の長さ $|\alpha|$

命令の連接 $\alpha :: x$

上記の定義を用いると，疑似命令列探索のアルゴリズムは以下のように記述できる．

入力 二引数三値関数の表 table，疑似命令列の長さの上限値 LM，使用する変数の数 (X，Y を除く)N，空の疑似命令列 α

出力 Success(α') か Fail．α' は入力の二引数三値関数を入力の条件下で実現する疑似命令列．

アルゴリズム

Search(table, LM, N, α) =

1. **if** $|\alpha| > LM$ **then** return Fail **else**
2. **if** check(α,table) **then** return Success(α) **else**
3. **foreach** $x \in$ INS(N) **do**
   let $\alpha' = \alpha :: x$
   $Result =$ Search(table, LM, N, α')
   in if $Result ==$ Success(α') **then** return Success(α') **else** continue **end**

補助関数 check(α,table) は受け取った疑似命令列が *table* の関数を実現しているかを調べ，実現してる場合は True を，そうでない場合は False を返す．入力として表 table' と疑似命令列 α' が与えられたときの判定方法は，疑似命令列 α' を実行した後の A レジスタが table'(X,Y) と等しいかどうかである．

4.3 疑似命令列探索プログラム

第 4.2 節で示したアルゴリズムを実装する．第 4.3.1 節では実装の際に行った工夫について述べ，第 4.3.2 節で作成したプログラムの評価実験を行う．

表 11 1 クリアの関数 1clear(X,Y) の表

		x_i		
		0	1	2
y_i	0	1	1	1
	1	1	1	1
	2	1	1	1

4.3.1 実装

前節で提案したアルゴリズムの実装では，プログラム内で X，Y の初期値を 012012012t，000111222t とすることにより，表の全ての場合の判定を一度に行っている．プログラム内の値は 9trits 表現であるが，それは二引数三値関数を表現するのには九つの組み合わせで十分であり，最上位（最下位）の 1trit を効率のため省略しているためである．また，入力の条件をみたす疑似命令列をすべて出力するように実装した．

実装したプログラムは，深さ優先探索に従って疑似命令列の探索を行う．ただし，ヒューリスティクスを導入し，ある程度の枝刈りを行うことで効率化を図っている．枝刈りは次の方法で行った．

1. 最初の命令は ROT に限定する
2. ROT は連続しない

1 に限定するのは，OPR は引数に A レジスタの値をとるため最初の命令としては使えないからである．また 2 は，ROT が連続した場合，後に行った命令だけ実行する場合と実行結果が変わらないためである．

プログラムの動作例として，入力を 1clear（X,Y）の関数表 (表 11) と疑似命令列の長さの上限値 3，使用する変数を 3 つとしたとき，プログラムの出力結果は長さ 1 の疑似命令列が 1 個，長さ 2 のもの 3 個，長さ 3 のものが 32 個となる．長さ 1 のものは $CON1$ を ROT したものである．

4.3.2 評価

作成した疑似命令列探索プログラムの評価のため，いくつかの二引数三値関数をプログラムにあたえ，各関数を実現する最短の疑似命令列の長さとその長さの疑似命令列の種類の数，ならびに探索にかかった時間を表 12 にまとめた．なお，使

表 12　プログラムの評価実験結果

関数	命令長	個数	時間 (sec)
0clear	2	3	0
1clear(表 11)	2	3	0
inc(表 10)	10	4	465
sum	-	-	T.O.
carry	9	6	44

T.O. タイムアウト 5400 秒

用する変数は 3 つとし，関数 0clear と関数 1clear においてそれぞれ ROT 命令一つでの実現は除いている．処理時間の計測には getrusage 関数を用いて，実験は CPU Xeon W5590 (3.33GHz/4core 8thread/L2cache 4*256KB/L3cache 8MB) デュアル，メモリ 48GB の計算サーバ上で行った．

関数 sum に関して，このタイムアウトでは長さ 11 の場合までの探索が可能であり，この条件下では長さ 11 以下で関数 sum を実現する疑似命令列が存在しないことが確かめられた．

5 加算を実現する低級アセンブリプログラムの実装

本節では，第 3 節で提案した加算アルゴリズムを低級アセンブリプログラムとして実装する．まず第 5.1 節で疑似命令列で実現し，それから第 5.2 節で低級アセンブリプログラムに実装する．そして第 5.3 節で，従来のインクリメントを用いた加算との比較評価を行う．

5.1 加算アルゴリズムの疑似命令列での実現

第 3 節で提案した加算アルゴリズムを低級アセンブリ言語で実装するにあたり，直接アルゴリズムから低級アセンブリ言語で実装するのは困難であるので，まず加算アルゴリズムを疑似命令列で実現する．

加算アルゴリズムは疑似命令列で実現する上で，以下のモジュールに分割できる．

1. 入力 X_1 を X_2 にコピーする
2. 入力 Y_1 を Y_2 にコピーする
3. sum(X_1,Y_1) を行う
4. carry(X_2,Y_2) を行う
5. 3 の計算結果を X_1 にコピーする
6. 4 の計算結果を左に 1trit シフトし，最下位 trit を 0 クリアの後，Y_1 にコピーする
7. 1 から 6 までで使用した作業用変数を初期値に復元する

このモジュール群の内，変数コピーモジュールと 0 クリアモジュール，変数復元モジュールは既に飯澤らによって実現されており，左シフトも右ローテートを 9 回行うことで容易に実現できる．本研究で新たに実現するのは，桁上げを考えない加算を行う関数 sum の疑似命令列と桁上げのみを計算する関数 carry の疑似命令列である．どちらの関数も二引数三値関数であり，そのような関数を実現する疑似命令列を発見する手法としては，次の二つがある．

- 既存の疑似命令列や，Malbolge 特有の op 関数の性質からノウハウにより見つける
- 第 4 章で作成した疑似命令列探索プログラムを利用する

関数 sum の疑似命令列は前者の手法により，関数 carry の疑似命令列は後者の手法により発見した．

5.2 加算アルゴリズムの低級アセンブリ言語での実装

第 5.1 節で作成した疑似命令列を参考に，今度は以下の方針に従い，第 3 節の加算アルゴリズムを低級アセンブリ言語で実装した．以降，このプログラムを**加算モジュール**と呼ぶ．

- メモリを節約するためにできるだけ使用するフラグの数を減らす
- 第 5.1 節の疑似命令列のモジュール毎に，低級アセンブリプログラムでもモジュールを記述する．ただし一部に命令数を減らす工夫をしている

プログラム作成には長坂らの開発した Malbolge デバッガ・低級アセンブリデバッガを利用した [5]．

5.3 効率の評価

本研究で実現した加算と，従来のインクリメントを用いて実現した加算の効率の評価を行う．

まず理論的な評価を行う．評価対象は，それぞれの加算の実現に用いられる疑似命令列のステップ数とする．従来の加算を実現するステップ数は，n を

加算に必要なインクリメントの数とすると，全体で $143n - 10$ ステップである．本研究で実現した加算のステップ数は 637 ステップ．よって理論的にはインクリメントが 5 回以上必要な加算において，本研究で実現した加算の方が効率よく計算を行えるという結果が得られた．

上記の評価は定量的に行っているが定性的にも比較を行うことができる．その場合，入力される値は m trits で表現されるものと仮定し，評価を行う．この場合，本研究で実現した加算の計算量は桁数に比例することになるので $O(m)$ となる．一方，従来の加算の計算量は桁数に依存することに加え，最悪の場合インクリメントを $3^m/2$ 回繰り返す必要があるので，$O(3^m)$ となる．よって，定性的な評価においても本研究で実現した加算の方が優れているといえる．

次に実験的な評価を行う．従来のインクリメントを用いた加算の Malbolge プログラムで必要なインクリメントの回数 *inc_size* をパラメータとして，Malbolge インタプリタ上での処理時間を比較した．以降，比較するプログラムをそれぞれ本研究のプログラム，従来のプログラムと呼ぶ．それぞれプログラムは低級アセンブリプログラムから低級アセンブラで Malbolge コードに変換したものである．処理時間の計測には getrusage 関数を用いて，5 回の計測の平均値を結果とした．計測した *inc_size* の値は 1，3，10，30，100，300 で，実験は CPU Athlon 64 X2 4800+ (2.4GHz/L2cache 2*1MB)，メモリ 4GB の計算サーバ上で行った．

その結果が図 12 である．従来のプログラム（ addition by increment）では *inc_size* が増えるに従って処理時間も長くなるのに対し，本研究のプログラム（ addition）はほぼ一定の時間で計算を終えることが出来た．ただし，理論上では *inc_size* が 5 以上であれば本研究のプログラムの方が速く計算できるはずであるが，実験では *inc_size* が 43 以上の場合に本研究のプログラムの方が速いという結果が得られた．

原因としては，疑似命令列と低級アセンブリプログラムの実現手法の違いが挙げられる．疑似命令列がプログラムの本質である演算命令のみで構成されているのに対し，低級アセンブリプログラムでの実装にはプログラムの実行制御命令が含まれる．そのことにより演算命令と演算命令の間での実行制御にかかる時間分の遅延が，理論と実験の結果の差になっているのだと思われる．低級アセンブリプログラムの実装の改善により，この差は縮められるのではないかと考え，低級アセンブリプログラムでの実行ステップ数を減らす方針で同一の疑似命令列を再実装した．改善前と改善後の低級アセンブリプログラムのそれぞれの総ステップ数，使用フラグ数は表 13 のようになった． 改善後のプログラム（ upgrade addition）では，*inc_size* が 10 以上の場合にインクリメントを用いた加算のプログラムより速いという結果が得られ（ 図 12），実装の改善により理論的な値に近づけるということが実験的に確かめられた．

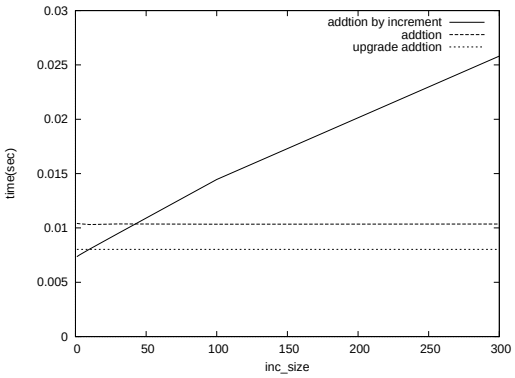

図 12　比較実験結果

表 13　実装の改善

	改善前	改善後
総ステップ数	10809	4324
使用フラグ数	9	16

6　高級アセンブリ言語への加算命令の追加

本節では，第 5 節で作成した加算モジュールを利用し，高級アセンブリ言語への加算命令の追加方針を提案し，その方針に基づき実際に追加を行う．

6.1　高級アセンブリ言語の加算命令

本節で追加を考える高級アセンブリ言語の加算命令は，次の仕様とする．

表 14　基本モジュール一覧

番号	モジュール名
(1)	インクリメント モジュール
(2)	デクリメント 及び条件分岐モジュール
(3)	変数コピーモジュール
(4)	マスクモジュール
(5)	入力モジュール
(6)	出力モジュール
(7)	停止モジュール
(8)	IP に関するモジュール
(9)	REG0,REG1 に関するモジュール

表 15　命令と基本モジュールとの関係

命令	必要な基本モジュール
インクリメント 命令	(1),(3),(8),(9)
デクリメント 命令	(1),(2),(3),(8),(9)
変数コピー命令	(1),(3),(8),(9)
条件分岐命令	(1),(2),(3),(8),(9)
マスク 命令	(1),(3),(4),(8),(9)
ローテート 命令	(1),(3),(8),(9)
入力命令	(1),(3),(5),(8),(9)
出力命令	(1),(3),(6),(8),(9)
停止命令	(7)

- 加算命令 “ADD X,Y”
  $Y := X + Y$ を計算する．ただし，オーバーフローが発生した場合，正しい計算結果の最上位 trit を切り捨てた値を Y に代入する．

6.2　高級アセンブリ言語への加算命令の追加方針

第 2 節でも述べたように，高級アセンブリプログラムは基本モジュール方式を利用して低級アセンブリプログラムで実現されている．これはある仮想マシンを Malbolge 上に構築しており，その仮想マシンはプログラムカウンタ IP とレジスタ REG0,REG1 を持つ．高級アセンブリプログラムは基本モジュール方式のメインルーチンにあたり，高級アセンブリ言語の各命令と同じ機能を持つ低級アセンブリプログラムである基本モジュールをサブルーチンコールのように呼び出している．

高級アセンブリ言語に加算命令を追加するために，高級アセンブリ言語の実行系の解析を行った．基本モジュール群には大きく分けて表 14 のモジュールが存在することが文献[7] で述べられている．解析の結果，各モジュールと高級アセンブリ言語の各命令の関係は表 15 であることが明かとなった．停止命令は特殊であるので，以降は停止命令以外について話を進める．

表 15 から分かるように，全ての命令で (1)，(3)，(8)，(9) のモジュールを使用している．これはこの四つのモジュールが命令の操作だけではなく，高級アセンブリプログラムの実行自体に使用されているためである．(1) と (8) のモジュールが使われているのは，高級アセンブリプログラムのプログラムカウンタである IP を命令や引数を読み込むごとにインクリメントするためである．また (3) と (9) のモジュールが使われているのは，引数をとる命令の場合，まず引数の値をレジスタ REG0,REG1 にコピーし，そして各レジスタに対して命令に対応する操作を行ったあと，必要ならば REG0,REG1 の値を引数の変数や IP にコピーするという仕様になっているためである．

以上より，高級アセンブリ言語に新たに加算命令を追加するために以下の二点を行う必要がある．

- 基本モジュール群に加算モジュールを組み込む．その際，加算モジュールの入力の二数に REG0,REG1 をとるように変更しておく
- 他の命令と同様に IP のインクリメントに (1) と (3) のモジュールを，変数・レジスタ間コピーに (8) と (9) のモジュールを使用するように高級アセンブリプログラム実行系のフラグを再設定する

上記の方針に基づき高級アセンブリプログラムの実行系を拡張し，加算命令を追加することに成功した．

7　おわりに

本論文では，以下のようにして高級アセンブリ言語への加算命令の追加を行った．

- 低級アセンブリ言語での加算モジュールの作成
  - Malbolge 向けの加算アルゴリズムの検討
  - 二引数三値関数を実現する疑似命令列の探索手法の提案
  - 加算アルゴリズムの疑似命令列での実現
  - 加算アルゴリズムの低級アセンブリ言語での実装
- 加算モジュールを用いた加算命令の実現

また，低級アセンブリ言語を用いた加算アルゴリズムの実装や高級アセンブリ言語の実行系の解析，拡張を行ったことにより，低級アセンブリプログラミングを困難にしている主な原因が明らかとなった．

1. 限定的な命令 使用できる命令は基本的に Malbolge と変わらないので非常に限定的である．

2. 特定アドレスへの複数回アクセス 低級アセンブリプログラムでは，同じ名前の変数は一つのアドレスにしか書くことができない．そのためプログラム実行中に一度通り過ぎた変数に再び何らかの操作を行いたい場合，MOV_PC 命令を使用して前に戻る必要がある．ある変数に対する命令は必ずその変数よりも前に記述しなければならないため，ジャンプバックは目的の変数の少し前まで行う．また命令が実行されると自動的に PC がインクリメントされてしまうため，同じ変数に対する連続のアクセスであってもジャンプバックは複数回必要になる．ここでジャンプバックが無条件分岐だと無限ループに陥ってしまうことは容易に想像できる．その解決のために用いられるのがフラグであるが，フラグを用いることによってさらに以下の困難性が生まれる．

3. フラグによる実行制御 低級アセンブリプログラムはフラグによりその実行を制御される．しかし，静的解析が非常に困難であるため，プログラム実行中の各フラグの状態を知るためには動的解析を行う他なく，プログラムの理解やデバックが非常に困難である．特にメモリの節約のために一つのフラグを複数の場所に使用すると，解析はさらに困難となる．大量のフラグを用意し，一つのフラグは一度しか使わないことにすることでこの問題はやや軽減することができるが，Malbolge のメモリは固定であるので必要以上のフラグを使用することは避けるべきである．

4. アドレス値の設定 第 2.6 節でも述べたように，低級アセンブラは入力の低級アセンブリプログラムに合わせてエントリーラベルのアドレス値の指定やデータのメモリ領域の指定をそのソースコードに記述する必要がある．指定した値によっては，出力の Malbolge プログラムが正しく動かない場合もある．この値の決定は，入力のプログラムをひとまず適当な値でアセンブルし，出力された Malbolge プログラムとアセンブル時に得られるある情報を参考にノウハウにより導出した正しいと思われる値を設定し再びアセンブルするという，繰り返しの試行により行われる．

上記の困難性に対する解決策は，現在のところノウハウに依存している部分がほとんどであり，今後の課題として各困難性に対するシステマティックな解決策を考案することが望ましい．

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，本論文で行ったように命令を豊かにしてしまうと解析困難性が低下してしまう恐れがある．特に高級アセンブリ言語で実装を行った場合，現在の手法では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，低級アセンブリプログラミング時にコード難読化を施したり，乱数を用いた構造変換などを行うことで，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．

参 考 文 献

[1] 門田暁人，高田義広，鳥居宏次：プログラム難読化法の実験的評価，情報処理学会研究報告 ソフトウェア工学研究会報告，Vol. 96， No. 32， pp. 33–40， 1996.

[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] Ben Olmstead：”Malbolge: Programming from Hell”, http://www.bouletfermat.com/danny/malbolge/ , 1998.

[4] Andrew Cooke：“malbolge：hello world”, http://www.acooke.org/malbolge.html.

[5] 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.

[6] 飯澤恒，坂部俊樹，酒井正彦，草刈圭一朗，西田直樹：難解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] 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.