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
SS2016-12,KBSE2016-18(2016-07)

難読性の高いMalbolgeコードを生成するコンパイラのための中間言語

河邉　翔平[†]　酒井　正彦[††]　西田　直樹[††]　関　浩之[††]

†,†† 名古屋大学 大学院情報科学研究科　〒466–8603 愛知県名古屋市千種区不老町
E-mail: †kobe@trs.cm.is.nagoya-u.ac.jp, ††{sakai,nishida,seki}@is.nagoya-u.ac.jp

あらまし　Malbolgeは最も難解と言われ，プログラム保護に期待されるプログラミング言語である．近年，高級アセンブラが開発されMalbolgeプログラムの開発が可能になったものの，この手法で生成されたMalbolgeプログラムは難読性に問題がある．その原因は，生成されたプログラムの実行中のメモリ上に，高級アセンブリプログラムの命令系列が出現することにある．本稿では，これを解決する高級アセンブリ言語からMalbolgeを生成するコンパイラの構築を目的とし，それに必要となる中間言語として疑似命令列に制御構造を導入した言語を設計し，それからMalbolgeプログラムの生成系を構築する．ここで，疑似命令は，人手でMalbolgeプログラムを作成する際に利用される制御を持たない命令である．

キーワード　難読化，難解プログラミング言語，Malbolge

An intermediate language for a compiler generating highly obfuscated Malbolge codes

Shohei KOBE[†], Masahiko SAKAI[††], Naoki NISHIDA[††], and Hiroyuki SEKI[††]

† Graduate School of Information Science, Nagoya University
Furo–cho, Chikusa–ku, Nagoya–City, Aichi, 464–8603 Japan
E-mail: †kobe@trs.cm.is.nagoya-u.ac.jp, ††{sakai,nishida,seki}@is.nagoya-u.ac.jp

Abstract　Malbolge is said to be one of the most esoteric programming languages, hence it is promising as a program protection method. Although it became possible to develop Malbolge programs by using the recently developed high-level assembler, a hole was found that enables analysis of Malbolge programs generated by this technique. The reason is that the sequence equivalent to an input high-level assembly program appears on memory at some moment when executing the Malbolge program. For solving this problem, this paper aims at constructing a compiler that generates Malbolge programs from high level assembly languages. We design an intermediate language for this purpose by introducing control structures into pseudo-instruction sequences, and constructed a generator of Malbolge codes from intermediate codes. Here, pseudo-instructions, used for hand construction of Malbolge programs, have no control feature among them.

Key words　Obfuscate, Esoteric Programming Language, Malbolge

1. はじめに

難解プログラミング言語とはプログラミングが困難となるように設計されたプログラミング言語である．このような言語でプログラミング手法を確立することは知的財産権の保護等に役立つと期待されている．Malbolge [1] は難解プログラミング言語の中でも特に難解な言語であるが，プログラムの作成も困難であることが知られている．しかしながら，近年，飯澤らによってMalbolgeの低級アセンブリ言語と高級アセンブリ言語が設計 [2], [3] され，また長坂，安藤，加藤らによってMalbolgeのプログラミング環境が次第に整備されてきている [4], [5], [7]〜[9]．高級アセンブラ [11] はプログラミングに最低限必要な機能である，変数の初期値設定，1の加減算，零判定分岐などを持つため，ようやくプログラミングが可能になってきたと言える．しかしながら，高級アセンブラから出力される低級アセンブリプログラムの構成が，入力のプログラムの命令に1対1で対応するアドレスから高級アセンブリ言語の命令を実現する低級アセンブリプログムを呼び出す構成となっているため，これを変換して得られるMalbolgeプログラムの実行中に高級アセンブリプログラムの命令系列が出現してしまう [6]．これはプログラ

This article is a technical report without peer review, and its polished and/or extended version may be published elsewhere.
Copyright ©2016 by IEICE

表 1 低級アセンブリ言語の命令

命令	動作		制約
ROT *label*	A,[*label*] := rotr([*label*])	D:= *label* + 1	D< *label* ≦D+n
REV ROT	ROT 命令の復元	D:=D+1	
OPR *label*	A,[*label*] := op(A,[*label*])	D:= *label* + 1	D< *label* ≦D+n
REV OPR	OPR 命令の復元	D:=D+1	
SKIP *label*		D:= *label*	D< *label* ≦D+n
JMP *label*		D:= [*label*]	D< *label* ≦D+n
REV JMP	JMP 命令の復元	D:=D+1	
INPUT	A:= *getchr*()	D:=D+2	
REV INPUT	INPUT 命令の復元	D:=D+1	
OUTPUT	*putchr*(A)	D:=D+2	
REV OUTPUT	OUTPUT 命令の復元	D:=D+1	
FLAG	フラグが ON の場合: フラグが OFF の場合: 実行後，FLAG の値を更新する	D:= [D+1] D:=D+2	
NEXT FLAG	FLAG の値を更新する	D:=D+1	

ム保護の観点からは弱点であり，低級アセンブリプログラムへの変換方法に工夫が必要である．これに対する解決法としては高級アセンブラをコンパイラに変更することが考えられるが，低級アセンブリ言語の難解さから直接低級アセンブリプログラムに変換することは難しい．そのため高級アセンブリ言語と低級アセンブリ言語の間の中間言語が必要となる．

本稿ではコンパイラを作成するために，低級アセンブリプログラムの作成手法に着目し，疑似命令を拡張した中間言語を設計する．ここで疑似命令は人手で低級アセンブリプログラムを作成する際に利用される制御を持たない命令である．またこの言語で記述されたプログラムから低級アセンブリプログラムへの変換法を示す．

2. 準 備

低級アセンブリ言語は，Malbolge の命令が実行後に変化するという記述困難性を打破するために開発された言語である．低級アセンブリ言語も Malbolge と同様に，10 桁の 3 進数を 1 ワードに持つ仮想機械上の命令列として定められる．ここで，3 進数の一桁，すなわち 0,1,2 のいずれかの数を *trit* と呼び，3 進数の値を 1111111111t のように最後に t をつけて表す．Malbolge の仮想機械は，レジスタとしてアキュムレータ A の他にデータレジスタ D とプログラムカウンタ PC を持つが，低級アセンブリ言語においてはレジスタ PC は陽に用いることはできず，レジスタ D がプログラムカウンタの役割を果たすことに注意を要する．以下では，本稿に関連する低級アセンブラの重要な点について説明する．

定数および命令の前に"ラベル名:"を置くことでラベルが付加される．低級アセンブリ言語においては，ラベル付きの定数が変数定義の役割を果たす．各命令は表 1 のように定義される．ここで n は低級アセンブラの実装で定められる特定の自然数であり，[*label*] は *label* でラベル付されたデータを表す．また，演算 rotr は右循環シフト，演算 op は表 2 で定義される trit 毎の演算であり，crz と書かれることがある．さらに A,[*label*] := op(A,[*label*]) のように，:= の左辺の両方共に右辺の値が代入されることを表す．特に，演算命令 ROT，OPR の実行制御は独特であり，注意が必要である．(4. 節で述べる．)

表 2 trit 毎の演算 op(x,y)

y\\x	0	1	2
0	1	0	0
1	1	0	2
2	2	2	1

表 3 固定ユニット

アドレス		データ (Malbolge の命令)	
59046	(2222222220t)	74	(SNOP)
59047	(2222222221t)	74	(SNOP)
59048	(2222222222t)	74	(SNOP)
0	(0000000000t)	98	(JMP)

フラグは"FLAG [初期状態/周期] ラベル名"の形式で宣言する．周期 p は 2, 4, 5, 6, 9 のいずれかであり，本稿では周期 2 のフラグのみを扱う．フラグの値は 0 から $p-1$ の自然数であり，0 のとき ON，それ以外のとき OFF に対応する．その値の更新は，p を法とした 1 の加算によって行われる．

ユニットは自由な 1 ワード長のデータ列であり，その配置アドレスは 94 の剰余系で指定できる．低級アセンブラは表 3 のユニットを常に生成する．このユニットは 5.5 節で説明する末尾値分岐に利用される．

3. 制御付き疑似命令列

本節では低級アセンブリプログラムを生成するための中間言語として制御付き疑似命令列を設計し，これの構文と意味について述べる．中間言語は低級アセンブリプログラムの生成の際に用いるため，まず，これまでに低級アセンブリプログラムを作成する際に用いてきた手法を説明し，それに準じて中間言語を設計する．

低級アセンブリ言語の演算命令は配置可能なアドレスが限られること，実行後に強制的なジャンプが行われることから逐次実行でさえ記述するのが難しい．そこで現在までに知られている方法では，まず，プログラムで実行したい逐次処理をいくつかの疑似命令列で表しそれらの実行の組み合わせ方法を設計していた．例えば 1 を加算するコードは長さ 13 の疑似命令列を 10 回繰り返すことで実現できる [3]．この設計から低級アセンブリプログラムを得るには，あとで述べるように注意深いコーディングが必要となる．

まず，疑似命令について説明する．疑似命令における値は1 ワード長である．疑似命令列では，レジスタは 1 ワードの値を格納するアキュムレータ A のみであるが，その他に好きなだけ変数を用いることができる．疑似命令列は，図 1 に示すように，順に1 度づつ実行する疑似命令を並べたものである．ここで ROT X は，変数 X を循環的に右シフトする疑似命令であり，OPR X は Malbolge 固有の trit 毎の演算 op を表す．疑似命令の構文と意味は，表 4 のようにまとめられる．

以上のことを考慮して，疑似命令列の実行を制御する制御命令を加えた言語(制御付き疑似命令列) を設計する．

```
ROT X
OPR X
OPR' X
```

図 1 疑似命令列の例

表 4 疑似命令

命令	意味
ROT X	右ローテート命令. A,X := rotr(X)
OPR X	演算命令. A,X := op(A,X)
INPUT	入力. A := getchar()
OUTPUT	出力. putchar(A)

```
IF Flag
 ROT X
ELSE
 OPR X
 OUTPUT
IFEND
OPR X
ROT X
```

図 2 制御付き疑似命令列の例

3.1 追加する制御命令

疑似命令に追加する命令に求められることとして，追加によって記述したいプログラムを表現できること，それらの命令が低級アセンブリプログラムに変換できる程度の複雑さであることが重要である．これらを総合的に考えた上で，疑似命令に追加する制御命令を設計した．ここで，疑似命令列では1ワード長の数を値として持つ変数だけが許されているのに対して，制御付き疑似命令列ではブール値を持つブール変数を利用することができる．ブール変数は，低級アセンブリ言語の用語を用いてフラグと呼ぶ．

導入した制御命令(表5)を簡単に説明する．

(1) 2値制御:

フラグの定義，フラグの値を設定する命令と，フラグの値に基づく処理の制御を行う．

(2) 繰り返し:

命令列を，与えられた定数の回数(あるいは無限回)繰り返し実行する．ただし，途中から BREAK で抜けることを許す．

(3) 末尾値制御:

変数の最下位 trit の値によって処理の制御を行う．ただし，正常な動作のためには変数 X の最下位以外のすべての trit の値が 2 である必要がある．そうでない場合には，予測不能な動作をする．

制御付き疑似命令列の例を図2に示す．このプログラムは，フラグ Flag が ON のとき ROT X を実行し，OFF のとき OPR X と OUTPUT を順に実行し，その後に OPR X と ROT X を順に実行する．

制御付き疑似命令列の低級アセンブリ言語への変換方法については次節で述べるが，以下では制御命令をこれ以上複雑に設計しなかった理由を簡単に述べる．まず，(3) の末尾値制御の導入には，低級アセンブリプログラムにおいて値による制御が，最下位 trit の値で分岐するしか方法がないことが背景にある [10]．このように低級アセンブリプログラムにおいて変数の値に基づく実行制御が困難であることから，(1) の繰り返し構造の設計においても，変数の値の回数だけ繰り返すようにはしなかった．

表 5 疑似命令に追加する命令

	命令	意味
2値制御	FLAG flag v	初期値 $v \in \{\mathrm{ON}, \mathrm{OFF}\}$ のフラグ flag を定義する
	SET flag	flag を ON にする
	RESET flag	flag を OFF にする
	IF flag 　命令列 1 ELSE 　命令列 2 IFEND	 flag が ON のとき命令列 1 を実行する flag が OFF のとき命令列 2 を実行する
繰り返し	REPEAT *num* 　命令列 REPEATEND	*num* 回命令列を繰り返す ただし，*num* が FOREVER のときは無限に繰り返す
	BREAK	繰り返しから抜ける
末尾値制御	SWITCH X CASE0 　命令列 1 CASE1 　命令列 2 CASE2 　命令列 3 SWITCHEND	前提: X の最下位以外の全ての trit が 2 X の最下位 trit が 0 のとき命令列 1 を実行 X の最下位 trit が 1 のとき命令列 2 を実行 X の最下位 trit が 2 のとき命令列 3 を実行

次に，このように設計した制御付き疑似命令列のプログラムの表現能力を議論する．高級アセンブリ言語はチューリング完全である [4] ことが示されており，また，制御付き疑似命令列をそのコンパイラの中間言語として設計していることから，高級アセンブリ言語の各命令の実現について考える．

(1) インクリメント命令 INC X は変数 X に 1 を加えるものであり，ある疑似命令列を定数回繰り返し実行することで実現できる [3]．

(2) デクリメント命令 DEC X は変数 X から 1 を減ずるものであり，文献 [4] の手法に基づいて，疑似命令列と末尾値制御を利用して実現できる．

(3) 変数のコピー命令 MV X Y は変数 X の値を変数 Y にコピーする命令であり，疑似命令列で実現される [3]．

(4) ゼロ分岐命令 BRANCH X,Label は，X の値が 0 の場合に Label の場所に分岐する．これは，疑似命令列と末尾値制御を利用して実現できる．

以上のことから疑似命令と表5からなる制御つき疑似命令列により，高級アセンブリプログラムを表現できる．

4. 制御のない疑似命令列からの低級アセンブリプログラム生成

低級アセンブリ言語には，変数に対する命令は変数の実体が置かれている場所の情報に配置するという強い制限がある．例えば，1111111111t を初期値に持つ変数 X とそれに対する 2 種類の命令は典型的には図3に示すように置かれる．ここで，ラ

```
LrotX:  ROT X
LoprX:  OPR X
X:      1111111111t
Label:
```

図 3 低級アセンブラにおける変数と命令の関係

```
LoprX:  OPR X
X:      1111111111t
Label:  REV OPR
        IF Flag
        BRANCH LoprX
```

図 4 ROT X を 2 回繰り返す低級アセンブリプログラム

ベル LrotX の位置から実行すると，ROT X 命令の実行後には変数 X の次の位置（図 3 においては Label の位置）の命令に実行が移る．同様にラベル LoprX の位置から実行すると，OPR X 命令の実行後には Label の位置に実行が移る．このため，ラベル Label 以下の部分には，次に実行すべき命令へのジャンプが必要であり，基本的にはフラグを用いて「何度目に呼ばれたか」という情報からジャンプ先を切り替える．このことを OPR X を 2 回繰り返し実行する図 4 の低級アセンブリプログラムを用いて説明する．このプログラムの実行前に Flag の値が ON であったと仮定し，ラベル LoprX から実行を開始すると以下のように実行が進む．ここで，IF はフラグが OFF のときに次の BRANCH 命令をスキップするが，実行の際にはスキップするかどうかにかかわらず該当するフラグの値を反転させる．

1. OPR X を実行
2. REV OPR を実行
3. Flag の値が ON なので，IF Flag により次の命令に移る．その際に Flag の値が反転し OFF になる．
4. BRANCH LoprX により LoprX にジャンプ
5. 1. から 2. までを実行
6. Flag の値が OFF なので，IF Flag により次の命令をスキップする．その際に Flag の値が反転し ON になる．

また，REV OPR 命令は，OPR 命令の実行後に必ず実行する必要のある復元命令である．

以上の手法では，以下の2 つの問題が生ずる．一つ目は，変数を扱う命令は OPR と ROT の 2 種類あるため，どちらを実行したかによって実行すべき復元命令が異なる．もうひとつは，該当の処理が終えた後に次に実行すべき疑似命令に対応する場所にジャンプすることである．最初の問題に対して，これまでは場当たり的に低級アセンブリコードを作成されてきたので，システム的な生成法の検討が必要である．二つ目については，安藤らの作成した低級アセンブリプログラムで利用されている手法が，コード長を気にしなければ利用可能である．

変数 X に対してこれらの問題を解決した低級アセンブリプログラムのひな形を図 5 に示す．以下ではこれについて説明する．まず，最初の問題である復元命令の切り替えは図 5(a) のように実現できる．ここで，FlagRevX は復元命令の切り替えに用いる初期値が OFF のフラグである．また，NEXT 命令はフラグ値を反転させる命令である．動作の鍵となる点は，一連の IF FlagRevX の実行を開始する時点での，フラグ FlagRevX が，ROT X を実行した場合には ON，OPR X を実行した場合には OFF に設定されている点である．

```
LrotX:  NEXT FlagRevX
        ROT X
LoprX:  OPR X
X:      1111111111t
        IF FlagRevX
        BRANCH LrrotX
        REV OPR
        NEXT FlagRevX
        SKIP LjX
LrrotX: REV ROT
```

(a)（前半部）

```
LjX:  IF FlagX1
      BRANCH J1
      NEXT FlagX1
      IF FlagX2
      BRANCH J2
      NEXT FlagX2
        :
      IF FlagX(n − 1)
      BRANCH J(n − 1)
      NEXT FlagX(n − 1)
      JMP Jn
```

(b)（後半部）

図 5 変数 X の実行モジュール

2 番目の問題点である，次に実行すべき疑似命令に対応するコードへのジャンプに用いられる手法を紹介する．（図 5(b) 参照[注1]）この方法では，変数 X に対する命令の i 回目の実行の後にラベル Ji に実行を移すために，それぞれ初期値を ON とするフラグ FlagXi を使用している．i 回目の実行後には，FlagX1 から FlagXi までのフラグの値が OFF で，FlagX$(i+1)$ 以降のフラグの値が ON になるように設計していることが鍵となる点である．これにより，i 回目の実行後にラベル Ji にジャンプでき，逐次的な疑似命令列を低級アセンブリプログラムに変換可能である．

例えば，図 1 の疑似命令列の実装には，図 5 の変数の実行モジュールにおいて，n を 3，J1 を LoprX，J2 を LoprX として実現できる．このとき，LrotX から実行開始すると，三つの疑似命令の実行後に J3 に実行を移すことになる．

5. 低級アセンブリプログラムへの変換法

本節では設計した制御付き疑似命令列から低級アセンブリプログラムへの変換法について述べる．

これまで述べてきたように，低級アセンブリプログラムは多数の分岐を組み合わせて作成せざるを得ず，ソースプログラムをモジュールに分解したうえで，それぞれから生成した低級アセンブリプログラムを組み合わせることは大変困難である．そのため，低級アセンブリプログラムの生成には制御付き疑似命令列の大域的な情報を抽出し，その情報を用いることで低級アセンブリプログラムを構成することが必要である．

以下では，この構成法の概要を述べる．最初に，前節で述べた手法を応用することで，入力された制御付き疑似命令列に出現する変数の種類ごとに低級アセンブリプログラムの断片を生成する．また，それぞれの制御命令に対応した低級アセンブリプログラムの断片，ならびに，それらに必要となるフラグの定義を与える．これらを並べたものが変換の出力となる．すなわち，図 6 に示すように，変数が関連する命令である ROT と OPR に対しては，各変数ごとにモジュールを生成し，その他の命令については命令ごとにモジュールを生成する．

以下では，各モジュールの生成法を述べる．その際に各モジュールのエントリポイントのラベルの情報が必要となるため，

（注1）：最終行の JMP 命令の実行後には REV JMP の実行が必要となるため，実際にはフラグと IF を利用した分岐を用いる．

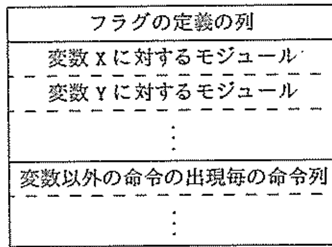

図 6 出力される低級アセンブリプログラムの構成

```
LLℓ:  INPUT
      DUP
      REV INPUT
```

図 7 INPUT から生成される低級アセンブリ命令系列

```
Label:  IF Flag
        BRANCH Label
```

図 8 フラグ *Flag* の値を ON に初期化する低級アセンブリ命令系列

行番号 ℓ の命令のエントリポイントを以下のように定め，それを Entry(ℓ) で参照する．

- 変数の実行モジュールのエントリポイントは ROT と OPR のどちらを実行するかによって二つ必要である．変数 X の名前を利用して，それぞれ，LrotX と LoprX とする．
- 変数以外のモジュールのエントリポイントは，制御付き疑似命令列における行番号 k を利用して，LLk とする．

5.1 変数の実行モジュール生成

変数 X の実行モジュールは，前節で述べた手法を応用して作成する．まず，ROT X と OPR X 命令が n 箇所現れているとし，それら n 個のうちの i 番目の命令の行番号を a_i で表すことにする．このとき，図 5 のモジュールにおける各ジャンプ先 Ji ($1 \leqq i \leqq n$) として，それぞれ，i 番目の OPR/ROT 疑似命令の行番号の次の行番号 a_i+1 にある疑似命令のエントリポイントのラベル Entry(a_i+1) を置くことで変数の実行モジュールが構成できる．

5.2 INPUT の変換

それぞれの INPUT の出現毎に以下の命令系列を生成する．

LLℓ は該当の INPUT の行番号 ℓ から生成されるエントリポイントである．また，低級アセンブリ言語においては INPUT 命令の次には必ず DUP を置くことになっている．

なお，OUTPUT についても同様な系列を生成する．

5.3 2 値制御の変換

フラグの定義"FLAG *Fname* v" は対応する低級アセンブリ言語の周期 2 の命令を用いて，v が ON の場合には FLAG [0/2] *Fname* で，b が OFF の場合には FLAG [1/2] *Fname* に変換できる．

SET *Flag* は図 8 に示すコードで，RESET *Flag* は，これとフラグの値を反転させる命令 NEXT *Flag* の組み合わせによって，それぞれ実現できる．

しかしながら，2 値制御構造の実現には，フラグの値による分岐処理以上の処理が必要で，具体的には，変数の実行モジュールの動作を乱さないような処理が必要となる．以下ではこれを説明する．

```
LLℓif:  IF Flag
        BRANCH Lif1
        NEXT Flag
        ( フラグの操作)
        JMP Entry(ℓelse + 1)
Lif1    NEXT Flag
        JMP Entry(ℓif + 1)
```

(a) IF の実現

```
LLℓelse: ( フラグの操作)
         JMP Entry(ℓifend + 1)
```

(b) ELSE の実現

```
LLℓifend:  JMP Entry(ℓifend + 1)
```

(c) IFEND の実現

図 9 IF, ELSE, IFEND の実行モジュール

```
              IF Frep1
              BRANCH Lrepeatend1
              IF Frep2
              BRANCH Lrepeatend1
                 ⋮
              IF Frepm
              BRANCH Lrepeatend1
              JMP Entry(ℓrepeatend + 1)
Lrepeatend1:
              ( 後半部)
              JMP Entry(ℓrepeat + 1)
```

図 10 REPEATEND から生成されるコード

図 2 の疑似命令列を例として，Flag が ON で ROT X が実行された後の状況を考えよう．このとき，変数 X の実行モジュールは 1 回呼ばれているため，その実行後のジャンプを制御するフラグ FlagX1 が OFF で FlagX2 と FlagX3 は ON になっている．このため IFEND の直後の OPR X 命令の実行後に OUTPUT を実行してしまう．これに対処するために，IF と ELSE に挟まれた命令系列の実行後に，ELSE と IFEND に挟まれた命令系列をあたかも実行したかのようにフラグの変更が必要となる．図 2 の疑似命令列の場合には，FlagX2 を ON から OFF に反転させる必要が生ずる．同様に，ELSE と IFEND に挟まれた命令系列の実行前にも FlagX1 を ON から OFF に反転させる必要がある．これらのフラグの扱いに注意することで，2 値制御構造を低級アセンブリプログラムとして実現できる．

対応する IF *Flag*, ELSE, IFEND が，それぞれ ℓ_{if} 行目，ℓ_{else} 行目，ℓ_{ifend} 行目に置かれているとする．このとき，それぞれの命令系列は図 9 のように実現される．

5.4 繰り返し制御の変換

5.4.1 REPEATEND 命令の変換

説明の都合上，REPEATEND の変換から説明する．ここでは，(a) これから実行する繰り返し回数に従った制御，ならびに，(b) 各変数毎の実行モジュール内で用いられる，実行後のジャンプを制御するフラグ FlagXi の初期化を行う．

まず，(a) の繰り返し回数の管理については，以下のように行う．該当の REPEATEND 命令とそれに対する REPEAT *num* 命令が，それぞれ $\ell_{repeatend}$ 行目と ℓ_{repeat} 行目に置かれているとする．このとき，繰り返しの対象となる疑似命令列は $\ell_{repeat}+1$ 行目から $\ell_{repeatend}-1$ 行目に置かれていることになる．実行中にこのあと必要な繰り返しの数を j 回($j \leqq num$)とし，$j-1$ の 2 進数の補数としてフラグの列 Frep$m$, Frep$(m-1), \ldots,$ Frep1 で表す．例えば，$m=3$ においてあと 6 回実行が残っている場合には，$6-1$ の 2 進数表現 101 の補数 010 で表す．このとき，繰り返しを行うコードを図 10 に示す．これにより残りの実行

```
Lbranch: 22222222220t
         JMP L2
         JMP L1
         JMP L0
```

図 11 固定ユニットの利用法

回数が減る，すなわち，その補数表現が 1 ずつ増加し，全てが 1 になったときすなわち OFF の時に繰り返しから抜ける．

次に (b) について，すなわち，先に述べた各変数 X 毎の実行モジュール内で用いられる，実行後のジャンプを制御するフラグ FlagXi の初期化について考える．OPR X や ROT X 命令が全体で n 回出現するとし，それらのうちで該当の REPEAT と REPEATEND 間には j 番目から k 番目が出現するとする．このとき，j 番目の命令の実行前には FlagX1 から FlagX$(j-1)$ までのフラグが OFF であり，それ以降のフラグが ON になっている．一方，k 番目の命令の実行後には FlagX1 から FlagXk までのフラグが OFF であり，それ以降のフラグが ON になっている．したがって，繰り返しの際には，FlagXj から FlagXk までのフラグを ON に設定すればよく，フラグの値を ON に設定するコード（ 図 8）を用いて，各フラグ Frep$i$ を初期設定することで実現できる．なお，(b) に関するコードは図 10 のコード中の（ 後半部）に配置される．

5.4.2 REPEAT 命令の変換

ここでは，REPEAT num に対して，各フラグ FlagXi の値を，$num-1$ の2 進数の補数になるように設定するだけでよい．これは，図 8 と，フラグの値を反転させる命令 NEXT *Flag* の組み合わせによって，容易に実現できる．

5.4.3 BREAK 命令の変換

BREAK 命令は REPEAT-REPEATEND 内のループから外に脱出する命令である．所期の動作である，該当の REPEATEND 命令の次行の命令のエントリポイントにジャンプすることのほかに，5.4.1 節の (b) と同様，実行後のジャンプを制御するフラグ FlagXi の初期化が必要である．BREAK が実行されたときに，該当の REPEAT と REPEATEND 間には j 番目から k 番目が出現するときは，それ以降の OPR X や ROT X 命令の実行後の制御を正常に行うためには値が OFF である必要のある FlagXj から FlagXk までのフラグの値が ON になっている可能性がある．したがって，次行の命令のエントリポイントにジャンプする前に，これらの値を OFF に変更する必要がある．これらは，REPEAT, REPEATEND におけるフラグの処理と同様，容易に設計可能である．

5.5 末尾値制御の変換

低級アセンブリが自動的に生成する固定ユニットを利用して実現する．この固定ユニットの利用法は大変巧妙であり，図 11 のような低級アセンブリプログラムにおいて，変数をあたかも命令であるかのように実行することで行う．該当の変数 Lbranch の値は，222222222it（ $0 \leqq i \leqq 2$) のいずれかである必要がある．この変数を実行すると，i の値に応じて Li にジャンプする．

末尾値制御の変換は，この機能を利用して実現できる．このとき，変数の実行モジュールの設計を変更する必要があるが説明が複雑となるため，詳細は割愛する．

6. まとめ

本稿では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への変換に利用できる中間言語として制御付き疑似命令を設計し，それから低級アセンブリプログラムを生成する手法を提案した．

提案した変換法では制御命令を変換した際に逐次実行を制御するフラグの操作を行うコードを生成する．このコードのコード長は制御命令の制御空間に比例するため，制御空間が大きい場合や，複数の制御命令で重複している場合には，変換されたプログラムのコード長が非常に大きくなる可能性がある．Malbolge はメモリ空間に制限がある言語であるため変換したプログラムのコード長は短いことが望ましい．したがって，制御空間に依存しない短いコード長でフラグの操作を行えるように最適化することは今後の課題である．

制御付き疑似命令列は高級アセンブリ言語の実現のために提案したが，制御付き疑似命令列の表現能力は高く，より高級な言語からの変換も可能であると考えている．これらの検討・実現は今後の課題である．

謝辞 本研究は一部，科研費 #26540027 の助成を受けている．

文 献

[1] Ben Olmstead, Malbolge, 1988.
[2] 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，電子情報通信学会技術報告，Vol.105，No.129，pp.25–30, 2005.
[3] 飯澤恒，難読言語 Malbolge に基づくプログラム難読化に関する研究，修士学位論文，名古屋大学情報科学研究科，2006.
[4] 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，電子情報通信学会技術報告，Vol.110，No.227，pp.55–60, 2010.
[5] 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，2011.
[6] 曽優也，難読言語 Malbolge の逆コンパイル困難性に関する研究，学士学位論文，高知工科大学情報システム工学科，2011.
[7] 安藤聡，酒井正彦，坂部俊樹，草刈圭一朗，西田直樹，三値関数を実現する Malbolge 命令列の発見のための SAT エンコーディング，電子情報通信学会技術報告，Vol.112, No.275, pp.7–12, 2012.
[8] 加藤起駒，酒井正彦，坂部俊樹，草刈圭一朗，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，電子情報通信学会技術報告，Vol.113, No.159, pp.73–78, 2013.
[9] 加藤起駒，酒井正彦，坂部俊樹，西田直樹，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，電子情報通信学会技術報告，Vol.114, No.127, pp.99–104, 2014.
[10] 加藤 起駒，難解言語 Malbolge の低級アセンブリ言語における条件分岐の記述と高級アセンブリプログラミング環境の構築，修士学位論文，名古屋大学情報科学研究科，2015.
[11] Malbolge, Malbolge20, http://www.trs.cm.is.nagoya-u.ac.jp/Malbolge/.