# MACRO PEG and Hamiltonian Cycle Problem

古賀 智裕*

2016/07/26†

**概 要**

有向ハミルトン閉路問題を単一の $G \in$ first order MACRO PEG(以下, foMPEG と書く) で解くことに成功したので, ここに書き留めておく. 副産物として, P≠NP という仮定のもとで, foMPEG は良い最適化を持たない (入力文字列長の多項式時間内で foMPEG の動作が終わるような最適化が存在しない) ことが示せた. なお, この文書で使う $G \in$ foMPEG は, 引数に指定される parsing expression の中に非終端記号が全く出現しないので, いわゆる first order MACRO PEG よりも更に限定された形のものになっている. このことから, そのように限定された foMPEG ですら, P≠NP という仮定のもとで, 良い最適化を持たないことになる.

## 1 イントロダクション

MACRO PEG [3] とは, PEG [2] にマクロのような機能を持たせて PEG を拡張したものである. この文書では, 以下の 2 つの定理を証明し, そのあとで, MACRO PEG の最適化について考察する:

**定理 1.1** 有向ハミルトン閉路問題を単一の $G \in$ foMPEG で解くことが出来る.

**定理 1.2** もし P≠NP ならば, ある $G \in$ foMPEG が存在して, foMPEG 自体をどのように最適化しても, $G$ の評価にかかるステップ数は入力文字列長の多項式時間内では終わらない.

**補足** この文書の MACRO PEG では, 引数の評価戦略は call-by-name のつもりである. また, NOT predicate を ! とは書かず, ( ] と書いている. すなわち, $(e]$ という記述は $!e$ のつもりである. また, 随所に出てくる . (ドット) は, PEG のときと同じく,

無条件で 1 文字消費して成功する (ただし, 入力文字列が空文字のときだけは失敗する)

という parsing expression である. また, 引数の書き方は, $A(x_1, x_2, \cdots, x_n)$ ではなく $A[x_1][x_2]\cdots[x_n]$ という書き方を採用している. 私が個人的に作った MACRO PEG ではこのような書き方を読み込むようにしているので, その書き方に合わせているだけである. ちなみに, 私の MACRO PEG では, $n$ 個の引数を取る非終端記号 $A$ に対して, $A[x_1][x_2]\cdots[x_n]$ の他に $A[x_1][x_2]\cdots[x_i]$ $(i < n)$ という表現も認識するようになっており, このような表現を別の非終端記号の引数に渡すことも可能である. ただし, 以下の MACRO PEG では, このような機能は使っていない. すなわち, $n$ 個の引数を取る非終端記号 $A$ に対しては, 常に $A[x_1][x_2]\cdots[x_n]$ という使い方しかしない.

*こが としひろ, toshihiro1123omega_f_ma_mgkvv_sigma_w7.dion.ne.jp _sigma_ → @ omega →
†最終更新日 2018/12/14

## 2　定理 1.1 の証明

**定理 1.1 の証明**　[3] の ”grammar: modifiers” では, リストのような機能が MACRO PEG の上で再現されている. 以下では, その手法を使うことにする. 以下の foMPEG を $G = (V_N, V_T, R, e_S)$ と置く. ただし, $V_T = \{\text{”1”}, \text{” : ”}\}$ とする.

```
true = (1] / ((1]];
false = (1] 1;
one = 1 (one / true);

chk1 = ((one :  chk10]];
chk10 = (.]  / one :  one :  chk10;
chk2 = (( chk20[true] (.]  ]];
chk20[x] = 1 chk20[x 1] / :  chk21[x 1];
chk21[x] = (x] ( (.]  / one :  chk21[x] );

eq[v][w] = ((v]] ((w]] eq[v 1][w 1] / ( ((v]] ((w]] ] (v] (w];
consume = .  consume / true;

Hamilton = H[true] consume;
H[n] = ((n 1]] H[n 1] / (n 1] H0[n :][1];
H0[n][v] = ((v]] (H1[false][true][n][v][v] / H0[n][v 1]);

H1[track][tlen][n][v][w]
= ((tlen 1]] H2[track][tlen][n][v][w][true][true][true][true] /
   (tlen 1] eq[v][w];

H2[track][tlen][n][v][w][p][px][py][y]
= (n p (.]]  ( ((n p px 1]] H2[track][tlen][n][v][w][p][px .][py][y] /
  (n p px 1] H3[track][tlen][n][v][w][p][px .][py][y] );

H3[track][tlen][n][v][w][p][px][py][y]
= ((n p px py 1]] H3[track][tlen][n][v][w][p][px][py .][y 1] /
   (n p px py 1] H4[track][tlen][n][v][w][p][px][py .][y];

H4[track][tlen][n][v][w][p][px][py][y]
= (( n p w :  (track] ]] H1[track / y :][tlen 1][n][v][y] /
  H2[track][tlen][n][v][w][p px py][true][true][true];

S = chk1 chk2 Hamilton;
```

$e_S$ := S

上記のコードで使われている `track` という引数が, [3] の ”grammar: modifiers” で再現されているリストのような機能とほぼ同じものになっている.

さて, 上記のコードで有向ハミルトン閉路問題が解けることを説明する. まず, $n \geq 3$ 個の頂点から成る有向グラフ $G_1$ を任意に取る. $G_1$ から, 以下のアルゴリズムによって $z \in \{\ 1\ ,\ :\ \}^*$ を生成する: まず, 頂点には適当に番号を振って $v_1, v_2, \cdots, v_n$ と名づける. このとき, 対応する文字列 $z$ は

$$z = 111\cdots11:\quad (1\text{ が }n\text{ 個並んでいて, 末尾に”}:\text{”がある})$$

としておく. 次に, $v_i$ から $v_j$ に向かって矢印が引かれているとき,

$$111\cdots11:111\cdots11:\quad (\text{左には }1\text{ が }i\text{ 個あり, 右には }1\text{ が }j\text{ 個ある}) \tag{1}$$

という文字列を $z$ の末尾に追加する. グラフ $G_1$ の全ての矢印 $v_i \to v_j$ に対して, 対応する (1) のような文字列を $z$ の末尾に順次追加していく. 追加する順番は何でもよい. また, 同じ $v_i \to v_j$ を重複して $z$ の末尾に追加する必要はない (追加しても全く問題ないが, 追加しない方が $G \in$ foMPEG での計算が早く終わる). こうして得られる $z$ を入力文字列として, $e_S = \mathtt{S}$ に適用すると,

- その結果が成功なら, グラフ $G_1$はハミルトン閉路を含む.
- その結果が失敗なら, グラフ $G_1$はハミルトン閉路を含まない.

という状況になる. このことから, 上記の $G \in$ foMPEG によって有向ハミルトン閉路問題が解けることが分かる. 例として,

$$\begin{aligned} &G_1 = (V, E),\quad V = \{v_1, v_2, v_3, v_4\}, \\ &E = \{v_1 \to v_2,\ v_2 \to v_3,\ v_3 \to v_1,\ v_3 \to v_4,\ v_4 \to v_1\} \end{aligned}$$

という有向グラフを考える. これに対応する $z \in \{\ 1\ ,\ :\ \}^*$ は

$$z = 1111:\quad 1:11:\quad 11:111:\quad 111:1:\quad 111:1111:\quad 1111:1:$$

というものになる (見易さのため, 文字列中にスペースを設置してあるが, 本来はスペースなしで書く). あるいは, 次のようにしてもよい:

$$z = 1111:\quad 1111:1:\quad 111:1:\quad 1:11:\quad 11:111:\quad 111:1111:$$

どちらにせよ, このような $z$ を上記の $e_S = \mathtt{S}$ に適用すると, その結果は「成功」となる. よって, この $G_1$ はハミルトン閉路を含むことになるが, 実際に $v_1 \to v_2 \to v_3 \to v_4 \to v_1$ というハミルトン閉路が存在している. 今度は

$$\begin{aligned} &G_1 = (V, E),\quad V = \{v_1, v_2, v_3, v_4\}, \\ &E = \{v_1 \to v_2,\ v_2 \to v_3,\ v_3 \to v_1,\ v_3 \to v_4\} \end{aligned}$$

という有向グラフを考える. これに対応する $z \in \{\ 1\ ,\ :\ \}^*$ は

$$z = 1111:\quad 1:11:\quad 11:111:\quad 111:1:\quad 111:1111:$$

というものになる. あるいは, 次のようにしてもよい:

$$z = 1111:\quad 111:1:\quad 1:11:\quad 11:111:\quad 111:1111:$$

どちらにせよ, このような $z$ を上記の $e_S = \mathtt{S}$ に適用すると, その結果は「失敗」となる. よって, この $G_1$ はハミルトン閉路を含まないことになるが, 実際にハミルトン閉路は存在しない.

次に, 上記の $G \in$ foMPEG の実装の中身を解説することにする. $z$ を $e_S = \mathtt{S}$ に適用すると, `chk1` `chk2` `Hamilton` が試される. まず, `chk1` では, 入力文字列 $z$ が

$$1^+ : \ (1^+ : 1^+ : \ )^*$$

という形になっているかをチェックしており, もしそうでないなら失敗となる. よって, `chk1` を通過できたら,

$$z = 1^n : 1^{a_1} : 1^{b_1} : 1^{a_2} : 1^{b_2} : \cdots 1^{a_m} : 1^{b_m} : \quad (m \geq 0) \tag{2}$$

という形になる[1]. 次の `chk2` では, (2) の表示において

$$\forall i \in [1, m] \ [\ a_i \leq n, \ b_i \leq n \ ]$$

が成り立つかどうかをチェックしており, もし成り立たないなら, そこで失敗となる. これらの `chk1`, `chk2` が通過できたときのみ, `Hamilton` が実行される. `Hamilton` では, まず `H[n]` が実行される. この関数によって, 引数 `n` には, (2) における $1^n :$ という文字列が格納されることになる. そのまま `H0[n][v]` に移る. ここでは, `n` の初期値は $1^n :$ であり, ずっとその値のままである. また, `v` の初期値は 1 である. また,

```
H0[n][v] = ((v]] (H1[false][true][n][v][v] / H0[n][v 1]);
```

であるから, `v` の値は

$$1, \ 11, \ 111, \ 1111, \ \cdots, \ 1^n$$

まで順番に増えていき[2], そのような `v` に対して順次 `H1[false][true][n][v][v]` が試される. ちなみに, `v` は $n$ 個ある $G_1$ の頂点のうち「スタート地点」となる頂点を選んでいるつもりである. よって, スタート地点となる頂点を $v_1, v_2, \cdots, v_n$ と順番に選んでいき, そのような $v_i$ に対して順次 `H1[false][true][n]`$[v_i][v_i]$ が試されるような感じである. その `H1[track][tlen][n][v][w]` では, `track` に通過済みの頂点が全てリストとして登録されているつもりであり, `tlen` は `track` の要素数のつもりである. スタート地点は未登録と見なすので, `track` の初期値は空リストであり, `tlen` は 0 である. これを MACRO PEG の引数として表現すると, `track = false`, `tlen = true` となる. また, 処理が進むごとに, `track` は

$$1^{c_1} : \ / \ 1^{c_2} : \ / \ \cdots \ / \ 1^{c_k} : \quad (c_i \in [1, n], \ c_i \text{は全て異なる})$$

という形になり, `tlen` は $1^+$ の形になる. もし `tlen = `$1^k$ ならば, `track` の要素数は $k$ 個のつもりである. 残りの `n`, `v`, `w` という引数についてだが, `n` は上述した $1^n :$ に常に固定されている. また, `v` は $1^+$ の形の文字列のうち特定の文字列 1 つ ($v_i$ に応じて 1 が適切な個数だけ並ぶ) に常に固定されている. また, `w` は「現在の頂点」であり, その初期値は `v` であり, 以降は, `w` は常に何らかの $1^+$ の形をしている. この状態で `H1[track][tlen][n][v][w]` の内部に入ると, それは

[1] $m = 0$ のときは, $1^{v_1} : 1^{b_1} : \cdots 1^{v_m} : 1^{b_m} :$ の部分は存在しない.
[2] $z$ の先頭部分は $1^n :$ であるから, `v = `$1^n$ までしか行われない.

```
((tlen 1]] H2[track][tlen][n][v][w][true][true][true][true] /
  (tlen 1] eq[v][w];
```

であるから, tlen によって処理が分岐する. ((tlen 1]] が真であるのは, track の要素数が $n$ 未満のときのみである[3]. このときは, H2 に入り, ハミルトン閉路の探索を行う. これ以外のときは, track の要素数は $n$ である. このときは, 全ての頂点に 1 回ずつ行き渡った状態なので, 現在の地点がスタート地点 $v_i$ であるかをチェックすればよい. すなわち, v と w が同じ頂点を表しているかを確認すればよい. これは eq[v][w] で確認できる[4]. もしここが成功ならば, ハミルトン閉路が存在することになり, それで $e_S$ の処理が終了する. さて, H2 に戻って, 説明を続ける. この H2 から先では, $z$ の先頭部分にある $1^n:$ を $z$ から「無視」したときの[5], 残りの

$$1^{a_1}:1^{b_1}:1^{a_2}:1^{b_2}:\cdots 1^{a_m}:1^{b_m}: \tag{3}$$

を探索用のデータとして, 有向ハミルトン閉路を探索をする. p は, この文字列を指すポインタのように動く. p の初期値は p = true であるから, p は初期状態では (3) の先頭を指している. ただし, 普通に

```
p
```

という指定の仕方で使ってしまうと, $z$ の本来の先頭部分である $1^n:$ の先頭を指すことになってしまい, 意図した動作にならない. そこで, p を使うときは

```
n p
```

という使い方をすることにする. これで, p は (3) の先頭を指すようになる. すると, 処理が進むごとに, p は

```
p = ...(ドットがいくつか並んでいる)...
```

という状態に変化する. しかも, p が

$$1^{a_k}:1^{b_k}:1^{a_{k+1}}:1^{b_{k+1}}:\cdots 1^{a_m}:1^{b_m}: \quad (1\le k\le m+1)$$

の先頭を指すように, ドットの個数が調整される. ただし, $k=m+1$ のときは, p は (3) の末尾を指す. もしそうなっていたら,「失敗」させるべきである. なぜなら, H2 に来たら tlen は $n$ 未満なのだから, まだハミルトン閉路は完成しておらず, それにも関わらず p が (3) の末尾を指すならば, もはやハミルトン閉路は完成できないことになり, よって「失敗」させるべきである. これは, H2 内部の

```
(n p (.]]  ( ((n p px 1]] H2[track][tlen][n][v][w][p][px .][py][y] /
 (n p px 1] H3[track][tlen][n][v][w][p][px .][py][y] );
```

の先頭にある (n p (.]] のところで「失敗」扱いにできる. 一方で, $k\le m$ のときは, (n p (.]] は成功するので, 次の処理に進む. ここでは, px の値が, px = true から出発して

```
px = ...(ドットが ak + 1 個)...
```

[3] この tlen 1 は, $z$ の先頭部分の $1^n:$ に対して試される. よって, tlen = $1^k$ とするとき, tlen 1 が成功するのは $k+1\le n$ のときのみである. すなわち, $k<n$ のときのみである. すなわち, track の要素数が $n$ 未満のときのみである.

[4] この eq[v][w] は, $z$ の先頭部分の $1^n:$ に対して試されるので, いささかトリッキーではあるが, v と w が同じ頂点を表しているかをチェックできる.

[5] $z$ から $1^n:$ の部分を「消費する」わけではない.

の状態になってから H3 に移る．H3 では, py の値が, py = true から出発して

py = ...(ドットが $b_k + 1$ 個)...

の状態になる．さらに, $y$ の値が, y = true から出発して

y = 111$\cdots$11 (1 が $b_k$ 個)

の状態になる．この状態で, 最後の H4 に移る．H4 では,

(( n p w : (track] ]]

がまず実行される．ここでは,

(4) $1^{a_k}$ : に対応する頂点が「現在の頂点」(= w) に一致するなら成功.

(5) さらに, そのあとの $1^{b_k}$ : に対応する頂点が track に登録済みでないなら成功.

となる．(4) のチェックは n p w :で行い, (5) のチェックは (track] で行う．両方とも成功したなら, (( n p w : (track] ]] 全体が成功となる．この場合, $1^{a_k}$ : $1^{b_k}$ : のデータに対応する矢印はハミルトン閉路のパーツとして使えるので, $1^{b_k}$ に対応する頂点 (= y) を track に登録し[6], tlen は 1 だけ増やし, 現在の頂点は y に変更して, H1 に移る．これを行っているのが

H1[track / y :][tlen 1][n][v][y]

の部分である．もしこれが成功したなら, それでハミルトン閉路が見つかったことになる．もし失敗したなら, 探索用のデータの残りを探索すべきであるから,

H2[track][tlen][n][v][w][p px py][true][true][true]

が実行される．もちろん, さっきの (( n p w : (track] ]] が失敗したときも, こちらが実行される．なお, p px py のところは何を意味しているのかというと, 単に p を次のデータ $1^{a_{k+1}}$ : $1^{b_{k+1}}$ : のところまで移動させているだけである．以上で, $G \in$ foMPEG の解説を終わる．

**補足** 上記の $G \in$ foMPEG は, 任意の入力文字列に対して成功または失敗となり, 無限ループには陥らないことが証明できる．このことは, 処理の流れを追っていけばだいたい明らかであるが, きちんと証明すると面倒くさい．詳細は省略する．

[6] 実際には y : を格納する．

## 3　定理 1.2 の証明

**定理 1.2 の証明**　P≠NP とする．定理 1.1 の証明の中で実装した $G \in$ foMPEG を取っておく．foMPEG 自体を最適化することにより, $G$ の計算が入力文字列長の多項式時間内で終わると仮定する．従って, ある $k \geq 0$ が存在して, 任意の入力文字列 $z$ に対して, $z$ を $G$ に適用すると $O(|z|^k)$ の時間内で $G$ の計算が終わる．さて, $n \geq 3$ 個の頂点から成る有向グラフ $G_1$ を任意に取る．定理 1.1 の証明の中で定義した $z \in \{\ 1\ ,\ :\ \}^*$ を考えると,

$$z = 1^n : 1^{a_1} : 1^{b_1} : 1^{a_2} : 1^{b_2} : \cdots 1^{a_m} : 1^{b_m} :$$

という形をしている．また, $(a_i, b_i)$ は重複するものがあったら省いても構わないので, $(a_i, b_i)$ は全て異なるとしてよい．$a_i, b_i \in [1, n]$ だから, $(a_i, b_i)$ の個数は最大でも $n^2$ である．すなわち, $m \leq n^2$ である．よって,

$$\begin{aligned} |z| = (n+1) + \sum_{i=1}^{m}(a_i + b_i + 2) \leq (n+1) + \sum_{i=1}^{n^2}(2n+2) &= (n+1) + (2n+2)n^2 \\ &< (n+1)^3 + 2(n+1)^3 \\ &= 3(n+1)^3 \end{aligned}$$

となる．さて, $z$ を $G$ に適用すると, その計算は $O(|z|^k)$ の時間内で終わるので, 特に $O((3(n+1)^3)^k)$ の時間内で終わる．$G$ が成功なら $G_1$ はハミルトン閉路を持ち, $G$ が失敗なら $G_1$ はハミルトン閉路を持たないのだったから, 以上より, $G_1$ がハミルトン閉路を持つか否かが $O((3(n+1)^3)^k)$ の時間内 (多項式時間内) で判明することになる．これは P≠NP に矛盾する．

# 4 MACRO PEG の最適化に関する考察

定理 1.2 により, もし P≠NP ならば, foMPEG の良い最適化は存在しない. しかし, これは foMPEG 全体を最適化の対象にするからそうなるに過ぎないのであり, foMPEG の中でも限定されたものだけを対象にすれば, 最適化の余地は残されている. たとえば,「全ての規則が引数を持たない」という制約を加えれば, そこで得られる foMPEG は PEG になるので, 入力文字列の長さが $n$ のときに $O(n)$ で動作するような最適化が可能となる. ただし, PEG まで限定したら MACRO PEG の意味がないので, もう少し限定の仕方を緩めたい. そこで, この節では, 次のような限定の仕方をすることにする:

**定義 4.1** $G \in$ foMPEG であって, 引数に与えられている parsing expression の中に非終端記号が全く含まれていないような $G$ の集合を「限定 foMPEG」もしくは「 restricted foMPEG」もしくは「 rfoMPEG」と書くことにする.

**例** [5] に出てくる, 次の $G = (V_N, V_T, R, e_S) \in$ foMPEG を考える:

$V_T$ := { a } $V_N$ := { T, TT, G1, GG1, G2, GG2, G3, GG3, true }

```
true = (a] / ((a]];

T[x] = TT[true][x];
TT[i][x] = (( i a ]] x TT[ i a ][x] / ( i a ];

G1[x] = GG1[true][x];
GG1[i][x] = (( i a ]] GG1[ i a ][T[x]] / ( i a ] x;

G2[x] = GG2[true][x];
GG2[i][x] = (( i a ]] GG2[ i a ][G1[x]] / ( i a ] x;

G3[x] = GG3[true][x];
GG3[i][x] = (( i a ]] GG3[ i a ][G2[x]] / ( i a ] x;
```

$e_S$ := G3[true]

この場合, GG1 の定義に GG1[ i a ][T[x]] という項目があり, 第二引数には T[x] が指定されていて, T という非終端記号が出現してしまっているので, この $G$ は rfoMPEG ではない. というか, この $G$ では, GG2 の定義に GG2[ i a ][G1[x]] という項目があり, 第二引数には G1[x] が指定されていて, G1 という非終端記号が出現してしまっているので, 余計に rfoMPEG ではない.

**例**　[3] に出てくる, 以下の $G = (V_N, V_T, R, e_S) \in$ foMPEG を考える (偶数回文のみを全て受理する):

$V_T$ := { a, b }　$V_N$ := { P, true }

```
true = (a] / ((a]];
P[r] = a P[a r] / b P[b r] / r;
```

$e_S$ := P[true] (.]

この場合, どの引数にも非終端記号が出現していないので, この $G$ は rfoMPEG である.

**考察 4.2** 引数の中に非終端記号が出現できるなら, かなり複雑な MACRO PEG が表現できるに決まっているので, 最適化が難しくなることは明白である. そこで,

引数の中に非終端記号が出現してはならない

という, ありがちだが自然と思われる限定の仕方をする. これが, 今回の rfoMPEG である. これなら最適化の余地が残されているように見える. しかし, 実際にはそうはならない. 実は, 次の定理が成り立つ:

**定理 4.3** もし P≠NP ならば, ある $G \in$ rfoMPEG が存在して, rfoMPEG 自体をどのように最適化しても, $G$ の評価は入力文字列長の多項式時間内では終わらない.

**証明**　定理 1.1 の証明の中で実装した $G \in$ foMPEG を取る. この $G$ は, よく見ると $G \in$ rfoMPEG を満たしていることが分かる. よって, ここから先は定理 1.2 の証明と全く同じ議論ができる.

**考察 4.4** 上記の定理 4.3 により, rfoMPEG という限定の仕方でさえも, P≠NP という仮定のもとで, rfoMPEG には良い最適化が存在しないことになる. そこで, 限定の仕方をさらに強めてみる. 定理 1.1 の証明中の $G$ が「機能しなくなる」ような限定の仕方を考えたい. まず考えられる限定の仕方は, $G$ に出現するリストの機能 (track) を封印することである. track は $G$ のアルゴリズムの中でも本質的に重要な部分であるから, ここを封印されたら, 最適化の余地は出てくる. だが, これを封印すると, MACRO PEG のメリットが 1 つ失われることになる. 既に述べたように, track の手法は, [3] の"grammar: modifiers"で使われている手法を適用したものである. もしこの手法を封印するなら, "grammar: modifiers"も封印されることになる. それでは面白くない. 他に封印できそうな箇所と言えば,「for 文」に相当する手法を封印することである. たとえば,

```
H0[n][v] = ((v]] (H1[false][true][n][v][v] / H0[n][v 1]);
```

この部分では, v の値が

$$1,\ 11,\ 111,\ 1111,\ \cdots,\ 1^n$$

まで順番に増えていき, そのような v に対して順次 H1[false][true][n][v][v] が試されるので, for 文に相当する処理を行っている. そこで, この手法を封印してみたらどうかと考えてみるわけだが, for 文が封印されたら, MACRO PEG のメリットが 1 つ失われる. たとえば, [3] の"grammar:

exponent”では, for 文の手法が使われているのだが, もしこの手法を封印するなら, ”grammar: exponent”も封印されることになる. それでは面白くない. しかし, そのような封印をしないなら,

- for 文は使用可能,
- track のようなリストの手法は使用可能

という状況になる. 実は, 定理 1.1 の証明中の $G$ は, ほぼこの 2 種類の手法しか使ってない. 従って, どちらかを封印しなければ最適化の余地が残されてないように見える. しかし, どちらを封印しても, MACRO PEG としては面白くないように見える.

· · · 個人的には, for 文の機能には面白さを感じる一方で,「やり過ぎ」ではないかと感じることもある. よって, どちらを封印するかと聞かれたら, 個人的には for 文の方を封印したい. ただし, for 文は MACRO PEG の中でかなりシンプルに実装できてしまうので, どうやったら上手く封印できるのかは分からない. また, 仮に封印できても, そこは出発点であり, MACRO PEG の最適化をどうするかが本題である. そこもよく分からない.

# 5　付録

この付録では, PEL $\subsetneq$ MACRO PEL についての考察を行う.

**考察 5.1** PEL $\subsetneq$ MACRO PEL が成り立つことは [4] で既に示したのであるが, 実は, 今回の手法を使っても示せる (P$\neq$NP という仮定のもとで). P$\neq$NP と仮定する. 定理 1.1 の証明の中で実装した $G \in$ foMPEG を取る. $L := L(G)$ と置く. 明らかに $L \in$ MACRO PEL である. 一方で, $L \notin$ PEL である. なぜなら, もし $L \in$ PEL ならば, $L = L(G')$ を満たす $G' \in$ PEG が取れる. Birman [1] の結果により, $G'$ は任意の入力文字列 $z$ に対して無限ループしないとしてよい. また, PEG の最適化により, $G'$ は任意の入力文字列 $z$ に対して $O(|z|)$ の時間内に計算が終わるとしてよい. さて, $n \geq 3$ を任意に取り, 頂点が $n$ 個の有向グラフ $G_1$ を任意に取る. 定理 1.2 の証明のようにして, $G_1$ に対応する文字列 $z \in \{\ 1\ ,\ :\ \}^*$ を取る. $|z| < 3(n+1)^3$ である. $z$ を $G'$ に適用すると, 成功または失敗であり, その計算は $O(|z|)$ の時間内に終わるので, 特に $O(3(n+1)^3)$ の時間内に終わる. もし $G'$ が成功ならば, $z \in L(G') = L = L(G)$ となるので, $z$ は $G$ に適用しても成功することが確定する[7]. $G$ の構成の仕方により, $G_1$ にはハミルトン閉路が存在する. 一方で, $G'$ が失敗のときは, $z \notin L(G') = L = L(G)$ となるので, $z$ は $G$ に適用すると成功しないことが確定する. $G$ の構成の仕方により, $z$ を $G$ に適用すると失敗することが確定する[8]. $G$ の構成の仕方により, $G_1$ にはハミルトン閉路が存在しない. まとめると, 次のようになる:

- $z$ を $G'$に適用すると成功または失敗であり, $O(3(n+1)^3)$ の時間内で計算が終わる.
- $G'$が成功ならば, $G_1$はハミルトン閉路を持つ.
- $G'$が失敗ならば, $G_1$はハミルトン閉路を持たない.

よって, ハミルトン閉路の有無が $n$ の多項式時間内に判明することになり, P$\neq$NP に矛盾する. 以上より, $L \notin$ PEL である. よって, $L \in$ MACRO PEL かつ $L \notin$ PEL となったので, PEL $\subsetneq$ MACRO PEL である.

# 参考文献


[1] Birman, A.: The TMG Recognition Schema, Ph.D. Thesis, Princeton University, 1970.

[2] Ford, B.: Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. In: POPL 2004: Proceedings of the 31st ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 111-122. ACM, New York (2004)

[3] Kota Mizushima: Macro PEG: PEG with macro-like rules
https://github.com/kmizu/macro_peg

[4] Toshihiro Koga: PEG and MACRO PEG
https://github.com/T-K-1/peg_and_macro_peg/blob/master/peg_and_macro_peg.pdf

[5] Toshihiro Koga: Time Complexity of MACRO PEG.
https://github.com/T-K-1/peg_and_macro_peg/blob/master/time_complexity_of_macro_peg.pdf


[7] $z$ を実際に $G$ に適用する必要はなく, 成功することが既に確定している.
[8] $z$ を実際に $G$ に適用する必要はなく, 失敗することが既に確定している.