

Regular expressions à la carte

kbkz.tech #9

吉村 優

https://twitter.com/_yyu_

<http://qiita.com/yyu>

<https://github.com/y-yu>

March 20, 2016

自己紹介



自己紹介

VM 型の正規表現エンジンを実装する
[http://qiita.com/yyu/items/
84b1a00459408d1a7321](http://qiita.com/yyu/items/84b1a00459408d1a7321)



自己紹介

VM 型の正規表現エンジンを実装する

[http://qiita.com/yyu/items/
84b1a00459408d1a7321](http://qiita.com/yyu/items/84b1a00459408d1a7321)

正規表現から LLVM へのコンパイラを実装する

[http://qiita.com/yyu/items/
a0ef2d2204c137707f3f](http://qiita.com/yyu/items/a0ef2d2204c137707f3f)



自己紹介



VM 型の正規表現エンジンを実装する

[http://qiita.com/yyu/items/
84b1a00459408d1a7321](http://qiita.com/yyu/items/84b1a00459408d1a7321)

正規表現から LLVM へのコンパイラを実装する

[http://qiita.com/yyu/items/
a0ef2d2204c137707f3f](http://qiita.com/yyu/items/a0ef2d2204c137707f3f)

正規表現の JIT コンパイラを実装する

[http://qiita.com/yyu/items/
3c4deb39d6b0a7955572](http://qiita.com/yyu/items/3c4deb39d6b0a7955572)



VM 型の正規表現エンジンを実装する

[http://qiita.com/yyu/items/
84b1a00459408d1a7321](http://qiita.com/yyu/items/84b1a00459408d1a7321)

正規表現から LLVM へのコンパイラを実装する

[http://qiita.com/yyu/items/
a0ef2d2204c137707f3f](http://qiita.com/yyu/items/a0ef2d2204c137707f3f)

正規表現の JIT コンパイラを実装する

[http://qiita.com/yyu/items/
3c4deb39d6b0a7955572](http://qiita.com/yyu/items/3c4deb39d6b0a7955572)

正規表現の微分でサブマッチング

[http://qiita.com/yyu/items/
1638fd59bedce27ca3a4](http://qiita.com/yyu/items/1638fd59bedce27ca3a4)

アウトライン

アウトライン

① 正規表現とは？

アウトライン

- ① 正規表現とは？
- ② マッチング

アウトライン

- ① 正規表現とは？
- ② マッチング
- ③ 正規表現の限界

アウトライン

- ① 正規表現とは？
- ② マッチング
- ③ 正規表現の限界
- ④ 正規表現 vs C++

正規表現とは？

正規表現とは？

“ **正規表現**とは、文字列の集合を一つの文字列で表現する方法の一つである。 ”

Wikipedia - 正規表現

正規表現とは？

“ **正規表現**とは、文字列の集合を一つの文字列で表現する方法の一つである。 ”

Wikipedia - 正規表現

“ もともと正規表現は形式言語理論において正規言語を表すための手段として導入された。 ”

Wikipedia - 正規表現

正規言語

“ **正規言語**とは、以下に示す性質を満たす形式言語のことである。

”

Wikipedia - 正規言語

“ **正規言語**とは、以下に示す性質を満たす形式言語のことである。

- 正規表現で記述可能

”

Wikipedia - 正規言語

“ **正規言語**とは、以下に示す性質を満たす形式言語のことである。

- 正規表現で記述可能
- 非決定性有限オートマトン (NFA) で受理可能

”

Wikipedia - 正規言語

“ **正規言語**とは、以下に示す性質を満たす形式言語のことである。

- 正規表現で記述可能
- 非決定性有限オートマトン (*NFA*) で受理可能
- 決定性有限オートマトン (*DFA*) で受理可能

”

Wikipedia - 正規言語

正規表現の定義

文字の集合 Σ 上の正規表現は次のように定義される

正規表現の定義

文字の集合 Σ 上の正規表現は次のように定義される

- 空集合を表す ϕ は正規表現

正規表現の定義

文字の集合 Σ 上の正規表現は次のように定義される

- 空集合を表す ϕ は正規表現
- 空文字を表す ϵ は正規表現

正規表現の定義

文字の集合 Σ 上の正規表現は次のように定義される

- 空集合を表す ϕ は正規表現
- 空文字を表す ϵ は正規表現
- $a \in \Sigma$ は正規表現

正規表現の定義

文字の集合 Σ 上の正規表現は次のように定義される

- 空集合を表す ϕ は正規表現
- 空文字を表す ϵ は正規表現
- $a \in \Sigma$ は正規表現
- X と Y が正規表現であるとき次のものも正規表現

正規表現の定義

文字の集合 Σ 上の正規表現は次のように定義される

- 空集合を表す ϕ は正規表現
- 空文字を表す ϵ は正規表現
- $a \in \Sigma$ は正規表現
- X と Y が正規表現であるとき次のものも正規表現
 - ▶ $X \mid Y$ (選択)

正規表現の定義

文字の集合 Σ 上の正規表現は次のように定義される

- 空集合を表す ϕ は正規表現
- 空文字を表す ϵ は正規表現
- $a \in \Sigma$ は正規表現
- X と Y が正規表現であるとき次のものも正規表現
 - ▶ $X \mid Y$ (選択)
 - ▶ XY (結合)

正規表現の定義

文字の集合 Σ 上の正規表現は次のように定義される

- 空集合を表す ϕ は正規表現
- 空文字を表す ϵ は正規表現
- $a \in \Sigma$ は正規表現
- X と Y が正規表現であるとき次のものも正規表現
 - ▶ $X \mid Y$ (選択)
 - ▶ XY (結合)
 - ▶ X^* (0 回以上の繰り返し)

マッチング

主に次のような方法がある

マッチング

主に次のような方法がある

- 有限オートマトンを用いた方法

マッチング

主に次のような方法がある

- 有限オートマトンを用いた方法
- Virtual Machine を用いた方法

マッチング

主に次のような方法がある

- 有限オートマトンを用いた方法
- Virtual Machine を用いた方法
- 正規表現の微分を用いた方法

非決定性有限オートマトン (NFA)

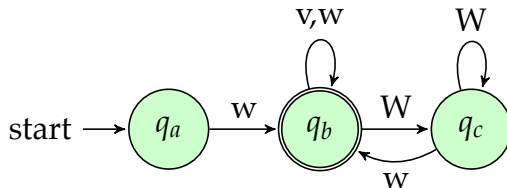
“ **非決定性有限オートマトン**は、有限オートマトンの一種であり、ある状態と入力があったとき、次の遷移先が一意に決定しないことがあるものである。 ”

[Wikipedia - 非決定性有限オートマトン](#)

非決定性有限オートマトン (NFA)

“非決定性有限オートマトンは、有限オートマトンの一種であり、ある状態と入力があったとき、次の遷移先が一意に決定しないことがあるものである。”

Wikipedia - 非決定性有限オートマトン

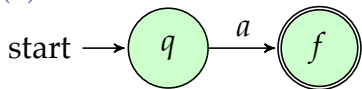


NFA を用いたマッチング

- 全ての正規表現には対応する非決定性有限オートマトンが存在する

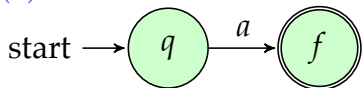
正規表現と NFA

文字 $N(a)$

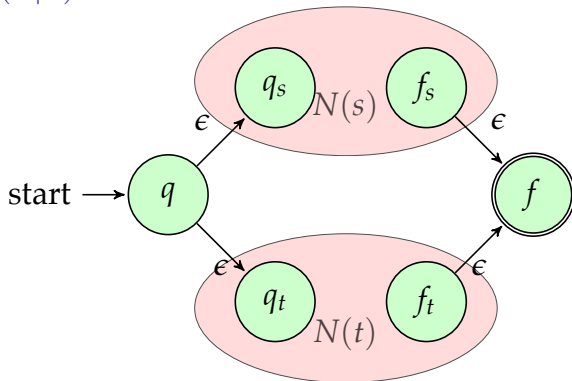


正規表現と NFA

文字 $N(a)$

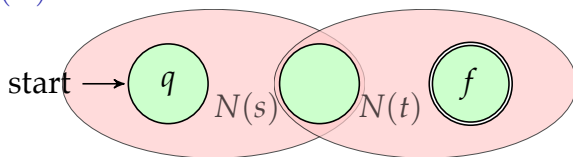


選択 $N(s \mid t)$



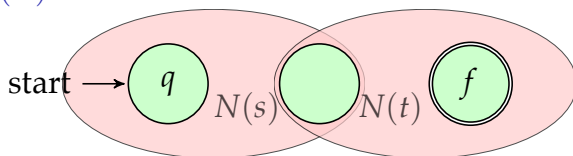
正規表現と NFA

結合 $N(st)$

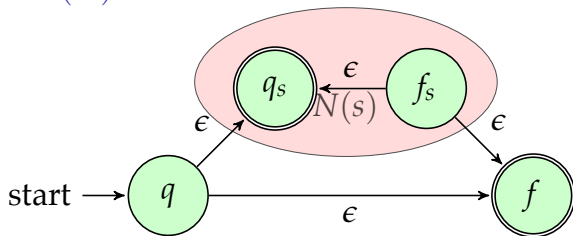


正規表現と NFA

結合 $N(st)$



繰り返し $N(s^*)$



NFA を用いたマッチング

- 全ての正規表現には対応する非決定性有限オートマトンが存在する
- 遷移先が一意に決まらないので、バックトラックをする必要がある

決定性有限オートマトン (DFA)

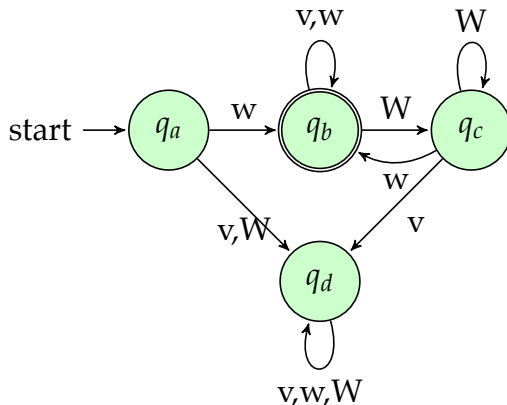
“決定性有限オートマトンは、状態と入力によって次に遷移すべき状態が一意に定まる有限オートマトンである。”

[Wikipedia - 決定性有限オートマトン](#)

決定性有限オートマトン (DFA)

“決定性有限オートマトンは、状態と入力によって次に遷移すべき状態が一意に定まる有限オートマトンである。”

Wikipedia - 決定性有限オートマトン



DFA を用いたマッチング

- 非決定性有限オートマトンから機械的に変換できる（サブセット構成など）

DFA を用いたマッチング

- 非決定性有限オートマトンから機械的に変換できる（サブセット構成など）
- 入力によって状態が一意に決まるので、バックトラックをする必要がない

DFA を用いたマッチング

- 非決定性有限オートマトンから機械的に変換できる（サブセット構成など）
- 入力によって状態が一意に決まるので、バックトラックをする必要がない
- 非決定有限オートマトンから変換すると、最悪の場合、**状態数が指数関数的に増加する**

DFA を用いたマッチング

- 非決定性有限オートマトンから機械的に変換できる（サブセット構成など）
- 入力によって状態が一意に決まるので、バックトラックをする必要がない
- 非決定有限オートマトンから変換すると、最悪の場合、**状態数が指数関数的に増加する**
- そのため、必要になった状態だけ変換するというテクニックがある

Virtual Machine (VM) を使ったマッチング

Virtual Machine (VM) を使ったマッチング

次のような VM を用いて正規表現のマッチングが可能

Virtual Machine (VM) を使ったマッチング

次のような VM を用いて正規表現のマッチングが可能

- PC と SP という 2 つのレジスタがある

Virtual Machine (VM) を使ったマッチング

次のような VM を用いて正規表現のマッチングが可能

- PC と SP という 2 つのレジスタがある
- 次の命令がある

Virtual Machine (VM) を使ったマッチング

次のような VM を用いて正規表現のマッチングが可能

- PC と SP という 2 つのレジスタがある
- 次の命令がある

`char c`

SP の先頭の文字と `c` を比較

Virtual Machine (VM) を使ったマッチング

次のような VM を用いて正規表現のマッチングが可能

- PC と SP という 2 つのレジスタがある
- 次の命令がある

`char c`

SP の先頭の文字と `c` を比較

`match`

マッチに成功 (スレッドを終了)

Virtual Machine (VM) を使ったマッチング

次のような VM を用いて正規表現のマッチングが可能

- PC と SP という 2 つのレジスタがある
- 次の命令がある

`char c`

SP の先頭の文字と `c` を比較

`match`

マッチに成功 (スレッドを終了)

`jmp x`

アドレス `x` へジャンプ (PC を `x` にする)

Virtual Machine (VM) を使ったマッチング

次のような VM を用いて正規表現のマッチングが可能

- PC と SP という 2 つのレジスタがある
- 次の命令がある

`char c`

SP の先頭の文字と `c` を比較

`match`

マッチに成功 (スレッドを終了)

`jmp x`

アドレス `x` へジャンプ (PC を `x` にする)

`split x, y`

スレッドを二つに分割する。片方は PC を `x` にし、もう片方は PC を `y` にする

Virtual Machine (VM) を用いたマッチング

正規表現を次のように変換する

Virtual Machine (VM) を用いたマッチング

正規表現を次のように変換する

文字 (*c*)

`char c`

Virtual Machine (VM) を用いたマッチング

正規表現を次のように変換する

文字 (c)

`char c`

連結 (e_1e_2)

e_1 の命令列

e_2 の命令列

Virtual Machine (VM) を用いたマッチング

正規表現を次のように変換する

文字 (c)

char c

選択 ($e_1 \mid e_2$)

split $L_1 L_2$

L_1 : e_1 の命令列

jmp L_3

L_2 : e_2 の命令列

L_3 :

連結 ($e_1 e_2$)

e_1 の命令列

e_2 の命令列

Virtual Machine (VM) を用いたマッチング

正規表現を次のように変換する

文字 (c)

char c

選択 ($e_1 \mid e_2$)

split $L_1 L_2$
 L_1 : e_1 の命令列
 jmp L_3
 L_2 : e_2 の命令列
 L_3 :

連結 ($e_1 e_2$)

e_1 の命令列
 e_2 の命令列

繰り返し (e^*)

L_1 : split $L_1 L_3$
 L_2 : e の命令列
 jmp L_1
 L_3 :

Virtual Machine (VM) を用いたマッチング

正規表現/aa*bb*/を VM のバイトコードへ変換する

```
0      char a
1      split 2, 4
2      char a
3      jmp 1
4      char b
5      split 6, 8
6      char b
7      jmp 5
8      match
```

Virtual Machine (VM) を用いたマッチング

正規表現/aa*bb*/を VM のバイトコードへ変換する

```
0      char a
1      split 2, 4
2      char a
3      jmp 1
4      char b
5      split 6, 8
6      char b
7      jmp 5
8      match
```

- VM の命令を LLVM や JVM に変換すれば、より高速になる

正規表現の微分を用いたマッチング

* この文字列 $c : wc$ は文字 c が文字列の先頭の 1 文字で、 cw は文字列の先頭以外の残りを表す

正規表現の微分を用いたマッチング

正規表現の微分

$c : wc$ という文字列があるとする*。ある正規表現 r が文字列 $w : wc$ にマッチするならば、 r を文字 c で微分した正規表現 r_c は文字列 wc にマッチする。

* この文字列 $c : wc$ は文字 c が文字列の先頭の 1 文字で、 cw は文字列の先頭以外の残りを表す

正規表現の微分を用いたマッチング

正規表現の微分

$c : wc$ という文字列があるとする*。ある正規表現 r が文字列 $w : wc$ にマッチするならば、 r を文字 c で微分した正規表現 r_c は文字列 wc にマッチする。

- ① マッチング対象の文字列から 1 文字ずつ取り出し、正規表現を微分していく

* この文字列 $c : wc$ は文字 c が文字列の先頭の 1 文字で、 cw は文字列の先頭以外の残りを表す

正規表現の微分を用いたマッチング

正規表現の微分

$c : wc$ という文字列があるとする*。ある正規表現 r が文字列 $w : wc$ にマッチするならば、 r を文字 c で微分した正規表現 r_c は文字列 wc にマッチする。

- ① マッチング対象の文字列から 1 文字ずつ取り出し、正規表現を微分していく
- ② 文字列が空になった時に、微分された正規表現が空文字を受け理するならばマッチングに成功

* この文字列 $c : wc$ は文字 c が文字列の先頭の 1 文字で、 cw は文字列の先頭以外の残りを表す

正規表現の限界

`/^1?$ | ^(11+)\1+$/`

正規表現の限界

1 が非素数個ある文字列の正規表現

$/^1?\$|^ (11+)\backslash 1+\$ /$

正規表現の限界

1 が非素数個ある文字列の正規表現

$/^1?\$|^ (11+)\backslash 1+\$ /$

例

次のような文字列がマッチする

- $\underbrace{1}_{1?}$
- $\underbrace{11}_{11+?} \underbrace{11}_{\backslash 1}$
- $\underbrace{111}_{11+?} \underbrace{111}_{\backslash 1} \underbrace{111}_{\backslash 1}$

正規表現と非正規表現

正規表現と非正規表現

- **ポンピング補題**などを使うと、正規表現で表せる集合かどうか証明できる

正規表現と非正規表現

- **ポンピング補題**などを使うと、正規表現で表せる集合かどうか証明できる
- たとえば、次のものは**正規**である
 - ▶ ある正規表現の補集合を表す正規表現
 - ▶ 先読み

正規表現と非正規表現

- **ポンピング補題**などを使うと、正規表現で表せる集合かどうか証明できる
- たとえば、次のものは**正規**である
 - ▶ ある正規表現の補集合を表す正規表現
 - ▶ 先読み
- たとえば、次のものは**非正規**である
 - ▶ 後方参照
 - ▶ 再帰

正規表現と非正規表現

- **ポンピング補題**などを使うと、正規表現で表せる集合かどうか証明できる
- たとえば、次のものは**正規**である
 - ▶ ある正規表現の補集合を表す正規表現
 - ▶ 先読み
- たとえば、次のものは**非正規**である
 - ▶ 後方参照
 - ▶ 再帰
- 非素数にマッチする正規表現は後方参照を用いていたので、非正規表現である

正規表現と非正規表現

- **ポンピング補題**などを使うと、正規表現で表せる集合かどうか証明できる
- たとえば、次のものは**正規**である
 - ▶ ある正規表現の補集合を表す正規表現
 - ▶ 先読み
- たとえば、次のものは**非正規**である
 - ▶ 後方参照
 - ▶ 再帰
- 非素数にマッチする正規表現は後方参照を用いていたので、非正規表現である
- 非正規になると、正規表現が持つよい性質が失われる

正規表現 vs C++[†]

[†]この内容は新屋さんの成果です

Hello, world!

Hello, world!

```
#include <iostream>

int main()
{
    std::cout << "Hello, World!\n";
    return 0;
}
```

Hello, world!

Hello, world!

```
#include <iostream>

int main()
{
    std::cout << "Hello, World!\n";
    return 0;
}
```

- Hello, world!を表示する C++のプログラム

Hello, world!

Hello, world!

```
#include <iostream>

int main()
{
    std::cout << "Hello, World!\n";
    return 0;
}
```

- Hello, world!を表示する C++のプログラム
- 全体で 86 Byte

Grass

Grass

“ *Grass is a functional grass-planting programming language.* ”

Grass the grass-planting programming language

Grass

Grass

“ *Grass is a functional grass-planting programming language.* ”

Grass the grass-planting programming language

- C++と同じくチューリング完全なプログラム言語

Grass

Grass

“ *Grass is a functional grass-planting programming language.* ”

Grass the grass-planting programming language

- C++と同じくチューリング完全なプログラム言語
- w と W と v の組み合わせでプログラムが記述できる

Grass

Grass

“ *Grass is a functional grass-planting programming language.* ”

Grass the grass-planting programming language

- C++と同じくチューリング完全なプログラム言語
- w と W と v の組み合わせでプログラムが記述できる
- 文法が次のように定義される

$$app ::= W^+ w^+$$

$$abs ::= w^+ app^*$$

$$prog ::= abs \mid prog \vee abs \mid prog \vee app^*$$

Grass

Grass

“ *Grass is a functional grass-planting programming language.* ”

Grass the grass-planting programming language

- C++と同じくチューリング完全なプログラム言語
- w と W と v の組み合わせでプログラムが記述できる
- 文法が次のように定義される

$$app ::= W^+ w^+$$

$$abs ::= w^+ app^*$$

$$prog ::= abs \mid prog \vee abs \mid prog \vee app^*$$

- 文法を**正規表現**で表せる
 $w^+(W+w^+)^*((vw^+)^*(W+w^+)^+)^*$

Hello, world!

Hello, world!

“

[illegible]

“

<http://d.hatena.ne.jp/rst76/20080708/1215507578>

Hello, world!

Hello, world!

“

[illegible]

“

<http://d.hatena.ne.jp/rst76/20080708/1215507578>

- Grass で Hello, world! を表示するプログラム

Hello, world!

Hello, world!

“

[illegible]

“

<http://d.hatena.ne.jp/rst76/20080708/1215507578>

- Grass で Hello, world! を表示するプログラム
- 613 Byte

RANS

RANS

“RANS's concept is very simple, just calculates the number from the given string on a regular language.”

<http://sinya8282.github.io/RANS/>

RANS

RANS

“RANS's concept is very simple, just calculates the number from the given string on a regular language.”

<http://sinya8282.github.io/RANS/>

ある文字列が、ある正規表現が表す集合の何番目に位置するのか
計算するプログラム

Hello, world!

Hello, world!を表示するプログラムは

Hello, world!

Hello, world!を表示するプログラムは

```
206602040924489026801307911538805854455198744538070639182
690880761572424911476339257257386704995410169812907136566
327541800453415305636200196255437899960016989064923911660
800780259871999465815761462533370076235558393587722548474
1358948834773649859816115717
```

番目の Grass プログラム

Hello, world!

Hello, world!を表示するプログラムは

```
206602040924489026801307911538805854455198744538070639182
690880761572424911476339257257386704995410169812907136566
327541800453415305636200196255437899960016989064923911660
800780259871999465815761462533370076235558393587722548474
1358948834773649859816115717
```

番目の Grass プログラム

- バイナリにすれば 107 Byte

Hello, world!

Hello, world!を表示するプログラムは

```
206602040924489026801307911538805854455198744538070639182
690880761572424911476339257257386704995410169812907136566
327541800453415305636200196255437899960016989064923911660
800780259871999465815761462533370076235558393587722548474
1358948834773649859816115717
```

番目の Grass プログラム

- バイナリにすれば 107 Byte
- C++と比べて 21 Byte 差

Hello, world!

Hello, world!を表示するプログラムは

```
206602040924489026801307911538805854455198744538070639182
690880761572424911476339257257386704995410169812907136566
327541800453415305636200196255437899960016989064923911660
800780259871999465815761462533370076235558393587722548474
1358948834773649859816115717
```

番目の Grass プログラム

- バイナリにすれば 107 Byte
- C++と比べて 21 Byte 差
- 正規表現を工夫すれば、もっと小さくなる（かも）

参考文献



新屋良磨, 鈴木勇介, 高田謙.

正規表現技術入門 — 最新エンジン実装と理論的背景
(WEB+DB PRESS plus).

技術評論社, 4 2015.

- 1 自己紹介
- 2 正規表現とは？
 - 正規言語
- 3 マッチング
 - NFA を用いたマッチング
 - DFA を用いたマッチング
 - VM を用いたマッチング
 - 正規表現の微分を用いたマッチング
- 4 正規表現の限界
 - 正規表現と非正規表現
- 5 正規表現 vs C++
 - C++
 - Grass
 - RANS
- 6 参考文献

Thank you for listening!

ポンピング補題

L が正規言語[‡] であるならば、次が成り立つ。

[‡]正規表現で表現できる言語であるという意味

ポンピング補題

L が正規言語[‡]であるならば、次が成り立つ。

任意の正規言語 L について、言語 L についてのみ依存する反復長 $l > 0$ が存在し、 $|w| \geq l$ となる全ての $w \in L$ について次が成り立つ

- ① $|y| > 0$ かつ $|xy| \leq l$ となる②を満たす w の分割 $w = xyz$ が存在する
- ② すべての $i \geq 0$ について $xy^iz \in L$ が成り立つ

[‡]正規表現で表現できる言語であるという意味

ポンピング補題

非素数言語を $L_{np} = \{1^{\alpha \cdot \beta} \mid \alpha > 1, \beta > 1\}$ として、
 $w = 11111 \in L_{np}, l = 1$ とする

ポンピング補題

非素数言語を $L_{np} = \{1^{\alpha \cdot \beta} \mid \alpha > 1, \beta > 1\}$ として、
 $w = 111111 \in L_{np}, l = 1$ とする

反例

$l = 1$ かつ $|y| > 0, |xy| \leq l = 1$ より、 $|x| = 0, |y| = 1$ となる。従って $w = \underbrace{\quad}_x \underbrace{1}_y \underbrace{11111}_z$ とすると、

$|xy^2z| = |xyz| + |y| = 6 + 1 = 7$ となり、7は素数であることから、 $xy^2z \notin L_{np}$ となり、ポンピング補題を満たさない