

# 動的なコード生成を用いた正規表現評価器の実装

新屋 良磨, 河野 真治

当研究室では, Continuation based C という, 状態遷移記述に適した C の下位言語を提案している. Continuation based C は ステートメントより大きく, 関数よりも小さなプログラミング単位としてコードセグメントを持ち, コードセグメントからの継続を基本としている. 本研究では, 与えられた正規表現から, 等価な有限状態オートマトンに変換し, オートマトンにおける状態遷移を Continuation based C による継続に変換する正規表現コンパイラを Python で実装した. なお, ここで言うコンパイルとは, プログラム内部での中間表現への変換だけでなく, 実行時バイナリの生成までを指す.

## 1 はじめに

コンパイラ理論の発展と共に, コンパイルにかかる時間はより短く, また得られるプログラムはアセンブラレベルで最適化が施され, より高速になってきている.

完全に静的なコンパイルが可能な対象として, 正規表現評価器に着目した. 現在, 正規表現の評価器は, プログラミング言語の組み込み機能やライブラリ等, さまざまな実装が存在するが, それらの殆どは仮想マシン方式を採用している [2]. 仮想マシンを採用した実装でも, 正規表現を内部表現に変換する処理を行っており, それらを “コンパイル” と呼ぶことが多い. 本研究で実装した評価器の “コンパイル” とは, 正規表現を内部形式に変換することではなく, 正規表現から実行バイナリを生成することを指す (3.3 節). 本研究では, 実行バイナリの生成にはコンパイラ基盤である LLVM, GCC を用いており, 評価器全体の実装としては Python で実装した.

本論文では, まず正規表現のコンパイル方法について

説明し, 実装した評価器の性能調査のために, 正規表現を用いてテキストマッチ処理を行う grep と同等の機能を実装し, GNU grep との比較を行う.

## 2 正規表現

### 2.1 正規表現によるテキストマッチ

正規表現は与えられた文字列を受理するかどうかを判定できるパターンマッチングの機構であり, sed, grep, awk を始めとするテキスト処理ツールに広く利用されている. 正規表現には定められた文法によって記述され, 例えば, 正規表現 “ $a^*b$ ” は “ $a$ ” の 0 回以上の繰り返し直後, “ $b$ ” で終わる文字列 (“ $b$ ”, “ $ab$ ”, “ $aaaab$ ”) を受理し, “ $a(b|c)$ ” は “ $a$ ” で始まり, 直後が “ $b$ ” または “ $c$ ” で終わる文字列 (“ $ab$ ”, “ $ac$ ”) を受理する.

### 2.2 正規表現の演算

本論文では, 以下に定義された演算をサポートする表現を正規表現として扱う.

1. 接続 二つの言語  $L$  と  $M$  の接続 ( $LM$ ) は,  $L$  に属する列を一つとり, そのあとに  $M$  に属する列を接続することによってできる列全体から成る集合である.

2. 集合和 二つの言語  $L$  と  $M$  集合和  $(L|M)$  は,  $L$  または  $M$  (もしくはその両方) に属する列全体からなる集合である.
3. 閉包 言語  $L$  の閉包  $(L^*)$  とは,  $L$  の中から有限個の列を重複を許して取り出し, それらをつなげてできる列全体の集合である.

正規表現は, この 3 つの演算について閉じており, この 3 つの演算によって定義される表現は, 数学的には正規表現と定義されている. 本論文では, 特に区別のない限り, 正規表現と正規表現を同じものとして扱う.

### 2.3 grep

正規表現は, テキストのパターンをシンプルに記述できるという利点から, テキストファイルから, 任意のパターンにマッチするテキストを検索するなどの用途に使用される.

GNU grep は, それを実現するソフトウェアの一つであり, 引数として与えられたファイルから, 与えられた正規表現にマッチするテキストを含む行を出力する機能を持っている.

“与えられた正規表現にマッチするテキストを含む” というのは, 行の先頭から末尾まで正規表現によるマッチングを行い, 正規表現が受理状態になった時点で “含む” という解釈を行う. つまり, 正規表現 “ $(a|s)t$ ” は, “ $at$ ” または “ $st$ ” を受理する正規表現であり, テキスト行 “ $math.$ ” の 23 文字目の “ $at$ ” に一致するので grep は “ $math.$ ” を出力する. また正規表現 “ $a^*$ ” は, “ $a$ ” の 0 回以上の繰り返しを受理する正規表現であり, 空文字も受理するので, grep は全ての行を出力することになる.

## 3 正規表現評価器の実装

正規表現は等価な NFA に, また NFA は等価な DFA に変換することが可能である [4]. 以下にその変換手方を説明する.

### 3.1 正規表現から NFA への変換

NFA (Non-deterministic Finite Automaton) は, 入力に対して複数の遷移先を持つ状態の集合であり, 遷移先が非決定的 (Non-deterministic) である. ここ

では, NFA を 5 個組  $(Q, \Sigma, \delta, q_0, F)$  で定義する. ただし,

1.  $Q$  は状態の有限集合.
2.  $\Sigma$  は入力記号の有限集合.
3.  $q_0$  は  $Q$  の要素で, 開始状態と呼ぶ.
4.  $F$  は  $Q$  の部分集合で, 受理状態と呼ぶ.
5.  $\delta$  は, 状態と入力記号に対して状態の集合を返す遷移関数. ( $\epsilon$  遷移を許す)

正規表現が, 等価な NFA に変換できるということを, 2.2 で定義した 3 つの演算について対応する NFA に変換できることから示す.

1. 接続 図 1 は正規表現 “ $AB$ ” に対応する NFA.
2. 集合和 図 2 は正規表現 “ $A|B$ ” に対応する NFA.
3. 閉包 図 3 は正規表現 “ $A^*$ ” に対応する NFA.

図 2, 3 において, ラベルのない矢印は無条件の遷移を現しており,  $\epsilon$  遷移と呼ばれる. また, 二重丸で囲まれた状態は受理状態を現しており, NFA において入力が終了した時点で, 受理状態を保持している場合に

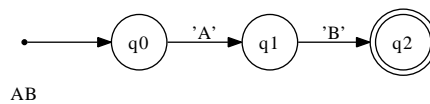


図 1 “A” と “B” の接続

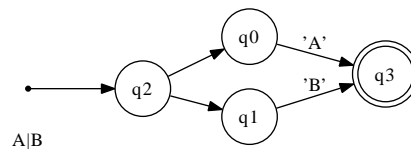


図 2 “A” と “B” の集合和

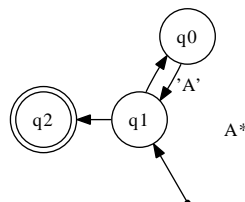


図 3 “A” の閉包

限り, その文字列を受理したことになる. なお, NFA は同時に複数の遷移先をもつことがあるので, テキストのマッチング途中で複数の状態を保持することができる.

現在実装されている正規表現評価器の多くは, 正規表現を内部的に NFA に変換して評価を行っている [1]. NFA による実装は, 後述する後方参照や最長一致に対応しやすいが, 同時に遷移する可能性のある複数の状態を保持する必要があるため, 正規表現の複雑度に多じてマッチングの時間が多くなってしまう場合がある. 文献 [1] では, “ $a?a?aaa$ ” のような “ $a^n a^n$ ” のように表現 (“ $a?$ ” は “ $a$ ” “空文字” を認識する拡張された正規表現の一つ) の評価において, NFA ベースの正規表現評価器では遷移する状態の数が増えてしまうでマッチングにかかる処理時間が  $n$  の指数的に増加する問題をベンチマーク結果と共に指摘している. 文献 [6] では正規表現から NFA ベースで効率的なマッチング処理を行う評価器を IBM 7094 機械語で生成する例が紹介されている.

### 3.2 NFA から DFA への変換

非決定的な遷移を行う NFA から, 決定的な遷移を行う DFA (*Deterministic Finite Automaton*) に変換する手法を説明する. なお, 遷移が決定的であるということは, 1 つの入力に対して, 遷移する状態がただ 1 つであるということを示す. DFA は, NFA と同様な 5 個組で  $(Q, \Sigma, \delta, q_0, F)$  定義できる. ただし, DFA において  $\delta$  において  $\varepsilon$  遷移は認められず, また任意の状態  $q$  と入力  $\sigma$  について,  $\delta(q, \sigma) = q'$  となる  $q'$  は  $Q$  の要素となる. つまり, 遷移先が決定的であるということに他ならない.

以下に  $\varepsilon$  遷移を許す  $\varepsilon$ -NFA  $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$  から等価な DFA  $D = (Q_D, \Sigma, \delta_D, q_D, F_D)$  を構成する手順を示す.

1.  $Q_D$  は  $Q_E$  の部分集合全から成る集合であり, おの中で  $D$  において到達可能な状態は,  $\varepsilon$  遷移に関して閉じている  $Q_E$  の部分集合  $S$  に限られる. ここで, 状態  $q$  において  $\varepsilon$  遷移に関して閉じている集合全体を  $ECLOSE(q)$  と表す.  $ECLOSE$

を使って  $S$  を定義すると,  $S = \bigcup_{q \in S} ECLOSE(q)$

を満たす  $S$ .

2.  $q_D = ECLOSE(q_0)$ . すなわち,  $E$  の開始状態の  $\varepsilon$  閉包.
3.  $F_D$  は  $E$  の状態の集合で, 受理状態を少なくとも一つ含むもの全体からなる集合である. すなわち,  $F_D = \{S | S \in Q_D \wedge S \cap F_E \neq \emptyset\}$
4.  $\delta_D(S, a)$  は  $Q_D$  の要素  $S$  と  $\Sigma$  の要素  $a$  に対して次のように計算される.

(a)  $S_k = \{p_1, p_2, \dots, p_k\}$  とする.

(b)  $\bigcup_{i=1}^k \delta(p_i, a)$  を求め, その結果を  $\{r_1, r_2, \dots, r_m\}$  とする.

(c) このとき,  $\delta_D(S, a) = \bigcup_{j=1}^m ECLOSE(r_j)$

この方法によって得られた DFA  $D$  は NFA  $E$  と同等の言語を認識し, また NFA の元となる正規表現と同等である.

### 3.3 DFA からの実行バイナリ生成

DFA からの実行バイナリ生成には, 生成するコードについて 3 種類の実装を行った.

1. DFA  $\rightarrow$  Continuation based C  $\rightarrow$  GCC によるコンパイル
2. DFA  $\rightarrow$  C  $\rightarrow$  GCC によるコンパイル
3. DFA  $\rightarrow$  LLVM 中間表現  $\rightarrow$  LLVM によるコンパイル

以下, Continuation based C, LLVM の説明と, それを利用した DFA からの実行バイナリ生成の方法を説明する.

#### 3.3.1 Continuation based C

Continuation based C (以下 CbC) は, プログラムの基本単位としてコードセグメントを持ち, コードセグメント間の軽量継続を基本とした C の下位言語である. 本研究室での先行研究により CbC コンパイラは, GNU C Compiler 上で実装されており [7], GCC の末尾再帰最適化を強制することで, 関数と同様の記述が可能で, かつ関数呼び出しに伴うリターンアドレスの保存や, スタックの成長のない, “引数付き goto” として継続を実装している. 本研究では

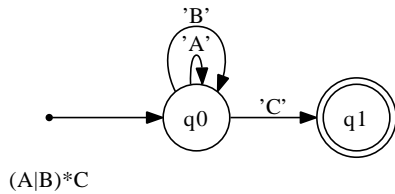


図 4 正規表現“(A|B)\*C”に対応する DFA

gcc-4.5 上に実装された CbC コンパイラを用いた。

以下に、正規表現“(A|B)\*C”に対応する DFA と、DFA の各状態に対応する CbC のコードセグメントの生成例を示す。

```

__code state_0(unsigned char *s, unsigned
  char* cur, unsigned char* buf, FILE
    *f, char* filename) {
  switch(*s++) {
    case 65: /* match A */
      goto state_0(s, cur, buf, f, filename);
    case 66: /* match B */
      goto state_0(s, cur, buf, f, filename);
    case 67: /* match C */
      goto state_1(s, cur, buf, f, filename);
    default: goto reject(s, cur, buf, f, filename);
  }
}
__code state_1(unsigned char *s, unsigned
  char* cur, unsigned char* buf, FILE
    *f, char* filename) {
  goto accept(s, cur, buf, f, filename);
}

```

#### 図 4 の DFA に対応する CbC コードセグメント

DFA の遷移とは直接関係のない引数 (ファイル名やバッファへのポインタ等) が目立が、CbC では環境をコードセグメント間で引数として明示的に持ち運ぶ軽量継続を基盤としたプログラミングスタイルが望ましい。今回コンパイラによって生成した CbC ソースコードでは、大域変数は持たず、必要な変数は全て引数に載せている。CbC の state\_1, state\_0 から呼ばれている accept, reject はそれぞれ受理と非受理を表す。accept ではテキスト行を出力して次の行へ、reject では次の文字へと処理を移すコードセグメントへ継続を行う。

生成した CbC ソースコードを、GCC 上に実装した CbC コンパイラによってコンパイルすることで実行

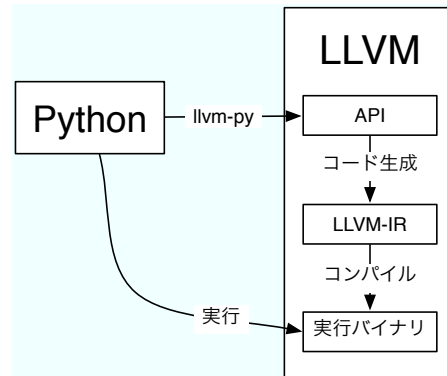


図 5 LLVM を用いた実装

バイナリを得る。

#### 3.3.2 C

C による実装では、CbC のコードセグメントに代わり関数を用いて DFA を実装した。DFA による遷移を関数呼び出しで行っているため、実行時のスタックの使用領域や、スタック操作によるオーバーヘッドが予想される。

#### 3.3.3 LLVM

LLVM (Low Level Virtual Machine) は、さまざまなコード最適化/分析機能を備えた、モジュール単位で利用可能なコンパイラ基盤である [5]。

CbC/C による実装では、DFA から CbC/C のソースコードに変換し、GCC によってコンパイルを行っている。LLVM による実装では、LLVM の中間表現である LLVM-IR を、提供されている API を用いて直接操作を行い、コンパイルを経て実行バイナリを得る。Python から LLVM API の利用は、LLVM API の Python ラッパーである llvm-py<sup>†1</sup>を使用した。

LLVM による実装でも、C による実装と同様に、DFA の状態遷移を switch 文と関数呼び出しによって表現している。

<sup>†1</sup> <http://www.mdevan.org/llvm-py/>

## 4 評価

### 4.1 実験

本実験で実装した正規表現評価器の CbC/C/LLVM による三つの実装に対して、コンパイル時間及びマッチングにかかる時間の比較を行った。なお、GCC によるコンパイルには最適化オプション“-O3”を、LLVM のも同様の最適化オプションを用いてコンパイル/マッチングを行っている。

#### 実験環境

- CPU : Core 2 Duo 950 @3.0GHz
- Memory : 16GB
- GCC : 4.5.0
- LLVM: 2.4

#### コンパイル

$n$  個の単語を正規表現の和集合演算 “[|]” で繋げたパターンに対し、各実装のコンパイル時間の比較を行った。

表 1 ベンチマーク:compile

n (単語数)	1	10	50	100
DFA 変換 [ms]	0.19	3.28	22.2	73.8
DFA の状態数	8	50	187	381
GCC-C [s]	0.34	0.78	2.18	4.27
GCC-CbC[s]	0.75	1.03	9.14	9.43
LLVM [s]	0.044	0.08	0.246	0.472

表 1 から、LLVM によるコンパイルが GCC に比べ 10 倍程高速に行われている。LLVM による実装では、API を通じて直接 LLVM の中間表現を操作することで、ファイル I/O やパース等のオーバーヘッドもない。

#### マッチング

マッチング時間の比較では、様々な正規表現を用いて比較を行った結果、3 つの実装でマッチング時間にあまり差が見られなかった。生成されるコードはコードセグメント/関数と、switch 文によるシンプルな実装であることから、コンパイルされたバイナリの性能にあまり差が出ていないものだと思う。

本実装の中で最もマッチングが高速だった GCC-C で生成した正規表現評価器を用いて grep に相当するプログラムを実装し、実際にテキストファイルからのパターンマッチを行い、それぞれの評価を行った。

#### 実験環境

- CPU : Core i7 950 @3.0GHz
- Memory : 16GB
- GCC : 4.4.1
- Text : Wikipedia 日本語版全記事 (XML, UTF-8, 4.7GB, 8000 万行)

表 2 に結果を示す。

表 2 ベンチマーク:grep

テストケース	fixed-string	simple-regex	complex-regex
本実装 (GCC-C)[s]	13.90	15.45	14.83
(コンパイル [s])	0.15	0.17	0.23
GNU grep 2.6.3[s]	2.93	5.65	16.86
GNU grep 2.5.4[s]	2.96	6.37	188.51

以下に、それぞれのテストケースのパターン、grep にマッチした行数、および考察を記す。なお、ここで扱う正規表現の“複雑さ”とは、DFA に変換した時点の状態数、遷移規則の多さを基準としている。

#### fixed-string 固定文字列によるマッチング

パターン : “*Wikipedia*”

マッチ行数: 348936 行

GNU grep では、与えられたパターン内に、確実に含まれる文字列 (固定文字列) が存在する場合は、Boyer-Moore 法等の高速な文字列探索アルゴリズムを用いてフィルタをかけることで、DFA によるマッチングを減らし、高速化している。本実装の grep では、そのようなフィルタリングは施しておらず、どのようなパターンにたいしても先に述べた DFA ベースのマッチングを行う。fixed-string のテストケースではその差が出ている。

#### simple-regex 単純な正規表現でのマッチング

パターン : “ $\backslash * + \backslash [ \backslash ]$ ”

マッチ行数: 3439028 行

このパターンは行頭から '\*' の 1 回以上の繰り返しの後に文字列 “ \[\\] ” が続く行にマッチする (Wikipedia における “関連項目”). このパターンにも、確実に含まれる文字列が存在するので、GNU grep では Boyer-Moore 法でフィルタリングを行った後マッチングを行うことで本実装と比べて高速な結果となっている。

**complex-regex** 複雑な正規表現でのマッチングパターン :“(Python|Perl|Pascall|Prolog|PHP|Ruby|Haskell|Lisp|Scheme)”

マッチ行数: 1503 行

このパターンは、9 個の単語を和集合演算 “|” で並べたもので、確実に含まれる文字列は存在せず、先の二つに比べて GNU grep は遅い結果となっている。

GNU grep 2.5.4 は 188 秒と、本実装及び GNU grep 2.6.3 に対して非常に遅い結果となっているが、これは後述する `mbrtowc(3)` によるマルチバイト文字の変換処理によるオーバーヘッドによるものである。

3 つのテストケースの結果を見てみると、本実装はそれぞれ実行時間にあまり差がなく、またコンパイル時間はマッチングにかかる時間と比べて無視できるほど短い時間である。また、`complex-regex` のテストケースでは GNU grep よりも高速な結果が出ており、本実装は (フィルタリングを用いない) 純粋な DFA マッチングにおいて、本実装によって動的に生成されたコードの性能が高いことが分かる。

## 4.2 特徴

本実験によって実装された正規表現評価器の特徴を、GNU grep との比較をさみながら説明する。

### 正規表現からの動的なコード生成

本実装による一番の特徴として、正規表現から変換を行うことで得られる等価な DFA を、C/CbC/LLVM に変換しコンパイルすることで正規表現に応じた実行バイナリを生成することが挙げられる。またコンパイラ理論におけるさまざまな最適化が期待される。grep による検索のような、与えられるパターン (正規表現)

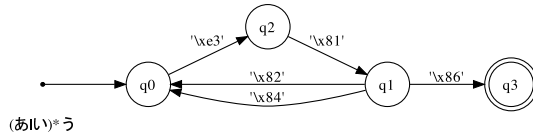


図 6 正規表現 “(a|i)\*u” に対応する DFA

に対してマッチ対象のテキストファイルが十分に大きい場合、正規表現のコンパイルにかかる時間はマッチングにかかる時間によって隠される。

GNU grep では、本実装と同様に DFA ベースのマッチングを行うが、DFA の各状態は構造体によって表され、状態遷移は各状態毎に保持している遷移先ポイントによる配列を、1Byte 単位でテーブルルックアップを行うことで実装されている。

### UTF-8 対応

本実装は、マルチバイト文字の代表的な符号化方式である UTF-8 に対応しており、正規表現の演算は 1Byte 単位ではなく、1 文字単位で適用される。マルチバイト文字を含む正規表現のサンプルとして、“(a|i)\*u” を DFA に変換した図 6 を載せる。図における '\x' に始まる文字は 16 進数表記で、'\x82' は 16 進数 82 を表す。

GNU grep 2.5.x では、マルチバイト文字に対応しているものの、プログラム内部で `libc mbrtowc(3)` を用いて固定サイズであるワイド文字に変換して処理を行っており、テストケース `complex-regex` ではそのオーバーヘッドが顕著に現れている。2010 年 3 月にリリースされた GNU grep 2.6 から、UTF-8 に対して本実装と同様に内部的に対応することで、`mbrtowc(3)` による変換コストが無くなっている。

### 柔軟な実装

本実験で実装した正規表現評価器は、Python によって実装されており、全体で 3000 行程度の軽量なプログラムとなっている。比較対象の GNU grep は、C 言語によって実装されており、プログラム全体は 10000 行を超える。

さらに、本実装から動的に生成されるコードも、コードセグメント/関数と switch を基準としたシンブ

ルな記述で高い可読性を持ちつつ、細かい最適化を GCC/LLVM の最適化技術を利用することで実行速度も非常に高速なものとなっていることが実験結果から分かる。

## 5 まとめと今後の課題

本実験では、正規表現をコードセグメント/関数による状態遷移を行うコードに変換し、GNU grep との比較を行い良好な結果を得た。また、コードセグメントと継続を基本とした言語である CbC での実装に適した例題の生成系として、今後の CbC コンパイラ開発に役立てていきたい。

今後の課題として、バックリファレンス等の拡張された正規表現の機能への対応、Boyer-Moore 法等の

フィルタの導入による高速化などにあたっていきたい。

## 参考文献

- [1] Cox, R : Regular Expression Matching Can Be Simple And Fast, 2007
- [2] Cox, R : Regular Expression Matching: the Virtual Machine Approach, 2009
- [3] Cox, R : Regular Expression Matching in the Wild, 2010
- [4] Hopcroft, J, E. Motowani, R. Ullman, J : オートマトン言語理論計算論 I (第二版), pp. 39-90.
- [5] Lattner, Chris. Adve, Vikram : The LLVM Compiler Framework and Infrastructure, 2004
- [6] Thompson, K : Regular Expression Search Algorithm, 1968
- [7] 与儀 健人 : 組込み向け言語 Continuation based C の GCC 上の実装, 2010