

1 章

正規表現マツチャ

ブライアン・カーニハン(Brian Kernighan)

正規表現は、テキスト中のパターンを指定する記法の1つです。そして実質的には、正規表現はパターンマッチのための専用言語を構成しています。正規表現にはさまざまな方言がありますが、それらの間で「パターン中のほとんどの文字はその文字と同じ文字にマッチする」という点は共通しています。ただし、いくつかのメタ文字と呼ばれる文字には特別な意味があります。例えば、*は0回以上の繰り返しを意味し、[...]は角括弧内に書かれた文字のどれか1つを意味します。

実際には、テキストエディタなどのプログラム中で探索を行うときは、単なる文字列探索が大部分です。そのため、正規表現もprintのような単なる文字列であることが多いのです。printという正規表現は、printf、sprintf、print paperなどの文字列にマッチします。UnixやWindowsのファイル名を指定するために使う、ワイルドカードと呼ばれるパターンの記法では、*は任意文字数の文字列にマッチします。したがって、*.cというパターンは.cで終わるようなファイル名すべてにマッチします。

正規表現には極めて多くのバリエーションがあり、ユーザが「これらの場合は同じだろう」と思うような場面でさえ、実は違っていることがあります。ジェフリー・フリードル(Jeffrey Friedl)の『Mastering Regular Expressions』(O'Reilly)[†]は、このさまざまな正規表現について網羅的に調査して書かれた本です。

正規表現は実は、スティーブン・クリーネ(Stephen Kleene)が1950年代半ばに、有限オートマトンのための記法として発明したものです。実際正規表現は、それが表現するもの(文字列)が何かという点では、有限オートマトンが表現するもの(文字列)と同等なのです。

正規表現が実際にプログラムの中で使われたのは、ケン・トンプソン(Ken Thompson)が1960年代半ばに、QEDテキストエディタのとあるバージョンに取り入れたのが最初です。トンプソンは1967年に、正規表現に基づいた効率のよいテキストのマッチング機構に関する特許を申請しました。その特許は1971年に承認されましたが、これはソフトウェア特許の最も最初のもの1つです(米国特許 3,568,156号、テキストマッチングアルゴリズム、1971年3月2日)。

正規表現はQEDからUnixのedエディタに移植され、続いてUnixの真髄とも言えるツールgrepに組み込まれました。grepはトンプソンがedを大改造して作り出したものです。これらの広く使われたツールのおかげで、正規表現は初期のUnixコミュニティに普及しました。

トンプソンの最初のマツチャ(matcher、あてはまりを調べるプログラム)は2つの独立したアイデアを組み

[†] 訳注：日本語版は次のとおりです。フリードル著、田和訳、『詳説 正規表現 第2版』オライリー・ジャパン、2003、ISBN-13: 978-4873111308。

込んだもので、非常に高速でした。アイデアの1つ目は、マッチングを実行するマシン命令をその場で生成するというもので、これによって解釈実行の速度ではなくマシン命令の速度での実行が可能になります。アイデアの2つ目は、各段階において可能なすべてのあてはまり方を保持し進んで行くというもので、これによって、1つのあてはまり方が合わなかった時に別のあてはまり方に切替えるための後戻り(バックトラック)が不要になります。

その後にはトンプソンが書いたedなどのテキストエディタでは、マッチャのコードとしてはもっと単純な、バックトラックを行うものが使われました。理論的に言えば、バックトラックを行う方法は遅いのですが、実際に利用者が使うパターンがバックトラックを必要とするものであることはめったにありません。ですから、edやgrepのアルゴリズムとコードはたいいていの用途には十分でした。

その後grepから派生してegrepやfgrepなどのマッチャが作られましたが、これらは正規表現の記法により豊かなバリエーションを追加し、またパターンの内容に関わらず高速に実行できるものでした。その後、より高機能な正規表現がC言語のライブラリとして普及し、さらにAwkやPerlなどのスクリプト言語の一部として組み込まれました。

プログラミング作法

1998年、ロブ・パイク(Rob Pike)と私は、『プログラミング作法』(原題『The Practice of Programming』、Addison-Wesley刊)という本を執筆していました[†]。その本の最終章は「記法」という題名で、よい記法の選択がよいプログラム、よいプログラミングにつながるという例を多数集めていました。記法の例としては、簡単なデータ指定(例えばprintfのようなもの)、表からのコード生成などがありました。

私たちはUnixを使ってきて、正規表現に基づくツールに30年近くも馴染んできていたので、記法の1つとして正規表現に関する議論を含めたいと思ったのは自然の成り行きでした。そうすると、実装も含めることが不可欠に思えました。そして私たちはツールに注力してきていましたから、シェルのワイルドカードのようなものよりは、grepで使われているような種類の正規表現を取り上げるのがベストに思えました。そうすれば、grep自身の設計についての議論にも触れられますから。

問題は、既存の正規表現パッケージはどれも大き過ぎたということでした。手元のマシン上で動いていたgrepは、500行以上(本のページにして10ページ分)のサイズでしたし、余計なものがくっつきすぎていました。オープンソースの正規表現パッケージは、どれも巨大なものばかりでした——というのは、それらは汎用性、柔軟性、速度などに力点を置いて開発されてきているからです。それでは教育用に適しているとは到底言えません。

そこで私はロブに、正規表現の基本的な考え方が読み取れる最小限の、ただしそれでいて有用でつまらないパターンが書けるようなパッケージを探そうと提案しました。コードが本の1ページに納まるような理想的だと思いました。

ロブは自分の部屋に入って行きました。今思い返してみると、1~2時間も経たないうちだったと思います。彼は30行のCのコードを携えて部屋から出て来ました。そのコードが、『プログラミング作法』の第9章に掲載されているものです。そのコードは、次の記法を使える正規表現マッチャを実装しています。

[†] 訳注：日本語版は次の通りです。カーニハン、パイク著、福崎訳、『プログラミング作法』、アスキー、2000。ISBN-13: 978-4756136497。

文字	意味
c	文字 c そのもの
.	任意の 1 文字
^	入力文字列の先頭にマッチ
\$	入力文字列の末尾にマッチ
*	直前の文字の 0 回以上の反復

これは十分有用なクラスの正規表現だと言えます。私が日頃正規表現を使っている経験に照らすと、正規表現を使う場合の優に 95% 以上はこのクラスに含まれます。多くの場面において、正しい問題を解こうとすることが、美しいプログラムを作り出す上での大きな一歩となるものです。ロブは、さまざまな選択肢の中から、非常に小さいが、重要であり、うまく定義された、拡張可能な一群の機能を選択したという点で称賛されるべきです。

ロブの実装それ自体も、コンパクトで、エレガントで、効率的で、実用性があるという点で、美しいプログラムの最上の例です。そのプログラムは、再帰をうまく使っているという点でも、私が見てきたうちで最上のものでしたし、C 言語のポインタの威力を示すものでもありました。本の執筆時点での私たちの力点は、プログラムを使いやすく（そしてたぶん書きやすく）する上で「よい記法」がいかに重要かを示すことだったのですが、このプログラムはアルゴリズム、データ構造、テスト、性能向上、そしてその他の重要な話題を実地に見せてくれる素晴らしい例題となりました。

実装

『プログラミング作法』では、正規表現マッチャは `grep` を真似した単独プログラムの一部として組み込まれていました。しかし、マッチャのコードはまわりのコードから容易に切り離せるようになっています。メインプログラムの部分はここでは特に取り上げません。その機能は、多くの Unix のツールと同様であり、標準入力または指定されたファイル群から入力を読み、正規表現にマッチする行を打ち出す、というだけです。マッチャのコードを以下に示します。

```
/* match: テキスト中の任意位置にある正規表現を探索 */
int match(char *regexp, char *text)
{
    if (regexp[0] == '^')
        return matchhere(regexp+1, text);
    do { /* 文字列が空の場合でも調べる必要あり */
        if (matchhere(regexp, text))
            return 1;
    } while (*text++ != '\0');
    return 0;
}
```

```

/* matchhere: テキストの先頭位置にある正規表現のマッチ */
int matchhere(char *regexp, char *text)
{
    if (regexp[0] == '\\0')
        return 1;
    if (regexp[1] == '*')
        return matchstar(regexp[0], regexp+2, text);
    if (regexp[0] == '$' && regexp[1] == '\\0')
        return *text == '\\0';
    if (*text != '\\0' && (regexp[0] == '.' || regexp[0] == *text))
        return matchhere(regexp+1, text+1);
    return 0;
}

/* matchstar: 「c*」型の正規表現をテキストの先頭位置からマッチ */
int matchstar(int c, char *regexp, char *text)
{
    do { /* 「*」は「0回以上の繰り返し」であることに注意 */
        if (matchhere(regexp, text))
            return 1;
    } while (*text != '\\0' && (*text++ == c || c == '.'));
    return 0;
}

```

議論

関数 `match(regexp, text)` は、テキスト中のどこかに正規表現がマッチするかどうかを調べ、マッチすれば1、しなければ0を返します。マッチするあてはまり方が複数ある場合は、最も左側でマッチするあてはまりのうち、最も短いものを返します。

`match` の基本的な動作は単純です。正規表現の最初の文字が `^` のときは(先頭位置でのマッチ)、あてはまりは必ず文字列の最初からでなければなりません。つまり、正規表現が `^xyz` の場合、`xyz` へのマッチが成功するのは、`xyz` が文字列の先頭にある場合に限られ、文字列の途中にある場合は成功しないのです。そのため、先頭に `^` がある場合は、`^` より後の正規表現を先頭位置から始まるあてはまりのみで検査します。

先頭が `^` でない場合は、正規表現は文字列のどの位置からマッチしても構いません。ですからその場合は、正規表現を文字列の各位置に対して順次あてはめて検査して行きます。つまり、正規表現が `xyz` の場合、文字列中のどこに `xyz` が含まれていてもよく、そのような `xyz` が複数ある場合はその最初のものがマッチすることになります。

文字列の各位置について順次調べて行くのに、`do-while` ループが使われていることに注意してください。C 言語では `do-while` ループは比較のまれにしか使われません。ですから、`while` でなくて `do-while` が使われている場合は常に、「なぜこのループの終了条件は、何か処理をしてしまう前にループの先頭で検査するのではなく、処理をしてしまった後ループの末尾で検査しているのか?」という疑問を持つべきなのです。でも、ここでは `do-while` ループを使うのが適切です。というのは、`*` を使うと正規表現が長さ0の文字列にあてはまる場

合があるわけですから、たとえ最初から終了条件(文字列が空文字列である)が成り立っていたとしても、ループの中に入って長さ0のあてはまりが可能かどうかを調べるべきなのです。

仕事の大部分は関数 `matchere(regex, text)` で処理されます。この関数は、渡された文字列のちょうど先頭位置に正規表現があてはまりかどうかを調べます。`matchere`は、まず正規表現の最初の要素が文字列の最初の文字にマッチするかどうかを調べます。もしマッチが失敗すれば、この位置でのあてはまりはないので、0を返します。しかしもしマッチが成功した場合は、正規表現の次の要素と、文字列の次の文字に進み、それらがマッチするかを調べる段階に進むことになります。それは具体的には、`matchere`を再帰的に呼び出すことで行います。

もちろん実際にやるべきことはもう少し複雑です。というのは、正規表現の特殊な機能を扱う必要がありますし、もちろん再帰を止める必要もあるからです。比較的簡単なのは正規表現のおしまいで来たかどうかを調べる場合、つまり `regex[0] == '\0'` かどうかを調べる場合で、これが成り立った場合には、ここまでの正規表現の各要素はすべてあてはめられているので、正規表現全体があてはめ終わったことになります。

正規表現が1文字の後ろに*がついた形のもの場合は、クロージャ(ある文字の反復)のマッチを調べる関数 `matchstar` が呼び出されます。`matchstar(c, regex, text)`は、文字 `c` の反復について、反復回数を0回、1回、…と順に増やしながら、それ以降の文字列への正規表現の残り部分のあてはまりを `matchere` を呼んで調べます。`matchere` が成功すれば、全体としてのあてはまりも成功です。そうでなく、反復回数を増やそうとして増やせない(文字列の終わりか、次の文字が `c` でない)場合は、マッチは失敗になります。

このアルゴリズムは反復回数を0から増やしていくため、「最短マッチ」のアルゴリズムとなります。最短マッチは、`grep`のように単純なパターンマッチで、マッチするかしないかだけをできるだけ速く判定したい場合には、適しています。一方、反復回数をできるだけ長い方から順に減らして行って調べる「最長マッチ」は、人間の直観により近く、テキストエディタなどでマッチした部分を別のものに置き換える場合には、ほぼ常に最短マッチより適しています。今日使われている正規表現ライブラリの大部分は、最短マッチと最長マッチの両方の機能を提供しています。そして、『プログラミング作法』でも、`matchstar`の最長マッチ版を掲載しています。これについては、後で説明します。

正規表現の最後が\$で終わっている場合には、文字列もその場所で終わりにしている場合のみマッチ成功となります。

```
if (regex[0] == '$' && regex[1] == '\0')
    return *text == '\0';
```

これらのどれでもない場合、つまり文字列がまだ終わりでない場合 (`*text != '\0'`)、もし文字列の先頭文字が正規表現の先頭文字と一致していればこの要素についてはOKで、正規表現の残り文字と文字列の残りのマッチを調べるために `matchere` を再帰的に呼び出します。この再帰呼び出しが、このアルゴリズム全体の鍵であり、コードがこれほど短く簡潔である理由でもあります。

もしここまですて来たどのマッチも成功しなければ、文字列のこの位置でのマッチは失敗なので、`matchere` は0を返します。

このコードはC言語のポインタを多用しています。再帰の各段階において、要素がマッチしたら、その後の再帰呼び出し前にポインタ演算 (`regex+1` や `text+1` など) を用いて、再帰呼び出しされる関数が正規表現の

次の要素、文字列の次の要素を受け取るようにします。再帰呼び出しの深さはパターンの長さ(通常の用途ではごく短い)を超えませんから、スタックがあふれる心配は無用です。

代替案

前掲のコードは非常にエレガントでうまく書かれていますが、完全無欠というわけではありません。どこを変えてみたいと思うのでしょうか？ 私なら、`matchhere`の中で\$を*より先に扱うように直すかもしれません。この場合、機能は何ら変わりませんが、少しだけ自然さが増すように感じますし、やさしい場合を先に扱い、難しい場合を後にする、というのがよいやり方です。

ただし、一般的に言えば、どの判定を先にやるかという順番は機能に影響します。例えば、`matchstar`の中の次の判定を見てみましょう。

```
} while (*text != '\0' && (*text++ == c || c == '.'));
```

次の文字が何であれ、その文字を処理して先に進まなければなりません。ですから、`text++`によるポインタの増加は必ず実行されなければならないのです。

さらに、このコードは終了条件についても細心の注意を払っています。一般に、マッチが成功したかどうかは、正規表現の最後まで来た時に文字列も終わりになったかどうかで分かります。両方が同時に終わりになったら、マッチは成功です。どちらかが先に終わったら、マッチは失敗です[†]。このことは、次のコードを見れば一層はっきりします。

```
if (regexp[0] == '$' && regexp[1] == '\0')
    return *text == '\0';
```

これら以外の場合にも同様に、ややこしい終了条件を扱う必要があります。

さて、`matchstar`の「最も左側でマッチするあてはまりのうち、最も長いもの」へのあてはめを行う版を示しましょう。それにはまず、文字`c`の最も長い繰り返しとなるような文字列の範囲を求めます。それから、その位置以降に正規表現の残りがあてはまるかどうかを、`matchhere`を呼び出して調べます。成功すればそこで1を返しますが、失敗した場合はそのたびに文字列の範囲を1つ短くして再度`matchhere`を呼んで調べます。範囲の長さが0になるまでやって、それでも駄目なら全体として失敗ですから0を返します。

```
/* matchstar: 「c*」型正規表現; 最も左側でマッチする最も長いもの */
int matchstar(int c, char *regexp, char *text)
{
    char *t;

    for (t = text; *t != '\0' && (*t == c || c == '.'); t++)
        ;
}
```

[†] 訳注：ここでは、最後に\$がある正規表現について議論しています。最後が\$でない場合は、正規表現が終わった時に文字列がまだ残っていてもマッチが成功します。ただしその場合でも、置き換えなどの処理を正しく行うためには、文字列の次の位置が正規表現にあてはまった範囲のきっかり次になっている必要があります。

```

do { /* 「*」は「0以上の繰り返し」であることを注意 */
    if (matchhere(regex, t))
        return 1;
} while (t-- > text);
return 0;
}

```

例えば、「(.*)」という正規表現を考えてみましょう。これは丸括弧で囲まれた任意の文字列とマッチします。ここで文字列として次のものを与えたとします。

```
for (t = text; *t != '\0' && (*t == c || c == '.'); t++)
```

最長マッチであれば、最初の「(」から最後の「)」までの範囲全体がマッチしますが、最短マッチだと最初の「)」までが範囲となってしまいます。(もちろん、最長マッチが常に好都合というわけではなく、2番目の「(」の位置で最長マッチを行かせた場合でも、最後の「)」までが範囲となってしまいますが。)

発展

『プログラミング作法』の目的は、よいプログラミングを教えることにありました。本を書いていた時点では、ロブと私はまだベル研究所に所属していたので、大学などの教室で本がどのように使われるだろうかということを経験していませんでした。ですから、本に含めた材料のいくつかが教室でうまく活用できたと分かって嬉しく思いました。私自身、(大学の教員になったので)2000年以降、このコードをプログラミングにおける重要なポイントを教える手段として使ってきました。

まずこのコードは、再帰がどのくらい有用であり、うまく使った場合に明快なコードができるかを明示しています。再帰の例題としては、クイックソートとか、階乗とか(!!)、木のたどりとかが多く使われますが、そんなありふれたものとは一線を画しています。

このコードはまた、性能実験のよい題材ともなります。このコードの性能は、Unixシステムに搭載されているgrepと大して変わりません。つまり、再帰を使ったからといってひどく遅くなるわけではなく、したがってこのコードをさらにチューニングする必要はないと分かるわけです。

さらに、このコードはよいアルゴリズムの重要性を浮かび上がらせるものともなっています。もしパターンにいくつかの*が含まれていると、素直な実装では多くのバックトラックが必要となり、場合によっては本当に遅くなってしまいます。

Unixの標準のgrepもこのバックトラックから来る特性を持っています。例えば、次のコマンドを試してみてください。

```
grep 'a.*a.*a.*a'
```

これをごく普通のマシンで4MBのサイズのテキストファイルに対して実行してみたところ、20秒ほどかかりました。

正規表現から非決定性有限オートマトンを生成し、それを決定性有限オートマトンに変換して実行すると

いう実装であれば(egrepはそうになっています)、このような意地悪な場合でもずっとよい性能を発揮します。上のデータをegrepで処理してみると、所要時間は0.1秒未満であり、しかも実行時間は一般にパターンが何であっても変わりませんでした。

正規表現の記法をさまざまに拡張するという形で、いろいろな課題を提供できます。いくつか例を挙げましょう。

1. 他のメタ文字、例えば+(1つ前の文字の1回以上の反復)、?(1つ前の文字が0個または1個存在)などを追加しなさい。メタ文字を普通の文字として扱わせる何らかの方法(例えば\$という文字を指定したければ\\$のように指定するなど)を追加しなさい。
2. 正規表現の処理を、コンパイルフェーズと実行フェーズに分離しなさい。つまり、コンパイルフェーズで正規表現を内部的なデータ構造に変換し、それによってマッチングのコードがより簡潔になったり、マッチングの実行速度が向上するようにするわけです。このようなフェーズの分離は、ここで示したような単純な正規表現のクラスに対しては必要ないでしょうが、正規表現がより多くの機能を持ち、1つの正規表現を多数の行に対して適用するgrepのようなアプリケーションでは意味を持ちます。
3. 文字クラスと呼ばれる機能を追加しなさい。これはgrepの記法では[abc]や[0-9]のように書き、それぞれ「a、b、cのどれか」や「数字」のように「指定した文字のどれか」にあてはまるものです。これを実装するには複数のやり方がありますが、一番自然なのは正規表現の文字の代わりに次のような構造体を使うようにするというものです。

```
typedef struct RE {
    int    type; /* CHAR, STAR 等の種別 */
    int    ch; /* 文字そのもの */
    char  *ccl; /* 文字クラスの場合に使用 */
    int    nccl; /* 文字クラスで ^(~以外)指定なら 1 */
} RE;
```

これを使う場合、正規表現はRE型の配列に格納し、元のコードで正規表現を表すchar*を渡していたところはRE*を渡すように直します。この場合、厳密に言えばコンパイルと実行を分離しなくても実装はできますが、やってみると分離した方がずっと実装しやすいと分かるはず。最初にこのようなデータ構造にコンパイルしてから実行した方がいいよ、というアドバイスを受け入れた学生はほぼ例外なく、込み入ったパターンのデータ構造を解釈しながら処理しようとする学生よりも、よい成果を上げています。

明快で曖昧さのない文字クラスの仕様を書くというのは難しい問題であり、それを完璧に実装するというのは一層難しい問題です。その際には、たくさんの退屈で教育的でないコードを書く必要があります。ですから私は、最近では通常、この課題をもっと単純化して、Perlのような短縮記法(例えば\dで数字、\Dで数字以外を表す)を実装すればいいということにしています。この方が、角括弧で囲んだ中にa-zのような文字範囲を指定できるという元の記法よりだいぶ楽です。

4. 不透明な型を使って、RE構造体や実装の細部を隠すようにしなさい。これはC言語を使ってオブジェクト指向プログラミングの利点を学ぶよいやり方です(C言語の機能ではどのみち、オブジェクト指向に関してこれより大して進んだことはできません)。実際にはこの課題では、オブジェクト指向言語が持つ構文上のサポートの代替として、`RE_new()`とか`RE_match()`など決まったプレフィックスを持つ関数群を定義し、それらとそれらが扱うデータ構造を組にして正規表現クラスを作ることになります。
5. 正規表現の機能を変更して、各種のシェルが持つワイルドカード機能のようなものにしなさい。マッチは特に指定しなくても文字列の先頭から末尾まで全体にわたるあてはまりとし、*は任意文字列にあてはまり、?は任意の1文字にあてはまるようにします。実装方法としては、既に作ったプログラムを直すようにしてもいいですし、入力を既に作ったプログラム用の正規表現に変換して実行するというのもよいです。
6. コードをJava言語に変換しなさい。元のコードはC言語のポインタを実にうまく使っていますから、別の言語でそれをどのように代替するかを考えるのはよい練習になります。Java版では、`String.charAt`を使う(ポインタの代わりに文字列の何文字目かを扱う)か、または`String.substring`を使う(文字列の残りを取り出して別の文字列とするので、C言語のポインタ版にやや近い)かのどちらかでしょう。どちらの版もCのコードほど明快にはなりませんし、コンパクトでもありません。性能はこの課題には含まれませんが、Java版がC版のおよそ6~7倍の実行時間が掛かるというのは興味深い点です。
7. ここで扱った正規表現をJavaのクラス`Pattern`とクラス`Matcher`で実行するように変換する包囲クラスを書きなさい。`Pattern`と`Matcher`ではコンパイルと実行を全く違う形で分けています。この課題は既存のクラスや関数の上に別の顔を乗せるもので、アダプタ(Adapter)パターンやファサード(Facade)パターンのよい実例となります。

私はこのコードを、テスト手法を探求するためにもたくさん使って来ました。正規表現は多様な機能を持つため、テストが簡単すぎてつまらないということがありませんが、一方で機械的なテストのために十分な量のデータをさっと書き下すことができる程度には簡単です。上に挙げたような機能拡張の課題の一環として、私は学生に十分な数のテストをコンパクトな言語(これも記法の例ですね)で記述して用意し、自分のコードをそのテストデータで検証するように求めています。そしてもちろん、自分のコードだけでなく他の学生のコードの検証にもそのテストデータを使います。

結論

ロブが最初にこのコードを書いたとき、私はそのコンパクトさ、エレガントさに驚愕しました。このコードは私が想像していたよりも、ずっと小さく、ずっと強力だったのです。後知恵で考えれば、このコードがこれほど小さくて済む理由をいくつも思いつきます。

まず、正規表現の機能がうまく選ばれていて、とても有用で、かつ実装に対して多くの洞察が得られるようになっています。それでいて、無用な飾りは除かれています。例えば、先頭や末尾にマッチを固定するための`^`と`$`の機能はたった3、4行のコードで実装されていますが、これは一般の場合を均質的に実装する前に特殊な場合を明快に処理するよい例となっています。*によりクロージャを指定する機能も欠かせません。

というのは、正規表現における基本的な記法ですし、長さに変化する部分を指定する唯一の方法となっているからです。これに対し、さらに+や?を追加したとしても、より多くの洞察が得られるということはないでしょう。ですから、これらは演習問題ということにしているのです。

第二に、再帰がうまく使われている点が挙げられます。この基本的なプログラミング技法は、通常のループを使ったものと比べてほぼ常に、より小さく、より明快で、よりエレガントなコードを生み出します。ここでもそれは例外ではありません。正規表現の先頭から1要素、文字列から1文字を取り除いて、残り部分は再帰呼び出しで処理するというのは、よく使われる階乗の例題や文字列の長さを求める例題と似てはいますが、ずっと興味深く有用な形となっています。

第三に、このコードはプログラミング言語の機能を実にうまく使っています。もちろん、ポインタを下手に使ってしまうことの弊害はよく知られていますが、ここではポインタは、個々の文字を取り出したり次の文字に進むことを自然な形で記述するコンパクトな表現を作り出すのに使われています。配列の添字とか部分文字列の取り出しでも同じ効果は実現できますが、このコードではポインタを使い、それをC言語の自動インクリメント機能や真偽値の自動変換機能と組み合わせることで、よりうまい扱いができています。

そういうわけで私は、このコードほど少ない行数で多くのことを実現し、これほど多くの洞察とアイデアの展開をもたらしてくれるコードを、他に知りません。