# Implementations of Regular Expressions

Craig E. Ward
CMSI 583 Theory of Computation
Loyola Marymount University

April 28, 2003

**Abstract**

This paper is a survey of some of the software implementations of Regular Expressions. Tools developed primarily for education or research are reviewed as well as several of the more popular engines in production use. The influence of POSIX standards will be addressed. It will be shown that many of the tools diverge from theory in order to bring greater utility and practicality to problem solving.

## 1 Introduction

*Regular Expressions* are a powerful tool in use in many software systems for matching patterns in large amounts of text. They are based on an algebra of *regular sets* first described by Stephen Kleene in the 1950s.[1] The first published description of an implementation, the *grep* utility, was by Ken Thompson in 1968.[2] While the algebra of regular sets provides the foundation for all implementations, current tools go beyond the theory to fill practical needs in production environments.

Tools primarily used for educational or research purposes tend to match theory much more closely than tools intended for use in business or other non-academic settings. Two tools, *JFLAP* and *Grail+*, used for teaching will be reviewed. Following these, the

[1] Jeffrey E. F. Friedl, *Mastering Regular Expressions* (O'Reilly & Associates, 1997) p. 60.
[2] *ibid*

development of tools that built on the experience of *grep* will be examined and will include a description of what these tools did that diverged from the theory of regular sets. The paper will conclude with a discussion of what all of this may mean to the relationship between theory and practice.

## 2 Teaching Tools

Teaching tools tend to process definitions of automata. Their roles are more of illustration and experimentation  than production.

### 2.1 *Grail*+

The first tool to examine is the *Grail*+ system from the University of Western Ontario, London, Ontario, Canada.

*Grail*+ is a suite of command line programs and a library of C++ classes and functions. The class library can be linked with other programs. The command line programs are implemented as UNIX-style filters, *i.e.*, they may be used as standard UNIX filters in a shell environment.  Table 1 contains a partial list of the filters in version 2.5 of *Grail*.[3]

| **Filter** | **Description** |
|---|---|
| fmcment | complement a machine |
| fmcomp | complete a machine |
| fmcat | catenate two machines |
| fmcross | cross product of two machines |
| fmdeterm | make a machine deterministic |

[3] For a complete list, see Darrel Raymond and Derick Wood, "*Grail*: Engineering Automata in C++" 1996, p. 11.

| **Filter** | **Description** |
|---|---|
| fmstar | star of a machine |
| iscomp | test a machine for completeness |
| isdeterm | test a machine for determinism |
| isempty | test for equivalence to empty set |
| restar | Kleene star of a regular expression |
| retofm | convert a regular expression to a machine |
| retofl | convert a regular expression to a finite language |

Table 1: Some *Grail* filters

The *Grail* system introduces some special characters to facilitate writing regular expressions and defining automata. Two double-quote characters (“”) denote an empty string and two braces ({}) denote the empty set. These are concessions to the ASCII character set. The following are examples of regular expressions in *Grail*:[4]

- a+b
- ((a+bcde*)+c)*
- {}
- “”a

Unfortunately, the Grail system is not supported on any of the systems available to the author. Porting to other systems should be feasible, but would require more time than is available.

[4] Darrell Raymond and Derick Wood, “*Grail*: Engineering Automata in C++” 1996, p. 4.

## 2.2 *JFLAP*

*JFLAP* is a graphical tool from Duke University for teaching Formal Languages and Automata Theory. The system is Java-based and should be usable on any system with a Java 2™ JVM. (The screen shots in this report are from JFLAP 4.0 Beta , released March 3, 2003, on a system running MacOS X 10.2.5.)

*JFLAP* was created to give students a "hands-on" experience with NFA and DFA machines.[5] The beta version review here includes regular expressions and other aspects of computing theory. Figure 1 shows the initial *JFLAP* screen.

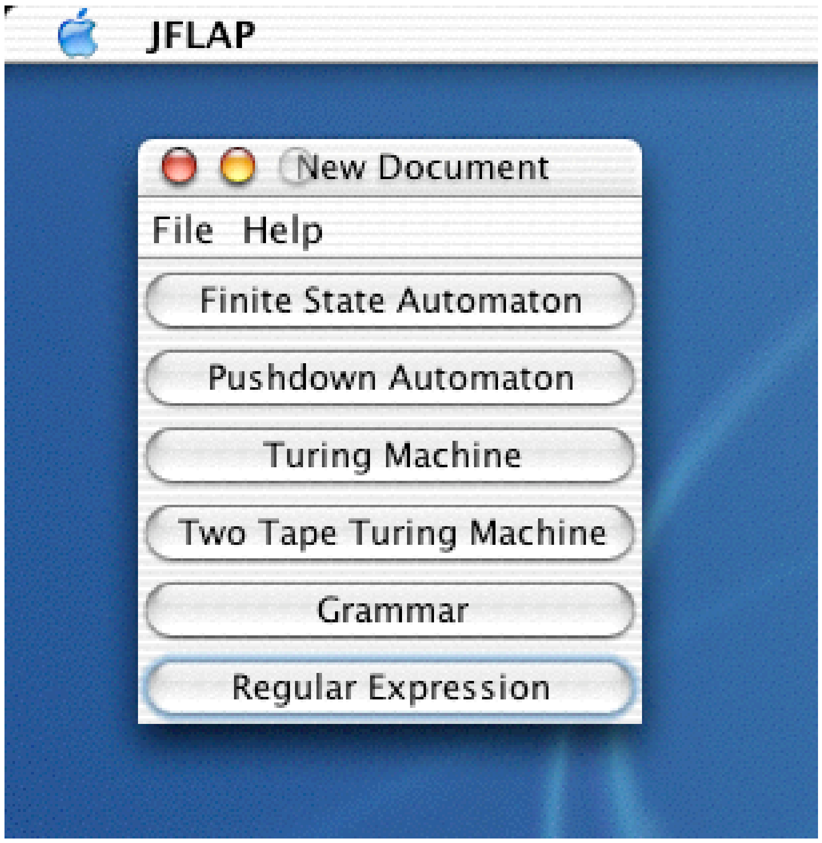

Figure 1: Opening screen of *JFLAP*.

[5] Susan H. Rodger and Eric Gramond, "JFLAP: An Aid to Studying Theorems in Automata Theory" ACM SIGCSE Bulletin Volume 30 , Issue 3 (September 1998), 302.

Once the program is opened, the user can select any of the supported types or open a previously saved file. The system is able to run NFAs and DFAs as well as convert between them and regular expressions. It does not run the regular expressions directly, but a student can convert a regular expression into an NFA and either run it or convert it further into a DFA. (Converting these machines back into regular expressions does not create the same regular expression text, but it is the same language.) Figure 2 illustrates what the screen looks while running a DFA in one of four ways.

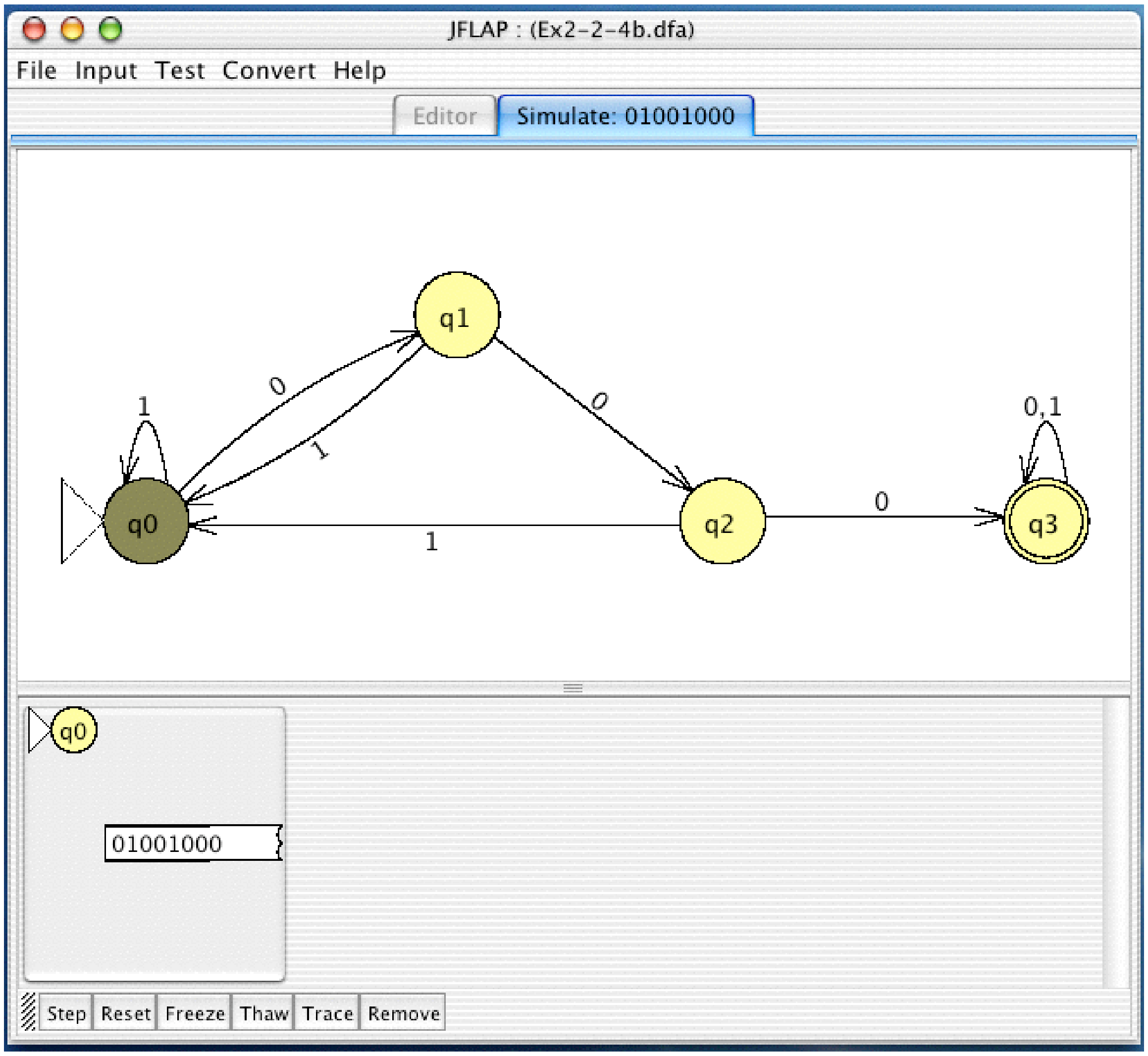

Figure 2: Running a DFA in *JFLAP*.

JFLAP uses the standard syntax for entering regular expressions, parentheses group symbols, "+" for union, and "*" for closure. The dialog box itself is simple and is not displayed here.

## 3 Production Tools

The last set of tools to be examined are all in-use in one way or another in production environments. Regular expressions,  referred to as *regexs*, aid software engineers, system administrators, web application designers, and others to solve a variety of tasks.

The first tool to implement regexs was a text editor called *ed*. It had a function for displaying lines of text in a file that matched a pattern: `g/Regular Expression/p`, read "Global Regular Expression Print". The routine proved so useful that it became a standalone utility: *grep*.[6]

The *grep* utility lead to the creation of many similar utilities in an *ad hoc* manner. Among the most used tools are *egrep* (modern *extended grep*), *awk*, GNU Emacs (a Lisp processor), Tcl, Python, *flex*, Java, JavaScript, and perhaps the most important one of all, Perl. The full history of this development is beyond the scope of this paper. It should be noted, however, the situation has improved.

### 3.1 *Regex Shorthand Symbols*

The standard descriptions of regular expressions use a small set of special symbols or metacharacters. Parentheses are used to group other regular expression characters. The plus sign, "+" is used to show the union of two regular expressions. A star, "*" is used to indicate the Kleene closure of the set. Production implementations add to these and change them.  Metacharacters are used to simplify matching ranges of characters,

[6] Friedl, p. 61.

counting characters, match by position, and other enhancements. Table 2 shows a selection of the most common metacharacters. (Particular tools may require different symbols, but the usage will be mostly the same.)

| **Character** | **Feature** | **Usage** |
|---|---|---|
| . | Dot | Match any one character |
| […] | Character class | Match any character in class |
| [^…] | Negated character class | Match any character **not** listed |
| \char | Escaped character | When char is a metacharacter, or not otherwise special, match as literal. Also used in some tools for machine-dependant characters (backspace, newline, *etc*.). |
| ? | Question | Match 0 or one of preceding. |
| * | Star | Mach 0 or more of preceding. |
| + | Plus | Match one or more of preceding. |
| {min, max} | Specified range | *Min* matches required, *max* allowed. |
| ^ | Caret | Matches at the start of string (line). |
| $ | Dollar | Matches at end of string (line). |
| \< | Word boundary | Matches position at start of a word. |
| \> | Word boundary | Matches position at end of a word. |
| \| | Alternation | Matches either expression it separates. |
| (…) | Parentheses | Limits scope of an expression, provides grouping for quantifiers (?, *, +, {min,max}), and "captures" for back references. |
| \1, \2, … | Back reference | Matches text previously matched within first, second, *etc*., of set of parentheses. Some tools use $1, $2, … . |

Table 2: Common Metacharacters[7]

These metacharacters provide a short hand that allows for concise, powerful patterns for regular expressions to match. They also open the way to confusion and unexpected results to the unwary. The meaning of metacharacter can change or it could stop being a metacharacter depending on where in the regular expression it appears. For example, caret is an anchor metacharacter outside of a character class and a negation metacharacter inside of a character class.

The POSIX standard for regular expressions added named character classes. Examples include "[:alnum:]" for all alphabetic and numeric characters and "[:punct:]"

[7] Adapted from Friedl, p. 29.

for punctuation characters.[8] What these mean can be changed by a locale setting in the runtime environment.[9]

## 3.2 *Engine Types*

There are two basic regex engine types. Their names reveal the theoretical foundations of each. These are *DFA* and *NFA*. The addition of the POSIX standard effectively creates a third type, the *POSIX-NFA* engine.

### 3.2.1 DFA Engines

The DFA class of engines are modeled by Deterministic Finite Automata. These engines are noted for being fast and efficient, and, according to Friedl, boring.[10] Friedl describes the speed as coming from the ability of a DFA engine to simultaneously ("almost magically"[11]) keep track of all possible matches. Although he does not explicitly say so, this is likely a result of the deterministic characteristic of DFAs. Each state has one new state to move to for each input symbol. Paths that will not generate a match lead to dead ends.

### 3.2.2 NFA Engines

The other class of regex engines is modeled by Nondeterministic Finite Automata. The POSIX standard for regular expressions further divides this class into Traditional- and POSIX-NFAs.

[8] Friedl, p. 80.
[9] Friedl, p. 65.
[10] Friedl, p. 101.
[11] *ibid*.

The POSIX standard requires that regular expression engines match the “longest of the leftmost” pattern.[12] For DFA engines, this is what they do naturally. NFA engines require a change .

The traditional NFA engine will stop with the first possible match. This might not be the only or the longest match. The POSIX standard requires that regex engines not stop with the first possible match. This has made NFA engines more complex and slower, but with a benefit of greater flexibility.

Both varieties of engines must keep track of where they have been in order to backtrack from a failed match. At each point where an NFA has a choice of transitions, it must choose one and remember the others. If an attempted match fails, the backtracking rewinds the machine and chooses another attempt.[13]

As in the case of DFA engines, Friedl does not explicitly state where this behavior comes from. It does, however, seem a natural consequence of the nondeterministic nature of NFAs. An implementation must be able to manage the *lambda* moves as well as the multiple possible choices of the transition function.

Table 3 lists several popular implementations and the automata model used.

| **Tool** | **Version** | **Engine Type** |
|---|---|---|
| *awk* | Generic | DFA |
| GNU *awk* | Recent | DFA with some NFA |
| *egrep* | Generic | DFA |
| MKS *egrep* | Generic | POSIX NFA |
| GNU Emacs | All | Traditional NFA (POSIX NFA available) |
| Perl | All | Traditional NFA |
| Tcl | All | Traditional NFA |
| JavaScript | 1.5 | NFA, subset of Perl 5 regular expressions[14] |
| Java | 1.4 | NFA, subset of Perl 5 regular expressions[15] |

Table 3: Some popular tools their regex engine types. [16]

[12] Friedl, p. 117.
[13] Friedl, p. 103
[14] David Flanagan, *JavaScript: The Definitive Guide 4th Edition* (O’Reilly & Associates: 2002) p. 147.
[15] Ron Hitchens, Java NIO (O’Reilly & Associates: 2002) p. 153.

## 3.3 *Beyond Regular Expressions*

What is it about these implementations that takes them beyond traditional theory? The concepts of backtracking for grouping, greedy metacharacters, and alternation. Friedl makes the case the term *regular expression* no longer fits, that these expressions are actually irregular.[17] Each of these features will be examined.

### 3.3.1 Backtracking for Grouping

Many regex tools have the ability to select sub-matches from a match and reuse that text later in the evaluation of the regex. This provides a powerful utility. But with that power comes danger. If the regex is poorly crafted, it can cause a significant performance hit. Friedl uses as an example a small Perl script that searches a text file for repeat words.[18] The regex must remember what it has already matched in order to know if the next word is the same.

### 3.3.2 Greedy Metacharacters

As a general rule, metacharacters attempt to match the longest possible string. This can cause later parts of the regex to fail to match. Engines get around this by implementing a rule that says that metacharacters must "give back" characters if letting them go would allow a later part of the regex to succeed.[19]

[16] Adapted from Friedl, p. 90 with additions as indicated.
[17] Friedl, p. 104.
[18] Friedl, p. 31.
[19] Friedl, p. 94.

### 3.3.3 Alternation

Complex regexs can be created using the alternation symbol, "|", grouped with parentheses. This simplifies the coding and amplifies the possibilities for inefficient evaluations.

As an example, the regex `^(Subject:|From:)` will find the Subject and From fields in a file of email. While this example is straight forward, imagine a context where one of the possible matches would occurred much more frequently than the others. If that sub expression were the last in the sequence, a lot of resources would be wasted checking for the less common sub expressions.

This is a significant issue for NFA engines. The "longest-leftmost" match rule of the POSIX standard requires alternation be greedy and check all the possibilities even after an initial match is found. Crafting an efficient NFA regex is a hard earned skill.

## 4 Conclusions

This paper has only scratched the surface of regular expressions. The examples of metacharacters do not necessarily match any particular tool, but do indicate what types of metacharacters to look for. If anyone were forced to learn just one, Perl looks like the clear winner. (The most recent additions to the regex world, JavaScript and Java, both use Perl as their model.)

The educational tools review, especially *JFLAP*, should have a prominent role in teaching the fundamentals. Clearly the fundamentals are important if only to aid the regex developer to better code efficient NFA expressions.

It seems equally clear that practical implementations have moved beyond basic theory. The simple fact that production engines are used for pattern matching and not for explicitly evaluating whether a particular string of symbols is in the language of an automaton is the main force behind changing the implementations to diverge, if ever so slightly, from established theory. The why of this is best summed up by a quote from Ian Graham in a presentation he gave at Carnegie Mellon University: "It's much easier to hack that to make a good proof."[20]

Of course, that doesn't mean that no one should try.

[20] Ian Graham, "Kleene Would Be Shocked: Redrawing the Link between Theory and Modern Regex Engines" REU Summer 2002 Symposia, Aladdin Center, Carnegie Mellon University, Slide 37.

## 5 Bibliography

Flanagan, David. *JavaScript: The Definitive Guide, 4th Edition*. Sebastopol: O'Reilly & Associates, Inc. 2002.

Friedl, Jeffrey E. F. *Mastering Regular Expressions*. Sebastopol: O'Reilly & Associates, Inc. 1997.

Graham, Ian. "Kleene Would Be Shocked: Redrawing the Link between Theory and Modern Regex Engines." Presentation at REU Summer 2002 Symposia, Aladdin Center, Carnegie Mellon University, 2002. URL: http://www.aladdin.cs.cmu.edu/reu/.

Ron Hitchens. *Java NIO*. Sebastopol: O'Reilly & Associates, Inc. 2002.

Hopcroft, John E., Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation, 2nd Edition*. New York: Addison Wesley. 2001.

Raymond, Darrell, and Derick Wood, "*Grail*: Engineering Automata in C++." University of Western Ontario. 1996. URL: http://www.csd.uwo.ca/research/grail/grail.html.

Rodger, Susan H. and Eric Gramond, "JFLAP: An Aid to Studying Theorems in Automata Theory" ACM SIGCSE Bulletin Volume 30 , Issue 3 (September 1998), 302. URL: http://www.cs.duke.edu/~rodger/tools/jflaptmp/.