Regular expressions: Why every modern computer scientist should know and love them

Laurette Pretorius and Biffie Viljoen

June 2001

Abstract

The classical approach to teaching theoretical computer science focuses on the mathematical foundations for studying formal systems and algorithms, and on the concepts and languages for capturing the essence of both the algorithmic and descriptive aspects of systems, from specification to efficient implementation.

While this approach remains valid and essential, we need to emphasise new ways in which the theory and mathematical foundations of computing can be taught and used to deal with our rapidly changing discipline.

Regular expressions are usually taught as part of a theory of computation course, which more often than not enjoys the status of "not essential for practically-minded students" or "not compulsory" in a computer science curriculum.

In this paper we explain what regular expressions are. We briefly refer to their role in the foundations of computer science. We then spend some time demonstrating their importance in practical applications by giving two examples, viz. Linux and XML.

1 Introduction

The classical approach to teaching theoretical computer science focuses on the mathematical foundations for studying formal systems and algorithms, and on the concepts and languages for capturing the essence of both the algorithmic and descriptive aspects of systems, from specification to efficient implementation.

While this approach remains valid and essential, we need to emphasise new ways in which the theory and mathematical foundations of computing can be taught and used to deal with our rapidly changing discipline.

Our purpose with this talk is clear: After today you should never again doubt the usefulness of regular expressions, and you should never again underestimate the contribution of theoretical computer science to even the most practical parts of our exciting discipline! After today we hope that you will encourage your students accordingly.

But, let us start with a question or two:

(1) **Question:** Is there anyone here today who has not, or does not, or will not ever teach or use at least one of the following modern computing packages, environments, platforms, frameworks, utilities or applications:

- Unix, Linux, egrep, grep
- Perl, Python, Tcl
- JavaScript, VBScript
- XML, XSL
- Delphi
- Emacs, Expect, vi
- etc. and the list goes on!

**Answer:** Since any modern computer scientist should at least take note of these contemporary environments, we anticipate the answer 'No'.
(2) **Question:** Name at least one thing that all these have in common.

**Answer:** There may be many, but there is at least one, and yes, you have guessed it - they all make use of regular expressions!

One could even claim that an expert user who wants to consistently produce efficient, elegant and correct solutions to real problems using these mentioned environments should be able to exploit the regular expression "engine" of the particular application. We hope that by the end of this talk you will all agree with us!

2 The fifties

Regular expressions were defined by Kleene in 1956. Well-known names like McCulloch and Pitts, McNaughton, Chomsky, Rabin and Scott, among many others, made significant mathematical contributions to this field in the 1950s. The mathematical result that is of importance to us today is the equivalence theorem we shall soon state.

3 The seventies

By 1970 the theory of formal languages and automata was well established. Its application in text manipulation systems including lexical analyzers for compilers, text editors, and file manipulation systems was also already reported in the literature, with Knuth, Morris and Pratt’s algorithm for fast pattern matching in strings, based on finite automata, a key contribution.

In 1974 I was introduced to this subject and the Chomsky hierarchy of grammars and languages through the well-known book, by now a classic, *The Theory of Parsing, Translation and Compiling* by von Aho and Ullman, published in 1972. The purpose was to provide a formal basis for computer programming languages, compiler construction, and computability via Turing machines. I was then an Honours student in Computer Science at the University of Stellenbosch. The following definition and theorem are taken from this book [11]:

**Definition:**

Regular expressions over the finite alphabet $\Sigma$ and the regular sets they denote are defined recursively as follows:

1. $\phi$ is a regular expression denoting the regular set $\emptyset$, the empty set, also denoted by $\{}$.
2. $e$, the empty string, is a regular expression denoting the regular set $\{e\}$.
3. $a$ in $\Sigma$ is a regular expression denoting the regular set $\{a\}$.
4. If $p$ and $q$ are regular expressions denoting regular sets $P$ and $Q$, respectively, then
   a. $(p + q)$ is a regular expression denoting $P \cup Q$.
   b. $(pq)$ is a regular expression denoting $PQ$.
   c. $(p)^*$ is a regular expression denoting $P^*$.
5. Nothing else is a regular expression.

One thing to understand about this definition is that it has a very simple and useful structure. First it states a small number of basic building blocks, then it gives a small number of operations that may be performed on already constructed "things" of a particular kind to build more complex "things" of the same kind, obviously starting off with the basic building blocks. So, once you know what the basic building blocks for regular expressions are ($\phi$, $e$, and the elements of $\Sigma$) and what the operations are that you are allowed to perform on them (choice, concatenation, and Kleene star) in order to produce more complex regular expressions you know the basics of regular expressions.
Table 1: The Chomsky hierarchy.

The equivalence theorem:
The following statements are equivalent:

(1) $L$ is a regular set.
(2) $L$ is a right-linear language.
(3) $L$ is a finite automaton language.
(4) $L$ is a nondeterministic finite automaton language.
(5) $L$ is denoted by a regular expression.

Not all the notions mentioned in the theorem are of importance to us today, but we state the full theorem for completeness. We would like to focus your attention on the equivalence between (3) and (5) since it will give us a clue as why regular expressions are of such great practical use. Remember that another word for finite automaton is finite state machine.

4 The nineties and beyond

Since the early 1990s Bifflie and I have been involved in the teaching of formal languages and automata theory to second and third year students, using Cohen’s very readable textbook, *Introduction to Computer Theory*, published for the first time in 1991. The approach is similar, culminating once again in the Chomsky hierarchy of formal languages (see table 1), the notion of what a computable function is, and the so-called Church-Turing thesis. This is how we introduce regular expressions to you in the next section. For the interested reader we recommend the excellent survey paper [8].

5 What are regular expressions?

Before we give a formal definition, let us take an informal look at such expressions, following the style and notation of [3]. These expressions consist of letters from some alphabet $\Sigma$, and the empty string (or word) $\Lambda$, together with Kleene stars, plus signs, and parentheses, the so-called metacharacters.

Suppose, for example, our alphabet is $\{a, b\}$.

- First we introduce the use of the *Kleene star operator* $\ast$. To apply it to a single letter $a$ (say), we write the letter in boldface and then attach the Kleene star: $a^\ast$. This simple expression will indicate some sequence of $a$’s, even none, i.e. zero a’s, or one $a$, or two $a$’s, or ... Thus

  $$a^\ast = \Lambda, a, aa,aaa,aaaa, \ldots$$

  We should think of $a^\ast$ as any string of $a$’s – the length of the string can be chosen. The associated (regular) language is the (infinite) set $\{\Lambda,a,aa,aaa,aaaa,\ldots\}$.

- Next we consider *concatenation*. The expression $ab^\ast$, consisting of an expression $a$, concatenated with $b^\ast$, indicates all strings over the alphabet consisting of a single letter $a$ followed by any number of $b$’s, i.e.

  $$ab^\ast = a,ab,abb,abbb,abbbbb,\ldots$$

The Kleene star can also be applied to a concatenation of letters in parentheses. For example,
\((\text{bab})^* = \Lambda, \text{bab}, \text{babab}, \text{bababab}, \ldots\)

- Finally, we introduce the use of the \textit{choice operator} \(+\). If \(x\) and \(y\) are strings of letters from some alphabet, then \(x + y\) stands for either \(x\) or \(y\). Thus \(x + y\) again offers a choice. While the Kleene star operator requires us to choose the \textit{number of repetitions} of an expression, the \(+\) operator implies a choice \textit{between two different expressions}. For example,

\[
\text{aba} + \Lambda + \text{bbb} + b = \text{aba}, \Lambda, \text{bbb}, b.
\]

By combining the use of \(*\) and \(+\), we can construct an infinite number of regular expressions for any given alphabet. Furthermore, any regular expression involving \(*\) (except the expression \(\Lambda^*\)) results in an infinite number of possible strings.

Before we give the formal definition, let us look at an example of a regular expression where both \(*\) and \(+\) are used:

\[(a + b)^* (aa + bb) (a + b)^* .\]

This expression indicates that we can choose \(a\)'s and \(b\)'s any number of times, then either \(aa\) or \(bb\), then \(a\)'s or \(b\)'s any number of times. This means that all strings over the alphabet \(\{a, b\}\) which contain at least one double letter can be found from this expression. Examples are \(ababa\), \(bbbbb\), \(abbaab\).

Let us now give the formal definition.

\textbf{Definition:}

Given an alphabet \(\Sigma\), regular expressions are formed as follows:

- **Rule 1:** Every letter of \(\Sigma\) can be made into a regular expression by writing it in boldface. \(\Lambda\) itself is a regular expression.
- **Rule 2:** If \(r_1\) and \(r_2\) are regular expressions, then so are
  - (i) \((r_1)\),
  - (ii) \(r_1 r_2\),
  - (iii) \(r_1 + r_2\),
  - (iv) \((r_1)^*\), often abbreviated to \(r_1^*\).
- **Rule 3:** Nothing else is a regular expression.

We say that a string is \textit{generated by a regular expression} if it can be obtained from the expression by making appropriate choices at \(*\) and at \(+\). The word \(bbbaaaabab\), for example, is generated by the regular expression

\[(a + b)^*(a(a + b)^*a(a + b)^*).\]

The language that is generated or defined by a given regular expression, is the set consisting of all the strings that can be made up from the expression by making all possible choices when \(*\) or \(+\) occurs. Thus the language defined by the above expression is the (infinite) language consisting of all possible strings of \(a\)'s and \(b\)'s containing at least two \(a\)'s.

Let us now turn to two other ways in which to define a language.

(1) As you know, a \textit{finite automaton} or \textit{FA} is a collection of three things:

  - (i) a finite set \(S\) of states, one of which is the start state and some are final states,
  - (ii) an alphabet \(\Sigma\) of possible input letters, and
  - (iii) a function which maps \(S \times \Sigma\) into \(S\).

Every FA \textit{defines} or \textit{accepts} some language, namely the set of all the strings which are accepted by the machine, i.e. all strings which will end up in a final state when they are run on the FA. An FA is a deterministic machine.
Another kind of language-recognizing machine is a transition graph, or TG. A TG differs from an FA in two ways: input strings are formed (i.e. input is not necessarily a single letter at a time), and we have a relation (not a function) from $S \times \Sigma^*$ into $S$. This means that a TG is nondeterministic. A string is accepted if there is at least one way of reaching a final state when the string is run on the machine. The language defined or accepted by a TG is the set of all strings which are accepted by the machine.

We conclude this section with the equivalence theorem, also called Kleene’s Theorem, and a last example.

**The equivalence theorem:**
Any language that can be defined by a regular expression, or a finite automaton, or a transition graph, can be defined by all three methods.

As a final example, let us consider the language with alphabet \( \{a, b\} \) consisting of all strings with length at least two and starting and ending on different letters:

\[ \{ab, ba, aab, abb, bba, bbaa, \ldots \} \]

defined by the regular expression

\[ a(a + b)^*b + b(a + b)^*a. \]

An FA and a TG which define this language are given in figure 1. Notice that regular expressions,

![Diagram](image-url)

Figure 1: The FA and TG for \( a(a + b)^*b + b(a + b)^*a \)

FAs and TGs are finite compact precise ways of denoting (possibly) infinite sets of strings of finite length.

6 Why and how are they introduced elsewhere?

Apart from being mathematically elegant and pleasing, regular expressions/finite automata are of practical use because they are also suitable for efficient implementation. A discussion of these algorithms, their implementation, and their complexity does not fall within the scope of this talk (see, for example [10, 2, 12]). However, a good starting point for the interested reader is [6]. We also do not discuss the successful application of finite state automata in natural language processing (see, for example [5, 13]). Let us now turn to two examples of how regular expressions are used in practice.
<table>
<thead>
<tr>
<th>Metacharacter(s)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>Match any character</td>
</tr>
<tr>
<td>*</td>
<td>Match zero or more preceding regular expressions</td>
</tr>
<tr>
<td>`</td>
<td>Match following regular expressions at beginning of line</td>
</tr>
<tr>
<td>$</td>
<td>Match preceding regular expressions at end of line</td>
</tr>
<tr>
<td>[ ]</td>
<td>Match any one of the enclosed characters;</td>
</tr>
<tr>
<td>[- ]</td>
<td>a hyphen, -, indicates a range of consecutive characters</td>
</tr>
<tr>
<td>[ ]</td>
<td>Do not match enclosed character(s)</td>
</tr>
<tr>
<td>{ }</td>
<td>Match a range of instances</td>
</tr>
<tr>
<td>\</td>
<td>Turn off the special meaning of the character that follows</td>
</tr>
<tr>
<td>+</td>
<td>Match one or more preceding regular expressions</td>
</tr>
<tr>
<td>?</td>
<td>Match zero or one preceding regular expression</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>( )</td>
<td>Group regular expressions</td>
</tr>
</tbody>
</table>

Table 2: Regular expression metacharacters for `egrep`.

6.1 Linux

Linux is the core of a free Unix operating system for the PC and a variety of other hardware platforms. It is a well-respected system taking its rightful place in educational and corporate networks, and is turning up everywhere, with people using it for Web servers, file servers, and workstations. A variety of its tools and commands use regular expressions as a powerful way of searching and processing text. Linux users think of regular expressions as a way of describing a pattern of characters to a program so that it can either display or modify the occurrences of that pattern. The Linux utility or command `egrep`, which is an acronym for extended global regular expression print, is one of the most basic applications of regular expressions in Linux. Other uses occur, for example, in the editors vi, ed, ex, sed, awk and emacs.

The `grep` command has the following form:

```
egrep [options] <regexp> <filename> [<filename> ...]
```

Keeping in mind that the square brackets indicate optionality, we see that the essential parts of `egrep` is `<regexp>` and `<filename>` to be replaced by our regular expression and the name of the actual file to which it should be applied, respectively.

In order to use `egrep` we need to sort out its regular expression alphabet, as well as the special so-called metacharacters, and their meanings.

The alphabet is usually not specified explicitly, but assumed to be all characters allowed by the relevant package, say ASCII for our purposes. However, this is something you need to clarify before using any particular regular expression “engine”. The `egrep` metacharacters are given in table 2.

It is useful to remember that the regular expression in the `egrep` command should be put between (single) quotes in order to bypass the Linux shell and pass the regular expression to `egrep`.

How do these symbols relate to those of section 5? Well, the Kleene star, *, is still the Kleene star, the parentheses ( and ) also have the original meaning, but our + of section 5 is now |. All the other characters in the table are introduced for convenience in `egrep`, but can be defined in terms of those of section 5. For example, in the notation of section 5, a+ is aa*, a? is A+a, and [ab] is a+b.

As an example consider an `egrep` command that will match email addresses on the file `mymail`:

```
egrep "[a-zA-Z][a-zA-Z0-9]*\([a-zA-Z0-9]+@[a-zA-Z]+\)\([a-zA-Z]+\)\+" mymail,
```

tested on Linux Redhat. Clearly, pretol@umisa.ac.za will be matched, and also preto11@yahoo.com, and nicolet@cs.up.ac.za, but not pret123.04567.9umisa.ac.za
6.2 XML

A markup language is a mechanism to identify or describe structures in a document by "marking" them with so-called nametags or markup tags. A well-known example is HTML (Hypertext Markup Language), a wide web language that provides us with a very useful medium to share information. However, in order to use or understand an HTML document you have to be familiar with the HTML markup tags. It is also easy to understand that different kinds of documents may require different markup tags. For example, a document representing a mathematics text book may need tags that are different from those needed for a telephone directory, or a dictionary. The contradictory needs of increasing the set of HTML tags while maintaining some kind of a standard set was recognised as a drawback of HTML. Another well-known markup language is \texttt{LaTeX}, which I used to markup the text of this talk!

One of the most prominent formalisms for defining marked-up documents, and which overcomes i.a. this problem is the meta-markup-language, XML, which stands for eXtensible Markup Language. In XML the structure of marked-up documents is described by means of DTDs (Document Type Definitions) which describe the set of rules that each document of a given document type must conform to. Indeed, the markup tags and how they should be used in a particular document are specified in the associated DTD. As expected, the mathematics text book and the telephone directory will have different DTDs!

Since the topic of this talk is regular expressions, I shall give a compact, formal definition of an XML document and a DTD, showing the role that regular expressions play, and illustrate these by means of a simple example [1]. In most XML books this requires at least half a dozen pages, and they often do not even mention regular expressions - another good reason to expose students to formal methods in computer science!

**Definition:**
An XML document is a pair \((D,d)\), where \(D\) is a DTD and \(d\) is the document instance.

- The document instance is made up of units, called *elements*, which denote the logical components of the document and are delimited by *markup tags*.
- The DTD specifies the logical structures, and hence the markups, that are admissible, in terms of a set of *element type definitions*.
- In an XML document \((D,d)\) the document instance \(d\) has to conform to the DTD \(D\), according to the definition of conformance, a topic not covered in this talk.
- We consider two sets, \(T\) of *terminals*, and \(E\) of *element types*. To each element type \(E \in E\) we associate a *start tag* \(< E >\) and an *end tag* \(< /E >\). These sets are important for us since together they will form the alphabet of our promised regular expression! In our example

\[
\#PCDATA \in T,
\]

where \#PCDATA represents a generic string with no structure, and

\[
\{\text{Mail,From,To,Address,Subject,Body}\} \subseteq E
\]

with the structure of these element types specified by means of DTDs.
- The symbol \(S\) denotes any member of the set \(T \cup E\).

We are now ready to define a DTD:

**Definition:**
A Document Type Definition \(D\) is a pair \((P,R)\), where \(P\) is a set of *element type definitions,* and \(R \in E\) is the *root element type*, i.e. the element type that specifies the *document type*.

- In our example \texttt{Mail} is the root element type.
• Each element type definition associates with each $E \in E$ a unique $\alpha$, where $E$ is the defined element type, and $\alpha$, the so-called content model, is a regular expression over the alphabet $T \cup E$, viz.

$$\alpha \text{ is either } S \text{ or } \Lambda \text{ or } \alpha_1|\alpha_2 \text{ or } \alpha_1,\alpha_2 \text{ or } \alpha^* \text{ or } \alpha^+,$$

where $|$ is the operator of section 5, the comma denotes concatenation, $?$ and $+$ are as in section 6, and the Kleene star is as usual.

• The string any abbreviates $(E_1 | \cdots | E_n)^*$ where $E_1, \ldots, E_n$ are the element types in $P$.

So, we see that in DTDs, which is the notion that affords XML documents their portability, regular expressions play a central role!

In the example below we show an XML document for mail documents, with the DTD embedded in the corresponding document instance.

Example:

```xml
<?xml version='1.0'?>
<!DOCTYPE Mail [ 
  <!ELEMENT Mail (From,To, Subject)?, Body)> 
  <!ELEMENT From (Address)> 
  <!ELEMENT To (Address)+> 
  <!ELEMENT Address (#PCDATA)> 
  <!ELEMENT Subject (#PCDATA)> 
  <!ELEMENT Body (#PCDATA | any)> ]>

<Mail> 
  <From>pretol@unisa.ac.za </Address> 
</From> 
  <To> 
    <Address>biffie@unisa.ac.za </Address> 
    <Address>nicolet@cs.up.ac.za </Address> 
  </To> 
  <Subject>sacla 2001 talk </Subject> 
  <Body> 
    My apologies for submitting the talk so late! 
    Hope I may still be on the program. 
    Cheers, 
    Laurette. 
  </Body> 
</Mail>
```

7 Concluding remarks

We trust that you now agree that every modern computer scientist should know and love regular expressions. In conclusion, we hope that we were able to convince you of the importance of teaching regular expressions in particular, and theoretical computer science in general, to all your students. It is only by truly understanding the foundations of and the concepts underlying our discipline that we will be able to continue coping with the rapidly changing IT world!

References


