A Short Introduction to Regular Expressions and Context Free Grammars.

Theodore Norvell. Software Engineering 7893

Typseset November 8, 2002

1 Alphabets, Sequences and Languages

An *alphabet* is a finite set. We call the members of the alphabet *symbols*; often they are characters.

A *finite sequence* over an alphabet Σ is a function from $\{0, 1, \ldots, N-1\}$ (for some integer $N \geq 0$) to the Σ. The size of the function's domain is the *length* of the sequence. We write sequences like this: aab is a sequence mapping 0 to symbol a, 1 to a, and 2 to b. We write a sequence of length 1 as a (whether the symbol or the sequence is meant will be clear from context) and the sequence of length 0 as ϵ.

A *language* over an alphabet is a set of finite sequences over the alphabet. For example if we have an alphabet of two distinct symbols a and b, then each of the following is a language over that alphabet

$$\{\} \qquad \{\epsilon\} \qquad \{a, b, ab, ba\} \qquad \{a, ab, aba, abab, ababa, ...\} \qquad \{\epsilon, a, aa, aaa, aaaa, ...\}$$

The first example is the empty language, whereas the second example is a language containing only the empty sequence; these are different. Notice that while all the sequences in a language are finite, the language itself can be infinite (as in the last two examples).

2 Regular expressions

Listing all the elements of a language is fine for finite languages, but not good for infinite languages. We need a notation to describe languages. *Regular expressions* is one such notation. Various software tools support regular expression using various syntaxes; the syntax I use here is typical, but not unique. There are 5 basic ways to form a regular expression.

Alphabet members:. Any member of the alphabet is a regular expression. It describes the language that consists of one sequence which consists of that one symbol exactly once. For example, if our alphabet contains a symbols a and b, we have

Regular Expression	Language
a	$\{a\}$
b	$\{b\}$

Epsilon: The symbol ϵ is a regular expression. It describes the language $\{\epsilon\}$.

Alternation: If E and F are regular expressions then $E \mid F$ is also a regular expression; it describes the set of all sequences described by either E or F (or both).

Regular Expression	Language
$a \mid b$	$\{a, b\}$
$\epsilon \mid a$	$\{\epsilon, a\}$
$a \mid b \mid a$	$\{a, b\}$

Catenation: If E and F are regular expressions, then $E\ F$ is a regular expression; it describes all sequences you can make by sticking together a sequence described by E (on the left) and a sequence described by F (on the right). Examples:

Regular Expression	Language
$a\ b$	$\{ab\}$
$\epsilon\ a$	$\{a\}$
$a\ b\ a$	$\{aba\}$
$(a \mid b)\ (c \mid d)$	$\{ac, ad, bc, bd\}$
$a\ (b \mid \epsilon)$	$\{a, ab\}$
$((a\ b) \mid c)\ d$	$\{abd, cd\}$
$(a \mid b)(a \mid b)(a \mid b)$	$\{aaa, aab, aba, aabb, baa, bab, bba, babb\}$

We consider catenation to have higher precedence than alternation, so the second last example can be written $(a\ b \mid c)\ d$.

As the last example hints, even for finite languages, using a regular expression can be far more concise than listing all the elements.

Using the notations so far, we can describe any nonempty finite language, but no infinite languages. The key to describing infinite languages is the following notation.

Repetition: If E is a regular expression, then E^* is a regular expression describing sequences that are made up of the catenation of a finite number (0 or more) of sequences, each described by E. If we allowed infinitely large regular expressions (which we don't!), then E^* would describe the same language as

$$\epsilon \mid E \mid E\ E \mid E\ E\ E \mid \cdots$$

Examples:

Regular Expression	Language
a^*	$\{\epsilon, a, aa, aaa, ...\}$
$(ab)^*$	$\{\epsilon, ab, abab, ababab, ...\}$
$(a \mid b)^*$	$\{\epsilon, a, b, aa, ab, ba, bb, aaa, ...\}$
$(a\ b \mid c)^*$	$\{\epsilon, c, ab, cc, abc, cab, ccc, abab, ...\}$

Repetition has higher precedence than either alternation or catenation, so ab^* means $a(b^*)$

That is all there is to regular expressions. However for brevity, we abbreviate certain constructs. $\{E\}_m^n$ means at least m and at most n repetitions of E. The defaults for m and n are 0 and infinity respectively, thus $\{E\}$ is the same as E^*. Note that the use of braces here is quite unrelated to their use in delimiting sets. Examples:

Regular Expression	Language
$\{ab\}_2^4$	$\{abab, ababab, abababab\}$
$\{ab\}_1$	$\{ab, abab, ababab, ...\}$
$\{a \mid b\}^2$	$\{\epsilon, a, b, aa, ab, ba, bb\}$
$\{a\ b \mid c\}$	$\{\epsilon, c, ab, cc, abc, cab, ccc, abab, ...\}$

When the alphabet is totally ordered (e.g. the characters in a character set), then $[a-b]$ is used to describe all single symbol sequences consisting of a symbol between a and b (inclusive). For example, if we use the characters of the ASCII character set and order them numerically, then [a−z] means any lower case letter.

Regular expressions are often used in file editors for finding strings within files (vi and Ultraedit are two such editors). The AWK, Perl and Javascript languages all support regular expressions for searching strings. The Unix utilities grep, fgrep, and egrep use regular expressions for searching files. Regular expressions have even been used to describe digital circuits. lex and flex are program generators that can convert a set of regular expressions into a C program that digests files according to the regular expressions; JavaCC is a similar program generator for Java. lex and flex are often used to write lexical analyzers for compilers, but can be applied in far ranging ways.

Given a sequence and a regular expression, it is possible to decide if the sequence is described by the regular expression in $O(N)$ time and $O(1)$ space, where N is the length of the sequence.

Here are some examples. A regular expression describing hexadecimal numbers in Java:

$$\texttt{0}\ (\texttt{x} \mid \texttt{X})\ \{[\texttt{0}-\texttt{9}] \mid [\texttt{a}-\texttt{f}] \mid [\texttt{A}-\texttt{F}]\}_1\ (\texttt{l} \mid \texttt{L} \mid \epsilon)$$

A regular expression describing DOS file names

$$\{H\}_0^8.\{H\}_0^3$$

where H is ([0 − 9] | [a − z] | [A − Z] | − |! | @ | # | $ | % | ^ |etc.).

3 Extended Context Free Grammars

Regular expressions are a nice formalism, but they can not describe all languages. Any language that can be described by a regular expression is called a *regular language*. To go beyond regular expressions, we look at extended context free grammars.

Extended context free grammars are a generalization of regular expressions. Using extended context free grammars we can describe more languages.[1]

An *extended context free grammar* consists of an alphabet, a nonempty finite set of symbols not in the alphabet (called the set of *nonterminal symbols*) and a set of *productions*. Each production is consists of a nonterminal symbol and a regular expression; we write it as $N \rightarrow E$. Each nonterminal symbol appears on the left-hand side of exactly one production. The alphabet of the regular expressions is the union of the alphabet of the grammar and the set of nonterminal symbols. One nonterminal is singled out to be *the start nonterminal*.

Here is an example grammar: It has an alphabet of $\{l, x, r\}$, nonterminal symbols $\{S, B\}$, where S is the start nonterminal, and productions

$$\begin{aligned} S &\rightarrow l\ B\ r \\ B &\rightarrow l\ B\ r \mid x \end{aligned}$$

[1] But still not every language. There are more powerful description techniques that allow you to describe languages that no context free grammar can describe.

It is a philosophical issue whether there exists a notation that can describe every language. Most people, who have thought about it, think that there is no such language, on the basis of proofs that seem to show how to find, for any proposed description notation, a language that it can not describe. Others counter that no one can "properly" describe a language that can not be described in, for example, the C programming language, and that the existence of such an indescribable things is doubtful. Luckily this question, while very interesting, does not need to be answered for the purposes of this course.

The idea is that you start with the start nonterminal and replace it with any sequence in the language described by its regular expression. Then you pick any occurrence of a nonterminal in the resulting sequence, and replace it with any sequence in the language described by the nonterminal's regular expression. Repeat until there are no nonterminals left. Any sequence you can reach by this process is considered in the *language described by the grammar*.

Let's write $s \Rightarrow t$ to mean that sequence t can be obtained from sequence s by picking one occurrence of a nonterminal in s are replacing it by one sequence in the language described by that nonterminal's regular expression. Now using the example grammar we can see:

	S	Replace S with lBr
$\Rightarrow$	lBr	Replace B with lBr
$\Rightarrow$	$llBrr$	Replace B with x
$\Rightarrow$	$llxrr$	

so $llxrr$ is in the language described by the grammar. A little thought will show that the language is

$$\{lxr, llxrr, lllxrrr, ...\}$$

Sufficiently more thought will convince you that this language is not regular.

Exaclty those languages that can be described by an extended context free grammar are called *context free languages*.[2]

Programming languages are usually described by a combination of regular expressions, to show how source files are broken into tokens, and an extended context free grammar to show how tokens are assembled into syntactically correct programs. Some aspects of programming languages are beyond the capabilities of extended context free grammars to describe; e.g., for most programing languages it is impossible to create an extended context free grammar that describes only programs that do not contain type errors. Still, the extended context free grammar provides a nice framework on which to base programs that not only decide syntactic correctness, but also matters such as type correctness and even what machine code to generate, i.e., complete compilers. Extended context free grammars also find application in describing complex input (and output) formats of all kinds, for example HTML, or communications protocols, even user interfaces.

Given a sequence and an extended context free grammar, it is possible to decide if the sequence is described by the grammar in $O(N^3)$ time and $O(N^2)$ space, where N is the length of the sequence. This isn't very good, so tools that are based on extended context free grammars usually restrict the grammars they accept so as to allow strategies that result in $O(N)$ time and $O(N)$ space. These restrictions restrict the set of languages the tools can handle; this is rarely a problem in practice, since people tend to create languages that can be parsed quickly. yacc is a popular tool for generating C programs that dissect a file according to a given (restricted) context free grammar. JavaCC (mentioned above) is a similar tool for Java.

Here are some examples of extended context free grammars. A grammar for expressions in a

[2] There is a related notation that is called *context free grammars*. In a context free grammar, the regular expressions used must be alternations of catenations of alphabet symbols and ϵ. The word "extended" refers to the use of full regular expressions. While the formalism of context free grammars is simpler than that of extended context free grammars, the grammars themselves are a bit harder to read and write. The two notations can describe exactly the same set of languages, that is, the context free languages.

Context free grammars are also called "Backus-Naur Form" grammars, or just "BNF grammars". Extended context free grammars are often called "Extended BNF grammars", or just "EBNF grammars".

simple programming language, with start symbol E.

$$\begin{array}{lcl} E & \rightarrow & P\ (B\ P)^* \\ P & \rightarrow & id \mid num \mid lpar\ E\ rpar \mid -\ P \\ B & \rightarrow & + \mid - \mid * \mid / \mid < \mid \leq \mid = \mid \neq \end{array}$$

We can add to the above grammar and change the start symbol to B to get a grammar for the statements of a simple programming language

$$\begin{array}{lcl} B & \rightarrow & S^* \\ S & \rightarrow & id := E \mid put\ E \mid get\ id \\ & \mid & if\ E\ then\ B\ (else\ B \mid \epsilon)\ end \mid while\ E\ do\ B\ end \end{array}$$

Now suppose we have a program

```
get i
if i<0 then
      put -i
else
      put i
end
```

Lexical analysis turns this into a sequence of tokens:

$$get\ id\ if\ id < num\ then\ put\ -\ id\ else\ put\ id\ end$$

We can see that this sequence is in the language described by the grammar as follows

	B	Replace B with $S\ S$
$\Rightarrow$	$S\ S$	Replace first S with $get\ id$
$\Rightarrow$	$get\ id\ S$	Replace S with $if\ E\ then\ B\ else\ B\ end$
$\Rightarrow$	$get\ id\ if\ E\ then\ B\ else\ B\ end$	And so on

Compilers go through a similar process of finding a proof that an input file conforms to the grammar of the language. The compiler uses the structure discovered in this syntactic analysis to drive nonsyntactic checking (such as type-checking) and code generation.