Chapter 3: Regular Languages

In this chapter, we study:

- Regular expressions and languages;
- Four kinds of finite automata;
- Algorithms for processing regular expressions and finite automata;
- Properties of regular languages;
- Applications of regular expressions and finite automata to searching in text files and lexical analysis.

Copyright © 2003-5 Alley Stoughton

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

The LaTeX source of these slides, the associated book, and the distribution of the Forlan toolset are available on the WWW at http://people.cis.ksu.edu/~stough/forlan/.

Section 3.1: Regular Expressions and Languages

In this section, we:

- Define several operations on languages;
- Say what regular expressions are, what they mean, and what regular languages are;
- Begin to show how regular expressions can be processed by Forlan.

(3.1) Concatenation of Languages

The *concatenation* of languages L_1 and L_2 ($L_1 \mathbin{@} L_2 \in \mathbf{Lan}$) is the language

$$\{\, x_1 \mathbin{@} x_2 \mid x_1 \in L_1 \text{ and } x_2 \in L_2 \,\}.$$

I.e., $L_1 \mathbin{@} L_2$ consists of all strings that can be formed by concatenating an element of L_1 with an element of L_2.

For example,

$$\begin{aligned}\{\mathsf{ab}, \mathsf{abc}\} \mathbin{@} \{\mathsf{cd}, \mathsf{d}\} &= \{(\mathsf{ab})(\mathsf{cd}), (\mathsf{ab})(\mathsf{d}), (\mathsf{abc})(\mathsf{cd}), (\mathsf{abc})(\mathsf{d})\} \\ &= \{\mathsf{abcd}, \mathsf{abd}, \mathsf{abccd}\}.\end{aligned}$$

(3.1) Concatenation of Languages (Cont.)

Concatenation of languages is associative: for all $L_1, L_2, L_3 \in \mathbf{Lan}$,

$$(L_1 \mathbin{@} L_2) \mathbin{@} L_3 = L_1 \mathbin{@} (L_2 \mathbin{@} L_3).$$

And, $\{\%\}$ is the identity for concatenation: for all $L \in \mathbf{Lan}$,

$$\{\%\} \mathbin{@} L = L \mathbin{@} \{\%\} = L.$$

Furthermore, $\emptyset$ is the zero for concatenation: for all $L \in \mathbf{Lan}$,

$$\emptyset \mathbin{@} L = L \mathbin{@} \emptyset = \emptyset.$$

We often abbreviate $L_1 \mathbin{@} L_2$ to $L_1 L_2$.

(3.1) Raising a Language to a Power

We define the *language* $L^n \in \mathbf{Lan}$ *formed by raising a language* L *to a power* $n \in \mathbb{N}$ by recursion on n:

$$L^0 = \{\%\}, \text{ for all } L \in \mathbf{Lan};$$
$$L^{n+1} = LL^n, \text{ for all } L \in \mathbf{Lan} \text{ and } n \in \mathbb{N}.$$

We assign this operation higher precedence than concatenation, so that LL^n means $L(L^n)$ in the above definition.

For example, we have that

$$\begin{aligned}\{\mathsf{a},\mathsf{b}\}^2 &= \{\mathsf{a},\mathsf{b}\}\{\mathsf{a},\mathsf{b}\}^1 = \{\mathsf{a},\mathsf{b}\}\{\mathsf{a},\mathsf{b}\}\{\mathsf{a},\mathsf{b}\}^0 \\ &= \{\mathsf{a},\mathsf{b}\}\{\mathsf{a},\mathsf{b}\}\{\%\} = \{\mathsf{a},\mathsf{b}\}\{\mathsf{a},\mathsf{b}\} \\ &= \{\mathsf{aa},\mathsf{ab},\mathsf{ba},\mathsf{bb}\}.\end{aligned}$$

(3.1) Raising a Language to a Power (Cont.)

Proposition 3.1.1
For all $L \in \mathbf{Lan}$ *and* $n, m \in \mathbb{N}$, $L^{n+m} = L^n L^m$.

Proof. An easy mathematical induction on n. The language L and the natural number m can be fixed at the beginning of the proof. □

Thus, if $L \in \mathbf{Lan}$ and $n \in \mathbb{N}$, then

$$L^{n+1} = LL^n \qquad \text{(definition)},$$

and

$$L^{n+1} = L^n L^1 = L^n L \qquad \text{(Proposition 3.1.1)}.$$

(3.1) Raising a Language to a Power (Cont.)

Another useful fact about language exponentiation is:

Proposition 3.1.2
For all $w \in \mathbf{Str}$ *and* $n \in \mathbb{N}$, $\{w\}^n = \{w^n\}$.

Proof. By mathematical induction on n. $\square$

For example, we have that $\{01\}^4 = \{(01)^4\} = \{01010101\}$.

(3.1) The Kleene Closure of a Language

The *Kleene closure* (or just *closure*) of a language L ($L^* \in \mathbf{Lan}$) is the language

$$\bigcup\{\, L^n \mid n \in \mathbb{N}\,\}.$$

Thus, for all w,

$$\begin{aligned} w \in L^* \quad &\text{iff} \quad w \in A, \text{ for some } A \in \{\, L^n \mid n \in \mathbb{N}\,\} \\ &\text{iff} \quad w \in L^n \text{ for some } n \in \mathbb{N}. \end{aligned}$$

In other words:

- $L^* = L^0 \cup L^1 \cup L^2 \cup \cdots$;
- L^* consists of all strings that can be formed by concatenating together some number (maybe none) of elements of L (the same element of L can be used as many times as is desired).

(3.1) The Kleene Closure of a Language (Cont.)

For example,

$$\begin{aligned}\{\mathsf{a}, \mathsf{ba}\}^* &= \{\mathsf{a}, \mathsf{ba}\}^0 \cup \{\mathsf{a}, \mathsf{ba}\}^1 \cup \{\mathsf{a}, \mathsf{ba}\}^2 \cup \cdots \\ &= \{\%\} \cup \{\mathsf{a}, \mathsf{ba}\} \cup \{\mathsf{aa}, \mathsf{aba}, \mathsf{baa}, \mathsf{baba}\} \cup \cdots\end{aligned}$$

Suppose $w \in \mathbf{Str}$. By Proposition 3.1.2, we have that, for all x,

$$\begin{aligned}x \in \{w\}^* \quad &\text{iff} \quad x \in \{w\}^n, \text{ for some } n \in \mathbb{N}, \\ &\text{iff} \quad x \in \{w^n\}, \text{ for some } n \in \mathbb{N}, \\ &\text{iff} \quad x = w^n, \text{ for some } n \in \mathbb{N}.\end{aligned}$$

(3.1) The Kleene Closure of a Language (Cont.)

If we write $\{0, 1\}^*$, then this could mean:

- All strings over the alphabet $\{0, 1\}$ (Section 2.1); or
- The closure of the language $\{0, 1\}$.

Fortunately, these languages are equal, and this kind of ambiguity is harmless.

(3.1) Precedences of Language Operations

We assign our operations on languages relative precedences as follows:

- Highest: closure ($(\cdot)^*$) and raising to a power ($(\cdot)^n$);
- Intermediate: concatenation ($@$, or just juxtapositioning);
- Lowest: union ($\cup$), intersection ($\cap$) and difference ($-$).

For example, if $n \in \mathbb{N}$ and $A, B, C \in \mathbf{Lan}$, then $A^* B C^n \cup B$ abbreviates $((A^*)B(C^n)) \cup B$.

Can $((A \cup B)C)^*$ be abbreviated? No—removing either pair of parentheses will change its meaning.

(3.1) Regular Expressions

Let the set $\mathbf{RegLab}$ of *regular expression labels* be

$$\mathbf{Sym} \cup \{\%, \$, *, @, +\}.$$

Let the set $\mathbf{Reg}$ of *regular expressions* be the least subset of $\mathbf{Tree}_{\mathbf{RegLab}}$ such that:

- (empty string) $\% \in \mathbf{Reg}$;
- (empty set) $\$ \in \mathbf{Reg}$;
- (symbol) for all $a \in \mathbf{Sym}$, $a \in \mathbf{Reg}$;
- (closure) for all $\alpha \in \mathbf{Reg}$, $*(\alpha) \in \mathbf{Reg}$;
- (concatenation) for all $\alpha, \beta \in \mathbf{Reg}$, $@(\alpha, \beta) \in \mathbf{Reg}$;
- (union) for all $\alpha, \beta \in \mathbf{Reg}$, $+(\alpha, \beta) \in \mathbf{Reg}$.

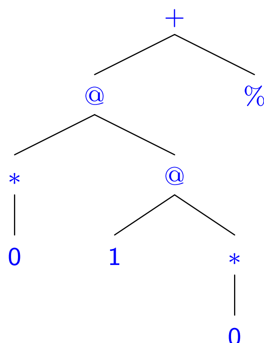
Whenever possible, we will use the mathematical variables α, β and γ to name regular expressions. Since regular expressions are $\mathbf{RegLab}$-trees, we may talk of their sizes and heights.

(3.1) Example Regular Expression

For example,

$$+(@(*(0), @(1, *(0))), \%),$$

i.e.,



is a regular expression.

(3.1) Principle of Induction on $\mathbf{Reg}$

The *principle of induction on* $\mathbf{Reg}$ says that

$$\text{for all } \alpha \in \mathbf{Reg}, P(\alpha)$$

follows from showing

- $P(\%)$;
- $P(\$)$;
- for all $a \in \mathbf{Sym}$, $P(a)$;
- for all $\alpha \in \mathbf{Reg}$, if $P(\alpha)$, then $P(*(\alpha))$;
- for all $\alpha, \beta \in \mathbf{Reg}$, if $P(\alpha)$ and $P(\beta)$, then $P(@(\alpha, \beta))$;
- for all $\alpha, \beta \in \mathbf{Reg}$, if $P(\alpha)$ and $P(\beta)$, then $P(+(\alpha, \beta))$.

(3.1) Abbreviating Regular Expressions

To increase readability, we use infix and postfix notation, abbreviating:

- $*(\alpha)$ to α^* or $\alpha*$;
- $@(\alpha, \beta)$ to $\alpha \mathbin{@} \beta$;
- $+(\alpha, \beta)$ to $\alpha + \beta$.

We assign the operators $(\cdot)^*$, $@$ and $+$ the following precedences and associativities:

- Highest: $(\cdot)^*$;
- Intermediate: $@$ (right associative);
- Lowest: $+$ (right associative).

(3.1) Abbreviating Regular Expressions (Cont.)

We parenthesize regular expressions when we need to override the default precedences and associativities, and for reasons of clarity.

We often abbreviate $\alpha \mathbin{@} \beta$ to $\alpha\beta$.

For example, we can abbreviate the regular expression $+(@(*(0), @(1, *(0))), \%)$ to $0^* \mathbin{@} 1 \mathbin{@} 0^* + \%$ or $0^*10^* + \%$.

Can $((0 + 1)2)^*$ be further abbreviated? No—removing either pair of parentheses would result in a different regular expression.

(3.1) Ordering of Regular Expressions

We order the elements of $\mathbf{RegLab}$ as follows:

$$\% < \$ < \text{symbols in order} < * < @ < +.$$

We order regular expressions first by their root labels, and then, recursively, by their children, working from left to right.

For example, we have that

$$\% < *(\%) < *(@(\$, *(\$))) < *(@(\mathsf{a}, \%)) < @(\%, \$),$$

i.e.,

$$\% < \%^* < (\$\$^*)^* < (\mathsf{a}\%)^* < \%\$.$$

(3.1) The Meaning of Regular Expressions

The *language generated by* a regular expression α ($L(\alpha)$) is defined by recursion:

$$\begin{aligned}
L(\%) &= \{\%\}; \\
L(\$) &= \emptyset; \\
L(a) &= \{a\}, \text{ for all } a \in \mathbf{Sym}; \\
L(*(\alpha)) &= L(\alpha)^*, \text{ for all } \alpha \in \mathbf{Reg}; \\
L(@(\alpha, \beta)) &= L(\alpha) \mathbin{@} L(\beta), \text{ for all } \alpha, \beta \in \mathbf{Reg}; \\
L(+(\alpha, \beta)) &= L(\alpha) \cup L(\beta), \text{ for all } \alpha, \beta \in \mathbf{Reg}.
\end{aligned}$$

(Note that $L(\alpha)$ is, indeed, a language, whenever α is a regular expression.) We say that w *is generated by* α iff $w \in L(\alpha)$.

(3.1) Meaning Example

For example,

$$\begin{aligned}
L(0^*10^* + \%) &= L(+(@(*(0), @(1, *(0))), \%)) \\
&= L(@(*(0), @(1, *(0)))) \cup L(\%) \\
&= L(*(0))L(@(1, *(0))) \cup \{\%\} \\
&= L(0)^* L(1) L(*(0)) \cup \{\%\} \\
&= \{0\}^* \{1\} L(0)^* \cup \{\%\} \\
&= \{0\}^* \{1\} \{0\}^* \cup \{\%\} \\
&= \{\, 0^n 1 0^m \mid n, m \in \mathbb{N} \,\} \cup \{\%\}.
\end{aligned}$$

E.g., 0001000, 10, 001 and $\%$ are generated by $0^*10^* + \%$.

(3.1) Raising a Regular Expression to a Power

We define the *regular expression* $\alpha^n \in \mathbf{Reg}$ *formed by raising a regular expression* α *to a power* $n \in \mathbb{N}$ by recursion on n:

$$\begin{aligned}
\alpha^0 &= \%, \text{ for all } \alpha \in \mathbf{Reg}; \\
\alpha^1 &= \alpha, \text{ for all } \alpha \in \mathbf{Reg}; \\
\alpha^{n+1} &= \alpha\alpha^n, \text{ for all } \alpha \in \mathbf{Reg} \text{ and } n \in \mathbb{N} - \{0\}.
\end{aligned}$$

We assign this operation the same precedence as closure, so that $\alpha\alpha^n$ means $\alpha(\alpha^n)$ in the above definition.

For example, $(0 + 1)^3 = (0 + 1)(0 + 1)(0 + 1)$.

Proposition 3.1.3
For all $\alpha \in \mathbf{Reg}$ *and* $n \in \mathbb{N}$, $L(\alpha^n) = L(\alpha)^n$.

Proof. An easy mathematical induction on n. □

For example, $L((0 + 1)^3) = L(0 + 1)^3 = \{0, 1\}^3$.

(3.1) The Alphabet of a Regular Expression

We define the *alphabet of* a regular expression α ($\mathbf{alphabet}(\alpha) \in \mathbf{Alp}$) by recursion:

$$
\begin{aligned}
\mathbf{alphabet}(\%) &= \emptyset; \\
\mathbf{alphabet}(\$) &= \emptyset; \\
\mathbf{alphabet}(a) &= \{a\} \text{ for all } a \in \mathbf{Sym}; \\
\mathbf{alphabet}(*(\alpha)) &= \mathbf{alphabet}(\alpha), \text{ for all } \alpha \in \mathbf{Reg}; \\
\mathbf{alphabet}(@(\alpha,\beta)) &= \mathbf{alphabet}(\alpha) \cup \mathbf{alphabet}(\beta), \text{ for all } \alpha,\beta \in \mathbf{Reg}; \\
\mathbf{alphabet}(+(\alpha,\beta)) &= \mathbf{alphabet}(\alpha) \cup \mathbf{alphabet}(\beta), \text{ for all } \alpha,\beta \in \mathbf{Reg}.
\end{aligned}
$$

(Note that $\mathbf{alphabet}(\alpha)$ is, indeed, an alphabet, whenever α is a regular expression.) For example, $\mathbf{alphabet}(0^*10^* + \%) = \{0,1\}$.

(3.1) The Alphabet of a Regular Expression (Cont.)

Proposition 3.1.4
For all $\alpha \in \mathbf{Reg}$, $\mathbf{alphabet}(L(\alpha)) \subseteq \mathbf{alphabet}(\alpha)$.

Proof. An easy induction on α. □

In other words, the proposition says that every symbol of every string in $L(\alpha)$ comes from $\mathbf{alphabet}(\alpha)$.

For example, since $L(1\$) = \{1\}\emptyset = \emptyset$, we have that

$$
\begin{aligned}
\mathbf{alphabet}(L(0^* + 1\$)) &= \mathbf{alphabet}(\{0\}^*) \\
&= \{0\} \\
&\subseteq \{0,1\} \\
&= \mathbf{alphabet}(0^* + 1\$).
\end{aligned}
$$

(3.1) Regular Languages

A language L is *regular* iff $L = L(\alpha)$ for some $\alpha \in \mathbf{Reg}$. We define

$$\begin{aligned}\mathbf{RegLan} &= \{\, L(\alpha) \mid \alpha \in \mathbf{Reg}\,\} \\ &= \{\, L \in \mathbf{Lan} \mid L \text{ is regular}\,\}.\end{aligned}$$

Since every regular expression can be described by a finite sequence of ASCII characters, we have that $\mathbf{Reg}$ is countably infinite. Since $\{0^0\}$, $\{0^1\}$, $\{0^2\}$, ..., are all regular languages, we have that $\mathbf{RegLan}$ is infinite. But, since $\mathbf{Reg}$ is countably infinite, it follows that $\mathbf{RegLan}$ is also countably infinite.

Since $\mathbf{Lan}$ is uncountable, it follows that $\mathbf{RegLan} \subsetneq \mathbf{Lan}$, i.e., there are non-regular languages.

(3.1) Synthesizing a Regular Expression

Let's consider the problem of finding a regular expression that generates the set of all strings of 0's and 1's with an even number of 0's.

A string with this property would begin with some number of 1's (possibly none). After this, the string would have some number of parts (possibly none), each consisting of a 0, followed by some number of 1's, followed by a 0, followed by some number of 1's.

The above considerations lead us to the regular expression $1^*(01^*01^*)^*$.

(3.1) Processing Regular Expressions in Forlan

The Forlan module `Reg` defines an abstract type `reg` (in the top-level environment) of regular expressions, as well as various functions and constants for processing regular expressions, including:

```
val input    : string -> reg
val output   : string * reg -> unit
val size     : reg -> int
val height   : reg -> int
val compare  : reg * reg -> order
val alphabet : reg -> sym set
val emptyStr : reg
val emptySet : reg
val fromSym  : sym -> reg
val closure  : reg -> reg
val concat   : reg * reg -> reg
val union    : reg * reg -> reg
val fromStr  : str -> reg
val power    : reg * int -> reg
```


(3.1) Example Regular Expression Processing

Here are some example uses of the functions of `Reg`:

```
- val reg = Reg.input "";
@ 0*10* + %
@ .
val reg = - : reg
- Reg.size reg;
val it = 9 : int
- val reg' = Reg.fromStr(Str.power(Str.input "", 3));
@ 01
@ .
val reg' = - : reg
- Reg.output("", reg');
010101
val it = () : unit
- Reg.size reg';
val it = 11 : int
```

(3.1) Examples (Cont.)

```
- Reg.compare(reg, reg');
val it = GREATER : order
- val reg'' = Reg.concat(Reg.closure reg, reg');
val reg'' = - : reg
- Reg.output("", reg'');
(0*10* + %)*010101
val it = () : unit
- SymSet.output("", Reg.alphabet reg'');
0, 1
val it = () : unit
- val reg''' = Reg.power(reg, 3);
val reg''' = - : reg
- Reg.output("", reg''');
(0*10* + %)(0*10* + %)(0*10* + %)
val it = () : unit
- Reg.size reg''';
val it = 29 : int
```


(3.1) Using JTR to View and Edit Regular Expressions as Trees

The Java program JTR (for "Java TRee") can be used to display and edit regular expressions as trees. It can be invoked via Forlan, or run standalone. See the Forlan WWW site for more information.