Chapter 3: Regular Languages

In this chapter, we study:

- regular expressions and languages;
- five kinds of finite automata;
- algorithms for processing and converting between regular expressions and finite automata; and
- applications of regular expressions and finite automata to hardware design, searching in text files and lexical analysis.

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; and
- begin to show how regular expressions can be processed by Forlan.

Language Operations

If L_1 and L_2 are languages, then:

- $L_1 \cup L_2$ is a language;
- $L_1 \cap L_2$ is a language;
- $L_1 - L_2$ is a language.

E.g., consider union. If L_1 and L_2 are languages, then $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$, for some alphabets Σ_1 and Σ_2. Thus $\Sigma_1 \cup \Sigma_2$ is an alphabet, and $L_1 \cup L_2 \subseteq (\Sigma_1 \cup \Sigma_2)^*$.

Language Concatenation

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

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

For example,

$$\begin{aligned} \{01, 10\} \mathbin{@} \{\%, 11\} &= \{(01)\%, (10)\%, (01)(11), (10)(11)\} \\ &= \{01, 10, 0111, 1011\}. \end{aligned}$$

Language Concatenation (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 $L_1 L_2$.

Raising a Language to a Power

We define the *language* $L^n \in \mathbf{Lan}$ *formed by raising* a language L *to the 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.

E.g., $L^1 = L^{0+1} = LL^0 = L\{\%\} = L$.

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

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

Kleene Closure

The *Kleene closure* (or just *closure*) *of* a language L (L^*) is the language

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

Thus, for all w,

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

For example,

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

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 $A^*BC^n \cup B$ abbreviates $((A^*)B(C^n)) \cup B$.

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

More Operations on Sets of Strings in Forlan

In Section 2.3, we introduced the Forlan module `StrSet`, which defines various functions for processing finite sets of strings, i.e., finite languages.

This module also defines the functions

```
val concat : str set * str set -> str set
val power  : str set * int -> str set
```

which implement our concatenation and exponentiation operations on finite languages.

More Operations in Forlan (Cont.)

Here are some examples of how these functions can be used:

```
- val xs = StrSet.fromString "ab, cd";
val xs = - : str set
- val ys = StrSet.fromString "uv, wx";
val ys = - : str set
- StrSet.output("", StrSet.concat(xs, ys));
abuv, abwx, cduv, cdwx
val it = () : unit
- StrSet.output("", StrSet.power(xs, 0));
%
val it = () : unit
- StrSet.output("", StrSet.power(xs, 1));
ab, cd
val it = () : unit
- StrSet.output("", StrSet.power(xs, 3));
ababab, ababcd, abcdab, abcdcd, cdabab, cdabcd,
cdcdab, cdcdcd
val it = () : unit
```

Regular Expressions

Let the set **RegLab** of *regular expression labels* be

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

Let the set **Reg** of *regular expressions* be the least subset of **Tree RegLab** such that:

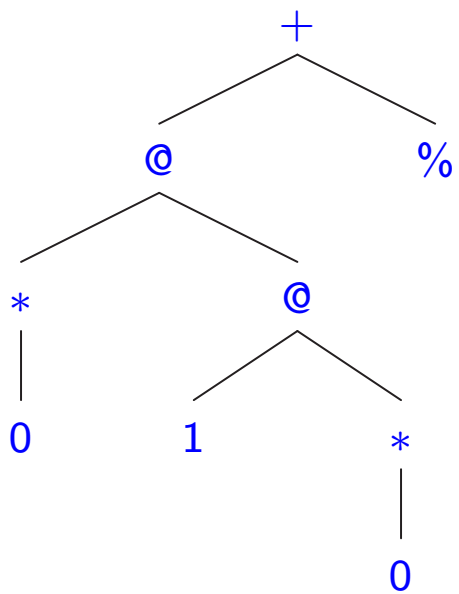
- (**empty string**) $\% \in \mathbf{Reg}$;
- (**empty set**) $\$ \in \mathbf{Reg}$;
- (**symbol**) for all $a \in \mathbf{Sym}$, $a \in \mathbf{Reg}$;
- (**closure**) for all $\alpha \in \mathbf{Reg}$, $*(\alpha) \in \mathbf{Reg}$;
- (**concatenation**) for all $\alpha, \beta \in \mathbf{Reg}$, $@(\alpha,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

Example Regular Expression

For example,

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

i.e.,



is a regular expression.

Induction on Regular Expressions

Theorem 3.1.7 (Principle of Induction on Regular Expressions) *Suppose $P(\alpha)$ is a property of a regular expression α.*
If

- $P(\%)$,
- $P(\$)$,
- *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* $P(+(\alpha, \beta))$,

then

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

We refer to (†) as the inductive hypothesis.

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).

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.

The Meaning of Regular Expressions

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

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

We say that w *is generated by* α iff $w \in L(\alpha)$.

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 $0^*10^* + \%$.

Raising a Regular Expression to a Power

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

$$\alpha^0 = \%, \text{ for all } \alpha \in \mathbf{Reg};$$

$$\alpha^1 = \alpha, \text{ for all } \alpha \in \mathbf{Reg};$$

$$\alpha^{n+1} = \alpha\alpha^n, \text{ for all } \alpha \in \mathbf{Reg} \text{ and } n \in \mathbb{N} - \{0\}.$$

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 3.1.8

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

Proof. By mathematical induction on n, case-splitting in the inductive step. □

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

The Alphabet of a Regular Expression

We define $\textbf{alphabet} \in \textbf{Reg} \to \textbf{Alp}$ by structural recursion:

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

We say that $\textbf{alphabet}\,\alpha$ is *the alphabet of* a regular expression α.

For example, $\textbf{alphabet}(0^*10^* + \%) = \{0,1\}$.

The Alphabet of a Regular Expression (Cont.)

Proposition 3.1.11
For all $\alpha \in \mathbf{Reg}$, $\mathbf{alphabet}(L(\alpha)) \subseteq \mathbf{alphabet}\,\alpha$.

Proof. An easy induction on α. $\square$

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

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

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

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

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 emptyStr : reg
val emptySet : reg
val fromSym  : sym -> reg
val closure  : reg -> reg
val concat   : reg * reg -> reg
val union    : reg * reg -> reg
```

Processing Regular Expressions in Forlan (Cont.)

```
val equal    : reg * reg -> bool
val fromStr  : str -> reg
val power    : reg * int -> reg
val alphabet : reg -> sym set
```

Forlan Syntax for Regular Expressions

The Forlan syntax for regular expressions is the infix/postfix one introduced above, where α `@` β is always written as $\alpha\beta$, and we use parentheses to override default precedences/associativities, or simply for clarity.

For example, `0*10* + %` and `(0*(1(0*))) + %` are the same regular expression. And, `((0*)1)0* + %` is a different regular expression, but one with the same meaning. Furthermore, `0*1(0* + %)` is not only different from the two preceding regular expressions, but it has a different meaning.

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
```

Examples (Cont.)

```
- 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
```

Examples (Cont.)

```
- Reg.output("", Reg.fromString "(0*(1(0*))) + %");
0*10* + %
val it = () : unit
- Reg.output("", Reg.fromString "(0*1)0* + %");
(0*1)0* + %
val it = () : unit
- Reg.output("", Reg.fromString "0*1(0* + %)");
0*1(0* + %)
val it = () : unit
- Reg.equal(Reg.fromString "0*10* + %",
=            Reg.fromString "(0*1)0* + %");
val it = false : bool
```

Graphical Editor for Regular Expression Trees

The Java program JForlan, can be used to view and edit regular expression trees. It can be invoked directly, or run via Forlan. See the Forlan website for more information.