ECS 120 Lesson 7 – Regular Expressions, Pt. 1

Oliver Kreylos

Friday, April 13th, 2001

1 Outline

Thus far, we have been discussing one way to specify a (regular) language: Giving a machine that reads a word and tells whether it is in the language or not. Though this is a valid and unambiguous specification, it is sometimes not a very helpful one. Specifying languages by automata has two major shortcomings: First, when given a language, it is often difficult to construct an automaton that accepts it; second, when given an automaton, it is often difficult to understand which language it accepts. *Regular expressions* are an alternative specification method for regular languages: They are easier to construct, and it is easier to see which language they describe by just looking at the expression. Both benefits stem from the fact that regular expressions describe the structure of words contained in a language, rather than giving a machine that must be "run" in order to decide a word.

Regular expressions are very common in computer applications because they are a powerful way to describe patterns in texts. Text editors (in their search and replace functions), programming languages such as PERL, and UNIX utilities such as grep, awk and lex all use regular expressions to describe patterns. Programming language compilers typically use regular expressions to define the lowest-level constructs of program source code ("tokens"), and the stage of the compiler responsible for recognizing tokens ("parser") is automatically constructed from those regular expressions using the lex utility.

2 Regular Expressions

Regular expressions over an alphabet Σ define languages over Σ by describing the structure of words in a language. They are based on the three *regular operations*: Union, concatenation and Kleene Star. They are very similar to arithmetic expressions like $3 + (4 \cdot 5)$: They consist of constants and operators, and they construct complex expressions from simpler building blocks. As opposed to arithmetic expressions, their values are not numbers, but languages. Examples:

- `hello` specifies the language consisting of the single word `hello`.
- `hello`$\cup$`world` specifies the language consisting of the two words `hello` and `world`.
- (`aa`)* specifies the language of all words consisting of an even number of `a`s.
- `a`$^*\circ$`bb`, often written just as `a`*`bb`, specifies the set of all words consisting of any number of `a`s followed by two `b`s.

Since every regular expression defines one language, we will write $L(R)$ to denote the language defined by regular expression R.

3 Formal Definition of Regular Expressions

Regular expressions over an alphabet Σ are defined in a recursive fashion, very similarly to arithmetic expressions. We start by defining the simplest regular expressions, and then define operations to create more complex ones from simpler building blocks:

1. $\emptyset$ is a regular expression defining the empty language, $L(\emptyset) = \emptyset \subset \Sigma^*$.
2. ϵ is a regular expression defining the language consisting only of the empty word, $L(\epsilon) = \{\epsilon\} \subset \Sigma^*$.
3. If $a \in \Sigma$ is a character, then a is a regular expression over Σ defining the language consisting of the single one-character word a, $L(a) = \{a\} \subset \Sigma^*$.

4. If R_1 and R_2 are two regular expressions over the same alphabet Σ, then $(R_1 \cup R_2)$ is a regular expression over Σ defining the union of the languages of R_1 and R_2, $L\big((R_1 \cup R_2)\big) = L(R_1) \cup L(R_2) \subset \Sigma^*$.

5. If R_1 and R_2 are two regular expressions over the same alphabet Σ, then $(R_1 \circ R_2)$, often abbreviated as $(R_1 R_2)$, is a regular expression over Σ defining the concatenation of the languages of R_1 and R_2, $L\big((R_1 \circ R_2)\big) = L\big((R_1 R_2)\big) = L(R_1)L(R_2) \subset \Sigma^*$.

6. If R is a regular expression over Σ, then (R^*) is a regular expression over Σ defining the Kleene Star of the language of R, $L\big((R^*)\big) = L(R)^* \subset \Sigma^*$.

The regular expressions generated by these definitions are fully parenthesized to avoid ambiguities. Here is how the earlier example expressions look like following the recursive definition:

- $((((\texttt{h} \circ \texttt{e}) \circ \texttt{l}) \circ \texttt{l}) \circ \texttt{o}) = ((((\texttt{he})\texttt{l})\texttt{l})\texttt{o})$
- $\big(((((\texttt{h} \circ \texttt{e}) \circ \texttt{l}) \circ \texttt{l}) \circ \texttt{o}) \cup ((((\texttt{w} \circ \texttt{o}) \circ \texttt{r}) \circ \texttt{l}) \circ \texttt{d})\big) = \big(((((\texttt{he})\texttt{l})\texttt{l})\texttt{o}) \cup ((((\texttt{wo})\texttt{r})\texttt{l})\texttt{d})\big)$
- $\big((a \circ a)^*\big) = \big((aa)^*\big)$
- $\big((a^*) \circ (b \circ b)\big) = \big((a^*)(bb)\big)$

Even when dropping the explicit concatenation operator $\circ$, the fully parenthesized expressions are almost impossible to read. Therefore, we introduce rules for implicit parenthezation, similar to those used in arithmetics:

- $R_1 R_2 R_3 := \big((R_1 R_2) R_3\big) = \big(R_1 (R_2 R_3)\big)$. Since the concatenation operator is associative, we can drop parentheses in sequences of concatenation operations entirely.
- $R_1 \cup R_2 \cup R_3 := \big((R_1 \cup R_2) \cup R_3\big) = \big(R_1 \cup (R_2 \cup R_3)\big)$. Since the union operator is associative as well, we can drop parentheses in sequences of union operations entirely.
- $R_1 \cup R_2 R_3 := \big(R_1 \cup (R_2 R_3)\big)$. Concatenation has precedence over union.
- $R_1 {R_2}^* := \big(R_1 ({R_2}^*)\big)$. Kleene Star has precedence over concatenation.

- $R_1 \cup {R_2}^* := \big(R_1 \cup ({R_2}^*)\big)$. Kleene Star has precedence over union.

We also define the following shorthand notation:

- If $A = \{a_1, a_2, \ldots, a_n\} \subset \Sigma \cup \{\epsilon\}$ is a set of characters from Σ or the symbol ϵ, then A is a shorthand for $a_1 \cup a_2 \cup \cdots \cup a_n$, the regular expression denoting the language $L(A) = \{a_1, a_2, \ldots, a_n\} \subset \Sigma^*$.

Here are some more relevant examples for regular expressions over the ASCII alphabet. In the following, let $L := \{\texttt{A}, \ldots, \texttt{Z}, \texttt{a}, \ldots, \texttt{z}\}$ be the set of letters, and $D := \{\texttt{0}, \ldots, \texttt{9}\}$ the set of decimal digits.

- DD^* describes all words starting with a digit, followed by any number of digits. This is the set of all positive integers in decimal notation.
- $\{\texttt{+}, \texttt{-}, \epsilon\}DD^*$ describes the language of all integer constants with an optional sign.
- $\{\texttt{+}, \texttt{-}, \epsilon\}(DD^* \cup DD^* \,.\, D^* \cup D^* \,.\, DD^*)\big((\{\texttt{E}, \texttt{e}\}\{\texttt{+}, \texttt{-}, \epsilon\}DD^*) \cup \epsilon\big)$ describes the language of all floating-point constants with an optional sign and exponential part, as recognized by the C programming language.
- $(L \cup \texttt{\_})(L \cup D \cup \texttt{\_})^*$ describes the language of all valid identifiers in the C programming language (not taking reserved words into account).

4 Equivalence of Regular Expressions and Finite State Machines

Earlier we have claimed that the class of languages that can be described by regular expressions is exactly the class of regular languages. We are now going to prove this statement. First we will show that the language $L(R)$ generated by any regular expression R is accepted by some NFA M. Second, we show that the language $L(M)$ accepted by any automaton M is generated by some regular expression R.

5 Construction of Automata from Regular Expressions

We will prove the existence of an automata that accepts the language generated by a regular expression by *structural induction*. This means, we will

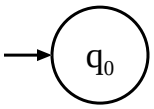

Figure 1: An NFA M accepting the empty language, $L(M) = \emptyset$.

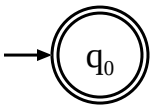

Figure 2: An NFA M accepting the language consisting only of the empty word, $L(M) = \{\epsilon\}$.

follow the recursive definition of regular expressions and construct automata accepting the languages generated by simple regular expressions first, and will then show how to combine those automata to accept the languages generated by more complex ones. For all the following constructions, we will assume that all regular expressions are over some alphabet Σ.

5.1 Case 1: $R = \emptyset$

If $R = \emptyset$, then $L(R) = \emptyset$. The empty language is accepted by the NFA $M_\emptyset := \big(\{q_0\}, \Sigma, \delta, q_0, \emptyset\big)$ where $\delta(q_0, a) = \emptyset$ for all $a \in \Sigma_\epsilon$. A transition diagram for this automaton is shown in Figure 1.

5.2 Case 2: $R = \epsilon$

If $R = \epsilon$, then $L(R) = \{\epsilon\}$. The language consisting only of the empty word is accepted by the NFA $M_\epsilon := \big(\{q_0\}, \Sigma, \delta, q_0, \{q_0\}\big)$ where $\delta(q_0, a) = \emptyset$ for all $a \in \Sigma_\epsilon$. A transition diagram for this automaton is shown in Figure 2.

5.3 Case 3: $R = a$

If $R = a$ for some character $a \in \Sigma$, then $L(R) = \{a\}$. The language consisting only of the word a is accepted by the NFA $M_a := \big(\{q_0, q_1\}, \Sigma, \delta, q_0, \{q_1\}\big)$ where

$$\forall q \in \{q_0, q_1\}, x \in \Sigma_\epsilon : \delta(q, x) = \begin{cases} \{q_1\}, & \text{if } q = q_0 \text{ and } x = a \\ \emptyset & \text{otherwise} \end{cases}$$

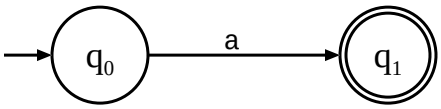

Figure 3: An NFA M accepting the language consisting only of the word a, $L(M) = \{a\}$.

A transition diagram for this automaton is shown in Figure 3.

5.4 Case 4: $R = R_1 \cup R_2$

If $R = R_1 \cup R_2$ for two regular expressions R_1 and R_2, then $L(R) = L(R_1) \cup L(R_2)$. To construct an automaton that accepts $L(R)$, we first construct the two automata M_1 and M_2 that accept $L(R_1)$ and $L(R_2)$, respectively. Then we use the union construction shown earlier to build an automaton M that accepts $L(R_1) \cup L(R_2)$.

5.5 Case 5: $R = R_1 R_2$

If $R = R_1 R_2$ for two regular expressions R_1 and R_2, then $L(R) = L(R_1)L(R_2)$. To construct an automaton that accepts $L(R)$, we first construct the two automata M_1 and M_2 that accept $L(R_1)$ and $L(R_2)$, respectively. Then we use the concatenation construction shown earlier to build an automaton M that accepts $L(R_1)L(R_2)$.

5.6 Case 6: $R = {R_1}^*$

If $R = {R_1}^*$ for a regular expression R_1, then $L(R) = L(R_1)^*$. To construct an automaton that accepts $L(R)$, we first construct an automaton M_1 that accepts $L(R_1)$. Then we use the Kleene Star construction shown earlier to build an automaton M that accepts $L(R_1)^*$.