Harvey Mudd College
Computer Science 81
Spring 2007

Derivatives of Regular Expressions

Robert Keller

22 January 2007

There are many different notations for regular expressions. We work with the most basic version. Versions that are used in software tools, particularly various Unix tools, are more complex because they introduce many new symbols as short cuts, but no additional power.

Our Definition of Regular Expression

A regular expression over an alphabet Σ, and the language denoted by the expression is defined inductively as follows:

	Expression R	L(R), the language denoted by R	Interpretation
Basis	$\emptyset$	$\emptyset$	empty set
	ε	$\{\varepsilon\}$	set of just the empty string
	σ where $\sigma \in \Sigma$	$\{\sigma\}$	set of one one-letter string
Induction rules	(R + S)	L(R) $\cup$ L(S)	union
	(RS)	L(R)L(S)	concatenation
	(R)*	L(R)*	star

Inductive definitions permit proofs by structural induction, as in the construction of an NFA from any regular expression.

We often take short cuts in writing regular expressions, by omitting parentheses by using precedence rules. * binds more tightly than concatenation. Concatenation binds more tightly than union. For example, abc* + d abbreviates ((a (b (c*))) + d).

Identities for Regular Expressions

These identities are derived from considering the interpretation of regular expressions. They are equalities not in that the expressions are the same, but in that the languages they denote are the same, just like many other algebraic identities. Here R, S, and T are arbitrary regular expressions.

1. $R + \emptyset = \emptyset + R = R$
2. $R\emptyset = \emptyset R = \emptyset$
3. $R\varepsilon = \varepsilon R = R$
4. $(R + S)T = RT + ST$
5. $R(S + T) = RS + RT$

6. $(R + S)^* = (R^*S^*)^*$
7. $(R^*)^* = R^*$
8. $R(ST) = (RS)T$
9. $R(SR)^* = (RS)^*R$
10. $R^n + R^* = R^*$ for any n, including $n = 0$ ($R^0 = \varepsilon$), $n = 1$ ($R^1 = R$).
11. $R + S = S + R$
12. $R + (S + T) = (R + S) + T$
13. $(RS)^* = (R + S)^*$, provided that $\varepsilon \in L(R)$.

Note that identities involving only + (union) are known from set theory.

Derivatives of Regular Expressions

The idea of derivative of a regular expression can be used for constructing a DFA from an RE by hand, which is often more convenient than creating an NFA then performing the subset construction to get a DFA. The derivative of a regular expression R with respect to an alphabet symbol $\sigma \in \Sigma$, written R/σ, is defined by structural induction as follows:

	Expression	**Derivative**	**Rule number**
Basis	$\varnothing/\sigma$	$\varnothing$	1
	ε/σ	$\varnothing$	2
	σ/σ	ε	3
	σ/σ' where $\sigma' \neq \sigma$	$\varnothing$	4
Induction rules	$(RS)/\sigma$	$(R/\sigma)S$ if $\varepsilon \notin L(R)$	5
		$(R/\sigma)S + S/\sigma$ if $\varepsilon \in L(R)$	6
	$(R + S)/\sigma$	$(R/\sigma) + (S/\sigma)$	7
	$(R)^*/\sigma$	$(R/\sigma)\, R^*$	8

Derivatives provide a formal way to manipulate languages via regular expressions. The idea is that:

$$L(R/\sigma) = \{x \in \Sigma^* \mid \sigma x \in L(R)\}$$

In other words, $L(R/\sigma)$ is the set of all strings in L(R) that begin with σ, but with the initial σ lopped off. (Strings that don't begin with σ do not contribute to $L(R/\sigma)$.)

Examples:

Suppose that $\Sigma = \{0, 1\}$. We have the following derivatives:

Directly from rules:

$\varnothing/0 = \varepsilon/0 = \varnothing/1 = \varepsilon/1 = 1/0 = 0/1 = \varnothing$

$0/0 = 1/1 = \varepsilon$

Applying rules recursively, and using identities to simplify:

(00)/0 (using rule 5) = (0/0) 0 = ε0 = 0
(01)/0 (using rule 5)= (0/0) 1 = ε1 = 1
(00)/1 (using rule 5) = (0/0) 0 = ε0 = 0
(01)/1(using rule 5) = (0/0) 1 = ε1 = 1

(0 + 1)/0 (using rule 7) = 0/0 + 1/0 = ε + ∅ = ε
(0 + 1)/1(using rule 7) = 0/1 + 1/1 = ∅ + ε = ε

(0*1)/0 = ((0*/0)1 + 1/0 (using rule 6) = ((0/0)0*)1 + ∅ = (ε0*)1 = 0*1
(0*1)/1 = ((0*/1)1 + 1/1 (using rule 6) = ((0/1)0*)1 + ε = (∅0*)1 + ε = ∅1 + ε = ε

Application

Derivatives of regular expressions can be used to construct a DFA from a regular expression without creating an NFA, as follows:

1. Simplify R first, if possible, as that will save work later.

2. Set New = {R}, Old = ∅.

3. While New ≠ ∅
   a. Let R be an arbitrary element of New. Move R from New to Old.

   b. Let S be the derivatives of R with respect to each alphabet symbol using the above rules, simplifying where possible. Keep track of the expression from which each derivative is derived.

   c. Add to New any simplified expressions not equivalent to an expression in New ∪ Old.

4. Create a DFA with the regular expressions in Old as states.

5. The initial state is the original expression R.

6. There is a connection from R to R' labeled with σ exactly when R' is equivalent to R/σ.

7. The accepting states are those corresponding to expressions that have ε in their language.

Example

R = (01)*1 + 1

We can simplify this right away to (01)*1, since 1 ∈ L((01)*1).

New = {(01)*1}

Move (01)*1 from New to Old.

Extract derivatives of (01)*1:
((01)*1)/0 = ((01)*/0)1 = (1(01)*)1

((01)*1)/1 = ((01)*/1)1 + (1/1) = ((0/1)(01)*)1 + (1/1) = ∅ + ε = ε

New = {(1(01)*)1, ε}

Extract derivatives of (1(01)*)1:

(1(01)*)1/0 = ∅
(1(01)*)1/1 = (01)*1, which is in Old.

Extract derivatives of ε:

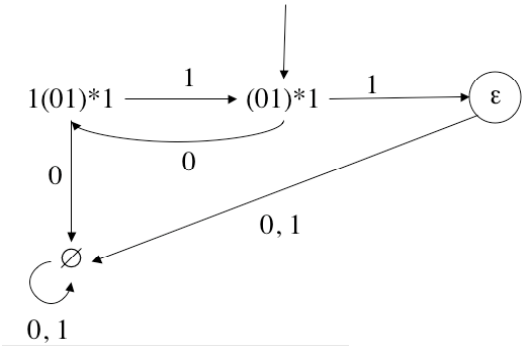
ε/0 = ∅, which is in Old.
ε/1 = ∅, which is in Old.

New = {∅}

Extract derivatives of ∅:
∅/0 = ∅, which is in Old.
∅/1 = ∅, which is in Old.

New is empty. Old = {(01)*1, 1(01)*1, ε, ∅}.

Drawn below is the DFA thus derived.



We can check visually that a regular expression representing its language is (01)*1, as desired.