CS_275 Automata and Formal Language Theory

Course Notes

Part II: The Recognition Problem (II)

Chapter II.2.: Basics of Regular Languages and Expressions

Anton Setzer
(Based on a book draft by J. V. Tucker and K. Stephenson)
Dept. of Computer Science, Swansea University

http://www.cs.swan.ac.uk/∼csetzer/lectures/automataFormalLanguage/current/index.html

March 17, 2017

II.2.1. Regular Languages (12.2)

II.2.2. Regular Expressions (13.8)

II.2.1. Regular Languages (12.2)

II.2.2. Regular Expressions (13.8)

Finite Languages are Regular

grammar	$G^{ab,aabb,aaabbb}$
terminals	a, b
nonterminals	S
start symbol	S
productions	$S \longrightarrow ab$ $S \longrightarrow aabb$ $S \longrightarrow aaabbb$

This grammar is not regular, since there can only be one terminal in the right hand string. But we can amend this:

Finite Languages are Regular

grammar	$G^{ab,aabb,aaabbb}$
terminals	a, b
nonterminals	$S, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9$
start symbol	S
productions	$S \longrightarrow aS_1,\ S_1 \longrightarrow b$ $S \longrightarrow aS_2,\ S_2 \longrightarrow aS_3,\ S_3 \longrightarrow bS_4,\ S_4 \longrightarrow b$ $S \longrightarrow aS_5,\ S_5 \longrightarrow aS_6,\ S_6 \longrightarrow aS_7,\ S_7 \longrightarrow bS_8,$ $S_8 \longrightarrow bS_9,\ S_9 \longrightarrow b$

Finite Languages are Regular

If one wishes, the above grammar can of be optimised as follows:

grammar	$G^{ab,aabb,aaabbb}$
terminals	a, b
nonterminals	$S, S_1, S_3, S_4, S_7, S_8, S_9$
start symbol	S
productions	$S \longrightarrow aS_1,\ S_1 \longrightarrow b$ $S_1 \longrightarrow aS_3,\ S_3 \longrightarrow bS_4,\ S_4 \longrightarrow b$ $S_3 \longrightarrow aS_7,\ S_7 \longrightarrow bS_8,$ $S_8 \longrightarrow bS_9, S_9 \longrightarrow b$

Lemma II.2.1.1.

Lemma (II.2.1.1.)

All finite languages are regular, and a regular grammar for them can be computed.

Proof: Extend the example above.

A Left-Linear Grammar for $a^m b^n$

The following left-linear grammar generates $\{a^m b^n \mid m, n \geq 1\}$.

grammar	$G^{left-linear, a^m b^n}$
terminals	a, b
nonterminals	S, T
start symbol	S
productions	$S \longrightarrow Sb$ $S \longrightarrow Tb$ $T \longrightarrow Ta$ $T \longrightarrow a$

A Right-Linear Grammar for $a^m b^n$

The following right-linear grammar generates $\{a^m b^n \mid m, n \geq 1\}$:

grammar	$G^{right-linear, a^m b^n}$
terminals	a, b
nonterminals	S, T
start symbol	S
productions	$S \longrightarrow aS$ $S \longrightarrow aT$ $T \longrightarrow bT$ $T \longrightarrow b$

Right-Linear Grammar for Numbers

Here is a right-linear grammars for numbers without leading zeros. We use "|" as for BNF.

grammar	G^{Number}
terminals	$0, 1, \ldots, 9$
nonterminals	$Number, Digits$
start symbol	$Number$
productions	$Number \longrightarrow 0$
	$Number \longrightarrow 1\ Digits \mid 2\ Digits \mid \cdots \mid 9\ Digits$
	$Digits \longrightarrow 0\ Digits \mid 1\ Digits \mid \cdots \mid 9\ Digits$
	$Digits \longrightarrow \epsilon$

Right-Linear Grammar for Numbers

Why didn't we use the following as in BNF?

grammar	G^{Number}
terminals	$0, 1, \ldots, 9$
nonterminals	*Number*, *Digit*, *NonZeroDigit*, *Digits*
start symbol	*Number*
productions	$Number \longrightarrow Digit \mid NonZeroDigit\ Digits$ $Digits \longrightarrow Digit \mid Digit\ Digits$ $Digit \longrightarrow 0 \mid NonZeroDigit$ $NonZeroDigit \longrightarrow 1 \mid 2 \mid \cdots \mid 9$

Answer:

Right-Linear Grammar for Post Codes

The next grammar generates the postcodes of the form SA1 8PP or in general LLd dLL for digits d and capital letters L without any leading zeros. We use the notation | as in BNF. We write ␣ for blank

Right-Linear Grammar for Post Codes

grammar	$G^{Postcode}$
terminals	$0, 1, \ldots, 9, \mathrm{A}, \mathrm{B}, \ldots, \mathrm{Z}, \llcorner\lrcorner$
nonterminals	*postcode*, *letter*2, *digit*1, *blank*1, *digit*2, *letter*3, *letter*4
start symbol	*postcode*
productions	$\begin{array}{lll} \textit{postcode} & \longrightarrow & \mathrm{A}\ \textit{letter2} \mid \mathrm{B}\ \textit{letter2} \mid \cdots \mid \mathrm{Z}\ \textit{letter2} \\ \textit{letter2} & \longrightarrow & \mathrm{A}\ \textit{digit1} \mid \mathrm{B}\ \textit{digit1} \mid \cdots \mid \mathrm{Z}\ \textit{digit1} \\ \textit{digit1} & \longrightarrow & 0\ \textit{blank1} \mid 1\ \textit{blank1} \mid \cdots \mid 9\ \textit{blank1} \\ \textit{blank1} & \longrightarrow & \llcorner\lrcorner\ \textit{digit2} \\ \textit{digit2} & \longrightarrow & 0\ \textit{letter3} \mid 1\ \textit{letter3} \mid \cdots \mid 9\ \textit{letter3} \\ \textit{letter3} & \longrightarrow & \mathrm{A}\ \textit{letter4} \mid \mathrm{B}\ \textit{letter4} \mid \cdots \mid \mathrm{Z}\ \textit{letter4} \\ \textit{letter4} & \longrightarrow & \mathrm{A} \mid \mathrm{B} \mid \cdots \mid \mathrm{Z} \end{array}$

Example Derivation

Here is a derivation of SA2␣␣8PP $\in L(G^{Postcode})$:

$$
\begin{aligned}
\textit{postcode} &\Rightarrow \text{S } \textit{letter}2 \\
&\Rightarrow \text{SA } \textit{digit}1 \\
&\Rightarrow \text{SA1 } \textit{blank}1 \\
&\Rightarrow \text{SA1␣␣ } \textit{digit}2 \\
&\Rightarrow \text{SA1␣␣8 } \textit{letter}3 \\
&\Rightarrow \text{SA1␣␣8P } \textit{letter}4 \\
&\Rightarrow \text{SA1␣␣8PP}
\end{aligned}
$$

Easier Proof that Postcodes are Regular

Can you give an easier proof that the language of postcodes is regular (both left-linear and right-linear)?

Multi-step Regular Grammars

- In general we can extend regular grammars by allowing productions such as

$$\begin{aligned} S &\longrightarrow abB \\ B &\longrightarrow aS \\ B &\longrightarrow baS \end{aligned}$$

  So instead of having only **one** terminal symbol, we can have **several**.
- As long as we remain left-linear or right-linear
  - i.e. the terminal symbols are **always to the right** or **always to the left** of the non-terminal on the right hand side of a rule

we obtain grammars which can be reduced to regular grammars.

Lemma II.2.1.2.

Lemma (II.2.1.2.)

1. *Assume a grammar G which has only productions of the form*

$$A \longrightarrow Bw \text{ or } A \longrightarrow w$$

for some $w \in T^*$, $A, B \in N$. Then $L(G) = L(G')$ for some left-linear grammar G', which can be computed from G.

2. *Assume a grammar G which has only productions of the form*

$$A \longrightarrow wB \text{ or } A \longrightarrow w$$

for some $w \in T^*$, $A, B \in N$. Then $L(G) = L(G')$ for some right-linear grammar G', which can be computed from G.

Multi-step Right-Linear/Left-Linear/Regular Grammars

We call grammars as above **multi-step right-linear/left-linear/regular grammars**.

Proof Idea for Lemma II.2.1.2.

- First omit all so called **silent productions**, i.e. productions of the form $A \longrightarrow B$ for some non-terminals A, B.
  - This requires some work.
- Then replace in the right-linear case productions

$$A \longrightarrow a_1 a_2 \cdots a_n B$$

with $n \geq 2$ by productions

$$\begin{array}{lcl} A & \longrightarrow & a_1 A_1, \\ A_1 & \longrightarrow & a_2 A_2, \\ \cdots & & \\ A_{n-1} & \longrightarrow & a_n B \end{array}$$

for some new nonterminals A_i.

- Full details can be found in the additional material

Mixing of Left- and Right-Linear

Remark

In a regular grammar we are not allowed to mix left-linear and right-linear grammars. Otherwise we would obtain truly context-free languages.

Example (Mixing Left/Right-Linear Rules)

The following grammar generates the language
L(G) = ?
which (as we will see later) is context-free but not regular.

grammar	G
terminals	a, b
nonterminals	S, T
start symbol	S
productions	$S \longrightarrow ab$ $S \longrightarrow aT$ $T \longrightarrow Sb$

II.2.1. Regular Languages (12.2)

II.2.2. Regular Expressions (13.8)

Operators for Forming Languages

Definition

Let $L_1, L_2, L \subseteq T^*$ be languages over the alphabet T.

1. The **concatenation** $L_1.L_2$ of L_1 and L_2 is defined as

$$L_1.L_2 := \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\}$$

2. The **union** $L_1 \mid L_2$ of L_1 and L_2 is defined as

$$L_1 \mid L_2 := L_1 \cup L_2$$

   The union is sometimes denoted by $+$.

3. The **iteration** or **Kleene-star** L^* of L is defined as

$$L^* := \{w_1 w_2 \cdots w_n \mid n \geq 0, w_1, \ldots, w_n \in L\}$$

Regular Expressions

Regular expressions are denotations for languages formed from the $\emptyset$ and from the languages $\{a\}$, where a is an element of the alphabet, by using the above mentioned operations.

Regular Expressions

Definition

Let T be an alphabet. We define the set of **regular expressions** over an alphabet T inductively together with the language $L(E)$ for each regular expression E.

- $\emptyset$ is a regular expression, $L(\emptyset) := \emptyset$.
- ϵ is a regular expression, $L(\epsilon) := \{\epsilon\}$.
- For $a \in T$ we have a is a regular expression, $L(a) := \{a\}$. One usually writes $\mathbf{a}$ for the regular expression, when the symbol is a.
- If E, F are regular expressions, then
  - $(E) \mid (F)$ is a regular expression, $L((E) \mid (F)) := L(E) \cup L(F)$.
  - $(E)(F)$ is a regular expression, $L((E)(F)) = L(E).L(F)$.
  - $(E)^*$ is a regular expression, $L((E)^*) = L(E)^*$.

We omit unnecessary brackets and usually write $E \mid F$ instead of $(E) \mid (F)$, EF instead of $(E)(F)$, E^* instead of $(E)^*$, if there is no confusion.

Use of Regular Expressions

- We will usually omit writing $L(E)$, so write

$$(\mathbf{0}\ \mathbf{1})\ \mathbf{0}^*$$

instead of

$$L((\mathbf{0}\ \mathbf{1})\ \mathbf{0}^*)$$

which is

$$(\{0\}.\{1\}).(\{0\})^*$$

which is

$$\{010^n \mid n \in \mathbb{N}\} \text{ or } \{01\ \underbrace{0 \cdots 0}_{n \text{ times}} \mid n \in \mathbb{N}\}$$

- We will as well identify regular expressions which denote the same language. Therefore we can omit more brackets e.g. we can write

$$\mathbf{0\ 1\ 0}$$

instead of

$$(\mathbf{0}\ \mathbf{1})\ \mathbf{0}$$

Use of Regular Expressions

- If the alphabet only contains single characters, we can omit the blank in concatenation, and write

$$\mathbf{010} \text{ instead of } \mathbf{0\ 1\ 0}$$

- * only refers to the last item, unless there are brackets:
  - $\mathbf{01}^* = \{0(1^n) \mid n \in \mathbb{N}\}$
  - $(\mathbf{01})^* = \{(01)^n \mid n \in \mathbb{N}\}$

Examples of Regular Expressions

- The set of non-zero digits is defined as

$$\mathit{NonzeroDigit} = \mathbf{1} \mid \mathbf{2} \mid \cdots\cdots \mid \mathbf{9}$$

- The set of digits is defined as

$$\mathit{Digit} = \mathbf{0} \mid \mathit{NonZeroDigit}$$

- The set of numbers without leading zero is

$$\mathit{Number} = \mathbf{0} \mid (\mathit{NonZeroDigit}\ \mathit{Digit}^*)$$

- The set of capital letters is defined by

$$\mathit{CapitalLetter} = \mathbf{A} \mid \mathbf{B} \mid \cdots \mid \mathbf{Z}$$

Examples of Regular Expressions

- The set of module codes in this department is

$$\textit{CSModuleCodes} = \textbf{CS}\ (- \mid \textbf{C} \mid \textbf{F} \mid \textbf{P})\ (\textbf{0} \mid \textbf{1} \mid\ \ \textbf{2} \mid \textbf{3} \mid \textbf{M})\ \textit{Digit}\ \textit{Digit}$$

Examples of Regular Expressions

- The set of postcodes can be defined as

$$\begin{aligned} \mathit{postcode} = {} & \mathit{CapitalLetter}\ \mathit{CapitalLetter}\ \mathit{Digit}\ \llcorner\lrcorner \\ & \mathit{Digit}\ \mathit{CapitalLetter}\ \mathit{CapitalLetter} \end{aligned}$$

Regular Expressions are Non-recursive

- Please note that regular expressions are non-recursive.
  For instance in

$$\begin{aligned} \mathit{postcode} = {} & \mathit{CapitalLetter}\ \mathit{CapitalLetter}\ \mathit{Digit}\ \llcorner\lrcorner \\ & \mathit{Digit}\ \mathit{CapitalLetter}\ \mathit{CapitalLetter} \end{aligned}$$

  "postcode" doesn't occur on the right-hand side.
- Note that this is different from grammars (including BNF) where recursion is allowed.
  For instance we can have productions such as

$$S \longrightarrow aSa$$

  or in BNF

$$\langle \mathrm{S} \rangle ::= \mathbf{a} \langle \mathrm{S} \rangle \mathbf{a}$$

Regular Expressions in Programming

- Regular Expressions occur very often in programming.
- They occur in
  - Linux/Unix (command grep/egrep),
  - in scripting languages (Perl, Python, Ruby),
  - (one of the main innovations of Ruby over Python was an improved notation $\sim$ for matching of regular expressions),
- in SQL,
- are supported in most programming languages by libraries.

Notations for Regular Expressions

- One writes $[a_1 \cdots a_n]$ for $a_1 \mid \cdots \mid a_n$.
- One writes $[a - z]$ for $[a, b, c, \ldots z]$ similarly for $[0 - 9]$, $[A - Z]$.
- $[a - zA - Z] := [a - z]|[A - Z]$, $[a - z?] := [a - z]|?$ etc.
- One writes L^+ or $L+$ for $L\ L^*$ (so $L^+ := \{w_1 \cdots w_n \mid n \geq 1, w_1, \ldots, w_n \in L\}$, the set of words formed from L by using at least one word in L.
  - **Question:** Is L^+ the set of non-empty words formed from elements of L^*?
    Answer:
- Lots of other useful operators for constructing regular expressions have been defined.
- Each language has its own set and of regular expressions (using often different notations), and its own syntax. Sometimes operators are introduced which go beyond regular languages.

Example Use of Regular Expressions

- Assume you have files called automatatheorych1.tex, automatatheorych2.tex, automatatheorych3.tex ,. . .
  Concatenation all of them into one file:

```
cssetzer@cs-svr1:> cat automatatheory[0-9].tex >
                                automatatheoryall.tex
```

- Process lines in a file containing entries separated by ",", do something if the first field is a student number (a string consisting of digits only). Python code

```
file = open(filename)
regExpStud = re.compile('^[0-9]*$')
for line in file:
    a = line.split(',')
    if regExpStud.match(a[0]):
        print a[1][:-1] #cut off trailing '\n'
file.close()
```

Example Web Addresses

Consider links in http pages of the form:

Swansea University

displayed as

Swansea University

The set of weblinks can be defined as
(in most language ␣ would be written as a blank, " would be preceded by a \, and the whole string would be put into quotation marks):

$$\text{webLinks} = \text{<a␣href="[a-zA-Z0-9/. :]*">[a-zA-Z0-9␣]*}$$

E.g. in Python one would write

weblinks="[a-zA-Z0-9]*"

Usage of Regular Expressions in Computer Security

- In computer security one very often needs to check for occurrences of certain patterns.
- For instance in order to locate a certain virus, which might consist of 3 pieces of code s_1, s_2, s_3, separated by some normal code, one could search for the regular expression

$$s_1[a-z]^* s_2[a-z]^* s_3$$

(How do you obtain that s_1, s_2 and s_3 might occur in different order?)

- Of course in general you need to check for much more sophisticated patterns.

Usage of Regular Expressions in Computer Security

- In order to check that a password is safe enough, which might mean it consists of digits and lower case characters, and at least one digit and one lower case character, would mean that you whether it matches

$$(([a-z] \mid [0-9])^*[a-z]([a-z] \mid [0-9])^*[0-9]([a-z] \mid [0-9])^*) \mid$$
$$(([a-z] \mid [0-9])^*[0-9]([a-z] \mid [0-9])^*[a-z]([a-z] \mid [0-9])^*)$$

  Of course you would usually use a much more sophisticated regular expression.

Usage of Regular Expressions in Computer Security

- Detecting in request certain malicious patterns in requests from the outside can often be expressed as a regular expression and you search for matches in this income stream which match that expression.

SQL Injection

- Regular expression can be used to detect attempts of SQL injection.
- Example of SQL injection (from Wikipedia on SQL Injection):
  Assume the following statement in a code

statement =
"SELECT * FROM users WHERE username = ' " + username +
" ' and password = ' " + password + " ';"

- This statement is supposed to be sent to the SQL server.
  Then one checkes the resulting entries for whether the supplied password matches modulo encryption one of the password entries for that user name.

SQL Injection

- Assume an attacker tries to login with password

  ' or '1'='1

  Then the statement sent to the SQL server will be

  SELECT * FROM users WHERE username = 'username'
  and passport =" or '1'='1';

  which matches all users with username.

SQL Injection

- This might allow you to check whether your password matches any user, which makes it more likely to get a match and allow you to login.
- In order to avoid such kind of attack you can check whether the username matches any malicious pattern.
  Such patterns can be expressed by regular expressions.

Usage of Regular Expressions in Computer Security

- The above were just some (here very simple) examples how regular expressions can be used to detect in computer security certain patterns corresponding to attacks or weaknesses of a system.

Closure of Regular Languages

The main lemma for showing that regular expressions define regular languages is as follows:

Lemma (II.2.2.1.)

Let G, G' be both left-linear grammars or both right-linear grammars. Then we can define a left-linear or right-linear grammars G_i s.t.

1. $L(G_1) = L(G) \mid L(G')$,
2. $L(G_2) = L(G).L(G')$,
3. $L(G_3) = L(G)^*$.

These grammars can be computed from G and G'.

Proof

A proof can be found in the additional material for this subsection.

Regular Expressions define Regular Languages

Lemma (II.2.2.2.)

Let E be a regular Expression. Then there exist both left-linear and right-linear grammars G, G' s.t.

$$L(E) = L(G) = L(G')$$

G and G' can be computed from L.

Proof: By Lemma II.2.2.1, and the fact that the finite languages $\emptyset$, $\{\epsilon\}$ and $\{a\}$ are regular.
Full details can be found in Additional Material.