II.2.1. Regular Languages (12.2)

Finite Languages are Regular

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

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

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 $G_{\text{left-linear}, a^m b^n}$

terminals $a, b$

nonterminals $S, T$

start symbol $S$

productions

$S \rightarrow Sb$
$S \rightarrow Tb$
$T \rightarrow Ta$
$T \rightarrow a$
A Right-Linear Grammar for $a^m b^n$

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

<table>
<thead>
<tr>
<th>grammar</th>
<th>$G_{\text{right-linear}, a^m b^n}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminals</td>
<td>$a, b$</td>
</tr>
<tr>
<td>nonterminals</td>
<td>$S, T$</td>
</tr>
<tr>
<td>start symbol</td>
<td>$S$</td>
</tr>
<tr>
<td>productions</td>
<td>$S \rightarrow aS$</td>
</tr>
<tr>
<td></td>
<td>$S \rightarrow aT$</td>
</tr>
<tr>
<td></td>
<td>$T \rightarrow bT$</td>
</tr>
<tr>
<td></td>
<td>$T \rightarrow b$</td>
</tr>
</tbody>
</table>

Right-Linear Grammar for Numbers

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

<table>
<thead>
<tr>
<th>grammar</th>
<th>$G_{\text{Number}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminals</td>
<td>0, 1, ..., 9</td>
</tr>
<tr>
<td>nonterminals</td>
<td>Number, Digits</td>
</tr>
<tr>
<td>start symbol</td>
<td>Number</td>
</tr>
<tr>
<td>productions</td>
<td>Number $\rightarrow$ 0</td>
</tr>
<tr>
<td></td>
<td>Number $\rightarrow$ 1 Digits</td>
</tr>
<tr>
<td></td>
<td>Digits $\rightarrow$ 0 Digits</td>
</tr>
<tr>
<td></td>
<td>Digits $\rightarrow$ $\epsilon$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>grammar</th>
<th>$G_{\text{Number}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminals</td>
<td>0, 1, ..., 9</td>
</tr>
<tr>
<td>nonterminals</td>
<td>Number, Digit, NonZeroDigit, Digits</td>
</tr>
<tr>
<td>start symbol</td>
<td>Number</td>
</tr>
<tr>
<td>productions</td>
<td>Number $\rightarrow$ Digit</td>
</tr>
<tr>
<td></td>
<td>Digits $\rightarrow$ Digit Digits</td>
</tr>
<tr>
<td></td>
<td>Digit $\rightarrow$ 0</td>
</tr>
<tr>
<td></td>
<td>NonZeroDigit $\rightarrow$ 1</td>
</tr>
</tbody>
</table>

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 $\sqsubseteq$ for blank.
Right-Linear Grammar for Post Codes

<table>
<thead>
<tr>
<th>grammar</th>
<th>$G_{Postcode}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminals</td>
<td>0, 1, ..., 9, A, B, ..., Z, ⌞⌟</td>
</tr>
<tr>
<td>nonterminals</td>
<td>postcode, letter2, digit1, blank1, digit2, letter3, letter4</td>
</tr>
<tr>
<td>start symbol</td>
<td>postcode</td>
</tr>
<tr>
<td>productions</td>
<td>postcode $\rightarrow$ A letter2</td>
</tr>
<tr>
<td></td>
<td>letter2 $\rightarrow$ A digit1</td>
</tr>
<tr>
<td></td>
<td>digit1 $\rightarrow$ 0 blank1</td>
</tr>
<tr>
<td></td>
<td>blank1 $\rightarrow$ ⌞⌟ digit2</td>
</tr>
<tr>
<td></td>
<td>digit2 $\rightarrow$ 0 letter3</td>
</tr>
<tr>
<td></td>
<td>letter3 $\rightarrow$ A letter4</td>
</tr>
<tr>
<td></td>
<td>letter4 $\rightarrow$ A</td>
</tr>
</tbody>
</table>

Example Derivation

Here is a derivation of $SA_{2\ldots}8PP \in L(G_{Postcode})$:

```
postcode $\Rightarrow$ S letter2
   $\Rightarrow$ SA digit1
   $\Rightarrow$ SA1 blank1
   $\Rightarrow$ SA1\ldots digit2
   $\Rightarrow$ SA1\ldots 8 letter3
   $\Rightarrow$ SA1\ldots 8P letter4
   $\Rightarrow$ SA1\ldots 8PP
```

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
  $$
  S \rightarrow abB \\
  B \rightarrow aS \\
  B \rightarrow baS
  $$

  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
   \[ A \rightarrow Bw \text{ or } A \rightarrow w \]
   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
   \[ A \rightarrow wB \text{ or } A \rightarrow w \]
   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 $G$.

Proof Idea for Lemma II.2.1.2.

- First omit all so called silent productions, i.e. productions of the form $A \rightarrow B$ for some non-terminals $A, B$.
  - This requires some work.
- Then replace in the right-linear case productions
  \[ A \rightarrow a_1a_2\cdots a_nB \]
  with $n \geq 2$ by productions
  \[
  \begin{align*}
  A & \rightarrow a_1A_1, \\
  A_1 & \rightarrow a_2A_2, \\
  \vdots & \\
  A_{n-1} & \rightarrow a_nB
  \end{align*}
  \]
  for some new nonterminals $A_i$.
- Full details can be found in the additional material.

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.

<table>
<thead>
<tr>
<th>grammar</th>
<th>$G$</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminals</td>
<td>$a, b$</td>
</tr>
<tr>
<td>nonterminals</td>
<td>$S, T$</td>
</tr>
<tr>
<td>start symbol</td>
<td>$S$</td>
</tr>
</tbody>
</table>
| productions | $S \rightarrow ab$
| | $S \rightarrow aT$
| | $T \rightarrow Sb$ |

Operators for Forming Languages

Definition

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

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

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

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

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

   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 finite sets 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, $L(\emptyset) := \emptyset$.
- $\epsilon$ is a regular expression, $L(\epsilon) := \{\epsilon\}$.
- For $a \in T$ we have $a$ is a regular expression, $L(a) := \{a\}$. One usually writes $a$ for the regular expression, when the symbol is $a$.
- If $E, F$ are regular expressions, then
  - $(E) (F)$ 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 $(0 \ 1) \ 0^*$ instead of $L((0 \ 1) \ 0^*)$ which is $\{(0)\}.\{1\}.\{0\)^*$

- We will as well identify regular expressions which denote the same language. Therefore we can omit more brackets e.g. we can write $0 \ 1 \ 0$ instead of $(0 \ 1) \ 0$

Remark Regarding Previous Years

When teaching CS-236 2009/10, regular expressions were constructed directly from $\emptyset$, $\{\epsilon\}$, $\{a\}$ using $\ , \mid , \ast$. So

- We write now $\epsilon$ instead of $\{\epsilon\}$.
- We write now $a$ instead of $\{a\}$.
- We write now $EF$ instead of $E \ast F$. 

- If the alphabet only contains single characters, we can omit the blank in concatenation, and write $010$ instead of $0 \ 1 \ 0$
Examples of Regular Expressions

- The set of non-zero digits is defined as
  \[ \text{NonzeroDigit} = 1 \mid 2 \mid \cdots \mid 9 \]

- The set of digits is defined as
  \[ \text{Digit} = 0 \mid \text{NonZeroDigit} \]

- The set of numbers without leading zero is
  \[ \text{Number} = 0 \mid (\text{NonZeroDigit} \; \text{Digit}^*) \]

- The set of capital letters is defined by
  \[ \text{CapitalLetter} = A \mid B \mid \cdots \mid Z \]

Examples of Regular Expressions

- The set of module codes in this department is
  \[ \text{CSModuleCodes} = \text{CS} - (0 \mid 1 \mid 2 \mid 3 \mid M) \; \text{Digit} \; \text{Digit} \]

Examples of Regular Expressions

- The set of postcodes can be defined as
  \[ \text{postcode} = \text{CapitalLetter} \; \text{CapitalLetter} \; \text{Digit} \; \text{Digit} \; \text{CapitalLetter} \; \text{CapitalLetter} \]

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 $[a \cdots z]$ for $[a, b, c, \ldots z]$ similarly for $[0 \cdots 9]$.
- 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`, ... Concatenate 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

  ```python
  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()
  ```

Closure of Regular Languages

In order to show that all regular expressions are regular we first show the following

**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. $L(G_3) = L(G) \ast$.

These grammars can be computed from $G$ and $G'$.

Proof

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