3.15: Applications of Finite Automata and Regular Expressions

In this section we consider three applications of the material from Chapter 3:

- searching for regular expressions in files;
- lexical analysis; and
- the design of finite state systems.

Representing Character Sets and Files

Our first two applications involve processing files whose characters come from some character set, e.g., the ASCII character set.

Although not every character in a typical character set will be an element of our set **Sym** of symbols, we can *represent* all the characters of a character set by elements of **Sym**. E.g., we might represent the ASCII characters newline and space by the symbols $\langle\mathsf{newline}\rangle$ and $\langle\mathsf{space}\rangle$, respectively.

So, we will work with a mostly unspecified alphabet Σ representing some character set. We assume that the symbols 0–9, a–z, A–Z, $\langle\mathsf{space}\rangle$ and $\langle\mathsf{newline}\rangle$ are elements of Σ. A *line* is a string consisting of an element of $(\Sigma - \{\langle\mathsf{newline}\rangle\})^*$; and, a *file* consists of the concatenation of some number of lines, separated by occurrences of $\langle\mathsf{newline}\rangle$.

Representing Character Sets and Files

In what follows, we write:

- **[any]** for the regular expression $a_1 + a_2 + \cdots + a_n$, where $a_1, a_2, \ldots, a_n$ are all of the elements of Σ except $\langle\text{newline}\rangle$, listed in the standard order;
- **[letter]** for the regular expression

$$\mathsf{a} + \mathsf{b} + \cdots + \mathsf{z} + \mathsf{A} + \mathsf{B} + \cdots + \mathsf{Z}; \text{ and}$$

- **[digit]** for the regular expression

$$0 + 1 + \cdots + 9.$$

Searching for Regular Expression in Files

Given a file and a regular expression α whose alphabet is a subset of $\Sigma - \{\langle\text{newline}\rangle\}$, how can we find all lines of the file with substrings in $L(\alpha)$? (E.g., α might be $\mathsf{a}(\mathsf{b} + \mathsf{c})^*\mathsf{a}$; then we want to find all lines containing two a's, separated by some number of b's and c's.)

It will be sufficient to find all lines in the file that are elements of $L(\beta)$, where $\beta = [\textbf{any}]^*\,\alpha\,[\textbf{any}]^*$.

To do this, we can first translate β to a DFA M with alphabet $\Sigma - \{\langle\text{newline}\rangle\}$. For each line w, we simply check whether $\delta_M(s_M, w) \in A_M$, selecting the line if it is.

If the file is short, however, it may be more efficient to convert β to an FA N, and use the algorithm from Section 3.6 to find all lines that are accepted by N.

Lexical Analysis

A lexical analyzer is the part of a compiler that groups the characters of a program into lexical items or tokens. The modern approach to specifying a lexical analyzer for a programming language uses regular expressions. E.g., this is the approach taken by the lexical analyzer generator Lex.

Lexical Analzyer Specifications

A lexical analyzer specification consists of a list of regular expressions $\alpha_1, \alpha_2, \ldots, \alpha_n$, together with a corresponding list of code fragments (in some programming language) $\mathit{code}_1, \mathit{code}_2, \ldots, \mathit{code}_n$ that process elements of Σ^*.

For example, we might have

$$\begin{aligned}
\alpha_1 &= \langle \mathsf{space} \rangle + \langle \mathsf{newline} \rangle, \\
\alpha_2 &= [\textbf{letter}]\,([\textbf{letter}] + [\textbf{digit}])^*, \\
\alpha_3 &= [\textbf{digit}]\,[\textbf{digit}]^*\,(\% + \mathsf{E}\,[\textbf{digit}]\,[\textbf{digit}]^*), \\
\alpha_4 &= [\textbf{any}].
\end{aligned}$$

The elements of $L(\alpha_1)$, $L(\alpha_2)$ and $L(\alpha_3)$ are whitespace characters, identifiers and numerals, respectively. The code associated with α_4 will probably indicate that an error has occurred.

Lexical Analzyer Specifications

A lexical analyzer meets such a specification iff it behaves as follows.

At each stage of processing its file, the lexical analyzer should consume the *longest* prefix of the remaining input that is in the language generated by one of the regular expressions.

It should then supply the prefix to the code associated with the earliest regular expression whose language contains the prefix.

However, if there is no such prefix, or if the prefix is %, then the lexical analyzer should indicate that an error has occurred.

Lexical Analyzer Specifications

What happens when we process the file
123Easy⟨space⟩1E2⟨newline⟩ using a lexical analyzer meeting our example specification?

The longest prefix of 123Easy⟨space⟩1E2⟨newline⟩ that is in one of our regular expressions is 123. Since this prefix is only in α_3, it is consumed from the input and supplied to code_3.

The remaining input is now Easy⟨space⟩1E2⟨newline⟩. The longest prefix of the remaining input that is in one of our regular expressions is Easy. Since this prefix is only in α_2, it is consumed and supplied to code_2.

The remaining input is then ⟨space⟩1E2⟨newline⟩. The longest prefix of the remaining input that is in one of our regular expressions is ⟨space⟩. Since this prefix is only in α_1 and α_4, we consume it from the input and supply it to the code associated with the earlier of these regular expressions: code_1.

Lexical Analzyer Specifications

The remaining input is then 1E2⟨newline⟩. The longest prefix of the remaining input that is in one of our regular expressions is 1E2. Since this prefix is only in α_3, we consume it from the input and supply it to code_3.

The remaining input is then ⟨newline⟩. The longest prefix of the remaining input that is in one of our regular expressions is ⟨newline⟩. Since this prefix is only in α_1, we consume it from the input and supply it to the code associated with this expression: code_1.

The remaining input is now empty, and so the lexical analyzer terminates.

Generating Lexical Analyzers from Specifications

What is a simple method for generating a lexical analyzer that meets a given specification? (More sophisticated methods are described in compilers courses.)

First, we convert the regular expressions $\alpha_1, \ldots, \alpha_n$ into DFAs $M_1, \ldots, M_n$. Next we determine which of the states of the DFAs are dead/live.

Generating Lexical Analyzers from Specifications

Given its remaining input x, the lexical analyzer consumes the next token from x and supplies the token to the appropriate code, as follows.

First, it initializes the following variables to error values:

- a string variable acc, which records the longest prefix of the prefix of x that has been processed so far that is accepted by one of the DFAs;
- an integer variable $mach$, which records the smallest i such that $acc \in L(M_i)$; and
- a string variable aft, consisting of the suffix of x that one gets by removing acc.

Then, the lexical analyzer enters its main loop, in which it processes x, symbol by symbol, in *each* of the DFAs, keeping track of what symbols have been processed so far, and what symbols remain to be processed.

Main Loop

If, after processing a symbol, at least one of the DFAs is in an accepting state, then the lexical analyzer stores the string that has been processed so far in the variable acc, stores the index of the first machine to accept this string in the integer variable $mach$, and stores the remaining input in the string variable aft.

If there is no remaining input, then the lexical analyzer supplies acc to code $code_{mach}$, and returns; otherwise it continues.

Main Loop

If, after processing a symbol, none of the DFAs are in accepting states, but at least one automaton is in a live state (so that, without knowing anything about the remaining input, it's possible that an automaton will again enter an accepting state), then the lexical analyzer leaves *acc*, *mach* and *aft* unchanged.

If there is no remaining input, the lexical analyzer supplies *acc* to $code_{mach}$ (it signals an error if *acc* is still set to the error value), resets the remaining input to *aft*, and returns; otherwise, it continues.

Main Loop

If, after processing a symbol, all of the automata are in dead states (and so could never enter accepting states again, no matter what the remaining input was), the lexical analyzer supplies string acc to code $code_{mach}$ (it signals an error if acc is still set to the error value), resets the remaining input to aft, and returns.

Example

Let's see what happens when the file 123Easy$\langle$newline$\rangle$ is processed by the lexical analyzer generated from our example specification.

- After processing 1, M_3 and M_4 are in accepting states, and so the lexical analyzer sets *acc* to 1, *mach* to 3, and *aft* to 23Easy$\langle$newline$\rangle$. It then continues.
- After processing 2, so that 12 has been processed so far, only M_3 is in an accepting state, and so the lexical analyzer sets *acc* to 12, *mach* to 3, and *aft* to 3Easy$\langle$newline$\rangle$. It then continues.
- After processing 3, so that 123 has been processed so far, only M_3 is in an accepting state, and so the lexical analyzer sets *acc* to 123, *mach* to 3, and *aft* to Easy$\langle$newline$\rangle$. It then continues.

Example

- After processing E, so that 123E has been processed so far, none of the DFAs are in accepting states, but M_3 is in a live state, since 123E is a prefix of a string that is accepted by M_3. Thus the lexical analyzer continues, but doesn't change acc, $mach$ or aft.
- After processing a, so that 123Ea has been processed so far, all of the machines are in dead states, since 123Ea isn't a prefix of a string that is accepted by one of the DFAs. Thus the lexical analyzer supplies $acc = 123$ to $code_{mach} = code_3$, and sets the remaining input to $aft = \text{Easy}\langle\text{newline}\rangle$.
- In subsequent steps, the lexical analyzer extracts Easy from the remaining input, and supplies this string to code $code_2$, and extracts $\langle\text{newline}\rangle$ from the remaining input, and supplies this string to code $code_1$.

Design of Finite State Systems

Deterministic finite automata give us a means to efficiently check membership in a regular language.

In terms of time, a single left-to-right scan of the string is needed. And we only need enough space to encode the DFA, and to keep track of what state we are in at each point, as well as what part of the string remains to be processed.

But if the string to be checked is supplied, symbol-by-symbol, from our environment, we don't need to store the string at all.

Consequently, DFAs may be easily and efficiently implemented in both hardware and software.

One can design DFAs by hand, and test them using Forlan.

But DFA minimization plus the operations on automata and regular expressions of Section 3.12, give us an alternative—and very powerful—way of designing finite state systems.

First Example

As the first example, suppose we wish to find a DFA M such that $L(M) = X$, where $X = \{\, w \in \{0,1\}^* \mid w$ has an even number of 0's or an odd number of 1's $\}$.

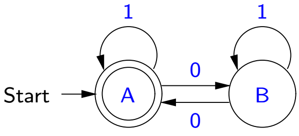
First, we can note that $X = Y_1 \cup Y_2$, where

$$Y_1 = \{\, w \in \{0,1\}^* \mid w \text{ has an even number of } 0\text{'s} \,\}, \text{ and}$$
$$Y_2 = \{\, w \in \{0,1\}^* \mid w \text{ has an odd number of } 1\text{'s} \,\}.$$

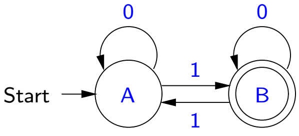
Since we have a union operation on EFAs (Forlan doesn't provide a union operation on DFAs), if we can find EFAs accepting Y_1 and Y_2, we can combine them into a EFA that accepts X. Then we can convert this EFA to a DFA, and then minimize the DFA.

First Example

Let N_1 and N_2 be the DFAs



(N_1)



(N_2)

It is easy to prove that $L(N_1) = Y_1$ and $L(N_2) = Y_2$.

Let M be the DFA

$$\textbf{renameStatesCanonically}(\textbf{minimize}\, N),$$

where N is the DFA

$$\textbf{nfaToDFA}(\textbf{efaToNFA}(\textbf{union}(N_1, N_2))).$$

First Example

Then

$$\begin{aligned}
L(M) &= L(\textbf{renameStatesCanonically}(\textbf{minimize}\, N)) \\
&= L(\textbf{minimize}\, N) \\
&= L(N) \\
&= L(\textbf{nfaToDFA}(\textbf{efaToNFA}(\textbf{union}(N_1, N_2)))) \\
&= L(\textbf{efaToNFA}(\textbf{union}(N_1, N_2))) \\
&= L(\textbf{union}(N_1, N_2)) \\
&= L(N_1) \cup L(N_2) \\
&= Y_1 \cup Y_2 \\
&= X,
\end{aligned}$$

showing that M is correct.

First Example

But how do we figure out what the components of M are, so that, e.g., we can draw M?

In a simple case like this, we could apply the definitions **union**, **efaToNFA**, **nfaToDFA**, **minimize** and **renameStatesCanonically**, and work out the answer.

First Example

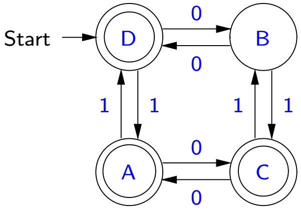
Instead, we can use Forlan to compute the answer. Suppose dfa1 and dfa2 of type dfa are N_1 and N_2, respectively. The we can proceed as follows:

```
- val efa =
=       EFA.union(injDFAToEFA dfa1, injDFAToEFA dfa2);
val efa = - : efa
- val dfa' = nfaToDFA(efaToNFA efa);
val dfa' = - : dfa
- DFA.numStates dfa';
val it = 5 : int
- val dfa =
=       DFA.renameStatesCanonically
=       (DFA.minimize dfa');
val dfa = - : dfa
- DFA.numStates dfa;
val it = 4 : int
```

First Example

```
- DFA.output("", dfa);
{states} A, B, C, D {start state} D
{accepting states} A, C, D
{transitions}
A, 0 -> C; A, 1 -> D; B, 0 -> D; B, 1 -> C; C, 0 -> A;
C, 1 -> B; D, 0 -> B; D, 1 -> A
val it = () : unit
```

Thus M is:



Of course, this claim assumes that Forlan is correctly implemented.

Second Example

Given a string $w \in \{0,1\}^*$, we say that:

- w *stutters* iff aa is a substring of w, for some $a \in \{0,1\}$; and
- w is *long* iff $|w| \geq 5$.

So, e.g., 1001 and 10110 both stutter, but 01010 and 101 don't.

Let the language **AllLongStutter** be

$$\{\, w \in \{0,1\}^* \mid \text{for all substrings } v \text{ of } w, \text{ if } v \text{ is long, then } v \text{ stutters} \,\}.$$

Since every substring of 0010110 of length five stutters, every long substring of this string stutters, and thus the string is in **AllLongStutter**.

On the other hand, 0010100 is not in **AllLongStutter**, because 01010 is a long, non-stuttering substring of this string.

Second Example

Let's consider the problem of finding a DFA that accepts this language.

One possibility is to reduce this problem to that of finding a DFA that accepts the complement of **AllLongStutter**. Then we'll be able to use our set difference operation on DFAs to build a DFA that accepts **AllLongStutter**, which we can then minimize.

To form the complement of **AllLongStutter**, we negate the formula in **AllLongStutter**'s expression. Let **SomeLongNotStutter** be the language

$$\{\, w \in \{0,1\}^* \mid \text{there is a substring } v \text{ of } w \text{ such that } v \text{ is long and doesn't stutter} \,\}.$$

Lemma 3.15.1
AllLongStutter $= \{0,1\}^* -$ **SomeLongNotStutter**.

Second Example

Next, it's convenient to work bottom-up for a bit. Let

$$\begin{aligned}
\textbf{Long} &= \{\, w \in \{0,1\}^* \mid w \text{ is long} \,\}, \\
\textbf{Stutter} &= \{\, w \in \{0,1\}^* \mid w \text{ stutters} \,\}, \\
\textbf{NotStutter} &= \{\, w \in \{0,1\}^* \mid w \text{ doesn't stutter} \,\}, \text{ and} \\
\textbf{LongAndNotStutter} &= \{\, w \in \{0,1\}^* \mid w \text{ is long and doesn't stutter} \,\}.
\end{aligned}$$

The following lemma is easy to prove:

Lemma 3.15.2

(1) $\textbf{NotStutter} = \{0,1\}^* - \textbf{Stutter}$.

(2) $\textbf{LongAndNotStutter} = \textbf{Long} \cap \textbf{NotStutter}$.

Second Example

Clearly, we'll be able to find DFAs accepting **Long** and **Stutter**, respectively. Thus, we'll be able to use our set difference operation on DFAs to come up with a DFA that accepts **NotStutter**. Then, we'll be able to use our intersection operation on DFAs to come up with a DFA that accepts **LongAndNotStutter**.

What remains is to find a way of converting **LongAndNotStutter** to **SomeLongNotStutter**. Clearly, the former language is a subset of the latter one. But the two languages are not equal, since an element of the latter language may have the form xvy, where $x, y \in \{0, 1\}^*$ and $v \in$ **LongAndNotStutter**.

This suggests the following lemma:

Lemma 3.15.3
SomeLongNotStutter $= \{0, 1\}^*$ **LongAndNotStutter** $\{0, 1\}^*$.

Second Example

Because of the preceding lemma, we can construct an EFA accepting **SomeLongNotStutter** from a DFA accepting $\{0, 1\}^*$ and our DFA accepting **LongAndNotStutter**, using our concatenation operation on EFAs. We can then convert this EFA to a DFA.

Now we'll turn these ideas into reality, mirroring operations on languages with the corresponding operations on regular expressions and finite automata.

The book first shows how our DFA can be constructed and proved correct.

But we'll skip directly to constructing the DFA in Forlan.

Second Example

We put the following code in the file `stutter1.sml`:

```
val regToEFA  = faToEFA o regToFA;
val efaToDFA  = nfaToDFA o efaToNFA;
val regToDFA  = efaToDFA o regToEFA;
val minAndRen =
      DFA.renameStatesCanonically o DFA.minimize;

val allStrReg = Reg.fromString "(0 + 1)*";
val allStrDFA = minAndRen(regToDFA allStrReg);
val allStrEFA = injDFAToEFA allStrDFA;

val longReg =
      Reg.concat
      (Reg.power(Reg.fromString "0 + 1", 5),
       Reg.fromString "(0 + 1)*");
val longDFA = minAndRen(regToDFA longReg);
```

Second Example

We put the following code in the file `stutter2.sml`:

```
val stutterReg =
      Reg.fromString "(0 + 1)*(00 + 11)(0 + 1)*";
val stutterDFA = minAndRen(regToDFA stutterReg);

val notStutterDFA =
      minAndRen(DFA.minus(allStrDFA, stutterDFA));

val longAndNotStutterDFA =
      minAndRen(DFA.inter(longDFA, notStutterDFA));
val longAndNotStutterEFA =
      injDFAToEFA longAndNotStutterDFA;
```

Second Example

And, we put the following code in the file `stutter3.sml`:

```
val someLongNotStutterEFA' =
      EFA.concat
      (allStrEFA,
       EFA.concat(longAndNotStutterEFA,
                  allStrEFA));
val someLongNotStutterEFA  =
      EFA.renameStatesCanonically someLongNotStutterEFA';

val someLongNotStutterDFA =
      minAndRen(efaToDFA someLongNotStutterEFA);
val allLongStutterDFA     =
      minAndRen
      (DFA.minus(allStrDFA, someLongNotStutterDFA));
```

Second Example

Then, we proceed as follows:

```
- use "stutter1.sml";
[opening stutter1.sml]
val regToEFA = fn : reg -> efa
val efaToDFA = fn : efa -> dfa
val regToDFA = fn : reg -> dfa
val minAndRen = fn : dfa -> dfa
val allStrReg = - : reg
val allStrDFA = - : dfa
val allStrEFA = - : efa
val longReg = - : reg
val longDFA = - : dfa
val it = () : unit
```

Second Example

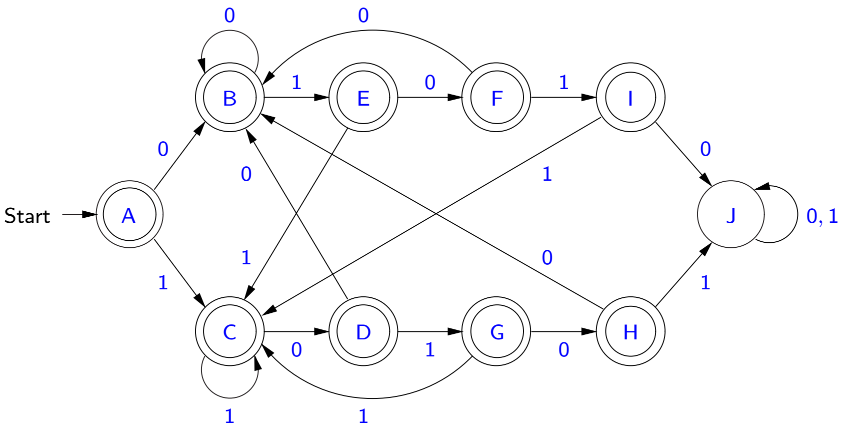
```
- use "stutter2.sml";
[opening stutter2.sml]
val stutterReg = - : reg
val stutterDFA = - : dfa
val notStutterDFA = - : dfa
val longAndNotStutterDFA = - : dfa
val longAndNotStutterEFA = - : efa
val it = () : unit
- use "stutter3.sml";
[opening stutter3.sml]
val someLongNotStutterEFA' = - : efa
val someLongNotStutterEFA = - : efa
val someLongNotStutterDFA = - : dfa
val allLongStutterDFA = - : dfa
val it = () : unit
```

Second Example

```
- DFA.output("", allLongStutterDFA);
{states} A, B, C, D, E, F, G, H, I, J {start state} A
{accepting states} A, B, C, D, E, F, G, H, I
{transitions}
A, 0 -> B; A, 1 -> C; B, 0 -> B; B, 1 -> E; C, 0 -> D;
C, 1 -> C; D, 0 -> B; D, 1 -> G; E, 0 -> F; E, 1 -> C;
F, 0 -> B; F, 1 -> I; G, 0 -> H; G, 1 -> C; H, 0 -> B;
H, 1 -> J; I, 0 -> J; I, 1 -> C; J, 0 -> J; J, 1 -> J
val it = () : unit
```

Second Example

Thus, **allLongStutterDFA** is


Start
A
B
C
D
E
F
G
H
I
J
0
1
0, 1