

Section 3.14: Applications of Finite Automata and Regular Expressions

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

- searching for regular expressions in files;
- lexical analysis.

Copyright © 2003–4 Alley Stoughton

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

The L^AT_EX source of these slides, the associated book, and the distribution of the Forlan toolset are available on the WWW at <http://people.cis.ksu.edu/~stough/forlan/>.

1

Representing Character Sets and Files

Both of our 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 `<newline>` and `<space>`, respectively.

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

2

Representing Character Sets and Files (Cont.)

In what follows, we write:

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

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

$$a + b + \cdots + z + A + B + \cdots + Z;$$

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

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

3

(3.14) 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 $a(b+c)^*a$; 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 $\beta = [\text{any}]^* \alpha [\text{any}]^* \langle \text{newline} \rangle$.

To do this, we can first translate β to a DFA M with alphabet Σ . 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 3.5 to find all lines that are accepted by N .

4

(3.14) 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.

5

(3.14) Lexical Analyzer Specifications

A lexical analyzer specification consists of a list of regular expressions $\alpha_1, \alpha_2, \dots, \alpha_n$, together with a corresponding list of code fragments (in some programming language) $code_1, code_2, \dots, code_n$ that process elements of Σ^* .

For example, we might have

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

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.

6

(3.14) Lexical Analyzer Specifications (Cont.)

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.

7

(3.14) Lexical Analyzer Specifications (Cont.)

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 `code3`.

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 `code2`.

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: `code1`.

8

(3.14) Lexical Analyzer Specifications (Cont.)

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 `code3`.

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: `code1`.

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

9

(3.14) 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 $\alpha_1, \dots, \alpha_n$ into DFAs M_1, \dots, M_n . Next we determine which of the states of the DFAs are dead/live.

(3.14) Generating Lexical Analyzers from Specifications (Cont.)

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 $acc \in L(M_i)$;
- a string variable aft , consisting of the suffix of x that one gets by removing acc .

11

(3.14) Generating Lexical Analyzers from Specifications (Cont.)

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.

12

(3.14) 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.

13

(3.14) Main Loop (Cont.)

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.

14

(3.14) Main Loop (Cont.)

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.

15

(3.14) Example

Let's see what happens when the file `123Easy<newline>` 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<newline>`. 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<newline>`. 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<newline>`. It then continues.

16

(3.14) Example (Cont.)

- 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* = **Easy**`<newline>`.
- In subsequent steps, the lexical analyzer extracts **Easy** from the remaining input, and supplies this string to code $code_2$, and extracts `<newline>` from the remaining input, and supplies this string to code $code_1$.