Homework #2: CMPT-755

Distributed on Mon, Sep 20; due on Mon, Sep 27

Anoop Sarkar – anoop@cs.sfu.ca

(1) This assignment will use DFA recognition to build an engine for lexical analysis. It will accept the definition of the tokens as a set of datafiles. In this assignment you will test your code on some simple but instructive token definitions that will help test the correctness of the lexical analysis engine.

Section 3.8 of the Dragon book lays out the architecture of a lexical analysis engine. The engine takes a specification of the tokens and uses a finite-state machine simulator to convert the input buffer into a series of tokens. Rather than specify a single large finite-state machine for all the tokens (like we assumed in our previous assignment), it is easier to specify a pattern (or a regular expression, or a finite-state machine) for each token.

In abstract terms, we can define a set of tokens (`T_A, T_B, T_C`) for each pattern p_i:

`T_A`	p_1
`T_B`	p_2
`T_C`	p_3

One particular way to define such a specification is with the use of regular expressions (we will use this specification from the Dragon book, Example 3.18, as a running example):

`T_A`	a
`T_B`	abb
`T_C`	a^*b^+

The lexical analysis engine should always pick the longest match possible, and in case of two patterns matching a prefix of the input of equal length, we break the tie by picking the pattern that was listed first in the specification (e.g. the token `T_B` is preferred over `T_C` for the input string *abb*, and for the same input string, token `T_A` followed by `T_C` would be incorrect).

Approach 1: We can take the specification in terms of regular expressions for each token, convert each regexp into an NFA using Thompson's construction, then take the union of these NFAs, then convert it into a DFA (making sure to remember the mapping between the DFA final states and the original NFA final states and the order of token definitions). Then we can use the DFA simulation algorithm to convert an input file into a sequence of tokens. In this approach, the specification of the lexical analyzer can directly use regular expressions as shown above.

Approach 2: It is generally much harder to write down a DFA for all the tokens simultaneously with a specified final state for each token. However, we can exploit the fact that it is easy to write a DFA for each token in isolation and use a hybrid NFA/DFA matching algorithm. Figure 1 contains the DFAs for the three regular expressions above. Once we have these DFAs we can combine them using ideas from the NFA simulation algorithm (see Algorithm 3.4 in the Dragon book). In fact, instead of using set of states, we can simplify NFA simulation in this case by simply keeping track of the length of a single pattern match for each DFA and select the next token in the sequence by picking the longest match that appears first in the token specification list. In this approach, the specification of the lexical analyzer can be defined as follows:

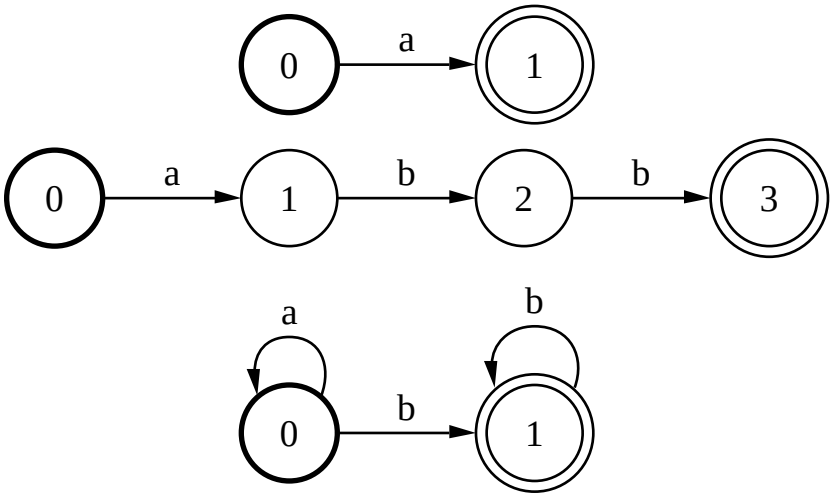

Figure 1: DFA equivalents for the regular expressions *a* (`dfa1`), *abb* (`dfa2`), and a^*b^+ (`dfa3`)

```
T_A dfa1.txt
T_B dfa2.txt
T_C dfa3.txt
```

where, `dfa1.txt` is the file containing the DFA for regexp *a*, `dfa2.txt` is the file with the DFA for regexp *abb* and `dfa3.txt` for a^*b^+ (see Figure 1 for a graphical depiction of what the files would contain).

There is no difference in marks between Approach 1 and 2. Approach 2 involves writing much less code but involves encoding all the patterns as DFAs and then writing them into files by hand. Approach 1, on the other hand, is more flexible, allowing the use of regular expressions to write patterns. And since it can determinize and minimize one single DFA for the entire specification, it can be substantially faster as well.

What is required: Choose one of the two approaches above and submit a program that implements the approach written in C++ (or in ANSI C) that can be run as follows:

```
lexan lex.txt lex-input.txt
```

The file `lex.txt` contains the specification for all the tokens for the lexical analyzer. Please indicate in your submission (in a README file) whether your implementation uses regular expressions in this file (approach 1), or DFA filenames (approach 2).

The file `lex-input.txt` contains the text that is passed through the lexical analyzer that uses the specification in `lex.txt` to convert this text into a stream of tokens.

Let us assume that `lex.txt` contains the specification shown in the example above. If the input text file contains *aaba*, the program should produce the following output token types and their values (lexemes):

```
T_C     aab
T_A     a
```

If the input text file contains *aabaaabbbbabba*, the program should produce the following output token types and their values (lexemes):

```
T_C     aab
T_C     aaabbbb
T_B     abb
T_A     a
```

If the input text file contains *aabaaabbbbsbba* which includes an illegal input character `s`, the program should produce the following output token types and their values (lexemes):

```
T_C     aab
T_C     aaabbbb
illegal token
```

You can add more elaborate error reporting if you wish.

Here is another example of a lexical specification file (using approach 1) for some keywords. This is closer to the actual patterns used in a working compiler:

```
KW_INT    int
KW_DOUBLE double
KW_STRING string
KW_VOID   void
KW_WHILE  while
```

With the above specification (and assuming appropriate whitespace handling), an input file containing *while int string void while double while* would produce the tokenization:

```
KW_WHILE        while
KW_INT          int
KW_STRING       string
KW_VOID         void
KW_WHILE        while
KW_DOUBLE       double
KW_WHILE        while
```

In order to make testing your code possible, provide a file called `compileit` or a `Makefile` that produces an executable called `lexan`. This program should be invoked and produce output exactly as in the example runs shown above.