Programming assignment
String Matching

The goal of this programming assignment is to create an application that finds a given string of characters (referred to as *pattern*, the number of symbols in the pattern is *patternsize*) in a text (number of symbols in the text is *textsize*). The brute force solution that simply goes through text and compares *patternsize* number of symbols to the corresponding symbols in the text has complexity *O(patternsize * textsize)* and implemented as a pair of nested loops (see code). There is a number of much more efficient algorithms – check *http://www-igm.univ-mlv.fr/~lecroq/string/*

We are going to implement "*Deterministic Finite Automaton algorithm"* (which is just a clever name for a *Finite State Machine*). This algorithm is comprised of 2 steps – first is the preprocessing step, which creates a FSM from the given pattern (text is not used during this step) and the second is the actual traversal of the text, which will have the complexity *O(textsize)* if implemented using jump table or *O(textsize * log(patternsize)*) if implemented using `std::map` (the extra *log(patternsize)* is exactly the complexity of finding a symbol in `std::map`). For this assignment you'll be using the second approach (see `state.h`).

State and transitions
State in this assignment is defined as the largest suffix of text traversed so far that matches (equals to) the prefix of the pattern (should be same size as suffix). For example if text traversed so far is "aaaaab", and pattern is "abac". then suffix of size 2 of the text "ab" is equal to a prefix of size 2 of the pattern "ab",

```
aaaaab
    abac
```

but the same is not true for the suffix of size 3:

```
aaaaab
   abac
```

Therefore state is "ab".
Another definition (less formal) of state is "how many characters in the text are matching pattern so far".

Reading the next character from the text will either take us to the state "aba", if the character is 'a', or back to empty state "" (no suffix of the text is equal to the prefix of pattern). Thus the next character defines a *transition* from one state to another. For the previous example we have the following transitions:

```
"ab"--a-->"aba"
"ab"--*-->""
```

first is the transition corresponding to character 'a', and second line uses a wildcard – all other characters except 'a'.

Preprocessing step
Suppose we a looking for a pattern "abac", then the skeleton of the FSM is made of 5=*patternsize*+1 states corresponding to 1`)"" 2)"a" 3)"ab" 4)"aba" 5)"abac"` and 4 transitions

```
""---a-->"a"
"a"---b-->"ab"
"ab"---a-->"aba"
```

`"aba"---c-->"abac"`
that is, if currently in state `"a"` and the next character in text is 'b', transition to state `"ab"`, etc. To complete the FSM, we have to specify transition for all remaining characters. Most of those transitions will be to `""`. But there are some remaining transitions which do not end up in the `""` state. Consider state "aba" and the next character 'b'. Then the resulting text is
ab**ab**
and suffix of size 2 matches the pattern, so the corresponding transition is
`"aba"---b-->"ab"`.
Terminology: state `""` is also called *initial* state.

Matching step
This is the "punchline" of the algorithm – here is pseudo-code

```
state=""
while read next character from text (ch) {
  state=transition(state,ch)
}
```

the complexity of the algorithm is

textsize * complexity(`state=transition(state,ch)`)

If all transitions are written in an array which is indexed by the alphabet, for example for state `"aba"`, the array will be

```
array: "a" "ab"  ""   ""   ""   ""  ....
index: a=0  b=1  c=2  d=3  e=4  f=5 ....
```

then

complexity(state=transition(state,ch))=O(1)

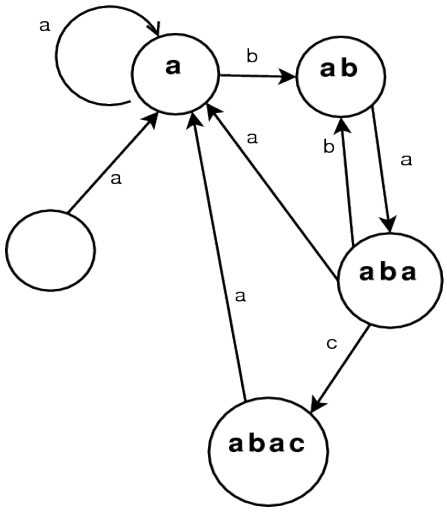
constant time. Notice that "jump table" solution requires the knowledge of the alphabet (all possilble characters) which may be unavailable.

If transitions are stored in a map (which is sometimes called *associative array*), then the complexty is

log(size of map)

In my state implementation map only keeps track of the non-trivial (not ending up in the initial state) transitions, therefore the size of the map is at most the size of pattern (patternsize).

Here is a complete FSM for pattern "abac".



Note: to implement jump table using arrays State and FSMFind have to take an extra argument – alphabet (see comment above) which will make client's code more difficult to read. But it's possible to remove log(patternsize) from complexity and NOT have alphabet by using hashes. NOT REQUIRED.

Design considerations

- To allow any alphabets in our string matching, State and FSMFind are implemented as template classes. The only template parameter is Symbol, which is just "char" for all tests, except test10.
- Notice that FSMFind may find several occurences of the pattern, to return all of them (or rather their positions) to the client, FSMFind may
  1) create an array of positions and return it in the end of processing. Cumbersome and client has to wait till the whole text is processed.
  2) FSMFind may call user code each time it finds the pattern.

The latter approach is more flexible, and may be easily implemented with callback functions. To make callback function a little bit more C++-style (safe and flexible), we define AbstractCallback class and allow `AbstractCallback*` pointers to be registered with FSMFind (method Register). There may be several callbacks for each event (events are OnDone and OnFind), so FSMFind should have an `std::vector<AbstractCallback*>` for each of the 2 events.

What to submit

```
state.h, state.cpp,
fsmfind.h, fsmfind.cpp
concrete-callback.h, concrete-callback.cpp
```