Class 09: From Specification to Optimal DFA (1)

Held: Wednesday, 14 September 2011

Summary: Today we start to consider how to move from the readable but declarative regular expression notation to the executable but obtuse finite automaton notation.

Related Pages:
- EBoard.
- Reading: Aho et al., 3.7.
- Due: Tokenizing Pascal.

Notes:
- Don’t forget that assignment 2 is due tonight.

Overview:
- NFAs.
- Where we’re going: From regular expression to code.
- From regular expressions to NFAs.
- Examples.

Nondeterministic Finite Automata
- It turns out that it is interesting and useful to consider a variant of finite automata in which
  - some transitions are “free”, in that they do not require any symbols in order to make the transition. Typically such edges are labeled with epsilon.
  - there may be multiple edges with the same label exiting the same node
- Such finite automata are often called nondeterministic finite automata or NFAs.
- When is a string in the language given by such an automaton? When there is some path through the automaton that uses the appropriate symbols and free transitions. [We’d like the computer to guess the correct path.]
- Do we gain any power from this nondeterminism? Surprisingly, no. As we’ll see over the next few classes class, nondeterministic finite automata can easily be converted to deterministic finite automata.

Limitations of Regular Expressions, DFAs, and the Ilk
- Covered in previous class, but really belongs here.
- Many useful languages cannot be represented by regular expressions.
- For example, “strings of a’s and b’s with equal numbers of a’s and b’s”
Here’s an even simpler one: $a^k b^k$

How would you prove that you can’t represent that last one? (No, I won’t put it here, since many of you read along.)

- **Hint:** Think about the F in DFA.

### From Regular Expression to Code

- Regular expression to NFA
- NFA to DFA
- DFA to optimal DFA
- Optimal DFA to code.

### From Regular Expression to NFA

- Our regular expression to NFA converter builds a finite automaton with one final state (and one start state). It uses the recursive definitions of regular expressions to build a finite automaton for a compound expression from the finite automata for the base expressions.
- Each created NFA has a unique start state and a unique final state.
- The NFA for the empty string has two states with an epsilon transition between the states. The first state is the start state. The second state is the final state.

```
epsilon
-> o -----------> 0
```

- The NFA for a single symbol, $s$, is similar, except that the transition is labeled with the symbol.

```
s
-> o -----------> 0
```

- The NFA for alternation, $(R_0|R_1)$, has a new start state with epsilon edges to the start states of the NFA for the subexpressions and a new final state with epsilon edges from the final states of the NFA for the subexpressions.

```
  --> o-NFA0-O -->
  /                \epsilon
/epsilon          \
-> o                --> 0
\epsilon          /
  \                /epsilon
--> o-NFA1-O -->
```

- The finite automaton for concatenation, $R_0 R_1$, uses the start state of the NFA for the first expression and the final state of the NFA for the second expression. There is an epsilon transition from the final state of the first NFA to the start state of the second NFA.

```
epsilon
-> o-NFA0-O---------->o-NFA1-O
```

- Alternatively, we can merge the final state of the first NFA with the start state of the second
NFA.

- The NFA for repetition, $R^*$, has a new start state and a new final state. There are also epsilon transitions
  - from the start state to the final state,
  - from the start state to the start state of the sub-NFA,
  - from the final state of the sub-NFA to the final state, and
  - from the final state of the sub-NFA to the start state of the sub-NFA.

- If it seems to you that some of these extra states and epsilon transitions are unnecessary, you’re probably right. However, you can’t get rid of all of them.
  - For example, consider what happens if you don’t create any extra states for concatenation or repetition.

Some Examples

- Let’s try something simple, like strings of a’s and b’s that start with an a and end with a b.
  
  $a(a|b)^*b$

- Let’s also try something ugly, like integer constants in C. We’ll take a few shortcuts, particularly for sets.

  $((([1-9][0-9]*) | (0[1-9]*) | (0x[1-9A-Fa-f]+)) [sSuU]?[lL]?)$