Class 10: From Specification to Optimal DFA (2)

Held: Friday, 16 September 2011

Summary: Today we continue our consideration of how to move from the readable but declarative regular expression notation to the executable but sometimes obtuse finite automaton notation.

Related Pages:
- EBoard.

Notes:
- No new assignment right now. Just keep reading in the book.
- EC for town hall meeting next week. (You should go anyway - What Kington has to say about the budget it interesting and important.)
- EC for next week’s CS extra on Grad School in CS.

Overview:
- Examples.
- From NFA to DFA.
- From DFA to optimal DFA.
- Lexical analysis using finite automata.
- Limitations of regular expressions and finite automata.

Examples
- We’ll do some examples of converting regular expressions to NFAs.

From NFA to DFA
- How do we turn our lovely NFAs into a deterministic computing machine?
- Basically, we build a DFA that simultaneously follows all the paths that we can go through in the NFA.
  - Each state in the DFA is a set of states in the corresponding NFA.
- The algorithm is again fairly simple
- Terminology:
  - qx is a state in the NFA and q0 is its start state
  - QX is a state in the DFA and Q0 is its start state
  - The epsilon-closure of a DFA state consists of all the states in the NFA that we can get to with epsilon moves from the states in the DFA state.
Q0 = { q0 }
// but there are some states we can reach from q0 at no cost
Q0 = epsilon-closure(Q0)
while there are states in the DFA we haven’t processed
    pick one such state, Qn
    for each symbol s
        let tmp be a new, empty, set
        for each q in Qn
            add delta(q, s) to tmp
        end for
        tmp = epsilon-closure(tmp)
        if tmp is not in the DFA then
            let Qi be a new state
            Qi = tmp
            add Qi to the DFA
        else
            let Qi be the state equivalent to tmp
        end if
        add an edge from Qn to Qi in the automaton
    end for
end while
for each Qi
    if there is a q in Qi that is a final state then
        Qi is a final state
    end if
end for

From DFA to Optimal DFA

- As you may be able to tell, the DFAs constructed by the NFA to DFA construction are is fairly large.
- Can we build an equivalent DFA that’s smaller? Yes!
- Strategy: Group states into sets of equivalent states.
- Surprisingly, it’s not enough to do this bottom-up (that is, if two states have the same edges, join them together).
  - Sometimes you need to join states in pairs (or more)
- Even more surprisingly, it’s better to do it top-down (assume that all states can be joined and refine that decision)
- Here’s some pseudocode

Assume all non-final states can be treated as the same state
Assume all final states can be treated as the same state
For each group of states treated as equivalent
as the same state
    For each symbol, s
        If there are two "equivalent" states q1, q2 such that edge(q1, s) and edge(q2, s) lead to non-equivalent states,
            split q1 and q2 into different equivalencies
            figure out where the other states in the original equivalency go
        End For // each symbol
    End for // each pair of states
From Token Definitions to DFA

- How do we convert a series of token definitions (using regular expressions) to a tokenizing DFA?
- We build an NFA for each of the token definitions.
- We augment the final states with token type and priority.
- We put 'em together in one big automaton by adding a new start state with epsilon transitions to the start states of all the other automata.
- We convert the result to a DFA.
- When we make the determination of a final state, we use the associated token type of the highest priority final state in the NFA.

Tokenizing with Finite Automata

- We’ve seen how to match with finite automata, but how do we tokenize?
- We only tokenize with deterministic finite automata.
- We begin by attaching a token (or, sometimes, a set of instructions) to each final state.
  - When we reach an appropriate final state (possibly not the first final state), we stop and return the corresponding token.
- How do we decide whether a final state is appropriate? We always keep track of the last final state we encountered, but peek ahead until we find another final state or hit a dead end.

```java
/**
 * Find the first token in the candidate string, starting
 * at a particular position.
 */
public token findToken(String candidate, starting_pos)
begin
    State current_state = q0;
    for i = starting_pos to the length of candidate
        current_state = edge(current_state,candidate.symbolAt(i))
        if current_state is a final state
            final_found = current_state
            final_pos = i
        end if
        if current_state is undefined then
            exit the for loop
        end if
    end for
    if (final_found is defined) then
        return the token given by final_found at position final_pos
    else
        no token can be found
    end if
end
```
Limitations of Regular Expressions, DFAs, and the Ilk

- 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.