1 How does generation work?

- There is a systematic way to map any regular expression to a lexical analyzer
- Three steps
  - regular expression ⇒nondeterministic finite state automaton
  - nondeterministic finite state automaton ⇒deterministic finite state automaton
  - deterministic finite state automaton ⇒scanner program

2 Finite State Automata

- Consist of a finite number of *states* and *transitions*
- Transitions are labelled with input characters.
- There is one *start state*.
- A subset of the states are called *final states*.
- A finite state automaton starts in the start state, and for each input symbol follows an edge labelled with that symbol.
- It *accepts* an input string iff it ends up in a final state.
- Examples: Blackboard.

3 (Non)Deterministic Finite State Automata

- In a *nondeterministic finite state automaton* (NFA), there can be more than one edge originating from the same node and labelled with the same label.
- Or there can be a special ϵ-edge, which can be followed without consuming any input symbols.
- By contrast, in a *deterministic finite state automaton* all edges leaving some node have pairwise disjoint label sets and there are no ϵ-labels.

4 From a Regular Expression to an NFA

5 From an NFA to a DFA

- Problem: Executing an NFA needs *backtracking*, which is inefficient.
- We would prefer a DFA.
- Idea: Do all possible choices in parallel.
- Construct a DFA, which has a state for each possible set of NFA states.
- A DFA state is final if the set of its NFA states contains a final state.
- Since the number of states of an NFA is finite (say N), the number of possible sets of states is also finite (bounded by 2^N).
- Often the number of reachable sets of states is much smaller.

6 Algorithm

- First step: For a set of states S, let closure(S) be the largest set of states, that is reachable from S using only ϵ-transitions.
- Algorithm to compute T = closure(S):

```
T = S;
do {
    T' = T;
    for each state s ∈ T {
        for each edge e from s to some s' {
            if (e is labelled with ϵ) {
                T = T ∪ s';
            }
        }
    }
} while (T != T')
```

- This is an example of a fixpoint algorithm.

7 Algorithm (2)

- Second step: For a set of states S and an input symbol c, let DFAedge(S,c) be the set of states that can be reached from S by following an edge labelled with c.
- Algorithm to compute T=DFAedge(S,c)

```
T = ∅;
for each state s ∈ S {
    for each edge e from s to some s' {
        if (e is labelled with c) {
            T = T ∪ closure({s'});
        }
    }
}
```

8 Simulating a DFA

- Using the machinery developed so far, we can already *simulate* a DFA, given an NFA.
- Let s be the start state. Then the simulation works as follows

```
d = closure({s});
while (ch != EOF) {
        d = DFAedge(d, ch);
        nextCh();
}
```

- Manipulating these sets at runtime is still very inefficient.

9 DFA Construction

- DFA-states are numbered from 0.
- 0 is the error state, corresponding to the empty set of NFA-states. The DFA goes into state 0, iff the NFA would have blocked because no edge matched the input symbol.
- `states` is an array which maps each DFA-state to the set of NFA states it represents. `trans` is a matrix of transitions from state numbers to state numbers.

10 DFA Construction (2)

- Algorithm:

```
states[0] = ∅; states[1] = closure({s});
j = 0; p = 2;    /* states[0..j–1] done, state[j..p–1] to do */
while (j < p) {
        for each input character c {
                d = DFAedge (states[j], c);
                if (d == states[i] for some i < p)
                        trans[j, c] = i;
                else {
                        states[p] = d;
                        trans[j, c] = p;
                        p = p + 1;
                }
        }
        j = j + 1;
}
```

11 Executing a DFA

- use trans

```
s = 1;
while (ch != EOF) {
        s = trans[s, ch];
        nextCh();
}
```

12 Executing a DFA

- generate switch

```
s = 1;
while (ch != EOF) {
    switch (s) {
    case 0: error(); break;
    case 1:
        switch (ch) {
        case 'a': s = 3; break;
        ...
        }
        break;
    ...
    }
    nextCh();
}
```

13 Summary: Lexical Analysis

- Lexical analysis turns input characters into tokens.
- Lexical syntax is described by regular expressions.
- We have learned two ways to construct a lexical analyzer from a grammar for lexical syntax.
- By hand, using a program scheme
- By machine, using JLex to construct of DFA.
- Scanner generator / hand-written scanner
  - Speed
  - Size
  - Flexibility
  - Maintenance
  - Ease of Coding