Language Processors (E2.15)

Lecture 5: Lexical Analysis

Objectives

- To introduce non-deterministic finite automata (NFA), and demonstrate how they can be converted to DFAs
- To introduce Thomson's algorithm for converting regular expressions to NFA
- To introduce the DFA state minimization algorithm

Introduction (1)

Review: Lexical analysis, regular expressions, finite state automata

- Lexical analysis is concerned with converting an input stream of characters into an output stream of *tokens.*
- *Tokens* are the basic constituent elements of the language – for example, the keywords "begin, for, if, then, else" for PASCAL.
- We already saw that for a regular expression defining the structure of the tokens of the language, there is a FSA that can be constructed to recognize whether a series of characters is a valid part of the language (a valid token)

--

- The lexical analyser is usually a routine in the compiler, that when it is called, reads the input stream and returns the next token that it finds.
- Tokens have associated attributes – we want them too!

Introduction (2)

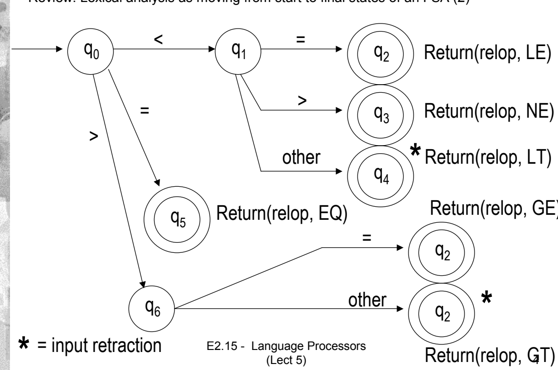
Review: Lexical analysis as moving from start to final states of an FSA (1)

- Lexical analysis can be viewed as the process of starting from the start state, reading the input one character at the time, and moving between states according to the input
- If we are in an accepting state when the input stream is terminated, then the input has been recognised as valid – appropriate messages and attributes are returned.
- Example, for the relational operator token (relop) in Pascal has the following regular expression:

relop → < | <= | = | <> | > | >=

Introduction (2)

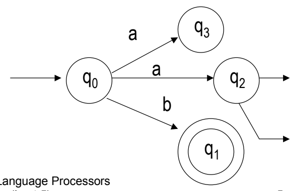
Review: Lexical analysis as moving from start to final states of an FSA (2)



NFA (1)
DFA and NFA (1)

- Deterministic FSA allow only one transition from one state to another for a given input
- Non-deterministic FSA allow more than one transition possibilities for a given input,
- for example from q_0, given *a* as input we can go either to q_2, or q_3

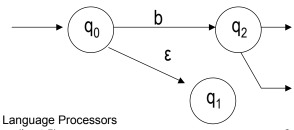
state	input	
	a	b
q_0	$\{q_2, q_3\}$	q_1
….		



NFA (1)
DFA and NFA (2)

- Another difference between DFA and NFA is that NFA allow *ε-transitions*
- An ε-transition is a transition between states without reading any input
- Indicates equivalent states

state	input	
	b	ε
q_0	$\{q_2\}$	$\{q_1\}$
….		

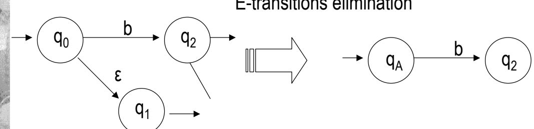


NFA (2)
Equivalency (1)

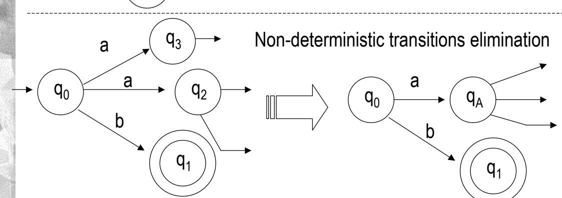
- NFA can be converted to a DFA (through a process known as *subset construction*)
- The main idea: merge sets of states of the NFA in single states in the DFA; adopt the viewpoint of the input characters:
  - Any two states in the NFA that are connected by an ε-transition are essentially the same, since we can move between them without consuming any characters => we can represent them using only one state in the DFA
  - For any symbol that can result to multiple transitions we can consider that we can have a single transition to a set of states (the union of all those states reachable by a transition on the current symbol). This set can be combined into a single DFA state.

NFA (2)
Equivalency (2) - examples

E-transitions elimination



Non-deterministic transitions elimination



NFA (3)

Subset construction algorithm (1)

- The subset construction algorithm requires the definition of two functions:
  - The ε-closure function takes as input a state and returns the set of states reachable from it based on one or more ε-transitions. This set will always include the state itself
  - The function move (State, Char) which takes a state and a character and returns the set of states reachable from it with one transition using this character.
- The algorithm then proceeds as follows:

NFA (3)

Subset construction algorithm (2)

1. The start state of the DFA is the ε-closure of the start state of the NFA
2. For the created state of (1) and for any created state in (2) do:

   For each possible input symbol:
   - Apply the move function to the created state with the input symbol
   - For the resulting set of states, apply the ε-closure function; this might or might not result in a new set of states
3. Continue (2) until no more new states are being created.
4. The final states of the DFA are the ones containing any of the final states of the NFA

[Potential exponential expansion in the state space]

NFA (4)

NFA – a useless concept?

- So, NFA can be reduced to DFA [which are easier to program]!
- Why NFA?!
  - Instrumental for the automatic construction of DFA from regular expressions
  - RE -> DFA tough; instead we proceed as

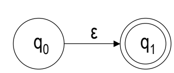
    RE -> NFA -> DFA
  - ε-transitions a key element!
- Time for Thompson's algorithm for converting a RE to an NFA:

Regular Expressions to NFA

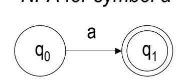
Thompson's algorithm (1)

- Key idea: construct an NFA for each symbol and for each operator (|, union, Kleene *); join the resulting NFA with ε-transitions (in precedence order)

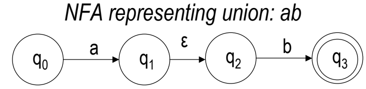
NFA representing the empty string



NFA for symbol a



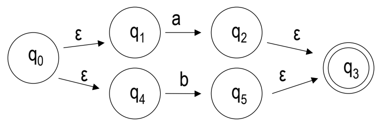
NFA representing union: ab



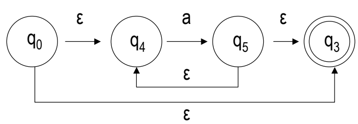
Regular Expressions to NFA

Thompson's algorithm (2)

NFA representing a | b



*NFA representing Kleene' star: a**

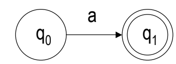


Regular Expressions to NFA

Thompson's algorithm (3) - Example

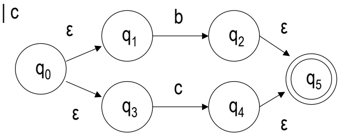
Converting regular expression a(b|c)* to a NFA

We start with each symbol, e.g. *a*:



(and similarly for b and c)

Then: b | c

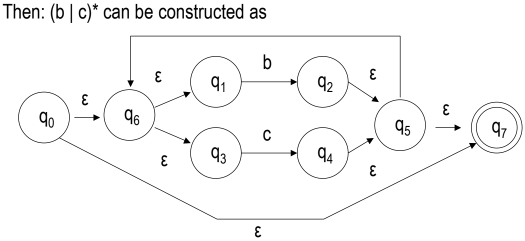


Regular Expressions to NFA

Thompson's algorithm (3) – Example (2)

…Converting regular expression a(b|c) to a NFA (cont.)*

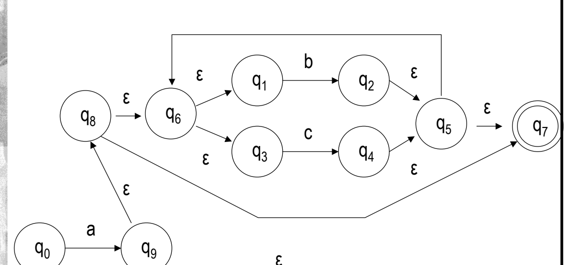
Then: (b | c)* can be constructed as



Regular Expressions to NFA

Thompson's algorithm (3) – Example (3)

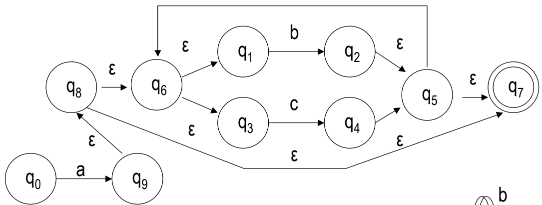
And finally: a(b|c)* can be constructed as



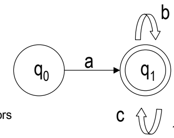
Regular Expressions to NFA

Thompson's algorithm (3) – Example (4)

?! But can't we do much better by having instead of:



the equivalent:



Automatic construction (1)

Overview of the process

- Yes, but the algorithmic nature of Thompson's construction means that NFA can be created automatically from RE
- Subsequently they can be converted to DFA (the subset construction algorithm), for which its easier to automatically generate code
- There are algorithms for minimizing DFA (ensuring that the constructed DFA has the smallest number states possible)

Automatic construction: process overview

Automatic construction (2)

DFA minimization

- Start by assuming that all the states in the DFA are equivalent
- Work through the states, putting different states in separate sets
- Two states are considered different if:
  - One is a final state and the other one isn't
  - The transition function maps them to different states, based on the same input character.

1. The minimization algorithm starts by initially creating two sets of states, final and non-final.
2. For each state-set created from (1), it examines the transitions for each state and for each input symbol. It the transition is to a different state for any two states, then they are put into different state-sets.
3. Repeat until no new state sets are being created by 2.

Automatic construction (3)

Code generation for the DFA

- Once the DFA has been constructed, code to simulate it can be generated.
- The state transition table defines function move (s,ch) which returns the next state from s given ch as input character. Then:

```
S:=S0
c:=nextchar();
while c <> eof do
    s:= move (s, c)
    c:= nextchar();
end;
if s is in FinalStates then
    return "yes";
else return "no";
```

Summary

- NFA provide a useful intermediary point between RE and DFA.
- Lexical analysers can be generated by hand, or automatically by converting regular expressions to NFA (Thompson’s algorithm), then the NFA to a DFA (subset construction algorithm), minimizing the DFA states, and converting into code.

Next lecture:

Lexical analysis using LEX

Recommended Reading

- Chapter 3 of Aho et al
- Section 2.1 of Grune et al.