Compiler Construction

Lexical Analysis

Department of Computer Science

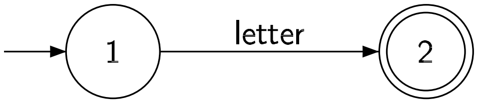
King Saud University

Finite Automata (1)

- Used to recognize the tokens specified by a regular expression
- Can be converted to an algorithm for matching input strings
- A Finite Automaton (FA) consists of:
  - A finite set of states
  - A set of transitions (or moves) between states
    - The transitions are labeled by characters form the alphabet
  - A special start state
  - A set of final or accepting states

Finite Automata (2)

- A finite automaton for ***letter(letter|digit)**** is shown below



- We may label a transition with more than one character for convenience
- We start at the start state
- We make a transition if next input character matches label on transition
- If no move is possible, we stop
- If we end in an accepting state then
  - input sequence of characters is valid
- Otherwise, we do not have a valid sequence

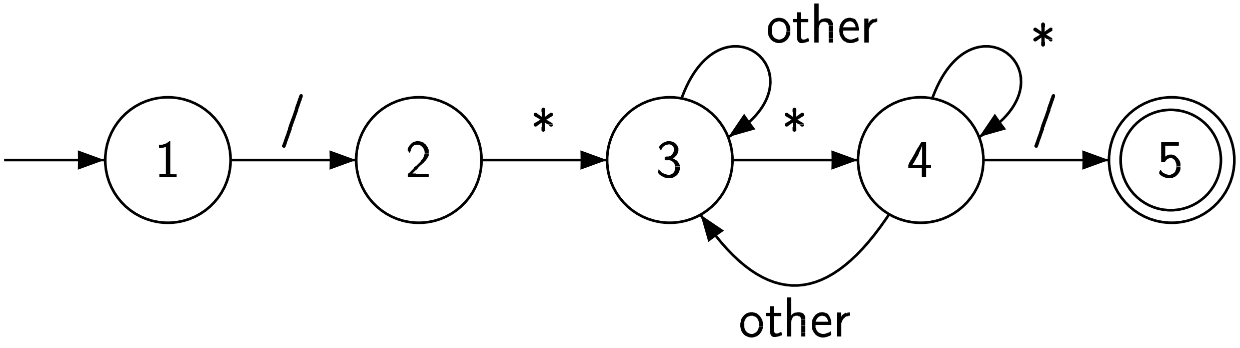
Deterministic Finite Automata

- Has a unique transition for every state and input character
- Can be represented by a transition table T
  - Table T is indexed by state s and input character c
  - $T[s][c]$ is the next state to visit from state s if the input character is c
  - T can also be described as a transition function
  - $T : S \times \Sigma \longrightarrow S$ maps the pair (s, c) to $next\_s$

Deterministic Finite Automata

- DFA and transition table for a C comment are show below
  - Blank entries in the table represent an error state
  - A full transition table will contain one column for each character (may waste space)
  - Characters are combined into character classes when treated identically in a DFA

State	/	*	*other*
1	2		
2		3	
3	3	4	3
4	5	4	3
5			

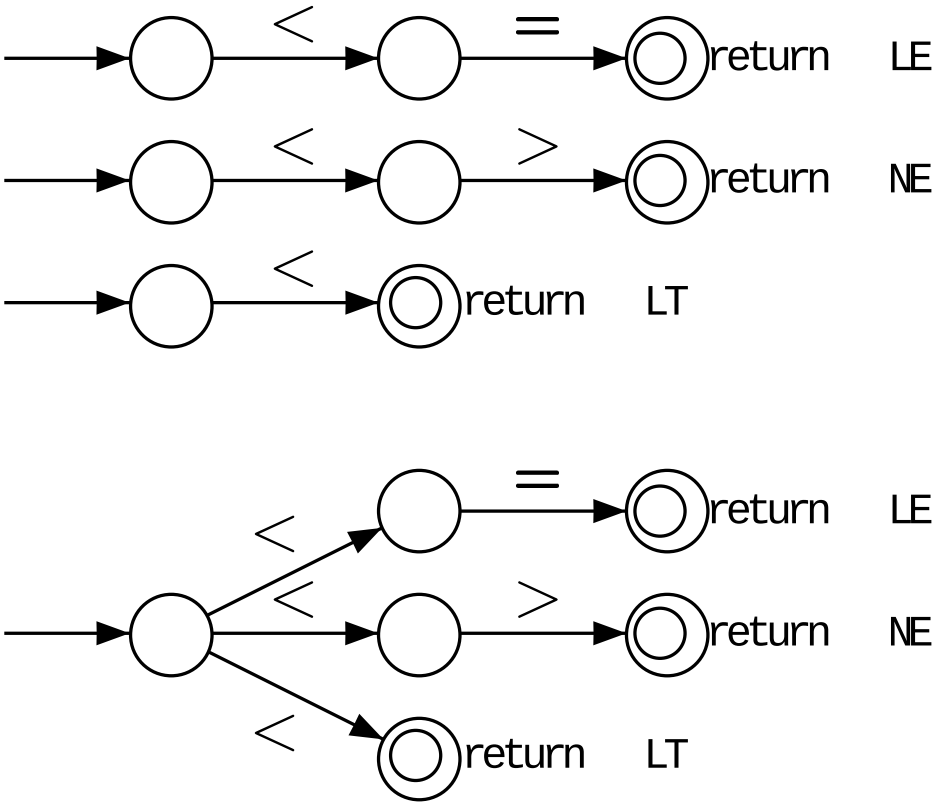


Combining DFAs

- In a programming language there are many tokens
- Each token is recognized by its own DFA
- We need to combine DFAs together into one large DFA
    - Unite the starting states of various DFAs into one starting state
    - Simple if each token begins with a different character
    - Becomes more complex if some tokens have a common prefix

Combinig DFAs (2)

- Consider the DFAs for <, <=, and <>
  - They share a common prefix <
  - They are combined into one DFA as shown below

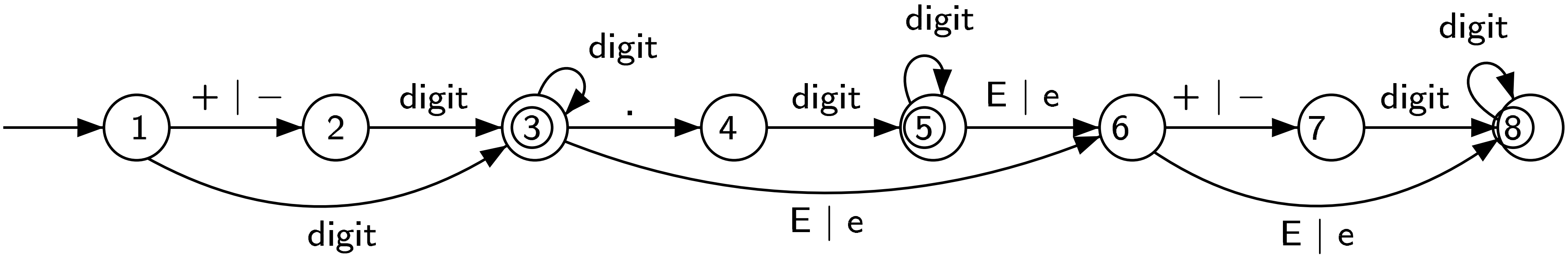


Algorithmic Aspects of a DFA

- A DFA diagram is just an outline of a scanning algorithm
- A DFA does NOT describe every aspect of the algorithm
- What happens when making a transition? A typical action is to
  - Save the character read in a string buffer belonging to a single token
  - The string value is the lexeme of the token
- What happens when we reach an accepting state?
  - If no further transition is possible, we return the token recognized
  - If further transitions are possible, we continue to match the longest string

Algorithmic Aspects of a DFA (2)

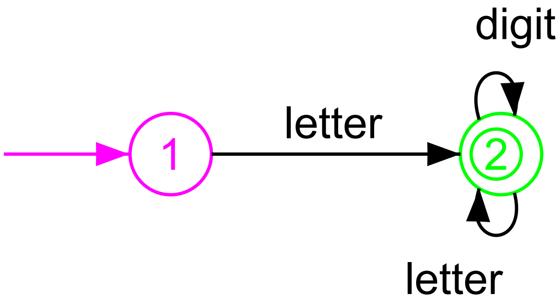
- What happens when no transition exist from an non-accepting state?
  - We can backtrack to the last accepting state, if we visited one
    - The extra characters read, called lookahead characters, are returned back to input
  - We can return an error token if no accepting state is visited



Converting a DFA into an Algorithm

- We can convert a DFA into an algorithm by:
  - Using a variable, `state` , to maintain the current state
  - Writing transitions as case statements inside a loop
  - The first case statement tests the current state
  - The nested case statements tests the input character `ch`
  - The `unput(ch)` statement returns `ch` back to input

Converting a DFA into an Algorithm (2)



```
state := 1;
input(ch);
while not eof do
  case state of
    1:case ch of
        letter:state := 2; input(ch);
        else exit while ;
      end case ;
    2:case ch of
        letter,digit:input(ch);
        else unput(ch); exit while ;
      end case ;
  end case ;
end while ;
if state=2 then return id; else error;
end if
```

Converting a DFA into an Algorithm (3)

```
state := 1;
input(ch);
while not eof do
  case state of
    1:case ch of
        '/':state := 2; input(ch);
        else exit while;
      end case;
    2:case ch of
        '*':state:=3; input(ch);
        else exit while;
      end case;
    3:case ch of
        '*':state:=4; input(ch);
        else state:=3; input(ch);
      end case;
    4:case ch of
        '*':state:=4; input(ch);
        '/':state :=5; exit while;
        else state:=3; input(ch);
      end case;
  end case;
end while;
if state=2 then return id; else error;
end if
```

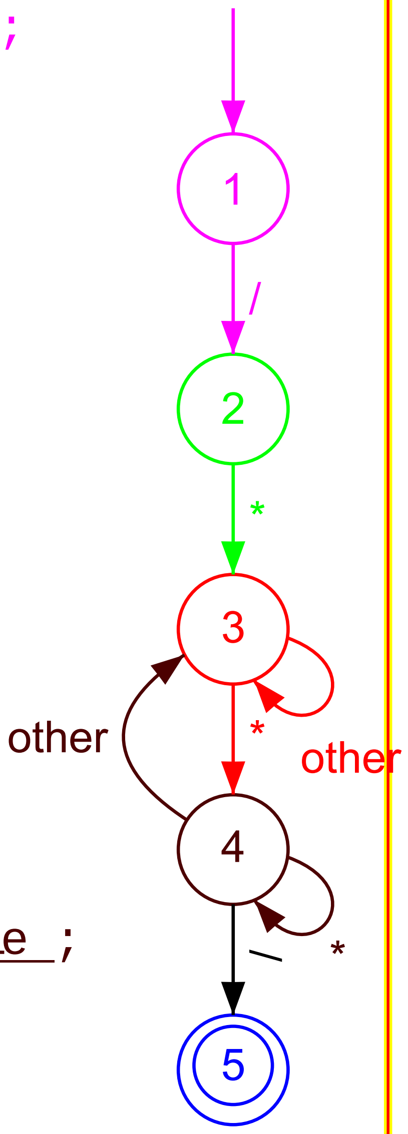


Table-Driven Generic Algorithm for a DFA (1)

- A DFA can be implemented as a generic algorithm
  - Driven by a transition table
- Suitable for scanner generators such as Lex
- Advantages of a generic algorithm:
  - Size of code is reduced
  - Same code works with different DFAs
  - Transition table is only modified
  - Code is easier to change and maintain
- Disadvantages:
  - Transition table can be very large
  - Much of the table space is unused
  - Table compression is required

Table-Driven Generic Algorithm for a DFA (2)

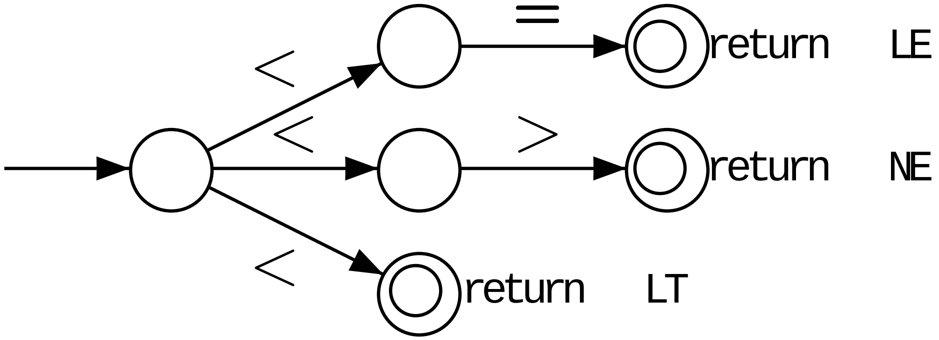
```
state := 1;
input(ch);
while not eof
    next_state := T[state][ch];
    if next_state = undefined then
        exit while;
    end if;
    state := next_state;
    input(ch);
end while;
if final(state) then
    unput(ch);      - - extra char
    return token;
else if previous final state
    backtrack to previous final state
    return token;
else
    error;
end if;
```

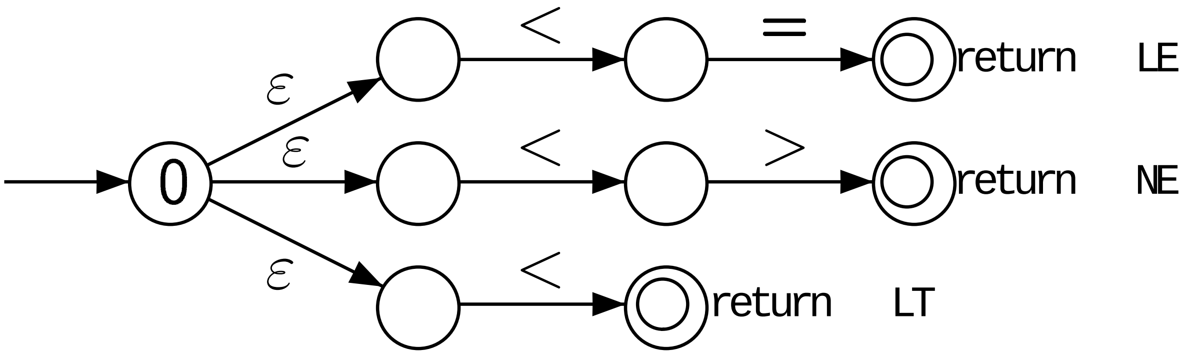
Nondeterministic Finite Automata (NFA) (1)

- An NFA is similar to a DFA except that:
  - Multiple transitions labeled by same character from same state are allowed
  - ε-transitions are allowed
- ε-transitions are spontaneous. They occur without consuming any character
- An NFA can be converted to an algorithm, except that:
  - There can be many transitions that must be tried to match an input sequence of chars
  - Transitions that have not been tried must be stored to backtrack to them on failure
  - Resulting algorithm of NFA is slower than the one that corresponds to a DFA

Nondeterministic Finite Automata (NFA) (1)

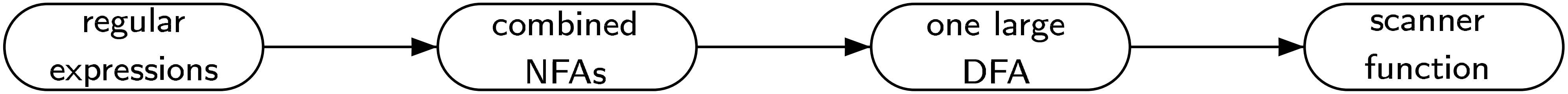
- DFAs with common prefixes can be combined into one large NFA by:
  - uniting their starting states,
  - or introducing a new start state and ε-transitions





From Regular Expressions to Scanner Function

- It is possible to transform regular expressions into a function
- First, regular expressions are transformed into NFAs
- Second, combined NFAs are converted into one large DFA
- Third, the DFA is converted into a scanner function



regular expressions → combined NFAs → one large DFA → scanner function

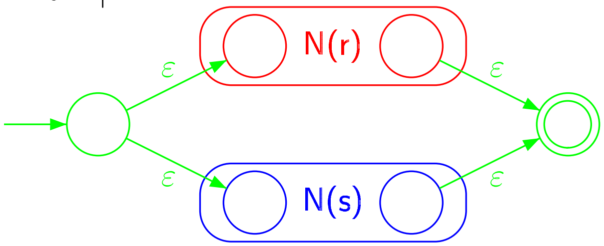
- The Thompson's construction transforms regular expressions into NFA
- The Subset construction is used to transform an NFA into a DFA

From a Regular Expression to an NFA

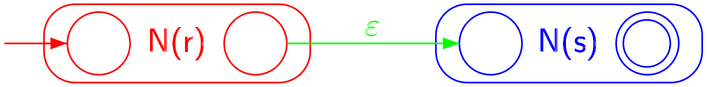
- Regular expressions are built out of:
  - Basic regular expressions **a** (where $a \in \Sigma$) and ε
  - Basic operations: concatenation rs, alternation $r|s$, and Kleene closure $r*$
- Regular expression for **a** and ε

- Thompson's construction of rs, $r|s$, and $r*$

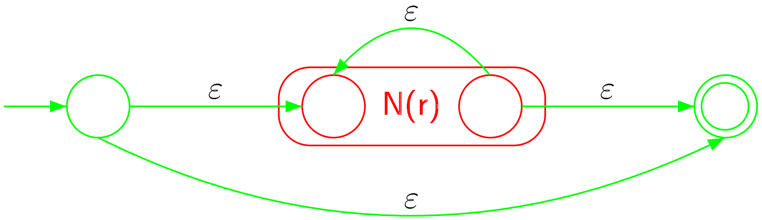
NFA for $r|s$



NFA for rs



NFA for $r*$



From an NFA to a DFA ■ Subset Construction (1)

- For any NFA N, we can construct a DFA M equivalent to it
  - Each state of M corresponds to a subset of the states of N
  - M will be in state $\{s_1, s_2, s_3\}$ after reading an input string iff N can be in s_1, s_2, or s_3
  - The initial state of M is the subset of all states that N could be in initially
    - This is the set of states reachable from the initial state of N following only ε-transitions

From an NFA to a DFA ■ Subset Construction (2)

- The set of states reachable following only ε-transitions is called the ε-closure
  - ε-closure(state s) = $\{s\} \cup$ {all states reachable from s following only ε-transitions}
  - Start state of M = ε-closure(start state of N)
- Once the start state of M is computed, we determine the successor states
  - Take any state S of M, S corresponds to a subset of states of N. $S = \{s_1, s_2, ...\}$
  - To compute S-successor under character c, we find the successors of $\{s_1, s_2, ...\}$ under c
  - The successors of $\{s_1, s_2, ...\}$ under c will be a new set of states $\{t1, t2, ...\}$
  - We compute $T = \varepsilon$-closure($\{t1, t2, ...\}$) ; ε-closure(set of states T)=$\cup_{t \in T}\varepsilon$-closure($t$)
  - T is included in M and a transition from S to T is labeled with c
- We continue adding states and transitions to M until all possible successors are added
- The process of adding new states to M must eventually terminate. Why?

Minimizing the Number of States in a DFA

- The DFA obtained by the subset construction algorithm can be minimized
- State s can be distinguished from state t in a DFA when for some string w:
  - Starting at state s and reading string w, we end up in an accepting state
  - Starting at state t and reading string w, we end up in a non-accepting state
- An algorithm that produces a minimum-state DFA is given in the next slide.

Minimizing the Number of States in a DFA: Algorithm

1)→ Construct an initial partition Π of the DFA set of states, S, with 2 groups:

- The set of final states F
- The set of non-final states $S \quad F$

2)→ For each group G of Π:

- Partition G into subgroups such that 2 states s and t of G are in the same subgroup iff:
  - $\forall a \in \Sigma$, states s and t have transitions on a to states in the same subgroup of Π
- Call the new partition Π_{new}. At worse, each state will be in a subgroup by itself

3)→ If $\Pi_{new} \neq \Pi$ then go back to step 2 with $\Pi := \Pi_{new}$; otherwise, proceed at step 4

4)→ Each group in the final Π becomes a state in the minimized DFA

- The states of a group G of Π cannot be distinguished and are merged into one state
- A transition from group G_1 to G_2 is marked with input symbol a when:
  - All states of G_1 make transition to states in G_2 on input symbol a