

# CS412/413

Introduction to Compilers  
Radu Rugina

Lecture 3: Finite Automata  
30 Jan 04

## Outline

- Regexp review
- DFAs, NFAs
- DFA simulation
- RE-NFA conversion
- NFA-DFA conversion

## Concepts

- **Tokens** = strings of characters representing the lexical units of the programs, such as identifiers, numbers, keywords, operators
  - May represent a unique character string (keywords, operators)
  - May represent multiple strings (identifiers, numbers)
- **Regular expressions** = concise description of tokens
  - A regular expressions describes a set of strings
- **Language** denoted by a regular expression = the set of strings that it represents
  - $L(R)$  is the language denoted by regular expression  $R$

## Regular Expressions

- If  $R$  and  $S$  are regular expressions, so are:

$\epsilon$       empty string  
 $a$       for any character  $a$   
 $RS$      (concatenation: "R followed by S")  
 $R|S$     (alternation: "R or S")  
 $R^*$     (Kleene star: "zero or more R's")

## Regular Expression Extensions

- If  $R$  is a regular expressions, so are:

$R?$       =  $\epsilon | R$  (zero or one  $R$ )  
 $R^+$      =  $RR^*$  (one or more  $R$ 's)  
 $(R)$      =  $R$  (no effect: grouping)  
 $[abc]$    =  $a|b|c$  (any of the listed)  
 $[a-e]$    =  $a|b|...|e$  (character ranges)  
 $[^ab]$    =  $c|d|...$   
(anything but the listed chars)

## Automatic Lexer Generators

- **Input to lexer generator: token spec**
  - list of regular expressions in priority order
  - associated **action** for each RE (generates appropriate kind of token, other bookkeeping)
- **Output: lexer program**
  - program that reads an input stream and breaks it up into tokens according to the REs. (Or reports lexical error -- "Unexpected character")

## Example: JLex

```

%%
digits = 0|[1-9][0-9]*
letter = [A-Za-z]
identifier = {letter}{letter|}[0-9_]*
whitespace = [ \t\n\r]+
%%
{whitespace} { /* discard */ }
{digits} { return new Token(INT, Integer.parseInt(yytext())); }
"if" { return new Token(IF, yytext()); }
"while" { return new Token(WHILE, yytext()); }
...
{identifier} { return new Token(ID, yytext()); }

```

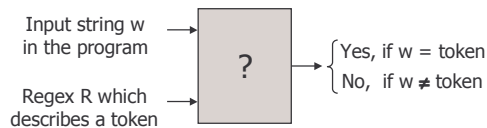
CS 412/413 Spring 2004

Introduction to Compilers

7

## How To Use Regular Expressions

- We need a mechanism to determine if an input string  $w$  belongs to the language denoted by a regular expression  $R$



- Such a mechanism is called an **acceptor**

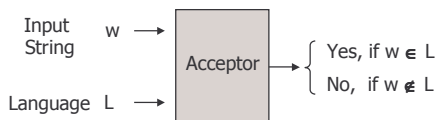
CS 412/413 Spring 2004

Introduction to Compilers

8

## Acceptors

- Accepter = determines if an input string belongs to a language  $L$



- Finite Automata** = acceptor for languages described by regular expressions

CS 412/413 Spring 2004

Introduction to Compilers

9

## Finite Automata

- Informally, finite automata consist of:
  - A finite set of **states**
  - Transitions** between states
  - An **initial state** (start state)
  - A set of **final states** (accepting state)
- Two kinds of finite automata:
  - Deterministic finite automata (DFA)**: the transition from each state is uniquely determined by the current input character
  - Non-deterministic finite automata (NFA)**: there may be multiple possible choices or some transitions do not depend on the input character

CS 412/413 Spring 2004

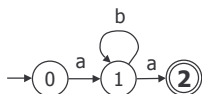
Introduction to Compilers

10

## DFA Example

- Finite automaton that accepts the strings in the language denoted by the regular expression  $ab^*a$

– A graph



– A transition table

	a	b
0	1	Error
1	2	1
2	Error	Error

CS 412/413 Spring 2004

Introduction to Compilers

11

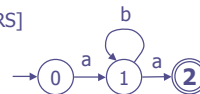
## Simulating the DFA

- Determine if the DFA accepts an input string

```

trans_table[NSTATES][NCHARS]
accept_states[NSTATES]
state = INITIAL

```



```

while (state != ERROR) {
    c = input.read();
    if (c == EOF) break;
    state = trans_table[state][c];
}
return accept_states[state];

```

CS 412/413 Spring 2004

Introduction to Compilers

12

## RE $\rightarrow$ Finite automaton?

- Can we build a finite automaton for every regular expression?
- Strategy: build the finite automaton inductively, based on the definition of regular expressions



CS 412/413 Spring 2004

Introduction to Compilers

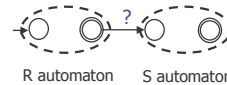
13

## RE $\rightarrow$ Finite automaton?

- Alternation  $R \mid S$



- Concatenation:  $R S$



CS 412/413 Spring 2004

Introduction to Compilers

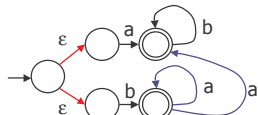
14

## NFA Definition

- A non-deterministic finite automaton (NFA) is an automaton where the state transitions are such that:
  - There may be  $\epsilon$ -transitions (transitions which do not consume input characters)
  - There may be multiple transitions from the same state on the same input character

Example:

regexp?



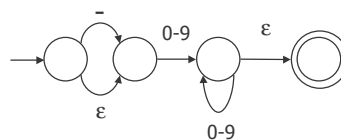
CS 412/413 Spring 2004

Introduction to Compilers

15

## RE $\Rightarrow$ NFA intuition

$-?[0-9]^+$



CS 412/413 Spring 2004

Introduction to Compilers

16

## NFA construction (Thompson)

- NFA only needs one stop state (why?)
- Canonical NFA:



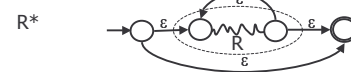
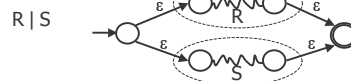
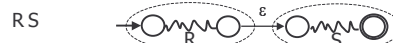
- Use this canonical form to inductively construct NFAs for regular expressions

CS 412/413 Spring 2004

Introduction to Compilers

17

## Inductive NFA Construction



CS 412/413 Spring 2004

Introduction to Compilers

18

## DFA vs NFA

- **DFA:** action of automaton on each input symbol is fully determined
  - obvious table-driven implementation
- **NFA:**
  - automaton may have choice on each step
  - automaton accepts a string if there is any way to make choices to arrive at accepting state / every path from start state to an accept state is a string accepted by automaton
  - not obvious how to implement!

CS 412/413 Spring 2004

Introduction to Compilers

19

## Simulating an NFA

- **Problem:** how to execute NFA?
  - “strings accepted are those for which there is some corresponding path from start state to an accept state”
- **Conclusion:** search all paths in graph consistent with the string
- **Idea:** search paths in parallel
  - Keep track of subset of NFA states that search could be in after seeing string prefix
  - “Multiple fingers” pointing to graph

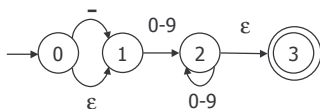
CS 412/413 Spring 2004

Introduction to Compilers

20

## Example

- Input string: -23
- NFA states:
  - {0,1}
  - {1}
  - {2,3}
  - {2,3}



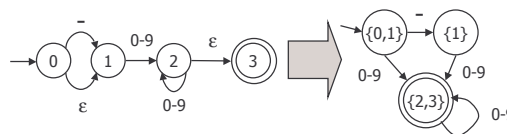
CS 412/413 Spring 2004

Introduction to Compilers

21

## NFA-DFA conversion

- Can convert NFA directly to DFA by same approach
- Create one DFA for each distinct subset of NFA states that could arise
- States: {0,1}, {1}, {2,3}



CS 412/413 Spring 2004

Introduction to Compilers

22

## Algorithm

- For a set  $S$  of states in the NFA, compute  $\epsilon$ -closure( $S$ ) = set of states reachable from states in  $S$  by  $\epsilon$ -transitions
  - $T = S$
  - Repeat  $T = T \cup \{s' \mid (s, s') \text{ is } \epsilon\text{-transition}\}$
  - Until  $T$  remains unchanged
  - $\epsilon$ -closure( $S$ ) =  $T$
- For a set  $S$  of states in the NFA, compute  $\text{DFAedge}(S, c)$  = the set of states reachable from states in  $S$  by transitions on character  $c$  and  $\epsilon$ -transitions
  - $\text{DFAedge}(S, c) = \epsilon$ -closure( $\{s \mid s' \in S, (s', s) \text{ is } c\text{-transition}\}$ )

CS 412/413 Spring 2004

Introduction to Compilers

23

## Algorithm

```

DFA-initial-state =  $\epsilon$ -closure(NFA-initial-state)
Worklist = { DFA-initial-state }

While ( Worklist not empty )
  Pick state  $S$  from Worklist
  For each character  $c$ 
     $S' = \text{DFAedge}(S, c)$ 
    if ( $S'$  not in DFA states)
      Add  $S'$  to DFA states and worklist
      Add an edge ( $S, S'$ ) labeled  $c$  in DFA

For each DFA-state  $S$ 
  If  $S$  contains an NFA-final state
    Mark  $S$  as DFA-final-state
    
```

CS 412/413 Spring 2004

Introduction to Compilers

24

## Putting the Pieces Together

