CS412/413

Introduction to Compilers
Radu Rugina

Lecture 3: Finite Automata
27 Jan 06

Last Lecture

- Tokens = strings of characters representing the lexical units in the program
  - E.g., identifiers, numbers, keywords, operators
- Regular expressions = concise description of tokens
- Language described by a regular expression
  - L(R) = the language of expression R

Regular Expressions

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

ε	empty string
a	character **a**
RS	concatenation
$R \mid S$	alternation
$R*$	Kleene star

Automatic Lexer Generators

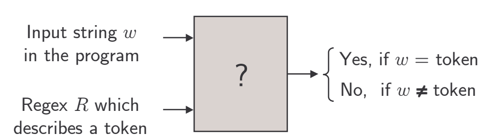
- Input to lexer generator: token spec
  - list of regular expressions in priority order
  - associated action for each RE (generates appropriate token object, other bookkeeping)
- Output: lexer program
  - program that reads an input stream and breaks it up into tokens according to the REs. (Or reports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

Example: JFlex

```
package FrontEnd;
import Error.LexicalError;
%%
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.valueOf(yytext()); }
"if"          { return new Token(IF, null); }
"while"       { return new Token(WHILE, null); }
{identifier}  { return new Token(ID, yytext()); }
.             { throw new LexicalError("illegal character"); }
```


How To Use Regular Expressions

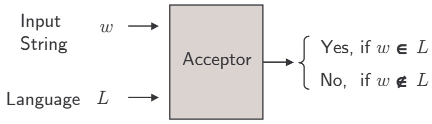
- We need a mechanism to determine if an input string w belongs to the language denoted by a regular expression R



- Such a mechanism is called an acceptor

Acceptors

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



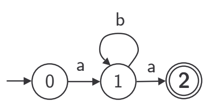
- Finite Automata = acceptor for languages described by regular expressions

Finite Automata

- Informally, a finite automaton 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

DFA Example

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



  - A transition table

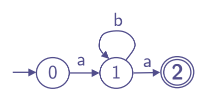
	a	b
0	1	Error
1	2	1
2	Error	Error

Simulating the DFA

- Determine if the DFA accepts an input string

```
table[NSTATES][NCHARS];
final[NSTATES];
state = INITIAL;

while (state != Error && !input.eof()) {
  c = input.read();
  state = table[state][c];
}
return (state != Error) && final[state];
```

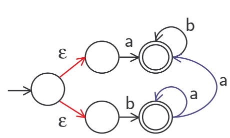


NFA Definition

- A non-deterministic finite automaton (NFA) is an automaton that can have:
  - ε-transitions (do not consume input characters)
  - multiple transitions from the same state on the same input character

Example:

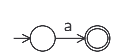
regexp?



Thompson: RE → NFA

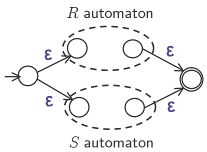
- Thompson's construction: build a finite automaton from a regular expression
  - Strategy: build the NFA inductively
- Empty string ε :

- Single character a:

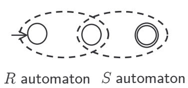


Thompson's Construction

- Alternation $R \mid S$

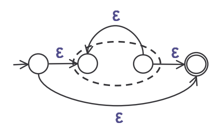


- Concatenation: $R\ S$



Thompson's Construction

- Kleene star $R*$



DFA versus NFA

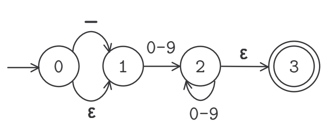
- DFA: automaton action is fully determined at each step
  - table-driven implementation
- NFA:
  - automaton might have choice at each step
  - Input string is accepted if one of the choices ends up in a final state
  - not obvious how to implement!

Simulating an NFA

- Need to search all the automaton paths that are consistent with the string
- Idea: search paths in parallel
  - Keep track of subset of NFA states that the search could be in after seeing a prefix of the input
  - "Multiple fingers" pointing to graph

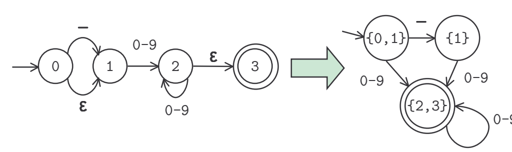
Example

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



NFA to DFA

- Automatic NFA to DFA conversion:
  - Create one DFA for each distinct subset of NFA states, e.g., {0,1}, {1}, **{2, 3}**



- Called the "subset construction"

Algorithm

- For a set S of states, define **ε**-closure(S) = states reachable from states in S by **ε**-transitions

```
T = S
Repeat   T = T U {s | s'∈T, (s',s) is ε-transition}
Until    T remains unchanged
ε-closure(S) = T
```

- For a set S of states, define DFAedge(S,c) = states reachable from S by transitions on c and **ε**-transitions

```
DFAedge(S,c) =
  ε-closure( { s | s'∈S, (s',s) is c-transition} )
```


Algorithm

```
DFAInitialState = ε-closure(NFAInitialState)
Worklist = { DFAInitialState }

While ( Worklist not empty ) :
    Pick state S from Worklist
    For each character c :
        S' = 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
```


Putting the Pieces Together