Outline

• RE review

• Construction of lexing automaton
  – DFAs, NFAs
  – DFA simulation
  – RE $\Rightarrow$ NFA conversion
  – NFA $\Rightarrow$ DFA conversion
  – (to be continued for set of prioritized REs)
Concepts

• **Tokens**: values representing lexical units of a program
  – May represent single character strings ("if", "+")
  – May represent set of strings (identifier, number)

• **Regular expressions (RE)**: concise descriptions of tokens
  – Each regular expression \( R \) describes language \( L(R) \), a set of strings corresponding to a given class of tokens
Regular Expressions

- If R and S are regular expressions, so are:
  - \( a \) for any character a
  - \( \varepsilon \) empty string
  - \( \emptyset \) the empty set
  - \( R \mid S \) (alternation: “R or S”)
  - \( RS \) (concatenation: “R followed by S”)
  - \( R^* \) (Kleene closure: “zero or more R’s”)

Regular Expression Extensions

- If \( R \) is a regular expression, so are:
  - \( R? \) \( = \varepsilon \mid R \) (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)
  - \( \text{name} = R \) named abbreviation
Automatic Lexer Generators

- **Input:** 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

```java
//
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()); }
```
How To Use Regular Expressions

• Given $R \in \mathit{RE}$ and input string $w$, need a mechanism to determine if $w \in L(R)$

$R \in \mathit{RE}$  
(that describes a token family)

Input string $w$  
(from the program)

?  

Yes, if $w$ is a token
No, if $w$ not a token

• Such a mechanism is called an acceptor
Acceptors

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

\[ w \in L \quad \text{Yes, if } w \in L \]
\[ w \not\in L \quad \text{No, if } w \not\in L \]

- **Finite Automata** are acceptors for languages described by regular expressions
Finite Automata

• Informally, finite automaton consist of:
  – A finite set of states
  – Transitions between states
  – An initial state (start state)
  – A set of final states (accepting states)

• 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, and some “spontaneous” transitions without input
DFA Example

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

- A graph

- A transition table

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>Error</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Error</td>
<td>Error</td>
</tr>
</tbody>
</table>
Simulating the DFA

- Determine if the DFA accepts an input string

```c
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 (state!=Error) && accept_states[state];
```
RE ⇒ 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

\[ a \]

\[ \epsilon \]

\[ \emptyset \]
RE → Finite automaton?

- Alternation $R|S$

- Concatenation: $RS$
NFA Definition

• A non-deterministic finite automaton (NFA) is an automaton where:
  – There may be ε-transitions (transitions that do not consume input characters)
  – There may be multiple transitions from the same state on the same input character

Example:
RE ⇒ NFA intuition

-?[0-9]+
NFA construction (Thompson)

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

  ![Diagram of canonical NFA form]

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

RS

R|S

R*

CS 412/413   Spring 2008   Introduction to Compilers
Inductive NFA Construction

$RS$

$R|S$

$R^*$
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!
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”

• Solution: search all paths in graph consistent with the string in parallel
  – Keep track of the subset of NFA states that search could be in after seeing string prefix
  – “Multiple fingers” pointing to graph
Example

- Input string: -23
- NFA states:
  \{0,1\}
  \{1\}
  \{2, 3\}
  \{2, 3\}
NFA → DFA conversion

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

• Called the “subset construction”
Algorithm

• For a set $S$ of states in the NFA, compute
  $\varepsilon$-closure($S$) = set of states reachable from states in $S$
  by one or more $\varepsilon$-transitions

  \[
  T = S \\
  \text{Repeat } T = T \cup \{s' \mid s \in T, (s,s') \text{ is } \varepsilon\text{-transition}\} \\
  \text{Until } T \text{ remains unchanged} \\
  \varepsilon\text{-closure}(S) = T
  \]

• For a set $S$ of $\varepsilon$-closed states in the NFA, compute
  DFAedge($S,c$) = the set of states reachable from states in $S$
  by transitions on symbol $c$ and $\varepsilon$-transitions

  \[
  \text{DFAedge}(S,c) = \varepsilon\text{-closure}(\{s' \mid s \in S, (s,s') \text{ is } c\text{-transition}\})
  \]
Algorithm

DFA-initial-state = ε-closure(NFA-initial-state)
Worklist = { DFA-initial-state }

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

Regular Expression $R \rightarrow$ RE $\Rightarrow$ NFA Conversion
NFA $\Rightarrow$ DFA Conversion
DFA Simulation

Input String $w \rightarrow \begin{cases} 
Yes, \text{ if } w \in L(R) \\
No, \text{ if } w \notin L(R) 
\end{cases}$
Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...), Russ Cox, January 2007