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:
  - \( \varepsilon \) empty string
  - \( a \) character \( a \)
  - \( RS \) concatenation
  - \( R|S \) 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;
int
    digit = 0/[1-9][0-9]*;
    letter = [A-Za-z];
    identifier = (letter)((letter)|(0-9|_))*
    whitespace = [ \t\n\r]+;
    (whitespace) { /* discard */ }
    (digit) { return new Token(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 \( R \)

```
Input string \( w \) in the program
Regexp \( R \) which describes a token

Yes, if \( w = \) token
No, if \( w \neq \) token

Such a mechanism is called an acceptor
```
Acceptors

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

\[
\text{Input String } w \rightarrow \text{Acceptor} \rightarrow \begin{cases} 
\text{Yes, if } w \in L \\
\text{No, if } w \notin L 
\end{cases}
\]

- 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

<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
#include <stdio.h>

int table[NSTATES][NCHARS];
int final[NSTATES];

int main()

int 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:
  - \( \epsilon \)-transitions (do not consume input characters)
  - multiple transitions from the same state on the same input character

Example:

```
\text{regexp?}
```

Thompson: RE \( \rightarrow \) NFA

- Thompson’s construction: build a finite automaton from a regular expression
- Strategy: build the NFA inductively

- Empty string \( \epsilon \):

- Single character \( a \):
Thompson’s Construction

- Alternation $R \mid S$
- Concatenation: $RS$

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 \( \varepsilon\text{-closure}(S) \) = states reachable from states in \( S \) by \( \varepsilon \)-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 states, define \( \text{DFADge}(S, c) \) = states reachable from \( S \) by transitions on \( c \) and \( \varepsilon \)-transitions

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

Algorithm

\[
\text{DFAInitState} = \varepsilon\text{-closure(NFAInitState)} \\
\text{Worklist} = \{ \text{DFAInitState} \} \\
\text{While ( Worklist not empty ) :} \\
\text{Pick state } S \text{ from Worklist} \\
\text{For each character } c : \\
S' = \text{DFADge}(S, c) \\
\text{if } (S' \text{ not in DFA states) } \\
\text{Add } S' \text{ to DFA states and Worklist} \\
\text{Add an edge } (S, S') \text{ labeled } c \text{ in DFA} \\
\text{For each DFA state } S \\
\text{If } S \text{ contains an NFA final state} \\
\text{Mark } S \text{ as DFA final state}
\]

Putting the Pieces Together

Regular Expression \( R \) \[
\text{RE } \Rightarrow \text{NFA Conversion} \\
\text{NFA } \Rightarrow \text{DFA Conversion} \\
\text{DFA Simulation} \\
\begin{cases} 
\text{Yes, if } w \in L(R) \\
\text{No, if } w \notin L(R)
\end{cases}
\]

Input String \( w \)