Compiler Design
Lexical Analysis
From Regular Expressions to Automata

conf. dr. ing. Ciprian-Bogdan Chirila
chirila@cs.upt.ro
http://www.cs.upt.ro/~chirila
Outline

- Conversion of a NFA to DFA
- Simulation of an NFA
- Construction of an NFA from a Regular Expression
From Regular Expressions to Automata

- regular expression describes
  - lexical analyzers
  - pattern processing software
- implies simulation of DFA or NFA
- NFA simulation is less straightforward

Techniques
  - to convert NFA to DFA
  - the subset construction technique
    - simulating NFA directly
    - when NFA to DFA is time consuming
  - to convert regular expression to NFA and then to DFA
Conversion of a NFA to a DFA

- subset construction
  - each state of DFA corresponds to a set of NFA states
- DFA states may be exponential in number of NFA states
- for lexical analysis NFA and DFA
  - have approximately the same number of states
  - the exponential behavior is not seen
Subset construction of an DFA from an NFA

- **Input**
  - an NFA \(N\)

- **Output**
  - DFA \(D\) accepting the same language as \(N\)

- **Method**
  - to construct a transition table \(D_{tran}\) for \(D\)
  - each state of \(D\) is a set of NFA states
  - to construct \(D_{tran}\) so \(D\) will simulate in parallel all possible moves \(N\) can make on a given input string
  - to deal with \(\varepsilon\)–transitions of \(N\) properly
# Operations on NFA states

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$-closure(s)</td>
<td>set of NFA states reachable from NFA state $s$ on $\varepsilon$-transition alone</td>
</tr>
<tr>
<td>$\varepsilon$-closure(T)</td>
<td>set of NFA states reachable from some NFA state $s$ in set $T$ on $\varepsilon$-transitions alone</td>
</tr>
<tr>
<td>move(T,a)</td>
<td>set of NFA states to which there is a transition on input symbol $a$ from some state $s$ in $T$</td>
</tr>
</tbody>
</table>
Transitions

- $s_0$ – start state
- N can be in any states of $\varepsilon$-closure($s_0$)
- reading input string $x$
  - N can be in the set of states $T$ after
- reading input $a$
  - N can go in $\varepsilon$-closure(move($T$, $a$))
- accepting states of D are all sets of N states that include at least one accepting state of N
The Subset Construction

while(there is an unmarked state T in Dstates)
{
    mark T;
    for(each input symbol a)
    {
        U=ε-closure(move(T,a));
        if (U is not in Dstates)
        {
            add U as unmarked state to Dstates;
            Dtran[T,a]=U;
        }
    }
}
Computing $\varepsilon$-closure($T$)

push all states of $T$ onto stack;
initialize $\varepsilon$-closure($T$) to $T$;
while(stack is not empty)
{
    pop $t$, the top element, off stack;
    for(each state $u$ with an edge from $t$ to $u$ labeled $\varepsilon$)
    {
        if($u$ is not in $\varepsilon$-closure($T$))
        {
            add $u$ to $\varepsilon$-enclosure($T$);
            push $u$ onto stack;
        }
    }
}
Example \((a|b)^*abb\)

- \(A = \varepsilon\text{-closure}(0)\) or \(A = \{0, 1, 2, 4, 7\}\)
Example \((a|b)^*abb\)

- \(A=\{0,1,2,4,7\}\)
- \(D_{\text{tran}}(A,a) = \varepsilon\)-closure(move(A,a))
- from \(\{0,1,2,4,7\}\) only \(\{2,7\}\) have a transition on \(a\) to \(\{3,8\}\)
Example \((a|b)^{*}abb\)

- \(D_{\text{tran}}[A,a] = \varepsilon\text{-closure}(\text{move}(A,a)) = \varepsilon\text{-closure}\{3,8\} = \{1,2,3,4,6,7,8\}\)
- \(D_{\text{tran}}[A,a] = B\)
Example \((a|b)^*abb\)

- from \(\{0,1,2,4,7\}\) only \(\{4\}\) has a transition on \(b\) to \(\{5\}\)
- \(D\text{tran}[A,b]=\varepsilon\text{-closure}(\{5\})=\{1,2,4,5,6,7\}\)
- \(D\text{tran}[A,b]=C\)
- …
Example \((a|b)^*abb\)

<table>
<thead>
<tr>
<th>NFA State</th>
<th>DFA State</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>{0,1,2,4,7}</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>{1,2,3,4,6,7,8}</td>
<td>B</td>
<td>B</td>
<td>D</td>
</tr>
<tr>
<td>{1,2,4,5,6,7}</td>
<td>C</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>{1,2,4,5,6,7,9}</td>
<td>D</td>
<td>B</td>
<td>E</td>
</tr>
<tr>
<td>{1,2,3,5,6,7,10}</td>
<td>E</td>
<td>B</td>
<td>C</td>
</tr>
</tbody>
</table>
Simulation of an NFA

- strategy in text editing programs
  - to construct a NFA from a regular expression
  - to simulate NFA using on-the-fly subset construction

- Input
  - input string $x$ terminated by $\text{eof}$
  - NFA $N$
    - start state $s_0$
    - accepting states $F$
    - transition function $\text{move}$

- Output
  - yes / no

- Method
  - to keep the current states $S$ reached from $s_0$
  - if $c$ is the next input read by $\text{nextChar()}$
  - we compute $\text{move}(S,c)$ and then we use $\varepsilon$-closure()
Algorithm: Simulating an NFA

01 $S = \varepsilon$-closure($s_0$);
02 $c = \text{nextChar}()$;
03 while ($c \neq \text{eof}$) {
04   $S = \varepsilon$-enclosure($\text{move}(S, c)$);
05   $c = \text{nextChar}()$;
06 }
07 if ($S \cap F \neq \emptyset$) return "yes";
08 else return "no";
Implementation of NFA Simulation

- two stacks each holding a set of NFA states
- a boolean array `alreadyOn`
- a two dimensional array `move[s,a]`
NFA Simulation Data Structures

- two stacks each holding a set of NFA states
  - used for the values of $S$ in both sides of assign
    - right side – oldStates
    - left side – newStates
  - $S = \varepsilon$-enclosure(move($S,c$));
  - newStates->oldStates
NFA Simulation Data Structures

- boolean array *alreadyOn*
  - indexed by NFA states
  - indicates which states are in *newStates*
  - array and stack hold the same information
  - it is much faster to interrogate the array than to search the stack

- two dimensional array *move*[s,a]*
  - the entries are set of states
  - implemented by linked lists
Implementation of step 1

01 $S=\varepsilon$-closure($s_0$);

addState($s$)
{
    push $s$ onto newStates;
    alreadyOn[$s$]=TRUE;
    for($t$ on move[$s,\varepsilon$])
        if(!alreadyOn($t$))
            addState($t$);
}
Implementation of step 4

04 \[ S = \varepsilon\text{-enclosure}(\text{move}(S, c)); \]

for (s on oldStates)
{
    for (t on move[s, c])
        if (!alreadyOn[t])
            addState(t);
    pop s from oldStates;
}

for (s on newStates)
{
    pop s from newStates;
    push s onto oldStates;
    alreadyOn[s] = FALSE;
}
Construction of an NFA from a Regular Expression

- to convert a regular expression to a NFA
- McNaughton-Yamada-Thompson algorithm
- syntax-directed
  - it works recursively up the parse tree of the regular expression
- for each subexpression a NFA with a single accepting state is built
Construction of an NFA from a Regular Expression

- **Input**
  - regular expression $r$ over an alphabet $\Sigma$

- **Output**
  - An NFA accepting $L(r)$

- **Method**
  - to parse $r$ into constituent subexpressions
  - basis rules for handling subexpressions with no operators
  - inductive rules for creating larger NFAs from subexpressions NFAs
    - union, concatenation, closure
Basis Rules for Constructing NFA

- for expression $\varepsilon$

  ![Diagram for expression $\varepsilon$]

- for expression $a$

  ![Diagram for expression $a$]
NFA for the Union of Two Regular Expressions

- $r = s \mid t$
- $N(s)$ and $N(t)$ are NFA’s for regular expressions $s$ and $t$
NFA for the Concatenation of Two Regular Expressions

- $r = st$
- $N(s)$ and $N(t)$ are NFA’s for regular expressions $s$ and $t$
Induction Rules for Constructing NFA

- \( r = s^* \)
- \( N(s) \) is the NFA for the regular expression \( s \)

- \( r = (s) \)
  - \( L(r) = L(s) \)
  - \( N(s) \) is equivalent to \( N(r) \)
Example

parse tree for (a|b)*abb
Example

- NFA for r1

- NFA for r2
Example

- NFA for $r_3 = r_1 \mid r_2$
Example

- NFA for $r5=(r3)^*$
Example

- NFA for $r7 = r5r6$

...
Bibliography