Optimization of DFA-Based Pattern Matchers

Compiler Design Lexical Analysis
asist. dr. ing. Ciprian-Bogdan Chirila
chirila@cs.upt.ro
http://www.cs.upt.ro/~chirila
Outline

- Important States of an NFA
- Functions Computed from the Syntax Tree
- Computing nullable, firstpos and lastpos
- Computing followpos
- Converting a Regular Expression Directly to a DFA
- Minimizing the Number of States of a DFA
- State Minimization of a Lexical Analyzers
- Trading Time for Space in DFA Simulation
Optimization of DFA-Based Pattern Matchers

- **First algorithm**
  - constructs a DFA directly from a regular expression
  - without constructing an intermediate NFA
  - with fewer states
  - used in Lex

- **Second algorithm**
  - minimizes the number of states of any DFA
  - combines states having the same future behavior
  - has $O(n \log(n))$ efficiency

- **Third algorithm**
  - produces more compact representations of transitions tables than the standard two dimensional ones
Important States of an NFA

- it has non-ε out transitions
- used when computing $\varepsilon$-closure($move(T,a)$) – the set of states reachable from $T$ on input $a$
- the set $moves(s,a)$ is non-empty if state $s$ is important

- NFA states are twofold if
  - have the same important states, and
  - either both have accepting states or neither does
Augmented Regular Expression

- important states
  - initial states in the basis part for a particular symbol position in the RE
  - correspond to particular operands in the RE
- Thompson algorithm constructed NFA
  - has only one accepting state which is non-important (has no out-transitions !!!)
- to concatenate a unique right endmarker # to a regular expression r
  - the accepting state of the NFA r becomes important state in the (r)# NFA
  - any state in the (r)# NFA with a transition to # must be an accepting state
Syntax Tree

- important states correspond to the positions in the RE that hold symbols of the alphabet
- RE representation as syntax tree
  - leaves correspond to operands
  - interior nodes correspond to operators
    - cat-node – concatenation operator (dot)
    - or-node – union operator |
    - star-node – star operator *
Syntax Tree Example \((a|b)^*abb#\)

cat nodes are represented as circles
Representation Rules

- syntax tree leaves are labeled by $\varepsilon$ or by an alphabet symbol
- to each leaf which is not $\varepsilon$ we attach a unique integer
  - the position of the leaf
  - the position of it's symbol
- a symbol may have several positions
  - symbol $a$ has positions 1 and 3 (on the next slide!!!)
- positions in the syntax tree correspond to NFA important states
Thompson Constructed NFA for \((a|b)^*abb\#\)

- important states are numbered
- other states are represented by letters
- the correspondence between
  - numbered states in the NFA and
  - the positions in the syntax tree
- will be presented next
Functions Computed from the Syntax Tree

- in order to construct a DFA directly from the regular expression we have to:
  - build the syntax tree
  - compute 4 functions referring (r)#
    - nullable
    - firstpos
    - lastpost
    - followpos
Computed Functions

- **nullable(n)**
  - true for syntax tree node n iff the subexpression represented by n
    - has $\varepsilon$ in its language
    - can be made null or the empty string even it can represent other strings

- **firstpos(n)**
  - set of positions in the n rooted subtree that correspond to the first symbol of at least one string in the language of the subexpression rooted at n
Computed Functions

- **lastpos(n)**
  - set of positions in the n rooted subtree that correspond to the last symbol of at least one string in the language of the subexpression rooted at n

- **followpos(n)**
  - for a position p
  - is the set of positions q such that
  - x=a_1a_2…a_n in L((r)#) such that
  - for some i there is a way to explain the membership of x in L((r)#) by matching a_i to position p of the syntax tree a_{i+1} to position q
Example

- $\text{nullable}(n) = \text{false}$
- $\text{firstpos}(n) = \{1,2,3\}$
- $\text{lastpos}(n) = \{3\}$
- $\text{followpos}(1) = \{1,2,3\}$
Computing nullable, firstpos and lastpos

<table>
<thead>
<tr>
<th>node n</th>
<th>nullable(n)</th>
<th>firstpos(n)</th>
<th>lastpos(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A leaf labeled ε</td>
<td>true</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>A leaf with position i</td>
<td>false</td>
<td>{i}</td>
<td>{i}</td>
</tr>
<tr>
<td>An or-node n=c₁</td>
<td>c₂</td>
<td>nullable(c₁) or nullable(c₂)</td>
<td>firstpos(c₁) U firstpos(c₂)</td>
</tr>
<tr>
<td>A cat-node n=c₁c₂</td>
<td>nullable(c₁) and nullable(c₂)</td>
<td>if (nullable(c₁)) firstpos(c₁) U firstpos(c₂) else firstpos(c₁)</td>
<td>if (nullable(c₂)) lastpos(c₂) U lastpos(c₁) else lastpos(c₂)</td>
</tr>
<tr>
<td>A star-node n=c₁*</td>
<td>true</td>
<td>firstpos(c₁)</td>
<td>lastpos(c₁)</td>
</tr>
</tbody>
</table>
Firstpos and Lastpos Example
Computing Followpos

A position of a regular expression can follow another position in two ways:

- if n is a **cat-node** $c_1c_2$ **(rule 1)**
  - for every position $i$ in $\text{lastpos}(c_1)$ all positions in $\text{firstpos}(c_2)$ are in $\text{followpos}(i)$

- if n is a **star-node** **(rule 2)**
  - if $i$ is a position in $\text{lastpos}(n)$ then all positions in $\text{firstpos}(n)$ are in $\text{followpos}(i)$
Followpos Example

- **Applying rule 1**
  - followpos(1) incl. {3}
  - followpos(2) incl. {3}
  - followpos(3) incl. {4}
  - followpos(4) incl. {5}
  - followpos(5) incl. {6}

- **Applying rule 2**
  - followpos(1) incl. {1,2}
  - followpos(2) incl. {1,2}
### Followpos Example Continued

<table>
<thead>
<tr>
<th>Node n</th>
<th>followpos(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{1,2,3}</td>
</tr>
<tr>
<td>2</td>
<td>{1,2,3}</td>
</tr>
<tr>
<td>3</td>
<td>{4}</td>
</tr>
<tr>
<td>4</td>
<td>{5}</td>
</tr>
<tr>
<td>5</td>
<td>{6}</td>
</tr>
<tr>
<td>6</td>
<td>∅</td>
</tr>
</tbody>
</table>
Converting a Regular Expression Directly to a DFA

- **Input**
  - a regular expression \( r \)

- **Output**
  - A DFA \( D \) that recognizes \( L(r) \)

- **Method**
  - to build the syntax tree \( T \) from \( (r)# \)
  - to compute \texttt{nullable, firstpos, lastpos, followpos}
  - to build
    - \( \text{Dstates} \) the set of DFA states
      - start state of \( D \) is \( \text{firstpos}(n_0) \), where \( n_0 \) is the root of \( T \)
      - accepting states = those containing the \# endmarker symbol
    - \( \text{Dtran} \) the transition function for \( D \)
Construction of a DFA directly from a Regular Expression

initialize Dstates to contain only the unmarked state firstpos(n₀), where n₀ is the root of syntax tree T for (r)\#;

while (there is an unmarked state S in Dstates)
{
    mark S;
    for (each input symbol a)
    {
        let U be the union of followpos(p) for all p in S that correspond to a;
        if (U is not in Dstates)
            add U as unmarked state to Dstates;
        Dtran[S,a] = U;
    }
}
Example for $r=(a|b)^*abb$

- $A=\text{firstpos}(n_0) = \{1,2,3\}$
- $D\text{tran}[A,a] = \text{followpos}(1) \cup \text{followpos}(3) = \{1,2,3,4\} = B$
- $D\text{tran}[A,b] = \text{followpos}(2) = \{1,2,3\} = A$
- $D\text{tran}[B,a] = \text{followpos}(1) \cup \text{followpos}(3) = B$
- $D\text{tran}[B,b] = \text{followpos}(2) \cup \text{followpos}(4) = \{1,2,3,5\} = C$
- ...
Example for \( r=(a|b)^*abb \)
Minimizing the Number of States of a DFA

- equivalent automata
  - \(\{A, C\} = 123\)
  - \(\{B\} = 1234\)
  - \(\{D\} = 1235\)
  - \(\{E\} = 1236\)

- exists a minimum state DFA
string $x$ distinguishes state $s$ from state $t$ if exactly one of the states reached from $s$ and $t$ by following the path $x$ is an accepting state

state $s$ is distinguishable from state $t$ if exists some string that distinguish them

the empty string distinguishes any accepting state from any non-accepting state
Minimizing the Number of States of a DFA

- **Input**
  - DFA $D$ with set of states $S$, input alphabet $\Sigma$, start state $s_0$, accepting states $F$

- **Output**
  - DFA $D'$ accepting the same language as $D$ and having as few states as possible
Minimizing the Number of States of a DFA

1. Start with an initial partition \( \Pi \) with two groups \( F \) and \( S-F \).

2. Apply the procedure
   
   \[
   \text{for(each group } G \text{ of } \Pi) \\
   \{ \\
   \text{partition } G \text{ into subgroups such that states } s \text{ and } t \text{ are in the same subgroup iff for all input symbol } a \text{ states } s \text{ and } t \text{ have transitions on } a \text{ to states in the same group of } \Pi \\
   \}
   \]

3. If \( \Pi_{\text{new}} = \Pi \) let \( \Pi_{\text{final}} = \Pi \) and continue with step 4, otherwise repeat step 2 with \( \Pi_{\text{new}} \) instead of \( \Pi \).

4. Choose one state in each group of \( \Pi_{\text{final}} \) as the representative for that group.
Minimum State DFA Construction

- the start state of $D'$ is the representative of the group containing the start state of $D$
- the accepting states of $D'$ are the representatives of those groups that contain an accepting state of $D$
- if
  - $s$ is the representative of $G$ from $\Pi_{final}$
  - exists a transition from $s$ on input $a$ is $t$ from group $H$
  - $r$ is the representative of $H$
- then
  - in $D'$ there is a transition from $s$ to $r$ on input $a$
Example

- \{A,B,C,D\}\{E\}
  - on input a:
    - A,B,C,D->\{A,B,C,D\}
    - E->\{A,B,C,D\}
  - on input b:
    - A,B,C->\{A,B,C,D\}
    - D->\{E\}
    - E->\{A,B,C,D\}
Example

- \{A,B,C\}\{D\}\{E\}
  - on input a:
    - A,B,C->\{A,B,C\}
    - D->\{A,B,C\}
    - E->\{A,BC\}
  - on input b:
    - A,C,-->\{A,B,C\}
    - B->\{D\}
    - D->\{E\}
    - E->\{A,B,C\}
Example

- \{AC\}{B}\{D\}{E}\n  - on input a:
    - A,C->\{B\}
    - B->\{B\}
    - D->\{B\}
    - E->\{B\}
  - on input b:
    - A,C,-->\{A,C\}
    - B->\{D\}
    - D->\{E\}
    - E->\{A,C\}
Example

<table>
<thead>
<tr>
<th>State</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>B</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>B</td>
<td>A</td>
</tr>
</tbody>
</table>
State Minimization in Lexical Analyzers

- to group together
  - all states that recognize a particular token
  - all states that do not indicate any token

- e.g. \{0137,7\} \{247\} \{8,58\} \{7\} \{68\} \{\emptyset\}
  - \{0137,7\} – do not indicate any token
  - \{8,58\} – announce a*b+
  - \{\emptyset\} - dead state
    - has transitions to itself on input a and b
    - is target state for states 8, 58, 68 on input a
State Minimization in Lexical Analyzers

- next, we **split**
  - 0137 from 7
    - they go to different groups on input `a`
  - 8 from 58
    - they go to different groups on input `b`

- **dead states can be dropped**
  - if we treat missing transitions as signal to end token recognition
Trading Time for Space in DFA Simulation

- transition function of a DFA
  - two dimensional table indexed by states and characters

- typical lexical analyzer has
  - hundreds of states
  - ASCII alphabet of 128 input characters
  - < 1 MB

- compilers “live” in small devices too
- 1 MB could be too much
Alternate Representations

- list of character-state pairs
- ending by a default state
  - chosen for any input character not on the list
  - the most frequently occurring next state
- thus, the table is reduced by a large factor
Bibliography