Lexical Analysis by Finite Automata

4DV006 – Compiler Construction

Dr Jonas Lundberg, office B3024

Jonas.Lundberg@lnu.se

Slides are available in Moodle

26 oktober 2014
Frontend Overview

Lexical Analysis: Identify atomic language constructs. Each type of construct is represented by a token. (e.g. 3.14 $\mapsto$ FLOAT, if $\mapsto$ IF, a $\mapsto$ ID).

Syntax Analysis: Checks if the token sequence is correct with respect to the language specification.

Semantical Analysis: Checks type relations + consistency rules. (e.g. if $\text{type}(lhs) = \text{type}(rhs)$ in an assignment $lhs = rhs$).

Each step involves a transformation from a program representation to another.
Lexical Analysis Overview

- Input program representation: Character sequence
- Output program representation: Token sequence
- Analysis specification: Regular expressions
- Recognizing (abstract) machine: Finite Automata
- Implementation: Finite Automata
# Lexical Analysis Specification

<table>
<thead>
<tr>
<th>Token</th>
<th>Patterns</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>WS</td>
<td>`(blank</td>
<td>tab</td>
</tr>
<tr>
<td>IF</td>
<td><code>if</code></td>
<td><code>genToken();</code></td>
</tr>
<tr>
<td>RelOp</td>
<td>`&lt;</td>
<td>&lt;=</td>
</tr>
<tr>
<td>ID</td>
<td><code>[a−zA−Z][a−zA−Z1−9]*</code></td>
<td><code>genToken(); updateSymTab();</code></td>
</tr>
</tbody>
</table>

- In theory: `(p_1|p_2|...|p_n)*`, where `p_i` are the above patterns, defines all lexically correct programs. \( \Rightarrow \) the set of lexically correct programs is a regular language.
- In practice: we recognize each pattern individually.
- This type of specification is input to AntLR in Practical Assignment 1.
An AntLR Lexical Specification

- AntLR is a compiler generator tool
- It reads a specification and generates a parser. (Actually, it generates Java classes implementing a parser.)
- It will be used in the practical assignment
- Below: A very simple lexical specification

```
ID : ('a'..'z'|'A'..'Z')+ ;
INT : '0'..'9'+ ;
NEWLINE: ''? '
' ;
WS : (' '|'	')+ {skip();} ;
MULT: '*' ;
ASSIGN: '=' ;
PLUS: '+' ;
MINUS: '-' ;
```
String recognition using Finite Automata

A finite automaton is an abstract machine that can be used to identify strings specified by regular expressions.

The above finite automata recognizes the pattern \((a|b)ba^*b\).

Every input character \((a\text{ or }b)\) causes a transition from one state to another.

A string is accepted if we end up in a final state once every character been processed.

No possible transition \(\Rightarrow\) rejection \(\Rightarrow\) the string is not part of the language.
Finite Automata (FA)

- A finite automata $M$ is a quintuple $M = \{Q, \Sigma, \delta, q_0, F\}$ where
  - $Q$ is a finite set of states,
  - $\Sigma$ is the input alphabet,
  - $\sigma : Q \times \Sigma \mapsto Q$ is the transition function,
  - $q_0 \in Q$ is the initial state,
  - $F$ is the set of final states.

- The graph corresponding to a given FA is called the transition graph.

- **Notice:** We can have many final states but only one initial state.
Two kinds of Automata

We have two kinds of finite automata:

1. **Deterministic FA (DFA):**
   - at most one possible transition for each input
   - no $\varepsilon$—transitions

2. **Nondeterministic FA (NFA):** If not DFA, then NFA.

An $\varepsilon$—transition indicates that we are in two states at the same time.

The following slides show one example of each type.
NFA accepting \((a|b)^* abb\)

\[
M = \{Q, \Sigma, \delta, q_0, F\}
\]

\[
Q = \{0, 1, 2, 3\}
\]

\[
\Sigma = \{a, b\}
\]

\[
q_0 = 0
\]

\[
F = \{3\}
\]

<table>
<thead>
<tr>
<th>(\delta)</th>
<th>(a)</th>
<th>(b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>({0, 1})</td>
<td>({0})</td>
</tr>
<tr>
<td>1</td>
<td>(\emptyset)</td>
<td>({2})</td>
</tr>
<tr>
<td>2</td>
<td>(\emptyset)</td>
<td>({3})</td>
</tr>
<tr>
<td>3</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
</tbody>
</table>

Corresponding transition graph

An NFA accepts (or recognizes) an input string \(x\) iff there is some path from start to final such that the edge labels along this path spell out \(x\).
A DFA accepts an input string $x$ iff, after reading $x$, we are in a final state.
DFA simulation algorithm (Algorithm 1)

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_0 )</td>
<td>Initial state</td>
</tr>
<tr>
<td>( F )</td>
<td>Set of final states</td>
</tr>
<tr>
<td>nextChar()</td>
<td>Returns next symbol from the input string, stops with EOF.</td>
</tr>
<tr>
<td>move(( q, c ))</td>
<td>Returns the state to which there is a transition from ( q ) given input ( c ).</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
q & := q_0; \\
c & := \text{nextChar}(); \\
\text{while } c \neq \text{EOF do}
\begin{align*}
q & := \text{move}(q, c); \\
c & := \text{nextChar}();
\end{align*}
\text{end while}
\end{align*}
\]

\[
\begin{align*}
\text{if } q \in F \text{ then}
\text{return } \text{"Accepted"}; \\
\text{else}
\text{return } \text{"Rejected"};
\end{align*}
\]

\[
\text{move(state,next) } \rightarrow \text{ new_state}
\begin{align*}
\text{switch (state) } & \{ \\
\text{case 0:} & \text{if (next=a) return 1; else return 0;}
\text{case 1:} & \text{if (next=a) return 1; else return 2;}
\text{case 2:} & \text{if (next=a) return 1; else return 3;}
\text{...}
\end{align*}
\]

Thus, implementing string recognition for a given DFA is straightforward.
Summary: Finite Automata

Known Theoretical Results

- **Kleene’s Theorem:** A language is recognized by a FA iff it is regular
  ⇒ For each regular expression there is a corresponding finite automata.
- For each NFA, there is a corresponding DFA.
- Given FA accepting strings defined by RE $r$, and an input string $x$. Simulation requires
  - **NFA:** Memory $O(|r|)$ and time $O(|r| \times |x|)$
  - **DFA:** Memory $O(2^{|r|})$ and time $O(|x|)$

where $|r|$ is the number of symbols and operators in $r$.

Conclusion

- NFA’s are easier to *construct* given an arbitrary regular expression.
- DFA *simulation* is both easier and faster.
- Hence, we construct an NFA, convert it into an DFA, and use the resulting DFA for simulation.
A Recipe for Scanner Construction

1. Identify what tokens $t_i$ you are interested in.
2. For each token $t_i$, write a matching regular expression $r_i$.
3. Convert the regular expressions $r_1, r_2, \ldots, r_n$ to an NFA:s $N_1, N_2, \ldots, N_n$ using Algorithm 2.
4. Put together the NFA:s $N_1, N_2, \ldots, N_n$ to a single NFA which final accepting states $F$ corresponds to the above identified lexeme groups $g_i$.
5. Convert the NFA to a DFA using Algorithm 3.

The steps 3-5 are briefly presented in the following slides. They are more thoroughly presented in the book by Aho, Ullman, and Sethi.
Example: Identifying INT, ID, and IF

Step 1 and 2: Lexical Specification

<table>
<thead>
<tr>
<th>Token</th>
<th>Sample Patterns</th>
<th>Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>if</td>
<td>if</td>
</tr>
<tr>
<td>INT</td>
<td>0, 78, 1367, 0067</td>
<td>$d^+$</td>
</tr>
<tr>
<td>ID</td>
<td>max, a, value33</td>
<td>$l(l</td>
</tr>
</tbody>
</table>

where $d$ and $l$ are character classes defined as:

- $d = [0 – 9]$
- $l = [a – zA – Z]$

Example available on course home page (see Reading Instructions)
**Step 3: RegExp → NFA (Algorithm 2)**

Input: A regular expression $r$ over an alphabet $\Sigma$
Output: An NFA $N$ accepting $L(r)$

- $r = ab$
- $r = a|b$
- $r = a^*$

- It is basically a top-down approach where each regular subexpression is replaced by a *part* of an NFA.
- The construction is straightforward but the result is often an ugly NFA that needs to be rewritten.

Algorithm 2: \((a|b)^*abb\)
Step 4: Adding NFA:s

Two possible, and equivalent, alternatives:

\[
\begin{array}{c}
\text{i} \quad \text{a} \quad \text{f} + \quad \text{i} \quad \text{b} \quad \text{f} + \quad \text{i} \quad \text{c} \quad \text{f} = \\
\text{..} \quad \text{..} \quad \text{..} + \quad \text{..} \quad \text{..} \quad \text{..} + \quad \text{..} \quad \text{..} \quad \text{..} \\
\text{b} \quad \text{a} \quad \text{f} + \quad \text{c} \quad \text{f} \quad \text{f} + \quad \text{..} \quad \text{..} \quad \text{..} \\
\text{c} \quad \text{b} \quad \text{f} + \quad \text{..} \quad \text{..} \quad \text{..} + \quad \text{..} \quad \text{..} \quad \text{..} \\
\text{..} \quad \text{..} \quad \text{..} + \quad \text{..} \quad \text{..} \quad \text{..} \\
\text{e} \quad \text{e} \quad \text{e} + \quad \text{e} \quad \text{e} \quad \text{e} \\
\text{..} \quad \text{..} \quad \text{..} + \quad \text{..} \quad \text{..} \quad \text{..} \\
\text{f} \quad \text{f} + \quad \text{f} \quad \text{f} \\
\text{or}
\end{array}
\]

The new NFA accepts all the strings accepted by the three NFAs.
NFA $\Rightarrow$ DFA (Preparation)

Now, assume that we have a single NFA recognizing all tokens we are interested in.

NFA $\Rightarrow$ DFA (Basic Idea)

- **Basic idea**: Set of NFA states $\Rightarrow$ new DFA state
- **Worst case**: $N$ NFA states $\Rightarrow 2^N$ DFA states
  (Usually not the case)

- **Algorithm outline**:
  1. DFA state 0 = all $\varepsilon$-equivalent states of NFA 0
  2. Which states can be reached from DFA 0 on input $x, y, z, \ldots$?
     $\Rightarrow$ new DFA states 1, 2, 3, \ldots
  3. Repeat (2) with DFA states 1, 2, 3, \ldots until no more DFA states are created.
Example: \((a|b)^*abb\)

NFA

DFA Construction

<table>
<thead>
<tr>
<th>DFA</th>
<th>NFA</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{0,1,3}</td>
<td>\n</td>
<td>\n</td>
</tr>
<tr>
<td>1</td>
<td>\n</td>
<td>\n</td>
<td>\n</td>
</tr>
<tr>
<td>2</td>
<td>\n</td>
<td>\n</td>
<td>\n</td>
</tr>
<tr>
<td>3</td>
<td>\n</td>
<td>\n</td>
<td>\n</td>
</tr>
<tr>
<td>4</td>
<td>\n</td>
<td>\n</td>
<td>\n</td>
</tr>
<tr>
<td>5</td>
<td>\n</td>
<td>\n</td>
<td>\n</td>
</tr>
<tr>
<td>6</td>
<td>\n</td>
<td>\n</td>
<td>\n</td>
</tr>
</tbody>
</table>
Example: \((a|b)^*abb\)

NFA

\[
\begin{array}{c}
0 \quad e \quad 1 \quad a \quad 2 \quad e \quad 3 \quad a \quad 4 \quad b \quad 5 \quad b \quad 6 \\
\end{array}
\]

DFA Construction

<table>
<thead>
<tr>
<th>DFA</th>
<th>NFA</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{0,1,3}</td>
<td>{2,4,1,3} ^{1}</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>{1,2,3,4}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>{1,2,3}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>{1,2,3}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>{1,2,3}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>{1,2,3}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>{1,2,3}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\^{1} \text{Indeterminate states}
Example: \((a|b)^*abb\)

NFA

DFA Construction

<table>
<thead>
<tr>
<th>DFA</th>
<th>NFA</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{0,1,3}</td>
<td>{2,4,1,3}(^1)</td>
<td>{2,1,3}(^2)</td>
</tr>
<tr>
<td>1</td>
<td>{1,2,3,4}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>{1,2,3}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NFA-to-DFA
Lexical Analysis by Finite Automata
Example: \((a|b)^*abb\)

NFA

DFA Construction

<table>
<thead>
<tr>
<th>DFA</th>
<th>NFA</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{0,1,3}</td>
<td>(2,4,1,3)(^1)</td>
<td>(2,1,3)(^2)</td>
</tr>
<tr>
<td>1</td>
<td>{1,2,3,4}</td>
<td>(2,4,1,3)(^1)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>{1,2,3}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: \((a|b)^*abb\)

NFA

DFA Construction

<table>
<thead>
<tr>
<th>DFA</th>
<th>NFA</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{0,1,3}</td>
<td>{2,4,1,3} (^1)</td>
<td>{2,1,3} (^2)</td>
</tr>
<tr>
<td>1</td>
<td>{1,2,3,4}</td>
<td>{2,4,1,3} (^1)</td>
<td>{2,5,1,3} (^3)</td>
</tr>
<tr>
<td>2</td>
<td>{1,2,3}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>{1,2,3,5}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: \((a|b)^*abb\)

NFA

\[
\begin{array}{c}
0 \xrightarrow{e} 1 \xrightarrow{a} 2 \xrightarrow{e} 3 \xrightarrow{a} 4 \xrightarrow{b} 5 \xrightarrow{b} 6 \\
\end{array}
\]

DFA Construction

<table>
<thead>
<tr>
<th>DFA</th>
<th>NFA</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{0,1,3}</td>
<td>{2,4,1,3} ^1</td>
<td>{2,1,3} ^2</td>
</tr>
<tr>
<td>1</td>
<td>{1,2,3,4}</td>
<td>{2,4,1,3} ^1</td>
<td>{2,5,1,3} ^3</td>
</tr>
<tr>
<td>2</td>
<td>{1,2,3}</td>
<td>{2,4,1,3} ^1</td>
<td>{2,1,3} ^2</td>
</tr>
<tr>
<td>3</td>
<td>{1,2,3,5}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NFA-to-DFA

The Software Technology Group

Lexical Analysis by Finite Automata
Example: \((a|b)^*abb\)

DFA Construction

<table>
<thead>
<tr>
<th>DFA</th>
<th>NFA</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{0,1,3}</td>
<td>{2,4,1,3}(^1)</td>
<td>{2,1,3}(^2)</td>
</tr>
<tr>
<td>1</td>
<td>{1,2,3,4}</td>
<td>{2,4,1,3}(^1)</td>
<td>{2,5,1,3}(^3)</td>
</tr>
<tr>
<td>2</td>
<td>{1,2,3}</td>
<td>{2,4,1,3}(^1)</td>
<td>{2,1,3}(^2)</td>
</tr>
<tr>
<td>3</td>
<td>{1,2,3,5}</td>
<td>{2,4,1,3}(^1)</td>
<td>{2,6,1,3}(^4)</td>
</tr>
<tr>
<td>4</td>
<td>{1,2,3,6}</td>
<td>{2,4,1,3}(^1)</td>
<td>{2,1,3}(^2)</td>
</tr>
</tbody>
</table>
### Step 5: NFA $\rightarrow$ DFA  (Algorithm 3)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>closure($s$)</td>
<td>States reachable from $s$ on $\varepsilon$-transitions - $s$ included</td>
</tr>
<tr>
<td>move($T$, $a$)</td>
<td>$\varepsilon$-equivalent states reachable from state set $T$ after input $a$</td>
</tr>
<tr>
<td>$\delta_{DFA}$, $Q_{DFA}$</td>
<td>transition table and set of states for the DFA to be constructed</td>
</tr>
</tbody>
</table>

add closure($q_0$) as an unmarked state to $Q_{DFA}$

while there is an unmarked state $T$ in $Q_{DFA}$ do
  mark $T$
  for each input symbol $a$ do
    $U :=$ move($T$, $a$)
    if $U \notin Q_{DFA}$ then
      add $U$ as an unmarked state to $Q_{DFA}$
    end if
    $\delta_{DFA}[T, a] := U$
  end for
end while
DFA for \((a|b)^*abb\)

Construction

<table>
<thead>
<tr>
<th>DFA</th>
<th>NFA</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{0,1,3}</td>
<td>{2,4,1,3}</td>
<td>{2,1,3}²</td>
</tr>
<tr>
<td>1</td>
<td>{1,2,3,4}</td>
<td>{2,4,1,3}</td>
<td>{2,5,1,3}³</td>
</tr>
<tr>
<td>2</td>
<td>{1,2,3}</td>
<td>{2,4,1,3}</td>
<td>{2,1,3}²</td>
</tr>
<tr>
<td>3</td>
<td>{1,2,3,5}</td>
<td>{2,4,1,3}</td>
<td>{2,6,1,3}⁴</td>
</tr>
<tr>
<td>4</td>
<td>{1,2,3,6}</td>
<td>{2,4,1,3}</td>
<td>{2,1,3}²</td>
</tr>
</tbody>
</table>

Result

<table>
<thead>
<tr>
<th>DFA</th>
<th>a</th>
<th>b</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>start</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>final</td>
</tr>
</tbody>
</table>
DFA for \((a|b)^*abb\)

**Construction**

<table>
<thead>
<tr>
<th>DFA</th>
<th>NFA</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{0,1,3}</td>
<td>{2,4,1,3}(^1)</td>
<td>{2,1,3}(^2)</td>
</tr>
<tr>
<td>1</td>
<td>{1,2,3,4}</td>
<td>{2,4,1,3}(^1)</td>
<td>{2,5,1,3}(^3)</td>
</tr>
<tr>
<td>2</td>
<td>{1,2,3}</td>
<td>{2,4,1,3}(^1)</td>
<td>{2,1,3}(^2)</td>
</tr>
<tr>
<td>3</td>
<td>{1,2,3,5}</td>
<td>{2,4,1,3}(^1)</td>
<td>{2,6,1,3}(^4)</td>
</tr>
<tr>
<td>4</td>
<td>{1,2,3,6}</td>
<td>{2,4,1,3}(^1)</td>
<td>{2,1,3}(^2)</td>
</tr>
</tbody>
</table>

**Result**

<table>
<thead>
<tr>
<th>DFA</th>
<th>a</th>
<th>b</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>start</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>final</td>
</tr>
</tbody>
</table>

NFA-to-DFA
Scanner Summary

Scanner Construction:

1. Identify relevant (token, RegExp) pairs
2. Convert all RegExp:s to a single NFA
3. Convert the NFA to a DFA
4. Implement a scanner for the DFA

Notice: The steps 2-4 can be written as algorithms
⇒ A scanner can be generated for given set of RegExp:s
⇒ This is what happens inside AntLR.
Written assignment (WA1)

Construct a DFA accepting signed integers and decimals.

<table>
<thead>
<tr>
<th>Item</th>
<th>Valid Expressions</th>
<th>Erroneous Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integers</td>
<td>34, 0, −12346</td>
<td>02, +67, −0</td>
</tr>
<tr>
<td>Floats</td>
<td>3.14, 0.02, −12.23, .47, -0.2</td>
<td>02.45, 23. , +2.1</td>
</tr>
</tbody>
</table>

Notice: We accept .47 for 0.47 but not 23. for 23.0.

- Use the algorithms presented at this lecture.
  1. Identify relevant RegExps \( r_{\text{int}}, r_{\text{float}} \) (No Algorithm!)
  2. Convert \( r_{\text{int}}, r_{\text{float}} \) to NFAs \( N_{\text{int}} \) and \( N_{\text{float}} \)
  3. Add \( N_{\text{int}} \) and \( N_{\text{float}} \) to a single NFA recognizing both ints and floats
  4. Convert the NFA to a DFA

- Act like a machine that executes the algorithms without thinking
  ⇒ No clever rewritings in between the steps.

- More information available at the course web site.

Deadline: 2014-11-16

Good Luck!