Lexical Analysis

Implementation: Finite Automata
Outline

• Specifying lexical structure using regular expressions
• Finite automata
  – Deterministic Finite Automata (DFAs)
  – Non-deterministic Finite Automata (NFAs)
• Implementation of regular expressions
  RegExp => NFA => DFA => Tables
Notation

- There is variation in regular expression notation
  - Union: \[ A \mid B \] \equiv A + B
  - Option: \[ A \varepsilon \] \equiv A?
  - Range: \[ a' + b' + \ldots + z' \] \equiv [a-z]
  - Excluded range:
    complement of \[ [a-z] \] \equiv [^a-z]
Regular Expressions in Lexical Specification

• Given a string $s$ and a reg. exp. $R$, is $s \in L(R)$?

• But a yes/no answer is not enough!

• Instead: partition the input into tokens

• We adapt regular expressions to this goal
Regular Expressions => Lexical Spec.

1. Write a rexp for the lexemes of each token
   – Number = digit +
   – Keyword = ‘if’+ ‘else’+ ...
   – Identifier = letter (letter + digit)*
   – OpenPar = ‘(‘
   – ...

Regular Expressions => Lexical Spec.

2. Construct $R$, matching all lexemes for all tokens

$R = \text{Keyword} + \text{Identifier} + \text{Number} + \ldots$

$= R_1 + R_2 + \ldots$
Regular Expressions => Lexical Spec.

3. Let input be $x_1...x_n$
   For $1 \leq i \leq n$ check
   
   $x_1...x_i \in L(R)$

4. If success, then we know that
   
   $x_1...x_i \in L(R_j)$ for some $j$

5. Remove $x_1...x_i$ from input and go to (3)
Ambiguities (1)

- There are ambiguities in the algorithm
- How much input is used? What if
  - $x_1...x_i \in L(R)$ and also
  - $x_1...x_K \in L(R)$
- e.g. = and ==
- Rule: Pick longest possible string in $L(R)$
  - The “maximal munch”
  - We as humans do that.
Ambiguities (2)

• Which token is used? What if
  – \( x_1 \ldots x_i \in L(R_j) \) and also
  – \( x_1 \ldots x_i \in L(R_k) \)

• e.g. ‘if’ could be an identifier or a keyword;
• which one to choose?
• Rule: use rule listed first (\( j \) if \( j < k \))
  – Treats “if” as a keyword, not an identifier
• i.e. the one listed first is given higher priority
Error Handling

• What if
  – No rule matches a prefix of input?
• Problem: Can’t just get stuck ...
• A compiler needs to give feedback to the user e.g. where the error is in the file (line number)
• Solution:
  – Write a rule matching all “bad” strings
  – Put it last (lowest priority)
Summary

• Regular expressions provide a concise notation for string patterns

• Use in lexical analysis requires small extensions
  – To resolve ambiguities
  – To handle errors

• Good algorithms known
  – Require only single pass over the input
  – Few operations per character (table lookup)
Finite Automata

• Regular expressions = specification
• Finite automata = implementation
• A finite automaton consists of
  – An input alphabet $\Sigma$
  – A set of states $S$
  – A start state $n$
  – A set of accepting states $F \subseteq S$
  – A set of transitions $\text{state } \rightarrow^{\text{input}} \text{state}$
Finite Automata

• Transition
  \[ s_1 \rightarrow^a s_2 \]
• Is read
• In state \( s_1 \) on input “a” go to state \( s_2 \)
• If end of input and in accepting state => accept
• Otherwise => reject
  – If it terminates in state \( s \) that not a member of F
  – Or it gets stuck because there is not transition from state \( s_1 \) on input a (i.e. never reaches the end of input).
Finite Automata State Graphs

- A state
- The start state
- An accepting state
- A transition
A Simple Example

• A finite automaton that accepts only “1”
Another Simple Example

• A finite automaton accepting any number of 1’s followed by a single 0
• Alphabet: \{0,1\}
And Another Example

• Alphabet \{0,1\}

• What language does this recognize?
Epsilon Moves

• Another kind of transition: $\varepsilon$-moves

• Machine can move from state A to state B without reading input
Deterministic and Nondeterministic Automata

- **Deterministic Finite Automata (DFA)**
  - One transition per input per state
  - No $\varepsilon$-moves

- **Nondeterministic Finite Automata (NFA)**
  - Can have multiple transitions for one input in a given state
  - Can have $\varepsilon$-moves
Execution of Finite Automata

• A DFA can take only one path through the state graph
  – Completely determined by input

• NFAs can choose
  – Whether to make $\varepsilon$-moves
  – Which of multiple transitions for a single input to take
Acceptance of NFAs

• An NFA can get into multiple states

- Input: 1 0 0 0
- States: \{A\} \{A,B\} \{A,B,C\}
- Rule: NFA accepts if it can get to a final state
NFA vs. DFA

• NFAs and DFAs recognize the same set of languages (regular languages)

• DFAs are faster to execute
  – There are no choices to consider
NFA vs. DFA

- For a given language NFA can be simpler than DFA

- DFA can be exponentially larger than NFA
Regular Expressions to Finite Automata

- High-level sketch

Diagram:
- Regular expressions
  - Lexical Specification
- NFA
- DFA
  - Table-driven Implementation of DFA
Regular Expressions to NFA (1)

- For each kind of rexp, define an NFA
  - Notation: NFA for rexp $M$

  ![NFA diagram]

- For $\varepsilon$

  ![NFA diagram]

- For input $a$

  ![NFA diagram]
Regular Expressions to NFA (2)

- For $AB$

- For $A + B$
Regular Expressions to NFA (3)

- For $A^*$
Example of RegExp -> NFA conversion

• Consider the regular expression

  \((1+0)^*1\)

• The NFA is
ε-closure of a state

• ε-closure of a state s is a set of states that consists of s and all other states that I can reach from s by making ε-moves only.

• Example
  – ε-closure(B) = {B, C, D}
  – ε-closure(G) = {G, H, I, A, B, C, D}
NFA to DFA. Remark

- An NFA may be in many states at any time
- How many different states?
- If there are N states, the NFA must be in some subset of those N states
- How many subsets are there?
  - $2^{N-1}$ i.e., finitely many
<table>
<thead>
<tr>
<th>NFA</th>
<th>DFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>States: $S$</td>
<td>States: subset of $S$</td>
</tr>
<tr>
<td>Start state: $s \in S$</td>
<td>Start state: $\varepsilon$-closure($s$)</td>
</tr>
<tr>
<td>Final states: $F$ subset of $S$</td>
<td>Final state: ${ X \mid X \cap F \neq \emptyset }$</td>
</tr>
<tr>
<td>The transition function:</td>
<td>The transition function:</td>
</tr>
<tr>
<td>$a(x) = { y \mid x \in X \land x \xrightarrow{a} y }$</td>
<td>$X \xrightarrow{a} Y$ if $Y = \varepsilon$-closure($a(X)$)</td>
</tr>
</tbody>
</table>
NFA to DFA: The Trick

- Simulate the NFA
- Each state of DFA
  - = a non-empty subset of states of the NFA
- Start state
  - = the set of NFA states reachable through ε-moves from NFA start state
- Add a transition $S \rightarrow^a S'$ to DFA iff
  - $S'$ is the set of NFA states reachable from any state in $S$ after seeing the input $a$, considering ε-moves as well
NFA -> DFA Example
Implementation

• A DFA can be implemented by a 2D table $T$
  – One dimension is “states”
  – Other dimension is “input symbol”
  – For every transition $S_i \rightarrow^a S_k$ define $T[i,a] = k$

• DFA “execution”
  – If in state $S_i$ and input $a$, read $T[i,a] = k$ and skip to state $S_k$
  – Very efficient
Table Implementation of a DFA
algorithm

i=0;
State=0;
While(input[i]){  
    State=A[state, input[i++]];  
}
Implementation of the NFA itself

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>$\xi$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$\epsilon$</td>
<td>$B$</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>$1$</td>
<td>$E$</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>$0$</td>
<td>$F$</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>$\epsilon$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Diagram:

```
A -> B -> C -> E -> G -> H -> I -> J
B -> D -> F -> G
```

Table:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>$\xi$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$\epsilon$</td>
<td>$\epsilon$</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>$\epsilon$</td>
<td>$1$</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>$\epsilon$</td>
<td>$0$</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>$\epsilon$</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>$\epsilon$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>$\epsilon$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: The table represents transitions in the NFA, where each column corresponds to an input symbol.
Trade off between speed and space

- **DFAs**
  - Faster: we are in one state at any given time.
  - less compact: there could be a large number of states $2^N-1$.

- **NFAs**
  - slower (the loop has to deal with set of states rather than one state),
  - concise