Today’s Agenda

- Last time we reviewed languages and grammars, and briefly started discussing regular expressions.
- Today I’ll restart the regular expression discussion, since it felt a bit rushed.
- I’ll then describe how to build finite automata that recognize regular expressions.
- On Monday, I’ll discuss how scanners are implemented.
Announcements

• Homework 1 will be out later today.
  – I’ll post on course website and send email.
  – Due next Friday (January 18).
• First part of the project (the scanner) will be assigned early next week.

Regular Expressions

• Defined over some alphabet $\Sigma$
  – For programming languages, alphabet is usually ASCII or Unicode
• If $re$ is a regular expression, $L(re)$ is the language (set of strings) generated by $re$
### Fundamental REs

<table>
<thead>
<tr>
<th>( re )</th>
<th>( L(re) )</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>{ a }</td>
<td>Singleton set, for each symbol ( a ) in the alphabet ( \Sigma )</td>
</tr>
<tr>
<td>( \varepsilon )</td>
<td>{ \varepsilon }</td>
<td>Empty string</td>
</tr>
<tr>
<td>( \emptyset )</td>
<td>{}</td>
<td>Empty language</td>
</tr>
</tbody>
</table>

These are the basic building blocks that other regular expressions are built from.

---

### Operations on REs

<table>
<thead>
<tr>
<th>( re )</th>
<th>( L(re) )</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>( rs )</td>
<td>( L(r)L(s) )</td>
<td>Concatenation: a string from ( r ) followed by a string from ( s )</td>
</tr>
<tr>
<td>( r</td>
<td>s )</td>
<td>( L(r) \cup L(s) )</td>
</tr>
<tr>
<td>( r^* )</td>
<td>( L(r)^* )</td>
<td>Kleene closure: sequence of 0 or more strings from ( r )</td>
</tr>
</tbody>
</table>

Precedence: \( * \) (highest), concatenation, \( | \) (lowest) Parentheses can be used to group REs as needed
**Examples**

<table>
<thead>
<tr>
<th>re</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>single + character</td>
</tr>
<tr>
<td>!</td>
<td>single ! character</td>
</tr>
<tr>
<td>!=</td>
<td>2 character sequence “!=“</td>
</tr>
<tr>
<td>xyzzy</td>
<td>5 character sequence “xyzzy”</td>
</tr>
<tr>
<td>(1</td>
<td>0)*</td>
</tr>
<tr>
<td>(1</td>
<td>0)(1</td>
</tr>
<tr>
<td>0</td>
<td>1(1</td>
</tr>
</tbody>
</table>

**Abbreviations**

The basic operations generate all possible regular expressions, but there are common abbreviations used for convenience. Some examples:

<table>
<thead>
<tr>
<th>Abbr.</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>r+</td>
<td>(rr*)</td>
<td>1 or more occurrences</td>
</tr>
<tr>
<td>r?</td>
<td>(r</td>
<td>ε )</td>
</tr>
<tr>
<td>[a-z]</td>
<td>(a</td>
<td>b</td>
</tr>
<tr>
<td>[abxyz]</td>
<td>(a</td>
<td>b</td>
</tr>
</tbody>
</table>
Exercise:
What do these represent?

<table>
<thead>
<tr>
<th>re</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[abc]+</td>
<td></td>
</tr>
<tr>
<td>[abc]*</td>
<td></td>
</tr>
<tr>
<td>[0-9]+</td>
<td></td>
</tr>
<tr>
<td>[1-9][0-9]*</td>
<td></td>
</tr>
<tr>
<td>[a-zA-Z][a-zA-Z0-9]*</td>
<td></td>
</tr>
</tbody>
</table>

Abbreviations

• Many systems allow abbreviations to make writing and reading definitions or specifications easier

  name ::= re

  – Restriction: abbreviations may not be circular (recursive) either directly or indirectly (else would be not be a regular language)
  • digit ::= [0-9] is okay
  • number ::= digit number is not
Example

• Possible syntax for numeric constants

\[
\begin{align*}
\text{digit} & :=[0-9] \\
\text{digits} & := \text{digit}+ \\
\text{number} & := \text{digits} \ (, \text{digits})? \\
& \quad ( [eE] (\ + \mid -)? \text{digits} ) ?
\end{align*}
\]

• Notice that this allows (unnecessary) leading 0s, e.g., 00045.6. (0, or 0.14 would be necessary 0s.)

• How would you prevent that?

Example

• Possible syntax for numeric constants

\[
\begin{align*}
\text{digit} & :=[0-9] \\
\text{nonzero_digit} & := [1-9] \\
\text{digits} & := \text{digit}+ \\
\text{number} & := (0 \mid \text{nonzero_digit} \text{digits})? \\
& \quad ( . \text{digits} )? \\
& \quad ( [eE] (\ + \mid -)? \text{digits} ) ?
\end{align*}
\]
Recognizing REs

- Finite automata can be used to recognize languages generated by regular expressions
- Can build by hand or automatically
  - Reasonably straightforward, and can be done systematically
  - Tools like Lex, Flex (for compilers written in C++), and JFlex (for compilers written in Java) do this automatically, given a set of REs.

Finite State Automaton

- Review from your CS theory class ...
- A finite set of states
  - One marked as initial state
  - One or more marked as final states
  - States sometimes labeled or numbered
- A set of transitions from state to state
  - Each labeled with symbol from $\Sigma$ (the alphabet), or $\varepsilon$
  - The symbols correspond to characters in the input stream.
Finite State Automaton

- Operate by reading input symbols (usually characters)
  - Transition can be taken if labeled with current symbol
  - $\varepsilon$-transition can be taken at any time
- Accept when final state reached and no more input
  - Slightly different in a scanner, where the FSA is used as a subroutine to find the longest input string that matches a token RE.
- Reject if no transition possible, or no more input and not in final state (DFA)

Example: FSA for “pig”
DFA vs NFA

- Deterministic Finite Automata (DFA)
  - No choice of which transition to take
- Non-deterministic Finite Automata (NFA)
  - Choice of transition in at least one case
  - \( \varepsilon \) transitions (arcs): If the current state has any outgoing \( \varepsilon \) arcs, we can follow any of them without consuming any input
  - Accept if some way to reach a final state on given input
  - Reject if no possible way to final state
  - Modeling choice option 1: guess path, backtrack if rejects
  - Option 2: “clone” at choice point, accept if any clone accepts

Example NFA

Input 1: GOSEAHAWKS

Status: Executing...
FAs in Scanners

- Want DFA for speed (no backtracking or cloning)
- But conversion from regular expressions to NFA is easier
- Luckily, there is a well-defined procedure for converting an NFA to an equivalent DFA

From RE to NFA: base cases

These correspond to the “Fundamental REs” shown earlier.

- NFA for symbol ‘a’
- NFA for empty string (ε)
- NFA for empty set (∅)
Concatenation: \( r s \)

The idea: When we find a string that matches the regular expression \( r \), we start trying to match the regular expression \( s \). Since this is an NFA, it’s okay if we guess wrong – we will make an \( \varepsilon \) transition from every prefix of the input that matches \( r \), and thus check all possible matches.

Union/Combination: \( r | s \)

The idea: Non-deterministically check if the input matches either \( r \) or \( s \). If either sub-machine reaches a final state, jump to the union machine’s final state. If the entire input has been consumed at this point (i.e., the entire string matches \( r \) or \( s \)), the union machine will accept.
Kleene star: $r^*$

The idea: At the start node (N1), we attempt to match either the empty string (to account for the possibility of zero occurrence of $r$) or a single match of $r$. Every time the $r$ machine find a potential match, it non-deterministically jumps back to N1 and repeats the process. Since this is an NFA, it’s okay if we guess the wrong match of $r$ – we’ll try all of them.

Example

• Draw the NFA for $(ab | c)$:
Example

• Draw the NFA for \((ab|c)\):

```
  a ─> O ─> b
  ^          |
  |          v
  c ─> O
```

Example

• Draw the NFA for \((ab|c)\):

```
  a ─> ε ─> b ─> ε ─> O
  ^          |
  |          v
  c ─> O
```
• Draw the NFA for \((ab|c)\):

\[
\begin{array}{c}
\text{a} \quad \text{b} \\
\end{array}
\]

(If a state has a single outgoing \(\varepsilon\)-transition, and no other transitions, you can merge it into the target.)

Example

Winter 2013  UW CSE 401 (Michael Ringenburg)
Exercise

• Draw the NFA for:  b(at | ag) | bug

From NFA to DFA

• Subset construction: construct a DFA from the NFA, where each DFA state represents a set of NFA states
• Key idea: the state of the DFA after reading some input is the set of all NFA states that could have reached after reading the same input
• Algorithm: example of a fixed-point computation
  – Find ε-closure (all states reachable via 0 or more ε-transitions) of start state. Create a DFA state corresponding to this set. Add it to the unvisited list.
  – While there exist unvisited DFA states, select one (call it \( d \)):
    • For each symbol \( s \) in the alphabet, determine the NFA states reachable by any NFA state in the set corresponding to \( d \).
    • Determine the ε closure of these states. Create a transition from \( d \) on symbol \( s \) to a state corresponding to this closure set.
    • If the state corresponding to this set is new, add it to the unvisited list.
Example

- Convert NFA to a DFA:

\[
\begin{array}{c}
\text{1, 2, 5} \\
\end{array}
\]
• Convert NFA to a DFA:

Example
Example

• Convert NFA to a DFA:

Winter 2013
UW CSE 401 (Michael Ringenburg)
Next Time

• Implementing a scanner
  – By hand
  – Via automated tools
• Enjoy your weekend
  – Go Hawks!