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.
• Office hours selected, starting next week:
  – Laure: Mondays (except 1/21 & 2/18), 4-5, CSE 218
  – Mike: Wednesdays, 2:30-3:30, CSE 212
    • Or by appointment on Tuesdays
  – Zach: Fridays, 1:30-2:30, CSE 218

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 Σ</td>
</tr>
<tr>
<td>ε</td>
<td>{ ε }</td>
<td>Empty string</td>
</tr>
<tr>
<td>Ø</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) ∪ 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>Sequence of one or more a’s, b’s and c’s</td>
</tr>
<tr>
<td>[abc]</td>
<td>Zero or more a’s, b’s, and c’s</td>
</tr>
<tr>
<td>[0-9]+</td>
<td>Non-negative integer (possibly with leading 0s)</td>
</tr>
<tr>
<td>[1-9][0-9]*</td>
<td>Positive integer (no leading 0s)</td>
</tr>
<tr>
<td>[a-zA-Z][a-zA-Z0-9_]*</td>
<td>One or more letters or digits, must start with a letter.</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

  digit ::= [0-9]
digits ::= digit+
number ::= digits ( . digits )?
  ( [eE] (+ | -)? digits ) ?

• 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

```plaintext
digit ::= [0-9]
nonzero_digit ::= [1-9]
digits ::= digit+
number ::= (0 | nonzero_digit digits?)
               ( . digits )?
               ( [eE] (+ | -)? digits ) ?
```

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”

Input 1: pig

Status: Executing...
Example: FSA for “pig”

Input 1: pig

Status: Executing...

Example: FSA for “pig”

Input 1: pig

Status: Executing...
Example: FSA for “pig”

Input 1: pig

Status: Accept! (In a final state, and no more input.)

Example: FSA for “pig”

Input 2: pit

Status: Executing...
Example: FSA for “pig”

Input 1: pit

Status: Executing...

Example: FSA for “pig”

Input 1: pit

Status: Executing...
Example: FSA for “pig”

Input 1: pit

Status: Reject! (No legal transitions on ‘t’.)

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...

Winter 2013
UW CSE 401 (Michael Ringenburg)

Example NFA

Input 1: GOSEAHAWKS

Status: Executing...

Winter 2013
UW CSE 401 (Michael Ringenburg)
Example NFA

Input 1: GOSEAHAWKS

Status: Executing...

Example NFA

Input 1: GOSEAHAWKS

Status: Executing...
Example NFA

Input 1: GOSEAHAWKS

Status: Executing...

Winter 2013  UW CSE 401 (Michael Ringenburg)
Example NFA

Input 1: GOSEAHAWKS

Status: Executing...

Winter 2013
UW CSE 401 (Michael Ringenburg)
Example NFA

Input 1: GOSEAHAWKS

Status: Executing...

Winter 2013
UW CSE 401 (Michael Ringenburg)
Example NFA

Input 1: GOSEAHAWKS

Status: Executing...

Winter 2013  UW CSE 401 (Michael Ringenburg)
Example NFA

Input 1: GOSEAHAWKS

Status: Accept!

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 \( ε \) transition from every prefix of the input that matches \( r \), and thus check all possible matches.
Union/Combination: $r \mid s$

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 \((ab | c)\):

\[ a \rightarrow \epsilon \rightarrow b \rightarrow \epsilon \]

\[ c \rightarrow \]

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

• Draw the NFA for \((ab|c)\):

```
\[ \begin{array}{c}
\text{S} \\
\varepsilon \\
a \rightarrow b \\
\varepsilon \\
\varepsilon \\
c \rightarrow \varepsilon \\
\varepsilon \\
\text{F}
\end{array} \]
```

Exercise

• Draw the NFA for: \(b(at|ag) | \text{bug}\)
Exercise

• Draw the NFA for: \( b(at|ag) \mid bug \)
Exercise

• Draw the NFA for: \( b(at|ag) \mid bug \)
From NFA to DFA

• Subset construction: construct a DFA from an NFA. Each DFA state represents a set of NFA states.
• Key idea: State of DFA after reading some input is the set of all states that NFA could have reached after reading the same input
• Algorithm (example of a fixed-point computation):
  – Find $\varepsilon$-closure (all states reachable via 0 or more $\varepsilon$-transitions) of start state. Create DFA state corresponding to this set. Add to 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 $\varepsilon$ closure of these states. Create a transition from $d$ on symbol $s$ to a DFA state corresponding to this closure set.
    • If this state is new, add to the unvisited list.

Example

• Convert NFA to a DFA:
Example

- Convert NFA to a DFA:

Epsilon closure of start state

Visit \( \{1,2,5\} \): Transitions on 'a'.
No \( \varepsilon \) transitions from 3.
Example

• Convert NFA to a DFA:

Visit \{1,2,5\}: Transitions on 'c'.

Epsilon closure of \{6\}
Example

- Convert NFA to a DFA:

![Diagram of NFA and DFA conversion]

Done with \{1,2,5\}

Visit \{3\}: Just one transition. Do ε closure of new state. Mark \{3\} as visited.
Example

- Convert NFA to a DFA:

Last two states have no transitions, but contain a final state, so mark as final.

Next Time

- Implementing a scanner
  - By hand
  - Via automated tools

- Enjoy your weekend
  - Go Hawks!