Topic #12: Regular Expressions
CSE 413, Autumn 2004
Programming Languages

http://www.cs.washington.edu/education/courses/413/04au/

Programming Language Specifications

• Since the 1960s, the syntax of every significant programming language has been specified by a formal grammar
  » First done in 1959 with BNF (Backus-Naur Form or Backus-Normal Form) used to specify the syntax of ALGOL 60
  » Borrowed from the linguistics community

Grammar for a Tiny Language

program := statement | program statement
statement := assignStmt | ifStmt
assignStmt := id = expr;
ifStmt := if ( expr ) stmt
expr := id | int | expr + expr
id := a | b | c | i | j | k | n | x | y | z
int := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Productions

• Meaning of
  nonterminal ::= <sequence of terminals and nonterminals>
  » “In a derivation, the non-terminal on the left can be replaced by the expression on the right”
• Often, there are two or more productions for a single nonterminal – can use either at different times
• Alternative notations:
  ifStmt ::= if ( expr ) stmt
  ifStmt → if ( expr ) stmt
  <ifStmt> ::= if ( <expr> ) <stmt>
Parsing

- Parsing: reconstruct the derivation (syntactic structure) of a program
- In principle, a single recognizer could work directly from the concrete, character-by-character grammar
- Instead:

Why Separate the Scanner and Parser?

- Simplicity & Separation of Concerns
  » Scanner hides details from parser (comments, whitespace, input files, etc.)
  » Parser is easier to build; has simpler input stream
- Efficiency
  » Scanner can use simpler, faster design

Principle of Longest Match

- In most languages, the scanner should pick the longest possible string to make up the next token if there is a choice
- Example
  
  ```
  return for bar != begin ning;
  ```
  should be recognized as 5 tokens

Languages & Automata Theory

- Alphabet: a finite set of symbols
- String: a finite, possibly empty sequence of symbols from an alphabet
- Language: a set, often infinite, of strings
- Finite specifications of (possibly infinite) languages
  » Automaton – a recognizer; a machine that accepts all strings in a language (and rejects all other strings)
  » Grammar – a generator; a system for producing all strings in the language (and no other strings)
- A language may be specified by many different grammars and automata
- A grammar or automaton specifies only one language

Regular Expressions and Finite Automata

- The lexical grammar (structure) of most programming languages can be specified with regular expressions
  » Sometimes a little ad-hoc “cheating” is useful
- Tokens can be recognized by a deterministic finite automaton
  » Can be either table-driven or built by hand based on lexical grammar

Regular Expressions

- Defined over some alphabet \( \Sigma \)
- If \( re \) is a regular expression, \( L(re) \) is the language (set of strings) generated by \( re \)
- Note that this is opposite of the way we often think about regular expressions
  » either way, the relevant set of strings is \( L(re) \)
**Fundamental Regular Expressions**

<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 a in $\Sigma$</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>

**Operations on Regular Expressions**

<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</td>
</tr>
<tr>
<td>r+s</td>
<td>$L(r)+L(s)$</td>
<td>Combination (union)</td>
</tr>
<tr>
<td>r*</td>
<td>$L(r)^*$</td>
<td>0 or more occurrences (Kleene closure)</td>
</tr>
</tbody>
</table>

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

**Abbreviations**

- The basic operations generate all possible regular expressions, but there are common abbreviations used for convenience.
  - Typical examples:
    - 1 of the given characters (a|b|x|y|z) [abxyz]
    - 1 character in given range (a|b|…|z) [a-z]
    - 0 or 1 occurrence (r | $\varepsilon$) r?
    - 1 or more occurrences (rr*) r+

<table>
<thead>
<tr>
<th>Abbr.</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>r+</td>
<td>$(r^*)$</td>
<td>1 or more occurrences</td>
</tr>
<tr>
<td>r?</td>
<td>$(r</td>
<td>x)$</td>
</tr>
<tr>
<td>[a-z]</td>
<td>(ab)…[x]</td>
<td>1 character in given range</td>
</tr>
<tr>
<td>[abxyz]</td>
<td>(ab)…(yz)</td>
<td>1 of the given characters</td>
</tr>
</tbody>
</table>

**Examples**

<table>
<thead>
<tr>
<th>re</th>
<th>$L(re)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>single character a</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>single character $\varepsilon$</td>
</tr>
<tr>
<td>![≥][!]=</td>
<td>specific 2-character sequence ![≥][!]=</td>
</tr>
<tr>
<td>![≥][!]=</td>
<td>a 2-character sequence: ![≥][!]=, $\le$, or $\ge$</td>
</tr>
<tr>
<td>![≥]</td>
<td>single character ![≥]</td>
</tr>
<tr>
<td>hogwash</td>
<td>7 character sequence</td>
</tr>
</tbody>
</table>

**More Examples**

<table>
<thead>
<tr>
<th>re</th>
<th>$L(re)$</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**

- Can name regular expressions to make writing and reading definitions easier, e.g.,
  - $\text{digit} := [0-9]$
    - Restriction: abbreviations may not be circular (recursive) either directly or indirectly
- Example: possible syntax for numeric constants
  - $\text{number} := \text{digits} (, \text{digits})? (\text{[eE]} (+ | -)? \text{digits})?$
  - $\text{digits} := \text{digit}+$
  - $\text{digit} := [0-9]$
Recognizing Regular Expressions

- Finite automata can be used to recognize strings generated by regular expressions
- Can build by hand or automatically
  - Not totally straightforward, but can be done systematically
  - Tools like Lex, Flex, and JLex do this automatically, given a set of REs

Finite State Automaton

- A finite set of states
  - One marked as initial state
  - One or more marked as final states
- A set of transitions from state to state
  - Each labeled with symbol from Σ, or ε
- Operate by reading input symbols (usually characters)
  - Transition can be taken if labeled with current symbol
  - ε-transition can be taken at any time

Accept or Reject

- Accept
  - if in final state and
    - no more input, or
    - no valid transition for the next symbol
- Reject
  - if not in final state and
    - no more input, or
    - no valid transition for the next symbol

DFA vs NFA

- Deterministic Finite Automata (DFA)
  - No choice of which transition to take under any condition
- Non-deterministic Finite Automata (NFA)
  - Choice of transition in at least one case
  - Accept if some way to reach final state on given input
  - Reject if no possible way to final state

Finite Automata in Scanners

- Want DFA for speed (no backtracking)
- Conversion from regular expressions to NFA is straightforward
- There is a procedure for converting a NFA to an equivalent DFA

From RE to NFA: base cases

regex: a

regex: ε
Exercise: Write NFA for a*[bc]d

Exercise: Write NFA for [ab]d*c

Exercise: Write NFA for b*{abc}*dc

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 states the NFA could have reached after reading the same input
- If NFA has \( n \) states, DFA has at most \( 2^n \) states
  - \( \Rightarrow \) DFA is finite, can construct in finite \# steps
- Resulting DFA may have more states than needed
  - See the books for construction and minimization details
Simple DFA example

• Idea: show a hand-written DFA for some typical programming language constructs
  » Can use to construct hand-written scanner
• Setting: Scanner is called whenever the parser needs a new token
  » Scanner stores current position in input
  » Starting there, use a DFA to recognize the longest possible input sequence that makes up a token and return that token

Scanner DFA Example (1)

Scanner DFA Example (2)

Scanner DFA Example (3)

Scanner DFA Example (4)

Strategies for handling identifiers vs keywords

• Hand-written scanner: look up identifier-like things in table of keywords to classify (good application of perfect hashing)
• Machine-generated scanner: generate DFA will appropriate transitions to recognize keywords
  • Lots ‘o states, but efficient (no extra lookup step)