Regular Expressions

CSE 413, Autumn 2005
Programming Languages

http://www.cs.washington.edu/education/courses/413/05au/

Agenda for Today

- Basic concepts of formal grammars
- Regular expressions
- Lexical specification of programming languages
- Using finite automata to recognize regular expressions

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

- The rules of a grammar are called *productions*
- Rules contain
  - Nonterminal symbols: grammar variables (*program*, *statement*, *id*, etc.)
  - Terminal symbols: concrete syntax that appears in programs (a, b, c, 0, 1, if, (...)
- Meaning of
  
  _nonterminal_ ::= <sequence of terminals and nonterminals>

  In a derivation, an instance of _nonterminal_ can be replaced by the sequence of terminals and nonterminals on the right of the production

- Often, there are two or more productions for a single nonterminal – can use either at different times

Alternative Notations

- There are several syntax notations for productions in common use
  - all mean the same thing
  - “the non-terminal on the left can be replaced by the expression on the right”

  _ifStmt_ ::= _if_ ( _expr_ ) _stmt_
  _ifStmt_ → _if_ ( _expr_ ) _stmt_
  _<ifStmt>_ ::= _if_ ( _<expr>_ ) _<stmt>_
Parsing & Scanning

- In real compilers the recognizer is split into two phases
  - Scanner: translate input characters to tokens
    - Also, report lexical errors like illegal characters and illegal symbols
  - Parser: read token stream and reconstruct the derivation

Recall: Characters vs Tokens

- Input text
  ```
  // this line is a simple comment
  if (x >= y) y = 42;
  ```
- Token Stream
  ```
  IF | LPAREN | ID(x) | OP_GEQ | ID(y) |
  RPAREN | ID(y) | OP_ASSIGN | INT(42) | SCOLON
  ```
  - Note: tokens are atomic items, not character strings
    - objects of class Token

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
    - But still often consumes a surprising amount of the compiler’s total execution time

Tokens

- Idea: we want a distinct token type (*lexical class*) for each distinct terminal symbol in the programming language
  - Examine the grammar to find these
- Some tokens may have attributes
  - Examples:
    - integer literal token will have the actual integer value (17, 42,...) as an attribute
    - identifiers will have a string with the actual id as an attribute and perhaps some type information
Typical Programming Language Tokens

- **Operators & Punctuation**
  - `+ - * / ( ) { } [ ] ; : < <= == != ! ...`
  - Each of these is a distinct lexical class
- **Keywords (reserved)**
  - `if while for goto return switch void ...`
  - Each of these is also a distinct lexical class (not a string)
- **Identifiers**
  - A single ID lexical class, but parameterized by actual id
- **Integer literals**
  - A single INT lexical class, but parameterized by int value
- **Other constants, etc.**

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 forbar != beginning;
  ```
  
  should be recognized as 5 tokens

  | RETURN | ID(forbar) | NEQ | ID(beginning) | SCOLON |

  not more (i.e., not parts of words or identifiers, or ! and = as separate 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 $\Sigma$
  » For programming languages, commonly ASCII or Unicode
• 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
  » generating strings vs matching strings
  » 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

| $re$ | $L(re)$ | Notes
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>rs</td>
<td>$L(r)L(s)$</td>
<td>Concatenation</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>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.

<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>$\varepsilon$)</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>
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>!</td>
<td>single character !</td>
</tr>
<tr>
<td>!=</td>
<td>specific 2-character sequence !=</td>
</tr>
<tr>
<td>![&lt;&gt;]=</td>
<td>a 2-character sequence: !=, &lt;=, or &gt;=</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

- Many systems allow naming the regular expressions to make writing and reading definitions easier
  
  name ::= re
  
  for example
  
  digit ::= [0-9]

  » Restriction: abbreviations may not be circular (recursive) either directly or indirectly

Example

- Possible syntax for numeric constants

  number ::= digits ( . digits )? ( [eE] (+ | -)? digits ) ?
  
  digits ::= digit+
  
  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 $\Sigma$, or $\varepsilon$
- Operate by reading input symbols (usually characters)
  » Transition can be taken if labeled with current symbol
  » $\varepsilon$-transition can be taken at any time

Example: FSA for “cat”

Accept or Reject

- Accept
  » if final state reached and no more input
  » if in an accepting state when no valid transition for the next symbol or no more input
- Reject
  » if no more input and not in final state
  » if no transition possible and not in accepting state
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)
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 with appropriate transitions to recognize keywords
  • Lots 'o states, but efficient (no extra lookup step)