regular grammars

Daniel Jackson
September 29, 2008
designing a quizzer
designing a quizzer

want a console program that

• reads a file of questions
• displays them one by one
• accepts user responses
• prompts again if out of bounds
• reports a total and message

What is the capital of France?
(a) Paris
(b) London
(c) Brussels

Must give answer between a and c. Try again.

b
Which of these bodies of water is the largest?
(a) Pacific Ocean
(b) North Sea
(c) Walden Pond

Which of these is not a continent?
(a) Asia
(b) Africa
(c) North America
(d) Finland

Which city is at approximately the same latitude as Boston?
(a) Rio de Janeiro
(b) Rome
(c) Amsterdam

What is a geocache?
(a) A form of payment that geologists use
(b) A place that things are hidden in a treasure-hunting game
(c) A memory used to accelerate the performance of GPS devices

b
You scored 3 points.
You're obviously a real globe trotter.
a state machine design

what's right with this?
  · it captures the correct traces

what's wrong with this?
  · it doesn’t show any structure
  · hard to see that you can loop on a question
another attempt

using Statecharts notation

- can show repeat attempts inside the state corresponding to one question
- but this can get clumsy
what’s going on?

consider a trace such as

```xml
<
displayQuestion, badAnswer,
displayErrMsg, goodAnswer, displayQuestion, badAnswer,
displayErrMsg, goodAnswer, displayQuestion, badAnswer,
displayErrMsg, goodAnswer,
>
```
hierarchical structure

we can structure it like this:

```xml
<
displayQuestion, badAnswer,
displayErrMsg, badAnswer,
goodAnswer,
displayQuestion,
goodAnswer,
displayQuestion,
badAnswer,
displayErrMsg,
goodAnswer>
```

Session

Question

BadAnswers
as a grammar

grammar

• defines set of traces

  Session ::= Question*
  Question ::= \texttt{displayQuestion} \texttt{BadAnswers} \texttt{goodAnswer}
  BadAnswers ::= (\texttt{badAnswer} \texttt{displayErrMsg})*

easily converted to code

• very roughly:

  void Session() {while (...) Question();}
  void Question() {\texttt{displayQuestion}(); \texttt{BadAnswers}(); \texttt{goodAnswer} }
  void BadAnswers() {while (...) {\texttt{badAnswer}; \texttt{displayErrMsg}}}
regular grammars
grammars

sentences

• a grammar defines a set of sentences
• a sentence is a sequence of symbols or terminals

productions, terminals and non-terminals

• a grammar is a set of productions
• each production defines a non-terminal
• a non-terminal is a variable that stands for a set of sentences
grammar

URL ::= Protocol :// Address
Address ::= Domain . TLD
Protocol ::= http | ftp
Domain ::= mit | apple | pbs
TLD ::= com | edu | org

terminals are

://, .., http, ftp, mit, apple, pbs, com, edu, org

non-terminals and their meanings

TLD = { com, edu, org }
Domain = { mit, apple, pbs }
Protocol = { http, ftp }
Address = { mit.com, mit.edu, mit.org, apple.com, apple.edu, apple.org, pbs.com, pbs.edu, pbs.org }
operators

production has form

• non-terminal ::= expression in terminals and non-terminals

expression operators

• sequence: an A is a B followed by a C
  \[ A ::= B \ C \]

• iteration: an A is zero or more B’s
  \[ A ::= B^* \]

• choice: an A is a B or a C
  \[ A ::= B \mid C \]

• option: an A is a B or is empty
  \[ A ::= B? \]
examples

a two-way switch

\[
\text{SWITCH ::= (up down)*}
\]

a Java identifier

\[
\text{Identifier ::= Letter (Letter | Digit)*}
\]

\[
\text{Letter ::= a | b | ... | Z}
\]

\[
\text{Digit ::= 0 | 1 | ... | 9}
\]

file handling protocol

\[
\text{FILE ::= open (read | write)* close?}
\]

trailing whitespace

\[
\text{TRAIL ::= (space | tab)* newline}
\]
JSP form

“JSP form”
- no ‘mixed productions’
- each is sequence, iteration, choice or option
- has nice diagrammatic form
- good basis for code synthesis

element

EXAMPLE

\[
\text{SWITCH ::= TOGGLE*} \quad \text{// SWITCH is an iteration} \\
\text{TOGGLE ::= up down} \quad \text{// TOGGLE is a sequence}
\]
diagram syntax for grammars

notation also called

- “structure diagram”
- “entity life history”

\[ A ::= B \cdot C \]
\[ A ::= B^* \]
\[ A ::= B | C \]
\[ A ::= B ? \]
example: quizzer

**textual grammar**

Session ::= Question*

Question ::= displayQuestion BadAnswers goodAnswer

BadAnswers ::= BadAnswer*

BadAnswer ::= badAnswer displayErrMsg

**diagrammatic grammar**

SESSION

QUESTION*

BADANSWERS

display Question

BADANSWER*

bad Answer

display ErrMsg

good Answer
grammars, machines, regexps
expressions vs. grammars

can always write a regular grammar in one production
  • then the RHS is called a regular expression
  • consists of terminals and operators

\[
\text{SWITCH ::= (up down)*}
\]

can also write a grammar with only sequence and choice
  • but may need special symbol for empty string \( \varepsilon \)
  • and allow recursion on the left (or right) but not both

\[
\text{SWITCH ::= } \varepsilon | (\text{up down SWITCH})
\]
  • (in general multiple productions)
grammars and machines

regular grammars vs state machines

- A state machine’s trace set is prefix closed: if $t^e$ is a trace, so is $t$
- Regular grammars can express trace sets that are not prefix closed
  
  Traces of $\text{(up down)}^*$ include $\langle \text{up, down} \rangle$ but not $\langle \text{up} \rangle$
- So grammars are more expressive

but can add “final” states to state diagrams

- Then define (full) traces as those that go from initial to final states
- Now grammars and machines are equally expressive
- They both define regular languages

in practice

- Use state machines for non-terminating systems
- Use grammars for terminating and non-terminating systems
grammars

language definitions

• usually include grammars
• not usually regular, but context free

a context free grammar

• just like a regular grammar in basic form
• but recursion can be anywhere

```
Statement:
  Block
  if ParExpression Statement
[else Statement]
  for ( ForInitOpt ; [Expression] ; ForUpdateOpt ) Statement
  while ParExpression Statement
  do Statement while ParExpression ;
  try Block ( Catches | [Catches] finally Block )
  switch ParExpression
  { SwitchBlockStatementGroups }
  synchronized ParExpression Block
  return [Expression] ;
  throw Expression ;
  break [Identifier]
  continue [Identifier] ;
  ExpressionStatement
  Identifier : Statement
```
regular expression matching

widely used in programming tools

- in Java, can use REs for parsing strings
  
  see String.split, java.util.Regex

- built-in to scripting languages such as Perl, Python, Ruby

available for find/replace in many editors

- remove trailing whitespace
  
  find pattern:  \[\t \]*\r  \[\t \]* means an iteration of tab or space
  replace pattern:  \r  replace by just newline character

- convert Pascal comment to Java/C comment
  
  find pattern:  {([^}\]*})  [^}\]* means an iteration of any char that is not }
  replace pattern:  /* \1 */  \1 binds to all chars that matched inside ()
summary
principles

for a structured stream

• use a grammar, not a state machine

regexps, grammars and state machines

• have equivalent expressive power
• but that doesn’t mean they’re equally natural on a given problem

key idea of a grammar

• grammar describes a set of sentences
• this set can be composed from smaller sets associated with non-terminals
references

for more information on regular grammars

• http://en.wikipedia.org/wiki/Regular_grammar
• http://en.wikipedia.org/wiki/Regular_language
• http://en.wikipedia.org/wiki/Regular_expression