coding stream processors with JSP

Daniel Jackson
October 1, 2009
last time

we saw how a machine can be described with a grammar

• set of traces of quiz machine

\[
\begin{align*}
\text{Session} & : \Rightarrow \text{Question}^* \\
\text{Question} & : \Rightarrow \text{displayQuestion} \ \text{BadAnswers} \ \text{goodAnswer} \\
\text{BadAnswers} & : \Rightarrow (\text{badAnswer} \ \text{displayErrMsg})^*
\end{align*}
\]

and we noted that you can easily convert to code

• in rough outline

\[
\begin{align*}
\text{void Session()} & \{ \text{while (...) Question();} \} \\
\text{void Question()} & \{ \text{displayQuestion(); BadAnswers(); goodAnswer} \} \\
\text{void BadAnswers()} & \{ \text{while (...) \{ badAnswer; displayErrMsg \}} \}
\end{align*}
\]

in today’s class

• I’ll explain how to do this in detail using the JSP method
review
review: the idea of 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
review: grammar syntax

production has form

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

regular expression operators

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

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

• choice: an A is a B or a C
  \[ A ::= B | C \]  sometimes instead of |, choices are put on separate lines

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

this kind of grammar is a regular grammar, in Extended Backus Naur Form
examples

a two-position 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

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

trailing whitespace

\[
\text{TRAIL ::= (space | tab)}^* \text{ newline}
\]
grammatical subtlety

grammar (below) and diagram (right) for a two-position switch

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

these have subtly different meaning!

- the grammar’s sentences are not the same as the traces of the machine
- a trace is a history up to some point; any prefix of a sentence is a trace
  - sentences: \(<>, <\text{up, down}>, <\text{up, down, up, down}>, \ldots\)
  - traces: \(<>, <\text{up}>, <\text{up, down}>, <\text{up, down, up}>, \ldots\)

so grammar actually contains more information

- for program that terminates (eg \text{FILE}), this information is useful
- it tells us when the program can terminate
- for non-terminating programs (eg, \text{SWITCH}) we just ignore it
JSP diagram syntax
JSP form

“JSP form”

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

diagram

```
SWITCH ::= TOGGLE*
// SWITCH is an iteration
TOGGLE ::= up down
// TOGGLE is a sequence
```
diagram syntax for grammars

notation also called

• “structure diagram”
• “entity life history”

A ::= B C
A ::= B*
A ::= B | C
A ::= B ?
how to write code to read a stream
basic recipe

basic idea

• structure code around grammar of input events

steps

• draw input grammar in JSP form
• construct list of operations for reading inputs, updating state, generating outputs
• assign operations to grammar components
• write conditions
• read code off annotated grammar
  sequence becomes statements in sequence
  iteration becomes while loop
  choice becomes if-then-else
example: quizzer

diagrammatic grammar

SESSION

QUESTION*

display Question

BADANSWERS

good Answer

BADANSWER*

bad Answer
display ErrMsg

textual grammar

Session ::= Question*

Question ::= displayQuestion BadAnswers goodAnswer

BadAnswers ::= BadAnswer*

BadAnswer ::= badAnswer displayErrMsg

© Daniel Jackson 2008
assigning operations

1. read line
2. initialize score
3. incr score
4. display error msg
5. display question
6. display score

© Daniel Jackson 2008
assigning reads

only tricky part is where to put the reads
  · often need lookahead to evaluate conditions
  · usually one lookahead is enough

in this case
  · read after display
  · read again after bad answer

1 read line
2 initialize score
3 incr score
4 display error msg
5 display question
6 display score
conditions

two conditions needed
  • for termination of QUESTION*
    while more questions in quiz
  • for termination of BADANSWER*

now just transcribe into code
  • follow the structure:

    while (more questions) {
      // displayQuestion
      while (answer is bad)
        // badAnswer
        // displayErrMsg
    }
    // goodAnswer
private static void runQuiz(Quiz quiz) throws IOException {
    for (Question question : quiz.getQuestions()) {
        System.out.println(question); // display question
        readLine(); // read line
        int maxIndex = 'a' + question.getNumOptions() - 1;
        while (!isEOF() && badResponse(maxIndex)) {
            System.out.println // display error message
                ("Answer must be between a and " + (char)maxIndex);
            readLine(); // read line
        }
        char choice = response.charAt(0); // increment
        score += question.getScore(choice); // ... score
    }
    System.out.println(quiz.interpretScore(score)); // display score
}

private static boolean badResponse(int maxIndex) {
    return response.length() != 1
        || response.charAt(0) < 'a'
        || response.charAt(0) > maxIndex;
}
designing a file format:
a more detailed example
designing operations

where do the operations come from?
  • from datatypes we invent
  • more on this later in the course

considerations
  • what objects do we need to construct?
    Quiz: the quiz object derived from the file
    Question: one of these for each question
    Option: one for each option of each question
    Scorer: for interpreting the total score
  • what observations do we need to make about them?
origins of operations

• for construction of objects
  Quiz from Questions
  Question from text and Options
  Option from text and value
  Scorer from range and message

  --> Quiz.new
  --> Question.new
  --> Option.new
  --> Scorer.new, Scorer.fromRange

• for observation of objects
  from Quiz, get Questions
  from Question, get text, num options
  from Question, get score for choice
  from Quiz, get interpretation of score

  --> Quiz.getQuestions
  --> Question.toString, getNumOptions
  --> Question.getScore(index)
  --> Quiz.interpretScore

• for internal observations
  from Option, get text and value
  from Scorer, get interpretation of score

  --> Option.getValue, toString
  --> Scorer.interpretScore
```java
public class Quiz implements Iterable<Question> {
    public Quiz (Scorer scorer, List<Question> questions)
    public String interpretScore (int score)
    public List<Question> questions()
}

public class Question {
    public Question (String text, List<Option> options)
    public int getNumOptions ()
    public int getScore (char index)
    public String toString ()
}

public class Option {
    public Option (String optionText, int value)
    public int getValue()
    public String toString ()
}

public class Scorer {
    public String interpretScore (int score)
    public Scorer rangeElse (final int lo, final int hi, final String msg)
}
```
designing a grammar

considerations

‣ should it be human readable?
‣ should it be human writeable?
‣ easy to parse (one lookahead)
‣ easy to detect errors (redundant)

issues arising in this case

‣ how to delineate new option? new question? scoring rules?
‣ allow linebreaks in options?
‣ predicate syntax? <, >?
‣ missing scoring rules?

What is the capital of France?
[1]Paris
[0]London
[0]Brussels

Which of these bodies of water is the largest?
[1]Pacific Ocean
North Sea
Walden Pond

Which of these is not a continent?
Asia
Africa
North America
[1]Finland

Which city is at approximately the same latitude as Boston?
Rio de Janeiro
[1]Rome
Amsterdam

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

0-1: Don't worry. Most people don't know anything either.
2-3: You're obviously a real globe trotter.
4-5: You know so much, you could be President!
decisions

choices I made and their rationale

• use linebreaks for end of question stem, option, scoring rule
  makes parsing easier, and gives cleaner look (fewer blank lines or special marks)
  prevents multiline messages, but displayer can break into lines
  allows option value to be omitted, since linebreak delineates option

• require all scoring predicates to be simple range, eg 0-3
  easier to parse, extra flexibility not very useful

• delineate option values [0] and scoring ranges 0-3: with special chars
  makes parsing easier, allows easy checking that these are numeric

• allow any number of blank lines between questions
  no harm to give a bit more flexibility

• allow scoring rules to be omitted
  just use a default message if no applicable rule
the grammar

plan to parse individual lines by random access, so use diagram only for structure down to lines express line structure textually:

Option ::= Value? Text
Value ::= [ digit+ ]
Text ::= char*
Rule ::= Range Message
Range ::= digit+ - digit+ :
Message ::= char*
where we are heading

grammar gives code structure immediately

while (more questions) {
    // process STEM
    while (more options)
        // process OPTION
    while (more blank lines)
        // process BLANK
}
while (more rules)
    // process RULE

issue is how to fill it out

• what operations, and in what order?
• what are the loop conditions?
primary state variables

- list of questions so far
  \[\text{List<Question> questions}\]
- list of options so far
  \[\text{List<Option> options}\]
- scorer so far
  \[\text{Scorer scorer}\]
- next line to be processed
  \[\text{String nextLine}\]

how to enumerate operations

- one variable at a time
- initializations, updates, finalizations
- then operations these depend on

operations

1. \(\text{nextLine} = \text{reader.readLine()}\)
2. \(\text{questions} = \text{new List<Question>()}\)
3. \(\text{questions.add}(q)\)
4. \(\text{scorer} = \text{new Scorer()}\)
5. \(\text{scorer} = \text{scorer.rangeElse}(\text{lo, hi, msg})\)
6. \(\text{quiz} = \text{new Quiz(questions, scorer)}\)
7. \(\text{options} = \text{new List<Option>()}\)
8. \(\text{options.add}(o)\)
9. \(q = \text{new Question(stem, options)}\)
10. \(o = \text{new Option (opt, val)}\)
11. \(\text{lo} = \ldots\text{nextLine}\ldots\)
12. \(\text{hi} = \ldots\text{nextLine}\ldots\)
13. \(\text{msg} = \ldots\text{nextLine}\ldots\)
14. \(\text{stem} = \ldots\text{nextLine}\ldots\)
15. \(\text{opt} = \ldots\text{nextLine}\ldots\)
16. \(\text{val} = \ldots\text{nextLine}\ldots\)
assigning operations

for each operation, ask

• how many times should it be executed?
• once per what?
• before which other ops?

then

• choose grammar element
• place operation in position

example

• 3 and 9 are once per QUESTION, at end
• 9 must go before 3
  3 questions.add(q)
  9 q = new Question(stem, options)
assigning operations, ctd

other examples

2 questions = new List<Question>()     // once per QUIZ, at start
3 questions.add(q)     // once per QUESTION, at end
4 scorer = new Scorer()    // once per QUIZ, at start
5 scorer = scorer.rangeElse(lo, hi, msg) // once per RULE
6 quiz = new Quiz(questions, scorer)  // once per QUIZ, at end
7 options = new List<Option>()   // once per QUESTION, at start
8 options.add(o)       // once per OPTION
9 q = new Question(options)    // once per QUESTION, at end
10 o = new Option (opt, val)    // once per OPTION
assignments, all but read

```java
1 nextLine = reader.readLine()
2 questions = new List<Question>()
3 questions.add(q)
4 scorer = new Scorer()
5 scorer = scorer.rangeElse(lo, hi, msg)
6 quiz = new Quiz(questions, scorer)
7 options = new List<Option>()
8 options.add(o)
9 q = new Question(stem, options)
10 o = new Option (opt, val)
11 lo = ...nextLine...
12 hi = ...nextLine...
13 msg = ...nextLine...
14 stem = ...nextLine...
15 opt = ...nextLine...
16 val = ...nextLine...
```
assigning the read operation

single readahead rule

• read once at the start
• read once after each record is consumed
completed assignments

1. nextLine = reader.readLine()
2. questions = new List<Question>()
3. questions.add(q)
4. scorer = new Scorer()
5. scorer = scorer.rangeElse(lo, hi, msg)
6. quiz = new Quiz(questions, scorer)
7. options = new List<Option>()
8. options.add(o)
9. q = new Question(stem, options)
10. o = new Option (opt, val)
11. lo = ...nextLine...
12. hi = ...nextLine...
13. msg = ...nextLine...
14. stem = ...nextLine...
15. opt = ...nextLine...
16. val = ...nextLine...
conditions

for each iteration or choice
  write a condition

examples

QUESTION*: !isEOF() && nextLine.charAt(0) != '['
OPTION*: !isBlankLine()
BLANK*: isBlankLine()
RULE*: !isEOF()

using auxiliary predicates, eg:

private boolean isBlankLine () {
  return nextLine != null && nextLine.trim().length()==0;
}
private boolean isEOF () {
  return nextLine==null;
}
putting it all together

now just read code off the diagram!

' can introduce methods for boxes

diagram:

example: QUESTION

7 options = new List<Option>()
14 stem = ...nextLine...
1 nextLine = reader.readLine()
while (!isBlankLine())
    readOption();
while (isBlankLine())
    1 nextLine = reader.readLine();
3 questions.add(q)
9 q = new Question(stem, options)
summary
principles

for a structured stream
• use a grammar, not a state machine
• state is in program counter, not coded explicitly in variables

derive code structure from stream structure
• express stream structure as regular grammar
• define operations, and assign to grammar elements
• read code off annotated structure

if in doubt, read ahead
• novices usually read too late; leads to ugly ifs and gotos
• design grammars for single lookahead if possible
references

on the JSP technique

・ the textbook (in COBOL!): http://www.amazon.com/Principles-Program-Design-APIC-Jackson/dp/0123790506
・ ‘getting it wrong’: http://mcs.open.ac.uk/mj665/GetWrong.pdf

some online materials

・ http://www.jacksonworkbench.co.uk/stevefergspages/jackson_methods