6.005 Elements of Software Construction
Fall 2008

6.005 elements of software construction

designing stream processors

Daniel Jackson

This lecture is the last in the series on state machines. It takes a different view -- how to implement a state machine that processes a stream of events, using the control structure of code. In contrast to the machine implementations based on the state and transition structure, this approach uses the implicit state in the program counter, defined by the nested structure of loops, conditions, etc.

The two important lessons in the lecture are (a) the idea of grammars and regular expressions and how to use them to model trace sets, and (b) the JSP method for synthesizing code from grammars.

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
e
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
a
Which of these is not a continent?
(a) Asia
(b) Africa
(c) North America
(d) Finland
d
Which city is at approximately the same latitude as Boston?
(a) Rio de Janeiro
(b) Rome
(c) Amsterdam
c
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 simple program but not a trivial one -- will illustrate many of the issues involved in reading streams. There will be two streams to consider: the stream of inputs from the user when actually running the quiz, and the stream of records read from the file template for the quiz. The first one is simpler, so we start with that.

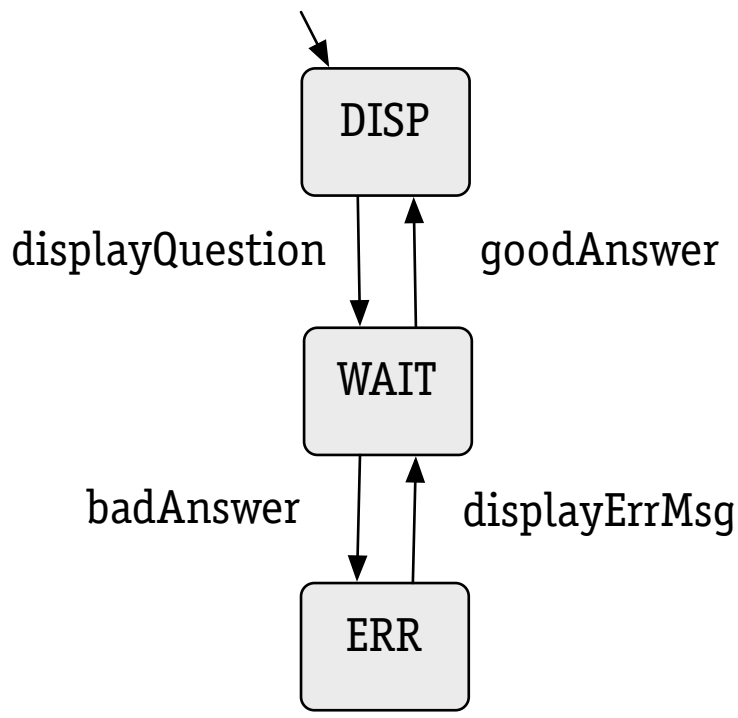
a state machine design

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

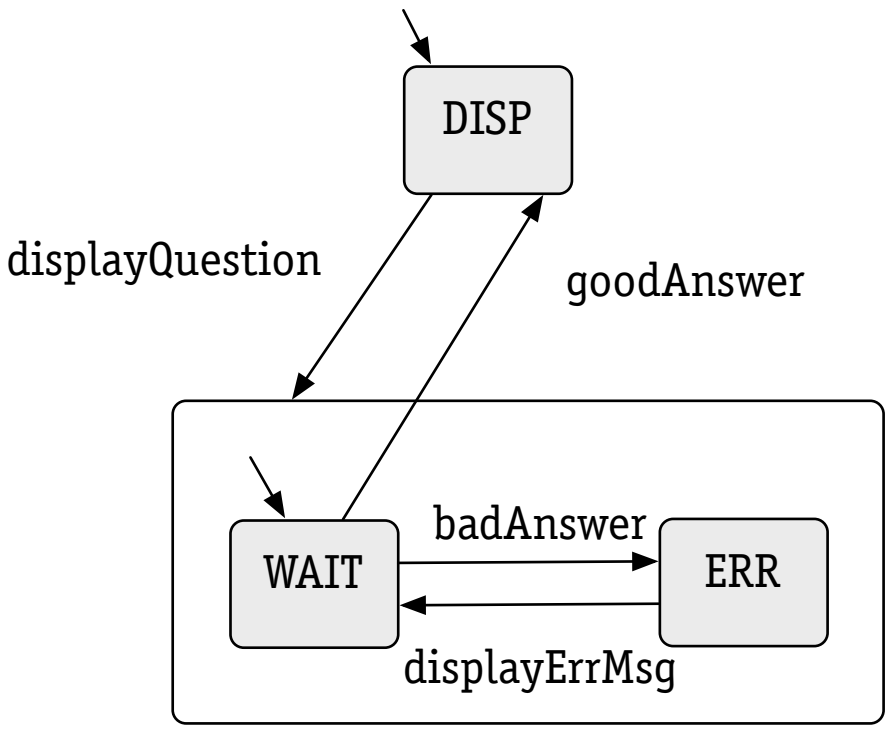


How do we model the behavior of the quizzer with a state machine? We can do it, but it fails to show the structure inherent in the problem. There's no indication here that the repeated handling of bad answers is done once per question -- that the bad answer handling is nested inside the question handling.

another attempt

using Statecharts notation

- can show repeat attempts inside the state corresponding to one question
- but this can get clumsy

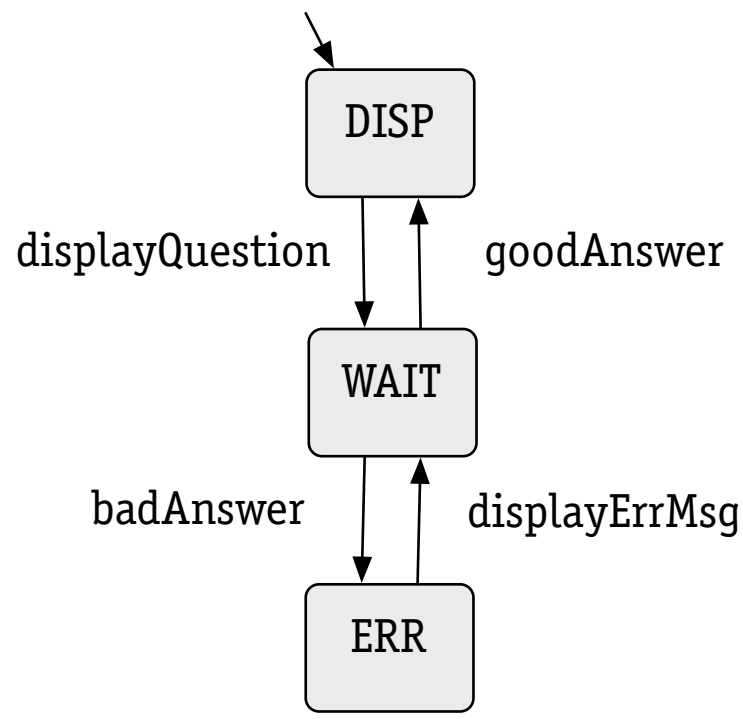


We can use the syntactic extensions of Statecharts to introduce some structure, and now we can see that the bad answer handling is a sort of subroutine. But this is not a good approach in general. First, there isn't a convention for how to do this systematically -- to prevent arbitrary exits out of the 'subroutine' for example -- and it can easily become a mess. Second, it doesn't scale very well. Third, the structure is still somewhat obscured. We need to observer that there's a cyclic path to see that there's an iteration going on here.

what's going on?

consider a trace such as

<
displayQuestion,
badAnswer,
displayErrMsg,
badAnswer,
displayErrMsg,
goodAnswer,
displayQuestion,
goodAnswer,
displayQuestion,
badAnswer,
displayErrMsg,
goodAnswer
>

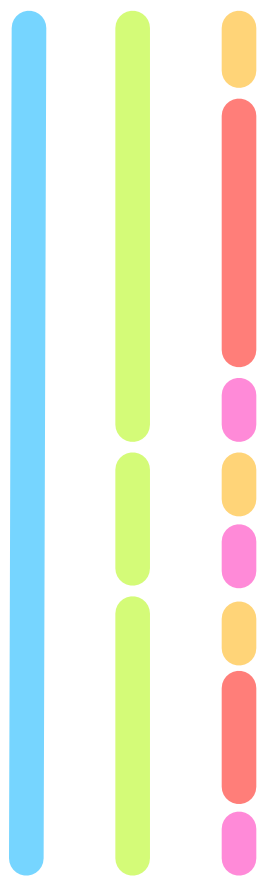


Take a step back and look at just the trace; this is where we find the structure. The states are just an artifact.

hierarchical structure

we can structure it like this:

<
displayQuestion,
badAnswer,
displayErrMsg,
badAnswer,
displayErrMsg,
goodAnswer,
displayQuestion,
goodAnswer,
displayQuestion,
badAnswer,
displayErrMsg,
goodAnswer
>



We can group the trace into subtraces, hierarchically. So we have a subtrace for each question, and that's divided into a subtrace for displaying the question, one for handling bad answers, and one for accepting the good answer.

as a grammar

grammar

- defines set of traces

  Session ::= Question*
  Question ::= **displayQuestion** BadAnswers **goodAnswer**
  BadAnswers ::= (**badAnswer displayErrMsg**)*

easily converted to code

- very roughly:

  void Session() {while (...) Question();}
  void Question() {**displayQuestion**(); BadAnswers(); **goodAnswer**}
  void BadAnswers() {while (...) {**badAnswer**; **displayErrMsg**}

This hierarchical structure can be expressed directly as a grammar. This is not only a more direct and clear representation of the structure than the state machine diagram, but it can be transcribed straightforwardly into structured code. We'll see how to do this in today's lecture.

A state machine diagram can be transcribed easily into code too, but in general it will be code with arbitrary gotos lacking structure, so that approach is generally only useful for autogenerated code, when the developer never has to look at the code but can do all development work at the diagram level.

regular grammars

This section of the lecture is a brief introduction to the idea of regular grammars and regular expressions, from a software engineering perspective. Some comments at the end of the lecture about the theoretical connection between grammars and state machines.

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

Grammars can be used for more than describing sequences of events, so we talk in general about sentences rather than traces.

example

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**}
URL = {

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 | C`

- option: an A is a B or is empty

  `A ::= B ?`

These are the regular expression operators. The symbol + is often used for an iteration of one or more; B+ is short for BB*.

examples

a two-way switch

SWITCH ::= (**up down**)*

a Java identifier

Identifier ::= Letter (Letter | Digit)*

Letter ::= **a** | **b** | ... | **Z**

Digit ::= **0** | **1** | ... | **9**

file handling protocol

FILE ::= **open** (**read** | **write**)* **close**?

trailing whitespace

TRAIL ::= (**space** | **tab**)* **newline**

SWITCH is an example of a grammar describing a trace set. IDENTIFIER and TRAIL illustrate the use of grammars for things that aren't about events. The FILE example illustrates a typical feature of APIs: that the methods must be invoked in some order -- in this case, that you can't read or write until the file has been opened. Closing the file is optional (for correctness, but if you open a lot of files and don't close them, you'll waste resources and damage performance).

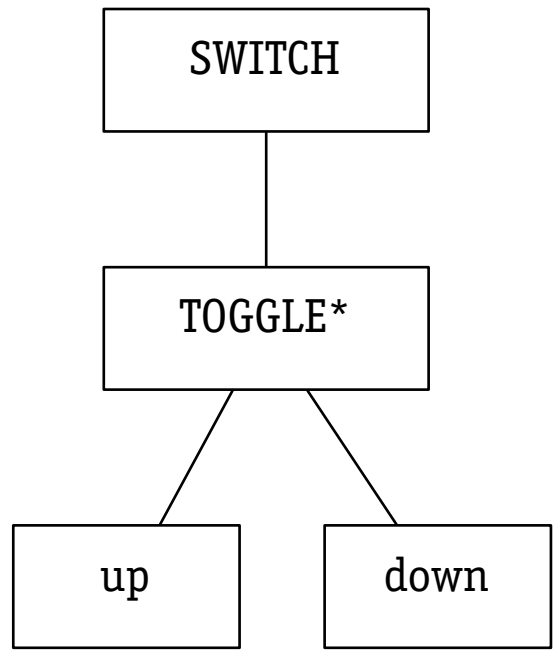
JSP form

"JSP form"

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

example

```
SWITCH ::= TOGGLE*          // SWITCH is an iteration
TOGGLE ::= up down          // TOGGLE is a sequence
```



The regular expression operators are technically all you need to describe a regular language: you just write a single expression for the language. That is, one production is enough. So why would you introduce more non-terminals and productions? We had an interesting discussion about this in class. The main reason is the same reason we use procedures in code: if the single expression would contain multiple occurrences of the same subexpression, we can introduce a non-terminal to factor that out, and thereby make it possible to change that subexpression in just one place. Someone else pointed out that sometimes a non-terminal represents a choice amongst a very large set of terminals, and in that case, it's easier just to leave the production for that non-terminal out and define it informally. In programming language grammars, for example, you'll find non-terminals for alphanumeric characters, and it's easier to define what that means in English than to actually list them all: ALPHANUMERIC ::= a | b | ...

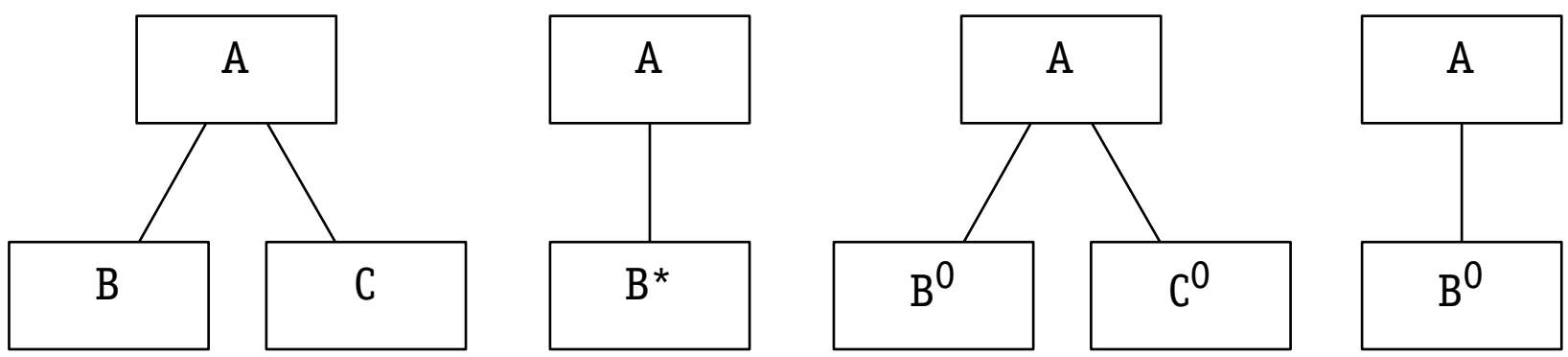
So, in general, when using grammars for modeling, you have to make good judgments about where to introduce non-terminals in just the same way that you decide when to introduce procedures in a program. But when grammars are being used in a more constructive way to generate code directly, it's appropriate to use a more prescriptive rule. The rule that JSP, the method I'm explaining today, uses, is that every production must be either a sequence, an iteration, a choice or an option. This leads to an easily understandable diagrammatic form in which each box in the diagram has a single type -- it's either one of these non-terminal types, or a terminal.

Tools that generate parsers from grammars also impose rules about the exact forms that productions must take.

diagram syntax for grammars

notation also called

- “structure diagram”
- “entity life history”



Here’s the repertoire of operators shown in diagrammatic JSP notation. Note that the same superscript 0 symbol is used for choice and option. In the choice, choosing the empty sequence is not one of the possibilities though. A common variant of iteration uses + in place of * to mean “one or more” rather than “zero or more”.

how to write code to read a stream

Now we'll see how to synthesize code directly from the grammar, first on the simpler example, with a bit of hand-waving and using pseudocode statements, then more carefully for the second example.

basic recipe

basic idea

- structure code around grammar of input events

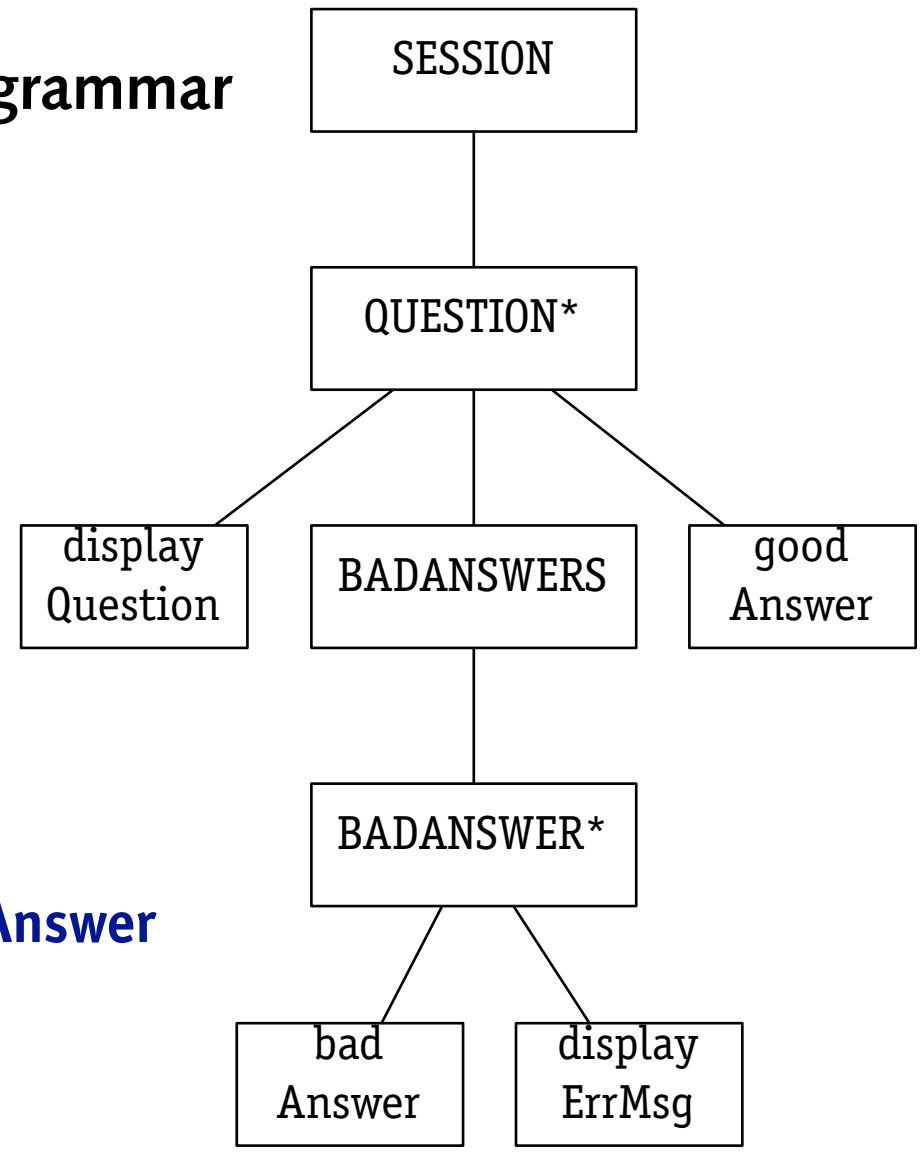
steps

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

This is the basic JSP recipe. The full method handles a much more general case, in which there are multiple streams being read and written, and you need to reconcile their structures. In many cases though -- like this one -- the output stream has a very simple relationship to the input stream and writes can just be embedded in the structure of the input stream.

example: quizzer

diagrammatic grammar

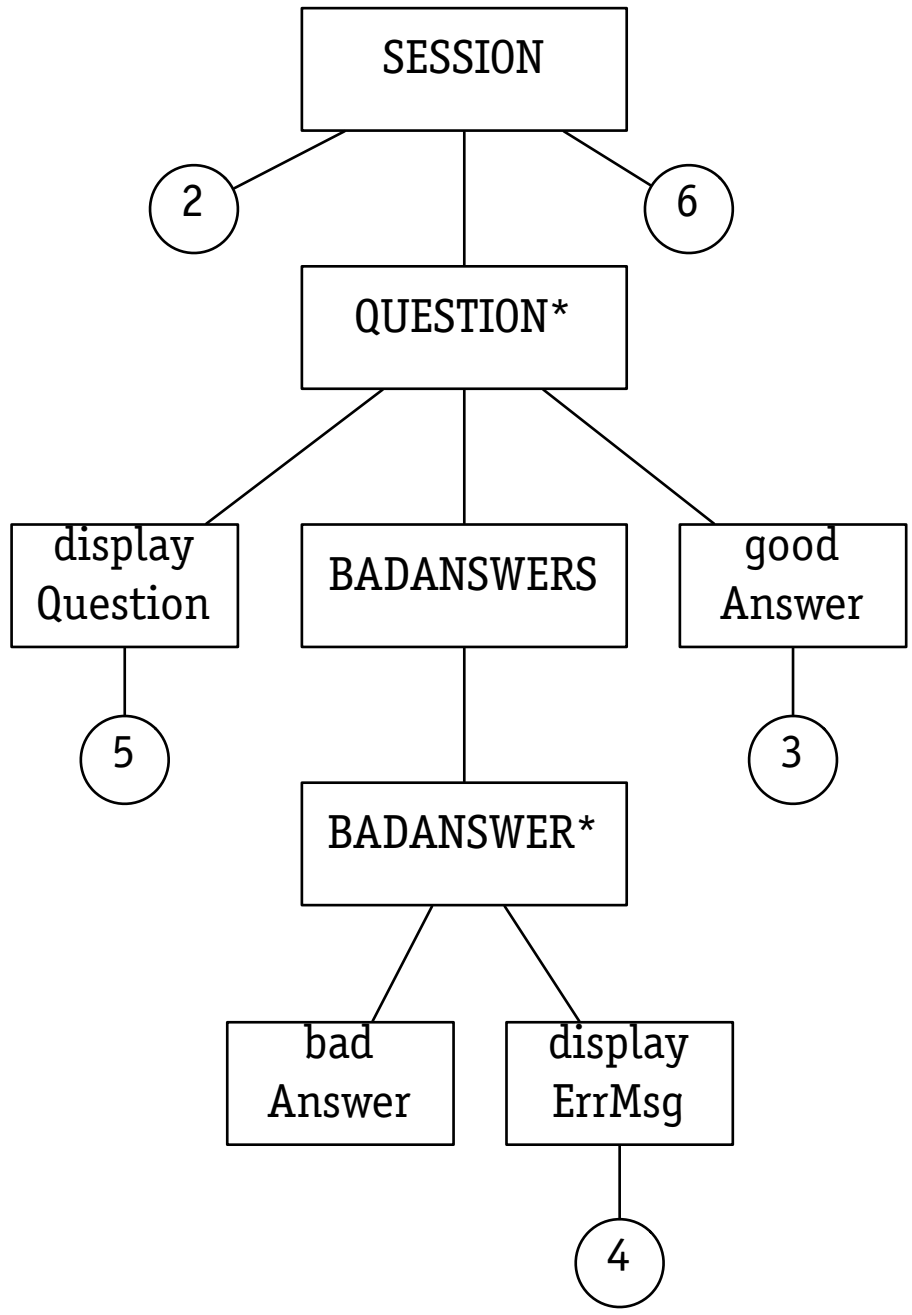


textual grammar

Session ::= Question*
Question ::= **displayQuestion** BadAnswers **goodAnswer**
BadAnswers ::= BadAnswer*
BadAnswer ::= **badAnswer displayErrMsg**

Here's the grammar for the input stream of the quizzer, shown in JSP diagram form, and as a textual grammar (for comparison -- you don't need that for the JSP method).

assigning operations



1 read line

2 initialize score

3 incr score

4 display error msg

5 display question

6 display score

Here's a list of basic operations: reading a line of input, initializing and incrementing the score, and the various display statements. To assign them to the diagram, you just ask the question "once per what?". For example, incrementing the score is done once per good answer, so we assign that operation, (3), to the goodAnswer component.

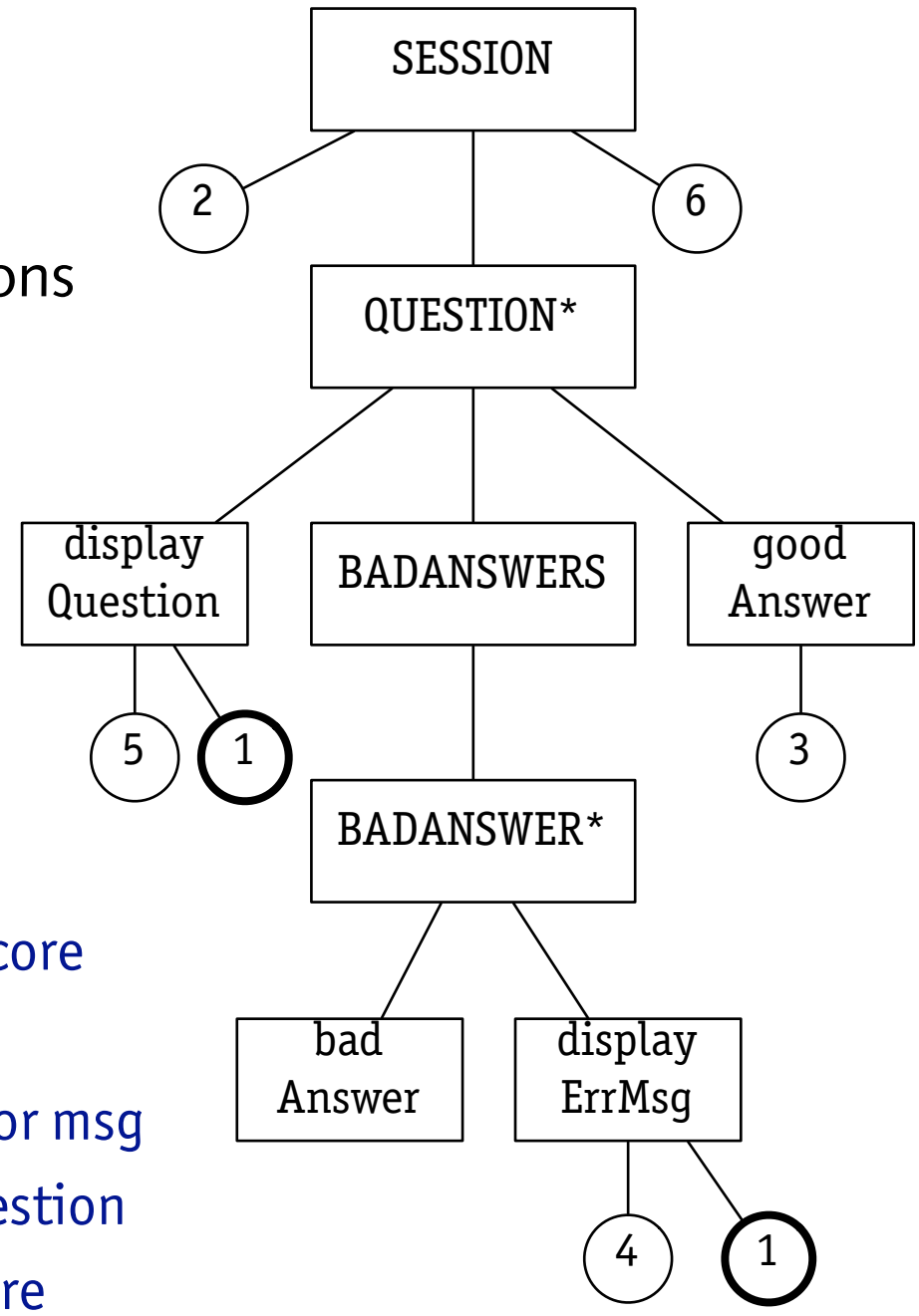
assigning reads

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

Reads are tricky, because you can't just assign them logically. You need to figure out how you're going to be able to evaluate the conditions (on loops and choices). In this case, for example, you need to determine whether there's another bad answer – and obviously you can't determine that until you've already read the answer to see if it's bad. Also you have to be careful when there are outputs also to order the inputs appropriately with respect to the outputs -- in this case, we have to do the reading for new input (1) after the displaying of an error message (4).

conditions

two conditions needed

- for termination of QUESTION*

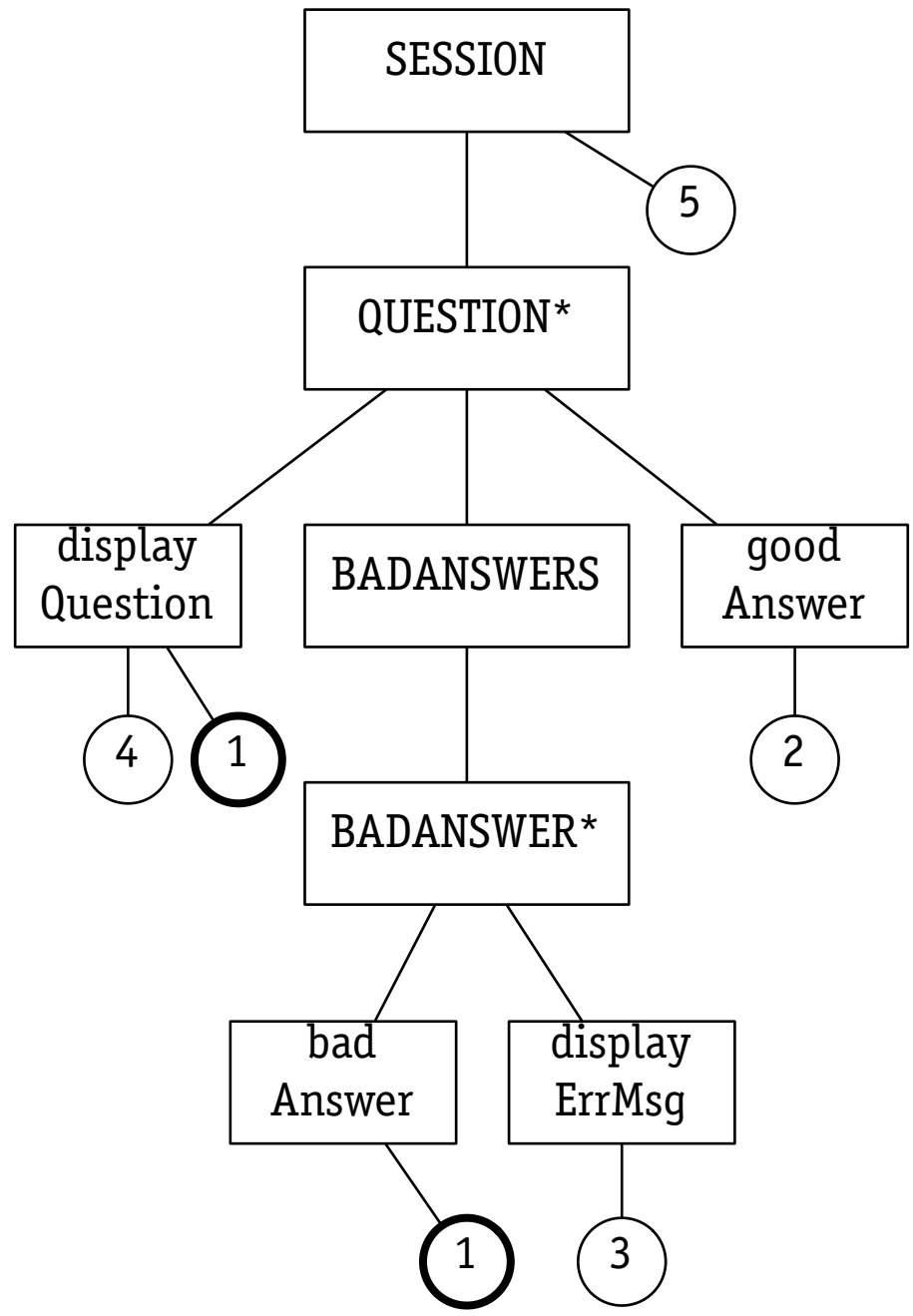
  while more questions in quiz

- for termination of BADANSWER*

now just transcribe into code

- follow the structure:

```
while (more questions) {
    // displayQuestion
    while (answer is bad)
        // badAnswer
        // displayErrMsg
    }
    // goodAnswer
}
```



Having assigned the operations, you write down the conditions for loops and choices -- in this case, just loops. Now we can just transcribe directly to code, with the structure following the structure of the diagram. Check that you understand how the code on the left is related to the diagram on the right, and see the next slide for the code fleshed out in full.

code

```
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

Now we consider the other stream processing problem: reading in the quiz template. This is more complicated in terms of the structure (but simpler in one respect -- there's are no output events to consider, so we don't need to worry about reading too early).

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?

The operation list that I showed earlier came out of thin air a bit. Here I'll explain more prescriptively how you construct it. A good place to start is to think about what abstract datatypes you need, and what their operations should. We'll be covering this in detail in the second third of the course.

The way to start thinking about the datatypes is to ask first what kind of objects we need to construct (questions, options, etc), and then what kinds of observations we'll need to make of them.

origins of operations

- for construction of objects
  - Quiz from Questions --> Quiz.new
  - Question from text and Options --> Question.new
  - Option from text and value --> Option.new
  - Scorer from range and message --> Scorer.new, Scorer.fromRange
- for observation of objects
  - from Quiz, get Questions --> Quiz.getQuestions
  - from Question, get text, num options --> Question.toString, getNumOptions
  - from Question, get score for choice --> Question.getScore(index)
  - from Quiz, get interpretation of score --> Quiz.interpretScore
- for internal observations
  - from Option, get text and value --> Option.getValue, toString
  - from Scorer, get interpretation of score --> Scorer.interpretScore

Here's how I thought about what methods I'd need: first the constructors, then the observers. The observers for Option and Scorer are internal in the sense that they're observations made with a Quiz or Question. For example, the program will call Quiz.interpretScore to interpret a score, and that will then call the method of the same name in Scorer. This design idiom is called "Facade" and limits dependences, by having access to types like Option and Scorer go through Quiz and Question. Take a look at slide 22 and you'll see that the code that executes the quiz only sees these two types, and is therefore decoupled from the other types.

datatypes designed

```
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) {
}
```


Here are the methods shown in Java. The Scorer.rangeElse method is the only one that's slightly tricky. I designed it like this because I knew I would want to construct a scorer incrementally, one rule at a time (where a rule consists of a range of values and a message to be displayed). So I chose to define a method that takes a scorer (the receiver argument), and range from lo to hi, and a message, and displays that message if the score is in that range, and otherwise uses the receiver scorer.

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!
```


Sometimes the grammar is given, but often you get to design it yourself. There are lots of considerations. Here the biggest question was whether to treat linebreaks as syntactically significant. By answering yes, I made the design of the program easier, and I made it easier to write the quiz template (because you don’t need to indicate which lines are question stems and which are options -- the options always follow line by line after the stem). This seemed reasonable to me because (a) most editors do soft line wrapping, so if you had a long question, you could edit in on multiple “lines” anyway, and (b) the displaying code can wrap the line if it’s too long.

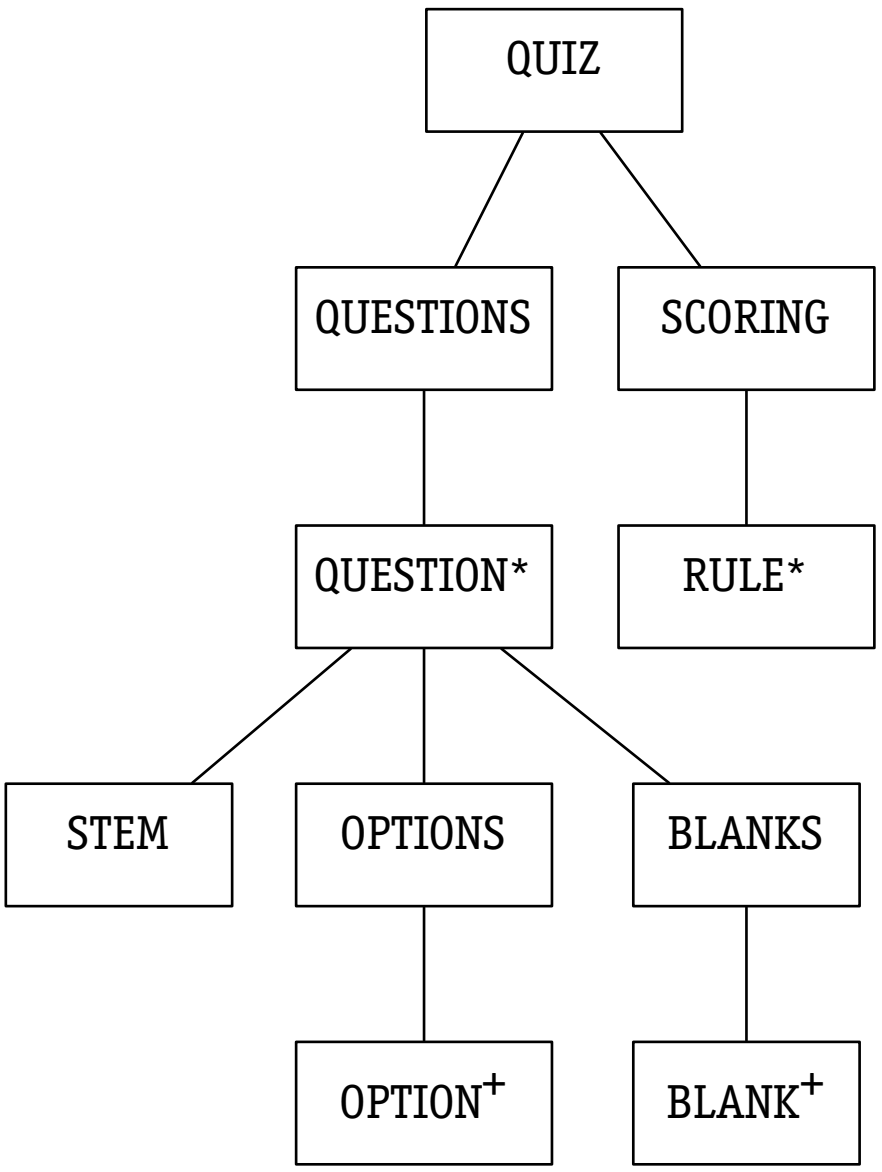
decisions

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

More discussion of various possibilities in the design of the grammar and what I decided to do.

the grammar



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

express line structure textually:

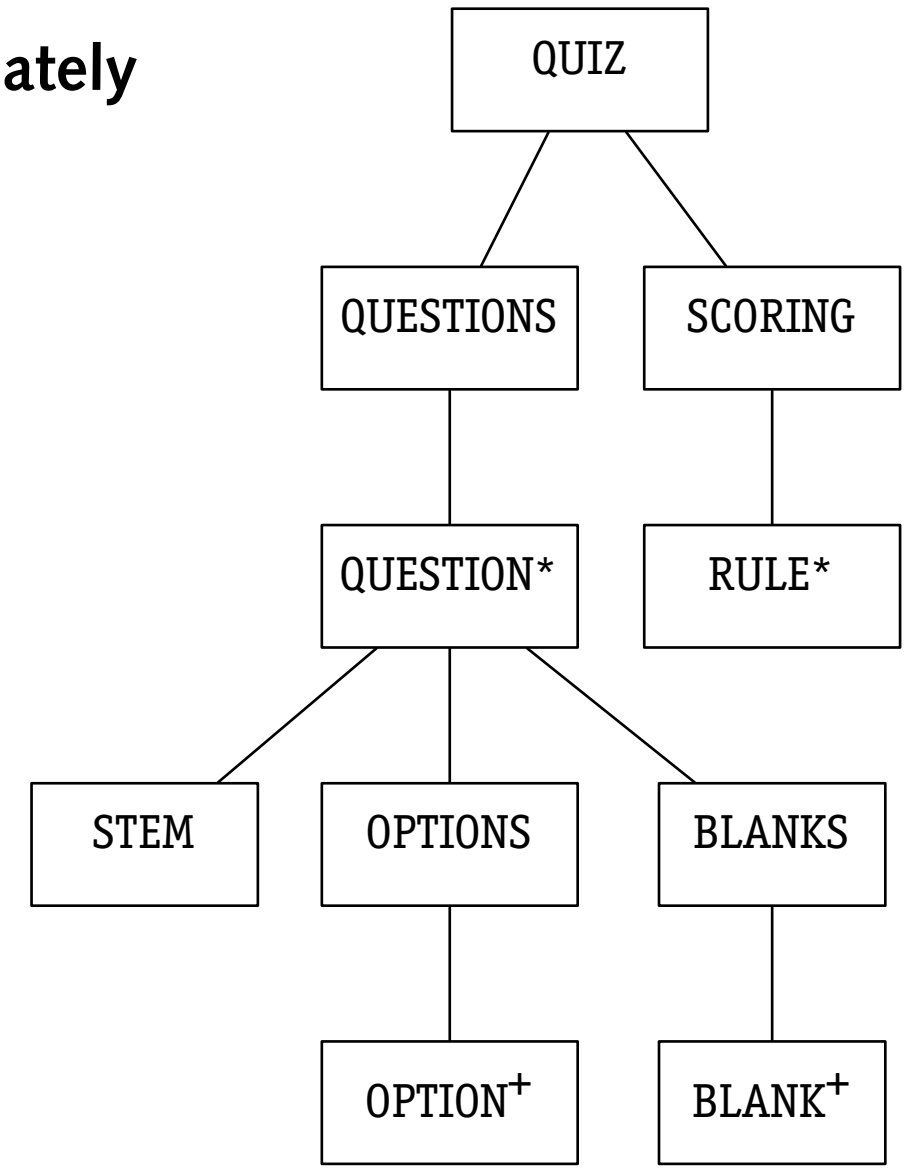
Option ::= Value? Text
Value ::= **[digit+]**
Text ::= **char***
Rule ::= Range Message
Range ::= **digit+ - digit+ :**
Message ::= **char***

Here's the JSP grammar for my design. I only took it down to lines, since I knew that for parsing a RULE or an OPTION I would use random access operations rather than using the JSP method character by character. For example, in RULE, I planned to call a method to find out where the hyphen and colon are, and then to use the String.substring method to pull out the hi and low bounds. I've written the structure of the parts not covered in the diagram in the textual grammar on the right just for completeness. If you expect people to satisfy some formatting rules, you have to give them precisely as a grammar.

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?

This shows the relationship between the control structure of the code that we're going to end up with and the structure of the stream, shown in the JSP diagram. To flesh out the code, we'll need to figure out what the operations are, and assign them to the appropriate components in the structure, and to determine the conditions for the loops.

listing operations

primary state variables

- list of questions so far
  List<Question> questions
- list of options so far
  List<Option> options
- scorer so far
  Scorer scorer
- next line to be processed
  String nextLine

how to enumerate ops

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

operations

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...

Here's how to go about listing the operations. First, list the basic variables you know you'll need. In this case, I know I'm going construct a quiz object from a list of questions, so I'll need to grow this list as I read the questions. Similarly, I'll need a list of options for each question. And as I mentioned before, I'm going to grow the scorer incrementally too, rule by rule. Finally, there's the line that's read from the stream.

Now for each of these variables, consider how it's initialized, how it's updated, and whether there's any final operation to be done on it. Take questions, for example: we need to initialize it to the empty list (2), and add a question to it (3). Some of these operations suggest new state variables and operations that will be needed. For example, when I add a question with (3), I need to form the question. So that suggests that I use the constructor (9), which then suggests I need to construct the stem (14) and the options (7). And so it goes on until it bottoms out. This might seem complicated, but it's pretty easy once you get the hang of it. The statements from 11 onwards all involve doing some parsing of nextLine using the substring and indexOf methods, the details of which I've omitted from the list. See the code for the details.

assigning operations

for each operation, ask

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

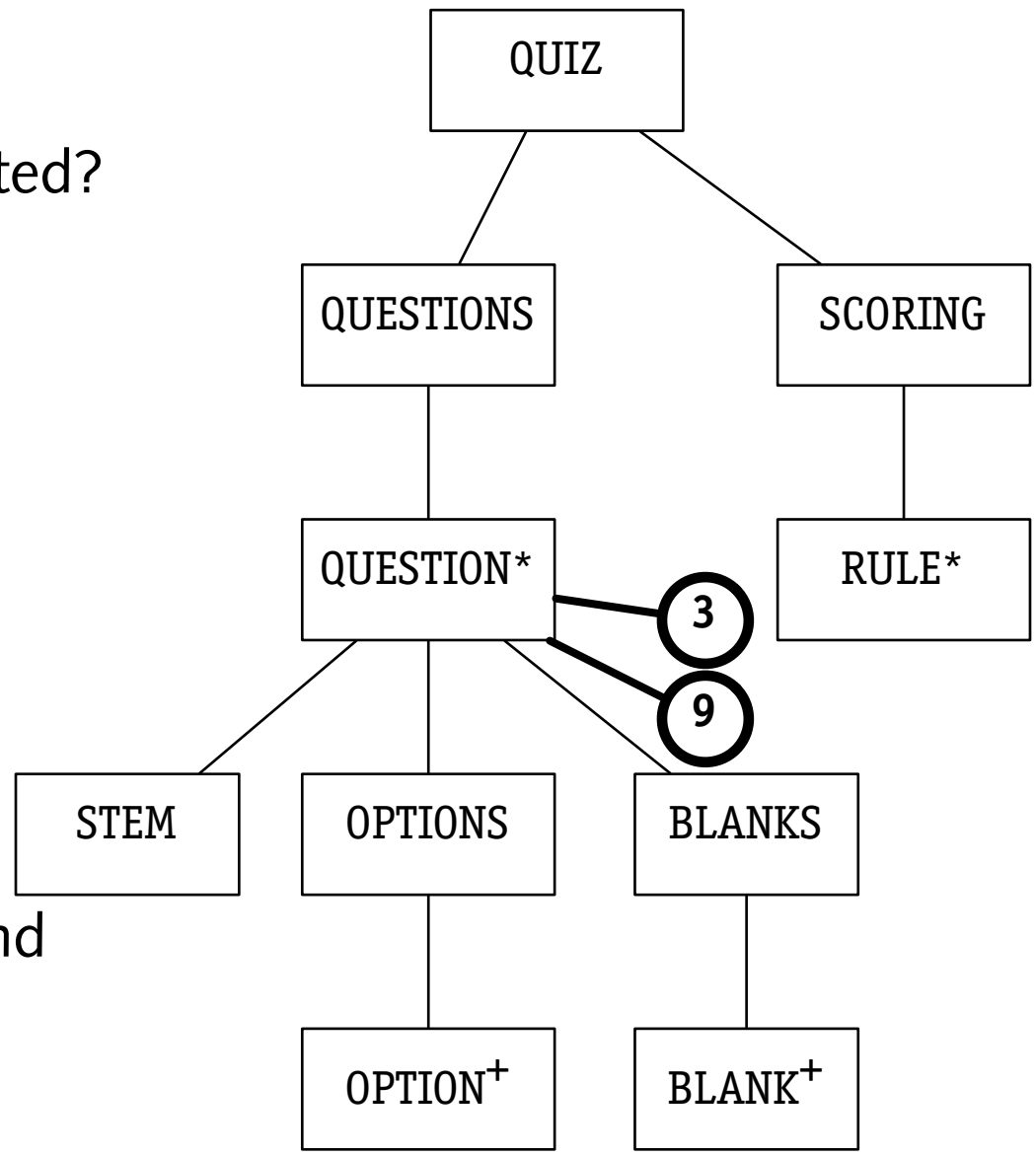
then

- 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)
```



Now we assign the operations, using the “once per what rule”, and considering any required dataflow order. So for example the adding of a question (9) and the creation of a new question (3) obviously happen once per question, at the end of that component, and the question has to be created before added, so (9) precedes (3).

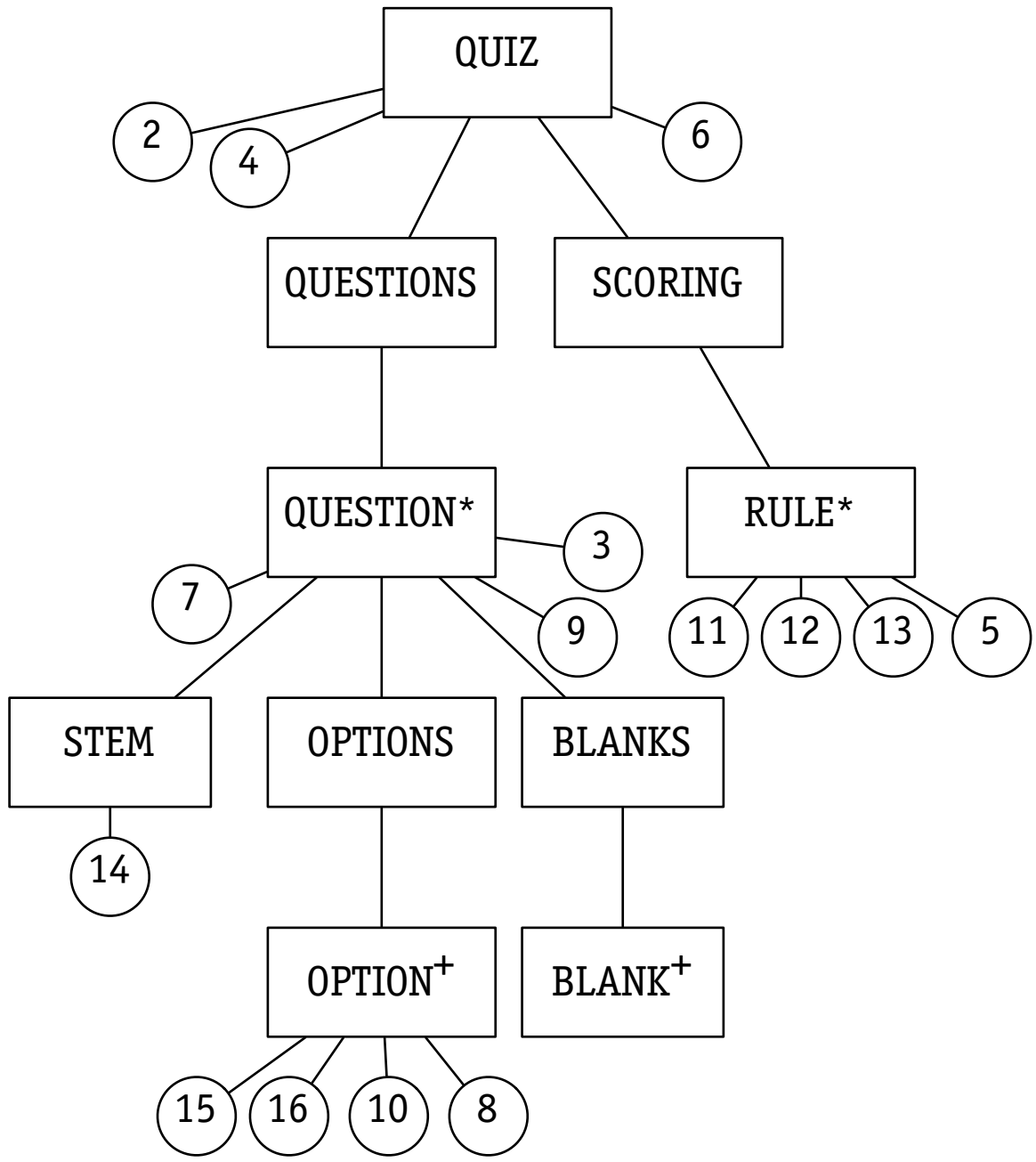
assigning operations, ctd

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
```


Here's what happens when I apply the "once per what" rule to each of the operations.

assignments, all but read

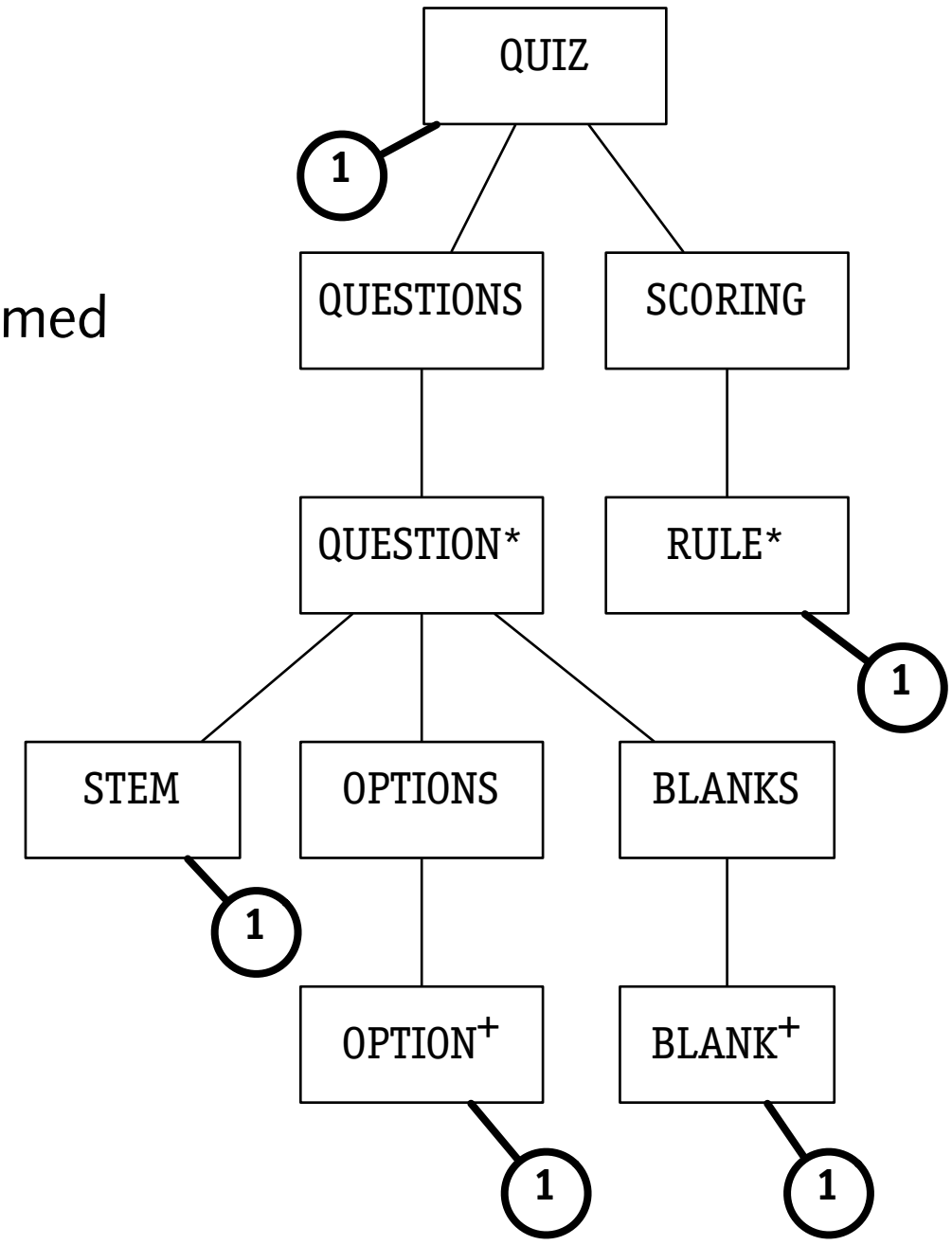


Now I assign them...

assigning the read operation

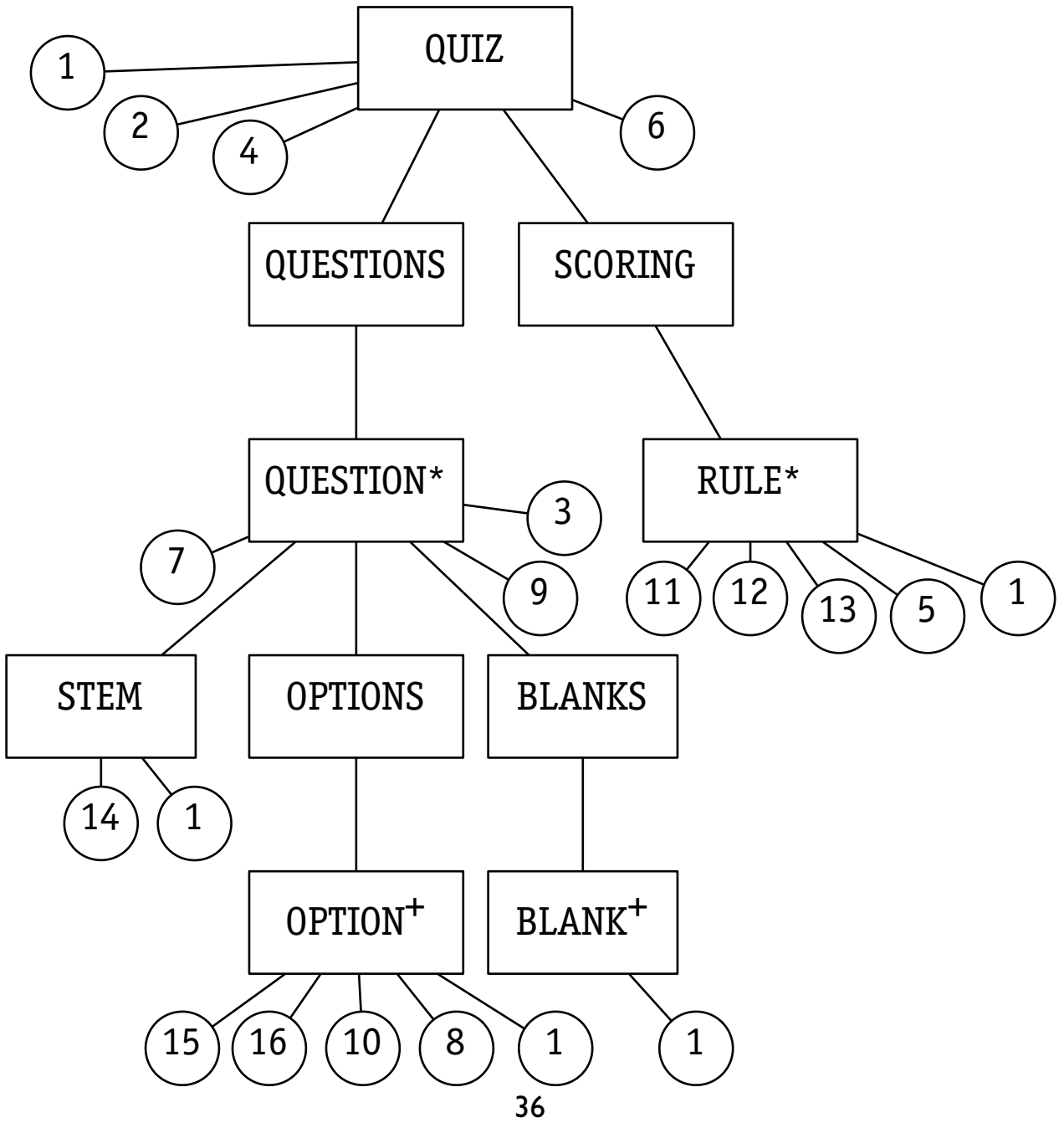
single readahead rule

- read once at the start
- read once after each record is consumed



The read operation is a bit trickier. Logically, you would just associated a read with each component that corresponds to a line: STEM, OPTION, BLANK and RULE. But this doesn't take account of the fact that the program will need to evaluate the looping conditions: we'll need to know if we have another question, for example. So in general the reads need to be done ahead of the conditions. This is called "readahead" or "lookahead". Usually looking ahead one record suffices. In fact, in this case, I designed the grammar to ensure that. So we just pull the reads back one step: we read right at the beginning, and then _after_ each record is consumed. See how I've placed the read ops -- the (1)'s -- in the diagram.

completed assignments



Here's the final structure with all the operations, read included, assigned.

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;
}
```


Now we write the conditions. Often it's convenient to introduce procedures to give cleaner structure to the conditions: here I've introduced ones for determining whether the next line is blank or represents end of file. note that the QUESTION* condition has to include the EOF check, because there may be no RULEs.

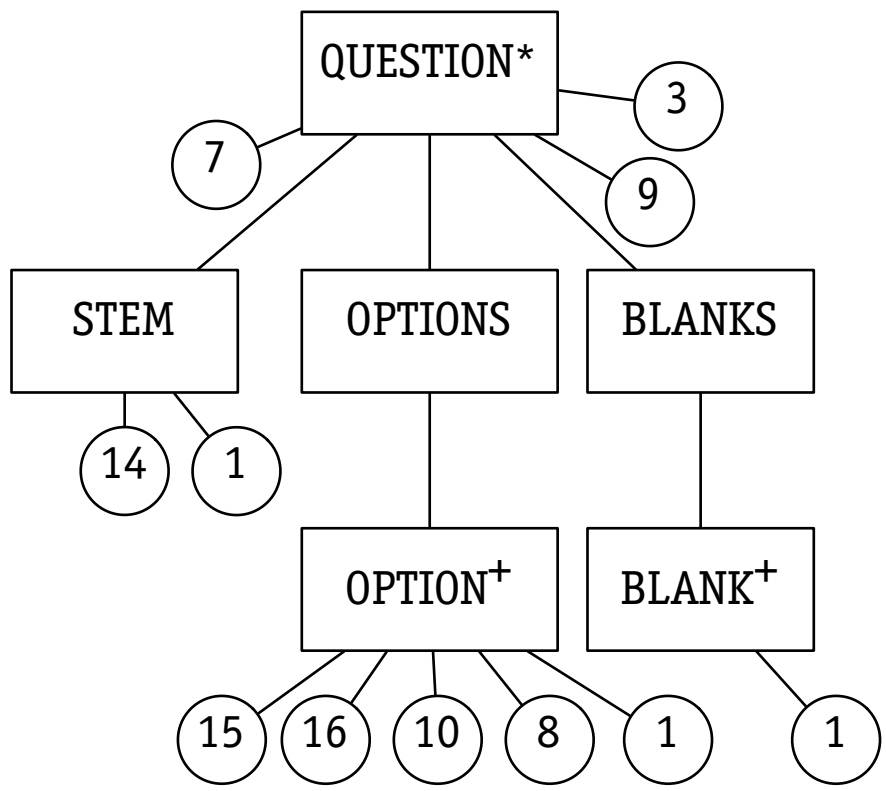
putting it all together

now just read code off the diagram!

- can introduce methods for boxes

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)
```



The final step is the easy one. We just populate the skeletal structure (see slide 30) from the diagram.

grammars, machines, regexps

A few general points and observations about the relationship between grammars and machines, and some comments about other uses of regular expressions.

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

```
SWITCH ::= (up down)*
```

can also write a grammar with only sequence and choice

- but allow special symbol for empty string **ε**
- and allow recursion on the left (or right) but not both

```
SWITCH ::= ε | (up down SWITCH)
```

- (in general multiple productions)

You can actually write a regular grammar without any of these regular expression operators, so long as ou have concatenation, and can show multiple options for a non-terminal -- here with the bar operator, but often done just by giving multiple productions for a single non-terminal. You need recursion to replace iteration, and you also need a special symbol for the empty string. If you can refer to a non-terminal recursively in any position on the RHS of a production, you don't have a regular grammar -- it becomes "context free". In a regular grammar, the recursive uses have to be on the extreme left or extreme right (and you can't mix both in a single grammar).

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 (**up down**)* include <**up**, **down**> but not <**up**>
- 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

Regular grammars are equivalent in expressive power to state machines, with one caveat. Our grammars allow us to describe the notion of a full trace. For example, the grammar for SWITCH ::= (up down)* defines the traces <>, <up, down>, <up, down, up, down>, but it does not include "incomplete" traces like <up, down, up>. Recall that our definition of the state machine notation was that every sequence of transitions that the machine allows is a trace, so the language of sentences it defines is "prefix closed", meaning that if t^e (t with e appended) is a trace, then so is just t. To make up for this difference, we can introduce "final" or "acceptor" states in the state machine, and say that an event sequence is only a trace if it starts at the initial state and ends at a final state. This is the definition of state machine usually used in theoretical CS, but it's not used in software engineering practice, because state machines are used to describe reactive processes that don't terminate, so the notion of a "complete trace" isn't needed.

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 non-terminals can be used recursively anywhere

extended Backus-Naur form (EBNF)

- the name for the (meta!) language of grammars in the form we've seen

The grammars you'll see in programming language definitions generally have full recursion -- an expression can contain arbitrary subexpressions, eg -- so they're context free and not regular, and could not be defined with state machines. A nice feature of the grammar/JSP approach is that it extends nicely to the context free case so long as the programming language you're transcribing into supports recursion (which Java does, of course).

This grammar notation, with productions and regular expression operators, is called EBNF.

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 C comment

  find pattern: `{([^}]*)}` [^}]* means an iteration of any char that is not }

  replace pattern: `/* \1 */` \1 binds to all chars that matched inside ()

Regular expressions are widely used for search and replace. The examples on this slide are from BBEdit, my favorite Mac text editor. Typically, regular expression packages have all kinds of features that actually take them well beyond regular languages.

summary

principles

for a structured stream

- use a grammar, not a state machine

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

Here are the principles from today. You should use the JSP strategy whenever you're processing a stream. It's simple and reliable, and you'll find that if you don't use it you can get into a real mess. See the article "Getting it Wrong" by Michael Jackson in the Cameron book (see next slide for cites).

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

on the JSP technique

- the original paper: http://mcs.open.ac.uk/mj665/ECI1976.pdf
- 'getting it wrong': http://mcs.open.ac.uk/mj665/GetWrong.pdf

some online materials

- http://en.wikipedia.org/wiki/Jackson_Structured_Programming
- http://www.jacksonworkbench.co.uk/stevefergspages/jackson_methods

Both of the JSP papers are short and easy to read. The Getting it Wrong paper is a nice illustration of the mess programmers often get into when they design a stream processing program without understanding how to structure the code around the structure of the stream.