Probabilistic Parsing and Semantic Attachments

J. David Beutel

ICS 661 Fall 2012 final project report

1. Introduction

This project was about enhancing syntactic parsing, by adding attachments for probability and semantics. It had two goals:

- add probabilistic parsing to the CKY parser from assignments 3 and 4, and
- add the generation of neo-Davidsonian meaning representations, via the evaluation of semantic attachments of First-Order Logic in Lambda notation.

Our textbook, Jurafsky and Martin[1], defines the terms of both goals, in section 14.2 and 18.2 respectively, so I will not repeat them here.

2. Description

I encountered some difficulties, but eventually accomplished both goals. I implemented everything in Groovy 2.0.5 (Laforge[2]), a dynamic extension of Java, using Spock 0.7 (Niederwieser[3]), a specification and test framework, and IntelliJ IDEA 11.1.4 Ultimate (Dmitriev[4]), an IDE, on Fedora 14 (Togami[5]), a Linux (Torvalds[6]) distribution. For the source code, see Appendix D., or https://github.com/jdbeutel/ics661-proj

It was simple to add annotations with probability to my existing CkyParser. But, the FOL in λ-notation was challenging. Because the parentheses level is critical to the λ-expressions, I realized that the FOL would need its own parser. The example grammar for semantic attachments (textbook figure 18.4) was nearly CNF, so I had hoped to use my CkyParser for that. But, I could not use CKY for the FOL, so I implemented an EarleyParser, and a FirstOrderLogic grammar with λ-expressions: a parser within a parser. Then I implemented a package of λ-terms to do the reductions, a builder of those terms from the FOL parse tree, and finally a SemanticGrammar to tie it all together and reach my second goal.

2.a. CkyParser

My Grammar had a list of Rules. I added an optional Attachment to each Rule, with a probability. The only notable part was, when Grammar creates new rules to convert to CNF, what probability to use? It uses 1 for convertTerminalsWithinRulesToDummyNonTerminals() and makeAllRulesBinary(), while convertUnitProductions() uses the product of the probability of the unit production being eliminated and the rule it goes to. This required a small change to assignment 4.

CkyParser differs from the textbook by not using the probability to optimize the parse. Instead, it generates all possible parses, and then sorts by probability. The probability of each CkyParse sub-tree is calculated recursively down both branches, the same way that the sub-tree is rendered to print. See Appendix A. for a sample run with data, which includes probabilities converted to CNF, and multiple parses sorted by probability.

This completed the first goal of the project.

2.b. EarleyParser

Embarking on the second goal of this project, I realized that I would need to parse the FOL attachments, and could not use the CkyParser for that. CkyParser converts the Grammar to CNF, but that would obscure important syntactic details in an FOL parse. For example, after eliminating unit productions, Variable and Constant terminals in the FOL grammar are both produced directly from Term, TermList, and TermOrFormula, hiding which they were in the original grammar. Also, I was not sure that I could convert the FOL attachments to CNF as simply as I had for the probabilities. So, I decided to implement EarleyParser.

EarleyParser uses the same Grammar, Rule, and Attachment as CkyParser. I followed the Earley algorithm from the textbook section 13.4.2. But, to support parsing of FOL symbols in non-terminal rules, as opposed to the lexicon that the textbook intended, I needed to add the advancing() method to the algorithm. The textbook notes that its scanner (and predictor) method differs slightly from the original Earley algorithm, as an optimization to avoid adding every terminal from a large lexicon such as "Verb" to the chart. That is good for grammars with lexicons, where terminals appear only in rules of terminal form. But, it does not work for FOL, or other programming languages, where terminal symbols appear in non-terminal rules. Rather than modify the FOL grammar by adding a terminal rule for every terminal, I modified the algorithm by adding the advancing() method. It works around this, while preserving that optimization, by advancing the state of a rule with a terminal matching the next actual input word, without adding a new state for every possible terminal.

To support the FOL symbols, I also needed to add a lexer Pattern to Parser. This was simple in Groovy. The default splits words by white-space, i.e., word boundaries. (See Illustration 1.)

```
abstract class Parser {

    List<String> words
    Grammar grammar

    Parser(String line, Grammar g, Pattern lexer = ~/\w+/) {
        grammar = g
        words = lexer.matcher(line).collect {it}
    }
```

Illustration 1: default lexer

Meanwhile, FirstOrderLogic splits on individual symbol characters, as well as white-space. (See Illustration 2.)

```
class FirstOrderLogic {

    static final SYMBOLIC_CHARS = '¬∧∨⇒∀∃().,'
    static final LAMBDA = 'λ'
    static final Pattern LEXER = ~("[${SYMBOLIC_CHARS}${LAMBDA}]|" + /\w+/)

    static EarleyParser parse(String input) {
        def result = new EarleyParser(input, GRAMMAR, LEXER)
```

Illustration 2: lexer with symbols

See Appendix B. for a sample run with the same input as CkyParser, for comparison. Note that the S-level probabilities are the same without CNF, suggesting that my transformation of probabilities for CNF was correct.

I used EarleyParser for SemanticGrammar as well as FOL, so they could both avoid CNF.

2.c. FirstOrderLogic

I did not try to evaluate FOL in terms of logical inference or productions, just λ-reductions. So, FirstOrderLogic is primarily a grammar using EarleyParser, and a method, buildLambda(), which takes that syntax tree and build a tree of terms from the lambda package. The method looks similar to a YACC program, or other such programming language parser. However, most of the words (symbols) are just passed along as payload, while the λ terms are wired up for reductions. Thus, the FOL part of the grammar goes largely unutilized.

I based the grammar of this class on the textbook figure 17.3. Unfortunately, the textbook does not provide an FOL grammar that includes λ-notation, it only explains how λ-reduction works. So, I added the λ productions to the grammar myself. One limitation of this grammar is ambiguity, as it has many loops from lower levels back to the top-level "Formula". Due to lack of time, to work around that ambiguity, I simply added a few parentheses as necessary to my FOL attachments. I would rather avoid that ambiguity by establishing precedence rules, but I am not sure how to do that with this grammar.

Note that the grammar defined here limits the FOL attachments in SemanticGrammar. Also, I could not include "S" as an "AbstractionVariable", because that non-terminal is already the grammar's start symbol. The grammar could be simplified, but I left it as is, due to a lack of time, and the possibility of adding logical inference or evaluation in the future.

See FirstOrderLogicSpec in Appendix D. for example parses and λ-reductions.

2.d. Lambda Package

The lambda package contains 7 classes and 1 interface; most extend the abstract SingleTerm. These could also be simplified, but I did not, due to lack of time. In particular, I distinguished between SingleTerm and TermList, which caused some difficulty as I had to convert back and forth, so I wonder if I could unify them. While an expression in lambda calculus is composed entirely of lambda terms, my TermList may include passive FOL words (symbols) that are just along for the ride, so to keep the scoping and grouping clear, I made the separate class for TermList. But, I wonder if I could treat all the FOL words as lambda free variables, and everything as single lambda terms. I suspect that would require a disambiguating grammar with precedence rules, however.

My original implementation of TermList was Lisp-ish, with a SingleTerm head and a recursive tail.

But, Groovy's support for lists, with literals, extra library methods, and closures, made the Groovy idioms easier to follow instead, so I switched to those.

As my understanding of the lambda calculus progressed, I got rid of the methods getBoundVariables() and alphaConversion(). In the end, alpha conversions are enacted by Application.reduction() via the freshen() and substitute() methods. The main implementation of freshen() is in Abstraction. I initially implemented alpha conversions backwards, converting the incoming term, and bound variables too. The children's game by Victor[9] helped me understand the lambda calculus groupings, but his "color rule" confused me about alpha conversions, as he seems to do it backwards. In the end, my alpha conversions are more like those described by Barendregt and Barendsen[8], and Wikipedia[7].

See LambdaSpec in Appendix D. for example construction and reduction of terms.

2.e. SemanticGrammar

The SemanticGrammar is mainly the example given by the textbook, but with probability in the attachment, and using YACC-style "$0" or "$1" instead of referencing the rule symbols to get the semantics of the child rule. That parsing is done by Attachment.parseLambda(), which handles "$n" and "$n($m)" itself, and delegates to FirstOrderLogic to parse anything else in that part of an attachment.

See Appendix C. for a sample run.

Analysis

My project achieved its goals. I feel like I understand λ-reductions (the term used by our textbook, or beta-reductions, the term used by Barendregt and Barendsen). FirstOrderLogic can parse and reduce arbitrary λ-expressions. SemanticGrammar reproduces the examples from our textbook.

However, this project does have limitations. The Groovy implementation of lambda is not as elegant as I had hoped. And, the semantic annotations seem to be limited and brittle. I did not have time to try expanding that grammar, but it is obvious that the semantic attachment for "opened" takes a direct object, while the one for "closed" does not. So, even that very limited grammar cannot always produce accurate semantic representations. Finally, I must admit that even if I have some understanding of λ-calculus now, it does not make me a functional programmer.

Conclusion

I learned that parsers and grammars need to vary by application. This project required several different kinds.

Also, regarding λ-calculus, I learned that free variables are more significant than bound ones. Bound variables are like local variables in Java; their scope limits their significance, so they can be safely renamed. Free variables, on the other hand, are like global variables. I suspect that free variables with the same name in different semantic attachments becomes the same variable. This might be a good thing to confirm with future work.

If I had had more time, I would have liked to make some additional or ambiguous semantic grammar, as well as logical inference or productions from FOL terms. And, I wonder if this project could have been implemented better in a functional programming language.

Finally, the textbook suggests some next steps for semantics. The most interesting to me is a unification-based approach, which seems like it might be less limited and brittle. (I am not sure if it

would involve the λ-calculus, though.) Two others are quantifier scope ambiguity and underspecification, and integration with the Earley parser (feedback between syntax and semantics).

Bibliography (and references)

1. Jurafsky, Daniel, and Martin, James H. Speech and Language Processing, 2nd ed. New Jersey: Pearson Education, Inc. 2009
2. Laforge, Guillaume, et al. Groovy, a dynamic language for the Java platform. http://groovy.codehaus.org/ Accessed 2012-12-10.
3. Niederwieser, Peter, et al. Spock, a testing and specification framework for Java and Groovy. http://code.google.com/p/spock/ Accessed 2012-12-10.
4. Dmitriev, Sergey, et al. IntelliJ IDEA. http://www.jetbrains.com/idea/ Accessed 2012-12-01.
5. Togami, Warren, et al. Fedora Project. https://fedoraproject.org/ Accessed 2012-12-10.
6. Torvalds, Linus, et al. Linux Kernel. http://www.kernel.org/ Accessed 2012-12-10.
7. http://en.wikipedia.org/wiki/Lambda_calculus Accessed 2012-12-10.
8. Barendregt, Henk, and Barendsen, Erik. Introduction to Lambda Calculus. 2000. ftp://ftp.cs.ru.nl/pub/CompMath.Found/lambda.pdf Retrieved 2012-11-30.
9. Victor, Bret. Alligator Eggs: A Puzzle Game Based on Lambda Calculus. http://worrydream.com/AlligatorEggs/ Retrieved 2012-12-10.

Appendix A. (CkyParser sample run)

```
[jdb@pikake:~/school/ics661/project/impl/src (master>)]$ cat ../small-L1.grammar
S -> VP                     [1]
VP -> Verb                  [.70]
VP -> Verb NP               [.20]
VP -> Verb NP PP            [.10]
NP -> Proper-Noun           [.50]
NP -> Det Nominal           [.30]
NP -> Nominal               [.20]
Nominal -> Noun             [.90]
Nominal -> Nominal PP       [.10]
PP -> Preposition NP        [1]
Det -> that                 [1]
Noun -> flight              [1]
Verb -> book                [1]
Proper-Noun -> Houston      [1]
Preposition -> through      [1]
[jdb@pikake:~/school/ics661/project/impl/src (master>)]$ groovy
parser/cky/CkyParser ../small-L1.grammar < ../small-probabilityInput


book that flight:
[S
    [Verb book {1}]
    [NP
        [Det that {1}]
        [Nominal flight {.90}]
     {.2700}]
 {.054000}]

book that flight through Houston:
[S
    [X1
        [Verb book {1}]
        [NP
            [Det that {1}]
            [Nominal flight {.90}]
         {.2700}]
     {.27000}]
```

```
    [PP
        [Preposition through {1}]
        [NP Houston {.50}]
     {.50}]
 {.013500000}]
;
[S
    [Verb book {1}]
    [NP
        [Det that {1}]
        [Nominal
            [Nominal flight {.90}]
            [PP
                [Preposition through {1}]
                [NP Houston {.50}]
             {.50}]
          {.045000}]
     {.01350000}]
 {.0027000000}]

flight book: not S
```

Appendix B. (EarleyParser sample run)

```
[jdb@pikake:~/school/ics661/project/impl/src (master>)]$ groovy
parser/earley/EarleyParser.groovy ../small-L1.grammar < ../small-probabilityInput


book that flight:
[S
    [VP
        [Verb book {1}]
        [NP
            [Det that {1}]
            [Nominal
                [Noun flight {1}]
             {.90}]
         {.2700}]
     {.054000}]
 {.054000}]

book that flight through Houston:
[S
    [VP
        [Verb book {1}]
        [NP
            [Det that {1}]
            [Nominal
                [Noun flight {1}]
             {.90}]
         {.2700}]
        [PP
            [Preposition through {1}]
            [NP
                [Proper-Noun Houston {1}]
             {.50}]
         {.50}]
     {.01350000}]
 {.01350000}]
;
[S
    [VP
```

```
        [Verb book {1}]
        [NP
            [Det that {1}]
            [Nominal
                [Nominal
                    [Noun flight {1}]
                 {.90}]
                [PP
                    [Preposition through {1}]
                    [NP
                        [Proper-Noun Houston {1}]
                     {.50}]
                 {.50}]
             {.045000}]
         {.01350000}]
     {.0027000000}]
 {.0027000000}]

flight book: not S
```

Appendix C.

```
[jdb@pikake:~/school/ics661/project/impl/src (master>)]$ groovy SemanticGrammar.groovy <
../semanticInput


Maharani closed:
[S
    [NP
        [ProperNoun Maharani {.2, [λM.(M(Maharani))]}]
     {.12, λM.(M(Maharani))}]
    [VP
        [Verb closed {.4, [λx.(∃e(Closed(e)∧ClosedThing(e,x)))]}]
     {.28, λx.(∃e(Closed(e)∧ClosedThing(e,x)))}]
 {.0336, λM.(M(Maharani))(λx.(∃e(Closed(e)∧ClosedThing(e,x))))}]
-
λM.(M(Maharani))(λx.(∃e(Closed(e)∧ClosedThing(e,x))))
λx.(∃e(Closed(e)∧ClosedThing(e,x)))(Maharani)
∃e(Closed(e)∧ClosedThing(e,Maharani))

every restaurant closed:
[S
    [NP
        [Det every {.2, [λP.(λQ.(∀x(P(x)⇒Q(x))))]}]
        [Nominal
            [Noun restaurant {1, [λr.(Restaurant(r))]}]
         {1, λr.(Restaurant(r))}]
     {.08, λP.(λQ.(∀x(P(x)⇒Q(x))))(λr.(Restaurant(r)))}]
    [VP
        [Verb closed {.4, [λx.(∃e(Closed(e)∧ClosedThing(e,x)))]}]
     {.28, λx.(∃e(Closed(e)∧ClosedThing(e,x)))}]
 {.0224, λP.(λQ.(∀x(P(x)⇒Q(x))))(λr.(Restaurant(r)))(λx.(∃e(Closed(e)∧ClosedThing(e,x))))}]
-
λP.(λQ.(∀x(P(x)⇒Q(x))))(λr.(Restaurant(r)))(λx.(∃e(Closed(e)∧ClosedThing(e,x))))
λQ.(∀x(λr.(Restaurant(r))(x)⇒Q(x)))(λx.(∃e(Closed(e)∧ClosedThing(e,x))))
∀x(λr.(Restaurant(r))(x)⇒λx.(∃e(Closed(e)∧ClosedThing(e,x)))(x))
∀x(Restaurant(x)⇒∃e(Closed(e)∧ClosedThing(e,x)))

Matthew opened a restaurant:
[S
    [NP
        [ProperNoun Matthew {.3, [λM.(M(Matthew))]}]
     {.18, λM.(M(Matthew))}]
    [VP
        [Verb opened {.6, [λW.(λz.(W(λx.(∃e(Opened(e)∧(Opener(e,z)∧Opened(e,x)))))))]}]
        [NP
            [Det a {.8, [λP.(λQ.(∃x(P(x)∧Q(x))))]}]
```

```
            [Nominal
                [Noun restaurant {1, [λr.(Restaurant(r))]}]
             {1, λr.(Restaurant(r))}]
         {.32, λP.(λQ.(∃x(P(x)∧Q(x))))(λr.(Restaurant(r)))}]
     {.0576, λW.(λz.(W(λx.(∃e(Opened(e)∧(Opener(e,z)∧Opened(e,x)))))))(λP.(λQ.(∃x(P(x)∧Q(x))))(λr.
(Restaurant(r))))}]
 {.010368, λM.(M(Matthew))(λW.(λz.(W(λx.(∃e(Opened(e)∧(Opener(e,z)∧Opened(e,x)))))))(λP.(λQ.
(∃x(P(x)∧Q(x))))(λr.(Restaurant(r)))))}]
-
λM.(M(Matthew))(λW.(λz.(W(λx.(∃e(Opened(e)∧(Opener(e,z)∧Opened(e,x)))))))(λP.(λQ.(∃x(P(x)∧Q(x))))(λr.
(Restaurant(r)))))
λW.(λz.(W(λx.(∃e(Opened(e)∧(Opener(e,z)∧Opened(e,x)))))))(λP.(λQ.(∃x(P(x)∧Q(x))))(λr.(Restaurant(r))))
(Matthew)
λz.(λP.(λQ.(∃x(P(x)∧Q(x))))(λr.(Restaurant(r)))(λy.(∃e(Opened(e)∧(Opener(e,z)∧Opened(e,y))))))(Matthew)
λP.(λQ.(∃x(P(x)∧Q(x))))(λr.(Restaurant(r)))(λy.(∃e(Opened(e)∧(Opener(e,Matthew)∧Opened(e,y)))))
λQ.(∃x(λr.(Restaurant(r))(x)∧Q(x)))(λy.(∃e(Opened(e)∧(Opener(e,Matthew)∧Opened(e,y)))))
∃x(λr.(Restaurant(r))(x)∧λy.(∃e(Opened(e)∧(Opener(e,Matthew)∧Opened(e,y))))(x))
∃x(Restaurant(x)∧∃e(Opened(e)∧(Opener(e,Matthew)∧Opened(e,x))))


Franco opened Frasca:
[S
    [NP
        [ProperNoun Franco {.25, [λF.(F(Franco))]}]
     {.150, λF.(F(Franco))}]
    [VP
        [Verb opened {.6, [λW.(λz.(W(λx.(∃e(Opened(e)∧(Opener(e,z)∧Opened(e,x)))))))]}]
        [NP
            [ProperNoun Frasca {.25, [λF.(F(Frasca))]}]
         {.150, λF.(F(Frasca))}]
     {.02700, λW.(λz.(W(λx.(∃e(Opened(e)∧(Opener(e,z)∧Opened(e,x)))))))(λF.(F(Frasca)))}]
 {.00405000, λF.(F(Franco))(λW.(λz.(W(λx.(∃e(Opened(e)∧(Opener(e,z)∧Opened(e,x)))))))(λF.
(F(Frasca))))}]
-
λF.(F(Franco))(λW.(λz.(W(λx.(∃e(Opened(e)∧(Opener(e,z)∧Opened(e,x)))))))(λF.(F(Frasca))))
λW.(λz.(W(λx.(∃e(Opened(e)∧(Opener(e,z)∧Opened(e,x)))))))(λF.(F(Frasca)))(Franco)
λz.(λF.(F(Frasca))(λx.(∃e(Opened(e)∧(Opener(e,z)∧Opened(e,x))))))(Franco)
λF.(F(Frasca))(λx.(∃e(Opened(e)∧(Opener(e,Franco)∧Opened(e,x)))))
λx.(∃e(Opened(e)∧(Opener(e,Franco)∧Opened(e,x))))(Frasca)
∃e(Opened(e)∧(Opener(e,Franco)∧Opened(e,Frasca)))


a closed every: not S


Franco: not S
```

Appendix D.

Grammar.groovy

```groovy
package grammar

/**
 * A context-free grammar with optional probability attachments.
 * This class is not thread-safe.
 */
class Grammar {

    final startSymbol = 'S'
    List<Rule> rules = []
    int dummyCounter = 1

    /**
     * Constructs a Grammar by parsing the given definition String.
     *
     * @param definition    lines defining the grammar, in a String suitable for
testing
     */
    Grammar(String definition) {
        for (line in definition.split('\n')) {
            addRule(line)
        }
        validateAttachments()
    }

    private void validateAttachments() {
        if (hasAttachments()) {
            for (s in nonTerminals) {
                def total = rulesFor(s).probability.sum()
                if (total != 1) {
                    throw new IllegalStateException("total probability of rules for $s
is $total instead of 1")
                }
            }
        }
    }

    boolean hasAttachments() {
        rules.find {it.attachment} as boolean
    }

    /**
     * Adds a Rule for the given line to this grammar.
     *
     * @param line  the definition of the Rule to add
     * @param i     optional index to insert in rules list, defaults to end
     */
    private void addRule(String line, int i = rules.size()) {
        def newRules = Rule.valuesOf(line, this)
        for (r in newRules) {
            if (rules.contains(r)) {
                throw new IllegalArgumentException("duplicate rule $r")
            }
            rules.add(i++, r)
        }
    }

    /**
     * Constructs a Grammar by parsing the given definition File.
     * This also validates that the given grammar contains a rule for the start symbol.
```

```
     * The String version of the constructor is less strict, for testing.
     *
     * @param definition    lines defining the grammar, in a File
     */
    Grammar(File definition) {
        this(canonicalEof(definition))
        if (!(startSymbol in nonTerminals)) {
            throw new IllegalArgumentException("$definition missing rule for start
symbol $startSymbol")
        }
    }

    /**
     * Reads the given file into a String and converts EOF to a canonical '\n'
regardless of OS.
     *
     * @param definition    the file to read
     * @return  a String of the text in the given file, with canonical EOF
     */
    private static String canonicalEof(File definition) {
        def s = ''
        definition.eachLine {
            s += it + '\n'
        }
        s
    }

    /**
     * @return the set of non-terminal symbols, a.k.a. N, derived from the rules
     */
    Set<String> getNonTerminals() {
        rules.nonTerminal as Set
    }

    /**
     * @return the set of terminal symbols, a.k.a. Sigma (disjoint from N), derived
from the rules
     */
    Set<String> getTerminals() {
        new HashSet(rules.symbols.flatten()) - nonTerminals
    }

    /**
     * Converts this grammar to CNF (Chomsky normal form).
     * The conversion is done in place, not copied to a new grammar.
     */
    void normalize() {
        convertTerminalsWithinRulesToDummyNonTerminals()
        convertUnitProductions()
        makeAllRulesBinary()
        removeRulesThatAreUnreachableFromStartSymbol()

        rules.each {assert it.normalForm}
        validateAttachments()
    }

    /**
     * Converts terminals within rules to dummy non-terminals.
     * Only terminals in rules with more than one symbol on the RHS are converted,
     * because a terminal in an RHS with only one symbol is already in normal form.
     * However, once a terminal is converted, all its occurrences are converted
     * to the same dummy; since this grammar is normalized in place,
     * this may generate unit productions, but they will be eliminated in the next
```

```
step.
     */
    private void convertTerminalsWithinRulesToDummyNonTerminals() {
        for (int i = 0; i < rules.size(); i++) {
            Rule r = rules[i]
            if (r.symbols.size() > 1) {
                String terminal
                while (terminal = r.symbols.find {it in terminals}) {
                    def dummy = nextDummySymbol
                    def attach = hasAttachments() ? '[1.0]' : ''
                    addRule("$dummy -> $terminal $attach", ++i)     // skip on next
loop
                    rules.findAll {it.symbols.size() > 1}.each
{it.changeSymbols(terminal, dummy)}
                }
            }
        }
    }

    /**
     * Eliminates unit productions, rules with an RHS of just one non-terminal.
     * That RHS is replaced with a copy of every rule with it on the LHS.
     * Each unit production adds an unnecessary step, because the grammar
     * matches the same language without them.
     */
    private void convertUnitProductions() {
        Rule r
        while (r = rules.find {it.unitProduction}) {
            String redundant = r.symbols[0]
            int i = rules.indexOf(r)
            rules.remove(i)
            for (q in rulesFor(redundant)) {
                def attach = ''
                if (hasAttachments()) {
                    def p = r.probability * q.probability
                    attach = "[$p]"
                }
                addRule("${r.nonTerminal} -> ${q.symbols.join(' ')} $attach", i++)
            }
        }
    }

    /**
     * Converts any rules with more than two symbols on the RHS into rules with just
two symbols.
     * This method follows the book's arbitrary method of converting the first two
symbols
     * into a dummy rule, and iterating on the same rule if it is still too long.
     * The new dummy is substituted for that pair in all rules where they are the first
symbols,
     * and the new dummy rule is inserted after the rule with the last substitution,
     * just for consistency with the example in the book.  That pair could be
substituted
     * wherever they appear in the symbols, perhaps, but I would rather keep this code
simple
     * than try to optimize it.
     */
    private void makeAllRulesBinary(){
        for (int i = 0; i < rules.size(); i++) {
            Rule r = rules[i]
            if (r.symbols.size() > 2) {
                def leadPair = r.symbols.subList(0, 2)
                def dummy = nextDummySymbol
```

```
                int lastIndex = i
                for (int j = i; j < rules.size(); j++) {
                    Rule q = rules[j]
                    if (q.symbols.size() > 2 && q.symbols.subList(0, 2) == leadPair) {
                        def theRest = q.symbols.subList(2, q.symbols.size())
                        q.symbols = [dummy] + theRest
                        lastIndex = j   // to put X2 after the last pair, like the book
does
                    }
                }
                def attach = hasAttachments() ? '[1.0]' : ''
                addRule("$dummy -> ${leadPair[0]} ${leadPair[1]} $attach", lastIndex+1)
                i-- // check current rule again
            }
        }
    }

    /**
     * Removes any rules that are not reachable from the start symbol.
     * For converting in place, this is necessary to clean up after
     * rules that have become redundant after eliminating unit productions.
     */
    private void removeRulesThatAreUnreachableFromStartSymbol(){
        def reached = [startSymbol] as Set
        int previous = 0
        while (reached.size() != previous) {
            previous = reached.size()
            def current = reached.collect {rulesFor(it).symbols}
            reached.addAll((List<String>) current.flatten())
        }
        def unreachable = nonTerminals - reached
        rules.removeAll {it.nonTerminal in unreachable}
    }

    /**
     * Finds all rules with the given non-terminal on their LHS.
     * This method scans all the rules, rather than maintaining
     * a map like this throughout the class, because I would rather
     * keep this code simple than try to optimize it.
     *
     *
     * @param nonTerminal   the LHS of the rules to find
     * @return  a list of rules with the given LHS
     */
    List<Rule> rulesFor(String nonTerminal) {
        rules.findAll {it.nonTerminal == nonTerminal}
    }

    /**
     * Finds all rules with an RHS of just the given terminal.
     * This method scans all the rules, rather than maintaining
     * a map like this throughout the class, because I would rather
     * keep this code simple than try to optimize it.
     *
     *
     * @param terminal   the whole RHS of the rules to find
     * @return  a list of rules with the given RHS
     */
    List<Rule> lexiconOf(String terminal) {
        rules.findAll {it.terminalForm && it.symbols[0] == terminal}
    }

    /**
```

```
     * Checks whether the given symbol is an LHS in the lexicon.
     * This method scans all the rules, rather than maintaining
     * a map like this throughout the class, because I would rather
     * keep this code simple than try to optimize it.
     *
     *
     * @param nonTerminal   the LHS of the rules to check
     * @return  whether the given symbol just the LHS of rules in terminal form
     */
    boolean isLexicon(String nonTerminal) {
        def matches = rulesFor(nonTerminal)
        matches && !matches.find {!it.terminalForm}
    }

    /**
     * Finds all binary rules with an RHS of the given pair of non-terminals.
     * This method scans all the rules, rather than maintaining
     * a map like this throughout the class, because I would rather
     * keep this code simple than try to optimize it.
     *
     *
     * @param b the first non-terminal on the RHS
     * @param c the second non-terminal on the RHS
     * @return  a list of binary rules with the given RHS
     */
    List<Rule> rulesTo(String b, String c) {
        rules.findAll {it.binaryForm && it.symbols[0] == b && it.symbols[1] == c}
    }

    /**
     * Gets the next available dummy symbol.
     * The symbol is an 'X' followed by an incrementing counter.
     * Symbols already in use are skipped.
     *
     * @return the next dummy symbol not already in use
     */
    private String getNextDummySymbol() {
        String s = null
        while (!s || s in nonTerminals || s in terminals) {
            s = 'X' + (dummyCounter++)
        }
        s
    }

    /**
     * @return the definition of this grammar in a format suitable for creation and
comparision
     */
    @Override
    String toString() {
        rules.join('\n')
    }
}
```

Rule.groovy

```groovy
package grammar

/**
 * A production in a context-free grammar.
 * This class is not thread-safe.
 */
class Rule {

    String nonTerminal
    List<String> symbols
    Attachment attachment
    Grammar grammar

    /**
     * Constructs one or more Rule from a line of a context-free grammar definition
with optional attachments.
     * The rules are not added to the Grammar; the caller needs to do that,
     * because it may have a specific place where it wants them to appear.
     *
     * @param line the single rule to parse, containing ' -> ' separator, and optional
' | ' separators and '[]' attachment
     * @param g the Grammar that will contain these Rules, determining whether or not a
given symbol is a terminal
     * @return the rules represented by the given line
     */
    static List<Rule> valuesOf(String line, Grammar g) {
        def parts = line.split(' -> ')
        if (parts.size() < 2) {
            throw new IllegalArgumentException("missing -> separator: $line")
        }
        if (parts.size() > 2) {
            throw new IllegalArgumentException("extra -> separators: $line")
        }
        def nonTerminal = parts[0].trim()
        def groups = parts[1].split(/ \| /)

        if (!nonTerminal) {
            throw new IllegalArgumentException("missing non-terminal to the left of ->
separator: $line")
        }
        def rules = []
        for (i in 0..groups.size()-1) {
            def group = groups[i].trim()
            def description = "| group $i to the right of -> separator: $line"
            def hasAttachment = group.contains('[')
            def attIdx = group.indexOf('[')
            def symbols = group.substring(0, hasAttachment ? attIdx :
group.length()).tokenize()
            if (!symbols) {
                throw new IllegalArgumentException("missing symbol(s) in $description")
            }
            def attachment = null
            if (hasAttachment) {
                if (!group.endsWith(']')) {
                    throw new IllegalArgumentException("missing attachment end ] in
$description")
                }
                try {
                    attachment = new Attachment(group.substring(attIdx+1,
group.length()-1))
```

```
                } catch (IllegalArgumentException e) {
                    throw new IllegalArgumentException("bad attachment in
$description", e)
                }
            }
            rules << new Rule(nonTerminal: nonTerminal, symbols: symbols, attachment:
attachment, grammar: g)
        }
        rules
    }

    /**
     * @return whether this rule is in normalized terminal form
     */
    boolean isTerminalForm() {
        symbols.size() == 1 && !unitProduction
    }

    /**
     * @return whether this rule is in normalized binary form
     */
    boolean isBinaryForm() {
        symbols.size() == 2 && symbols == nonTerminalSymbols
    }

    /**
     * @return  whether this Rule is valid for CNF
     */
    boolean isNormalForm() {
        (terminalForm || binaryForm)
    }

    /**
     * Checks whether this rule is a unit production.
     * A unit production is a rule with an RHS of just one non-terminal.
     *
     * @return whether this rule is a unit production
     */
    boolean isUnitProduction() {
        symbols.size() == 1 && symbols == nonTerminalSymbols
    }

    /**
     * @return List of symbols of this Rule that are non-terminals.
     * This does not include this Rule's nonTerminal property, which is obviously non-
terminal.
     */
    List<String> getNonTerminalSymbols() {
        symbols.findAll {it in grammar.nonTerminals}
    }

    /**
     * @return List of terminal symbols of this Rule.
     */
    List<String> getTerminals() {
        symbols - nonTerminalSymbols
    }

    /**
     * Replaces any and all occurrences of {@code from} with {@code to} on the right-
hand side of this rule.
     * The left-hand side of this rule is not changed.  The right-hand side remains the
same List instance,
```

```
     * but its contents may change.
     *
     * @param from  the symbol to change
     * @param to    the new symbol
     */
    void changeSymbols(String from, String to) {
        for (int i = 0; i < symbols.size(); i++) {
            if (symbols[i] == from) {
                symbols[i] = to
            }
        }
    }

    BigDecimal getProbability() {
        attachment?.probability
    }

    // for convenient duplicate check in List
    @Override
    boolean equals(Object other) {
        other instanceof Rule && nonTerminal == other.nonTerminal && symbols ==
other.symbols
    }

    // for consistency with equals()
    @Override
    int hashCode() {
        nonTerminal.hashCode() + symbols.hashCode()
    }

    /**
     * @return the definition of this rule in a format suitable for creation.
     */
    @Override
    String toString() {
        "$nonTerminal -> ${symbols.join(' ')}" + (attachment ? " [$attachment]" : '')
    }
}
```

Attachment.groovy

```groovy
package grammar

import fol.lambda.SingleTerm
import parser.earley.EarleyState
import fol.lambda.Application
import fol.FirstOrderLogic

/**
 * Data attached to a Rule.
 */
class Attachment {
    BigDecimal probability
    Closure lambdaClosure

    Attachment(String content) {
        def idx = content.indexOf(',')
        def p = (idx == -1 ? content : content.substring(0, idx))
        try {
            probability = new BigDecimal(p)
        } catch (NumberFormatException e) {
            throw new IllegalArgumentException("could not parse probability $p from
$content", e)
        }
        if (idx != -1) {    // semantic Lambda term
            parseLambda(content.substring(idx+1).trim())
        }
    }

    private void parseLambda(String s) {
        if (s ==~ /\$\d/) {                         // e.g., Nominal -> Noun  [1.0, $0]
            lambdaClosure = { EarleyState es ->
                es.components[s[1] as int].lambda
            }
        } else if (s ==~ /\$\d\(\$\d\)/) {          // e.g., S -> NP VP  [1.0, $0($1)]
            lambdaClosure = { EarleyState es ->
                SingleTerm x = es.components[s[1] as int].lambda
                SingleTerm y = es.components[s[4] as int].lambda
                new Application(abstractionTerm: x, term: y)
            }
        } else {                                    // e.g., Noun -> restaurant [1.0,
λr.Restaurant(r)]
            lambdaClosure = { FirstOrderLogic.parseLambda(s) }
        }
    }

    /**
     * @return a minimal String representation of the probability
     */
    static String canonicalProbability(p) {
        def s = p as String
        if (s.endsWith('.0')) {
            s = s[0..-3]
        }
        if (s.startsWith('0.') && s.length() > 2) {
            s = s[1..-1]
        }
        s
    }

    /**
```

```
     * @return the definition of this Attachment in a format suitable for creation.
     */
    @Override
    String toString() {
        canonicalProbability(probability)
    }
}
```

```groovy
package parser

import java.util.regex.Pattern
import grammar.Grammar

/**
 * Code common between the parser implementations.
 */
abstract class Parser {

    List<String> words
    Grammar grammar

    /**
     * Constructs a Parser, parsing the given line with the given Grammar.
     *
     * @param line the line of words to parse (i.e., a sentence)
     * @param g the grammar to use for the parse
     * @param lexer (optional) a regex identifying each separate word (i.e., token) in
the line
     */
    Parser(String line, Grammar g, Pattern lexer = ~/\w+/) {
        grammar = g
        words = lexer.matcher(line).collect {it}
    }

    /**
     * @return a list of roots of all accepted, full parse trees, or the empty list if
none are accepted
     */
    abstract List<Parse> getCompletedParses()

    /**
     * Renders all accepted, full parses as required for assignment 4.
     *
     * @return a rendering of all possible parses, or "not S" if none are accepted
     */
    String getCompletedParsesString() {
        completedParses?.join(';') ?: "not ${grammar.startSymbol}"
    }

    /**
     * Renders all accepted, full parses into easy-to-read indentation.
     *
     * @return a rendering of all possible parses, or "not S" if none are accepted
     */
    String getPrettyCompletedParsesString() {
        // using Parser.prettyPrint(), not the static prettyPrint() below.
        completedParses*.prettyPrint(0).join('\n;') ?: "not ${grammar.startSymbol}"
    }

    /**
     * Renders a completedParsesString in an easily readable and comparable format, for
testing.
     * The completedParses implementation's toString() needs to provide a compatible
input format.
     *
     * @param s a completedParsesString
     * @return the given parse formatted on multiple lines with indents
     */
```

```
    static String prettyPrint(String s) {
        def result = new StringBuilder()
        def level = -1
        while (s) {
            char c = s[0]
            s = s.substring(1)
            switch (c) {
                case '[':
                    level++
                    if (level) {
                        indent(result, level)
                    }
                    result << '['
                    break
                case ']':
                    level--
                    result << ']'
                    if (s.startsWith(' {') || s.startsWith(' (')) {
                        indent(result, level)
                    }
                    break
                case ';':
                    if (level == -1) {
                        result << '\n;\n'
                    } else {
                        result << ';'
                    }
                    break
                default:
                    result << c
                    break
            }
        }
        result
    }

    private static void indent(StringBuilder sb, level) {
        int lastCharIdx = sb.length() - 1
        if (sb.charAt(lastCharIdx) == ' ') {
            // delete trailing space, because IntelliJ automatically does the same
thing to the spec data
            sb.deleteCharAt(lastCharIdx)
        }
        sb << '\n' + ' ' * (level*4)
    }
}
```

Parse.groovy

```groovy
package parser

/**
 * Common interface for the results of a parse, representing the root of a subtree of
the parse tree.
 */
interface Parse {

    /**
     * Renders a pretty, multi-line parse subtree rooted at this node.
     *
     * @param level indentation levels to start from
     * @return a String having one line per node in this subtree
     */
    String prettyPrint(int level)
}
```

```groovy
package parser.cky

import java.util.regex.Pattern
import parser.Parser
import grammar.Grammar
import grammar.Rule

/**
 * Implementation of the CKY algorithm, a bottom-up parse of a CNF grammar, with
optional probability.
 * Instead of optimizing the parse by limiting it to the rule with the highest
probability
 * for any given non-terminal, this generates all possible parses (even with
probability),
 * and then sorts by probability (if any), for the sake of debugging.
 */
class CkyParser extends Parser {

    List<List<List<CkyParse>>> table = [].withDefault {[].withDefault {[]}}

    /**
     * Constructs a CkyParser, parsing the given line with the given Grammar.
     *
     * @param line the line of words to parse (i.e., a sentence)
     * @param g the grammar to use for the parse
     * @param lexer (optional) a regex identifying each separate word (i.e., token) in
the line
     */
    CkyParser(String line, Grammar g, Pattern lexer = ~/\w+/) {
        super(line, g, lexer)
        g.normalize()     // CKY requires CNF, so just in case g is not already
        parse()
    }

    /**
     * Does the CKY parse on this parser's line of words, according to its grammar.
     * This is the CKY algorithm from the textbook.
     */
    private void parse() {

        for (j in 1..words.size()) {    // for each word (or column from left)

            // The cell on the diagonal gets all terminal lexicon parses (A -> word)
for this word.
            String word = words[j-1]    // j-1 adjusts the algorithm to the 0-based
list
            List<Rule> lexicon = grammar.lexiconOf(word)
            List<CkyParse> lexiconParses = lexicon.collect {new CkyParse(it)}
            table[j-1][j].addAll(lexiconParses)     // diagonal (starting from row 0
and column 1)

            // Each cell in this column, from above the diagonal to row 0 (i.e.,
table[i][j]),
            // gets all possible combinations of binary parses (A -> B C) on the parses
so far.
            for (int i = j-2; i >= 0; i--) {    // using Java for() loop to skip j == 1
(the first column)
                for (k in i+1..j-1) {           // for each combination for [i,k] of
row i to [k,j] of column j
                    for (B in table[i][k]) {        // for each B = [i,k]
```

```
                        for (C in table[k][j]) {    // for each C = [k,j]
                            def matching = grammar.rulesTo(B.rule.nonTerminal,
C.rule.nonTerminal)  // all A -> B C
                            table[i][j].addAll(matching.collect {new CkyParse(it, B,
C)})
                        }
                    }
                }
            }
        }
    }

    /**
     * @return a list of roots of all accepted, full parse trees, or the empty list if
none are accepted
     */
    List<CkyParse> getCompletedParses() {
        def fullParses = table[0][words.size()]
// all A for [0,N]
        def sParses = fullParses.findAll {it.rule.nonTerminal == grammar.startSymbol}
// all S for [0,N]
        if (grammar.hasAttachments()) {
            sParses = sParses.sort {-it.probability}      // most-probable first
        }
        sParses
    }

    /**
     * Renders the whole parse table, for debugging.  (This is not required for
assignment 4.)
     *
     * @return a rendering of the parse table along the diagonals, from the terminals
to the apex
     */
    @Override
    String toString() {
        def s = ''
        for (j in 1..table.size()) {
            for (i in 0..table.size()-j) {
                def parses = table[i][i+j]
                if (grammar.hasAttachments()) {
                    parses = parses.sort {-it.probability}
                }
                s += "table[$i][${i+j}] = $parses\n"
            }
        }
        s
    }

    /**
     * For running from the command line (from the "src" dir),
     * this loads a grammar file and uses it to parse lines from stdin.
     *
     * @param args  command line arguments
     */
    static void main(String[] args) {

        if (args.size() != 1) {
            System.err.println "usage: groovy parser/cky/CkyParser grammarFile <
sentenceLines"
            System.exit 1
        }
```

```
        def g = new Grammar(new File(args[0]))
        System.in.eachLine {line ->
            println "\n$line: " + new CkyParser(line, g).prettyCompletedParsesString
            null    // just avoiding a warning about not returning a value from
eachLine
        }
    }
}
```

CkyParse.groovy

```groovy
package parser.cky

import grammar.Attachment
import grammar.Rule
import parser.Parse

/**
 * A node (i.e., subtree) in a CKY parse tree.
 */
class CkyParse implements Parse {
    Rule rule   // A -> B C, or A -> terminal
    CkyParse B, C

    /**
     * Constructor for a terminal rule.
     *
     * @param r a Rule in terminal form
     */
    CkyParse(Rule r) {
        assert r.terminalForm
        rule = r
    }

    /**
     * Constructor for a binary rule.
     *
     * @param r a Rule in binary form
     * @param b the left subtree node
     * @param c the right subtree node
     */
    CkyParse(Rule r, CkyParse b, CkyParse c) {
        assert r.binaryForm
        rule = r
        B = b
        C = c
    }

    BigDecimal getProbability() {
        def p = rule.probability
        p == null || rule.terminalForm ? p : p * B.probability * C.probability
    }

    String prettyPrint(int level = 0) {
        def indentBy = ' ' * 4
        def indent = '\n' + (indentBy * level)
        def attach = ''
        if (rule.attachment) {
            attach = " {${Attachment.canonicalProbability(probability)}}"
        }
        if (rule.terminalForm) {
            assert !B && !C
            "$indent[${rule.nonTerminal} ${rule.symbols[0]}$attach]"
        } else {
            "$indent[${rule.nonTerminal}${B.prettyPrint(level+1)}$
{C.prettyPrint(level+1)}$indent$attach]"
        }
    }

    /**
     * @return render of the subtree rooted at this node (recursively) in bracket
```

```
format
     */
    @Override
    String toString() {
        def attach = ''
        if (rule.attachment) {
            attach = " {${Attachment.canonicalProbability(probability)}}"
        }
        if (rule.terminalForm) {
            assert !B && !C
            "[${rule.nonTerminal} ${rule.symbols[0]}$attach]"
        } else {
            "[${rule.nonTerminal} $B $C$attach]"
        }
    }
}
```

```groovy
package parser.earley

import java.util.regex.Pattern
import parser.Parser
import grammar.Grammar
import grammar.Rule

/**
 * Implementation of the Earley parser, from the textbook.
 */
class EarleyParser extends Parser {

    List<List<EarleyState>> chart = [].withDefault {[]}

    /**
     * Constructs an EarleyParser, parsing the given line with the given Grammar.
     *
     * @param line the line of words to parse (e.g., a sentence)
     * @param g the grammar to use for the parse
     * @param lexer (optional) a regex identifying each separate word (i.e., token) in
the line
     */
    EarleyParser(String line, Grammar g, Pattern lexer = ~/\w+/) {
        super(line, g, lexer)
        parse()
    }

    /**
     * Parses this parser's line of words, according to its grammar.
     * This is the Earley algorithm from the textbook, which has an optimization
     * of the original algorithm to avoid adding a lexicon's whole set of terminals
     * to the chart.  But, I enhanced that to support FOL, to also allow terminals
     * in non-terminal rules (i.e., non-lexicons), with the advancing() method.
     */
    private void parse() {
        enqueue(dummyStartState, chart[0])
        for (i in 0..words.size()) {
            int j = 0
            while (j < chart[i].size()) {   // allowing chart[i] to grow as j goes
through it
                def state = chart[i][j++]
                if (state.complete) {
                    completer(state)
                } else {
                    if (grammar.isLexicon(state.b)) {
                        scanner(state)
                    } else if (state.b in grammar.terminals) {
                        advancing(state)
                    } else {
                        predictor(state)
                    }
                }
            }
        }
    }

    /**
     * This is used to start the parse chart, but is not advanced to completion.
     *
     * @return the dummy start state
```

```
     */
    private getDummyStartState() {
        Rule dummy = Rule.valuesOf("γ -> ${grammar.startSymbol}", grammar)[0]
        new EarleyState(dummy, 0, [0, 0], 'dummy start state')
    }

    /**
     * Enqueue a new state for every rule that, if completed, would advance the dot in
the given state.
     *
     * @param state to look forward (i.e., top-down) on how to advance
     */
    private void predictor(EarleyState state) {
        def B = state.b
        def j = state.inputDotIdx
        for (rule in grammar.rulesFor(B)) {
            enqueue(new EarleyState(rule, 0, [j, j], 'predictor'), chart[j])
        }
    }

    /**
     * Works backwards from the next input word,
     * to enqueue a state for every matching terminal rule.
     * This is an optimization by the textbook, limiting it to the actual input word,
     * to avoid enqueueing a state for every terminal in the lexicon.
     *
     * @param state with the dot on a terminal rule
     */
    private void scanner(EarleyState state) {
        def B = state.b
        def j = state.inputDotIdx
        def word = words[j]
        def rule = grammar.lexiconOf(word).find {it.nonTerminal == B}
        if (rule) {
            assert rule.nonTerminal == B && rule.symbols == [word]
            enqueue(new EarleyState(rule, 1, [j, j+1], 'scanner'), chart[j+1])
        }
    }

    /**
     * Fixes the textbook's optimization to support grammars with terminals in non-
terminal rules, like FOL.
     * This advances the state over the next input word, if the state's next symbol
matches it.
     *
     * @param state with the dot on a terminal symbol
     */
    private void advancing(EarleyState state) {
        def j = state.inputDotIdx
        if (words[j] == state.b) {
            enqueue(state.advancing(), chart[j+1])
        }
    }

    /**
     * Advances states satisfied by the completion of the given state.
     *
     * @param state completed state
     */
    private void completer(EarleyState state) {
        def B = state.rule.nonTerminal
        def j = state.inputStartIdx
        def k = state.inputDotIdx
```

```
        def incompleteStates = chart[j].findAll {!it.complete && it != dummyStartState}
        for (EarleyState match in incompleteStates.findAll {it.b == B && it.inputDotIdx
== j}) {
            enqueue(match.completer(state), chart[k])
        }
    }

    private enqueueSequence = 0

    /**
     * Adds a state to the given chartEntry, if it is not already in it, assigning the
state a sequential name.
     *
     * @param state to add to chartEntry
     * @param chartEntry in which to enqueue the state
     */
    private void enqueue(EarleyState state, List<EarleyState> chartEntry) {
        if (!(state in chartEntry)) {
            state.name = "S${enqueueSequence++}"
            chartEntry << state
        }
    }

    /**
     *
     * @return a list of roots of all accepted, full parse trees, or the empty list if
none are accepted
     */
    @Override
    List<EarleyState> getCompletedParses() {
        def N = words.size()
        def fullParses = chart[N].findAll {it.complete && it.inputStartIdx == 0 &&
it.inputDotIdx == N} // [0,N]
        def sParses = fullParses.findAll {it.rule.nonTerminal == grammar.startSymbol}
// all S for [0,N]
        if (grammar.hasAttachments()) {
            sParses = sParses.sort {-it.probability}      // most-probable first
        }
        sParses
    }

    /**
     * Renders the whole parse chart, for debugging.
     *
     * @return a flat rendering of the parse chart (i.e., not nested or treed)
     */
    @Override
    String toString() {
        (0..<chart.size()).collect { i ->
            "Chart[$i]\t" + chart[i]*.toFlatString().join('\n\t')
        }.join('\n\n')
    }

    /**
     * For running from the command line (from the "src" dir),
     * this loads a grammar file and uses it to parse lines from stdin.
     *
     * @param args  command line arguments
     */
    static void main(String[] args) {

        if (args.size() != 1) {
            System.err.println "usage: groovy parser/earley/EarleyParser grammarFile <
```

```
sentenceLines"
            System.exit 1
        }

        def g = new Grammar(new File(args[0]))
        System.in.eachLine {line ->
            println "\n$line: " + new EarleyParser(line, g).prettyCompletedParsesString
            null    // just avoiding a warning about not returning a value from
eachLine
        }
    }
}
```

EarleyState.groovy

```groovy
package parser.earley

import grammar.Attachment
import grammar.Rule
import groovy.transform.EqualsAndHashCode
import fol.lambda.SingleTerm
import fol.lambda.TermList
import parser.Parse

/**
 * Chart entries in an Earley parse.
 * These are also nodes (i.e., sub-trees) in the parse tree.
 * As such, completed components are part of the identity; this differs from the
textbook algorithm,
 * which could map different completed components to the same state (as an
optimization).
 */
@EqualsAndHashCode(excludes = ['function', 'name'])
class EarleyState implements Parse {

    Rule rule
    int dotIdx
    int inputStartIdx, inputDotIdx
    List<EarleyState> components = []
    String function     // not part of the identity
    String name         // not part of the identity

    EarleyState(Rule rule, int dotIdx, List inputIdxPair, String function) {
        this.rule = rule
        this.dotIdx = dotIdx
        (inputStartIdx, inputDotIdx) = inputIdxPair
        this.function = function
    }

    boolean isComplete() {
        dotIdx == rule.symbols.size()
    }

    String getB() {
        assert !complete
        rule.symbols[dotIdx]
    }

    /**
     * Factory method for a new EarleyState with the dot advanced over one completed
component.
     *
     * @param component completed for advance
     * @return new EarleyState
     */
    EarleyState completer(EarleyState component) {
        assert !complete && component.complete
        def result = new EarleyState(rule, dotIdx+1, [inputStartIdx,
component.inputDotIdx], 'completer')
        if (dotIdx > 0) {
            result.components += components[0..<dotIdx]
        }
        result.components[dotIdx] = component
        result
    }
```

```
    /**
     * Factory method for a new EarleyState with the dot advanced over one terminal
symbol.
     * This is an optimization of the textbook's optimization of scanner(),
     * to support terminal symbols in non-terminal-form rules (i.e., in lexicon, or
part of speech).
     *
     * @return new EarleyState
     */
    EarleyState advancing() {
        assert !complete && b in rule.terminals
        def result = new EarleyState(rule, dotIdx+1, [inputStartIdx, inputDotIdx+1],
'advancing')
        if (dotIdx > 0) {
            result.components += components[0..<dotIdx]
        }
        result.components[dotIdx] = null    // just the terminal symbol here
        result
    }

    /**
     * @return render of the subtree rooted at this node (recursively) in bracket
format
     */
    @Override
    String toString() {
        def subtrees = []
        for (i in 0..<rule.symbols.size()) {
            def s = rule.symbols[i]
            subtrees << (i < dotIdx && s in rule.nonTerminalSymbols ? components[i] :
s)
        }
        "[$name ${rule.nonTerminal} ${subtrees.join(' ')} ($inputStartIdx,$inputDotIdx)
$attachmentStr]"
    }

    private String getAttachmentStr() {
        if (rule.attachment) {
            def p = Attachment.canonicalProbability(probability)    // of current
subtree, not the rule.attachment's
            Closure cl = rule.attachment.lambdaClosure
            return cl ? " {$p, ${cl(this)}}" : " {$p}"
        } else {
            return ''
        }
    }

    String toFlatString() {
        [name, ruleWithDot, [inputStartIdx, inputDotIdx], function].join('\t')
    }

    String prettyPrint(int level = 0) {
        def indentBy = ' ' * 4
        def indent = '\n' + (indentBy * level)
        if (components.find {it}) {     // has nested states
            def subtrees = []
            for (i in 0..<rule.symbols.size()) {
                def c = components[i]
                subtrees << (c ? c.prettyPrint(level+1) : indent + indentBy +
rule.symbols[i])
            }
            return "$indent[${rule.nonTerminal}${subtrees.join('')}
```

```
$indent$attachmentStr]"
        } else {
            return "$indent[${rule.nonTerminal} ${rule.symbols.join(' ')}
$attachmentStr]"
        }
    }

    private String getRuleWithDot() {
        def withDot = [] + rule.symbols
        withDot.addAll(dotIdx, '•')     // insert
        "${rule.nonTerminal} -> ${withDot.join(' ')}"
    }

    BigDecimal getProbability() {
        def p = rule.probability
        if (p != null) {
            for (int i = 0; i < dotIdx; i++) {
                if (rule.symbols[i] in rule.nonTerminalSymbols) {
                    p *= components[i].probability
                } else {
                    assert !components[i]
                }
            }
        }
        p
    }

    SingleTerm getLambda() {
        def x = rule.attachment?.lambdaClosure?.call(this)
        if (x instanceof TermList) {
            assert x.size() == 1        // S -> foo wraps in TermList
            return (SingleTerm) x[0]
        } else {
            return (SingleTerm) x
        }
    }
}
```

```
package fol

import java.util.regex.Pattern

import parser.Parser
import grammar.Grammar
import parser.earley.*
import fol.lambda.*

/**
 * First-Order Logic with Lambda notation.
 * The grammar is based on textbook figure 17.3, plus λ-notation.
 * This supports λ-reductions (a.k.a. β-reductions),
 * but does not go so far as to perform logical inference or productions.
 */
class FirstOrderLogic {

    static final SYMBOLIC_CHARS = '¬∧∨⇒∀∃().,'
    static final LAMBDA = 'λ'
    static final Pattern LEXER = ~("[${SYMBOLIC_CHARS}${LAMBDA}]|" + /\w+/)

    static final GRAMMAR = new Grammar("""S -> Formula
        Formula -> LambdaFormula | QuantifiedFormula | LogicFormula
        Formula -> AtomicFormula | VariableApplication | ParentheticalFormula
        ParentheticalFormula -> ( Formula )
        LambdaFormula -> LambdaAbstraction | LambdaApplication
        LambdaAbstraction -> λ Variable . Formula | λ AbstractionVariable . Formula
        LambdaApplication -> LambdaAbstraction ( TermOrFormula )
        QuantifiedFormula -> Quantifier VariableList Formula
        LogicFormula -> Formula Connective Formula | ¬ Formula
        AtomicFormula -> Predicate ( TermList )
        VariableApplication -> AbstractionVariable ( TermOrFormula )
        TermOrFormula -> Term | Formula
        VariableList -> Variable | Variable , VariableList
        TermList -> Term | Term , TermList
        Term -> Function ( TermList ) | Constant | Variable
        Connective -> ∧ | ∨ | ⇒
        Quantifier -> ∀ | ∃
        Constant -> VegetarianFood | Maharani | AyCaramba | Bacaro | Centro | Leaf
        Constant -> Speaker | TurkeySandwich | Desk | Lunch | FiveDollars | LotOfTime
        Constant -> Monday | Tuesday | Wednesday | Thursday | Friday | Saturday |
Sunday
        Constant -> Yesterday | Today | Tomorrow | Now
        Constant -> NewYork | Boston | SanFrancisco
        Constant -> Matthew | Franco | Frasca
        Variable -> a | b | c | d | e | f | g | h | i | j | k | l | m
        Variable -> n | o | p | q | r | s | t | u | v | w | x | y | z
        AbstractionVariable -> A | B | C | D | E | F | G | H | I | J | K | L | M
        AbstractionVariable -> N | O | P | Q | R |     T | U | V | W | X | Y | Z
        Predicate -> Serves | Near | Restaurant | Have | VegetarianRestaurant
        Predicate -> Eating | Time | Eater | Eaten | Meal | Location
        Predicate -> Arriving | Arriver | Destination | EndPoint | Precedes
        Predicate -> Closed | ClosedThing | Opened | Opener
        Predicate -> Menu | Having | Haver | Had
        Function -> LocationOf | CuisineOf | IntervalOf | MemberOf """)

    static List<String> getVariablesNames() {
        GRAMMAR.rulesFor('Variable').collect { it.symbols[0] }
    }
```

```
    static List<String> getAbstractionVariablesNames() {
        GRAMMAR.rulesFor('AbstractionVariable').collect { it.symbols[0] }
    }

    static EarleyParser parse(String input) {
        def result = new EarleyParser(input, GRAMMAR, LEXER)
        def count = result.completedParses.size()
        switch (count) {
            case 0:
                throw new IllegalArgumentException("unparsable input $input\n chart:
$result")
            case 1:
                return result
            default:
                def prettyParses = Parser.prettyPrint(result.completedParsesString)
                def detail = "chart: $result \n has multiple parses: \n $prettyParses"
                throw new IllegalArgumentException("ambiguous input ($count parses)
$input\n $detail")
        }
    }

    static EarleyState parseTree(String input) {
        def p = parse(input)
        assert p.completedParses.size() == 1
        p.completedParses[0]
    }

    static TermList parseLambda(String input) {
        def ep = parse(input)
        (TermList) buildLambda((EarleyState) ep.completedParses[0])
    }

    private static buildLambda(EarleyState folParse) {
        assert folParse.complete
        def symbols = folParse.rule.symbols
        def translateToTermList = {
            def results = []
            for (i in 0..<symbols.size()) {
                def c = folParse.components[i]
                results << (c ? buildLambda(c) : new Symbol(symbols[i]))
            }
            new TermList(results.flatten())
        }
        def build = { int i -> buildLambda(folParse.components[i])}
        def buildSingle =  { int i ->
            def terms = build(i)
            assert terms instanceof TermList && terms.size() == 1
            (SingleTerm) terms[0]
        }
        def translations = [:].withDefault {translateToTermList}
        translations << [   // preserving default
                'LambdaAbstraction':    {new Abstraction(boundVar: (Variable) build(1),
expr: (TermList) build(3))},
                'LambdaApplication':    {new Application(abstractionTerm: (Abstraction)
build(0), term: buildSingle(2))},
                'Variable':             {new Variable(symbols[0])},
                'AbstractionVariable':  {new Variable(symbols[0])},
                'VariableApplication':  {new VariableApplication((Variable) build(0),
buildSingle(2))},
        ]
        def handler = (Closure) translations[folParse.rule.nonTerminal]
        handler()
    }
```

```
}
```

Abstraction.groovy

```groovy
package fol.lambda

import groovy.transform.EqualsAndHashCode
import fol.FirstOrderLogic

/**
 * LambdaAbstraction -> λ Variable . Formula | λ AbstractionVariable . Formula
 */
@EqualsAndHashCode
class Abstraction extends SingleTerm {
    Variable boundVar
    TermList expr

    Abstraction substitution(Variable v, SingleTerm e) {
        if (v == boundVar) {
            return this     // (λx.M)[x := N] ≡ λx.M    (stop recursion and preserve
binding)
        } else {            // (λy.M)[x := N] ≡ λy.(M[x := N]), if x ≠ y, provided y ∉
FV(N)
            assert !(boundVar in e.freeVariables) : 'needed alpha-conversion'
            return new Abstraction(boundVar: boundVar, expr: expr.substitute(v, e))
        }
    }

    TermList reduction() {
        new TermList([new Abstraction(boundVar: boundVar, expr: expr.reduction())])
    }

    // This reduction's substitutions will be safe if no term.freeVariables are bound
within abstraction.expr.
    // So, alpha-convert each colliding bound variable so that it is fresh (i.e., in
neither term.freeVariables
    // nor that abstraction's freeVariables).  This alpha-conversion does not disturb
the results,
    // because the variable bound in an abstraction is like a local variable; any
alpha-conversions of it
    // will not be visible after the abstraction is reduced in an application.
    Abstraction freshen(Collection<Variable> forbiddenVars) {
        if (boundVar in forbiddenVars) {
            // alpha-convert this boundVar, and freshen expr
            def v = findFresh(forbiddenVars)
            return new Abstraction(boundVar: v, expr: expr.substitute(boundVar,
v).freshen(forbiddenVars))
        } else {
            return new Abstraction(boundVar: boundVar, expr:
expr.freshen(forbiddenVars))
        }
    }

    private Variable findFresh(Collection<Variable> forbiddenVars) {
        forbiddenVars += freeVariables
        for (n in candidateNames()) {
            Variable w = new Variable(n)
            if (!(w in forbiddenVars)) {
                return w
            }
        }
        throw new IllegalStateException("no fresh variable names available for
$boundVar")
    }
```

```
    private List<String> candidateNames() {
        def name = boundVar.name
        List varNames = FirstOrderLogic.variablesNames
        def idx = varNames.indexOf(name)
        if (idx == -1) {
            varNames = FirstOrderLogic.abstractionVariablesNames
            assert name in varNames   // had to be one type or the other
            idx = varNames.indexOf(name)
            assert idx != -1
        }
        if (++idx < varNames.size()) {  // order by next name to try, and wrap
            varNames = varNames.subList(idx, varNames.size()) + varNames.subList(0, idx
- 1)
        }
        varNames
    }

    Set<Variable> getFreeVariables() {
        expr.freeVariables - boundVar            // FV(λx.M) = FV(M) - {x}
    }

    String toString() {
        def exprStr = expr.toString()   // hack: to get GString to use List's
overridden toString()
        "λ${boundVar}.($exprStr)"
    }
}
```

Application.groovy

```groovy
package fol.lambda

import groovy.transform.EqualsAndHashCode
import fol.FirstOrderLogic

/**
 * LambdaApplication -> LambdaAbstraction ( TermOrFormula )
 */
@EqualsAndHashCode
class Application extends SingleTerm {
    SingleTerm abstractionTerm
    SingleTerm term

    Application substitution(Variable v, SingleTerm e) {    // (M N)[x := P] ≡ (M[x := P]) (N[x := P])
        new Application(abstractionTerm: abstractionTerm.substitution(v, e), term: term.substitution(v, e))
    }

    Set<Variable> getFreeVariables() {
        abstractionTerm.freeVariables + term.freeVariables         // FV(M N) = FV(M) ∪ FV(N)
    }

    // Lambda-reduction (a.k.a. beta-reduction)
    TermList reduction() {
        switch (abstractionTerm) {

            case Abstraction:   // do the real reduction
                // This reduction's substitutions will be safe if no term.freeVariables are bound within abstraction.expr.
                // So, alpha-convert each colliding bound variable so that it is fresh (i.e., in neither term.freeVariables
                // nor that abstraction's freeVariables).  This alpha-conversion does not disturb the results,
                // because the variable bound in an abstraction is like a local variable; any alpha-conversions of it
                // will not be visible after the abstraction is reduced in an application.
                def freshened = abstractionTerm.expr.freshen(term.freeVariables)
                return freshened.substitute(abstractionTerm.boundVar, term)

            case Application:   // try to reduce abstractionTerm to an Abstraction
                def x = reduceToSingleTerm('abstractionTerm')
                def y = reduceToSingleTerm('term')
                return new TermList([new Application(abstractionTerm: x, term: y)])

            default:
                throw new IllegalStateException("abstractionTerm is $abstractionTerm")
        }
    }

    private SingleTerm reduceToSingleTerm(String property) {
        def t = this[property]
        def reduced = t.reduction().flatten()
        if (reduced.size() > 1) {
            throw new IllegalStateException("$property $t reduced to multiple terms $reduced")
        }
        (SingleTerm) reduced[0]
```

```
    }

    Application freshen(Collection<Variable> forbiddenVars) {
        new Application(abstractionTerm: abstractionTerm.freshen(forbiddenVars), term:
term.freshen(forbiddenVars))
    }

    String toString() {
        "$abstractionTerm($term)"
    }
}
```

SingleTerm.groovy

```
package fol.lambda

abstract class SingleTerm implements Term {

    abstract SingleTerm substitution(Variable v, SingleTerm e)

    abstract TermList reduction()

    abstract SingleTerm freshen(Collection<Variable> forbiddenVars)

    List<TermList> getNormalization() {
        new TermList([this]).normalization
    }

    String getNormalizationString() {
        new TermList([this]).normalizationString
    }
}
```

Symbol.groovy

```groovy
package fol.lambda

import groovy.transform.EqualsAndHashCode

/**
 * First Order Logic payload in the Lambda framework
 */
@EqualsAndHashCode
class Symbol extends SingleTerm {
    String symbol                       // disregarding logic etc

    Symbol(String s) {
        symbol = s
    }

    Symbol substitution(Variable v, SingleTerm e) {
        this    // unchanged
    }

    TermList reduction() {
        new TermList([this])    // cannot reduce
    }

    Symbol freshen(Collection<Variable> forbiddenVars) {
        this    // cannot freshen
    }

    Set<Variable> getFreeVariables() {
        []      // none
    }

    String toString() {
        symbol
    }
}
```

Term.groovy

```groovy
package fol.lambda

interface Term {

    Set<Variable> getFreeVariables()
}
```

TermList.groovy

```groovy
package fol.lambda

import fol.FirstOrderLogic

/**
 * Extension of Lambda Calculus to allow arbitrary lists of symbols of First Order Logic.
 * This uses ArrayList's equals() and hashCode(), not the Groovy @EqualsAndHashCode,
 * because the latter seems to rely on the subclass' (non-existent) properties and ignore the superclass'.
 */
class TermList extends ArrayList<SingleTerm> {

    TermList(Collection terms) {
        super(terms.collect { (SingleTerm) it instanceof String ? new Symbol(it) : it })

        if (size() == 3 && this[0] == new Symbol('(') && this[2] == new Symbol(')')) {
            // hack to remove certain extra parenthesis added to disambiguate parse
            remove(2)
            remove(0)
        }
    }

    TermList substitute(Variable v, SingleTerm e) {     // (M N)[x := P] ≡ (M[x := P]) (N[x := P])
        new TermList(this.collect { it.substitution(v, e) }.flatten())
    }

    Set<Variable> getFreeVariables() {
        // NB: cannot use this.freeVariables.flatten(); GPaths do not seem to work properly inside the List itself
        this.collect { it.freeVariables }.flatten() as Set<Variable>     // FV(M N) = FV(M) ∪ FV(N)
    }

    TermList reduction() {
        new TermList(this.collect { it.reduction() }.flatten())
    }

    TermList freshen(Collection<Variable> forbiddenVars) {
        new TermList(this.collect { it.freshen(forbiddenVars) })
    }

    List<TermList> getNormalization() {
        def result = [this]
        while (true) {
            def last = result[-1]
            def next = last.reduction()
            if (next == last) {     // normalized
                return result
            }
            if (next in result) {
                throw new IllegalStateException('' + this + ' does not normalize; ' + result + ' loops to ' + next)
            }
            result << next
        }
    }
```

```
    String getNormalizationString() {
        // Cannot use just Groovy's join(), because for the component Lists,
        // join() would use its own InvokerHelper.formatList() instead of this class'
toString().
        normalization*.toString().join('\n')
    }

    @Override
    public String toString() {
        def result = ''
        def prev = null
        for (term in this) {
            if (needsSpaceBetween(prev, term)) {
                result += ' '
            }
            result += term
            prev = term
        }
        result
    }

    private boolean needsSpaceBetween(Term t, Term u) {
        t && FirstOrderLogic.LEXER.matcher("${t.toString()[-1]}${u.toString()
[0]}").size() != 2
    }
}
```

Variable.groovy

```groovy
package fol.lambda

import groovy.transform.EqualsAndHashCode

/**
 * FOL or Lambda variable
 */
@EqualsAndHashCode  // using name for identity
class Variable extends SingleTerm {
    String name

    Variable(String s) {
        name = s
    }

    SingleTerm substitution(Variable v, SingleTerm e) {
        if (this == v) {        // x[x := N] ≡ N
            return e            // substituted!
        } else {
            assert name != v.name
            return this        // y[x := N] ≡ y, if x ≠ y
        }
    }

    TermList reduction() {
        new TermList([this])    // cannot reduce
    }

    Variable freshen(Collection<Variable> forbiddenVars) {
        this    // cannot freshen
    }

    Set<Variable> getFreeVariables() {
        [this]                  // FV(x) = {x}, where x is a variable
    }

    String toString() {
        name
    }
}
```

VariableApplication.groovy

```groovy
package fol.lambda

import groovy.transform.EqualsAndHashCode

/**
 * VariableApplication -> AbstractionVariable ( TermOrFormula )
 */
@EqualsAndHashCode
class VariableApplication extends SingleTerm {
    Variable boundAbstractionVar
    SingleTerm term

    VariableApplication(Variable v, SingleTerm t) {
        boundAbstractionVar = v
        term = t
    }

    SingleTerm substitution(Variable v, SingleTerm e) {
        if (v == boundAbstractionVar) {
            if (e instanceof Variable) {    // alpha-conversion of this boundAbstractionVar
                return new VariableApplication(e, term.substitution(v, e))
            }
            assert e.class in [Abstraction, Application, VariableApplication] : "substituted unsuitable for $v: $e"
            return new Application(abstractionTerm: e, term: term.substitution(v, e))
        } else {
            return new VariableApplication(boundAbstractionVar, term.substitution(v, e))
        }
    }

    TermList reduction() {
        new TermList([this])        // cannot reduce; don't know yet what alpha-conversions will be needed
    }

    VariableApplication freshen(Collection<Variable> forbiddenVars) {
        new VariableApplication(boundAbstractionVar.freshen(forbiddenVars), term.freshen(forbiddenVars))
    }

    Set<Variable> getFreeVariables() {
        boundAbstractionVar.freeVariables + term.freeVariables         // FV(M N) = FV(M) ∪ FV(N)
    }

    String toString() {
        "$boundAbstractionVar($term)"
    }
}
```

```
import grammar.Grammar
import parser.earley.EarleyParser
import fol.lambda.SingleTerm
import parser.earley.EarleyState
import fol.lambda.TermList

/**
 * A grammar with semantic attachments, based on the textbook's figure 18.4.
 * The semantics are in FirstOrderLogic with lambda expressions.
 */
class SemanticGrammar {

    static final GRAMMAR = new Grammar('''  S -> NP VP    [1, $0($1)]
        NP -> Det Nominal           [.4, $0($1)]
        NP -> ProperNoun            [.6, $0]
        Nominal -> Noun             [1, $0]
        VP -> Verb                  [.7, $0]
        VP -> Verb NP               [.3, $0($1)]
        Det -> every                [.2, λP.(λQ.∀x(P(x)⇒Q(x)))]
        Det -> a                    [.8, λP.(λQ.∃x(P(x)∧Q(x)))]
        Noun -> restaurant          [1, λr.Restaurant(r)]
        ProperNoun -> Matthew       [.3, λM.M(Matthew)]
        ProperNoun -> Maharani      [.2, λM.M(Maharani)]
        ProperNoun -> Franco        [.25, λF.F(Franco)]
        ProperNoun -> Frasca        [.25, λF.F(Frasca)]
        Verb -> closed              [.4, λx.∃e(Closed(e)∧ClosedThing(e,x))]
        Verb -> opened              [.6, λW.
(λz.W(λx.∃e(Opened(e)∧(Opener(e,z)∧Opened(e,x)))))]''')

    /**
     * Constructs an EarleyParser, parsing the given line with the semantic Grammar.
     *
     * @param line the line of words to parse (e.g., a sentence)
     */
    static EarleyParser parse(String input) {
        new EarleyParser(input, GRAMMAR)
    }

    /**
     * Gets complete parses of the given input (sorted by probability, if any).
     *
     * @param input to parse
     * @return a list of complete parses
     * @throws IllegalArgumentException if no complete parse is found
     */
    private static List<EarleyState> parses(String input) {
        def p = parse(input)
        def parses = (List<EarleyState>) p.completedParses
        if (!parses) {
            throw new IllegalArgumentException("unparsable input '$input': $p")
        }
        parses
    }

    /**
     * Gets the immediate (un-normalized) semantic representation of the most probable
parse of the given input.
     * The caller can normalize the result to get the derivation of the fully reduced
form.
     *
```

```
     * @param input to parse
     * @return the Lambda representation of the first (i.e., most probable) complete
parse of the input
     */
    static SingleTerm parseSemanticsDerivation(String input) {
        parses(input)[0].lambda
    }

    /**
     * Gets the normalized (i.e., fully reduced) semantic representation of the most
probable parse of the given input.
     *
     * @param input to parse
     * @return the fully reduced Lambda representation
     */
    static TermList parseSemantics(String input) {
        parseSemanticsDerivation(input).normalization[-1]
    }

    /**
     * Renders all accepted, full parses into easy-to-read indentation.
     *
     * @return a rendering of all possible parses, or "not S" if none are accepted
     */
    static String prettyCompletedSemanticParses(String input) {
        def p = parse(input)
        def parses = p.completedParses
        if (parses) {
            return parses.collect {
                it.prettyPrint(0) + '\n-\n' + it.lambda.normalizationString
            }.join('\n;')
        } else {
            return "not ${GRAMMAR.startSymbol}"
        }
    }

    /**
     * For running from the command line (from the "src" dir), this parses lines from
stdin.
     *
     * @param args  command line arguments (unused)
     */
    static void main(String[] args) {
        System.in.eachLine {line ->
            println "\n$line: " + prettyCompletedSemanticParses(line)
            null    // just avoiding a warning about not returning a value from
eachLine
        }
    }
}
```

```groovy
import spock.lang.Specification
import spock.lang.Unroll
import grammar.Grammar

/**
 * Test specification of Grammar.
 */
class GrammarSpec extends Specification {


    def 'L1 non-terminals'() {

        given:
        def g = new Grammar(L1_DEF)

        expect:
        g.nonTerminals == 'S NP Nominal VP PP Det Noun Verb Pronoun Proper-Noun Aux
Preposition'.split() as Set
    }

    def 'L1 terminals'() {

        given:
        def g = new Grammar(L1_DEF)
        def terminals = 'that this a the book flight meal money flights dinner include
prefer I she me you ' +
                'Houston NWA does can from to on near through'

        expect:
        g.terminals == terminals.split() as Set
    }

    // works around Spock (or JUnit?) problems with having both \n and . in test names
    static String backslashToNewline(String s) {
        s.replaceAll('\\\\', '\n')
    }

    @Unroll
    def 'grammar "#description" normalizes to "#expected"'() {

        given:
        def g = new Grammar(backslashToNewline(description))

        when:
        g.normalize()

        then:
        g.toString() == backslashToNewline(expected)

        where:
        description                                  || expected

        // converting terminals to non-terminals
        'S -> a B [1]\\B -> C d [1]\\C -> e f [1]'  || 'S -> X1 B [1]\\X1 -> a [1]\\B
-> C X2 [1]\\X2 -> d [1]\\C -> X3 X4 [1]\\X3 -> e [1]\\X4 -> f [1]'
        'S -> a b [1]'                               || 'S -> X1 X2 [1]\\X1 -> a [1]\\X2
-> b [1]'
        'S -> A a [1]\\A -> a [1]'                   || 'S -> A X1 [1]\\X1 -> a [1]\\A
-> a [1]'   // A -> X1 undone by unit productions
```

```
        // converting unit productions
        'S -> VP [1]\\VP -> Verb [1]\\Verb -> book [.3]\\Verb -> include [.3]\\Verb ->
prefer [.4]' || 'S -> book [.3]\\S -> include [.3]\\S -> prefer [.4]'
        'S -> VP Verb [1]\\VP -> Verb [1]\\Verb -> book [.7]\\Verb -> fly [.3]'
|| 'S -> VP Verb [1]\\VP -> book [.7]\\VP -> fly [.3]\\Verb -> book [.7]\\Verb -> fly
[.3]'

        // making all rules binary
        'S -> A B C [1]\\A -> a [1]\\B -> b [1]\\C -> c [1]'                    || 'S
-> X1 C [1]\\X1 -> A B [1]\\A -> a [1]\\B -> b [1]\\C -> c [1]'
        'S -> A B C D [1]\\A -> a [1]\\B -> b [1]\\C -> c [1]\\D -> d [1]'      || 'S
-> X2 D [1]\\X2 -> X1 C [1]\\X1 -> A B [1]\\A -> a [1]\\B -> b [1]\\C -> c [1]\\D -> d
[1]'
        'S -> A B C [1]\\A -> A B C [.2]\\A -> a [.8]\\B -> b [1]\\C -> c [1]'  || 'S
-> X1 C [1]\\A -> X1 C [.2]\\X1 -> A B [1]\\A -> a [.8]\\B -> b [1]\\C -> c [1]'
        'S -> a b c [1]'                                                        || 'S
-> X4 X3 [1]\\X4 -> X1 X2 [1]\\X1 -> a [1]\\X2 -> b [1]\\X3 -> c [1]'
    }

    @Unroll
    def 'grammar without attachments "#description" normalizes to "#expected"'() {

        given:
        def g = new Grammar(backslashToNewline(description))

        when:
        g.normalize()

        then:
        g.toString() == backslashToNewline(expected)

        where:
        description                                 || expected

        // converting terminals to non-terminals
        'S -> a B\\B -> C d\\C -> e f'  || 'S -> X1 B\\X1 -> a\\B -> C X2\\X2 -> d\\C
-> X3 X4\\X3 -> e\\X4 -> f'
        'S -> a b'                      || 'S -> X1 X2\\X1 -> a\\X2 -> b'
        'S -> A a\\A -> a'              || 'S -> A X1\\X1 -> a\\A -> a'   // A -> X1
undone by unit productions

        // converting unit productions
        'S -> VP\\VP -> Verb\\Verb -> book\\Verb -> include\\Verb -> prefer'    || 'S
-> book\\S -> include\\S -> prefer'
        'S -> VP Verb\\VP -> Verb\\Verb -> book\\Verb -> fly'                  || 'S
-> VP Verb\\VP -> book\\VP -> fly\\Verb -> book\\Verb -> fly'

        // making all rules binary
        'S -> A B C\\A -> a\\B -> b\\C -> c'                || 'S -> X1 C\\X1 -> A B\\A
-> a\\B -> b\\C -> c'
        'S -> A B C D\\A -> a\\B -> b\\C -> c\\D -> d'      || 'S -> X2 D\\X2 -> X1
C\\X1 -> A B\\A -> a\\B -> b\\C -> c\\D -> d'
        'S -> A B C\\A -> A B C\\A -> a\\B -> b\\C -> c'    || 'S -> X1 C\\A -> X1
C\\X1 -> A B\\A -> a\\B -> b\\C -> c'
        'S -> a b c'                                       || 'S -> X4 X3\\X4 -> X1
X2\\X1 -> a\\X2 -> b\\X3 -> c'
    }

    def "assignment 3 requirement"() {

        given:
        def g = new Grammar(L1_DEF)
```

```
        when:
        g.normalize()

        then:
        g.toString() == L1_CNF
    }

    def "normalization is stable"() {

        given:
        def g = new Grammar(input)

        when:
        g.normalize()
        def firstNormal = g.toString()

        and: 'again'
        g.normalize()
        def secondNormal = g.toString()

        then:
        firstNormal == secondNormal

        where:
        input << [L1_DEF, L1_CNF]
    }

    def "normalization is reflexive"() {

        given:
        def g = new Grammar(L1_CNF)

        when:
        g.normalize()

        then:
        g.toString() == L1_CNF
    }

    def 'split does not work the way Grammar would need it to'() {

        expect:
        'abc'.split(/\b/) == ['', 'abc']
        'a c'.split(/\b/) == ['', 'a', ' ', 'c']
        'a.c'.split(/\b/) == ['', 'a', '.', 'c']
        'λx.x'.split(/\b/) == ['', 'λx', '.', 'x']
        'a,b'.split(/\b/) == ['', 'a', ',', 'b']
        'a, b'.split(/\b/) == ['', 'a', ', ', 'b']
        'a) ∧ ¬b'.split(/\b/) == ['', 'a', ') ∧ ¬', 'b']
    }

    @Unroll
    def '#pattern matches "#input" as #expected'() {

        expect:
        pattern.matcher(input).collect {it} == expected

        where:
        input       | pattern       | expected
        'abc'       | ~/\w+/        | ['abc']
        'a c'       | ~/\w+/        | ['a', 'c']
        'λx.x'      | ~/\w+/        | ['x', 'x']     // not sure why λ is not in \w,
but don't want it there anyway
```

```
        'λx.x'      | ~/[λ.]|\w+/   | ['λ', 'x', '.', 'x']
        'a,b'       | ~/\w+/        | ['a', 'b']
        'a,b'       | ~/[,]|\w+/    | ['a', ',', 'b']
        'a, b'      | ~/\w+/        | ['a', 'b']
        'a, b'      | ~/[,]|\w+/    | ['a', ',', 'b']
        'a) ∧ ¬b'   | ~/\w+/        | ['a', 'b']
        'a) ∧ ¬b'   | ~/[)∧¬]|\w+/  | ['a', ')', '∧', '¬', 'b']
    }

    static final L1_DEF = """S -> NP VP     [.80]
S -> Aux NP VP               [.15]
S -> VP                      [.05]
NP -> Pronoun                [.35]
NP -> Proper-Noun            [.30]
NP -> Det Nominal            [.20]
NP -> Nominal                [.15]
Nominal -> Noun              [.75]
Nominal -> Nominal Noun      [.20]
Nominal -> Nominal PP        [.05]
VP -> Verb                   [.35]
VP -> Verb NP                [.20]
VP -> Verb NP PP             [.10]
VP -> Verb PP                [.15]
VP -> Verb NP NP             [.05]
VP -> VP PP                  [.15]
PP -> Preposition NP         [1]
Det -> that                  [.10]
Det -> this                  [.05]
Det -> a                     [.25]
Det -> the                   [.60]
Noun -> book                 [.10]
Noun -> flight               [.25]
Noun -> meal                 [.15]
Noun -> money                [.05]
Noun -> flights              [.35]
Noun -> dinner               [.10]
Verb -> book                 [.30]
Verb -> include              [.30]
Verb -> prefer               [.40]
Pronoun -> I                 [.40]
Pronoun -> she               [.05]
Pronoun -> me                [.15]
Pronoun -> you               [.40]
Proper-Noun -> Houston       [.60]
Proper-Noun -> NWA           [.40]
Aux -> does                  [.60]
Aux -> can                   [.40]
Preposition -> from          [.30]
Preposition -> to            [.30]
Preposition -> on            [.20]
Preposition -> near          [.15]
Preposition -> through       [.05]"""

    static final L1_CNF = """S -> NP VP [.80]
S -> X1 VP [.15]
X1 -> Aux NP [1]
S -> book [.005250]
S -> include [.005250]
S -> prefer [.007000]
S -> Verb NP [.0100]
S -> X2 PP [.0050]
S -> Verb PP [.0075]
S -> X2 NP [.0025]
```

```
S -> VP PP [.0075]
NP -> I [.1400]
NP -> she [.0175]
NP -> me [.0525]
NP -> you [.1400]
NP -> Houston [.1800]
NP -> NWA [.1200]
NP -> Det Nominal [.20]
NP -> book [.011250]
NP -> flight [.028125]
NP -> meal [.016875]
NP -> money [.005625]
NP -> flights [.039375]
NP -> dinner [.011250]
NP -> Nominal Noun [.0300]
NP -> Nominal PP [.0075]
Nominal -> book [.0750]
Nominal -> flight [.1875]
Nominal -> meal [.1125]
Nominal -> money [.0375]
Nominal -> flights [.2625]
Nominal -> dinner [.0750]
Nominal -> Nominal Noun [.20]
Nominal -> Nominal PP [.05]
VP -> book [.1050]
VP -> include [.1050]
VP -> prefer [.1400]
VP -> Verb NP [.20]
VP -> X2 PP [.10]
VP -> Verb PP [.15]
VP -> X2 NP [.05]
X2 -> Verb NP [1]
VP -> VP PP [.15]
PP -> Preposition NP [1]
Det -> that [.10]
Det -> this [.05]
Det -> a [.25]
Det -> the [.60]
Noun -> book [.10]
Noun -> flight [.25]
Noun -> meal [.15]
Noun -> money [.05]
Noun -> flights [.35]
Noun -> dinner [.10]
Verb -> book [.30]
Verb -> include [.30]
Verb -> prefer [.40]
Aux -> does [.60]
Aux -> can [.40]
Preposition -> from [.30]
Preposition -> to [.30]
Preposition -> on [.20]
Preposition -> near [.15]
Preposition -> through [.05]"""
}
```

CkyParserSpec.groovy

```
import spock.lang.Specification
import spock.lang.Unroll

import static parser.Parser.prettyPrint
import parser.cky.CkyParser
import grammar.Grammar

/**
 * Test specification of CkyParser.
 */
class CkyParserSpec extends Specification {

    @Unroll
    def 'input "#input" parses as #expected'() {

        given:
        def g = new Grammar(L1_DEF)

        when:
        def p = new CkyParser(input, g)

        then:
        prettyPrint(p.completedParsesString) == prettyPrint(expected)

        where:
        input                                || expected
        'book'                               || '[S book {.005250}]'
        'book that flight'                   || '[S [Verb book {.30}] [NP [Det that
{.10}] [Nominal flight {.1875}] {.00375000}] {.00001125000000}]'
        'book that flight through Houston'   || '[S [X2 [Verb book {.30}] [NP [Det that
{.10}] [Nominal flight {.1875}] {.00375000}] {.00112500000}] [PP [Preposition through
{.05}] [NP Houston {.1800}] {.009000}] {5.0625000000000E-8}];[S [VP [Verb book {.30}]
[NP [Det that {.10}] [Nominal flight {.1875}] {.00375000}] {.000225000000}] [PP
[Preposition through {.05}] [NP Houston {.1800}] {.009000}] {1.51875000000000E-8}];[S
[Verb book {.30}] [NP [Det that {.10}] [Nominal [Nominal flight {.1875}] [PP
[Preposition through {.05}] [NP Houston {.1800}] {.009000}] {.000084375000}]
{.000016875000000}] {5.0625000000000E-9}]'
        'does this flight include a meal'    || '[S [X1 [Aux does {.60}] [NP [Det this
{.05}] [Nominal flight {.1875}] {.00187500}] {.00112500000}] [VP [Verb include {.30}]
[NP [Det a {.25}] [Nominal meal {.1125}] {.00562500}] {.000337500000}]
{5.69531250000000000E-8}]'
        'I prefer NWA'                       || '[S [NP I {.1400}] [VP [Verb prefer
{.40}] [NP NWA {.1200}] {.00960000}] {.00107520000000}]'
        'I prefer a flight to Houston'       || '[S [NP I {.1400}] [VP [X2 [Verb prefer
{.40}] [NP [Det a {.25}] [Nominal flight {.1875}] {.00937500}] {.00375000000}] [PP
[Preposition to {.30}] [NP Houston {.1800}] {.054000}] {.000020250000000000000}]
{.00000226800000000000000000}];[S [NP I {.1400}] [VP [VP [Verb prefer {.40}] [NP [Det a
{.25}] [Nominal flight {.1875}] {.00937500}] {.000750000000}] [PP [Preposition to
{.30}] [NP Houston {.1800}] {.054000}] {.000006075000000000000}]
{6.804000000000000000E-7}];[S [NP I {.1400}] [VP [Verb prefer {.40}] [NP [Det a {.25}]
[Nominal [Nominal flight {.1875}] [PP [Preposition to {.30}] [NP Houston {.1800}]
{.054000}] {.000506250000}] {.0000253125000000}] {.000002025000000000000}]
{2.268000000000000000E-7}]'
        'book the flight'                    || '[S [Verb book {.30}] [NP [Det the
{.60}] [Nominal flight {.1875}] {.02250000}] {.00006750000000}]'
        'book flight'                        || '[S [Verb book {.30}] [NP flight
{.028125}] {.000084375000}]'
        'book flight that'                   || 'not S'  // missing rule
        'book the prefer'                    || 'not S'  // missing rule
        'flight flight'                      || 'not S'  // missing rule
```

```
        'book those flights'                || 'not S'  // 'those' is missing from the
lexicon (not a Det)
        'does this flight include a lunch'  || 'not S'  // 'lunch' is missing from the
lexicon (not a Nominal)
    }

    @Unroll
    def 'input "#input" prettyPrints to #expected'() {

        expect:
        prettyPrint(input) == GrammarSpec.backslashToNewline(expected)

        where:
        input                                           || expected
        '[S book {.123}]'                               || '[S book {.123}]'
        '[S [NP I {.123}] [VP [Verb prefer {.123}] [NP NWA {.123}] {.123}] {.123}]'
|| '[S\\    [NP I {.123}]\\    [VP\\        [Verb prefer {.123}]\\        [NP NWA
{.123}]\\     {.123}]\\ {.123}]'
        '[S book {.123}];[S chair {.123}]'          || '[S book {.123}]\\;\\[S chair
{.123}]'
    }

    def 'work-around to avoid unresolved test name in IntelliJ'() {

        expect:
        true        // not a real test
    }

    static final L1_DEF = """S -> NP VP     [.80]
S -> Aux NP VP              [.15]
S -> VP                     [.05]
NP -> Pronoun               [.35]
NP -> Proper-Noun           [.30]
NP -> Det Nominal           [.20]
NP -> Nominal               [.15]
Nominal -> Noun             [.75]
Nominal -> Nominal Noun     [.20]
Nominal -> Nominal PP       [.05]
VP -> Verb                  [.35]
VP -> Verb NP               [.20]
VP -> Verb NP PP            [.10]
VP -> Verb PP               [.15]
VP -> Verb NP NP            [.05]
VP -> VP PP                 [.15]
PP -> Preposition NP        [1]
Det -> that                 [.10]
Det -> this                 [.05]
Det -> a                    [.25]
Det -> the                  [.60]
Noun -> book                [.10]
Noun -> flight              [.25]
Noun -> meal                [.15]
Noun -> money               [.05]
Noun -> flights             [.35]
Noun -> dinner              [.10]
Verb -> book                [.30]
Verb -> include             [.30]
Verb -> prefer              [.40]
Pronoun -> I                [.40]
Pronoun -> she              [.05]
Pronoun -> me               [.15]
Pronoun -> you              [.40]
Proper-Noun -> Houston      [.60]
```

```
Proper-Noun -> NWA          [.40]
Aux -> does                 [.60]
Aux -> can                  [.40]
Preposition -> from         [.30]
Preposition -> to           [.30]
Preposition -> on           [.20]
Preposition -> near         [.15]
Preposition -> through      [.05]"""
}
```

EarleyParserSpec.groovy

```
import spock.lang.Specification

import static parser.Parser.prettyPrint
import grammar.Grammar
import parser.earley.EarleyParser

/**
 * Test specification of EarleyParser.
 */
class EarleyParserSpec extends Specification {

    def '"book that flight" parses as expected'() {

        given:
        def p = new EarleyParser('book that flight', new Grammar(L1_DEF))

        expect:
        p.toString() == EXPECTED_CHART
        prettyPrint(p.completedParsesString) == EXPECTED_PARSE
    }

    static final L1_DEF = """S -> NP VP | Aux NP VP | VP
NP -> Pronoun | Proper-Noun | Det Nominal
Nominal -> Noun | Nominal Noun  | Nominal PP
VP -> Verb | Verb NP | Verb NP PP | Verb PP | VP PP
PP -> Preposition NP
Det -> that | this | a
Noun -> book | flight | meal | money
Verb -> book | include | prefer
Pronoun -> I | she | me
Proper-Noun -> Houston | NWA
Aux -> does
Preposition -> from | to | on | near | through"""

    static final EXPECTED_CHART = """Chart[0]\tS0\tγ -> • S\t[0, 0]\tdummy start state
\tS1\tS -> • NP VP\t[0, 0]\tpredictor
\tS2\tS -> • Aux NP VP\t[0, 0]\tpredictor
\tS3\tS -> • VP\t[0, 0]\tpredictor
\tS4\tNP -> • Pronoun\t[0, 0]\tpredictor
\tS5\tNP -> • Proper-Noun\t[0, 0]\tpredictor
\tS6\tNP -> • Det Nominal\t[0, 0]\tpredictor
\tS7\tVP -> • Verb\t[0, 0]\tpredictor
\tS8\tVP -> • Verb NP\t[0, 0]\tpredictor
\tS9\tVP -> • Verb NP PP\t[0, 0]\tpredictor
\tS10\tVP -> • Verb PP\t[0, 0]\tpredictor
\tS11\tVP -> • VP PP\t[0, 0]\tpredictor

Chart[1]\tS12\tVerb -> book •\t[0, 1]\tscanner
\tS13\tVP -> Verb •\t[0, 1]\tcompleter
\tS14\tVP -> Verb • NP\t[0, 1]\tcompleter
\tS15\tVP -> Verb • NP PP\t[0, 1]\tcompleter
\tS16\tVP -> Verb • PP\t[0, 1]\tcompleter
\tS17\tS -> VP •\t[0, 1]\tcompleter
\tS18\tVP -> VP • PP\t[0, 1]\tcompleter
\tS19\tNP -> • Pronoun\t[1, 1]\tpredictor
\tS20\tNP -> • Proper-Noun\t[1, 1]\tpredictor
\tS21\tNP -> • Det Nominal\t[1, 1]\tpredictor
\tS22\tPP -> • Preposition NP\t[1, 1]\tpredictor

Chart[2]\tS23\tDet -> that •\t[1, 2]\tscanner
```

```
\tS24\tNP -> Det • Nominal\t[1, 2]\tcompleter
\tS25\tNominal -> • Noun\t[2, 2]\tpredictor
\tS26\tNominal -> • Nominal Noun\t[2, 2]\tpredictor
\tS27\tNominal -> • Nominal PP\t[2, 2]\tpredictor

Chart[3]\tS28\tNoun -> flight •\t[2, 3]\tscanner
\tS29\tNominal -> Noun •\t[2, 3]\tcompleter
\tS30\tNP -> Det Nominal •\t[1, 3]\tcompleter
\tS31\tNominal -> Nominal • Noun\t[2, 3]\tcompleter
\tS32\tNominal -> Nominal • PP\t[2, 3]\tcompleter
\tS33\tVP -> Verb NP •\t[0, 3]\tcompleter
\tS34\tVP -> Verb NP • PP\t[0, 3]\tcompleter
\tS35\tPP -> • Preposition NP\t[3, 3]\tpredictor
\tS36\tS -> VP •\t[0, 3]\tcompleter
\tS37\tVP -> VP • PP\t[0, 3]\tcompleter"""

    static final EXPECTED_PARSE = """[S36 S
    [S33 VP
        [S12 Verb book (0,1)]
        [S30 NP
            [S23 Det that (1,2)]
            [S29 Nominal
                [S28 Noun flight (2,3)]
             (2,3)]
         (1,3)]
     (0,3)]
 (0,3)]"""
}
```

```groovy
import spock.lang.Specification
import spock.lang.Unroll

import static parser.Parser.prettyPrint
import fol.FirstOrderLogic

/**
 * Test specification of FirstOrderLogic.
 */
class FirstOrderLogicSpec extends Specification {

    @Unroll
    def 'symbolic char #c is not a letter, digit, or whitespace'() {

        expect:
        !c.isLetterOrDigit()
        !c.isWhitespace()

        where:
        c << FirstOrderLogic.SYMBOLIC_CHARS.toCharArray()
    }

    def 'λ is a letter'() {

        given:
        char c = 'λ'

        expect:
        c.isLetter()
    }

    def 'FOL grammar is not normalized'() {

        given:
        def g = FirstOrderLogic.GRAMMAR

        expect:
        g.toString() == FOL_DEF
    }

    def 'FOL "Restaurant(Maharani)" parses as expected'() {

        given:
        def input = 'Restaurant(Maharani)'

        when:
        def p = FirstOrderLogic.parseTree(input)

        then:
        p.prettyPrint() == """
[S
    [Formula
        [AtomicFormula
            [Predicate Restaurant]
            (
            [TermList
                [Term
                    [Constant Maharani]
                ]
            ]
```

```
            )
        ]
    ]
]"""

        when:
        def l = FirstOrderLogic.parseLambda(input)

        then:
        l.toString() == input

        and: 'already in normal form'
        l.normalizationString == input
    }


    def 'FOL "Restaurant(Maharani)" parses as expected with details'() {

        given:
        def input = 'Restaurant(Maharani)'

        when:
        def p = FirstOrderLogic.parse(input)

        then:
        prettyPrint(p.completedParsesString) == """[S34 S
    [S33 Formula
        [S32 AtomicFormula
            [S19 Predicate Restaurant (0,1)]
         (
            [S29 TermList
                [S28 Term
                    [S27 Constant Maharani (2,3)]
                 (2,3)]
             (2,3)] ) (0,4)]
     (0,4)]
 (0,4)]"""

        when:
        def l = FirstOrderLogic.parseLambda(input)

        then:
        l.toString() == input
    }

    def 'FOL "Have(Speaker,FiveDollars)∧¬Have(Speaker,LotOfTime)" parses as expected'()
{

        given:
        def input = 'Have(Speaker,FiveDollars)∧¬Have(Speaker,LotOfTime)'

        when:
        def p = FirstOrderLogic.parseTree(input)

        then:
        p.prettyPrint() == """
[S
    [Formula
        [LogicFormula
            [Formula
                [AtomicFormula
                    [Predicate Have]
                    (
```

```
                    [TermList
                        [Term
                            [Constant Speaker]
                        ]
                        ,
                        [TermList
                            [Term
                                [Constant FiveDollars]
                            ]
                        ]
                    ]
                    )
                ]
            ]
            [Connective ∧]
            [Formula
                [LogicFormula
                    ¬
                    [Formula
                        [AtomicFormula
                            [Predicate Have]
                            (
                            [TermList
                                [Term
                                    [Constant Speaker]
                                ]
                                ,
                                [TermList
                                    [Term
                                        [Constant LotOfTime]
                                    ]
                                ]
                            ]
                            )
                        ]
                    ]
                ]
            ]
        ]
    ]
]"""

        when:
        def l = FirstOrderLogic.parseLambda(input)

        then:
        l.toString() == input

        and: 'already in normal form'
        l.normalizationString == input
    }

    def 'FOL "∀x(VegetarianRestaurant(x)⇒Serves(x,VegetarianFood))" parses as expected'() {

        given:
        def input = '∀x(VegetarianRestaurant(x)⇒Serves(x,VegetarianFood))'

        when:
        def p = FirstOrderLogic.parseTree(input)

        then:
        p.prettyPrint() == """
```

```
[S
    [Formula
        [QuantifiedFormula
            [Quantifier ∀]
            [VariableList
                [Variable x]
            ]
            [Formula
                [ParentheticalFormula
                    (
                    [Formula
                        [LogicFormula
                            [Formula
                                [AtomicFormula
                                    [Predicate VegetarianRestaurant]
                                    (
                                    [TermList
                                        [Term
                                            [Variable x]
                                        ]
                                    ]
                                    )
                                ]
                            ]
                            [Connective ⇒]
                            [Formula
                                [AtomicFormula
                                    [Predicate Serves]
                                    (
                                    [TermList
                                        [Term
                                            [Variable x]
                                        ]
                                        ,
                                        [TermList
                                            [Term
                                                [Constant VegetarianFood]
                                            ]
                                        ]
                                    ]
                                    )
                                ]
                            ]
                        ]
                    ]
                    )
                ]
            ]
        ]
    ]
]"""

        when:
        def l = FirstOrderLogic.parseLambda(input)

        then:
        l.toString() == input

        and: 'already in normal form'
        l.normalizationString == input
    }

    def 'FOL "λx.λy.Near(x,y)" parses as expected'() {
```

```
        given:
        def input = 'λx.λy.Near(x,y)'
        def canonicalInput = 'λx.(λy.(Near(x,y)))'
        def expected = """
[S
    [Formula
        [LambdaFormula
            [LambdaAbstraction
                λ
                [Variable x]
                .
                [Formula
                    [LambdaFormula
                        [LambdaAbstraction
                            λ
                            [Variable y]
                            .
                            [Formula
                                [AtomicFormula
                                    [Predicate Near]
                                    (
                                    [TermList
                                        [Term
                                            [Variable x]
                                        ]
                                        ,
                                        [TermList
                                            [Term
                                                [Variable y]
                                            ]
                                        ]
                                    ]
                                    )
                                ]
                            ]
                        ]
                    ]
                ]
            ]
        ]
    ]
]"""

        expect:
        FirstOrderLogic.parseTree(input).prettyPrint() == expected
        FirstOrderLogic.parseLambda(input).toString() == canonicalInput

        and: 'already in normal form'
        FirstOrderLogic.parseLambda(input).normalizationString == canonicalInput
    }

    def 'FOL "λx.(λy.Near(x,y))(Bacaro)" parses as expected'() {

        given:
        def input = 'λx.(λy.Near(x,y))(Bacaro)'
        def canonicalInput = 'λx.(λy.(Near(x,y)))(Bacaro)'

        when:
        def p = FirstOrderLogic.parseTree(input)

        then:
        p.prettyPrint() == """
```

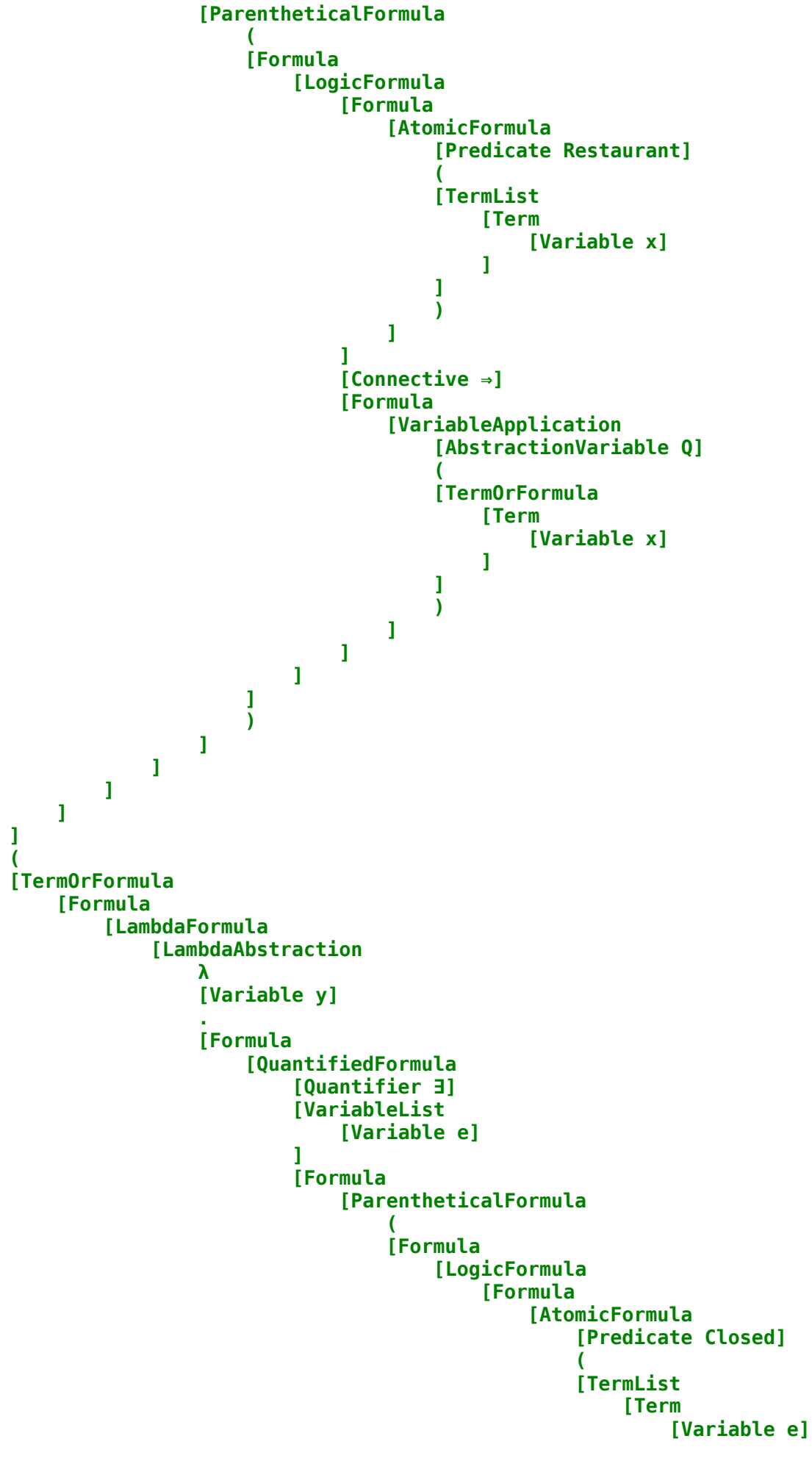
```
[S
    [Formula
        [LambdaFormula
            [LambdaApplication
                [LambdaAbstraction
                    λ
                    [Variable x]
                    .
                    [Formula
                        [ParentheticalFormula
                            (
                            [Formula
                                [LambdaFormula
                                    [LambdaAbstraction
                                        λ
                                        [Variable y]
                                        .
                                        [Formula
                                            [AtomicFormula
                                                [Predicate Near]
                                                (
                                                [TermList
                                                    [Term
                                                        [Variable x]
                                                    ]
                                                    ,
                                                    [TermList
                                                        [Term
                                                            [Variable y]
                                                        ]
                                                    ]
                                                ]
                                                )
                                            ]
                                        ]
                                    ]
                                ]
                            ]
                            )
                        ]
                    ]
                ]
                (
                [TermOrFormula
                    [Term
                        [Constant Bacaro]
                    ]
                ]
                )
            ]
        ]
    ]
]"""

        when:
        def l = FirstOrderLogic.parseLambda(input)

        then:
        l.toString() == canonicalInput

        and: 'already in normal form'
        l.normalizationString == [
                'λx.(λy.(Near(x,y)))(Bacaro)',
```

```
                'λy.(Near(Bacaro,y))',
        ].join('\n')
    }

    def 'FOL "λy.Near(Bacaro,y)" parses as expected'() {

        given:
        def input = 'λy.Near(Bacaro,y)'
        def canonicalInput = 'λy.(Near(Bacaro,y))'

        when:
        def p = FirstOrderLogic.parseTree(input)

        then:
        p.prettyPrint() == """
[S
    [Formula
        [LambdaFormula
            [LambdaAbstraction
                λ
                [Variable y]
                .
                [Formula
                    [AtomicFormula
                        [Predicate Near]
                        (
                        [TermList
                            [Term
                                [Constant Bacaro]
                            ]
                            ,
                            [TermList
                                [Term
                                    [Variable y]
                                ]
                            ]
                        ]
                        )
                    ]
                ]
            ]
        ]
    ]
]"""

        when:
        def l = FirstOrderLogic.parseLambda(input)

        then:
        l.toString() == canonicalInput

        and: 'already in normal form'
        l.normalizationString == canonicalInput
    }

    def 'FOL "Near(Bacaro,Centro)" parses as expected'() {

        given:
        def input = 'Near(Bacaro,Centro)'

        when:
        def p = FirstOrderLogic.parseTree(input)
```

```
        then:
        p.prettyPrint() == """
[S
    [Formula
        [AtomicFormula
            [Predicate Near]
            (
            [TermList
                [Term
                    [Constant Bacaro]
                ]
                ,
                [TermList
                    [Term
                        [Constant Centro]
                    ]
                ]
            ]
            )
        ]
    ]
]"""

        when:
        def l = FirstOrderLogic.parseLambda(input)

        then:
        l.toString() == input

        and: 'already in normal form'
        l.normalizationString == input
    }

    def 'FOL "λP.(λQ.∀x(P(x)⇒Q(x)))(λx.Restaurant(x))" parses as expected'() {

        given:
        def input = 'λP.(λQ.∀x(P(x)⇒Q(x)))(λx.Restaurant(x))'
        def canonicalInput = 'λP.(λQ.(∀x(P(x)⇒Q(x))))(λx.(Restaurant(x)))'

        when:
        def p = FirstOrderLogic.parseTree(input)

        then:
        p.prettyPrint() == """
[S
    [Formula
        [LambdaFormula
            [LambdaApplication
                [LambdaAbstraction
                    λ
                    [AbstractionVariable P]
                    .
                    [Formula
                        [ParentheticalFormula
                            (
                            [Formula
                                [LambdaFormula
                                    [LambdaAbstraction
                                        λ
                                        [AbstractionVariable Q]
                                        .
                                        [Formula
                                            [QuantifiedFormula
```

```
                                                [Quantifier ∀]
                                                [VariableList
                                                    [Variable x]
                                                ]
                                                [Formula
                                                    [ParentheticalFormula
                                                        (
                                                        [Formula
                                                            [LogicFormula
                                                                [Formula
                                                                    [VariableApplicatio
n
                                                                        [AbstractionVar
iable P]
                                                                        (
                                                                        [TermOrFormula
                                                                            [Term
                                                                                [Variab
le x]
                                                                            ]
                                                                        ]
                                                                        )
                                                                    ]
                                                                ]
                                                                [Connective ⇒]
                                                                [Formula
                                                                    [VariableApplicatio
n
                                                                        [AbstractionVar
iable Q]
                                                                        (
                                                                        [TermOrFormula
                                                                            [Term
                                                                                [Variab
le x]
                                                                            ]
                                                                        ]
                                                                        )
                                                                    ]
                                                                ]
                                                            ]
                                                        ]
                                                        )
                                                    ]
                                                ]
                                            ]
                                        ]
                                    ]
                                )
                            ]
                        ]
                    ]
                    (
                    [TermOrFormula
                        [Formula
                            [LambdaFormula
                                [LambdaAbstraction
                                    λ
                                    [Variable x]
                                    .
                                    [Formula
```

```
                                    [AtomicFormula
                                        [Predicate Restaurant]
                                        (
                                        [TermList
                                            [Term
                                                [Variable x]
                                            ]
                                        ]
                                        )
                                    ]
                                ]
                            ]
                        ]
                    ]
                    )
                ]
            ]
        ]
    ]
]"""

        when:
        def l = FirstOrderLogic.parseLambda(input)

        then:
        l.toString() == canonicalInput

        and:
        l.normalizationString == [
                'λP.(λQ.(∀x(P(x)⇒Q(x))))(λx.(Restaurant(x)))',
                'λQ.(∀x(λx.(Restaurant(x))(x)⇒Q(x)))',
                'λQ.(∀x(Restaurant(x)⇒Q(x)))',
        ].join('\n')
    }

    def 'FOL "λQ.∀x(Restaurant(x)⇒Q(x))(λy.∃e(Closed(e)∧ClosedThing(e,y)))" parses as
expected'() {

        given:
        def input = 'λQ.∀x(Restaurant(x)⇒Q(x))(λy.∃e(Closed(e)∧ClosedThing(e,y)))'
        def canonicalInput = 'λQ.(∀x(Restaurant(x)⇒Q(x)))(λy.
(∃e(Closed(e)∧ClosedThing(e,y))))'

        when:
        def p = FirstOrderLogic.parseTree(input)

        then:
        p.prettyPrint() == """
[S
    [Formula
        [LambdaFormula
            [LambdaApplication
                [LambdaAbstraction
                    λ
                    [AbstractionVariable Q]
                    .
                    [Formula
                        [QuantifiedFormula
                            [Quantifier ∀]
                            [VariableList
                                [Variable x]
                            ]
                            [Formula
```


[ParentheticalFormula
(
[Formula
[LogicFormula
[Formula
[AtomicFormula
[Predicate Restaurant]
(
[TermList
[Term
[Variable x]
]
]
)
]
]
[Connective ⇒]
[Formula
[VariableApplication
[AbstractionVariable Q]
(
[TermOrFormula
[Term
[Variable x]
]
]
)
]
]
]
)
]
]
]
]
]
(
[TermOrFormula
[Formula
[LambdaFormula
[LambdaAbstraction
λ
[Variable y]
.
[Formula
[QuantifiedFormula
[Quantifier ∃]
[VariableList
[Variable e]
]
[Formula
[ParentheticalFormula
(
[Formula
[LogicFormula
[Formula
[AtomicFormula
[Predicate Closed]
(
[TermList
[Term
[Variable e]

```
                                                                    ]
                                                                ]
                                                                )
                                                            ]
                                                        ]
                                                        [Connective ∧]
                                                        [Formula
                                                            [AtomicFormula
                                                                [Predicate ClosedThing]
                                                                (
                                                                [TermList
                                                                    [Term
                                                                        [Variable e]
                                                                    ]
                                                                    ,
                                                                    [TermList
                                                                        [Term
                                                                            [Variable
y]

                                                                        ]
                                                                    ]
                                                                ]
                                                                )
                                                            ]
                                                        ]
                                                    ]
                                                )
                                            ]
                                        ]
                                    ]
                                ]
                            ]
                        ]
                        )
                    ]
                ]
    ]
]"""

        when:
        def l = FirstOrderLogic.parseLambda(input)

        then:
        l.toString() == canonicalInput

        and:
        l.normalizationString == [
                'λQ.(∀x(Restaurant(x)⇒Q(x)))(λy.(∃e(Closed(e)∧ClosedThing(e,y))))',
                '∀x(Restaurant(x)⇒λy.(∃e(Closed(e)∧ClosedThing(e,y)))(x))',
                '∀x(Restaurant(x)⇒∃e(Closed(e)∧ClosedThing(e,x)))',
        ].join('\n')
    }

    def 'FOL "λQ.∀x(Restaurant(x)⇒Q(x))(λx.∃e(Closed(e)∧ClosedThing(e,x)))" also parses
as expected'() {

        given:
        def input = 'λQ.∀x(Restaurant(x)⇒Q(x))(λx.∃e(Closed(e)∧ClosedThing(e,x)))'

        expect:
```

```
        FirstOrderLogic.parseLambda(input).normalizationString == [
                'λQ.(∀x(Restaurant(x)⇒Q(x)))(λx.(∃e(Closed(e)∧ClosedThing(e,x))))',
                '∀x(Restaurant(x)⇒λx.(∃e(Closed(e)∧ClosedThing(e,x)))(x))',
                '∀x(Restaurant(x)⇒∃e(Closed(e)∧ClosedThing(e,x)))',
        ].join('\n')
    }

    def 'FOL "λX.X(Maharani)(λx.∃e(Closed(e)∧ClosedThing(e,x)))" parses as expected'()
{

        given:
        def input = 'λX.X(Maharani)(λx.∃e(Closed(e)∧ClosedThing(e,x)))'

        expect:
        FirstOrderLogic.parseLambda(input).normalizationString == [
                'λX.(X(Maharani))(λx.(∃e(Closed(e)∧ClosedThing(e,x))))',
                'λx.(∃e(Closed(e)∧ClosedThing(e,x)))(Maharani)',
                '∃e(Closed(e)∧ClosedThing(e,Maharani))',
        ].join('\n')
    }

    def 'FOL "λM.M(Matthew)(λW.(λz.W(λx.∃e(Opened(e)∧(Opener(e,z)∧Opened(e,x)))))(λP.
(λQ.∃x(P(x)∧Q(x)))(λr.Restaurant(r))))" parses as expected'() {

        given:
        def properNoun = "λM.M(Matthew)"
        def verb = "λW.(λz.W(λx.∃e(Opened(e)∧(Opener(e,z)∧Opened(e,x)))))"
        def det = "λP.(λQ.∃x(P(x)∧Q(x)))"
        def noun = "λr.Restaurant(r)"
        def np = "$det($noun)"
        def vp = "$verb($np)"
        def input = "$properNoun($vp)"
        def input2 = 'λM.M(Matthew)(λW.
(λz.W(λx.∃e(Opened(e)∧(Opener(e,z)∧Opened(e,x)))))(λP.(λQ.∃x(P(x)∧Q(x)))
(λr.Restaurant(r))))'
        def canonicalInput = 'λM.(M(Matthew))(λW.(λz.(W(λx.
(∃e(Opened(e)∧(Opener(e,z)∧Opened(e,x)))))))(λP.(λQ.(∃x(P(x)∧Q(x))))(λr.
(Restaurant(r)))))'

        when:
        def l = FirstOrderLogic.parseLambda(input)

        then:
        input == input2
        l.toString() == canonicalInput
        l.normalizationString == [
                'λM.(M(Matthew))(λW.(λz.(W(λx.
(∃e(Opened(e)∧(Opener(e,z)∧Opened(e,x)))))))(λP.(λQ.(∃x(P(x)∧Q(x))))(λr.
(Restaurant(r)))))',
                'λW.(λz.(W(λx.(∃e(Opened(e)∧(Opener(e,z)∧Opened(e,x)))))))(λP.(λQ.
(∃x(P(x)∧Q(x))))(λr.(Restaurant(r))))(Matthew)',
                'λz.(λP.(λQ.(∃x(P(x)∧Q(x))))(λr.(Restaurant(r)))(λy.
(∃e(Opened(e)∧(Opener(e,z)∧Opened(e,y))))))(Matthew)',
                'λP.(λQ.(∃x(P(x)∧Q(x))))(λr.(Restaurant(r)))(λy.
(∃e(Opened(e)∧(Opener(e,Matthew)∧Opened(e,y)))))',
                'λQ.(∃x(λr.(Restaurant(r))(x)∧Q(x)))(λy.
(∃e(Opened(e)∧(Opener(e,Matthew)∧Opened(e,y)))))',
                '∃x(λr.(Restaurant(r))(x)∧λy.
(∃e(Opened(e)∧(Opener(e,Matthew)∧Opened(e,y))))(x))',
                '∃x(Restaurant(x)∧∃e(Opened(e)∧(Opener(e,Matthew)∧Opened(e,x))))',
        ].join('\n')
    }
```

```
    @Unroll
    def 'FOL "#input" is #expected'() {

        when:
        FirstOrderLogic.parseTree(input)

        then:
        IllegalArgumentException e = thrown()
        e.message.contains(expected as String)

        where:
        input                                                                   || expected
        'foo'                                                                   || 'unparsable'
        '∀x VegetarianRestaurant(x) ⇒ Serves(x, VegetarianFood)'                || 'ambiguous
input (2 parses)'
        'λx.λy.Near(x,y)(Bacaro)'                                               || 'ambiguous
input (2 parses)'
        'λP.λQ.∀x P(x)⇒Q(x)(λx.Restaurant(x))'                                  || 'ambiguous
input (5 parses)'
        'λP.λQ.∀x(P(x)⇒Q(x))(λx.Restaurant(x))'                                 || 'ambiguous
input (2 parses)'
        'λQ.∀x Restaurant(x)⇒Q(x)(λy.∃e Closed(e)∧ClosedThing(e,y))'            || 'ambiguous
input (6 parses)'
        'λQ.∀x(Restaurant(x)⇒Q(x))(λy.∃e Closed(e)∧ClosedThing(e,y))'           || 'ambiguous
input (3 parses)'
        'λx.x(Maharani)(λx.∃e(Closed(e)∧ClosedThing(e,x)))'                     || 'unparsable'
        'λM.M(Matthew)(λW.λz.W(λx.∃e(Opened(e)∧(Opener(e,z)∧Opened(e,x))))(λP.(λQ.
∃x(P(x)∧Q(x)))(λr.Restaurant(r))))'  || 'ambiguous input (2 parses)'
        'λM.M(Matthew)(λW.(λz.W(λx.∃e Opened(e)∧(Opener(e,z)∧Opened(e,x))))(λP.(λQ.
∃x(P(x)∧Q(x)))(λr.Restaurant(r))))' || 'ambiguous input (3 parses)'
        'λM.M(Matthew)(λW.(λz.W(λx.∃e(Opened(e)∧Opener(e,z)∧Opened(e,x))))(λP.(λQ.
∃x(P(x)∧Q(x)))(λr.Restaurant(r))))'  || 'ambiguous input (2 parses)'
        'λM.M(Matthew)(λW.λz.W(λx.∃e(Opened(e)∧Opener(e,z)∧Opened(e,x)))(λP.(λQ.(∃x
P(x)∧Q(x)))(λr.Restaurant(r))))'    || 'ambiguous input (8 parses)'
    }

    static final FOL_DEF = """S -> Formula
Formula -> LambdaFormula
Formula -> QuantifiedFormula
Formula -> LogicFormula
Formula -> AtomicFormula
Formula -> VariableApplication
Formula -> ParentheticalFormula
ParentheticalFormula -> ( Formula )
LambdaFormula -> LambdaAbstraction
LambdaFormula -> LambdaApplication
LambdaAbstraction -> λ Variable . Formula
LambdaAbstraction -> λ AbstractionVariable . Formula
LambdaApplication -> LambdaAbstraction ( TermOrFormula )
QuantifiedFormula -> Quantifier VariableList Formula
LogicFormula -> Formula Connective Formula
LogicFormula -> ¬ Formula
AtomicFormula -> Predicate ( TermList )
VariableApplication -> AbstractionVariable ( TermOrFormula )
TermOrFormula -> Term
TermOrFormula -> Formula
VariableList -> Variable
VariableList -> Variable , VariableList
TermList -> Term
TermList -> Term , TermList
Term -> Function ( TermList )
Term -> Constant
Term -> Variable
```

```
Connective -> ∧
Connective -> ∨
Connective -> ⇒
Quantifier -> ∀
Quantifier -> ∃
Constant -> VegetarianFood
Constant -> Maharani
Constant -> AyCaramba
Constant -> Bacaro
Constant -> Centro
Constant -> Leaf
Constant -> Speaker
Constant -> TurkeySandwich
Constant -> Desk
Constant -> Lunch
Constant -> FiveDollars
Constant -> LotOfTime
Constant -> Monday
Constant -> Tuesday
Constant -> Wednesday
Constant -> Thursday
Constant -> Friday
Constant -> Saturday
Constant -> Sunday
Constant -> Yesterday
Constant -> Today
Constant -> Tomorrow
Constant -> Now
Constant -> NewYork
Constant -> Boston
Constant -> SanFrancisco
Constant -> Matthew
Constant -> Franco
Constant -> Frasca
Variable -> a
Variable -> b
Variable -> c
Variable -> d
Variable -> e
Variable -> f
Variable -> g
Variable -> h
Variable -> i
Variable -> j
Variable -> k
Variable -> l
Variable -> m
Variable -> n
Variable -> o
Variable -> p
Variable -> q
Variable -> r
Variable -> s
Variable -> t
Variable -> u
Variable -> v
Variable -> w
Variable -> x
Variable -> y
Variable -> z
AbstractionVariable -> A
AbstractionVariable -> B
AbstractionVariable -> C
```

```
AbstractionVariable -> D
AbstractionVariable -> E
AbstractionVariable -> F
AbstractionVariable -> G
AbstractionVariable -> H
AbstractionVariable -> I
AbstractionVariable -> J
AbstractionVariable -> K
AbstractionVariable -> L
AbstractionVariable -> M
AbstractionVariable -> N
AbstractionVariable -> O
AbstractionVariable -> P
AbstractionVariable -> Q
AbstractionVariable -> R
AbstractionVariable -> T
AbstractionVariable -> U
AbstractionVariable -> V
AbstractionVariable -> W
AbstractionVariable -> X
AbstractionVariable -> Y
AbstractionVariable -> Z
Predicate -> Serves
Predicate -> Near
Predicate -> Restaurant
Predicate -> Have
Predicate -> VegetarianRestaurant
Predicate -> Eating
Predicate -> Time
Predicate -> Eater
Predicate -> Eaten
Predicate -> Meal
Predicate -> Location
Predicate -> Arriving
Predicate -> Arriver
Predicate -> Destination
Predicate -> EndPoint
Predicate -> Precedes
Predicate -> Closed
Predicate -> ClosedThing
Predicate -> Opened
Predicate -> Opener
Predicate -> Menu
Predicate -> Having
Predicate -> Haver
Predicate -> Had
Function -> LocationOf
Function -> CuisineOf
Function -> IntervalOf
Function -> MemberOf"""
}
```

```groovy
import spock.lang.Specification


/**
 * Test specification of SemanticGrammar.
 */
class SemanticGrammarSpec extends Specification {

    def 'semantics of "Maharani closed"'() {

        given:
        def input = 'Maharani closed'

        expect:
        SemanticGrammar.parseSemantics(input).toString() ==
'∃e(Closed(e)∧ClosedThing(e,Maharani))'
        SemanticGrammar.parseSemanticsDerivation(input).normalizationString == [
                'λM.(M(Maharani))(λx.(∃e(Closed(e)∧ClosedThing(e,x))))',
                'λx.(∃e(Closed(e)∧ClosedThing(e,x)))(Maharani)',
                '∃e(Closed(e)∧ClosedThing(e,Maharani))',
        ].join('\n')
    }

    def 'semantics of "every restaurant closed"'() {

        given:
        def input = 'every restaurant closed'
        def det = 'λP.(λQ.(∀x(P(x)⇒Q(x))))'
        def noun = 'λr.(Restaurant(r))'
        def verb = 'λx.(∃e(Closed(e)∧ClosedThing(e,x)))'

        expect:
        SemanticGrammar.parseSemantics(input).toString() ==
'∀x(Restaurant(x)⇒∃e(Closed(e)∧ClosedThing(e,x)))'
        SemanticGrammar.parseSemanticsDerivation(input).normalizationString == [
                "$det($noun)($verb)",
                'λQ.(∀x(λr.(Restaurant(r))(x)⇒Q(x)))(λx.
(∃e(Closed(e)∧ClosedThing(e,x))))',
                '∀x(λr.(Restaurant(r))(x)⇒λx.(∃e(Closed(e)∧ClosedThing(e,x)))(x))',
                '∀x(Restaurant(x)⇒∃e(Closed(e)∧ClosedThing(e,x)))',
        ].join('\n')
    }

    def 'semantics of "Matthew opened a restaurant"'() {

        given:
        def input = 'Matthew opened a restaurant'
        def properNoun = "λM.(M(Matthew))"
        def verb = "λW.(λz.(W(λx.(∃e(Opened(e)∧(Opener(e,z)∧Opened(e,x)))))))"
        def det = "λP.(λQ.(∃x(P(x)∧Q(x))))"
        def noun = "λr.(Restaurant(r))"
        def np = "$det($noun)"
        def vp = "$verb($np)"
        def expected = "$properNoun($vp)"
        def canonicalExpected = 'λM.(M(Matthew))(λW.(λz.(W(λx.
(∃e(Opened(e)∧(Opener(e,z)∧Opened(e,x)))))))(λP.(λQ.(∃x(P(x)∧Q(x))))(λr.
(Restaurant(r)))))'

        expect:
        canonicalExpected == expected
```

```
        SemanticGrammar.parseSemantics(input).toString() ==
'∃x(Restaurant(x)∧∃e(Opened(e)∧(Opener(e,Matthew)∧Opened(e,x))))'
        SemanticGrammar.parseSemanticsDerivation(input).normalizationString == [
                expected,
                'λW.(λz.(W(λx.(∃e(Opened(e)∧(Opener(e,z)∧Opened(e,x)))))))(λP.(λQ.
(∃x(P(x)∧Q(x))))(λr.(Restaurant(r))))(Matthew)',
                'λz.(λP.(λQ.(∃x(P(x)∧Q(x))))(λr.(Restaurant(r)))(λy.
(∃e(Opened(e)∧(Opener(e,z)∧Opened(e,y))))))(Matthew)',
                'λP.(λQ.(∃x(P(x)∧Q(x))))(λr.(Restaurant(r)))(λy.
(∃e(Opened(e)∧(Opener(e,Matthew)∧Opened(e,y)))))',
                'λQ.(∃x(λr.(Restaurant(r))(x)∧Q(x)))(λy.
(∃e(Opened(e)∧(Opener(e,Matthew)∧Opened(e,y)))))',
                '∃x(λr.(Restaurant(r))(x)∧λy.
(∃e(Opened(e)∧(Opener(e,Matthew)∧Opened(e,y))))(x))',
                '∃x(Restaurant(x)∧∃e(Opened(e)∧(Opener(e,Matthew)∧Opened(e,x))))',
        ].join('\n')
    }
}
```

LambdaSpec.groovy

```groovy
package fol.lambda

import spock.lang.Specification

/**
 * Test specification of Lambda Calculus for First Order Logic.
 */
class LambdaSpec extends Specification {

    def 'basic expression containing abstraction'() {

        given:
        def exp = new TermList([
                new Abstraction(
                        boundVar: new Variable('y'),
                        expr: new TermList(['Near', '(', 'Bacaro', ',', new
Variable('y'), ')'])
                )
        ])

        expect:
        exp[0].boundVar == new Variable('y')
        exp[0].expr[4] == new Variable('y')

        and:
        exp.toString() == 'λy.(Near(Bacaro,y))'
    }

    def 'basic reduction'() {

        given:
        def app = new Application(
                abstractionTerm: new Abstraction(
                        boundVar: new Variable('y'),
                        expr: new TermList(['Near', '(', 'Bacaro', ',', new
Variable('y'), ')'])
                ),
                term: new Symbol('Centro')
        )

        expect:
        app.abstractionTerm.boundVar == new Variable('y')
        app.abstractionTerm.expr[4] == new Variable('y')
        app.term.symbol == 'Centro'

        and:
        app.toString() == 'λy.(Near(Bacaro,y))(Centro)'
        app.reduction().toString() == 'Near(Bacaro,Centro)'
        app.reduction() == new TermList(['Near', '(', 'Bacaro', ',', 'Centro', ')'])

        and: 'normalization'
        app.normalizationString == [
                'λy.(Near(Bacaro,y))(Centro)',
                'Near(Bacaro,Centro)'
        ].join('\n')
    }

    def 'basic alpha-conversion'() {

        given:
```

```
        def app = new Application(
                abstractionTerm: new Abstraction(
                        boundVar: new Variable('x'),
                        expr: new TermList([
                                new Abstraction(
                                        boundVar: new Variable('y'),
                                        expr: new TermList([new Variable('x')])
                                )
                        ])
                ),
                term: new Variable('y')
        )

        expect: 'substitution into the abstraction triggers alpha-conversion of the
bound variable'
        app.toString() == 'λx.(λy.(x))(y)'
        app.reduction().toString() == 'λz.(y)'
        app.reduction() == new TermList([
                new Abstraction(
                        boundVar: new Variable('z'),
                        expr: new TermList([new Variable('y')])
                )
        ])

        and: 'normalization'
        app.normalizationString == [
                'λx.(λy.(x))(y)',
                'λz.(y)'
        ].join('\n')
    }

    def 'variable application reduction'() {

        given:
        def app = new Application(
                abstractionTerm: new Abstraction(
                        boundVar: new Variable('P'),
                        expr: new TermList([new Abstraction(
                                boundVar: new Variable('Q'),
                                expr: new TermList([
                                        '∀',
                                        new Variable('x'),
                                        new VariableApplication(new Variable('P'), new
Variable('x')),
                                        '⇒',
                                        new VariableApplication(new Variable('Q'), new
Variable('x')),
                                ])
                        )])
                ),
                term: new Abstraction(
                        boundVar: new Variable('x'),
                        expr: new TermList(['Restaurant', '(', new Variable('x'), ')'])
                )
        )

        expect:
        app.toString() == 'λP.(λQ.(∀x P(x)⇒Q(x)))(λx.(Restaurant(x)))'
        app.reduction().toString() == 'λQ.(∀xλx.(Restaurant(x))(x)⇒Q(x))'
        app.reduction() == new TermList([new Abstraction(
                boundVar: new Variable('Q'),
                expr: new TermList([
                        '∀',
```

```
                        new Variable('x'),
                        new Application(
                                abstractionTerm: new Abstraction(
                                        boundVar: new Variable('x'),    // term's bound
vars don't need alpha-conversion
                                        expr: new TermList(['Restaurant', '(', new
Variable('x'), ')'])
                                ),
                                term: new Variable('x')
                        ),
                        '⇒',
                        new VariableApplication(new Variable('Q'), new Variable('x')),
                ])
        )])

        and: 'second level reduction'
        app.reduction().reduction().toString() == 'λQ.(∀x Restaurant(x)⇒Q(x))'
        app.reduction().reduction() == new TermList([new Abstraction(
                boundVar: new Variable('Q'),
                expr: new TermList([
                        '∀',
                        new Variable('x'),
                        'Restaurant',
                        '(',
                        new Variable('x'),
                        ')',
                        '⇒',
                        new VariableApplication(new Variable('Q'), new Variable('x')),
                ])
        )])

        and: 'normalization'
        app.normalizationString == [
                'λP.(λQ.(∀x P(x)⇒Q(x)))(λx.(Restaurant(x)))',
                'λQ.(∀xλx.(Restaurant(x))(x)⇒Q(x))',
                'λQ.(∀x Restaurant(x)⇒Q(x))'
        ].join('\n')
    }

    def 'finishing "every restaurant closed"'() {

        given: 'compact Variable references (since separate instances are tested in
specs above)'
        def x = new Variable('x')
        def e = new Variable('e')
        def Q = new Variable('Q')

        and:
        def everyRestaurant =  new Abstraction(
                boundVar: Q,
                expr: new TermList(['∀', x, 'Restaurant', '(', x, ')', '⇒', new
VariableApplication(Q, x)])
        )
        def closed = new Abstraction(
                boundVar: x,
                expr: new TermList([ '∃', e, 'Closed', '(', e, ')', '∧', 'ClosedThing',
'(', e, ',', x, ')'])
        )
        def app = new Application( abstractionTerm: everyRestaurant, term: closed)

        expect:
        everyRestaurant.toString() == 'λQ.(∀x Restaurant(x)⇒Q(x))'
        closed.toString() == 'λx.(∃e Closed(e)∧ClosedThing(e,x))'
```

```
        app.toString() == 'λQ.(∀x Restaurant(x)⇒Q(x))(λx.(∃e
Closed(e)∧ClosedThing(e,x)))'

        and: 'first level reduction'
        app.reduction().toString() == '∀x Restaurant(x)⇒λx.(∃e
Closed(e)∧ClosedThing(e,x))(x)'
        app.reduction() == new TermList(['∀', x, 'Restaurant', '(', x, ')', '⇒',
                new Application(abstractionTerm: closed, term: x)])

        and: 'second level reduction'
        app.reduction().reduction().toString() == '∀x Restaurant(x)⇒∃e
Closed(e)∧ClosedThing(e,x)'
        app.reduction().reduction() == new TermList(['∀', x, 'Restaurant', '(', x, ')',
'⇒', closed.expr].flatten())

        and: 'normalization'
        app.normalizationString == [
                'λQ.(∀x Restaurant(x)⇒Q(x))(λx.(∃e Closed(e)∧ClosedThing(e,x)))',
                '∀x Restaurant(x)⇒λx.(∃e Closed(e)∧ClosedThing(e,x))(x)',
                '∀x Restaurant(x)⇒∃e Closed(e)∧ClosedThing(e,x)'
        ].join('\n')
    }

    def 'Maharani closed'() {

        given: 'compact Variable references (since separate instances are tested in
specs above)'
        def x = new Variable('x')
        def e = new Variable('e')

        and:
        def maharani =  new Abstraction(
                boundVar: x,
                expr: new TermList([new VariableApplication(x, new
Symbol('Maharani'))])
        )
        def closed = new Abstraction(
                boundVar: x,
                expr: new TermList([ '∃', e, 'Closed', '(', e, ')', '∧', 'ClosedThing',
'(', e, ',', x, ')'])
        )
        def app = new Application( abstractionTerm: maharani, term: closed)

        expect:
        maharani.toString() == 'λx.(x(Maharani))'
        closed.toString() == 'λx.(∃e Closed(e)∧ClosedThing(e,x))'
        app.toString() == 'λx.(x(Maharani))(λx.(∃e Closed(e)∧ClosedThing(e,x)))'

        and: 'first level reduction'
        app.reduction().toString() == 'λx.(∃e Closed(e)∧ClosedThing(e,x))(Maharani)'
        app.reduction() == new TermList([new Application(abstractionTerm: closed, term:
new Symbol('Maharani'))])

        and: 'second level reduction'
        app.reduction().reduction().toString() == '∃e
Closed(e)∧ClosedThing(e,Maharani)'
        app.reduction().reduction() == new TermList([
                '∃', e, 'Closed', '(', e, ')', '∧', 'ClosedThing', '(', e, ',',
'Maharani', ')'])

        and: 'normalization'
        app.normalizationString == [
                'λx.(x(Maharani))(λx.(∃e Closed(e)∧ClosedThing(e,x)))',
```

```
                '\u03bbx.(\u2203e Closed(e)\u2227ClosedThing(e,x))(Maharani)',
                '\u2203e Closed(e)\u2227ClosedThing(e,Maharani)'
        ].join('\n')
    }

    def "groovy sublist indexes"() {

        given:
        def x = ['a', 'b', 'c']

        expect:
        x[2..-1] + x[0..1] == ['c', 'a', 'b']
        x.subList(0, 0) == []


        when:
        x[3..-1]

        then:
        thrown(IndexOutOfBoundsException)
    }
}
```