30 The Earley Parser: Dynamic Programming in Java

Chapter Objectives

Sentence parsing using dynamic programming
- Memoization of subparses
- Retaining partial solutions (parses) for reuse

The chart as medium for storage and reuse
- Indexes for word list (sentence)
- States reflect components of parse
- Dot reflects extent of parsing right hand side of grammar rule

Lists of states make up components of chart
- Chart linked to word list

Java Implementation of an Earley parser
- Context free parser
- Deterministic

Chart supports multiple parse trees
Forward development of chart composes components of successful parse
Backward search of chart produces possible parses of the sentence

Chapter Contents

30.1 Chart Parsing: An Introduction
30.2 The Earley Parser: Components
30.3 The Earley Parser: Java Code
30.4 The Completed Parser
30.5 Generating Parse Trees from Charts and Grammar Rules (Advanced Section)

30.1 Chart Parsing: An Introduction

The Earley parser (Earley 1970) uses dynamic programming to analyze strings of words. Traditional dynamic programming techniques (Luger 2009, Section 4.1) use an array to save (memoize) the partial solutions of a problem for use in the generation of subsequent partial solutions. In Earley parsing this array is called a *chart*, and thus this approach to parsing sentences is often called *chart parsing*.

In Chapter 9, Sections 1 and 2, we first presented the full algorithms along with the evolving chart for Earley parsing. In these sections, we presented pseudo-code, demonstrated the "dot" as a pointer indicating the current state of the parse for each grammar rule, and explicitly considered the state of the chart after each step of the algorithm. We refer to Sections 9.1, 9.2, and Luger (2009, Section 15.2.2) for these specific details if there is any concern about how the chart-parsing algorithm works. We feel that it is also interesting to compare the data representation and control structures used in the declarative Prolog environment, Chapter 9, with what we next present with the object-oriented representations of Java.

30.2 The Earley Parser: Components

We first discuss the data representations required by the Earley parsing algorithm presented in Sections 9.1 and 9.2. Of course, in the present chapter we will be using an object-oriented hierarchy to capture the components of the parser. Consider what the pseudo-code requires:

- **A sentence**. The sentence needs to be in a format that supports pointers to any word located in that sentence so that appropriate grammar rules can be applied.
- **A grammar**. The Earley parser needs a set of (context free) grammar rules that can be applied to interpret the components of the sentence. The parser itself has no knowledge of the parts of speech (POS) or production rules of the grammar.
- **An evolving chart**. The chart is used to save partial solutions (accepted parts of the parse) for later use. Thus, the chart is used to contain the states as they are produced during the algorithm. These states need to be stored in the order of their production and without repeats.
- **The states**. State will capture the current activity of the parser. Thus it will need to be a container for the current rule, which has a left-hand-side, `LHS`, and a right-hand-side, `RHS`. Besides instantiating a particular rule, state must also have the current `(i, j)` pair that presents the seen/unseen parts of the sentence for that rule.

We next consider how each of these constituents of the algorithm is represented as data structures in Java. In Section 30.3 we describe the Earley parser itself that will utilize these components.

The Sentence

First, we consider the set of descriptors that the sentence needs. The primary thing we require is the ability to index into specific word locations in the current sentence. This can be handled two ways: the use of a simple representation, or the use of a class. The simple representation would be a `String` array. This array would enable us easily to index to specific words in the sentence. Everything that we need for the algorithm is present.

We could also use a class to represent each indexed sentence. If `Sentence` is a class, we could incorporate other aspects of the sentence into that class, such as the segmentation of a `String` into individual words. If we were using the Earley algorithm in conjunction with another algorithm (which is often required), we may need to create the `Sentence` class so that we can separate the sentence's parsing from other code.

For this presentation, we use the simple representation of a sentence as a `String` array.

The Grammar

For the grammar rule processing required by the Earley parser we create a class. The application of each rule needs to know the specific rules of a grammar, and which non-terminals are parts of speech. So that both characteristics are easily contained, a `Grammar` class is a good choice. The `Grammar` class will need two important methods: `getRHS(String lhs)`, and `isPOS(String lhs)`:

`getRHS(String lhs)` will return all `RHS`'s of the grammar rules for any left-hand-side, `lhs`. If there are not any such rules, then it will indicate such.

`isPOS(String lhs)` will return `true` or `false` based on whether or not a component of the lhs is a part of speech.

To make the `Grammar` class easier to extend to more complicated grammars, the `Grammar` class itself does not instantiate any rules. It is a basic framework for the two important methods, and defines how the rules are contained and related. To create a specific grammar, the `Grammar` class needs to be extended to a new class. This new class will instantiate all the grammar rules. As just noted, the framework of the grammar rules is a mapping between a LHS and RHS. For each rule there will be only one LHS for each RHS, but it is likely in the full set of grammar rules that a particular LHS will have several possible RHSs. Later in the chapter, the exact framework for this matching is presented.

The Chart

A chart is an ordered list of the successively produced components of the parse. A major requirement is to determine whether any newly produced possible component of the parse is already contained in the chart.

To make it easier to maintain the charts correctly and consistently, we create a `Chart` class. We could have used a simpler structure, like `Vector,` to contain the states of the parse as they are produced, but the code to manipulate the `Vector` would then be distributed throughout the parser code. This dispersed code makes it much harder to make corrections, and to debug. When we create the `Chart` class, the code to manipulate the chart will be identical for all uses, and since the code is all in one place, it will be much easier to debug. Notice that the `Chart` class represents a single chart, not the evolving set of states that are created by the Earley algorithm. Since there is no additional functionality needed beside that already discussed, we make a `Chart` array for the evolving set of chart states, rather than making another class.

The States

A state component for the parser has one left-hand-side, LHS, one right-hand-side, RHS, for each rule that is instantiated, as well as indices from the sentence `String` array, an `(i j)` pair. Because these components all need to be represented, the easiest way to create the problem solving state, is to make a `State` class. Since the `State` class supports the full Earley algorithm, it will require `get` methods for returning the `LHS`, the `RHS`, and the `i j` indices. Also, as seen in the pseudo-code of Section 9.2, we need the ability to get the term after the dot in the `RHS`, as well as the ability to determine whether or not the dot is in the last (terminal) position. Methods to support these requirements must be provided.

Throughout our discussion, `LHS` and `RHS` have been mentioned, but not their implementation. Since the Earley parser uses context-free grammar rules, we create the `LHS` as a `String`. The `RHS` on the other hand, is a sequence of terms, which may or may not include a dot. Due to the fact that it is used in two separate classes, and the additional requirement of dot manipulation, we separated the RHS into its own class.

30.3 The Earley Parser: Java Code

The Earley parser, which manipulates the components described in Section 30.2, will have its own class. This makes it easier to contain and hide the details of the algorithm. The `EarleyParser` class can be implemented in either of two ways.

First, the class could be static. When one wanted to parse a sentence, the static method would be called, and with the grammar rules and the sentence to be parsed as arguments, return a boolean indicating whether the parse was successful. Alternatively, the chart itself could be returned and examined to determine whether there was a successful parse. Second, with the class not static, the `EarleyParser` would be instantiated with a `Grammar`, and then a parse method could be called with a sentence. After the parse method is called, another method would be called to obtain the charts (if the parse method returns a `boolean`). We take this second approach and start with the most basic class, the `RHS` class, and then work our way towards creating the `EarleyParser` class.

The RHS Class

The `RHS` is a `String` array containing a `boolean` that records whether its terms contains a dot, an `int` recording the offset of the dot (this will default to –1, indicating no dot), and a `final static String` containing the representation of the dot. The constructor determines if there is a dot and updates `hasDot` and `dot` accordingly.

```
public class RHS
{
    private String[] terms;
    private boolean hasDot = false;
    private int dot = -1;
    private final static String DOT = "@";
    public RHS (String[] t)
    {
        terms = t;
        for (int i=0; i< terms.length; i++)
        {
            if(terms[i].compareTo (DOT) == 0)
            {
                dot = i;
                hasDot = true;
                break;
            }
        }
    }
                        // Additional methods defined below.
}
```

RHS returns its terms, the String array, for use by the EarleyParser, as well as the String prior to and after the dot. This enables ease of queries by the EarleyParser regarding the terms in the RHS of the grammar rule. For example,. EarleyParser gets the term following the dot from RHS, and queries the Grammar to determine if that term is a part of speech.

```
public String[] getTerms ()
{
    return terms;
}
public String getPriorToDot ()
{
    if(hasDot && dot >0)
        return terms[dot-1];
    return "";
}
public String getAfterDot ()
{
    if(hasDot && dot < terms.length-1)
        return terms[dot+1];
    return "";
}
```

The final procedures required to implement RHS are manipulation of and queries concerning the dot. The queries determine whether there is a dot, and where the dot is located, last or first. When a dot is moved or added to a RHS, a new RHS is returned. This is done because whenever a dot is moved a new State must be created for it.

```
public boolean hasDot ()
{
    return hasDot;
}
public boolean isDotLast ()
{
    if(hasDot)
        return (dot==terms.length-1);
        return false;
}
public boolean isDotFirst ()
{
    if(hasDot)
        return (dot==0);
    return false;
}
```

```
public RHS addDot ()
{
    String[] t = new String[terms.length+1];
    t[0] = DOT;
    for (int i=1; i< t.length; i++)
        t[i] = terms[i-1];
    return new RHS (t);
}
public RHS addDotLast ()
{
    String[] t = new String[terms.length+1];
    for (int i=0; i< t.length-1; i++)
        t[i] = terms[i];
    t[t.length-1] = DOT;
    return new RHS (t);
}
public RHS moveDot ()
{
    String[] t = new String[terms.length];
    for (int i=0; i< t.length; i++)
    {
        if (terms[i].compareTo (DOT)==0)
         {
             t[i] = terms[i+1];
             t[i+1] = DOT;
             i++;
         }
         else
             t[i] = terms[i];
    }
    return new RHS (t);
}
```

There are two additional methods that we have not included here. These are overrides methods of `equals(Object o)`, and `toString()`. Equivalence indicates identical terms, and placement of the dot. `toString()` is overridden to make it easier to print during debug, and when the charts are printed. Next we present one of the two classes that contain a `RHS`.

The Grammar Class

The `Grammar` class does not instantiate the rules of a specific grammar. It contains a `HashMap` that links the left-hand-side (`LHS`) of a grammar rule, which is a `String`, to an array of `RHS`s, and a `Vector` of `String`s that are the parts of speech of the grammar.

```
public class Grammar
{
    HashMap<String, RHS[]> Rules;
    Vector<String> POS;
    public Grammar ()
    {
        Rules = new HashMap<String, RHS[]>();
        POS = new Vector<String>();
    }
                            // Additional methods defined below.
}
```

The `Grammar` class supports two methods: one returning all the `RHS`s associated with a LHS, and the second returning if a `String` is a part of speech.

```
public RHS[]  getRHS (String lhs)
{
    RHS[] rhs = null;
    if(Rules.containsKey (lhs))
    {
        rhs = Rules.get (lhs);
    }
    return rhs;
}
public boolean isPartOfSpeech (String s)
{
    return POS.contains (s);
}
```

For `EarleyParser` to function, the `Grammar` class must be extended. To do this we have created `SimpleGrammar` that demonstrates both creation of the rules and how these are added to the rule list.

```
public class SimpleGrammar extends Grammar
{
    public SimpleGrammar ()
    {
        super();
        initialize();
    }
    private void initialize()
    {
        initRules();
        initPOS();
    }
```

```
        private void initRules()
        {
            String[] s1 = {"NP", "VP"};
            RHS[] sRHS = {new RHS(s1)};
            Rules.put ("S", sRHS);
            String[] np1 = {"NP","PP"};
            String[] np2 = {"Noun"};
            RHS[] npRHS = {new RHS(np1),
                   new RHS(np2)};
            Rules.put ("NP", npRHS);
            String[] vp1 = {"Verb","NP"};
            String[] vp2 = {"VP", "PP"};
            RHS[] vpRHS = {new RHS(vp1),
                   new RHS(vp2)};
            Rules.put ("VP", vpRHS);
            String[] pp1 = {"Prep","NP"};
            RHS[] ppRHS = {new RHS(pp1)};
            Rules.put ("PP", ppRHS);
            String[] noun1 = {"John"};
            String[] noun2 = {"Mary"};
            String[] noun3 = {"Denver"};
            RHS[] nounRHS = {new RHS(noun1),
                     new RHS(noun2),
                     new RHS(noun3)};
            Rules.put ("Noun", nounRHS);
            String[] verb = {"called"};
            RHS[] verbRHS = {new RHS(verb)};
            Rules.put ("Verb", verbRHS);
            String[] prep = {"from"};
            RHS[] prepRHS = {new RHS(prep)};
            Rules.put ("Prep", prepRHS);
        }
        private void initPOS()
        {
            POS.add ("Noun");
            POS.add ("Verb");
            POS.add ("Prep");
        }
    }
```

The State class The `State` class contains a `String` representing the `LHS` of the rule, a `RHS` that contains the dotted right-hand-side of the rule, and `int`s describing Seen/UnSeen components. There are `get` methods, and functions for handling the dot.

```
public class State
{
    private String lhs;
    private RHS rhs;
    private int i,j;
    public State (String lhs, RHS rhs, int i, int j)
    {
         this.lhs = lhs;
         this.rhs = rhs;
         this.i = i;
         this.j = j;
    }
    public String getLHS ()
    {
         return lhs;
    }
    public RHS getRHS ()
    {
         return rhs;
    }
    public int getI ()
    {
         return i;
    }
    public int getJ ()
    {
         return j;
    }
    public String getPriorToDot ()
    {
         return rhs.getPriorToDot ();
    }
    public String getAfterDot ()
    {
         return rhs.getAfterDot ();
    }
    public boolean isDotLast ()
    {
         return rhs.isDotLast ();
    }
}
```

Finally, again, the function overrides of `equals(Object o)` and `toString()` are not included. Equivalent states are identified when the LHS, RHS, `i`, and `j` are all identical. `toString()` prints out the `State` in a readable format.

The Chart Class The `Chart` class contains a `Vector` of `States`. These are the states produced by the `EarleyParser`. The `States` are inserted into the `Vector` in order; this is necessary for the algorithm.

```
public class Chart
{
    Vector<State> chart;
    public Chart ()
    {
        chart = new Vector<State>();
    }
    public void addState (State s)
    {
        if(!chart.contains (s))
        {
            chart.add (s);
        }
    }
    public State getState (int i)
    {
        if(i < 0 || i >= chart.size ())
            return null;
        return (State)chart.get (i);
    }
}
```

`addState(State s)` determines whether `s` is already within the `Chart`. If `s` is not in the `Chart`, `s` is added to the end of `Vector`. Nothing is done when `s` is already in the `Chart`. `getState(int i)` returns the `State` at the `i`-th offset in `Vector`. There are checks to enforce that `i` is a valid offset. `toString()` is overridden in `Chart`, in addition to a `get` function that returns the size of the `Chart`.

30.4 The Completed Parser

We have now completed the design of the components of the Earley parser. Figure 30.1 presents the object hierarchy that supports this design. `EarleyParser`, which implements this design is presented in Section 30.4.1, while Section 30.4.2 describes `main` which presents two sentences and produces their chart parses.

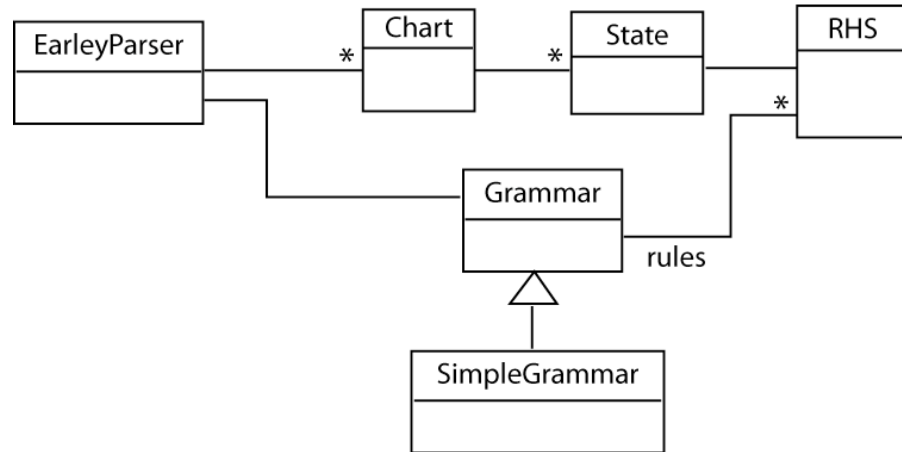

Figure 30.1 The design hierarchy for the EarleyParser class.

The EarleyParser Class

The `EarleyParser` class contains `Grammar` describing the grammar rules for parsing the perspective sentence. It also creates `String`, an array containing the sentence to be parsed, and `Chart`, an array containing the evolving states of the chart. `sentence` and `Chart` will change with each call of `parseSentence(…)`. Note that each of the methods of the EarleyParser class reflect the design components of Sections 30.2 and 30.3 and presented in Figure 30.1.

```
public class EarleyParser
{
    private Grammar grammar;
    private String[] sentence;
    private Chart[] charts;
    public EarleyParser (Grammar g)
    {
        grammar = g;
    }
    public Grammar getGrammar ()
    {
        return grammar;
    }
    public Chart[] getCharts ()
    {
        return charts;
    }
                // Additional methods defined below.
}
```

`parseSentence(…)` takes the sentence to be parsed, uses it to initialize `Chart` to have the number of words in the sentence + 1 number of chart states, and makes the dummy start state `("$ → @ S", 0, 0)` the first chart state. `parseSentence` then iterates through each of

the charts, and for every `State` in a `Chart`, checks to determine which procedure is called next: `completer(...)` - the dot is last, `scanner(...)` - the term following the dot is a part of speech, or `predictor(...)` - the term following the dot is not a part of speech. After all charts are visited, if the last `State` added to the last `Chart` is a finish state, `("$ → S @", 0, sentenceLength)`, the sentence was successfully parsed.

```
public boolean parseSentence (String[] s)
{
    sentence = s;
    charts = new Chart[sentence.length+1];
    for (int i=0; i< charts.length; i++)
        charts[i] = new Chart ();
    String[] start1 = {"@", "S"};
    RHS startRHS = new RHS (start1);
    State start = new State ("$",startRHS,0,0,null);
    charts[0].addState (start);
    for (int i=0; i<charts.length; i++)
    {
        for (int j=0; j<charts[i].size (); j++)
        {
            State st = charts[i].getState (j);
            String next_term = st.getAfterDot ();
            if (st.isDotLast ())
                                // State's RHS = ... @
                completer (st);
            else
            if(grammar.isPartOfSpeech (next_term))
                    // RHS = ... @ A ..., where A is a part of speech.
                scanner (st);
            else
                predictor (st); // A is NOT a part of speech.
        }
    }
            // Determine whether a successful parse.
    String[] fin = {"S","@"};
    RHS finRHS = new RHS (fin);
    State finish = new State ("$",finRHS,
        0,sentence.length,null);
    State last = charts[sentence.length].getState
        (charts[sentence.length].size ()-1);
    return finish.equals (last);
}
```

We next create the `predictor`, `scanner`, and `completer` procedures. First, the `predictor(State s)` adds all rules for the term after the dot in `s` to the `j`-th slot in `chart`.

```
private void predictor (State s)
{
    String lhs = s.getAfterDot ();
    RHS[] rhs = grammar.getRHS (lhs);
    int j = s.getJ ();
    for (int i=0; i< rhs.length; i++)
    {
         State ns = new State (lhs, rhs[i].addDot (),
              j, j, s);
        charts[j].addState (ns);
    }
}
```

`scanner(State s)` determines whether the part of speech term following the dot in `s` has a `RHS` that contains only the `j`-th word in the sentence. If so, this new state is added to the `(j+1)`-th chart.

```
private void scanner (State s)
{
    String lhs = s.getAfterDot ();
    RHS[] rhs = grammar.getRHS (lhs);
    int i = s.getI ();
    int j = s.getJ ();
    for (int a=0; a< rhs.length; a++)
    {
        String[] terms = rhs[a].getTerms ();
        if (terms.length == 1 &&
             j < sentence.length &&
             terms[0].compareToIgnoreCase
              (sentence[j]) == 0)
         {
             State ns = new State (lhs,
                  rhs[a].addDotLast (), j, j+1, s);
             charts[j+1].addState (ns);
        }
    }
}
```

Finally, `completer(State s)` determines whether any state in the `i`-th chart slot has a term following the dot that is the same as the `LHS` of `s`. If so, new `States` based on those terms found are created with a moved dot, and an updated `j`.

```
private void completer (State s)
{
    String lhs = s.getLHS ();
    for (int a=0; a<charts[s.getI ()].size (); a++)
    {
         State st = charts[s.getI ()].getState (a);
         String after = st.getAfterDot ();
         if(after != null &&
              lhs.compareTo (after)==0)
         {
              State ns = new State (st.getLHS (),
                   st.getRHS ().moveDot (),
                   st.getI (), s.getJ (), s);
              charts[s.getJ ()].addState (ns);
         }
     }
 }
```

After `EarleyParser` completes `parseSentence(…)`, the `getCharts()` method is called. These charts in conjunction with the grammar rules can be used to determine the parse trees of the sentence. We discuss methods for doing this in Section 30.5.

The EarleyParser: A Test Run

We next create methods that contain sentences that test `EarleyParser`. In our example, the included `Main`, contains two sentences and then uses `SimpleGrammar` to parse them. It then prints out the sentences and the associated `Charts` for each.

```
public class Main
{
    public static void main (String[] args)
    {
         String[] sentence1 =
              {"John", "called", "Mary"};
         String[] sentence2 =
              {"John", "called", "Mary",
              "from", "Denver"};
         Grammar grammar = new SimpleGrammar ();
         EarleyParser parser =
              new EarleyParser (grammar);
         test (sentence1,parser);
         test (sentence2,parser);
    }
```

```
    static void test (String[] sent,
         EarleyParser parser)
    {
         StringBuffer out = new StringBuffer ();
         for (int i=0; i<sent.length-1;i++)
              out.append (sent[i]+" ");
         out.append (sent[sent.length-1]+".");
         String sentence = out.toString ();
         System.out.println (
              "\nSentence: \""+sentence+"\"");
         boolean successful =
              parser.parseSentence (sent);
         System.out.println (
              "Parse Successful:" + successful);
         Chart[] charts = parser.getCharts ();
         System.out.println ("");
         System.out.println (
              "Charts produced by the sentence
                   \""+sentence+"\"");
         for (int i=0; i<charts.length; i++)
         {
              System.out.println ("Chart "+ i + ":");
              System.out.println (charts[i]);
         }
    }
}
```

30.5 *Generating Parse Trees from Charts and Grammar Rules (Advanced Section)*

All the topics discussed, as well as all the data structures and algorithms we have designed to this point in Chapter 30, have been used to build a chart that indicates whether or not a set of grammar rules are sufficient for parsing a string of words. In this final section we present some ideas for extracting parse trees from the charts created and the grammar rules that supported them. Thus, we have completed the *forward* component of the *forward/backward* algorithm (Luger 2009, Section 4.1 and Section 15.2.2) used in dynamic programming. We now present some ideas for completing the *backward* component of the dynamic programming algorithm: how we can use the chart and set of grammar rules to extract parse trees. We consider this an advanced topic, and so will present only the main components of an algorithm and leave its design as an exercise. Included with the software is our implementation of the stack-based approach to this problem.

To begin, we must determine how each state in the chart is created. One method for accomplishing this is to list all the ways that each state of the

chart can be produced. Another approach is to record this information when each of the states of the chart is first produced. We will discuss and implement this second method. To record the sources of a `State` we can add to the `State` class a `Vector` of `States`. This `Vector` will contain all `States` that produced this `State`. To maintain this, when a `State` is added to a `Chart`, if the `State` is already within the `Chart`, merge the sources of the `State` within the `Chart` and the one we attempted to add. This approach offers a method to look back through the charts quickly to find the possible parse trees.

It is important to remember that more then one parse tree can often be produced from `Chart`. An example is that the sentence "John called Mary from Denver" has two interpretations (parses): John called Mary, who is from Denver; and John called Mary, and John is in Denver. So however we produce the parse trees, we need to decide if we will attempt to produce all trees, or select only one. With appropriate forethought, it is easy to produce all parse trees.

To generate parse trees, the backward component of the dynamic programming algorithm, we begin with the final state `("$ → S @", 0, sentenceLength+1)`. For each source state for that final state, we iterate through the following:

1. If the source state is the start state `("$ → @ S", 0, 0)`, return the tree created. Otherwise continue.
2. Look at the current state we are evaluating. If the dot is last, then add `LHS` to the tree as the left-most child of the current evaluating node. We add it as the left-most child because we are building the tree right to left. This means that we will find the state with the last word of the sentence before any other preceding words. We will find the right-most child first, and want the subsequent children to be added to the left.
3. If the state's `LHS` is a part of speech, `POS`, then add the `RHS`'s first term as the only child of the node we just added. We are guaranteed that the `LHS` was just added as a child, because for this type of state there is a single term in the `RHS` and a final dot. For example: "`Noun → John @`". We have finished evaluating this state, so move to its source state and continue evaluation. There are two cases for this:
    a. There is only one source. Easy! Use that one source.
    b. There are multiple sources. Now we need to find the one that matches how we have been building the tree. To demonstrate what we mean, consider the more complex sentence "Old men and women like dogs." The ambiguity for this sentence is: only the men are old, or if both the men and women are old. "and" is a conjunction, which is a part of speech so it would match this rule. Here is the problem:

We have just finished with the state: `("Conj → and @", 2, 3)`. The sources of this state are both `("NP → NP @ Conj NP", 0,`

`2)` and `("NP → NP @ Conj NP", 1, 2)`. Which one should we use? First, notice that the only difference between the two source states, is the location of `i`. (The `i` describes where the start of each rule is.) In `("NP → NP @ Conj NP", 0, 2)`, the start is at the beginning of the sentence. For `("NP → NP @ Conj NP", 1, 2)`, the start is "men". Thus the parse difference between "old (men) and women" and "old (men and women)".

We need to look at where the previous states we have used to make the parse tree are looking. If for this particular tree, we had used `("NP → NP Conj @ NP", 0, 3)` then we need to use `("NP → NP @ Conj NP", 0, 2)`. For `("NP → NP Conj @ NP", 1, 3)` we would use `("NP → NP @ Conj NP", 1, 2)`. Therefore we must find the source state that matches the rule ignoring the position of the dot (this should be off by one), and the i.

4. At this point the current state may have been added to the tree, or it may not have. In either case we need to determine whether there are multiple sources for this state, and if there are, we need to iterate across each of them to determine which of the sources are valid. Before we do this, we need to update the current evaluating node. Above, we mentioned a current evaluating node. This is the node we are adding children to. Before we can continue, we need to determine if this node needs to change for the next iteration. There are three cases:
    a. The current state's dot is first. This means there are no more children that need to be added to this node. So we move to the parent of the current evaluating node.
    b. If the current state's `LHS` was just added to the tree and it was not a part of speech, then we will want to be adding the next nodes to the node we just added. So the current evaluating node moves to its left-most child.
    c. Otherwise, continue to use this node. This happens if we have just added a part of speech, `POS`, node and its child.
5. Next, we must iterate through all of these sources, and for each source state that meets the criteria, we attempt to continue building the tree from that state. One of the following must be true for this continuation to be accomplished:
    a. The source state's `LHS` is equal to the current state's term prior to the dot. Remember, we are moving from right to left, and the dot is moving from right to left. So if this is true, then the current state was generated because the source state completed a rule (the dot moved all the way to the right) and the `completer(…)` method was called.
    b. The source state's `RHS`, with dot moved to the right, and `LHS` matches on a state we have already evaluated.

6. If the criteria fail for all source states, then this was a dead end, and no tree is returned. If any of the source states are valid, start at step 1 with that state as the current state, and update the current evaluating node. The accepted trees (their may be no possible tree) are bundled together and returned.

From this algorithm, we can produce the multiple parse trees implicit in the Earley algorithm's successful production of the `Chart`. Example code implementing this algorithm is included with the Chapter 30 support materials.

The Earley parser code as well as the first draft of this chapter was written by Ms Breanna Ammons, MS in Computer Science, University of New Mexico.

Exercises

1. Describe the role of the dot within the right hand side of the grammar rules processed by the Earley parser. How is the location of the dot changed as the parse proceeds? What does it mean that the same right hand side of a grammar rule can have dots at different locations?

2. In the Earley parser the input word list and the states in the state lists have indices that are related. Explain how the indices for the states of the state list are created.

3. Describe in your own words the roles of the `predictor`, `completer`, and `scanner` procedures in the algorithm for Earley parsing. What order are these procedures called in when parsing a sentence, and why is that ordering important? Explain your answers to the order of procedure invocation in detail.

4. Use the Java parser to consider the sentence "John saw the burglar with the telescope". Parse also "Old men and women like dogs". Comment on the different parses possible from these sentences and how you might retrieve them from the chart.

5. Create a `Sentence` class. Have one of the constructors take a `String`, and have it then separate the `String` into words.

6. Code the algorithm for production of parse trees from the completed `Chart`. One method of recording the sources is presented in Section 30.5. You may find it useful to use a stack to keep track of the states you have evaluated for the parse tree.

7. Extend `EarleyParser` to include support for context-sensitive (Luger 2009, Section 15.9.5) grammar rules. What new structures are necessary to guarantee constraints across subtrees of the parse?