9 Dynamic Programming and the Earley Parser

Chapter Objectives

Language parsing with dynamic programming technique
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 parsing right side of grammar rule
- Lists of states make up components of chart
  - Chart linked to word list

Prolog implementation of an Earley parser
- Context free parser
  - Deterministic
- Chart supports multiple parse trees

Forwards development of chart composes components of successful parse
Backwards search of chart produces possible parses of word list
Earley parser important use of meta-interpreter technology.

Chapter Contents

9.1 Dynamic Programming Revisited
9.2 Earley Parsing: Pseudocode and an Example
9.3 The Earley Parser in Prolog

9.1 Dynamic Programming Revisited

The dynamic programming (DP) approach to problem solving was originally proposed by Richard Bellman (1956). The idea is straightforward: when addressing a large complex problem that can be broken down into multiple subproblems, save partial solutions as they are generated so that they can be reused later in the solution process for the full problem. This “save and reuse of partial solutions” is sometimes called *memoizing* the subproblem solutions for later reuse.

There are many examples of dynamic programming in pattern matching technology, for example, it has been used in determining a difference measure between two strings of bits or characters. The overall difference between the strings will be a function of the differences between its specific components. An example of this is a spell checker going to its dictionary and suggesting words that are “close” to your misspelled word. The spell checker determines “closeness” by calculating a difference measure between your word and words that it has in its dictionary. This difference is often calculated, using some form of the DP algorithm, as a

function of the differences between the characters in each word. Examples of the DB comparison of character strings are found in Luger (2009, Section 4.1.2).

A further example of the use of DP is for recognizing words in speech understanding as a function of the possible phonemes from an input stream. As phonemes are recognized (with associated probabilities), the most appropriate word is often a function of the combined conjoined probabilistic measures of the individual phones. The DP Viterbi algorithm can be used for this task (Luger 2009, Section 13.1).

In this section, we present the Earley parser, a use of dynamic programming to build a context-free parser that recognizes strings of words as components of syntactically correct sentences. The presentation and Prolog code of this chapter is based on the efforts of University of New Mexico graduate student Stan Lee. The pseudo-code of Section 9.2 is adapted from that of Jurafsky and Martin (2008).

9.2 The Earley Parser

The parsing algorithms of Chapter 8 are based on a recursive, depth-first, and left-to-right search of possible acceptable syntactic structures. This search approach can mean that many of the possible acceptable partial parses of the first (left-most) components of the string are repeatedly regenerated. This revisiting of early partial solutions within the full parse structure is the result of later backtracking requirements of the search and can become exponentially expensive and costly in large parses. Dynamic programming provides an efficient alternative where partial parses, once generated, are saved for reuse in the overall final parse of a string of words. The first DP-based parser was created by Earley (1970).

Memoization And Dotted Pairs

In parsing with Earley's algorithm the memoization of partial solutions (partial parses) is done with a data structure called a *chart*. This is why the various alternative forms of the Earley approach to parsing are sometimes called *chart parsing*. The chart is generated through the use of *dotted grammar rules*.

The dotted grammar rule provides a representation that indicates, in the chart, the state of the parsing process at any given time. Every dotted rule falls into one of three categories, depending on whether the dot's position is at the beginning, somewhere in the middle, or at the end of the right hand side, RHS, of the grammar rule. We refer to these three categories as the *initial*, *partial,* or *completed* parsing stages, respectively:

Initial prediction: Symbol → @ RHS_unseen

Partial parse: Symbol → RHS_seen @ RHS_unseen

Completed parse: Symbol → RHS_seen @

In addition, there is a natural correspondence between states containing different dotted rules and the edges of the parse tree(s) produced by the parse. Consider the following very simple grammar, where terminal symbols are surrounded by quotes, as in "mary":

```
Sentence  →  Noun  Verb
Noun  →  "mary"
Verb  →  "runs"
```

As we perform a top-down, left-to-right parse of this sentence, the following sequence of states is produced:

`Sentence`	→	`• Noun Verb`	*predict: Noun followed by Verb*
`Noun`	→	`• mary`	*predict:* `mary`
`Noun`	→	`mary •`	*scanned:* `mary`
`Sentence`	→	`Noun • Verb`	*completed: Noun;* *predict: Verb*
`Verb`	→	`• runs`	*predict:* `runs`
`Verb`	→	`runs •`	*scanned:* `runs`
`Sentence`	→	`Noun Verb •`	*completed: Verb,* *completed: sentence*

Note that the *scanning* and *completing* procedures deterministically produce a result. The *prediction* procedure describes the possible parsing rules that can apply to the current situation. Scanning and prediction creates the states in the parse tree of Figure 9.1.

Earley's algorithm operates by generating top-down and left-to-right predictions of how to parse a given input. Each prediction is recorded as a *state* containing all the relevant information about the prediction, where the key component of each state is a dotted rule. (A second component will be introduced in the next section.) All of the predictions generated after examining a particular word of the input are collectively referred to as the *state set*. For a given input sentence with n words, w_1 to w_n, a total $n + 1$ state sets are generated: $[S_0, S_1, \ldots, S_n]$. The initial state set, S_0, contains those predictions that are made before examining any input words, S_1 contains predictions made after examining W_1, and so on.

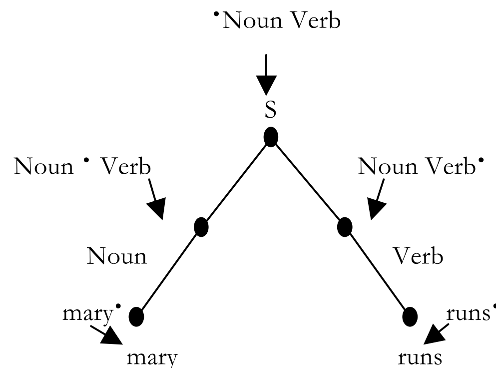


Figure 9.1 The relationship of dotted rules to the generation of a parse tree.

We refer to the entire collection of state sets as the *chart* produced by the parser. Figure 9.1 illustrates the relationship between state set generation and the examination of input words.

At this point we need to pause to get our terminology straight. Although, traditionally, the sets of states that make up each component of the parse are called *state sets*, the *order* of the generation of these states is important. Thus we call each component of the chart the *state list*, and describe it as [$State_1$, $State_2$, …, $State_n$]. This also works well with the Prolog implementation, Section 9.3, where the state lists will be maintained as Prolog lists. Finally, we describe each state of the state list as a sequence of specific symbols enclosed by brackets, for example, ($ → • S).

We now consider Earley's algorithm parsing the simple sentence `mary runs`, using the grammar above. The algorithm begins by creating a dummy start state, ($ → • S), that is the first member of state list S_0. This state represents the prediction that the input string can be parsed as a sentence, and it is inserted into S_0 prior to examining any input words. A successful parse produces a final state list S_n, which is S_2 in this example, that contains the state ($ → S •).

Beginning with S_0, the parser executes a loop in which each state, S_i, in the current state list is examined *in order* and used to generate new states. Each new state is generated by one of three procedures that are called the *predictor*, *scanner*, and *completer*. The appropriate procedure is determined by the dotted rule in state S, specifically by the grammar symbol (if any) following the dot in the rule.

In our example, the first state to be examined contains the rule ($ → • S). Since the dot is followed by the symbol S, this state is "expecting" to see an instance of S occur next in the input. As S is a nonterminal symbol of the grammar, the predictor procedure generates all states corresponding to a possible parse of S. In this case, as there is only one alternative for S, namely that S → Noun Verb, only one state, (S → • Noun Verb), is added to S_0. As this state is expecting a part of speech, denoted by the nonterminal symbol Noun following the dot, the algorithm examines the next input word to verify that prediction. This is done by the scanner procedure, and since the next word matches the prediction, `mary` is indeed a Noun, the scanner generates a new state recording the match: (Noun → mary •). Since this state depends on input word W_1, it becomes the first state in state list S_1 rather than being added to S_0. At this point the chart, containing two state lists, looks as follows, where after each state we name the procedure that generated it:

S_0:	[($ → • S),	dummy start state
	(S → • Noun Verb)]	predictor
S_1:	[(Noun → mary •)]	scanner

Each state in the list of states S_0 has now been processed, so the algorithm moves to S_1 and considers the state (Noun → mary •). Since this is a completed state, the completer procedure is applied. For each state expecting a Noun, that is, has the • Noun pattern, the completer

generates a new state that records the discovery of a `Noun` by advancing the dot over the `Noun` symbol. In this case, the completer produces the state (`S → • Noun Verb`) in S_0 and generates the new state (`S → Noun • Verb`) in the list S_1. This state is expecting a part of speech, which causes the scanner to examine the next input word W_2. As W_2 is a `Verb`, the Scanner generates the state (`Verb → runs •`) and adds it to S_2, resulting in the following chart:

S_0:	[($ → • S),	start
	(S → • Noun Verb)]	predictor
S_1:	[(Noun → `mary` •),	scanner
	(S → Noun • Verb)]	completer
S_2:	[(Verb → `runs` •)]	scanner

Processing the new state S_2, the completer advances the dot in (`S → Noun • Verb`) to produce (`S → Noun Verb •`), from which the completer generates the state (`$ → S •`) signifying a successful parse of a sentence. The final chart for `mary runs`, with three state lists, is:

S_0:	`[($ → • S),`	start
	`(S → • Noun Verb)]`	predictor
S_1:	`[(Noun → mary •),`	scanner
	`(S → Noun • Verb)]`	completer
S_2:	`[(Verb → runs •),`	scanner
	`(S → Noun Verb •),`	completer
	`($ → S •)]`	completer

Earley Pseudocode

To represent computationally the state lists produced by the dotted pair rules above, we create indices to show how much of the right hand side of a grammar rule has been parsed. We first describe this representation and then offer pseudo-code for implementing it within the Earley algorithm. Each state in the state list is augmented with an index indicating how far the input stream has been processed. Thus, we extend each state description to a (*dotted rule [i, j]*) representation where the *[i, j]* pair denotes how much of right hand side, RHS, of the grammar rule has been seen or parsed to the present time. For the right hand side of a parsed rule that includes zero or more seen and unseen components indicated by the •, we have (`A → Seen • Unseen, [i,j]`), where `i` is the start of `Seen` and `j` is the position of • in the word sequence.

We now add indices to the parsing states discussed earlier for the sentence `mary runs`:

`($ → • S, [0, 0])`
produced by predictor, `i = j = 0`, nothing yet parsed

`(Noun → mary •, [0,1])`
scanner sees `word[1]` between word indices `0` and `1`

```
(S → Noun • Verb, [0,1])
```
completer has seen Noun (mary) between chart 0 and 1

```
(S → Noun Verb •, [0,2])
```
completer has seen sentence S between chart 0 and 2

Thus, the state indexing pattern shows the results produced by each of the three state generators using the dotted rules along with the word index W_i.

To summarize, the three procedures for generating the states of the state list are: *predictor* generating states with index [j, j] going into chart[j], *scanner* considering word W_{j+1} to generate states indexed by [j, j+1] into chart[j+1], and *completer* operating on rules with index [i, j], i < j, adding a state entry to chart[j]. Note that a state from the dotted-rule, [i, j] always goes into the state list chart[j]. Thus, the state lists include chart[0], ..., chart[n] for a sentence of n words.

Now that we have presented the indexing scheme for representing the chart, we give the pseudo-code for the Earley parser. In Section 9.2.3 we use this code to parse an example sentence and in Section 9.3 we implement this algorithm in Prolog. We replace the "•" symbol with "@" as this symbol will be used for the dot in the Prolog code of Section 9.3.

```
function EARLEY-PARSE(words, grammar)  returns chart
      chart := empty
      ADDTOCHART(($ → @ S, [0, 0]), chart[0])
            % dummy start state

      for i from 0 to LENGTH(words) do
            for each state in chart[i] do
                 if rule_rhs(state) = … @ A …
                       and A is not a part of speech
                 then PREDICTOR(state)
                 else if rule_rhs(state) = … @ L …
                       % L is part of speech
                       then SCANNER(state)
                       else    OMPLETER(state)
                            % rule_rhs = RHS @
            end
      end
procedure PREDICTOR((A → … @ B …, [i,  j]))
     for each (B → RHS) in grammar do
           ADDTOCHART((B → @ RHS, [j,  j]), chart[j])
end

procedure SCANNER((A → … @ L …, [i,  j]))
     if (L → word[j]) is_in grammar
     then ADDTOCHART((L → word[j] @ , [j,  j + 1]),
           chart[j + 1])
end
```

```
procedure COMPLETER((B → …  @, [j, k]))
    for each (A → … @ B …, [i, j]) in chart[j] do
        ADDTOCHART((A → … B @ …, [i, k]), chart[k])
end
procedure ADDTOCHART(state, state-list)
    if state is not in state-list
    then ADDTOEND(state, state-lis
end
```

Earley Example

Our first example, the Earley parse of the sentence "Mary runs," was intended to be simple but illustrative, with the detailed presentation of the state lists and their indices. We now produce a solution, along with the details of the chart that is generated, for a more complex sentence, "John called Mary from Denver". This sentence is ambiguous (Did John use a phone while he was in Denver to call, or did John call that Mary that was from Denver). We present the two different parses of this sentence in Figure 9.2 and describe how they may both be recovered from the chart produced by parsing the sentence in an exercise. This retrieval process is typical of the dynamic programming paradigm where the parsing is done in a *forward* left-to-right fashion and then particular parses are retrieved from the chart by moving *backward* through the completed chart.

The following set of grammar rules is sufficient for parsing the sentence:

```
S  →  NP VP
NP  →  NP PP
NP  →  Noun
VP  →  Verb NP
VP  →  VP PP
PP  →  Prep NP
Noun  →  "john"
Noun  →  "mary"
Noun  →  "denver"
Verb  →  "called"
Prep  →  "from"
```

In Figure 9.2 we present two parse trees for the word string `john called mary from denver`. Figure 9.2a shows `john called (mary from denver)`, where Mary is from Denver, and in Figure 9.2b `john (called mary) (from denver)`, where John is calling from Denver. We now use the pseudo-code of the function EARLEY-PARSE to address this string. It is essential that the algorithm not allow any state to be placed in any state list more than one time, although the same state may be generated with different predictor/scanner applications:

1. Insert start state ($ → @ S, [0,0]) into chart[0]
2. Processing state-list S_0 = chart[0] for (i = 0):
   The *predictor* procedure produces within chart[0]:

```
($ → @ S, [0,0])) ==>
      (S → @ NP VP, [0,0])
(S → @ NP VP, [0,0])  ==>
      (NP → @ NP PP, [0,0])
(S → @ NP VP, [0,0])  ==>
      (NP → @ Noun, [0,0])
```

3. Verifying that the next word word[*i* + 1] = word[1] or "john" is a Noun:

   The *scanner* procedure initializes chart[1] by producing

```
(NP → @ Noun, [0,0]) ==>
      (Noun → john @, [0,1])
```

4. Processing S_1 = chart[1] shows the typical start of a new state list, with the *scanner* procedure processing the next word in the word string, and the algorithm then calling *completer*.

   The *completer* procedure adds the following states to chart[1]:

```
(NP → Noun @, [0,1])
(S → NP @ VP, [0,1])              from x1
(NP → NP @ PP, [0,1])             from x2
```

5. The completer procedure ends for S_1 as no more states have "dots" to advance, calling *predictor*:

   The *predictor* procedure generates states based on all newly-advanced dots:

```
(VP → @ Verb NP, [1,1])           from x1
(VP → @ VP PP, [1,1])        also from x1
(PP → @ Prep NP, [1,1])           from x2
```

6. Verifying that the next word, word[*i* + 1] = word[2] or "called" is a Verb:

   The *scanner* procedure initializes chart[2] by producing:

```
(VP → @ Verb NP, [1,1]) ==>
      (Verb → called @, [1,2])
```

Step 6 (above) initializes `chart[2]` by scanning `word[2]` in the word string; the *completer* and *predictor* procedures then finish state list 2.

The function EARLEY-PARSE continues through the generation of `chart[5]` as seen in the full chart listing produced next. In the full listing we have annotated each state by the procedure that generated it. It should also be noted that several partial parses, indicated by *, are generated for the chart that are not used in the final parses of the sentence. Note also that the fifth and sixth states in the state list of `chart[1]`, indicated by **, which predict two different types of VP beginning at index 1, are instrumental in producing the two different parses of the string of words, as presented in Figure 9.2.

```
chart[0]:
   [($ → @ S, [0,0])                    start state
   (S → @ NP VP, [0,0])                 predictor
   (NP → @ NP PP, [0,0])*               predictor
   (NP → @ Noun, [0,0])]                predictor
chart[1]:
   [(Noun → john @, [0,1])              scanner
   (NP → Noun @, [0,1])                 completer
   (S → NP @ VP, [0,1])                 completer
   (NP → NP @ PP, [0,1])*               completer
   (VP → @ Verb NP, [1,1])**            predictor
   (VP → @ VP PP, [1,1])**              predictor
   (PP → @ Prep NP, [1,1])*]            predictor
chart[2]:
   [(Verb → called @, [1,2])            scanner
   (VP → Verb @ NP, [1,2])              completer
   (NP → @ NP PP, [2,2])                predictor
   (NP → @ Noun, [2,2])[                predictor
chart[3]:
   [(Noun → mary @, [2,3])              scanner
   (NP → Noun @, [2,3])                 completer
   (VP → Verb NP @, [1,3])              completer
   (NP → NP @ PP, [2,3])                completer
   (S → NP VP @, [0,3])*                completer
   (VP → VP @ PP, [1,3])                completer
   (PP → @ Prep NP, [3,3])              predictor
   ($ → S @, [0,3])*]                   completer
chart[4]:
   [(Prep → from @, [3,4])              scanner
   (PP → Prep @ NP, [3,4])              completer
   (NP → @ NP PP, [4,4])*               predictor
   (NP → @ Noun, [4,4])]                predictor
chart[5]:
   [(Noun → denver @, [4,5])            scanner
   (NP → Noun @, [4,5])                 completer
   (PP → Prep NP @, [3,5])              completer
   (NP → NP @ PP, [4,5])*               completer
   (NP → NP PP @, [2,5])                completer
   (VP → VP PP @, [1,5])                completer
   (PP → @ Prep NP, [5,5])*             predictor
   (VP → Verb NP @, [1,5])              completer
   (NP → NP @ PP, [2,5])*               completer
   (S → NP VP @, [0,5])                 completer
   (VP → VP @ PP, [1,5])*               completer
   ($ → S @, [0,5])]                    completer
```

The complete chart generated by the EARLEY-PARSE algorithm contains 39 states separated into six different state lists, charts 0 – 5. The final state list contains the success state `($ → S @, [0,5])` showing that the string containing five words has been parsed and is indeed a sentence. As pointed out earlier, there are states in the chart, ten indicated by *, that are not part of either of the final parse trees, as seen in Figure 9.2.

9.3 The Earley Parser in Prolog

Finally, we present the Earley parser in Prolog. Our Prolog code, designed by Stan Lee, a graduate student in Computer Science at the University of New Mexico, is a direct implementation of the EARLEY-PARSE pseudo-code given in Section 9.2.2. When looking at the three procedures that follow - scanner, predictor, and completer – it is important to note that similarity.

The code begins with initialization, including reading in the word string, parsing, and writing out the chart after it is created:

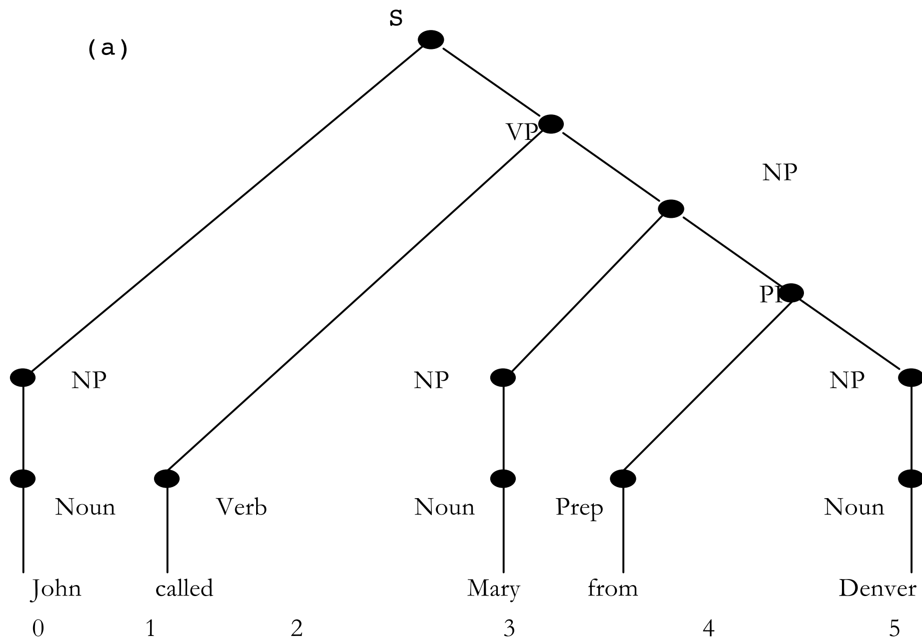
```
go :- go(s).
go(NT) :
      input(Words),
      earley(NT, Words, Chart),
      writesln(Chart).
```

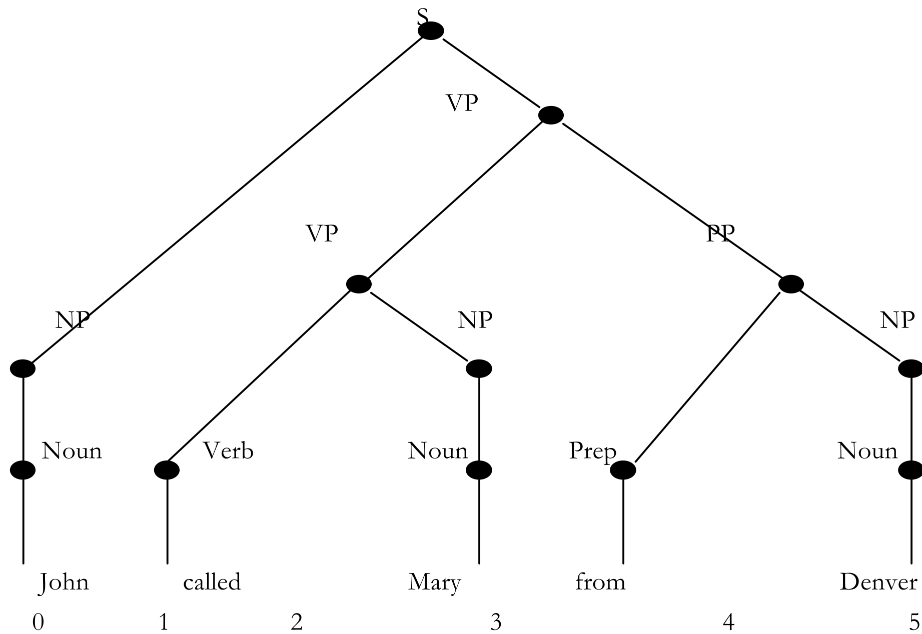
The `earley` predicate first generates the start state, `StartS` for the parser and then calls the state generator `state_gen` which produces the chart, `Chart`. `state_gen` checks first if the wordlist is exhausted and terminates if it is, next it checks if the current state list is completed and if it is, begins working on the next state, otherwise it continues to process the current state list:

```
earley(NonTerminal, Words, Chart) :-
      StartS = s($, [@, NonTerminal], [0,0]),
      initial_parser_state(Words, StartS, PS),
      state_gen(PS, Chart).
state_gen(PS, Chart) :-          %Si = [], Words = []
      final_state_set_done(PS, Chart)
state_gen(PS, Chart) :-        %Si = [], Words not []
      current_state_set_done(PS, NextPS),
      state_gen(NextPS, Chart).
state_gen(PS, Chart) :-                %Si = [S|Rest]
      current_state_rhs(S, RHS, PS, PS2),
                                        %PS2[Si] = Rest
      (
      append(_, [@, A|_], RHS),
      rule(A, _) ->           %A not a part of speech
      predictor(S, A, PS2, NextPS)
      ;
      append(_, [@, L|_], RHS),
      lex_rule(L, _) ->          %L is part of speech
      scanner(S, L, PS2, NextPS)
      ;
```

```
completer(S, PS2, NextPS) %S is completed state
),
state_gen(NextPS, Chart).
```

(a)





(b)

Figure 9.2. Two different parse trees for the word string representing the sentence "John called Mary from Denver". The index scheme for the word string is below it.

We next present the `predictor` procedure. This procedure takes a dotted rule `A -> ... @ B ...` and predicts a new entry into the state list for the symbol `B` in the grammar:

```
predictor(S, B, PS, NewPS) :-
     S = s(_, _, [I,J]),
     Findall
          (
          s(B, [@ | RHS], [J,J]),
          rule(B, RHS),
          NewStates
          ),
     add_to_chart(NewStates, PS, NewPS).
```

The `scanner` procedure considers the next word in the input string. If it is a part of speech, `Lex`, `scanner` creates a new state list and enters that part of speech, for example, the state (`Noun → denver @, [4,5]`), that begins `chart[5]` in Section 9.2.3. The `scanner` procedure prepares the way for the `completer` and `predictor` procedures. If the next word in the input stream is not the predicted part of speech, it leaves the chart unchanged:

```
scanner(S, Lex, PS, NewPS) :-
     S = s(_, _, [I,J]),
     next_input(Word, J, J1, PS),
     lex_rule(Lex, [Word]), !,
     add_to_chart( [s(Lex, [Word,@], [J,J1])], PS,
          NewPS).
scanner(_, _, PS, PS).
```

Finally, the `completer` procedure takes a completed state `S` that has recognized a pattern `B`, and adds a new state to the list for each preceding state that is looking for that pattern.

```
completer(S, PS, NewPS) :-
     S = s(B, _, [J,K]),
     Findall
          (
          s(A, BdotRHS, [I,K]),
          (
          in_chart( s(A, DotBRHS, [I,J]), PS),
          append(X, [@, B|Y], DotBRHS),
          append(X, [B, @|Y], BdotRHS % adv dot over B
          ),
          NewStates
          ),
     add_to_chart(NewStates, PS, NewPS).
```

We next describe the utility predicates that support the three main procedures just presented. The most important of these are predicates for maintaining the state of the parser itself. The parser-state, `PS`, is represented by a structure `ps` with five arguments: `PS = ps(Words, I, Si, SNext, Chart)`. The first argument of `ps` is the current string of words maintained as a list and the second argument, `I`, is the current index of `Words`. `Si` and `SNext` are the current and next state

lists, and Chart is the current chart. Thus, Si and SNext are always subsets of the current Chart. Notice that the "assignment" that creates the next state-list is done with unification (=).

The PS utilities perform initial and final state checks, determine if the current state list is complete, extract components of the current state and get the next input value. Parsing is finished when the Word list and the current state list Si, the first and third arguments of PS, are both empty. If Si is empty but the Word list is not, then the next state list becomes the new current state list and the parser moves to the next index as is seen in the current_state_set_done predicate:

```
initial_parser_state(Words, StartState, InitPS) :-
     InitPS = ps(Words, 0, [StartState], [],
          [StartState]).
final_state_set_done( ps([], _, [], _, FinalChart),
          FinalChart).
current_state_set_done( ps([_|Words], I, [], SNext,
          Chart), ps(   Words,  J, SNext, [],
          Chart)) :-
     J is I+1.

current_state_rhs(S, RHS, ps(Words, I, [S|Si],
          SNext, Chart), ps(Words, I,    Si,  SNext,
          Chart)) :-
S = s(_, RHS, _).
```

In the final predicate, S is the first state of the current state list (the third argument of ps, maintained as a list). This is removed, and the patterns of the right hand side of the current dotted grammar rule, RHS, are isolated for interpretation. The current state list Si is the tail of the previous list.

More utilities: The next element of Words in PS is between the current and next indices. The chart is maintained by checking to see if states are already in its state lists. Finally, there are predicates for adding states to the current chart.

```
next_input(Word, I, I1, ps([Word|_], I, _, _, _)) :-
     I1 is I+1.
add_to_chart([], PS, PS).
add_to_chart([S|States], PS, NewPS) :-
     in_chart(S, PS),!,
     add_to_chart(States, PS, NewPS).
add_to_chart([S|States], PS, NewPS) :-
     add_to_state_set(S, PS, NextPS),
     add_to_chart(States, NextPS, NewPS).
in_chart(S, ps(_, _, _, _, Chart)) :-
     member(S, Chart).
```

```
add_to_state_set(S, PS, NewPS) :-
     PS = ps(Words, I, Si, SNext, Chart),
     S = s(_, _, [_,J]),
     add_to_end(S, Chart, NewChart),
     (
     I == J ->                          %S is not a scan
state
        add_to_end(S, Si, NewSi),
        NewPS = ps(Words, I, NewSi, SNext, NewChart)
     ;
        add_to_end(S, SNext, NewSNext),
        NewPS = ps(Words, I, Si, NewSNext, NewChart)
     ).
add_to_end(X, List, NewList) :-
     append(List, [X], NewList).
```

The `add_to_state_set` predicate, first places the new state in the new version of the chart, `NewChart`. It then checks whether the current word list index `I` is the same as the second index `J` of the pair of indices of the state being added to the state list, testing whether `I == J`. When this is true, that state is added to the end (made the last element) of the current state list S_i. Otherwise, the new state was generated by the `scanner` procedure after reading the next word in the `input` word list. This new state will begin a new state list, `SNext`.

Finally, we present the output of the Prolog `go` and `earley` predicates running on the word list "John called Mary from Denver":

```
?- listing([input, rule, lex_rule]).
input([john, called, mary, from, denver]).
rule(s, [np, vp]).
rule(np, [np, pp]).
rule(np, [noun]).
rule(vp, [verb, np]).
rule(vp, [vp, pp]).
rule(pp, [prep, np]).
lex_rule(noun, [john]).
lex_rule(noun, [mary]).
lex_rule(noun, [denver]).
lex_rule(verb, [called]).
lex_rule(prep, [from]).
?- go.
s($, [@, s], [0, 0])
s(s, [@, np, vp], [0, 0])
s(np, [@, np, pp], [0, 0])
s(np, [@, noun], [0, 0])
s(noun, [john, @], [0, 1])
s(np, [noun, @], [0, 1])
s(s, [np, @, vp], [0, 1])
s(np, [np, @, pp], [0, 1])
```

```
s(vp, [@, verb, np], [1, 1])
s(vp, [@, vp, pp], [1, 1])
s(pp, [@, prep, np], [1, 1])
s(verb, [called, @], [1, 2])
s(vp, [verb, @, np], [1, 2])
s(np, [@, np, pp], [2, 2])
s(np, [@, noun], [2, 2])
s(noun, [mary, @], [2, 3])
s(np, [noun, @], [2, 3])
s(vp, [verb, np, @], [1, 3])
s(np, [np, @, pp], [2, 3])
s(s, [np, vp, @], [0, 3])
s(vp, [vp, @, pp], [1, 3])
s(pp, [@, prep, np], [3, 3])
s($, [s, @], [0, 3])
s(prep, [from, @], [3, 4])
s(pp, [prep, @, np], [3, 4])
s(np, [@, np, pp], [4, 4])
s(np, [@, noun], [4, 4])
s(noun, [denver, @], [4, 5])
s(np, [noun, @], [4, 5])
s(pp, [prep, np, @], [3, 5])
s(np, [np, @, pp], [4, 5])
s(np, [np, pp, @], [2, 5])
s(vp, [vp, pp, @], [1, 5])
s(pp, [@, prep, np], [5, 5])
s(vp, [verb, np, @], [1, 5])
s(np, [np, @, pp], [2, 5])
s(s, [np, vp, @], [0, 5])
s(vp, [vp, @, pp], [1, 5])
s($, [s, @], [0, 5])
Yes
?-
```

We present the Earley parser again in Java, Chapter 30. Although the control procedures in Java are almost identical to those just presented in Prolog, it is interesting to compare the representational differences between declarative and an object-oriented languages.

Next, in the final chapter of Part I, we discuss important features of Prolog and declarative programming. We present Lisp and functional programming in Part III.

Exercises

1. Describe the role of the dot within the right hand side of the grammar rules as they are processed by the Earley parser. How is the location of the dot changed as the parse proceeds? What does it mean when we say 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. Augment the Earley Prolog parser to consider the sentence "John saw the burglar with the telescope". Create two different possible pare trees from interpreting this string and comment on how the different possible parses are retrieved them from the chart.

5. Create an 8 – 10 word sentence of your own and send it to the Earley parser. Produce the chart as it changes with each additional word of the sentence that is scanned.

6. Create a grammar that includes adjectives and adverbs in its list of rules. What changes are needed for the Earley parser to handle these new rules? Test the Early parser with sentences that contain adjectives and adverbs.

7. In the case of "John called Mary from Denver" the parser produced two parse trees. Analyze Figure 9.4 and show which components of the full parse are shared between both trees and where the critical differences are.

8. Analyze the complexity of the Earley algorithm. What was the cost of the two parses that we considered in detail in this chapter? What are the worst- and best-case complexity limits? What type sentences force the worst case? Alternatively, what types of sentences are optimal?