# 11

# Parsing Techniques

## 11.1 Introduction

In the previous chapters, we used Prolog's built-in search mechanism and the DCG notation to parse sentences and constituents. This search mechanism has drawbacks however. To name some of them: its depth-first strategy does not handle left-recursive rules well and backtracking is sometimes inefficient. In addition, if DCGs are appropriate to describe constituents, we haven't seen means to parse dependencies until now.

This chapter describes algorithms and data structures to improve the efficiency of constituents parsing and to parse dependencies. It begins with a basic bottom-up algorithm and then introduces techniques using well-formed substring tables or charts. Charts are arrays to store parsing results and hypotheses. They are popular parsing devices because of some superior features: charts accept left-recursive rules, avoid backtracking, and can work with a top-down or bottom-up control.

Frequently, sentences show an ambiguous structure – exhibit more than one possible parse. Search strategies, either bottom-up or top-down, produce solutions blindly; the ordering of the resulting parse trees being tied to that of the rules. For most cases however sentences are not ambiguous to human readers who retain one single sensible analysis. To come to a similar result, parsers require a disambiguation mechanism.

Early disambiguation methods implemented common sense rules to assess parse trees and to discard implausible ones. Current solutions, inspired from speech recognition and part-of-speech tagging, use statistical techniques. They enable us to parse properly most ambiguous sentences. Recent approaches based on dependencies yield a very high rate of performance for unrestricted texts. This chapter outlines symbolic techniques as well as probabilistic methods applicable to constituency and dependency grammars.

## 11.2 Bottom-up Parsing

### 11.2.1 The Shift–Reduce Algorithm

We saw in Chap. 8 that left-recursive rules may cause top-down parsing to loop infinitely. Left-recursion is used to express structures such as noun phrases modified by a prepositional phrase, or conjunctions of noun phrases as, for example,

```
np --> np, pp.
np --> np, conj, np.
```

It is possible to eliminate left-recursive rules using auxiliary symbols and rules. However, this results in larger grammars that are less regular. In addition, parsing with these new rules yields slightly different syntactic trees, which are often less natural.

A common remedy to handle left-recursive rules is to run them with a bottom-up search strategy. Instead of expanding constituents from the top node, a bottom-up parser starts from the words. It looks up their parts of speech, builds partial structures out of them, and goes on from partial structure to partial structure until it reaches the top node. Figure 11.1 shows the construction order of partial structures that goes from the annotation of *the* as a determiner up to the root `s`.

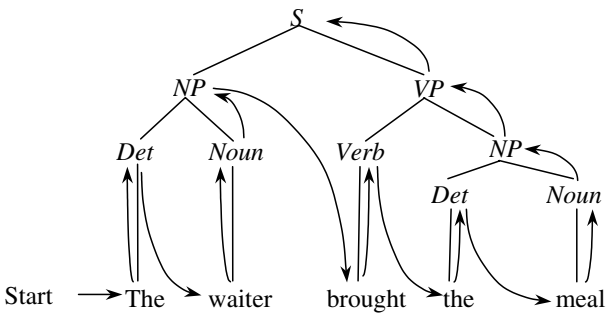


**Fig. 11.1.** Bottom-up parsing. The parser starts with the words and builds the syntactic structure up to the top node.

The shift and reduce algorithm is probably the simplest way to implement bottom-up parsing. As input, it uses two arguments: the list of words to parse and a symbol, `s`, `np`, for example, representing the parsing goal. The algorithm gradually reduces words, parts of speech, and phrase categories until it reaches the top node symbol – the parsing goal. The algorithm consists of a two-step loop:

1. **Shift** a word from the phrase or sentence to parse onto a stack.
2. Apply a sequence of grammar rules to **reduce** elements of the stack.

This loop is repeated until there are no more words in the list and the stack is reduced to the parsing goal. Table 11.1 shows an example of shift and reduce operations applied to the sentence *The waiter brought the meal*.

**Table 11.1.** Steps of the shift–reduce algorithm to parse *the waiter brought the meal*. At iteration 7, a further reduction of the stack yields `[s]`, which is the parsing goal. However, since there are remaining words in the input list, the algorithm fails and backtracks to produce the next states of the stack. The table does not show the exploration of paths leading to a failure.

| It. | Stack | S/R | Word list |
|---|---|---|---|
| 0 | | | `[the, waiter, brought, the, meal]` |
| 1 | `[the]` | Shift | `[waiter, brought, the, meal]` |
| 2 | `[det]` | Reduce | `[waiter, brought, the, meal]` |
| 3 | `[det, waiter]` | Shift | `[brought, the, meal]` |
| 4 | `[det, noun]` | Reduce | `[brought, the, meal]` |
| 5 | `[np]` | Reduce | `[brought, the, meal]` |
| 6 | `[np, brought]` | Shift | `[the, meal]` |
| 7 | `[np, v]` | Reduce | `[the, meal]` |
| 8 | `[np, v, the]` | Shift | `[meal]` |
| 9 | `[np, v, det]` | Reduce | `[meal]` |
| 10 | `[np, v, det, meal]` | Shift | `[]` |
| 11 | `[np, v, det, n]` | Reduce | `[]` |
| 12 | `[np, v, np]` | Reduce | `[]` |
| 13 | `[np, vp]` | Reduce | `[]` |
| 14 | `[s]` | Reduce | `[]` |

### 11.2.2 Implementing Shift–Reduce Parsing in Prolog

We implement both arguments of the `shift_reduce/2` predicate as lists: the words to parse and the symbol – or symbols – corresponding to the parsing goal. We represent grammar rules and the vocabulary as facts, as shown in Table 11.2.

**Table 11.2.** Rules and vocabulary of a shift–reduce parser.

| Rules | Vocabulary | |
|---|---|---|
| `rule(s, [np, vp]).` | `word(d, [the]).` | `word(v, [brought]).` |
| `rule(np, [d, n]).` | `word(n, [waiter]).` | `word(v, [slept]).` |
| `rule(vp, [v]).` | `word(n, [meal]).` | |
| `rule(vp, [v, np]).` | | |

Using this grammar, `shift_reduce` should accept the following queries:

```
?- shift_reduce([the, waiter, brought, the, meal],
  [s]).
Yes

?- shift_reduce([the, waiter, brought, the, meal],
  [np, vp]).
Yes
```

```
?- shift_reduce([the, waiter, slept], X).
X = [s];
X = [np, vp];
X = [np, v];
...
```

To implement this predicate, we need an auxiliary stack to hold words and categories where we carry out the reduction step. This initial value of the stack is an empty list

```
% shift_reduce(+Sentence, ?Category)
shift_reduce(Sentence, Category) :-
  shift_reduce(Sentence, [], Category).
```

Then `shift_reduce/3` uses two predicates, `shift/4` and `reduce/2`. It repeats the reduction recursively until it no longer finds a reduction. It then applies `shift`. The parsing process succeeds when the sentence is an empty list and `Stack` is reduced to the parsing goal:

```
% shift_reduce(+Sentence, +Stack, ?Category)
shift_reduce([], Category, Category).
shift_reduce(Sentence, Stack, Category) :-
  reduce(Stack, ReducedStack),
  write('Reduce: '), write(ReducedStack), nl,
  shift_reduce(Sentence, ReducedStack, Category).
shift_reduce(Sentence, Stack, Category) :-
  shift(Sentence, Stack, NewSentence, NewStack),
  write('Shift: '), write(NewStack), nl,
  shift_reduce(NewSentence, NewStack, Category).
```

`shift/4` removes the first word from the word list currently being parsed and puts it on the top the stack – here appends it to the end of the `Stack` list – to produce a `NewStack`.

```
% shift(+Sentence, +Stack, -NewSentence, -NewStack)
shift([First | Rest], Stack, Rest, NewStack) :-
  append(Stack, [First], NewStack).
```

`reduce/2` simplifies the `Stack`. It searches the rules that match a sequence of symbols in the stack using `match_rule/2` and `match_word/2`.

```
%reduce(+Stack, -NewStack)
reduce(Stack, NewStack) :-
  match_rule(Stack, NewStack).
reduce(Stack, NewStack) :-
  match_word(Stack, NewStack).
```

`match_rule/2` attempts to find the `Expansion` of a rule on the top of `Stack`, and replaces it with `Head` to produce `ReducedStack`:

```
match_rule(Stack, ReducedStack) :-
  rule(Head, Expansion),
  append(StackBottom, Expansion, Stack),
  append(StackBottom, [Head], ReducedStack).
```

`match_word/2` is similar:

```
match_word(Stack, NewStack) :-
  append(StackBottom, Word, Stack),
  word(POS, Word),
  append(StackBottom, [POS], NewStack).
```

The stack management of this program is not efficient because `shift/4`, `match_word/2`, and `match_rule/2` have to traverse it using `append/3`. It is possible to avoid the traversal using a reversed stack. For an optimization, see Exercise 11.1.

### 11.2.3 Differences Between Bottom-up and Top-down Parsing

Top-down and bottom-up strategies are fundamental approaches to parsing. The top-down exploration is probably more intuitive from the viewpoint of a Prolog programmer, at least for a neophyte. Once a grammar is written, Prolog relies on its built-in search mechanism to parse a sentence. On the contrary, bottom-up parsing requires additional code and may not be as natural.

Whatever the parsing strategy, phrase-structure rule grammars are written roughly in the same way. There are a couple of slight differences, however. As we saw, bottom-up parsing can handle left-recursive rules such as those describing conjunctions. In contrast, top-down parsers can handle null constituents like

```
det --> [].
```

Bottom-up parsers could not use such a rule with an empty symbol because they are able to process actual words only.

Both parsing methods may fail to find a solution, but in a different way. Top-down parsing explores all the grammar rules starting from the initial symbol, whatever the actual tokens. It leads to the expansion of trees that have no chance to yield any solution since they will not match the input words. On the contrary, bottom-up parsing starts with the words and hence builds trees that conform to the input. However, a bottom-up analysis annotates the input words with every possible part of speech, and generates the corresponding partial trees, even if they have no chance to result into a sentence.

For both strategies, Prolog produces a solution – whenever it exists – using backtracking. When Prolog has found a dead-end path, whether in the bottom-up or the top-down mode, it selects another path and explores this path until it completes the

parse or fails. Backtracking may repeat a same operation since Prolog does not store intermediate or partial solutions. We will see in the next section a parsing technique that stores incomplete solutions using a table or a **chart** and thus avoids parsing repetitions.

## 11.3 Chart Parsing

### 11.3.1 Backtracking and Efficiency

Backtracking is an elegant and simple mechanism, but it frequently leads to reparsing a same substructure to produce the final result. Consider the noun phrase *The meal of the day* and DCG rules in Fig. 11.2 to parse it.

```
np --> npx.        npx --> det, noun.
np --> npx, pp.

pp --> prep, np.
```

**Fig. 11.2.** A small set of DCG rules where left-recursion has been eliminated.

The DCG search algorithm first tries rule `np --> npx`, uses `npx` to parse *The meal*, and fails because of the remaining words *of the day*. It then backtracks with the second `np` rule, reuses `npx` to reparse *The meal*, and finally completes the analysis with `pp`. Backtracking is clearly inefficient here. The parser twice applies the same rule to the same group of words because it has forgotten a previous result: *The meal* is an `npx`.

Chart – or tabular – parsing is a technique to avoid a parser repeating a same analysis. A chart is a memory where the parser stores all the possible partial results at a given position in the sentence. When it needs to process a subsequent word, the parser fetches partial parse structures obtained so far in the chart instead of reparsing them. At the end of the analysis, the chart contains all possible parse trees and subtrees that it represents tidily and efficiently.

### 11.3.2 Structure of a Chart

A chart represents intervals between words as nodes of a graph. Considering a sentence of $N$ words, nodes are numbered from left to right, from 0 to $N$. The chart – which can also be viewed as a table – then has $N+1$ entries or positions. Figure 11.3 shows word numbering of the sentence *Bring the meal* and the noun phrase *The meal of the day*. A chart node is also called a vertex.

Directed arcs (or edges) connect nodes and define constituents. Each arc has a label that corresponds to the syntactic category of the group it spans (Fig. 11.4). Charts

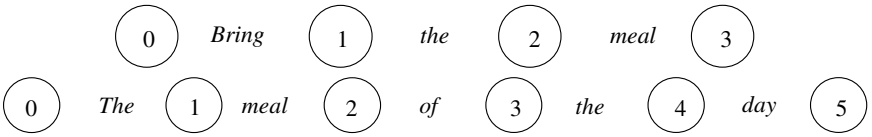


**Fig. 11.3.** Nodes of a chart.

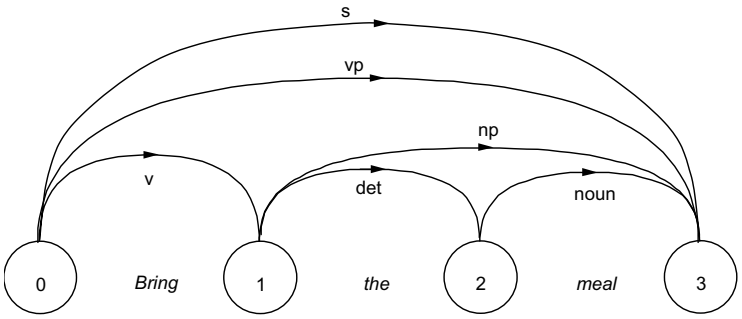


**Fig. 11.4.** Nodes and arcs of a chart associated with the sentence *Bring the meal*.

consist then of sets of nodes and directed labeled arcs. This algorithmic structure is also called a directed acyclic graph (DAG).

A chart can store alternative syntactic representations. As we saw in Chap. 8, the grammar in Fig. 11.5 yields two parse trees for the sentence

*Bring the meal of the day.*

```
s --> vp.          np --> det, noun.
vp --> v, np, pp.  np --> det, adj, noun.
vp --> v, np.      np --> np, pp.
                   pp --> prep, np.
```

**Fig. 11.5.** A small grammar for restaurant orders in English.

The chart of Fig. 11.6 shows the possible parses of this sentence. Rules `vp --> v, np` and `vp --> v, np, pp` create two paths that connect node 0 to node 6. Starting from node 0, the first one traverses nodes 1 and 6: arc `v` from 0 to 1, then `np` from 1 to 6. The second sequence of arcs traverses nodes 1, 3, 6: arc `v` from 0 to 1, then `np` from 1 to 3, and finally `pp` from 3 to 6.

### 11.3.3 The Active Chart

So far, we used charts as devices to represent partial or complete parse trees. Charts can also store representations of constituents currently being parsed. In this case, a chart is said to be active.

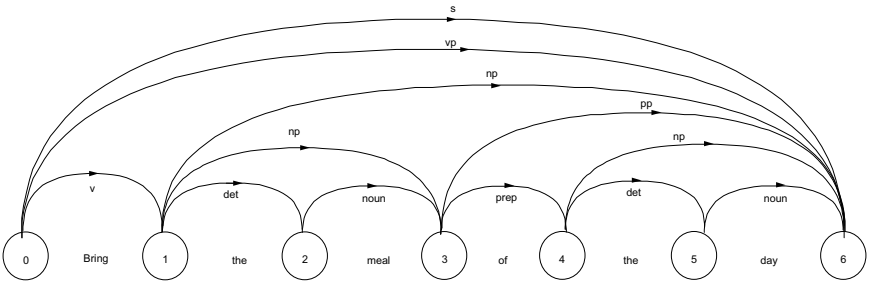


**Fig. 11.6.** A chart representing alternative parse trees.

In the classical chart notation, the parsing progress of a constituent is indicated using a dot (•) inserted in the right-hand side of the rules. A **dotted rule** represents what has been parsed so far, with the dot marking the position of the parser relative to the input. Thus

```
np --> det noun •
```

is a completely parsed noun phrase composed of a determiner and a noun. Since the constituent is complete, the arc is said to be inactive. Rules

```
np --> det • noun
np --> • det noun
```

describe noun phrases being parsed. Both correspond to constituent hypotheses that the parser tries to find. In the first rule, the parser has found a determiner and looks for a noun to complete the parse. The second rule represents the constituent being sought originally. Both arcs are said to be active since the parser needs more words from the input to confirm them.

Consider the sentence *Bring the meal*. Table 11.3 shows dotted-rules and arcs during the parsing process, and Fig. 11.7 shows a graphic representation of them.

**Table 11.3.** Some dotted-rules and arcs in the chart while parsing *Bring the meal of the day*.

| Positions | Rules | Arcs | Constituents |
|---|---|---|---|
| 0 | `s --> • vp` | [0, 0] | • *Bring the meal* |
| 1 | `vp --> v • np` | [0, 1] | *Bring* • *the meal* |
| 1 | `np --> • det noun` | [1, 1] | • *the meal* |
| 1 | `np --> • np pp` | [1, 1] | • *the meal* |
| 2 | `np --> det • noun` | [1, 2] | *the* • *meal* |
| 3 | `np --> det noun •` | [1, 3] | *the meal* • |
| 3 | `np --> np • pp` | [1, 3] | *the meal* • |
| 3 | `vp --> v np •` | [0, 3] | *Bring the meal* • |
| 3 | `s --> vp •` | [0, 3] | *Bring the meal* • |

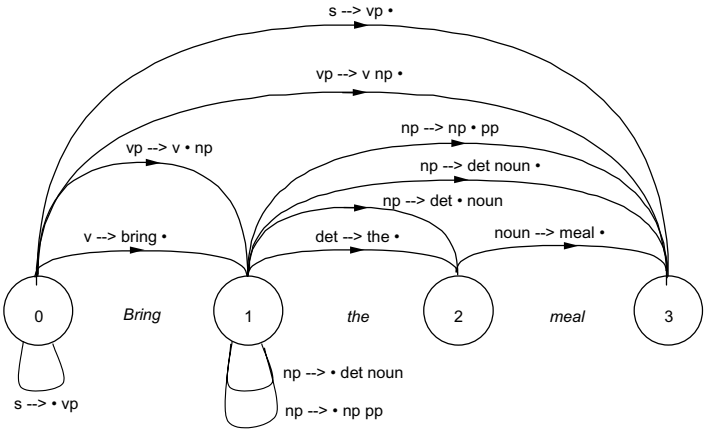


**Fig. 11.7.** Some arcs of a chart labeled with dotted-rules while parsing *Bring the meal of the day*.

Charts can be used with top-down, bottom-up, or more sophisticated strategies. We introduce now a top-down version due to Earley (1970). Its popularity comes from its complexity, which has been demonstrated as $0(N^3)$.

### 11.3.4 Modules of an Earley Parser

An Earley parser consists of three modules, the predictor, the scanner, and the completer, which are chained by the parsing process. The initial goal of the algorithm is to parse the start symbol, which is generally a sentence `s`. Here, we illustrate the algorithm with the noun phrase *The meal of the day* and the rules in Fig. 11.5. The start symbol is then `np` and is represented by the dotted-rule

`start --> • np`

**The Predictor.** At a given position of the parsing process, the predictor determines all possible further parses. To carry this out, the predictor selects all the rules that can process active arcs. Considering the dotted rule, `lhs --> c1 c2 ... • c... cn`, the predictor searches all the rules where `c` is the left-hand-side symbol: `c --> x1 ... xk`. The predictor introduces them into the chart as new parsing goals under the form `c --> • x1 ... xk`. The predictor proceeds recursively with nonterminal symbols until it reaches the parts of speech. Considering *The meal of the day* with `np` as the starting parsing goal and applying the predictor results in new goals shown in Fig. 11.8 and graphically in Fig. 11.9.

**The Scanner.** Once all possible predictions are done, the scanner accepts a new word from the input, here *the*. The parts of speech to the right of a dot are matched against the word, here in our example, rules `np --> • det noun` and `np --> • det adj noun`. The scanner inserts the word into the chart with all its matching

```
start --> • np          [0, 0]
np --> • det noun       [0, 0]
np --> • det adj noun   [0, 0]
np --> • np pp          [0, 0]
```

**Fig. 11.8.** Dotted-rules resulting from the recursive run of the predictor with starting goal `np`.

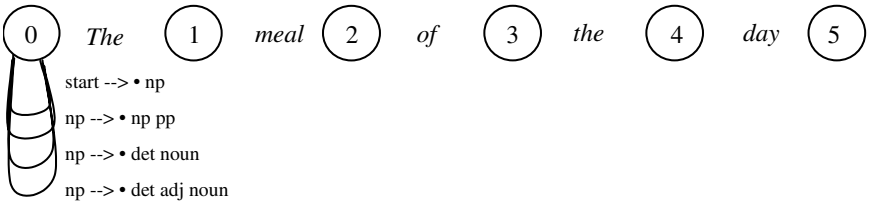


**Fig. 11.9.** Graphic representation of the predictor results.

part-of-speech readings under the form `pos --> word` • and advances the parse position to the next node, here

```
det --> the •    [0, 1]
```

as shown in Fig. 11.10.

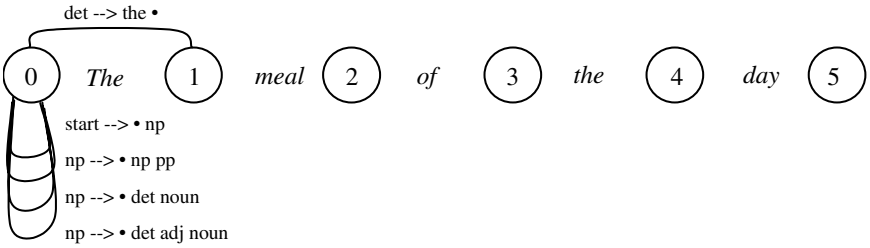


**Fig. 11.10.** The scanner accepts word *The* from the input.

**The Completer.** The scanner introduces new constituents under the form of parts of speech in the chart. The completer uses them to advance the dot of active arcs expecting them, and possibly complete the corresponding constituents. Once a constituent has been completed, it can in turn modify others expecting it in active arcs. The completer thus is applied to propagate modifications and to complete all possible arcs. It first determines which constituents are complete by looking for dots that have reached the end of a rule: `c --> x`$_1$ `...` `x`$_k$ •. The completer then searches all the active arcs expecting `c`, that is, the rules with a dot to the right of it: `lhs --> c`$_1$ `c`$_2$ `...` • `c ...` `c`$_n$, moves the dot over `c`: `lhs --> c`$_1$ `c`$_2$ `...` `c` •

. . . $c_n$, and inserts the new arc into the chart. It proceeds recursively from the parts of speech to all the possible higher-level constituents.

In our example, the only completed constituent is the part of speech `det`. The completer advances the dot over it in two active arcs and inserts them into the chart. It does not produce new completed constituents (Fig. 11.11).

```
np --> det • noun          [0, 1]
np --> det • adj noun      [0, 1]
```

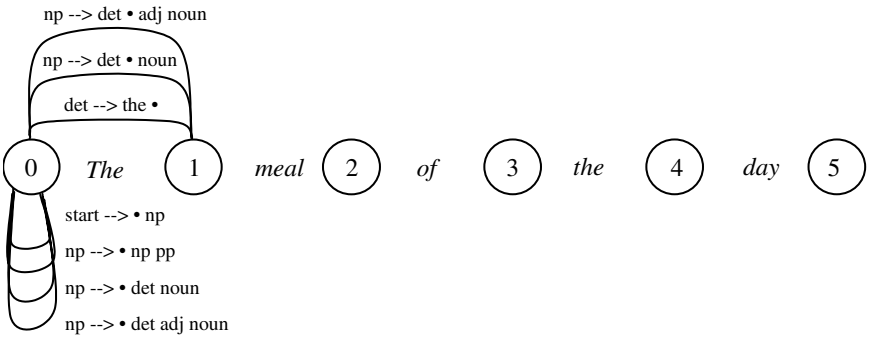


**Fig. 11.11.** The completer looks for completed constituents and advances the dot over them.

From node 1, the predictor is run again, but it does not yield new arcs. The scanner accepts word *meal*, advances the position to 2, and inserts

```
noun --> meal •    [1, 2]
```

as shown in Fig. 11.12.

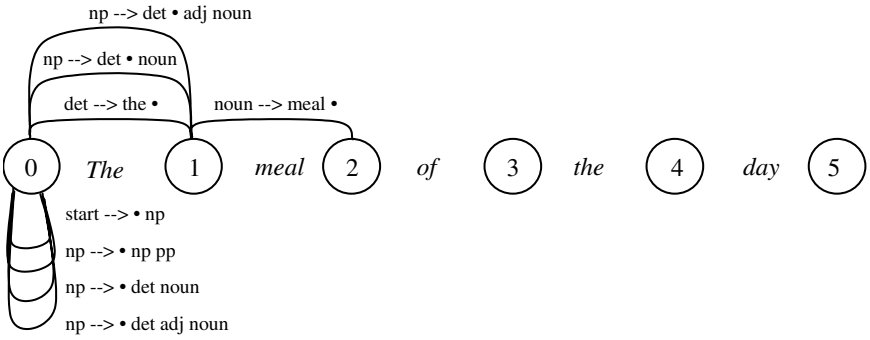


**Fig. 11.12.** Predictor and scanner are run with word *meal*.

At node 2, the completer can advance active arc

```
np --> det noun •      [0, 2]
```

and complete a higher-level constituent (Fig. 11.13).

```
np --> np • pp      [0, 2]
```

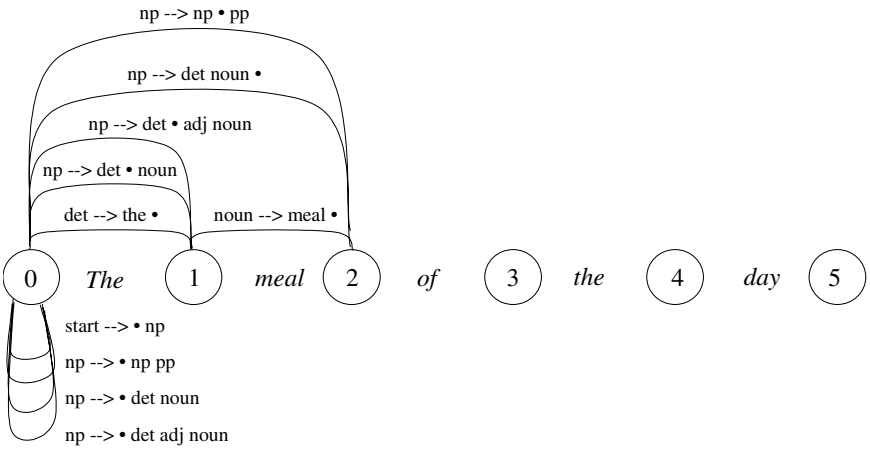


**Fig. 11.13.** The completer is run to produce new chart entries.

### 11.3.5 The Earley Algorithm in Prolog

To implement the algorithm in Prolog, we must first represent the chart. It consists of arcs such as

```
np --> np • pp     [0, 2]
```

which we represent as facts

```
arc(np, [np, '.', pp], 0, 2).
```

The start symbol is encoded as:

```
arc(start, ['.', np], 0, 0).
```

Although this data representation is straightforward from the description of dotted-rules, it is not efficient from the speed viewpoint. The arc representation can be improved easily, but an optimization may compromise clarity. We leave it as an exercise.

New arcs are stored in the chart, a list called `Chart`, using the `expand_chart/1` predicate, which checks first that the new entry is not already in the chart:

```
expand_chart([], Chart, Chart).
expand_chart([Entry | Entries], Chart, NewChart) :-
  \+ member(Entry, Chart),
  !,
  expand_chart(Entries, [Entry | Chart], NewChart).
expand_chart([_ | Entries], Chart, NewChart) :-
  expand_chart(Entries, Chart, NewChart).
```

The Earley algorithm is implemented by the predicate `earley_parser/2`. It uses five arguments: the input word sequence, the current position in the sequence `CurPos`, the final position `FinalPos`, the current chart, and the final chart. The `earley_parser` main rule consists of calling the `predictor`, `scanner`, and `completer` predicates through the $N + 1$ nodes of the chart:

```
earley_parser([], FinalPos, FinalPos, Chart, Chart):-
  !.
earley_parser(Words, CurPos, FinalPos, Chart,
      FinalChart) :-
  predictor(CurPos, Chart, PredChart),
  NextPos is CurPos + 1,
  scanner(Words, RestWords, CurPos, NextPos,
      PredChart, ScanChart),
  completer(NextPos, ScanChart, NewChart),
  !,
  earley_parser(RestWords, NextPos, FinalPos,
      NewChart, FinalChart).
```

The Earley algorithm is called by `parse/2`, which takes the word sequence `Words` and the start `Category` as arguments. The `parse/2` predicate initializes the chart with the start symbol and launches the parse. The parsing success corresponds to the presence of a completed start symbol, which is, in our example, the `arc(start, [np, '.'], 0, FinalNode)` fact in the Prolog database:

```
parse(Words, Category, FinalChart) :-
  expand_chart([arc(start, ['.', Category], 0, 0)],
      [], Chart),
  earley_parser(Words, 0, FinalPos, Chart,
      FinalChart),
  member(arc(start, [Category, '.'], 0, FinalPos),
      FinalChart).
```

Table 11.4 shows the transcription of `np` rules in Fig. 11.5 encoded as Prolog facts. It contains a small vocabulary to parse the phrase *The meal of the day*.

**The Predictor.** The predictor looks for rules to expand arcs from a current position (`CurPos`). To carry this out, `predictor/3` searches all the arcs containing the pattern: `[..., X, '.', CAT, ...]`, where `CAT` matches the left-hand side of a rule: `rule(CAT, RHS)`. This is compactly expressed using the

**Table 11.4.** Rules and vocabulary for the chart parser.

| Rules | Words | |
|---|---|---|
| `rule(np, [d, n]).` | `word(d, [the]).` | `word(prep, [of]).` |
| `rule(np, [d, a, n]).` | `word(n, [waiter]).` | `word(v, [brought]).` |
| `rule(np, [np, pp]).` | `word(n, [meal]).` | `word(v, [slept]).` |
| `rule(pp, [prep, np]).` | `word(n, [day]).` | |

`findall/3` built-in predicate. `predictor/3` then adds `arc(CAT, ['.' | RHS], CurPos, CurPos)` to the chart `NewChart`. `predictor/3` is run recursively until no new arc can be produced, that is, `NewChartEntries == []`. It then returns the predictor's chart `PredChart`.

```
predictor(CurPos, Chart, PredChart) :-
  findall(
    arc(CAT, ['.' | RHS], CurPos, CurPos),
    (
      member(arc(LHS, ACTIVE_RHS, InitPos, CurPos),
          Chart),
      append(B, ['.', CAT | E], ACTIVE_RHS),
      rule(CAT, RHS),
      \+ member(arc(CAT, ['.' | RHS], CurPos,CurPos),
          Chart)
    ),
    NewChartEntries),
  NewChartEntries \== [],
  expand_chart(NewChartEntries, Chart, NewChart),
  predictor(CurPos, NewChart, PredChart),
  !.
predictor(_, PredChart, PredChart).
```

Using chart entry `arc(np, [np, '.', pp], 0, 2)` and rules in Table 11.4:

```
?- predictor(2,[arc(np, [np, '.', pp], 0, 2)],Chart).
```

adds

```
arc(pp, ['.', prep, np], 2, 2)
```

to the `Chart` list.

**The Scanner.** The scanner gets a new word from the input and looks for active arcs that match its possible parts of speech to the right of the dot. The scanner stores the word with its compatible parts of speech as new chart entries. Again, we use `findall/3` to implement this search.

```
scanner([Word | Rest], Rest, CurPos, NextPos, Chart,
      NewChart) :-
  findall(
    arc(CAT, [Word, '.'], CurPos, NextPos),
    (
      word(CAT, [Word]),
      once((
        member(arc(LHS, ACTIVE_RHS, InitPos, CurPos),
          Chart),
        append(B, ['.', CAT | E], ACTIVE_RHS)))
    ),
    NewChartEntries),
  NewChartEntries \== [],
  expand_chart(NewChartEntries, Chart, NewChart).
```

**The Completer.** The completer looks for completed constituents, that is, for arcs with a dot at the end of the right-hand-side part of the rule. They correspond to `arc(LHS, COMPLETE_RHS, InitPos, CurPos)`, where `COMPLETE_RHS` matches `[..., X, '.']`. We use the goal `append(_, ['.'], COMPLETE_RHS)` to find them. The completer then searches arcs with a dot to the right of the `LHS` category of completed constituents: `[..., '.', LHS, ...]`, advances the dot over `LHS`: `[..., LHS, '.', ...]`, and stores the new arc with updated node positions. We use `findall/3` to implement the search, and `completer/3` is run recursively until there is no arc to complete.

```
completer(CurPos, Chart, CompChart) :-
  findall(
    arc(LHS2, RHS3, PrevPos, CurPos),
    (
      member(arc(LHS, COMPLETE_RHS, InitPos, CurPos),
          Chart),
      append(_, ['.'], COMPLETE_RHS),
      member(arc(LHS2, RHS2,PrevPos,InitPos), Chart),
      append(B, ['.', LHS | E], RHS2),
      append(B, [LHS, '.' | E], RHS3),
      \+ member(arc(LHS2, RHS3, PrevPos, CurPos),
          Chart)
    ),
    CompletedChartEntries),
  CompletedChartEntries \== [],
  expand_chart(CompletedChartEntries,Chart,NewChart),
  completer(CurPos, NewChart, CompChart),
  !.
completer(_, CompChart, CompChart).
```

**An Execution Example.** Table 11.5 shows the arcs added to the chart while parsing the phrase *The meal of the day*. The parser is queried by:

```
?- parse([the, meal, of, the, day], np, Chart).
```

Note that the `completer` calls at position 2 that completes `np`, and at position 5 that completes `np`, `pp`, and the starting goal `np`.

**Table 11.5.** Additions to the Prolog database.

| Module | New Chart Entries in the Database |
|---|---|
| | **Position 0** |
| start | arc(start, ['.', np], 0, 0) |
| predictor | arc(np, [., d, n], 0, 0), arc(np, [., d, a, n], 0, 0), arc(np, [., np, pp], 0, 0) |
| | **Position 1** |
| scanner | arc(d, [the, .], 0, 1) |
| completer | arc(np, [d, ., a, n], 0, 1), arc(np, [d, ., n], 0, 1) |
| predictor | [] |
| | **Position 2** |
| scanner | arc(n, [meal, .], 1, 2) |
| completer | arc(np, [d, n, .], 0, 2) |
| completer | arc(np, [np, ., pp], 0, 2), arc(start, [np, .], 0, 2) |
| predictor | arc(pp, [., prep, np], 2, 2) |
| | **Position 3** |
| scanner | arc(prep, [of, .], 2, 3) |
| completer | arc(pp, [prep, ., np], 2, 3) |
| predictor | arc(np, [., d, n], 3, 3), arc(np, [., d, a, n], 3, 3), arc(np, [., np, pp], 3, 3) |
| | **Position 4** |
| scanner | arc(d, [the, .], 3, 4) |
| completer | arc(np, [d, ., a, n], 3, 4), arc(np, [d, ., n], 3, 4) |
| predictor | [] |
| | **Position 5** |
| scanner | arc(n, [day, .], 4, 5) |
| completer | arc(np, [d, n, .], 3, 5) |
| completer | arc(np, [np, ., pp], 3, 5), arc(pp, [prep, np, .], 2, 5) |
| completer | arc(np, [np, pp, .], 0, 5) |
| completer | arc(np, [np, ., pp], 0, 5), arc(start, [np, .], 0, 5) |

### 11.3.6 The Earley Parser to Handle Left-Recursive Rules and Empty Symbols

The Earley parser handles left-recursive rules without looping infinitely. In effect, the predictor is the only place where the parser could be trapped into an infinite execution. This is avoided because before creating a new arc, the `predictor` predicate checks that it is not already present in the chart using the goal

```
\+ member(arc(CAT, ['.' | RHS], CrPos, CrPos), Chart)
```

So

```
start --> • np          [0, 0]
```

predicts

```
np --> • np pp          [0, 0]
np --> • det noun       [0, 0]
np --> • det adj noun   [0, 0]
```

but `np --> • np pp` predicts nothing more since all the possible arcs are already in the chart.

The Earley algorithm can also parse null constituents. It corresponds to examples such as *meals of the day*, where the determiner is encoded as `word(d, [])`. As we wrote it, the scanner would fail on empty symbols. We need to add a second rule to it to handle empty lists:

```
% The first scanner rule
scanner([Word | Rest], Rest, CurPos, NextPos, Chart,
      NewChart) :-
  findall(
    arc(CAT, [Word, '.'], CurPos, NextPos),
    (
      word(CAT, [Word]),
      once((
        member(arc(LHS, ACTIVE_RHS, InitPos, CurPos),
          Chart),
        append(B, ['.', CAT | E], ACTIVE_RHS)))
    ),
    NewChartEntries),
  NewChartEntries \== [],
  expand_chart(NewChartEntries, Chart, NewChart),
  !.

% The second rule to handle empty symbols
scanner(Words, Words, CurPos, NextPos, Chart,
      NewChart) :-
  findall(
    arc(CAT, [[], '.'], CurPos, NextPos),
```

```
    (
      word(CAT, []),
      once((
        member(arc(LHS, ACTIVE_RHS, InitPos, CurPos),
          Chart),
        append(B, ['.', CAT | E], ACTIVE_RHS)))
    ),
    NewChartEntries),
  NewChartEntries \== [],
  expand_chart(NewChartEntries, Chart, NewChart),
  !.
```

Let us add

```
word(d, []).
word(n, [meals]).
```

to the database to be able to parse *meals of the day*:

```
?- parse([meals, of , the, day], np, Chart).
```

## 11.4 Probabilistic Parsing of Context-Free Grammars

So far, parsing methods made no distinction between possible parse trees of an ambiguous sentence. They produced trees either through a systematic backtracking or simultaneously in a chart with the Earley algorithm. The reason is that the parsers considered all rules to be equal and tried them sequentially.

We know this is not the case in reality. Some rules describe very frequent structures, while others are rare. As a solution, a parser could try more frequent rules first, prefer certain rules when certain words occur, and rank trees in an order of likelihood. To do that, the parser can integrate statistics derived from bracketed corpora. Because annotation is done by hand, frequencies captured by statistics reflect preferences of human beings.

There are many possible **probabilistic parsing** techniques. They all aim at finding an optimal analysis considering a set of statistical parameters. A major difference between them corresponds to the introduction of lexical statistics or not – statistics on words as opposed to statistics on rules. We begin here with a description of non-lexicalized probabilistic context-free grammars, or PCFG.

## 11.5 A Description of PCFGs

A PCFG is a constituent context-free grammar where each rule describing the structure of a left-hand-side symbol is augmented with its probability $P(lhs \rightarrow rhs)$

**Table 11.6.** A small set of phrase-structure rules augmented with probabilities, $P$.

| Rules | $P$ | Rules | $P$ |
|---|---|---|---|
| `s --> np vp` | 0.8 | `det --> the` | 1.0 |
| `s --> vp` | 0.2 | `noun --> waiter` | 0.4 |
| `np --> det noun` | 0.3 | `noun --> meal` | 0.3 |
| `np --> det adj noun` | 0.2 | `noun --> day` | 0.3 |
| `np --> pronoun` | 0.3 | `verb --> bring` | 0.4 |
| `np --> np pp` | 0.2 | `verb --> slept` | 0.2 |
| `vp --> v np` | 0.6 | `verb --> brought` | 0.4 |
| `vp --> v np pp` | 0.1 | `pronoun --> he` | 1.0 |
| `vp --> v pp` | 0.2 | `prep --> of` | 0.6 |
| `vp --> v` | 0.1 | `prep --> to` | 0.4 |
| `pp --> prep np` | 1.0 | `pronoun --> he` | 1.0 |
| | | `adj --> big` | 1.0 |

(Charniak 1993). Table 11.6 shows a small set of grammar rules with imaginary probabilities.

According to figures in the table, the structure of a sentence consists 4 times out of 5 in a noun phrase and a verb phrase – $P(s \rightarrow np, vp) = 0.8$ – and 1 time out of 5 in a verb phrase – $P(s \rightarrow vp) = 0.2$. Such figures correspond in fact to conditional probabilities: knowing the left-hand-side symbol they describe proportions among the right-hand-side expansions. The probability could be rewritten then as

$$P(lhs \rightarrow rhs|lhs).$$

The sum of probabilities of all possible expansions of a left-hand-side symbol must be equal to 1.0.

Probabilities in Table 11.6 are fictitious and incomplete. A sentence has, of course, many more possible structures than those shown here. Real probabilities are obtained from syntactically bracketed corpora – treebanks. The probability of a given rule $lhs \rightarrow rhs_i$ is obtained by counting the number of times it occurs in the corpus and by dividing it by the count of all the expansions of symbol `lhs`.

$$P(lhs \rightarrow rhs_i|lhs) = \frac{Count(lhs \rightarrow rhs_i)}{\sum_j Count(lhs \rightarrow rhs_j)}.$$

Parsing with a PCFG is just the same as with a context-free grammar except that each tree is assigned with a probability. The probability for sentence $S$ to have the parse tree $T$ is defined as the product of probabilities attached to rules used to produce the tree:

$$P(T, S) = \prod_{rule(i)\text{producing}T} P(rule(i)).$$

Let us exemplify probabilistic parsing for an ambiguous sentence using the grammar in Table 11.6. *Bring the meal of the day* has two possible parse trees, as shown in Table 11.7. We consider trees up to the verb phrase symbol only.

**Table 11.7.** Possible parse trees for *Bring the meal of the day*.

| Parse trees |
|---|
| T1: `vp(verb(bring),`<br>`np(np(det(the), noun(meal)),`<br>`pp(prep(of), np(det(the), noun(day)))))` |
| T2: `vp(verb(bring),`<br>`np(np(det(the), noun(meal))),`<br>`pp(prep(of), np(det(the), noun(day))))` |

The probability of $T_1$ is defined as (Fig. 11.14):

$P(T_1, \text{Bring the meal of the day}) =$
$P(vp \rightarrow v, np) \times P(v \rightarrow Bring) \times P(np \rightarrow np, pp) \times$
$P(np \rightarrow det, noun) \times P(det \rightarrow the) \times P(noun \rightarrow meal) \times$
$P(pp \rightarrow prep, np) \times P(prep \rightarrow of) \times P(np \rightarrow det, noun) \times$
$P(det \rightarrow the) \times P(noun \rightarrow day) =$
$0.6 \times 0.4 \times 0.2 \times 0.3 \times 1.0 \times 0.3 \times 1.0 \times 0.6 \times 0.3 \times 1.0 \times 0.3 = 0.00023328,$

and that of $T_2$ as (Fig. 11.15) as:

$P(T_2, \text{Bring the meal of the day}) =$
$P(vp \rightarrow v, np, pp) \times P(v \rightarrow Bring) \times P(np \rightarrow det, noun) \times$
$P(det \rightarrow the) \times P(noun \rightarrow meal) \times P(pp \rightarrow prep, np) \times P(prep \rightarrow of) \times$
$P(np \rightarrow det, noun) \times P(det \rightarrow the) \times P(noun \rightarrow day) =$
$0.1 \times 0.4 \times 0.3 \times 1.0 \times 0.3 \times 1.0 \times 0.6 \times 0.3 \times 1.0 \times 0.3 = 0.0001944.$

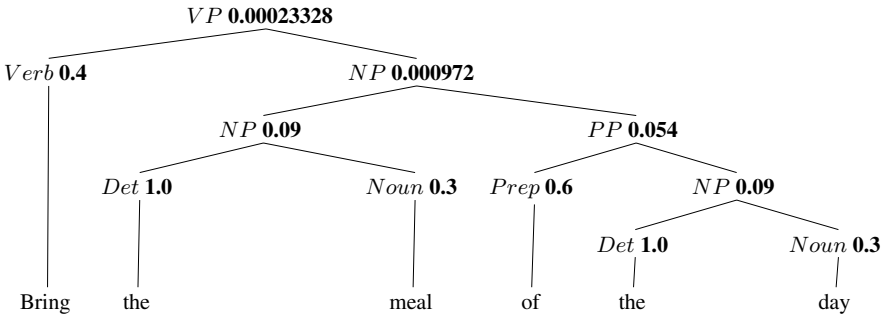


**Fig. 11.14.** Parse tree $T_1$ with nodes annotated with probabilities.

$T_1$ has a probability higher than that of $T_2$ and then corresponds to the most likely parse tree. Thus PCFG would properly disambiguate among alternative structures for this sentence. However, we can notice that PCFGs are certainly not flawless because they would not properly rank trees of *Bring the meal to the table*.

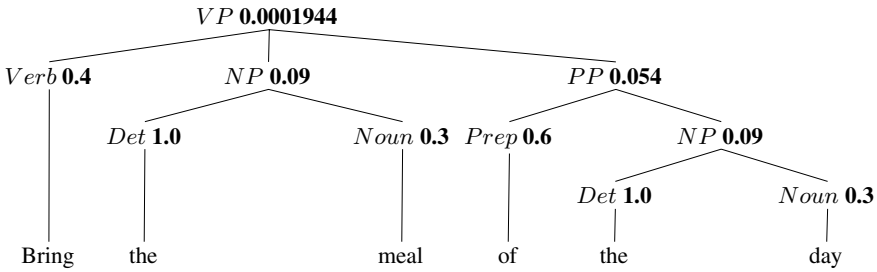


**Fig. 11.15.** Parse tree $T_2$ with nodes annotated with probabilities.

### 11.5.1 The Bottom-up Chart

Figures 11.14 and 11.15 show a calculation of parse tree probabilities using a bottom-up approach. Although it is possible to use other types of parsers, this strategy seems the most natural because it computes probabilities as it assembles partial parses. In addition, a chart would save us many recalculations. We will combine these techniques to build a probabilistic context-free parser. We introduce them in two steps. First, we present a symbolic bottom-up chart parser also known as the Cocke–Younger–Kasami (CYK) algorithm (Kasami 1965). We then extend it to probabilistic parsing in a next section.

The CYK algorithm uses grammars in Chomsky normal form (CNF, Chap. 10) where rules are restricted to two forms:

```
lhs --> rhs1, rhs2.
lhs --> [terminal_symbol].
```

However, the CYK algorithm can be generalized to any type of grammar (Graham et al. 1980).

Let $N$ be the length of the sentence. The idea of the CYK parser is to consider constituents of increasing length from the words – length 1 – up to the sentence – length $N$. In contrast to the Earley parser, the CYK algorithm stores completely parsed constituents in the chart. It proceeds in two steps. The first step annotates the words with all their possible parts of speech. Figure 11.16 shows the result of this step with the sentence *Bring the meal of the day*. It results in chart entries such as `arc(v, [bring, '.'], 0, 1)`, `arc(det, [the, '.'], 1, 2)`, etc. This first step is also called the base case.

The second step considers contiguous pairs of chart entries that it tries to reduce in a constituent of length $l$, $l$ ranging from 2 to $N$. Considering rule `lhs --> rhs1, rhs2`, the parser searches couples of arcs corresponding to `arc(rhs1, [..., '.'], i, k)` and `arc(rhs2, [..., '.'], k, j)` such that $i < k < j$ and $j - i = l$. It adds then a new arc: `arc(lhs, [rhs1, rhs2, '.'], i, j)` to the chart. Since constituents of length 2, 3, 4, ... $N$ are built in that order, it ensures that all constituents of length less than $l$ have already been built. This second step is called the recursive case.

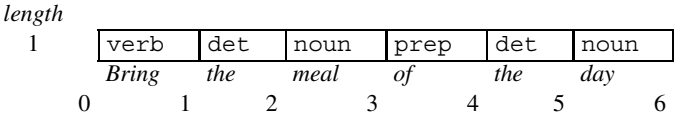


**Fig. 11.16.** Annotation of the words with their possible part of speech. Here words are not ambiguous.

Let us consider constituents of length 2 of our example. We can add two noun phrases that we insert in the second row, as shown in Fig. 11.17. They span nodes 1–3 and 4–6. Since their length is 2, no constituent can start in cell 5–6, otherwise it would overflow the array. We insert the symbol "—" in the corresponding cell. This property is general for any constituent of length $l$ and yields a triangular array.

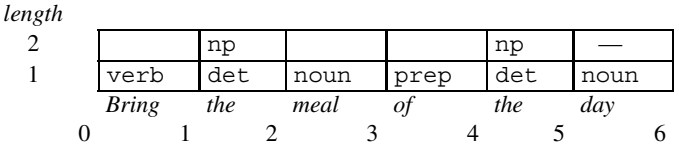


**Fig. 11.17.** Constituents of length 1 and 2.

The parse is complete and successful when length $N$, here 6, has been reached with the start symbol. Figure 11.18 shows constituents of length 3, and Fig. 11.19 shows the completed parse, where constituents are indexed vertically according to their length.

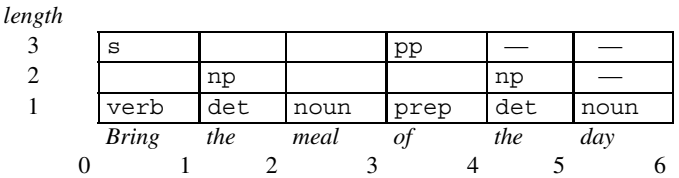


**Fig. 11.18.** Constituent of lengths 1, 2, and 3.

### 11.5.2 The Cocke–Younger–Kasami Algorithm in Prolog

From the algorithm description, the Prolog implementation is relatively straightforward. We use two predicates to carry out the base case and the recursive case: `tag_words/5` and `cyk_loop/4`.

Since we use a CNF, we need to rewrite some rules in Table 11.6:

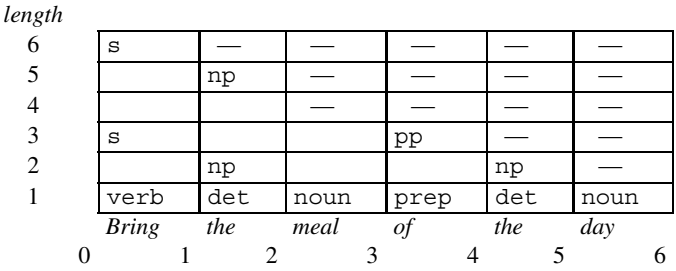


**Fig. 11.19.** The completed parse.

- `rule(np, [d, a, n])` is rewritten into `rule(np, [det, np])` and `rule(np, [a, n])`.
- `rule(vp, [v, np, pp])` is rewritten into `rule(vp, [vp, pp])` and `rule(vp, [v, np])`.
- `rule(s, [vp])` is rewritten into `rule(s, [vp, pp])` and `rule(s, [v, np])`.
- `rule(vp, [v])` is rewritten into `word(vp, [brought])`, `word(vp, [bring])`, and `word(vp, [slept])`.
- `rule(np, [pronoun])` is rewritten into `word(np, [he])`.

The parsing predicate `parse/2` consists of tagging the words (the base case) and calling the reduction loop (the recursive case).

```
parse(Sentence, Chart) :-
  tag_words(Sentence, 0, FinalPosition,[],WordChart),
  cyk_loop(2, FinalPosition, WordChart, Chart).
```

`tag_words/3` tags the words with their possible parts of speech and adds the corresponding arcs using the `expand_chart/1` predicate.

```
tag_words([], FinalPos, FinalPos, Chart, Chart).
tag_words([Word | Rest], Pos, FinalPos, Chart,
      WordChart) :-
  NextPos is Pos + 1,
  findall(
    arc(LHS, [Word, '.'], Pos, NextPos),
    word(LHS, [Word]),
    ChartEntries),
  expand_chart(ChartEntries, Chart, NewChart),
  tag_words(Rest, NextPos, FinalPos, NewChart,
      WordChart).
```

`cyk_loop/4` implements the recursive case. It proceeds from length 2 to the sentence length and attempts to reduce constituents using `inner_loop/5`. The new constituents are added to the chart using `expand_chart/3`.

```
cyk_loop(FinalPos, FinalPos, Chart, FinalChart) :-
  inner_loop(0, FinalPos, FinalPos,Chart,FinalChart).
cyk_loop(Length, FinalPos, Chart, FinalChart) :-
  inner_loop(0, Length, FinalPos, Chart, ILChart),
  NextLength is Length + 1,
  cyk_loop(NextLength, FinalPos, ILChart,FinalChart).

inner_loop(StartPos, Length, FinalPos, Chart,Chart):-
  FinalPos < StartPos + Length.
inner_loop(StartPos, Length, FinalPos, Chart,
      ILChart) :-
  EndPos is StartPos + Length,
  findall(
    arc(LHS3, [LHS1, LHS2, '.'], StartPos, EndPos),
    (
      member(arc(LHS1, RHS1,StartPos,MidPos), Chart),
      member(arc(LHS2, RHS2, MidPos, EndPos), Chart),
      StartPos < MidPos,
      MidPos < EndPos,
      rule(LHS3, [LHS1, LHS2])
    ),
    ChartEntries),
  expand_chart(ChartEntries, Chart, NewChart),
  NextStartPos is StartPos + 1,
  inner_loop(NextStartPos, Length, FinalPos,
      NewChart, ILChart).
```

### 11.5.3 Adding Probabilities to the CYK Parser

Considering sentence $S$, the parser has to find the most likely tree $T$ defined as the maximum probability

$$T(S) = \arg\max P(T).$$

Let us suppose that sentence $S$ consists of constituents $A$ and $B$: $S \rightarrow A, B$. The most likely parse tree corresponds to that yielding the maximum probability of both $A$ and $B$. This is valid recursively for substructures of $A$ and $B$ down to the words.

To obtain most likely constituents for any given length, we need to maintain an array that stores the maximum probability for all the possible constituents spanning all the word intervals $i...j$ in the chart. In other words, that means that if there are two or more competing constituents with the same left-hand-side label spanning $i...j$, the parser retains the maximum and discards the others. Let $lhs$ be the constituent label and $\pi(i, j, lhs)$ this probability.

The base case initializes the algorithm with part of speech probabilities:

$$\pi(i, i+1, part\_of\_speech \rightarrow word).$$

The recursive case maintains the probability of the most likely structure of `lhs`. It corresponds to

$$\pi(i, j, lhs) = \max(\pi(i, k, rhs_1) \times \pi(k, j, rhs_2) \times P(lhs \rightarrow rhs_1, rhs_2),$$

where the maximum is taken over all the possible values of $k$ with $i < k < j$ and all possible values of $rhs_1$, $rhs_2$ with $lhs \rightarrow rhs_1, rhs_2$ in the grammar.

## 11.6 Parser Evaluation

### 11.6.1 Constituency-Based Evaluation

We have a variety of techniques to evaluate parsers. The PARSEVAL measures (Black et al. 1991) are the most frequently cited for constituent parsing. They take a manually bracketed treebank as the reference – the gold standard – and compare it to the results of a parser.

PARSEVAL uses a metric similar to that of information extraction, that is, recall and precision. Recall is defined as the number of correct constituents generated by the parser, i.e., exactly similar to that of the manually bracketed tree, divided by the number of constituents of the treebank. The precision is the number of correct constituents generated by the parser divided by the total number of constituents – wrong and correct ones – generated by the parser.

$$\text{Recall} = \frac{\text{Number of correct constituents generated by the parser}}{\text{Number of constituents in the manually bracketed corpus}}.$$

$$\text{Precision} = \frac{\text{Number of correct constituents generated by the parser}}{\text{Total number of constituents generated by the parser}}.$$

A third metric is the number of crossing brackets. It corresponds to the number of constituents produced by the parser that overlap constituents in the treebank. Table 11.8 shows two possible analyses of *Bring the meal of the day* with crossing brackets between both structures. The number of crossing brackets gives an idea of the compatibility between structures and whether they can be combined into a single structure.

**Table 11.8.** Bracketing of order *Bring the meal of the day* and crossing brackets.

| Bracketing | Crossing brackets |
|---|---|
| (((bring) (the meal)) (of the day)) | ( ) ( ) |
| ((bring) ((the meal) (of the day))) | ( ) ( ) |

### 11.6.2 Dependency-Based Evaluation

Lin (1995) proposed another evaluation metric based on dependency trees. It also considers a treebank that it compares to the output of a parser. The error count is the number of words that are assigned a wrong head (governor). Figure 11.20 shows a reference dependency tree and a possible parse of *Bring the meal to the table*. The error count is 1 out of 6 links and corresponds to the wrong attachment of *to*. Lin (1995) also described a method to adapt this error count to constituent structures. This error count is probably simpler and more intuitive than the PARSEVAL metrics.

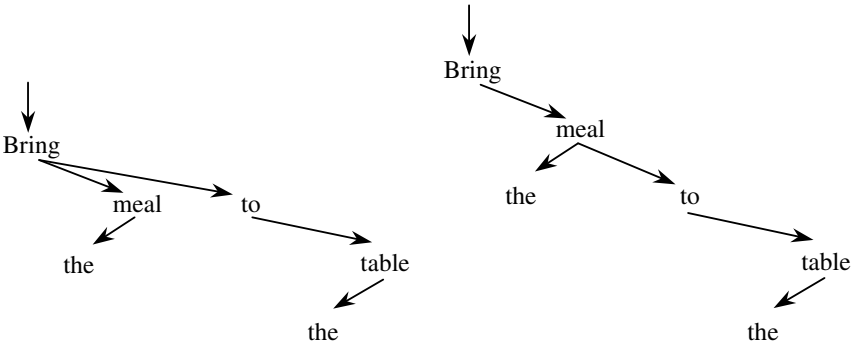


**Fig. 11.20.** Evaluation of dependency trees: The reference dependency tree (left) and a possible parse output (right).

### 11.6.3 Performance of PCFG Parsing

PCFGs rank the possible analyses. This enables us to select the most probable parse tree and to evaluate it. Charniak (1997) reports approximately 70% recall and 75% precision for this parsing method.

In terms of accuracy, PCFG parsing does not show the best performances. This is mainly due to its poor use of lexical properties. An example is given with prepositional-phrase attachment. While prepositional phrases attach to the preceding noun phrase 6 to 7 times out of 10 on average, there are specific lexical preferences. Some prepositions attach more often to verbs in general, while others attach to nouns. There are also verb/preposition or noun/preposition couples, showing strong affinities.

Let us exhibit them with orders

*Bring the meal to the table*

and

*Bring the meal of the day*

for which a parser has to decide where to attach prepositional phrases *to the table* and *of the day*. Alternatives are the verb *Bring* and the noun phrase *the meal*. Prepositional phrases headed by *of* attach systematically to the preceding noun phrase, here *the meal*, while *to* attaches here to the verb. Provided that part-of-speech annotation of both sentences is the same, the ratio

$$\frac{P(T1|\text{Bring the meal of the day})}{P(T2|\text{Bring the meal of the day})} = \frac{P(T1|\text{Bring the meal to the table})}{P(T2|\text{Bring the meal to the table})}, \\ = \frac{P(vp \rightarrow v,np) \times P(np \rightarrow np,pp)}{P(vp \rightarrow v,np,pp)}$$

depends only on rule probabilities and not on the lexicon. In our example, the PCFG does not take the preposition value into account: any prepositional phrase would always attach to the preceding noun, thus accepting an average error rate of 30 to 40%

## 11.7 Parsing Dependencies

Parsing dependencies consists of finding links between governors – or heads – and dependents – one word being the root of the sentence (Fig. 11.21). In addition, each link can be annotated with a grammatical function as shown in Table 11.9.

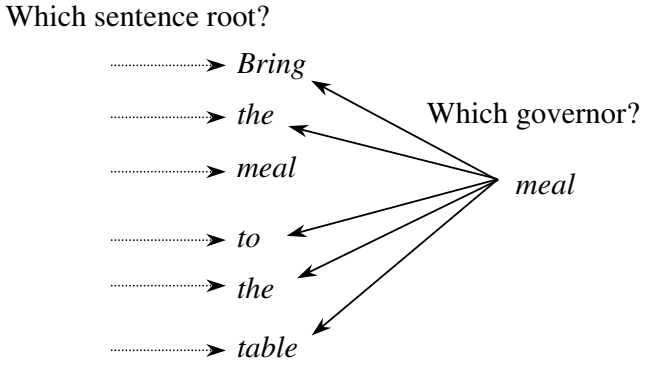


**Fig. 11.21.** Possible sentence roots and governors for word meal. There are $N$ possible roots and each remaining word has theoretically $N-1$ possible governors.

The dependency graph of a sentence is compactly expressed by the sequence

$$D = \{< \text{Head}(1), \text{Rel}(1) >, < \text{Head}(2), \text{Rel}(2) >, ..., < \text{Head}(n), \text{Rel}(n) >\},$$

which maps each word of index $i$ to its head, Head$(i)$, with relation Rel$(i)$. The head is defined by its position index in the sentence. The sentence *Bring the meal to the table* would yield

**Table 11.9.** A representation of dependencies due to Lin (1995). Direction gives the direction of the governor. Symbol '>' means that it is the first occurrence of the word to the right, '>>' the second one, etc. '*' denotes the root of the sentence.

| Word position | Word | Direction | Governor | Governor position | Function |
|---|---|---|---|---|---|
| 1 | *Bring* | * | | Root | Main verb |
| 2 | *the* | > | *meal* | 3 | Determiner |
| 3 | *meal* | < | *Bring* | 1 | Object |
| 4 | *to* | < | *Bring* | 1 | Location |
| 5 | *the* | > | *table* | 6 | Determiner |
| 6 | *table* | < | *to* | 4 | Prepositional complement |

$$D = \{< nil, \text{root} >, < 3, \text{det} >, < 1, \text{object} >, < 1, \text{loc} >, < 6, \text{det} >, < 4, \text{pcomp} >\},$$

where $< nil, \text{root} >$ denotes the root of the dependency graph.

There is a large range of techniques to parse dependencies. We introduce some of them in an order of increasing complexity, where more elaborate methods produce, in general, better results. We will begin with dependency rules, then shift–reduce for dependencies, constraint satisfaction, and finally statistical lexical dependencies.

### 11.7.1 Dependency Rules

Writing dependency rules or $D$-rules consists in describing possible dependency relations between word categories (Covington 1990): typically a head part of speech to a dependent part of speech (Fig. 11.22).

```
1. determiner ← noun.   4. noun ← verb.
2. adjective ← noun.    5. preposition ← verb.
3. preposition ← noun.  6. verb ← root.
```

**Fig. 11.22.** Examples of $D$-rules.

These rules mean that a determiner can depend on a noun (1) (or that a noun can be the head of a determiner), an adjective can depend on a noun (2) and a noun can depend on a verb (4). The rules express ambiguity. A preposition can depend either on a verb (5) as in *Bring the meal to the table* or on a noun (3) as in *Bring the meal of the day*. Finally, rule 6 means that a verb can be the root of the sentence.

$D$-rules are related to one or more functions. The first rule expresses the determinative function, the second one is an attributive function, and the third rule can be a subject, an object, or an indirect object function. Using a unification-based formalism, rules can encapsulate functions, as in:

$$\begin{bmatrix} category : noun \\ number : N \\ person : P \\ case : nominative \end{bmatrix} \leftarrow \begin{bmatrix} category : verb \\ number : N \\ person : P \end{bmatrix}$$

which indicates that a noun marked with the nominative case can depend on a verb. In addition, the noun and verb share the person and number features. Unification-based $D$-rules are valuable because they can easily pack properties into a compact formula: valence, direction of dependency relation (left or right), lexical values, etc. (Covington 1989; Koch 1993).

### 11.7.2 Extending the Shift–Reduce Algorithm to Parse Dependencies

Once we have written the rules, we need an algorithm to run a grammar on sentences. Nivre (2003) proposed a dependency parser that creates a graph that he proved to be both projective and acyclic. The parser is an extension to the shift–reduce algorithm. It uses oriented $D$-rules to represent left, $LEX(n') \leftarrow LEX(n)$, and right, $LEX(n) \rightarrow LEX(n')$, dependencies.

As with the regular shift–reduce, Nivre's parser uses a stack $S$ and a list of input words $W$. However, instead of finding constituents, it builds a set of arcs $A$ representing the graph of dependencies. The triplet $\langle S, W, A \rangle$ represents the parser state.

Nivre's parser uses two operations in addition to shift and reduce: left-arc and right-arc:

- Left-arc adds an arc $n' \rightarrow n$ from the next input word $n'$ to the top of the stack $n$ and reduces $n$ from the top of the stack. The grammar must contain the rule $LEX(n') \leftarrow LEX(n)$ and there must not be an arc $n'' \rightarrow n$ already in the graph.
- Right-arc adds an arc $n \rightarrow n'$ from the top of the stack $n$ to the next input word $n'$ and pushes $n'$ on the top of the stack. The grammar must contain the rule $LEX(n) \rightarrow LEX(n')$ and there must not be an arc $n'' \rightarrow n'$ already in the graph.

Table 11.10 shows the operations and their conditions.

The parsing algorithm is simple. The first step uses a POS tagger to annotate each word of the input list with its part of speech. Then, the parser applies a sequence of operations: left-arc, right-arc, reduce, and shift. Nivre (2003) experimented with three parsing strategies that depended on the operation priorities. The two first ones are:

- The parser uses constant priorities for the operations: left-arc > right-arc > reduce > shift.
- The second parser uses the constant priorities left-arc > right-arc and a rule to resolve shift/reduce conflicts. If the top of the stack can be a transitive head of the next input word, then shift; otherwise reduce.

**Table 11.10.** The parser transitions where $W$ is the initial word list; $I$, the current input word list; $A$, the graph of dependencies; and $S$, the stack.

| Actions | Parser actions | Conditions |
|---|---|---|
| Initialization | $\langle nil, W, \emptyset \rangle$ | |
| Termination | $\langle S, nil, A \rangle$ | |
| Left-arc | $\langle n\|S, n'\|I, A \rangle \rightarrow \langle S, n'\|I, A \cup \{(n', n)\} \rangle$ | $LEX(n) \leftarrow LEX(n') \in R$<br>$\neg\exists n''(n'', n) \in A$ |
| Right-arc | $\langle n\|S, n'\|I, A \rangle \rightarrow \langle n'\|n\|S, I, A \cup \{(n, n')\} \rangle$ | $LEX(n) \rightarrow LEX(n') \in R$<br>$\neg\exists n''(n'', n') \in A$ |
| Reduce | $\langle n\|S, I, A \rangle \rightarrow \langle S, I, A \rangle$ | $\exists n'(n', n) \in A$ |
| Shift | $\langle S, n\|I, A \rangle \rightarrow \langle n\|S, I, A \rangle$ | |

Nivre's parser can be extended to predict the parser's sequence of actions and to handle nonprojectivity. It then uses probabilities derived from a hand-annotated corpus and a machine-learning algorithm (memory-based learning or support vector machines). The extensions are described in Nivre and Scholz (2004) and Nivre and Nilsson (2005).

### 11.7.3 Nivre's Parser in Prolog

Before we start to write the parser, we need to represent the dependency rules, the arcs, and the input sentence.

- We encode the $D$-rules with a `d/4` predicate that describes the function and the dependency direction:
```
%drule(+HeadPOS, +DependentPOS, +Function,+Direction)
drule(noun, determiner, determinative, left).
drule(noun, adjective, attribute, left).
drule(verb, noun, subject, left).
drule(verb, pronoun, subject, left).
drule(verb, noun, object, right).
drule(verb, pronoun, object, right).
drule(verb, prep, adv, _).
drule(noun, prep, pmod, right).
drule(prep, noun, pcomp, right).
```
- We store the words and their position using the predicate `w(word, position)`, and we represent the sentence *The waiter brought the meal* as the list `[w(the, 1), w(waiter, 2), w(brought, 3), w(the, 4), w(meal, 5)]`.
- Finally, we store the dependency arcs as `d(Head, Dependent, Function)`.

The parser code is an extension of the regular shift–reduce. The `shift_reduce/2` predicate takes the sentence as input and returns the graph of dependencies.

To implement it, we need an auxiliary `shift_reduce/4` predicate with two additional variables: a stack and graph where we will store the current arcs. This initial value of the stack as well as the graph is the empty list.

```
% shift_reduce(+Sentence, -Graph)

shift_reduce(Sentence, Graph) :-
  shift_reduce(Sentence, [], [], Graph).
```

Then `shift_reduce/4` consists of four predicates: `left_arc/6`, `right_arc/6`, `shift/4`, and `reduce/3`. They are applied in the order just listed until the sentence is an empty list:

```
% shift_reduce(+Words, +Stack, +CurrentGraph,
%   -FinalGraph)

shift_reduce([], _, Graph, Graph).
shift_reduce(Words, Stack, Graph, FinalGraph) :-
  left_arc(Words, Stack, NewStack, Graph, NewGraph),
  write('left arc'), nl,
  shift_reduce(Words, NewStack, NewGraph,FinalGraph).
shift_reduce(Words, Stack, Graph, FinalGraph) :-
  right_arc(Words, NewWords, Stack, NewStack, Graph,
    NewGraph),
  write('right arc'), nl,
  shift_reduce(NewWords, NewStack, NewGraph,
    FinalGraph).
shift_reduce(Words, Stack, Graph, FinalGraph) :-
  reduce(Stack, NewStack, Graph),
  write(reduce), nl,
  shift_reduce(Words, NewStack, Graph, FinalGraph).
shift_reduce(Words, Stack, Graph, FinalGraph) :-
  shift(Words, NewWords, Stack, NewStack),
  write(shift),nl,
  shift_reduce(NewWords, NewStack, Graph,FinalGraph).
```

The `shift/4` predicate removes the first word from the word list currently being parsed and puts it on the top the stack. Here appends it to the end of the `Stack` list – to produce a `NewStack`

```
% shift(+WordList, -NewWordList, +Stack, -NewStack)

shift([First | Words], Words, Stack, [First| Stack]).
```

The `reduce/3` predicate reduces the `Stack` provided that the word has a head already in the graph.

```
% reduce(+Stack, -NewStack, +Graph)

reduce([w(Top, PosT) | Stack], Stack, Graph) :-
  member(d(_, w(Top, PosT), _), Graph).
```

The `right_arc/6` predicate adds an arc to the graph linking the top of the stack to the first word of the list with the conditions described in Table 11.9.

```
% right_arc(+WordList, -NewWordList, +Stack,
%  -NewStack, +Graph, -NewGraph)

right_arc([w(First, PosF) | Words], Words,
    [w(Top, PosT) |
    Stack], [w(First, PosF), w(Top, PosT) | Stack],
    Graph, [d(w(Top, PosT), w(First, PosF),
    Function) | Graph]) :-
  word(First, FirstPOS),
  word(Top, TopPOS),
  drule(TopPOS, FirstPOS, Function, right),
  \+ member(d(_, w(First, PosF), _), Graph).
```

The `left_arc/6` predicate adds an arc to the graph linking the first word of the list to the top of the stack with the conditions described in Table 11.9.

```
% left_arc(+WordList, +Stack, -NewStack, +Graph,
%   -NewGraph)

left_arc([w(First, PosF) |_], [w(Top, PosT) | Stack],
    Stack, Graph, [d(w(First, PosF), w(Top, PosT),
    Function) | Graph]) :-
  word(First, FirstPOS),
  word(Top, TopPOS),
  drule(FirstPOS, TopPOS, Function, left),
  \+ member(d(_, w(Top, PosT), _), Graph).
```

Let us use the words:

```
%word(+Word, +PartOfSpeech)

word(waiter, noun).
word(meal, noun).
word(the, determiner).
word(a, determiner).
word(brought, verb).
word(ran, verb).
```

Applying the parser to *The waiter brought the meal* yields:

```
?- shift_reduce([w(the, 1), w(waiter, 2),
w(brought, 3), w(the, 4), w(meal, 5)], G).

shift
left arc
shift
left arc
shift
shift
left arc
right arc

G = [d(w(brought, 3), w(meal, 5), object),
d(w(meal, 5), w(the, 4), determinative),
d(w(brought, 3), w(waiter, 2), subject),
d(w(waiter, 2), w(the, 1), determinative)]
```

### 11.7.4 Finding Dependencies Using Constraints

Another strategy to parse dependencies is to use constraints in addition to $D$-rules. The parsing algorithm is then framed as a constraint satisfaction problem.

Constraint dependency parsing annotates words with dependencies and functions tags. It then applies a set of constraints to find a tag sequence consistent with all the constraints. Some methods generate all possible dependencies and then discard inconsistent ones (Maruyama 1990, Harper et al. 1999). Others assign one single dependency per word and modify it (Tapanainen and Järvinen 1997).

Let us exemplify a method inspired by Harper et al. (1999) with the sentence *Bring the meal to the table*. Table 11.11 shows simplified governor and function assignments compatible with a word's part of speech.

**Table 11.11.** Possible functions according to a word's part of speech.

| Parts of speech | Possible governors | Possible functions |
|---|---|---|
| Determiner | Noun | det |
| Noun | Verb | object, iobject |
| Noun | Prep | pcomp |
| Verb | Root | root |
| Prep | Verb, noun | mod, loc |

The first step generates all possible governor and function tags. Using Table 11.11, tagging yields:

| Words | *Bring* | *the* | *meal* | *to* | *the* | *table* |
|---|---|---|---|---|---|---|
| Position | 1 | 2 | 3 | 4 | 5 | 6 |
| Part of speech | verb | det | noun | prep | det | noun |
| Possible tags | <nil, root> | <3, det> | <4, pcomp> | <3, mod> | <3, det> | <4, pcomp> |
| | | <6, det> | <1, object> | <1, loc> | <6, det> | <1, object> |
| | | | <1, iobject> | | | <1, iobject> |

Then, a second step applies and propagates the constraint rules. It checks that the constraints do not conflict and enforces the consistency of tag sequences. Rules for English describe for instance, adjacency (links must not cross), function uniqueness (there is only one subject, one object, one indirect object), and topology:

- A determiner has its governor to its right-hand side.
- A subject has its governor to its right-hand side when the verb is at the active form.
- An object and an indirect object have their governor to their left-hand side (active form).
- A prepositional complement has its governor to its left-hand side.

Applying this small set of rules discards some wrong tags but leave some ambiguity.

| Words | *Bring* | *the* | *meal* | *to* | *the* | *table* |
|---|---|---|---|---|---|---|
| Position | 1 | 2 | 3 | 4 | 5 | 6 |
| Part of speech | verb | det | noun | prep | det | noun |
| Possible tags | <nil, root> | <3, det> | <1, object> | <3, mod> | <6, det> | <4, pcomp> |
| | | | <1, iobject> | <1, loc> | | |

### 11.7.5 Parsing Dependencies Using Statistical Techniques

Using constraints and statistics, it is possible to build a dependency parser that reaches very high rates of accuracy. Here, we introduce an algorithm derived from that of Collins (1996, 1999, 2003) where parsing corresponds to finding the most likely dependency tree $DT$ given a sentence $S$. This can be formulated as

$$DT_{best} = \arg\max P(DT\,|S)\ .$$

Collins' statistical dependency parser uses a cascade of three statistical modules: a part-of-speech tagger, a noun group detector, and a dependency model. Inside a noun group, dependency links are not ambiguous and can be determined in a fairly straightforward fashion. Therefore the algorithm represents noun groups by their main nouns and thus decreases its complexity. It takes the reduced tagged sentence as an input.

Collins' algorithm uses a statistical model composed of two terms: one corresponding to noun groups, $NG$, and the other to dependencies, $D$. It can be rewritten as:

$$P(DT\,|S) = P(NG, D\,|S) = P(NG\,|S) \times P(\ D\,|S, NG)\ .$$

We will focus here on dependencies only and for sake of simplicity, we will suppose that noun groups are perfectly detected. The input of the parser is then a reduced sentence whose words are tagged for part of speech:

$$S =< (w_1, t_1), (w_2, t_2), ..., (w_n, t_n) > .$$

For *Bring the meal to the table*, this yields

| Position | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Word | *Bring* | *meal* | *to* | *table* |
| POS | verb | noun | prep | noun |

A dependency link for word $i$ is represented as $AF(i) = (h_i, R_i)$, where $h_i$ is the position of the governor and $R_i$ is the relation linking both words. In our example, $AF(2) = (1, object)$ means that the governor of *meal* is word 1, i.e., *bring*, with the object relation. According to the model, the most likely parse is the maximum of

$$P(D|NG, S) = \prod_{j=1}^{m} P(AF(j)|NG, S).$$

Probability estimates are obtained using a dependency treebank. Let $C(R, \langle a, b\rangle\ , \langle c, d\rangle\ )$ be the number of times dependency relation $R$ links words $a$ and $c$ with respective part-of-speech tags $b$ and $d$ in a same sentence in the training corpus and $C(\langle a, b\rangle\ , \langle c, d\rangle\ )$, the number of times words $a$ and $c$ with part-of-speech tags $b$ and $d$ are seen in a same sentence in the training corpus. $F(R|\ \langle a, b\rangle\ , \langle c, d\rangle\ )$ is the probability that $\langle a, b\rangle$ is a dependent of $\langle c, d\rangle$ with relation $R$ given that $\langle a, b\rangle$ and $\langle c, d\rangle$ appear in the same sentence. Its estimate is defined by:

$$F(R|\ \langle a, b\rangle\ , \langle c, d\rangle\ ) = \frac{C(R, \langle a, b\rangle\ , \langle c, d\rangle\ )}{C(\langle a, b\rangle\ , \langle c, d\rangle\ )}.$$

Dependencies are computed in a stochastic way. Link determination between words corresponds to the maximum of

$$\prod_{j=1}^{m} F(R_j|\ \langle w_j,\ t_j\rangle\ , \langle w_{h_j},\ t_{h_j}\rangle\ ),$$

where $m$ is the total number of words in the sentence, $F$ represents the probability of a dependency link $R_j$ between word $w_j$ of index $j$ with part-of-speech tag $t_j$ and word $w_{h_j}$ with index $h_j$ and part of speech $t_{h_j}$.

Since there is a great likelihood of sparse data – the figure of counts $C(\langle w_i,\ t_i\rangle\ , \langle w_j,\ t_j\rangle\ )$ are too low or equal to 0 – a combination of estimates has to be devised. Collins (1999) proposed considering, ranging from more to less accurate:

1. both words and both tags
2. $w_j$ and the two POS tags
3. $w_{h_j}$ and the two POS tags

4. the two POS tags alone

Estimate 1) is used first when it is available, else a combination of 2) and 3) else 4).

Estimate 4) is given by

$$\frac{C(R, \langle t_j \rangle, \langle t_{h_j} \rangle)}{C(\langle t_j \rangle, \langle t_{h_j} \rangle)}.$$

These probabilities do not take into account the distance between words and the directions between governor and dependents. This is introduced by a $\Delta$ variable whose approximation could be $\Delta_{j,h_j} = h_j - j$. In fact, Collins (1999) uses a more sophisticated measure that takes into account:

- The word order between the dependents, because, according to categories, some words tend to have their governor to their left-hand side or to their right-hand side. English is said to be right-branching: complements often occur to the right of the head. An example is given by prepositions, whose governor is almost systematically to the left.
- The distance, because most dependencies tend to be between adjacent or very close words.
- The verb crossings, because dependencies rarely jump over a verb.
- Punctuation.

As a simple example, we will give possible combinations for the reduced sentence *Bring meal to table*. Table 11.12 shows the dependencies with lexical statistics, and Table 11.13 gives dependencies with part of speech only. The probability figure is the product of four terms, one per word index. The maximum value corresponds to the most likely dependency tree.

**Table 11.12.** Probability of dependencies between words with a model including distances. The probability figure corresponds to the product of four probabilities. One is chosen per word index.

| **Word 1** | **Word 2** |
|---|---|
| $P(root \rightarrow bring/vb, \Delta_{root})$ | $P(root \rightarrow meal/noun, \Delta_{root})$ |
| $P(meal/noun \rightarrow bring/vb, \Delta_{1,2})$ | $P(bring/vb \rightarrow meal/noun, \Delta_{2,1})$ |
| $P(to/prep \rightarrow bring/vb, \Delta_{1,3})$ | $P(to/prep \rightarrow meal/noun, \Delta_{2,3})$ |
| $P(table/noun \rightarrow bring/vb, \Delta_{1,4})$ | $P(table/noun \rightarrow meal/noun, \Delta_{2,4})$ |

| **Word 3** | **Word 4** |
|---|---|
| $P(root \rightarrow to/prep, \Delta_{root})$ | $P(root \rightarrow table/noun, \Delta_{root})$ |
| $P(bring/vb \rightarrow to/prep, \Delta_{3,1})$ | $P(bring/vb \rightarrow table/noun, \Delta_{4,1})$ |
| $P(meal/noun \rightarrow to/prep, \Delta_{3,2})$ | $P(to/prep \rightarrow table/noun, \Delta_{4,2})$ |
| $P(table/noun \rightarrow to/prep, \Delta_{3,4})$ | $P(meal/noun \rightarrow table/noun, \Delta_{4,3})$ |

**Table 11.13.** Probability of dependencies between part of speech with a model including distances. The probability figure corresponds to the product of four probabilities, one per word index.

| **Word 1** | **Word 2** |
|---|---|
| $P(root \rightarrow vb, \Delta_{root})$ | $P(root \rightarrow noun, \Delta_{root})$ |
| $P(noun \rightarrow vb, \Delta_{1,2})$ | $P(vb \rightarrow noun, \Delta_{2,1})$ |
| $P(prep \rightarrow vb, \Delta_{1,3})$ | $P(prep \rightarrow noun, \Delta_{2,3})$ |
| $P(noun \rightarrow vb, \Delta_{1,4})$ | $P(noun \rightarrow noun, \Delta_{2,4})$ |

| **Word 3** | **Word 4** |
|---|---|
| $P(root \rightarrow prep, \Delta_{root})$ | $P(root \rightarrow noun, \Delta_{root})$ |
| $P(vb \rightarrow prep, \Delta_{3,1})$ | $P(vb \rightarrow noun, \Delta_{4,1})$ |
| $P(noun \rightarrow prep, \Delta_{3,2})$ | $P(prep \rightarrow noun, \Delta_{4,2})$ |
| $P(noun \rightarrow prep, \Delta_{3,4})$ | $P(noun \rightarrow noun, \Delta_{4,3})$ |

Collins used dependencies to parse constituents. To do this, he mapped function relations $R$ described in this section onto phrase-structure rules and represented dependencies between their respective words using a lexical head in each rule. He singled out one symbol in the right-hand side of each phrase-structure rule to be the governor of the remaining symbols. For example, the rules

```
s --> np, vp.
vp --> verb, np.
np --> det, noun.
```

select a noun as the head of a noun phrase, a verb as the head of the verb phrase, and `vp` as the head of the sentence. Proceeding from the bottom up, the Collins parser annotates dependencies with phrase-structure rules. In our example, the sentence rule is obtained, while the verb percolates to the root of the sentence through the verb phrase.

## 11.8 Further Reading

Parsing techniques have been applied to compiler construction as well as to human languages. There are numerous references reviewing formal parsing algorithms, both in books and articles. Aho et al. (1986) is a starting point.

Most textbooks in computational linguistics describe parsing techniques for natural languages. Pereira and Shieber (1987), Covington (1994), Gazdar and Mellish (1989), and Gal et al. (1989) introduce symbolic techniques and include implementations in Prolog. Allen (1994), Jurafsky and Martin (2000), and Manning and Schütze (1999) are other references that include surveys of statistical parsing. All these books mostly describe, if not exclusively, constituent parsing.

Prepositional phrase attachment is a topic that puzzled many of those adopting the constituency formalism. It often receives special treatment – a special section

in books. For an introduction, see Hindle and Rooth (1993). Techniques to solve it involved the investigation of lexical preferences that probably started a shift of interest toward dependency grammars.

While most research in English has been done using the constituency formalism – and many computational linguists still use it – dependency inspires much of the present work. Covington (1990) is an early example that can parse discontinuous constituents. Tapanainen and Järvinen (1997) describe a parsing algorithm using constraint rules and producing a dependency structure where links are annotated with functions. Constant (1991), El Guedj (1996), and Vergne (1998) provide accounts in French; Hellwig (1980, 1986) was among the pioneers in German. Some authors reformulated parsing a constraint satisfaction problem (CSP) sometimes combining it with a chart. Constraint handling rules (CHR) is a simple, yet powerful language to define constraints (Frühwirth 1998). Constraint handling rules are available in some Prologs, notably SWI Prolog. In 2006, the Tenth Conference on Computational Natural Language Learning (CoNLL-X) organized its shared task on multilingual dependency parsing. The conference site provides background literature, data sets, and an evaluation scheme (http://www.cnts.ua.ac.be/conll/). It is an extremely valuable source of information on dependency parsing.

Statistical parsing is more recent than symbolic approaches. Charniak (1993) is a good account to probabilistic context-free grammars, PCFG. Manning and Schütze (1999) is a comprehensive survey of statistical techniques used in natural language processing. See also the two special issues of *Computational Linguistics* (1993, vol. 19, nos. 1 and 2). Collins' dissertation (1999) is an excellent and accessible description of statistical dependency parsing. Bikel (2004) is a complement to it. Charniak (2000) describes another efficient parser.

Quality of statistics and rules is essential to get good parsing performance. Probabilities are drawn from manually bracketed corpora, and their quality depends on the annotation and the size of the corpus. A key problem is sparse data. For a good review on how to handle sparse data, see Collins (1999) again. Symbolic rules can be tuned manually by expert linguists or obtained automatically using inductive logic techniques, either for constituents or dependencies. Zelle and Mooney (1997) propose an inductive logic programming method in Prolog to obtain rules from annotated corpora.

## Exercises

**11.1.** The shift–reduce program we have seen stores words at the end of the list representing the stack. Subsequently, we use `append/3` to traverse the stack and match it to grammar rules. Modify this program so that words are added to the beginning of the list.

Hint: you will have to reverse the stack and the rules so that rule `s --> np, vp` is encoded `rule([vp, np | X], [s | X]).`

**11.2.** Trace the shift–reduce parser with a null symbol `word(d, [])` and describe what happens.

**11.3.** Modify the shift–reduce parser so that it can handle lists of terminal symbols such as `word(d, [all, the]).`

**11.4.** Complete arcs of Fig. 11.13 with the Earley algorithm.

**11.5.** Trace the Earley algorithm in Prolog with the sentence *Bring the meal of the day*.

**11.6.** In the implementation of the Earley's algorithm, we represented dotted rules as
`np --> np • pp [0, 2]`
by Prolog facts as
`arc(np, [np, '.', pp], 0, 2).`
This representation involves searching a dot in a list, which is inefficient. Modify the program so that it can use an arc representation, where the sequence of categories to the left and to the right of the dot are split into two lists, as with `arc(np, [np], [pp], 0, 2).`

**11.7.** The Earley chart algorithm accepts correct sentences and rejects ill-formed ones, but it does not provide us with the sentence structure. Write a `retrieve` predicate that retrieves parse trees from the chart.

**11.8.** Modify the Cocke, Younger, and Kasami Prolog program to include parsing probabilities to constituents in the chart.

**11.9.** Modify the Cocke, Younger, and Kasami Prolog program to produce the best parse tree as a result of the analysis. Hint: to retrieve the tree more easily, use an array of back pointers: an array storing for each best constituent over the interval `i...j`, the rule that produced it, and the value of `k`.

**11.10.** Implement the Collins dependency parser in Prolog.