Computational Linguistics: Parsing

Raffaella Bernardi

KRDB, Free University of Bozen-Bolzano

Via della Mostra 4, Room: 1.06, e-mail:

bernardi@inf.unibz.it

Contents

1. Summary

We have seen:

- Formal Grammars. Which one?
- CFG: rewriting rules.
- Recognizer vs. Parser.

Today: basic concepts about parsing natural language strings.

2. Parsing

In the first lecture, we have said that parsing is the process of recognizing an input string and assigning a structure to it.

Today we will look at **syntactic parsing**, i.e. the task of recognizing a sentence (or a constituent) and assigning a syntactic structure to it. We will look at **algorithms** (parsers) able to assign context free parse tree to a given input. Better, we shall consider algorithms which operate on a sequence of words (a potential sentence) and a context-free grammar (CFG), to build **one or more trees**.

3. Ambiguity

Why a parsing algorithm may create more than one tree?

Because natural languages are often ambiguous.

What does it mean?

In non-technical terms, “ambiguous” means “having more than one meaning”.

When defining precise grammatical concepts, it is necessary to specify more precisely the different ways in which ambiguity can arise. For the moment (i.e. until we discuss semantics in later chapters) we will be concerned only with syntactic ambiguity, whereby a sentence (or part of a sentence) can be **structured** in different ways. Such multiple classifications normally lead to multiple meanings, so the term “ambiguity” is not unreasonable.

4. Kinds of Ambiguities

More particularly, in our discussion of parsing we shall be concerned only with two types of ambiguity.

- **Lexical Ambiguity**: a single word can have more than one syntactic category; for example, “smoke” can be a noun or a verb, “her” can be a pronoun or a possessive determiner.
- **Structural Ambiguity**: there are a few valid tree forms for a single sequence of words; for example, which are the possible structures for “old men and women”?

  It can be grouped either as [[old men] and women] or [old [men and women]]. Similarly, “John saw the man in the park with the telescope” has several fairly plausible readings.

4.1. Structural Ambiguity

An important distinction must also be made between

- **Global (or total) Ambiguity**: in which an entire sentence has several grammatically allowable analyses.
- **Local (or partial) Ambiguity**: in which portions of a sentence, viewed in isolation, may present several possible options, even though the sentence taken as a whole has only one analysis that fits all its parts.

4.1.1. Global Ambiguity

Global ambiguity can be resolved only by resorting to information outside the sentence (the context, etc.) and so cannot be solved by a purely syntactic parser.

A good parser should, however, ensure that all possible readings can be found, so that some further disambiguating process could make use of them.

For instance,

John saw the woman in the park with the telescope

He was at home.

4.1.2. Local Ambiguity Local ambiguity is essentially what makes the organization of a parser non-trivial – the parser may find, in some situations, that the input so far could match more than one of the options that it has (grammatical rules, lexical items, etc). Even if the sentence is not ambiguous as a whole, it may not be possible for the parser to resolve (locally and immediately) which of the possible choices will eventually be correct.

“When Fred eats food gets thrown”

- [When Fred eats food] gets thrown??
- [When Fred eats] [food gets thrown]

“La vecchia porta è chiusa.”

- $[[\text{La}_{det}\text{vecchia}_n]_{np}][\text{porta}]_v$
- $[\text{La}_{det}[[\text{vecchia}_{adj}\text{porta}_n]_n]_{np}$.

But then “è chiusa” will disabiguate it.

“La vecchia porta la sbarra.”

4.2. Search

Parsing is essentially a **search problem** (of the kind typically examined in artificial intelligence):

- the initial state is the input sequence of words
- the desired final state is a complete tree spanning the whole sentence
- the operators available are the grammar rules and
- the choices in the search space consist of selecting which rule to apply to which constituents.

5. A good Parser

A parsing algorithm is provided with a grammar and a string, and it returns possible analyses of that string. Here are the main criteria for evaluating parsing algorithms:

- **Correctness**: A parser is correct if all the analyses it returns are indeed valid analyses for the string, given the grammar provided.
- **Completeness**: A parsing algorithm is complete if it returns every possible analysis of every string, given the grammar provided.
- **Efficiency**: A parsing algorithm should not be unnecessarily complex. For instance, it should not repeat work that only needs to be done once.

5.1. Correctness

A parser is correct if **all the analyses it returns are indeed valid analyses for the string, given the grammar provided**.

- In practice, we almost always require correctness.
- In some cases, however, we might allow the parsing algorithm to produce some analyses that are **incorrect**, and we would then **filter out** the bad analyses subsequently. This might be useful if some of the **constraints** imposed by the grammar were very **expensive to test** while parsing was in progress but very few possible analyses would actually be rejected by them.

5.2. Completeness

A parsing algorithm is complete if it returns **every possible analysis of every string, given the grammar provided.**

In some circumstances, completeness may not be desirable. For instance, in some applications there **may not be time** to enumerate all analyses and there may be good **heuristics** to determine what the "best" analysis is without considering all possibilities. Nevertheless, we will generally assume that the parsing problem entails returning all valid analyses.

6. Terminating vs. Complete

It is important to realize that there is a distinction between "complete" (i.e. in principle produces all analyses) and "terminating" (i.e. will stop processing in a **finite amount of time**).

A parsing mechanism could be devised which systematically computes every analysis (i.e. is complete) but if it is given a grammar for which there are an infinite number of analyses, it will not terminate.

```
np ---> pn
pn ---> np
```

7. Parse Trees: Example

Given the grammar:

```
s ---> np vp             tv ---> shot
np ---> pn               pn ---> vincent
vp ---> tv np            pn ---> marcellus
```

we want to build the parse tree corresponding to the sentence "vincent shot marcellus".

We know that

1. there must be three leaves and they must be the **words** "vincent", "marcellus", "shot".
2. the parse tree must have one root, which must be the **start symbol** s.

We can now use either the input words or the rules of the grammar to drive the process. Accordingly to the choice we make, we obtain a "bottom up" and "top-down" parsing, respectively.

8. Bottom up Parsing

The basic idea of bottom up parsing and recognition is:

- to **begin** with the concrete data provided by the input string — that is, the **words** we have to parse/recognize — and try to build bigger and bigger pieces of structure using this information.
- Eventually we hope to put all these pieces of structure together in a way that shows that we have found a sentence.

Putting it another way, bottom up parsing is about moving from concrete low-level information to more abstract high-level information.

This is reflected in a very obvious point about any bottom up algorithm: in bottom up parsing, **we use our CFG rules right to left**.

8.1. A bit more concretely

Consider the CFG rule $C \rightarrow P_1, P_2, P_3$.

Working bottom up means that we will try to find a P_1, a P_2, and a P_3 in the input that are right next to each other. If we find them, we will use this information to conclude that we have found a C.

That is, in bottom up parsing, the flow of information is from the right hand side (P_1, P_2, P_3) of the rules to the left hand side of the rules (C).

Let's look at an example of bottom up parsing/recognition start from a linguistics input.

8.2. An Example

"Vincent shot Marcellus". Working bottom up, we might do the following.

1. First we go through the string, systematically looking for strings of **length 1** that we can rewrite by using our CFG rules in a right to left direction.
2. Now, we have the rule $pn \rightarrow vincent$, so using this in a right to left direction gives us: *pn* shot marcellus.
3. But wait: we also have the rule $np \rightarrow pn$, so using this right to left we build: *np* shot marcellus.
4. We're still looking for strings of length 1 that we can rewrite using our CFG rules right to left — but we can't do anything with *np*.
5. But we can do something with the second symbol, "shot". We have the rule $tv \rightarrow shot$, and using this right to left yields: $np\ tv\ marcellus$.

6. Can we rewrite *tv* using a CFG rule right to left?

   No — so it's time to move on and see what we can do with the last symbol, "marcellus".

   We have the rule $pn \rightarrow$ marcellus, and this lets us build: *np tv pn*

7. We also have the rule $np \rightarrow pn$ so using this right to left we build: *np tv np*

8. Are there any more strings of length 1 we can rewrite using our context free rules right to left?

   No — we've done them all.

9. So now we start again at the beginning looking for strings of **length 2** that we can rewrite using our CFG rules right to left. And there is one: we have the rule $vp \rightarrow tv\ np$, and this lets us build: *np vp*

10. Are there any other strings of length 2 we can rewrite using our CFG rules right to left? Yes — we can now use: $s \rightarrow np\ vp$, we have built: *s*

11. And this means we are finished.

Working bottom up we have succeeded in rewriting our original string of symbols into the symbol *s* — so we have successfully recognized "Vincent shot Marcellus" as a sentence.

8.3. Example

Sara wears the new dress	$pn \longrightarrow$ sara
pnwears the new dress	$np \longrightarrow pn$
npwears the new dress	$tv \longrightarrow$ wears
np tv the new dress	$det \longrightarrow$ the
np tv det new dress	$adj \longrightarrow$ new
np tv det adj dress	$n \longrightarrow$ dress
np tv det adj n	$n \longrightarrow adj\ n$
np tv det n	$np \longrightarrow det\ n$
np tv np	$vp \longrightarrow tv\ np$
np vp	$s \longrightarrow np\ vp$
s	

8.4. Exercise

Given the lexicon below, build the CFG rules and use the same strategy described above to parse the input strings below.

1. “John saw the man with the telescope.”

```
pn ---> john        n ---> telescope
tv ---> saw         det ---> the
n ---> park         p ---> with
n ---> man          p ---> in
```

How many parse trees do you obtain?

```
s ---> np vp
np ---> det n
np ---> det n pp
np --> pn
vp ---> tv np
vp ---> tv np pp
pp ---> p np
```

8.5. Remarks on Bottom-up

A couple of points are worth emphasizing. This is just one of many possible ways of performing a bottom up analysis. All bottom up algorithms use CFG rules right to left — but there **many different ways** this can be done.

To give a rather pointless example: we could have designed our algorithm so that it started reading the input in the middle of the string, and then zig-zagged its way to the front and back. And there are many much more serious variations — such as the choice between depth first and breadth first search that we will look at later today.

In fact, the algorithm that we used above is crude and **inefficient**. But it does have one advantage — it is **easy** to understand and easy to put into Prolog.

9. Top down Parsing

As we have seen, in bottom-up parsing/recognition we start at the most concrete level (the level of words) and try to show that the input string has the abstract structure we are interested in (this usually means showing that it is a sentence). So we use our CFG rules right-to-left.

In top-down parsing/recognition we do the reverse.

- We **start at the most abstract level** (the level of sentences) and work down to the most concrete level (the level of words).
- So, given an input string, we start out by assuming that it is a sentence, and then try to prove that it really is one by using the rules left-to-right.

9.1. A bit more concretely

That works as follows:

1. If we want to prove that the input is of category s and we have the rule $s \rightarrow np\ vp$, then we will try next to prove that the input string consists of a noun phrase followed by a verb phrase.

2. If we furthermore have the rule $np \rightarrow det\ n$, we try to prove that the input string consists of a determiner followed by a noun and a verb phrase.

That is, we use the rules in a **left-to-right** fashion to expand the categories that we want to recognize until we have reached categories that match the preterminal symbols corresponding to the words of the input sentence.

9.2. An example

The left column represents the sequence of categories and words that is arrived at by replacing one of the categories (identical to the left-hand side of the rule in the second column) on the line above by the right-hand side of the rule or by a word that is assigned that category by the lexicon.

s	$s \longrightarrow np\ vp$
np vp	$vp \longrightarrow v\ np$
np v np	$np \longrightarrow det\ n$
np v det n	$n \longrightarrow adj\ n$
np v det adj n	$np \longrightarrow$ Sara
Sara v det adj n	$v \longrightarrow$ wears
Sara wears det adj n	$det \longrightarrow$ the
Sara wears the adj n	$adj \longrightarrow$ new
Sara wears the new n	$n \longrightarrow$ dress
Sara wears the new dress	

9.3. Further choices

Of course there are lots of choices still to be made.

- Do we scan the input string from right-to-left, from left-to-right, or zig-zagging out from the middle?
- In what order should we scan the rules? More interestingly, do we use depth-first or breadth-first search?

9.4. Depth first search

Depth first search means that whenever there is more than one rule that could be applied at one point, we **explore one possibility and only look at the others when this one fails**. Let's look at an example.

```
s  ---> np, vp.
np ---> pn.
vp ---> iv.
vp ---> tv, np.

lex(vincent,pn). %alternative notation for pn ---> vincent
lex(mia,pn).
lex(died,iv).
lex(loved,tv).
lex(shot,tv).
```

The sentence “Mia loved Vincent” is admitted by this grammar. Let’s see how a top-down parser using depth first search would go about showing this.

9.4.1. Example

	State		Comments
1.	`s`	*mia loved vincent*	`s ---> [np, vp]`
2.	`np vp`	*mia loved vincent*	`np ---> [pn]`
3.	`pn vp`	*mia loved vincent*	`lex(mia,pn)`
			We've got a match
4.	`vp`	*loved vincent*	`vp ---> [iv]`
			We're doing depth first search. So we ignore the other vp rule for the moment.
5.	`iv`	*loved vincent*	No applicable rule. Backtrack to the state in which we last applied a rule. That's state 4.
4'.	`vp`	*loved vincent*	`vp ---> [tv]`
5'.	`tv np`	*loved vincent*	`lex(loved,tv)`
			Great, we've got match!
6'.	`np`	*vincent*	`np ---> [pn]`
7'.	`pn`	*vincent*	`lex(vincent,pn)`
			Another match. We're done.

9.5. Reflections

It should be clear why this approach is called top-down: we clearly work from the abstract to the concrete, and we make use of the CFG rules **left-to-right**.

Furthermore, it is an example of depth first search because when we were faced with a choice, we selected one alternative, and worked out its consequences. If the choice turned out to be wrong, we **backtracked**.

For example, above we were faced with a choice of which way to try and build a *vp* — using an intransitive verb or a transitive verb.

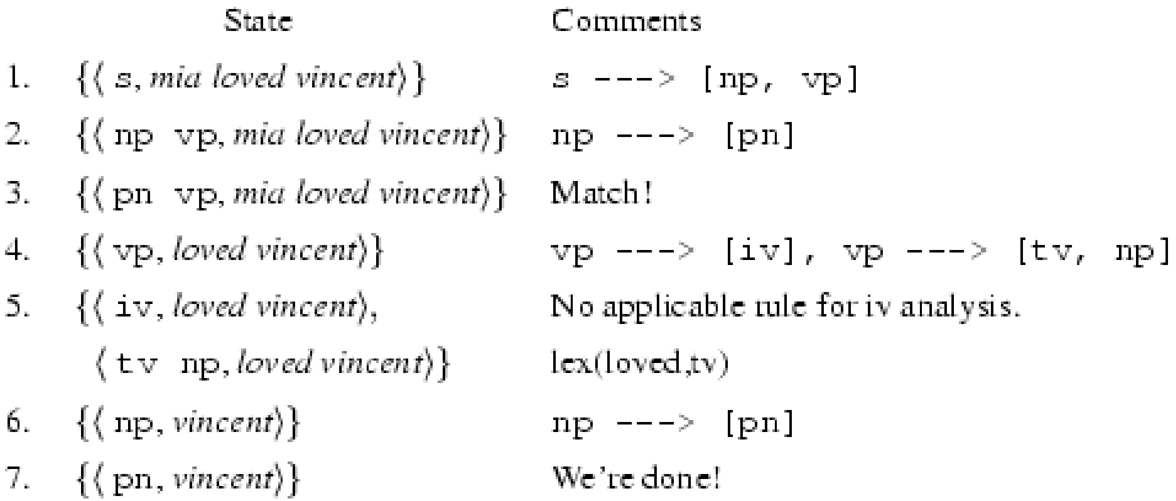
We first tried to do so using an intransitive verb (at state 4) but this didn't work out (state 5) so we backtracked and tried a transitive analysis (state 4'). This eventually worked out.

9.6. Breadth first search

The big difference between breadth-first and depth-first search is that in breadth-first search we **carry out all possible choices at once**, instead of just picking one.

It is useful to imagine that we are working with a **big bag containing all the possibilities** we should look at — so in what follows I have used set-theoretic braces to indicate this bag. When we start parsing, the bag contains just one item.

9.6.1. An example



	State	Comments
1.	{⟨ s, *mia loved vincent*⟩}	`s ---> [np, vp]`
2.	{⟨ np vp, *mia loved vincent*⟩}	`np ---> [pn]`
3.	{⟨ pn vp, *mia loved vincent*⟩}	Match!
4.	{⟨ vp, *loved vincent*⟩}	`vp ---> [iv], vp ---> [tv, np]`
5.	{⟨ iv, *loved vincent*⟩,	No applicable rule for iv analysis.
	⟨ tv np, *loved vincent*⟩}	lex(loved,tv)
6.	{⟨ np, *vincent*⟩}	`np ---> [pn]`
7.	{⟨ pn, *vincent*⟩}	We're done!

The crucial difference occurs at state 5. There we try both ways of building *vp* at once. At the next step, the intransitive analysis is discarded, but the transitive analysis remains in the bag, and eventually succeeds.

9.7. Comparing Depth first and Breadth first searches

- The **advantage of breadth-first** search is that it prevents us from zeroing in on one choice that may turn out to be completely wrong; this often happens with depth-first search, which causes a lot of backtracking.
- Its **disadvantage** is that we need to keep track of all the choices — and if the bag gets big (and it may get very big) we pay a computational price.

So which is better?

There is no general answer. With some grammars breadth-first search, with others depth-first.

9.8. Exercise

Try the two top-down approaches to parse “La vecchia porta sbatte” given the grammar below.

```
det ---> la                s --> np vp
adj ---> vecchia           vp --> iv
n ---> vecchia             vp --> tv np
n ---> porta               np --> det n
tv ---> porta              n --> adj n
iv ---> sbatte
```

10. Bottom-up vs. Top-down Parsing

Each of these two strategies has its own advantages and disadvantages:

1. Trees (not) **leading to an *s***
   - ▶ The top-down parsing: It never wastes time exploring tree that cannot result in an *s*.
   - ▶ The bottom-up parsing: trees that have no hope of leading to an *s* are generated.
2. Trees (not) **consistent with the input**:
   - ▶ The top-down parsing: It can waste time generating trees which are not consistent with the input.
   - ▶ The bottom-up parsing: It never generates tree which are not locally grounded in the actual input.

Used parsers usually combine the best features of the two approaches.

10.1. Going wrong with bottom-up

Say, we have the following grammar fragment:

```
s ---> np vp
np ---> det n
vp ---> iv
vp ---> tv np
tv ---> plant
iv ---> died
det ---> the
n ---> plant
```

Try to parse “the plant died” using a bottom-up parser.

10.2. Solution: Bottom up

Note, how “plant” is ambiguous in this grammar: it can be used as a common noun or as a transitive verb.

1. If we now try to bottom-up recognize “the plant died”, we would first find that “the” is a determiner, so that we could rewrite our string to “det plant died”.
2. Then we would find that “plant” can be a transitive verb giving us “det tv died”.
3. “det” and “tv” cannot be combined by any rule.
4. So, “died” would be rewritten next, yielding “det tv iv” and then “det tv vp”.
5. Here, it would finally become clear that we took **a wrong decision** somewhere: nothing can be done anymore and we have to backtrack.
6. Doing so, we would find that “plant” can also be a noun, so that “det plant died” could also be rewritten as “det n died”, which will eventually lead us to success.

10.3. Going wrong with top-down

Assume we have the following grammar

```
s ---> np vp
np ---> det n
np ---> pn
vp ---> iv
det ---> the
n ---> robber
pn ---> Vincent
iv ---> died
```

try to use it to top-down recognize the string "vincent died".

10.3.1. Solution: Top-Down

1. Proceeding in a top-down manner, we would first expand *s* to *np vp*.
2. Next we would check what we can do with the *np* and find the rule $np \rightarrow det\ n$.
3. We would therefore expand *np* to *det n*.
4. Then we either have to find a lexical rule to relate “vincent” to the category *det*, or we have to find a phrase structure rule to expand *det*.
5. Neither is possible, so we would **backtrack** checking whether there are any alternative decisions somewhere.

11. Using both

We have seen that using a pure **top-down approach, we are missing some important information provided by the words** of the input string which would help us to guid our decisions.

However, similarly, using a pure **bottom-up approach, we can sometimes end up in dead ends** that could have been avoided had we used some bits of top-down information about the category that we are trying to build.

The key idea of **left-corner parsing** is to **combine top-down processing with bottom-up processing** in order to avoid going wrong in the ways that we are prone to go wrong with pure top-down and pure bottom-up techniques.

11.1. Left Corner of a rule

The left corner of a rule is the first symbol on the right hand side.

For example,

- np is the left corner of the rule $s \rightarrow$ **np** vp, and
- iv is the left corner of the rule $vp \rightarrow$ **iv**.
- Similarly, we can say that "vincent" is the left corner of the lexical rule $pn \rightarrow$ **vincent**.

11.2. Left Corner parser

A left-corner parser starts with a **top-down prediction** fixing the category that is to be recognized, like for example *s*. Next, it takes a **bottom-up step** and then alternates bottom-up and top-down steps until it has reached an *s*.

1. The bottom-up processing steps work as follows. Assuming that the parser has just recognized a noun phrase, it will in the next step look for a rule that has an *np* as its left corner.

2. Let's say it finds $s \rightarrow np\ vp$. To be able to use this rule, it has to recognize a *vp* as the next thing in the input string.

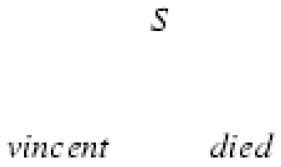
3. This imposes the top-down constraint that what follows in the input string has to be a verb phrase.

4. The left-corner parser will continue alternating bottom-up steps as described above and top-down steps until it has managed to recognize this verb phrase, thereby completing the sentence.

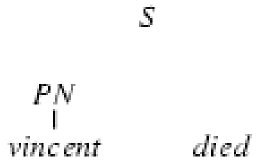
11.3. Example

Now, let's look at how a left-corner recognizer would proceed to recognize "vincent died".

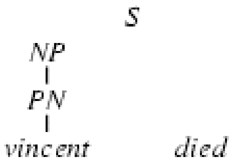
1. Input: vincent died. Recognize an *s*. (Top-down prediction.)



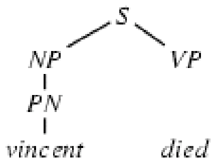
2. The category of the first word of the input is *pn*. (Bottom-up step using a lexical rule.)



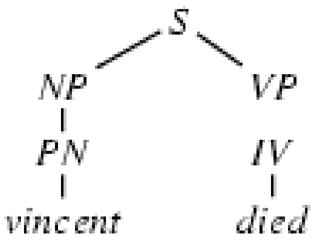
3. Select a rule that has *pn* at its left corner: $np \rightarrow pn$. (Bottom-up step using a phrase structure rule.)



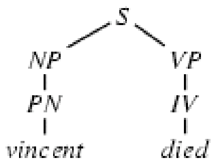
4. Select a rule that has *np* at its left corner: $s \rightarrow np\ vp$ (Bottom-up step.)



5. Match! The left hand side of the rule matches with , the category we are trying to recognize.



6. Input: died. Recognize a *vp*. (Top-down prediction.)
7. The category of the first word of the input is *iv*. (Bottom-up step.)
8. Select a rule that has *iv* at its left corner: $vp \rightarrow iv$. (Bottom-up step.)



9. Match! The left hand side of the rule matches with *vp*, the category we are trying to recognize.

Make sure that you see how the steps of bottom-up rule application alternate with top-down predictions in this example. Also note that this is the example that we used earlier on for illustrating how top-down parsers can go wrong and that, in contrast to the top-down parser, the left-corner parser doesn't have to backtrack with this example.

11.4. What did we improve and what not?

This left-corner recognizer handles the example that was problematic for the pure top down approach **much more efficiently**.

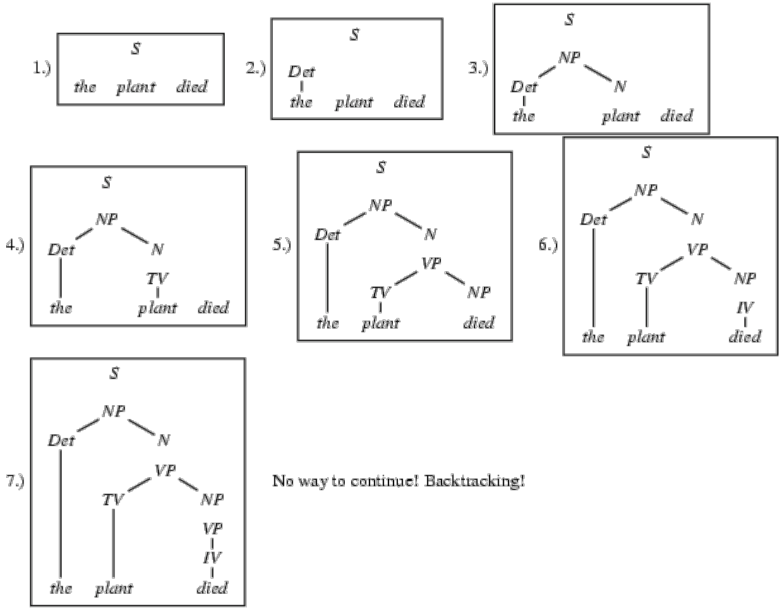
It finds out what is the category of “vincent” and then doesn’t even try to use the rule $np \rightarrow det\ n$ to analyze this part of the input. Remember that the top-down recognizer did exactly that.

But there are **no improvement** on the example that was problematic for the **bottom-up** approach. Just like the bottom up recognizer, the left-corner recognizer will first try to analyze “plant” as a transitive verb.

Let’s see step by step what the left-corner recognizer defined above does to process “the plant died” given the grammar.

Try it first your self.

11.5. Solution



No way to continue! Backtracking!

11.6. Comments

So, just like the bottom-up recognizer, the left-corner recognizer chooses the wrong category for “plant” and needs a long time to realize its mistake.

However, the left-corner recognizer provides the information that the constituent we are trying to build at that point is a “noun”. And **nouns can never start with a transitive verb** according to the grammar we were using.

If the recognizer uses this information, it would notice immediately that the lexical rule relating “plant” to the category transitive verb cannot lead to a parse.

11.7. Left Corner Table

The solution is to record this information in a table.

This left-corner table **stores which constituents can be at the left-corner of which other constituents**.

For the little grammar of the problematic example the left-corner table would look as follows:

```
s ---> np vp
np ---> det n
vp ---> iv
vp ---> tv np
tv ---> plant
iv ---> died
det ---> the
n ---> plant
```

s	np, det, s
np	det, np
vp	iv, tv, vp
det	det
n	n
iv	iv
tv	tv

12. Overgeneration: Agreement

For instance, can the CFG we have built distinguish the sentences below?

1. He hates a red shirt
2. *He like a red shirt
3. He hates him
4. *He hates he

12.1. Agreement between SUB and Verb

When working with agreement a first important fact to be taken into account is that we use

- a plural VP if and only if we have a plural NP, and
- a singular VP if and only if we have a singular NP.

For instance,

1. “the gangster dies”
2. * “the gangster die”

That is, we have to distinguish between singular and plural VPs and NPs.

12.2. First try

One way of doing this would be to **invent new non-terminal symbols** for plural and singular NPs and VPs. Our grammar would then look as follows.

We would have two rules for building sentences: one for building a sentence out of a singular NP (NPsg) and a singular VP (VPsg), and the other one for using a plural NP (NPpl) with a plural VP (VPpl).

Singular NPs are built out of a determiner and a singular noun (Nsg) and plural NPs are built out of a determiner and a plural noun (Npl). Note that we don't have to distinguish between singular and plural determiners as we are only using "the" at the moment, which works for both.

Similarly, singular VPs are built out of singular intransitive verbs (IVsg) and plural VPs out of plural intransitive verbs (IVpl).

Finally, we have singular and plural nouns and verbs in our lexicon.

12.3. Loss of efficiency

Now, the grammar does what it should:

1. "the gangster dies"
2. "the gangsters die"
3. *"the gangster die"
4. *"the gangsters dies".

However, compared to the grammar we started with, it has become **huge** – we have twice as many phrase structure rules, now. And we only added the information whether a noun phrase or a verb phrase is plural or singular.

Imagine we next wanted to add transitive verbs and pronouns. To be able to correctly accept "he shot him" and reject "him shot he", we would need case information in our grammar. And if we also wanted to add the pronouns "I" and "you", we would further have to distinguish between first, second and third person.

If we wanted to code all this information in the non-terminal symbols of the grammar, we would need non-terminal symbols for all combinations of these features. Hence, the **size of the grammar would explode** and the rules would probably become very difficult to read.

12.4. Second Try

We use features to represent case (subject, object), gender (female, masculine), number (singular, plural).

```
s --> np(subj), vp.
vp --> vt, np(obj).
vp --> vi.
np(CASE) --> pro(CASE).
np(_) --> det, n.
np(_) --> pn.
Lexicon
det --> the
n --> whiskey
pn --> bill
pro(subj) --> he
pro(obj) --> him
vi --> fights
vt --> kills
```

Try the second exercise.

12.5. Second try (cont'd)

While doing the exercise, you might have noticed that the extra argument — the feature — is simply **passed up the tree** by ordinary **unification**. And, depending on whether it can correctly unify or not, this feature controls the facts of English case avoiding duplications of categories.

Summing up,

- features let us get rid of lots of unnecessary rules in a natural way.
- In the lab we will see that PROLOG enables us to implement rules with feature information in a natural way.

This way of handling features however has some limits, in particular it does not provide an adequate syntax descriptions of the agreement phenomena in general terms.

13. Not done, Projects and Next

1. Chart
2. Parsing with Feature Structure Unification

Projects If you already know some parsing techniques you could implement an algorithm to be applied on a given grammar and parse some of the sentence it can generates.

Next So far we have being dealing only with syntax. But sentences have a meaning too. This is going to be the topic of next week.

Practical Info Change in the schedule: http://www.inf.unibz.it/~bernardi/Courses/
CompLing/09-10.html#handouts