1. Features: more than notational convenience?

At one level, features are just a convenience

The allow us to write lexicon entries and rules more transparently

But they also "capture generalisations"

If we write a pair of rules using some (in principle opaque) complex category labels, they are not obviously related in any way:

\[ S \rightarrow \text{NP}_s \text{ VP}_p \]
\[ S \rightarrow \text{NP}_p \text{ VP}_p \]

It appears as if we have to justify each of these independently

- or that we *might* have had one without the other
- or had \[ S \rightarrow \text{NP}_s \text{ VP}_p \] just as well

Whereas when we write

\[ S \rightarrow \text{NP} \text{ VP} [<\text{NP agreement}> = <\text{VP agreement}>] \]

we are making a stronger claim, even though 'behind the scenes' this single line corresponds to a collection of simple atomic-category rules
2. Infinity again: categories

Once you move to feature bundles as the values of features

- You can in principle have an infinite number of features
- And, as with infinite numbers of rules, that actually changes your position on the Chomsky hierarchy

One strand of modern grammatical theory

- From GPSG to HPSG to so-called sign-based grammatical theories

Puts essentially all the expressive power of the grammar into feature structures

3. Unification

When we write '=' between two feature paths or feature variables, we mean more than an equality test

Consider the noun phrase "a sheep", and the following rules

The resulting parse tree reveals that we have not only tested for compatibility between the various feature structures, we've actually merged them:

where by the ① we mean that all three agreement values are the the same feature structure

4. Unification, cont'd

The implications of unification run deep
The three occurrences of ① don’t just appear the same

- They are the same
- That is, a single structure, shared 3 times
- So any change to one in the future will be a change to all
- As would be the case with e.g. "the sheep runs" or "the sheep run"

J&M give a detailed introduction to unification, which is what this is called, in section 15.2, and a formal definition in section 15.4.

The directed acyclic graph (DAG) way of drawing feature structures used in J&M 15.4 makes clearer when necessary structure identity is the case, as opposed to contingent value equality.

5. Parsers

A parser is an algorithm that computes a structure for an input string given a grammar.

All parsers have two fundamental properties:

**Directionality**
The sequence in which the structures are constructed
- Almost always **top-down** or **bottom-up**

**Search strategy**
The order in which the search space of possible analyses is explored
- Usually **depth-first**, **breadth-first** or **best-first**

6. Recursive Descent Parsing

A recursive descent parser treats a grammar as a specification of how to break down a top-level goal into subgoals

This means that it works very similarly to a particular blind approach to constructing a rewriting interpretation derivation:

**Directionality**
**Top-down:**
- starts from the start symbol of the grammar
- works down to the terminals

**Search strategy**
**Depth-first:**
- expands the left-most unsatisfied non-terminal
- until it gets to a terminal
  - which either matches the next item in the input
  - or it doesn’t

7. Recursive Descent Parsing: preliminaries

We’re trying to build a parse tree, given
• a grammar
• an input, i.e. a sequence of terminal symbols

As for any other depth-first search, we may have to backtrack

• So we keep a stack of backtrack points
• And whenever we make a choice among several rules to try, we add a backtrack point consisting of
  ◦ a partial tree
  ◦ the remaining as-yet-unexplored rules
  ◦ and the as-yet-unconsumed items of input

Note that, to make the search go depth-first

• We're using a stack
• That is, we'll operate it last in, first out (LIFO)

Finally, we'll need a notion of where the focus of attention is in the tree we're building

• We'll call this the subgoal

8. Recursive Descent Parsing: Algorithm sketch

We start with

• a tree consisting of an 'S' node
  ◦ with no children
  ◦ This node is currently the subgoal
• An empty stack
• An input sequence

Repeatedly

1. If the subgoal is a non-terminal
   a. Choose a rule from the set of rules in the grammar whose left-hand sides match the subgoal
      a. For example, the very first time around the loop, we might choose

      \[ S \rightarrow NP \ VP \]

      b. add children to the subgoal node corresponding to the symbols in the right-hand side of the chosen rule, in order
         a. In our example, that's two children, NP and VP
      c. Make the first of these the new subgoal
         a. This is the other thing which makes this a depth-first search
   d. Go back to (1)
2. Otherwise (the subgoal is a terminal)
   a. If the input is empty, Backtrack
   b. If the subgoal matches the first item of input
      i. Consume the first item of the input
      ii. Advance the subgoal
      iii. Go back to (1)
   c. Otherwise (they don't match), Backtrack
9. Recursive Descent Parsing: Algorithm sketch, concluded

The three imperative actions in the preceding algorithm are defined as follows:

**Choose**
- Pick one member from the set of rules
  1. If the set has only one member, you're done
  2. Otherwise, **push** a new backtrack point onto the stack
      - With the unchosen rules, the current tree and subgoal and the current (unconsumed) input sequence

**Advance**
- Change the subgoal, as follows:
  1. If the current subgoal has a sibling to its right, pick that
  2. Failing which, if the current subgoal is not the root, set the subgoal to the current subgoal's parent, and go back to (1)
  3. Failing which, if the input is empty, we win
      - The current subgoal is the 'S' at the root, and it is the top node of a complete parse tree for the original input
  4. Otherwise, **Backtrack**

**Backtrack**
- Try to, as it were, change your mind. That is:
  1. Unless the stack is empty, **pop** the top backtrack point off the backtrack stack and
      a. Set the tree, subgoal and input from it
      b. **Choose** a rule from its set of rules
      c. Go back to step (1b) of the algorithm
  2. Otherwise (the stack is empty)
      - We lose!
      - There is no parse for the input with the grammar

10. Search Strategies

Schematic view of the search space:

In **depth-first** search the parser
- explores one branch of the search space at a time
- If this branch is a dead-end, it needs to **backtrack**

In **breadth-first** search the parser
- explores all possible branches in parallel
often rendered impossible by memory requirements

11. Shift-Reduce Parsing

Search strategy does not imply a particular directionality in which structures are built.

Recursive descent parsing searches depth-first and builds top-down.

Although Shift-reduce parsing also searches depth-first, in contrast it builds structures bottom-up.

It does this by repeatedly

1. shifting terminal symbols from the input string into the bottom of a stack
2. reducing some elements of the stack to the LHS side of a rule when they match its RHS

As described, this is just a recogniser

• You win if you end up with a single 'S' on the stack and no more input

Actual parsing requires more bookkeeping

Given certain constraints, it is possible to pre-compute auxiliary information about the grammar and exploit it during parsing so that no backtracking is required.

Modern computer languages are often parsed this way

• But grammars for natural languages don't (usually) satisfy the relevant constraints

12. Global and Local Ambiguity

All depth-first parsing is inherently serial, and serial parsers can be massively inefficient when faced with local ambiguity.

A string can have more than one structural analysis (called global ambiguity) for one or both of two reasons:

• Grammatical rules allow for different attachment options;
• Lexical rules that allow a word to be in more than one word class.

Within a single analysis, some sub-strings can be analyzed in more than one way (called local ambiguity), even if not all these sub-string analyses are compatible with some global analysis of the entire string.

Local ambiguity is very common in Natural Languages.

13. Complexity

Depth-first parsing strategies demonstrate other problems with "parsing as search":

1. Structural ambiguity in the grammar and lexical ambiguity in the words (that is, words occurring under more than one part of speech) may lead the parser down a wrong path
2. So the same sub-tree may be built several times
   ◦ whenever a path fails, the parser abandons any subtrees computed since the last backtrack point, backtracks and starts again
The complexity of this **blind backtracking** is exponential in the worst case because of repeated **re-analysis** of the same sub-string.

[Worked example on the whiteboard, for "gave a book to Robin" with this grammar:

- \( VP \rightarrow V_{ditr} \ NP \ NP \ NP \rightarrow \text{Det} \ N \)
- \( VP \rightarrow V_{trpp} \ NP \ PP \ NP \rightarrow \text{PropN} \)
- \( PP \rightarrow P \ NP \)

One solution to this problem is **Chart parsing**.