# ***Generic 'deep' NLP Application Architecture***

Many NLP applications can be adequately implemented with relatively shallow processing. For instance, spelling checking only requires a word list and simple morphology to be useful. I'll use the term 'deep' NLP for systems that build a meaning representation (or an elaborate syntactic representation), which is generally agreed to be required for applications such as NLIDs, email question answering and good MT.

The most important principle in building a successful NLP system is modularity. NLP systems are often big software engineering projects — success requires that systems can be improved incrementally.

The input to an NLP system could be speech or text. It could also be gesture (multimodal input or perhaps a Sign Language). The output might be non-linguistic, but most systems need to give some sort of feedback to the user, even if they are simply performing some action (issuing a ticket, paying a bill, etc). However, often the feedback can be very formulaic.

There's general agreement that the following system components can be described semi-independently, although as- assumptions about the detailed nature of the interfaces between them differ. Not all systems have all of these components:

1. Input preprocessing: speech recognizer or text preprocessor (non-trivial in languages like Chinese or for highly structured text for any language) or gesture recognizer. Such system might themselves be very complex, but I won't discuss them in this course — we'll assume that the input to the main NLP component is segmented text.
2. Morphological analysis: this is relatively well-understood for the most common languages that NLP has considered, but is complicated for many languages (e.g., Turkish, Basque).
3. Part of speech tagging: not an essential part of most deep processing systems, but sometimes used as a way of cutting down parser search space.
4. Parsing: this includes syntax and compositional semantics, which are sometimes treated as separate components

5. Disambiguation: this can be done as part of parsing, or (partially) left to a later phase
6. Context module: this maintains information about the context, for anaphora resolution, for instance.
7. Text planning: the part of language generation that's concerned with deciding what meaning to convey (I won't discuss this in this course)
8. Tactical generation: converts meaning representations to strings. This may use the same grammar and lexicon as the parser.
9. Morphological generation: as with morphological analysis, this is relatively straightforward for English.
10. Output processing: text-to-speech, text formatter, etc. As with input processing, this may be complex, but for now we'll assume that we're outputting simple text.

***Application specific components, for instance:***

1. For NL interfaces, email answering and so on, we need an interface between semantic representation (expressed as some form of logic, for instance) and the underlying knowledge base.

2. For MT based on transfer, we need a component that maps between semantic representations.

It is also very important to distinguish between the knowledge sources and the programs that use them. For instance, a morphological analyser has access to a lexicon and a set of morphological rules: the morphological generator might share these knowledge sources. The lexicon for the morphology system may be the same as the lexicon for the parser and generator.

**Other things might be required in order to construct the standard components and knowledge sources:**

1. lexicon acquisition

2. Grammar acquisition
3. Acquisition of statistical information

For a component to be a true module, it obviously needs a well-defined set of interfaces. What's less obvious is that it needs its own evaluation strategy and test suites: developers need to be able to work somewhat independently.

In principle, at least, components are reusable in various ways: for instance, a parser could be used with multiple grammars, the same grammar can be processed by different parsers and generators, a parser/grammar combination could be used in MT or in a natural language interface. However, for a variety of reasons, it is not easy to reuse components like this, and generally a lot of work is required for each new application, even if it's based on an existing grammar or the grammar is automatically acquired.

We can draw schematic diagrams for applications showing how the modules fit together.