800204: Week 9

Grammar Engineering

Andrew MacKinlay

October 6, 2010

Revision: CFGs

A basic no-feature CFG

```
S       ->   NP VP
NP      ->   Det N
VP      ->   IV
VP      ->   TV NP

Det     ->  'the' | 'these' | 'this'
N       ->  'dog' | 'dogs' | 'cat' | 'cats'
IV      ->  'run' | 'runs' | 'barked'
TV      ->  'chased'
```

Revision: the need for features

- That grammar was unsatisfactory for several reasons
- We can't handle *these dogs* vs **this dogs* distinction
- The similarity between transitive and intransitive verbs is not clear.

Revision: feature-based grammars

Adding some features (not very well)

```
S            ->   NP[NUM=sg] VP[NUM=sg]
S            ->   NP[NUM=pl] VP[NUM=pl]
NP[NUM=sg]   ->   Det[NUM=sg] N[NUM=sg]
NP[NUM=pl]   ->   Det[NUM=pl] N[NUM=pl]
VP[NUM=sg]   ->   V[SUBCAT=intrans, NUM=sg]
VP[NUM=pl]   ->   V[SUBCAT=intrans, NUM=pl]
VP[NUM=sg]   ->   V[SUBCAT=trans, NUM=sg] NP
VP[NUM=pl]   ->   V[SUBCAT=trans, NUM=pl] NP

Det[NUM=pl]  ->  'the' | 'these'
Det[NUM=sg]  ->  'the' | 'this'
N[NUM=sg]    ->  'dog' | 'cat'
N[NUM=pl]    ->  'dogs' | 'cats'
V[SUBCAT=intrans, NUM=pl] -> 'run' | 'barked'
V[SUBCAT=intrans, NUM=sg] -> 'runs' | 'barked'
V[SUBCAT=trans, NUM=pl]   -> 'chased'
V[SUBCAT=trans, NUM=sg]   -> 'chased'
```

Adding some features in a smarter way

```
S             ->    NP[NUM=?n] VP[NUM=?n]
NP[NUM=?n]    ->    Det[NUM=?n] N[NUM=?n]
VP[NUM=?n]    ->    V[SUBCAT=intrans, NUM=?n]
VP[NUM=?n]    ->    V[SUBCAT=trans, NUM=?n] NP

Det[NUM=pl]   ->   'these'
Det[NUM=sg]   ->   'this'
Det           ->   'the'
N[NUM=sg]     ->   'dog' | 'cat'
N[NUM=pl]     ->   'dogs' | 'cats'
V[SUBCAT=intrans, NUM=pl] -> 'run'
V[SUBCAT=intrans, NUM=sg] -> 'runs'
V[SUBCAT=trans]           -> 'chased'
V[SUBCAT=intrans]         -> 'barked'
```

Revision: feature-based grammars

- We can now handle distinctions like:
  - *The dog chased these cats* but **The dog chased*
  - *These dogs barked* but **These dog barked* or **This dogs barked*
  - *The cats run* or *The cat runs* but not *The cat run*
- Using more features, we can handle a whole range of syntactic constructions, eg:
  - Other verb subcategorisation, such as clausal complements (*You claim that you like children*)
  - Subject-verb inversion (*Who do you like?*)
  - Filler-gap constructions (*Who do you claim that you like?*)

Revision: First-order predicate logic

- We represent semantics here with first-order logic, which is flexible, well-understood and expressive
- Operators:
  - Negation: `-`, denoting *it is not that case that...*
  - Conjunction: `&`, denoting *and*
  - Disjunction: `|`, denoting *or*
  - Implication: `->`, denoting *if ... , then ...*
- Terms: Individual variables as used in formulae (eg x), and constants (eg *cyril*)
- Predicates: Unary, like intransitive verbs, or binary, often like transitive verbs
- Quantifiers: existential (`exists`) and universal (`all`)
- e.g. 'Every dog disappeared': `all x.(dog(x) -> disappear(x))`

Revision: types for logical expressions

- e denotes an entity: eg `cyril`
- t denotes a formula with a truth value: `bark(cyril)`
- $\langle \sigma, \tau \rangle$ denotes functions from σ to τ
- $\langle e, t \rangle$ denotes expressions from entities to truth values – unary predicates such as `bark` from intransitive verbs
- $\langle e, \langle e, t \rangle \rangle$ denotes expressions from entities to unary predicates – binary predicates such as `chase` from transitive verbs
- $\langle \langle e, t \rangle, t \rangle$ denotes expressions from unary predicates to truth values – from noun phrases

Revision: Lambda Calculus

- Lambda: a tool for abstracting over formulas and producing predicates.
- By applying the λ operator, we can bind a free variable and produce a unary predicate of type $\langle e, t \rangle$.

```
>>> lp.parse(r'\x.(walk(x) & chew_gum(x))')
<LambdaExpression \x.(walk(x) & chew_gum(x))>
```

- Since is is a unary predicate, we can apply to a constant – β-reduction, here giving type t:

```
>>> lp.parse(r'\x.(walk(x) & chew_gum(x)) (gerald)'
    ).simplify()
<AndExpression (walk(gerald) & chew_gum(gerald))>
```

- λ lets us temporarily create partially saturated predicates and control how arguments are applied

Revision: Other useful things with lambda calculus

- Higher-order abstractions
  - If we define variables P and Q of type $\langle e, t\rangle$ to use in lambda abstractions, we can create predicates that accept **other** lambda abstracts as arguments.
  - e.g. `\P.P(angus)` (type $\langle\langle e, t\rangle, t\rangle$) can take another lambda abstract such as `\x.walk(x)` (type $\langle e, t\rangle$) as an argument, giving `walk(angus)`
- These higher-order abstractions are used in **type-raising**:
  - A function F of type $\langle x, y\rangle$ takes x as an argument and outputs y.
  - Type-raising changes a primitive of type x into a function $\langle\langle x, y\rangle, y\rangle$
  - We can now apply the type-raised primitive to argument F.
  - E.g. `angus` $\rightarrow$ `\P.P(angus)` – now an NP takes a VP as its argument, instead of the other way around.
  - Used in constructing many semantic rules.

Revision: Compositionality, or Frege's Principle

Principle of Compositionality

The meaning of the whole is a a function of the meaning of the parts and the way they are syntactically combined

- This means that:
  - If we have sensible semantics for the individual tokens
  - And we combine that semantics into sensible semantics for the constituents containing those tokens
  - We should be able to combine *that* semantics into sensible semantics for the entire sentence.
- Of course coming up with 'sensible semantics' and ways to combine it is non-trivial.

A familiar-looking grammar with semantics

```
S[SEM=<?np(?vp)>]   -> NP[SEM=?np] VP[SEM=?vp]
NP[SEM=<?det(?n)>] -> Det[SEM=?det] N[SEM=?n]
NP[SEM=?pn]         -> PropN[SEM=?pn]
VP[SEM=?v]          -> IV[SEM=?v]
VP[SEM=<?v(?np)>]   -> TV[SEM=?v] NP[SEM=?np]

Det[SEM=<\Q P.exists x.(Q(x) & P(x))>] -> 'a'
Det[SEM=<\Q P.all x.(Q(x) -> P(x))>] -> 'every'
Det[SEM=<\Q P.all x.(Q(x) -> P(x))>] -> 'all'
Det[SEM=<\Q P.all x.(P(x) -> Q(x))>] -> 'only'
PropN[SEM=<\P.P(cyril)>]             -> 'Cyril'
PropN[SEM=<\P.P(angus)>]             -> 'Angus'
PropN[SEM=<\P.P(irene)>]             -> 'Irene'
N[SEM=<\x.dog(x)>]               -> 'dog' | 'dogs'
N[SEM=<\x.cat(x)>]               -> 'cat' | 'cats'
IV[SEM=<\x.bark(x)>]             -> 'barks' | 'bark'
TV[SEM=<\X y.X(\x.chase(y, x))>] -> 'chases' | 'chase'
TV[SEM=<\X y.X(\x.hate(y, x))>]  -> 'hates' | 'hate'
```

Putting interesting syntax back (1)

```
S[SEM=<?np(?vp)>]  -> NP[NUM=?n, SEM=?np] VP[NUM=?n, SEM=?vp]
NP[NUM=?n, SEM=<?d(?nom)>] -> Det[NUM=?n, SEM=?d] N[NUM=?n, SEM=?nom]
NP[NUM=sg, SEM=?pn]        -> PropN[SEM=?pn]
VP[NUM=?n, SEM=?v]         -> V[NUM=?n, SUBCAT=intrans, SEM=?v]
VP[NUM=?n, SEM=<?v(?np)>]  -> V[NUM=?n, SUBCAT=trans, SEM=?v] NP[SEM=?np]
```

Putting interesting syntax back (2)

```
Det[NUM=sg, SEM=<\Q P.exists x.(Q(x) & P(x))>] -> 'a'
Det[NUM=sg, SEM=<\Q P.all x.(Q(x) -> P(x))>] -> 'every'
Det[NUM=pl, SEM=<\Q P.all x.(Q(x) -> P(x))>] -> 'all'
Det[NUM=pl, SEM=<\Q P.all x.(P(x) -> Q(x))>] -> 'only'
PropN[SEM=<\P.P(cyril)>]                  -> 'Cyril'
PropN[SEM=<\P.P(angus)>]                  -> 'Angus'
PropN[SEM=<\P.P(irene)>]                  -> 'Irene'
N[NUM=sg, SEM=<\x.dog(x)>]                -> 'dog'
N[NUM=pl, SEM=<\x.dog(x)>]                -> 'dogs'
N[NUM=sg, SEM=<\x.cat(x)>]                -> 'cat'
N[NUM=pl, SEM=<\x.cat(x)>]                -> 'cats'
V[NUM=sg, SUBCAT=intrans, SEM=<\x.bark(x)>]  -> 'barks'
V[NUM=pl, SUBCAT=intrans, SEM=<\x.bark(x)>]  -> 'bark'
V[NUM=sg, SUBCAT=trans, SEM=<\X y.X(\x.chase(y, x))>] -> 'chases'
V[NUM=pl, SUBCAT=trans, SEM=<\X y.X(\x.chase(y, x))>] -> 'chase'
V[NUM=sg, SUBCAT=trans, SEM=<\X y.X(\x.hate(y, x))>]  -> 'hates'
V[NUM=pl, SUBCAT=trans, SEM=<\X y.X(\x.hate(y, x))>]  -> 'hate'
```

Stepping back: what is semantics good for though?

- We can build these abstract syntactic structures with a parser.
- They show how a sentence is put together but not (directly) what it *means*
- When we construct the semantics of a sentence, we're creating a representation of its meaning.
- What if we could do the inverse as well? Construct sentences from some semantics
- If a computer can extract a meaning representation from free text and use it extract knowledge, perform reasoning and produce new sentences, is it understanding language? Thinking?

Grammar Engineering

- We know that a grammar is a set of syntactic rules which we can use to decompose sentences of natural language.
- Grammar engineering is the task of constructing this set of rules.
- We use linguistic knowledge and observation to construct the grammar.
- Later on, we'll get a better idea about the 'engineering' part.

Grammar engineering considerations

- We have to balance a whole lot of conflicting tensions between parsing the right sentences but not allowing incorrect ones.
- Parsing a sentence and determining its grammaticality is interesting especially for (computational) linguists
- But parsing is not always the final objective – we often want to use it as input for some other task.
- This may come from the parse tree, or it may be useful to have precomputed semantic information
- We can also use semantics to judge whether we have found a good enough parse, instead of directly evaluating the parse tree.
- For these reasons, grammar engineering can also spill into semantics.

The grammar engineering project

- You'll start with the following grammar

```
% start S

S -> NP[NUM=?n] VP[NUM=?n]

NP[NUM=?n] -> PropN[NUM=?n]
NP[NUM=?n] -> Det[NUM=?n] N[NUM=?n]
NP[NUM=pl] -> N[NUM=pl]

VP[NUM=?n] -> V[SUBCAT=intrans, NUM=?n]
VP[NUM=?n] -> V[SUBCAT=trans, NUM=?n] NP
VP[NUM=?n] -> V[SUBCAT=clause, NUM=?n] S
```

(...continued)

The grammar engineering project

```
Det[NUM=sg] -> 'a' | 'this' | 'every'
Det[NUM=pl] -> 'these' | 'all' | 'several' | 'some'
Det -> 'the'

PropN[NUM=sg]-> 'Kim' | 'Jody'
N[NUM=sg] -> 'dog' | 'girl' | 'car' | 'child'
N[NUM=pl] -> 'dogs' | 'girls' | 'cars' | 'children'

V[SUBCAT=intrans, NUM=sg] -> 'disappears' | 'walks'
V[SUBCAT=trans, NUM=sg] -> 'sees' | 'likes'
V[SUBCAT=clause, NUM=sg] -> 'says' | 'claims'

V[SUBCAT=intrans, NUM=pl] -> 'disappear' | 'walk'
V[SUBCAT=trans, NUM=pl] -> 'see' | 'like'
V[SUBCAT=clause, NUM=pl] -> 'say' | 'claim'

V[SUBCAT=intrans, NUM=?n] -> 'disappeared' | 'walked'
V[SUBCAT=trans, NUM=?n] -> 'saw' | 'liked'
V[SUBCAT=clause, NUM=?n] -> 'said' | 'claimed'
```

Useful concepts: test sentences

- Test sentences are simply a set of sentences that we want our grammar to parse
- (They also include sentences we think should not be parseable, but more on that later)
- The more sentences we can parse from some set, the better the **coverage** of the grammar
- Having a suite of test sentences enables us to determine quickly whether our latest grammar rule has improved coverage, or taken us backwards
- A small test-set can help us to think about particular language phenomena we're interested in.
- The test suite can also include expected parse trees (or semantics) – but we won't do that here.

Creating test sentences

- A good set of test sentences should try and find things the grammar can't parse
- Within limits of course – including out-of-vocabulary items is uninteresting
- But we should try and use as wide a range of lexical items as possible
  - especially trying to cover those with different feature values
- We also want to make sure it includes the constructions the grammar covers
  - or those we want the grammar to cover soon
- We use (someone's) native-speaker intuition as well as our knowledge of the small grammar's capabilities

Example test sentences

- For this grammar, we might think of the following test sentences:
  - *Kim disappeared*
  - *these dogs walk*
  - *Jody saw a car*
  - *Kim likes all children*
  - *some dogs see Jody*
  - *the girl claims several children saw the car*
  - *every child said Kim disappeared*
  - *Kim said all children like this car*

Handling conjunctions

- Suppose we want to handle conjunctions like ‘and’.
- We might think of the following extra test cases:
  - *Kim and Jody walked*
  - *several children saw some dogs and a car*
  - *Kim says every child and several dogs disappeared*
  - *Jody and Kim like these cars*

More handling conjunctions

- What about similar constructions like:
  - *Some children and dogs disappeared*
  - *Jody likes all cars and dogs*
- `N -> N Conj N`
- And other slightly different ones like
  - *the child walked and disappeared*
  - *the dog saw Kim and disappeared*
  - *the girl sees and likes Kim*
- `VP -> VP Conj VP`
- `V -> V Conj V`

What about other forms of co-ordination?

- Gapping constructions:
  - In some circumstances we can omit the verb in conjoined VPs
  - *Some saw problems, and others opportunities*
  - or *Kim saw some dogs and Jody some cars*
- Somewhat similar:
  - *The child gave Jody a dog and Kim a car*

Compromises in Grammar Engineering

- Given enough time and patience, we could write a grammar to cover almost all syntactic constructions
- Given enough processing power and memory, we could use such a grammar for parsing
- In the real world, we usually want to:
    1. Capture as much real-world language as possible to maximise coverage
    2. With as small and elegant a grammar as we can make
    3. Although what we find interesting (or what the project specifies) may also be important
- So we can get a grammar as good as possible as quickly as possible – a small grammar can probably handle a large amount of the language.
- In later iterations, we'll probably put in more effort to handle rarer phenomena.

Deciding on compromises

- So, we need to choose which syntactic constructions we're going to handle first
- (2) suggests we should handle those constructions which can be most simply handled
- (1) suggests we should handle those constructions which are most frequently seen in observed language
- (3) suggests we should do whatever we want
- There may be some tension between (1) and (2), but they can often co-occur.
- In the project of course, the spec outranks everything else

Thinking about ‘frequent’ constructions

- How can we determine which syntactic constructions are frequent though?
- Analyse some corpus we’re interested in
  - If we haven’t got a grammar, how do we know what constructions are present though?
- If we’re building a grammar for some specialised target application, we can actively seek out sentences that users might enter.
- For our purposes, intuition of frequency will do, but we mainly consider simplicity

More on compromises

- Simplicity and frequency are why we handle intransitive and transitive verbs before handling verbs requiring NP and PP complements like *put*
- Here, we've added some rules for conjunctions that handle what are probably the frequent case
- We suspect that gapping is a) relatively infrequent and b) going to involve complex rules and possibly features percolating through our grammar
- Our grammar doesn't even know about adjectives, adverbs or prepositions yet.
- We should probably go for these lower-hanging juicier fruit first

Exciting Syntactic Phenomena

- Thinking about the project again, what are some of the suggested syntactic phenomena?
- Relative clauses
- Cleft sentences
- Adverbial clauses
- yes/no and *Wh*-questions

Relative clauses

- A **relative clause** is a post-modifier to a noun which relates the noun to some partial verb phrase that it participates in
- The clauses are introduced by a **relative pronoun** which acts as the subject or object of the verb in the relative clause.
  - *I met the man [who [__ invented hip-hop]]*
  - *The guy [who [__ likes souffle]] arrived*
  - *Someone stole the car [that [she wanted __]]*
  - *The singer [who [John likes __]] vanished*
- These are all *restrictive* – they constrain the noun.
- There are also *unrestrictive* clauses: *[The singer]/[Pavarotti], who John likes, vanished*, which add information onto a fully specified NP

Cleft Sentences

- **Cleft sentences** put emphasis on an element of the sentence that would not usually be emphasised, using some grammatical construction and a relative clause
- *it*-clefts have a pattern like *it was X who/that Y*, eg:
  - *It was Joe [who [__ just arrived]]*
  - *It is souffle [that [he dislikes __]]*
  - *It was Karen [who/that [she gave a pair of socks to __]]*
- Don't get confused with the unrestrictive relative clause reading.
- There are also *wh*-clefts (*What he dislikes is souffle*), pseudo-clefts, *all*-clefts, *there*-clefts and more.

Questions

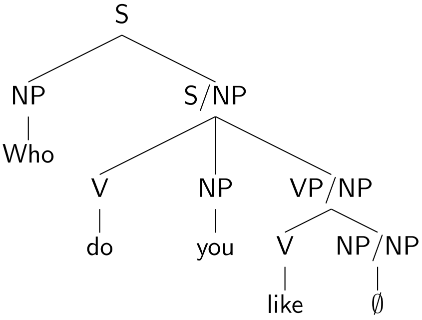
- Standard yes/no questions involve simply inverting the subject and some verb, although the inverted verb is restricted to a particular class of verbs (called?).
  - *You have seen John* / *Have you seen John?*
  - *You like the Pogues* / **Like you the Pogues?* / *Do you like the Pogues?*
- For *wh*-questions, we also have a *wh*-pronoun to stand in for some verb argument (eg subject, object or adjunct):
  - *Who have you seen*? / *Who saw you?* / *Who did I see you with?*
  - *What were you cooking?* / *What is worrying you?* / *What did you drive over?*
- Also *which*, *how*, *why*, *when* etc – each patterns differently.
- Note that we don't always have subject-verb inversion. Why not?

Review: Filler-gap constructions

- Filler-gap constructions have a **gap** instead of a verb argument licensed by a **filler** term – eg *who*.
- We can see this in some questions and relative clauses.
- The filler can be an unboundedly large distance from the gap
  - *Kim knows who [you like __].*
  - *Kim claims you know [who [Jody likes __]]*
  - *Who do [you like __]?*
  - *Who do you claim [you like __]?*
  - *Who do you claim Jody says [you like __]?*

Handling the question form of filler-gap constructions

- We've seen before that we need a few extra features to handle this
- Slash features: `Y/XP` ($\equiv$ `Y[SLASH=XP]` in NLTK), denoting an element of type Y missing an XP.
- So `S/NP` is a sentence missing a noun phrase.
- We'll let *who* take as a sibling a `S/NP`:



- Also note that this requires subject-verb inversion on `S/NP`:
- **Who you like?* / **Who you claim you like?*
- Let's add a feature `INV` on sentences.
- And that only certain verbs can occur in this inversion:
- *Who can you see?* / *Who do you like?*
- **Who see you?* / **Who like you?*
- Let's add a feature `AUX` and call these `[+AUX]` ($\equiv$ `[AUX=+]`)

Some key filler-gap rules

```
NP/NP        ->
S[-INV]      -> NP S/NP
S[-INV]      -> NP[NUM=?n] VP[NUM=?n]
S[-INV]/?x   -> NP[NUM=?n] VP[NUM=?n]/?x
S[+INV]      -> V[+AUX] NP VP
S[+INV]/?x   -> V[+AUX] NP VP/?x

VP[NUM=?n]/?x   -> V[SUBCAT=trans, NUM=?n] NP/?x
VP[NUM=?n]/?x   -> V[SUBCAT=clause, NUM=?n] S/?x
```

Overgeneration

- This grammar can parse ungrammatical sentences like **you does Jody claim she likes?*
- Parsing invalid strings is what we call **overgeneration**, the counterpoint to coverage.
- In some cases it is better to suggest a bad parse than no parse
- But at the same time we want to avoid clearly terrible parses

Why is overgeneration bad?

- Bad parses may be associated with ill-formed semantics, since it is probably for non-existent constructions
- The more parses we are producing the harder the parser needs to work – so we'd prefer only good parses
- The more invalid constructions we accept, the more likely the parser is to suggest it as the best parse
- Some of these problems are only apparent when the grammar gets large, but we should avoid overgeneration from the start

Tension between coverage and overgeneration

- So it's easy to get good coverage if we don't care about overgeneration.
    - We can get 100% coverage of every language on earth with two lines
- The converse is also true
- In a well-designed grammar, we are trying to optimise over both of these (as well as keep our grammar small)
- So checking for overgeneration helps keep us honest with what our grammar can do
- We need to check for both grammatical and ungrammatical sentences in our test cases, reporting an error when the response is unexpected.
- This is why test sentences are useful – we quickly find out if we go backwards

Revising Considerations when Developing a Grammar

- Generally we'll try and get good coverage, while making sure the grammar doesn't 'leak' too much
- So our priority list looks something like
  1. Capture as much real-world language as possible to maximise coverage
  2. Try and avoid admitting ungrammatical sentences
  3. With as small and elegant a grammar as we can make
  4. Looking at the phenomena we find most interesting

Coming up with negative examples

- Coming up with grammatical sentences is relatively easy
- Creating examples that shouldn't parse is a little harder – there are many more ungrammatical sentences
- We want to make sure the 'boundaries' are in the right spot, so one common technique is to minimally perturb some grammatical sentence – eg `NUM=sg` $\rightarrow$ `NUM=pl` on one token
- We should use our linguistic intuition and inspect the grammar to think of cases that we need to be careful of
- We can also randomly substitute a very different lexeme, or throw in some word-salad as a sanity-check

Inspiration for handling new phenomena

- Generally when we add new rules, we think of some target parse tree, and try and add grammatical rules to produce it
- But what if we don't know what the parse tree should look like?
- How can we work out broadly the analysis of some phenomenon?
  - Think of it yourself from scratch.
  - Extend some existing analysis by analogy
  - Read a linguist's analysis for some ideas – eg the Ohio handouts
  - Ask on the discussion forum
  - Try something out and see (actually, you'll always need to do this)
- We can also be happy to accept an imperfect parse tree, as long as it's sensible

Scaling up: how it works with industrial-scale grammars

- So far we haven't worked with grammars with more than about 20 syntactic rules and 50 lexical entries
- Grammars that can handle a sizeable portion of any language have many more rules and lexical items
- We're still concerned with coverage and avoiding over-generation
- But there are additional considerations and complexities
- As well as extra tools to handle these

A real-world hand-coded grammar

- The English Resource Grammar
  - Based on HPSG (not exactly parallel to what we're using here)
  - Standard American English
  - 153 syntactic rules
  - 971 lexical types (like fine-grained parts-of-speech)
  - $\sim$ 24000 manually-specified lexical entries
  - 80-90% coverage in several different domains

Increased Ambiguity

- The more rules we add to the grammar to increase coverage, the more possible parses that are available for a given sentence[1]
- One problem is structural ambiguity:
  - The more syntactic rules we have, the more chance that multiple rules will match to give a spanning parse tree
  - This is an unavoidable side-effect of the quest for coverage
  - For a simple example: PP-attachment – the parser must generate all alternatives for these.
    - *I shot an elephant with a gun in my pyjamas*
    - *I shot an elephant with a gun in its holster*
    - *I shot an elephant with three legs in the head*
    - *I shot an elephant with three legs in the area above its neck*

[1] §8.6: 'Pernicious Ambiguity'

Lexical Ambiguity

- The problem is not in syntactic rules alone
- A simple grammar can have at most one POS per word
- But a robust real-world grammar needs to deal with ubiquitous POS ambiguity.
- We treat *saw* as a verb, but it may refer to the cutting implement
- Even if a POS not particularly common for some word type, the possibility needs to be encoded in the grammar in case we see it
- In combination with structural ambiguity, we get highly ambiguous grammars
- With the ERG:
  - *Time flies like an arrow*: 6 parse trees
  - *Time flies like an arrow with a gold tip*: 41 parse trees
  - One standard corpus: 13 words/sentence with 4000 parses/sentence

Choosing between parses in ambiguous grammars

- If we get 40 parses for a short sentence and 4000 for many 13 word sentences, how do we know we're getting a good parse?
- What we need is a way to rank parses in terms of expected likelihood – so we can return a 'most likely parse'
- With CFGs, one way to achieve this is by augmenting each production with a probability[2]
- We get a probability estimate for a whole parse tree by multiplying the probabilities of each rule used.
- We will accept the highest probability tree as 'best'.

[2] §8.6: 'Weighted Grammar'

Robustly handling a larger lexicon

- Manually specifying a lexicon is very time-consuming
- Even if we have 24000 lexical entries, we'll still encounter unknown words
  - Especially if we move to a new domain
- If we encounter *Jody* **fargled** *the car*, what other language processing technique could help?
  - Most parsers make use of a POS-tagging component for at least unknown words
- What about *We will* **bus** *to Sydney* ? *bus* might not be an unknown word.

Why this is Engineering

- When building a grammar:
  - We're dealing with some idealised set of requirements (the test suite, requirements for tractability)
  - We have to handle the interaction of multiple components (grammars, parsing engines, POS-taggers, word segmenters, morphological analyzers)
  - It often involves collaboration between multiple contributors
  - We have to manage the complex interaction of large set of rules to come up with some imperfect but near-optimal solution

Alternatives to Hand-coding Grammars

- Many NLP researchers don't use hand-coded grammars
- Instead the grammar is determined by examining the trees in some pre-parsed treebank – for English, usually the Penn Treebank
- This provides an easy way to compare different parsers
- And an easy way to learn weights for parse ranking
- As long as you're happy with assumptions the developers of the original treebank made
- And you only care about the tiny handful of languages with a decent-sized treebank.

Considering what computers can do with language

- We can build our own grammars with some relatively simple rules.
- There are a range of manually or automatically created parsers that can extract meaning from most sentences of some target language
- We can also create sentences from some meaning representation
- Some researchers are also looking at automatically creating knowledge bases from web text (e.g. CMU in yesterday's NY times[3])
- So computers can
  - represent the meaning of a sentence
  - perform reasoning over it with respect to real-world knowledge
  - output new sentences on the basis of that reasoning
- Sounds a lot like humans. Are machines starting to think? Should we be worried?

[3] http://www.nytimes.com/2010/10/05/science/05compute.html

Summary

- Grammar Engineering is the task of creating a grammar able to parse sentences of some language
- It aims to maximise coverage, minimise overgeneration and maximise utility for our target domain
- It is usually tested on some constructed test suite and/or some corpus of attested sentences
- When we scale up to broad-coverage grammars, we can get good coverage
- But we also need to deal with ambiguity, complex rule interactions and unknown words in unseen text