Natural Language Systems

Todd Davies

December 29, 2015

Introduction

Enabling computers to use 'natural language' (the kind of language that people use to communicate with one another) is becoming more and more important. It allows people to communicate with them without having to use strange artificial languages and awkward devices like keyboards and mice; and it allows the computer to access the enormous amount of material that is stored as natural language text on the web.

This course provides an introduction to this area, mixing theory (if you don't understand the theory of how language works you cannot possibly write programs that understand it) with practice (if you haven't written or played with tools that embody the theory, you can't get a concrete handle on what the theory means).

Aims

The course unit aims to teach the techniques required to extend the theoretical principles of computational linguistics to applications in a number of critical areas.

- To demonstrate how the essential components of practical NLP systems are built and modified.
- To introduce the principal applications of NLP, including information retrieval & extraction, spoken language access to software services, and machine translation.
- To explain the major challenges in processing large-scale, real-world natural language.
- To explain the principles underlying speech recognition and synthesis, and to explore the power of 'black box' tools for these tasks.
- To give students an understanding of the issues involved in evaluating NLP systems.

Contents

1 Introduction

We want computers to be able to interact with us, just like we interact with them. This involves having them understand written text and voiced speech, as well as being able to synthesise speech and text themselves. This includes things like translation text and searching for key words in text.

A computer or a suite of programs that can do all of this is the goal for Natural Language Systems. The catch is, that language is hard and complicated, and to make computers do the things we want them to, we need to know how3 language works, and express this as an executable program.

Language is the representation of ideas, and the linkage of different ideas together in such a way as to create new ideas. In order to understand any one sentence (a sentence usually corresponds to one idea, event or action), we have to understand what each symbol in the language means in isolation, and understand how they're connected, and what the connections to do change the meaning of the ideas.

Many factors affect the meaning of a sentence, but the connection between words is always hierarchical, and we can represent sentences as trees:

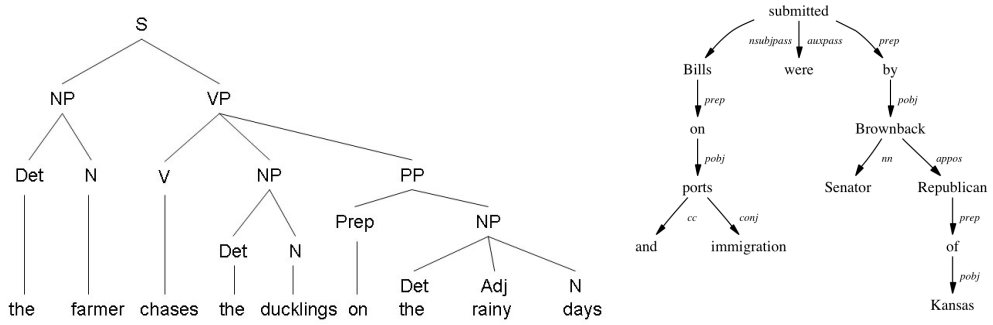


Figure 1: The left image is a phase structure tree, and the right image is a dependency tree.

A parse tree is all well and good, but to a computer, this is only slightly more useful than the original text. Though we have extracted some information out of the text, we still just have a hierarchy of words, but we want a hierarchy of ideas.

Having ideas instead of words allows us to infer more than what the text literally says:

- I'm fixing my motorbike → This person possesses a motorbike, and it is currently broken.
- The cake smells good → There is cake somewhere. Somebody is close enough to smell it.

But how can we do that?

2 Structural analysis

It is possibly to try and find out the meaning of a word simply by looking at what letters it is made up of. One way to do this is to split a word into **morphemes**, which are the most basic meaning-carrying components of a word, and try to associate a meaning with each. For example *undone* could be split into *un* and *done*, and meaning associated with each.

2.1 Tries

Tries are very handy datastructures for technical interviews, you should read up on them an implement one!

In order to examine the syntactic and semantic properties of the words, we need to represent them in the computer. A common way to do this is with a *trie*:

Tries are very memory efficient, since they if multiple words share the same prefix, then the prefix is only stored once in memory. Tries have a lookup time of $O(m)$, where m is the length of the word, which is quite good, and is better than a hash table in terms of speed in some cases. If you're stupid enough to represent your dictionary as a list of words, then you can do a binary search if its ordered (worst case $O(log(n) * m)$ comparisons (the m comes from having to possibly compare each character in the word)), or a linear search if it isn't ordered ($O(m * n)$ in the worst case!).

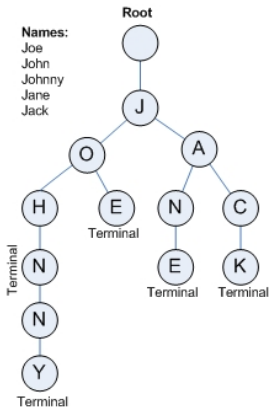


Figure 2: A trie storing some names.

2.2 Spelling rules

We want to understand why combining *big* and *est* produces *biggest* with an extra *g*. Why isn't it *bigest*? The reason why we want to understand this, is so we can go from a word that we're processing in text, and pick it apart into its components so we can better understand it.

That is to say, we're going from *biggest* to *big* + *est*.

The format of the rules we're using in the course is as follows:

```
[from] ==> [to]: [prevContext] _ [nextContext];
```

You can use `cX` and `vX` where `X` is an integer, and `c/v` denotes a consonant or vowel inside the context brackets.

For example, if we had a rule like:

```
[g] ==> []: [g] _ [e,s,t];
```

It would turn *biggest* into *big* + *est*.

2.3 Categorical descriptions

Even if we find the meaning of every word by splitting it info morphemes, we just end up with a collection of words that we know the construction of, but we're still no closer to understanding a sentence.

In order to find the relationships between words in a sentence, we should use the approach in Figure 1, and try to fit the words into some tree structure.

One way of doing this, is to specify lots of different forms that a sentence can take. For example 'noun verb noun' might describe a sentence such as 'Todd writes notes'. Providing that we have some *prototype* for a sentence that fits the words that we've been given, we can ascertain that we can in fact make a sentence out of these words.

There is a better approach though. Each word in a sentence changes its meaning in some way, and we can put words into buckets according to how the meaning of the sentence is changed; for example, a verb specifies what kind of event is happening. Having a verb on its own doesn't do us much good; we will know that *something* is happening, but not where, why, what, when etc. Verbs need other parts of sentence around them to make them work.

We can treat each word as part of a jigsaw, specifying what other words or phrases it needs in order to have meaning, and then fit the pieces together according to some schema.

We need to specify a schema with which we can specify what different words require:

```
<word> = <rules>
```

The rules are their own language, which has a few rules. Brackets are treated as they usually are in maths, and the only other special characters are forward and backward slashes. These indicate whether a word or phrase should be before or after the word.

```
writes = (s\np)/np
```

This indicates that the word 'writes' needs a noun phrase (`np`) to its right and then a noun phrase to its left to make a sentence (`s`).

With the right set of rules for each word, we can now parse sentences:

```
writes = (s\np)/np
Todd = np
notes = np
```

Todd	writes	notes
np	(s\np)/np	np
	s\np	
s		

Here, the words 'Todd' and 'notes' are *saturated* since they don't require anything else to make them into complete 'items'. 'writes' is unsaturated, since it needs other stuff to make it into a complete item. Word rules of these kind are called **Categorical Descriptions**.

2.4 Morphology

Now we've figured out how to decompose a word into morphemes using spelling rules, and we can fit these words into a sentence. However, we also want to know the precise meaning for each word (this helps when arranging them in a sentence too). We can do this by looking at each morpheme.

There are two types of morphology that we're going to look at:

Inflectional morphology
: This is when the stem of the word is incomplete, and other morphemes provide more information to specify exactly what we mean:

  - 'sing' + 'ing' = verb + present participle
  - 'work' + 'ed' = verb + past participle
  - 'work' + '' = noun + singular

Derivational morphology
: This is where the meaning of the stem is significantly changed by other morphemes. For example, 'smelly' could be combined with 'er' to give 'smellier', or 'est' to give 'smelliest'. Obviously we need spelling rules to do this correctly!

We need a way to specify what words can have what suffixes/affixes. For example, 'conscript' can be combined with 'tion' to give 'conscription', but not 'ly' to give 'conscriptly'.

Furthermore, there are spelling rules concerned with adding bits onto words; as we saw before, 'smelly' becomes 'smellier', not 'smellyer'. We will come across this later though.

It turns out that composing morphemes is similar to composing words. We can use the same notation:

```
'conscript' = noun>agr
'conscript' = verb>tns
'tion' = (noun>agr)<(verb>tns)
'ing' = tns
'ed' = tns
's' = agr
'' = agr
```

These descriptions allow you to construct the following words:

- conscript (noun)
- conscripts (noun)
- conscription (noun)
- conscriptions (noun)
- conscripting (verb)
- conscripted (verb)

However, the rules are not perfect, and will also allow you to make:

- conscriptingtion (noun)
- conscriptedtion (noun)
- conscriptingtions (noun)
- conscriptedtions (noun)

We can also make rules cancel out. If we have a rule that is 'verb>tns' and another that is 'tns>agr' then we can make a 'verb>agr' from them.

To see a worked example of these rules, and how they're applied, look on slides $64 - 110$ in the course notes.

One thing that is important to note, is that we can process words from left to right, and don't have to back-track. This means that processing is in linear time, which is fantastic (though we need a big dictionary of words, which is rather less fantastic).

2.5 Unknown words

Now we can read in a sentence, parse each word and extract the relations between the words to produce a meaningful parse tree, great! But... what if we don't have a word in the input sentence in our dictionary? We won't be able to fit it into our parse tree since it won't have any meaning to us. Looking at the morphemes doesn't tell us too much; having 'ing' at the end means that a word is probably a verb and is in the present tense, but doesn't tell us what's actually going on.

There are two different classes of classes of words *open* and *closed*. Open classes are verbs, adjectives, nouns etc where there are lots of them and you can easily add more (new nouns are created all the time). Closed classes are things like prepositions ('in', 'on'), and auxiliaries ('be', 'have', 'would') where you rarely if ever get new words being added.

So, if we cannot recognise a word using the morphological rules we used before, then we should **back off** to using a more robust but less accurate strategy. Lets define exactly what robust and accurate mean:

Robust
: This is when a system always gives an answer, even if the answer might be wrong. It's sometimes a good idea have a robust system, since then you can always take *some* action.

Accurate
: An accurate system always gets the answer 'right' when asked a question.

A common strategy in Natural Language Processing is to use a system that is highly accurate but less robust, but fall back on a less accurate but more robust system when the first doesn't give an answer.

So, we need to build a robust word recogniser for unknown words. To do this, we need to know what word it is, and what part of speech tag it has (noun, verb etc).

2.5.1 Stemming

If you were to come across the word 'blodge', you probably wouldn't know what it means (although there are, as always a number of definitions in the Urban Dictionary). If a word like 'blodge' were to appear in the middle of a sentence, you could probably still extract some information from it:

- I went to blodge → Blodge is a place.
- I went to blodged it → Blodging is a thing you can do. The blodging happened in the past.
- The car was blodge → You can describe something as having the property/quality 'blodge'.

The **Porter Stemmer** is a set of rules that tell you what bits of a word you can remove to get the stem of the word that is common to all forms of the word. If we run the porter stemmer on 'blodge', 'blodging' and 'blodged', then we get the stem as 'blodg'. Now we can at least identify all instances of 'blodg*' as the same word.

2.6 Part of speech tagging

When we look up a word in a dictionary, we get something like this:

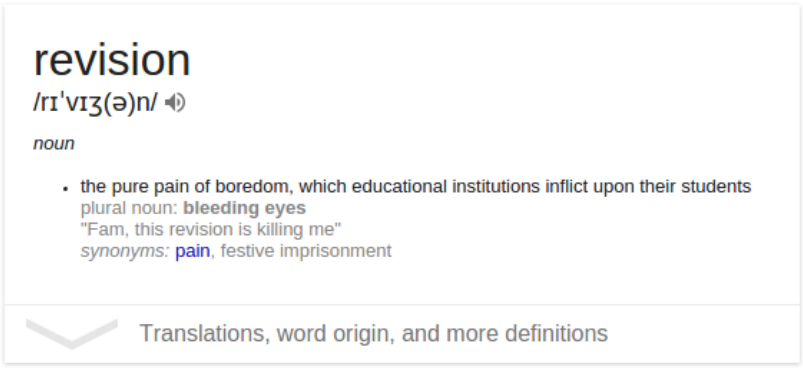


Figure 3: A dictionary definition of the word 'revision'.

Not only does a dictionary give us the definition of a word, but it also give us its part of speech tag. In the case of Figure 3 it is a noun. We need the POS tag in order to work out how words are related to each other, so we can put them in their proper place in a parse tree.

There are a number of ways to do part of speech tagging (this is what I'm doing for my third year project, and it's a rabbit hole that you don't want to go down if you're trying to revise for exams). Here are the ones you need to know for this course:

One thing to be aware of with using corpuses, is that they are often ordered. If you use the first N million words for training and the next M million for testing, then you could end up with disastrous results, because different areas of the corpus will be about different genres, and will contain different words.

Dictionary look up

The easiest way to do POS tagging (but also the worst), is to get a big massive dictionary, and find the part of speech tag for each word. Then, when you need to tag a word, then you just look it up in your big list.

However, you need lots and lots and lots of words to do this, and getting the tags for all of the words is a big task. Furthermore, if we're trying to handle unknown words, then by definition, we've not seen them before and they won't be in our dictionary.

Obviously, the bigger your list of words, the fewer words you'll need to handle. The British National Corpus has 100 million words in, which is in the right ballpark, but it is poorly tagged (the error rate is above 1% in parts). Other corpuses are available, but are either smaller, or you have to pay for them. In order to use the BNC (or any corpus) as a dictionary, you count how many times each word is tagged with each part of speech tag, and then assign the most common tag.

There are two main advantages to this method; first of all, it's really simple, and computationally easy. Using a sensible hash table, lookup is $O(1)$ and training is $O(n)$. It's so fast that the bottleneck is usually the IO operations of reading (and decoding) the corpus. The second advantage, is that you get alternative tags for words. A word might be in the corpus 50 times as a verb, but 20 times as a noun, and you can use this information in your processing.

As a **backoff** technique, you can also keep track of the POS probabilities for the last n letters of the word, so if the word isn't in your dictionary, you can use the last n letters of it to determine what tag to assign.

Transition probabilities and HMM's

HMM's use probabilities and context to determine what tag a word should get. To train a HMM, we collect statistics on what part of speech tags *follow and precede* other ones. For example, the probability of a noun being followed by another noun might be quite high, because of names (e.g. Todd Davies), and adjectives usually lead onto nouns, or other adjectives (e.g. hungry bored Todd).

These are called **bigram** probabilities (because you have information about following and preceding tags). A HMM based POS tagger works as follows:

- For each word, use a corpus to calculate how likely it is to belong to each class (just like we did for the dictionary look up approach). These are the emission probabilities.
- For each tag that we have found that might correspond to a specific word, look at the tag we assigned to the previous (and possibly next) word(s) and use the transition probabilities to choose the most likely tag.

We want to find the most likely sequence of tags for the whole sentence, which is why the hidden markov model works well. However, training can be slow and its hard to make use of the backward transition probabilities.

I think this is more important for the POS tagging part of the course than HMM's are.

Alternately, you can use no HMM at all, by using a function to evaluate the best next tag:

$$\begin{aligned} &F(dict(Word, tag_i), \\ &\quad \sum_k (P(tag(prevWord) = tag_k) \times P(tag_k \rightarrow tag_i)), \\ &\quad \sum_k (dict(nextWord, tag_k) \times P(tag_i \leftarrow tag_k))) \end{aligned}$$

You can define F to be anything, but in the course notes:

$$F(dictProbability, forward, backward) = \sqrt{dictProbability} \times (forward + backward)$$

Transformation Based Learning

This is the subject of my third year project, when I've finished it, I might remember to update these notes with a link to the code/my dissertation.

Doing statistical tagging is pretty tedious, and requires a lot of text. Is there another way of doing POS tagging? Maybe one that could require less data?

This is very possible, as was demonstrated in Eric Brill's PhD thesis in 1995. The idea is to have a *base tagger*, which is your first guess at what the tag is (this is any POS tagger, like the ones above for example), and then have rules that can correct mistakes in the output.

This presents its own problems though. For example, how do you generate rules? The solution is this:

1. Tag a small part of a corpus by hand, so the tags are 100% (or very close to it) accurate.
2. Then you get come up with a set of rule *templates*. Templates (in this course) are of the form `#name(entity1, entity2, entity3): oldTag > newTag if condition;` A set of templates looks something like this:
   - `#t0(T1,T2): T1 > T2 if tag[0] = T1`
     Change tag T1 to tag T2 in every instance.

- `#t1(T1,T2,T3): T1 > T2 if tag[-1] = T3`
  Change tag T1 to tag T2 if the previous tag was T3.
- `#w0(T1,T2,W1): T1 > T2 if word[0] = W1`
  Change tag T1 to tag T2 if the current word is equal to W1.

3. Tag the corpus with the base tagger (something like a dictionary lookup tagger, or transition probability tagger) plus any rules you've generated so far, and then find the errors that it made. For every error, **instantiate a rule based on the context of the error**.
4. Now you have lots of rules, you score each one by how many times it would have made a good change, and how many times it would have made a bad edit if you ran it on the whole corpus.
5. Select the top rule, and run steps 3-4 again until you have enough rules.

I'm not sure if I like the format of the rule templates here, since they require you to specify that you're turning one specific tag into another. An example of an instantiated rule is:

```
#t0(UN,NN,UN): UN > NN if tag[0]=UN;
```

This says turn every 'UN' into a 'NN'. What if we wanted to turn any tag into a 'NN' if it was preceded by a 'UN' though? We couldn't do this with this specific rule set, unless you had wildcards or something, e.g.:

```
#t1(*,NN,UN): * > NN if tag[-1]=UN;
```

When you're assigning a score to each rule, you should give a gross score and a net score. The gross score is how many problems the rule would fix, but the net score is how many problems it would fix minus the number of correct taggings it wrongly breaks. You might end up with something like this:

Rule ranking	Gross	Net
1	500	300
2	450	240
3	412	234
4	375	121
5	297	100
⋮	⋮	⋮

A brill tagger isn't really a part of speech tagger on its own, but it does go a long way to making existing POS taggers better. Depending on the characteristics of the base tagger, you can get anywhere from half a percentage extra accuracy, to a full ten percent extra. Remember though, if the base tagger is already 98% accurate, half a percent is a big improvement!

Confusion matrices No matter how good our POS tagging algorithms are, they're always going to get *something* wrong. Confusion matrices are a way to spot common errors that our taggers make. It is a table showing the correct tags along one side, and the tags the POS tagger came up with along the other. Something like in Table 1.

	NN	VB	PUN	...
NN	80	4	0	...
VB	12	74	1	...
PUN	0	0	12	...
⋮	⋮	⋮	⋮	⋱

Table 1: A sample confusion matrix. Along the top is the number of 'correct' tags, along the side is the number of tags we predicted.

Apart from using them to gloat about your awesome POS tagger to your natural language systems friends, confusion matrices are useful for improving the accuracy of your tagger. If you have multiple taggers, each with a different confusion matrix, then you can use the values in the matrix to determine which one tagger to believe on any word.

If Tagger A says a word is a noun, but Tagger B another says its a determiner, then what do you do? You can look at the confusion matrix, and see that 93% of the time, the Tagger A gets nouns right, but 98% of the time Tagger B gets determiners right, then you should go with Tagger B. It's kind of like if you herded a load of lecturers into a room and started asking them CS questions. If you asked a question about complexity theory, you're more likely to get a good answer from somebody who teaches an advanced algorithms course than you are from somebody who teaches a UX course.

NLP stands for Natural Language Processing, not Neuro-Linguistic Programming...

Special cases Sometimes, a POS tag is very ambiguous; the kind of thing that would have linguists arguing for ages trying to determine which was right. In these cases, we don't want to assign the wrong tag and mis-lead whatever system is using our tags further down the NLP pipeline. As a solution, we could define some special cases (the word 'that' is hard to tag, for example), and if we're not sure about a tag, make the tag equal to the word. This technique is called **underspecification**.

2.7 Regular expressions

Now we can find the part of speech tag for any word (or at least give a good guess for it), we can start to assign structure to the sentence. Given the words of a sentence and their part of speech tags, using a grammar to parse sentences can be done in $O(n^3)$ time **so long as lecixal ambiguity and out- of- place words are ignored**. The trouble with this, is that both these things really matter, and an algorithms that doesn't work, isn't very much use at all.

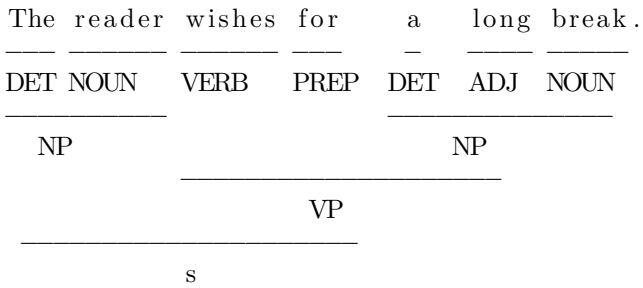
So, when our grammar does break down in those cases, we need to be able to *back off* to another more robust technique. This could be regular expressions. Say a noun phrase is made of an optional determiner, zero or more adjectives and then a noun:

```
NP = DET? ADJ* NOUN
```

A verb phrase is made of a verb and either one or two noun phrases, while a sentence is made of a noun phrase and a verb phrase:

```
VP = (VERB PREP? NP) | (VERB PREP? NP NP)
s = NP VP
```

Given a string that was tagged (correctly), you could apply the regular expression to the string to get the sentences:



If we tag a sentence from the BNC, and run it through a simple regex, then we can efficiently and effectively parse the sentence. First we need to go from the BNC tags (they have different types of nouns, adjectives etc, looking like 'NN4' for a type 4 noun) to more sensible tags:

```
'noun': 'NN.'
'adj': 'AJ.'
'det0': '(AT|D.).'
```

```
...
```

Then we need to recognise phrases:

```
'nmod': 'adj|noun'
'np0': '((det0? nmod* noun)|name|pron)'
...
```

Then you end up with something like:

<NP><DET>The</DET><NOUN>reader</NOUN></NP><VP><VERB>wishes</VERB>
<PREP>for</PREP><NP><DET>a</DET><ADJ>long</ADJ><NOUN>break</NOUN>
<NP></VP><PUN>.</PUN>

Regexes also don't give you alternate options for parsing. They give you one answer (i.e. 'the string you gave matches this structure'), or no answers at all.

This is good, and reasonable fast when your regex is small, but unfortunately, the regex has to get quite large for the output to be a correct parsing of the sentence, and then the regex parsing is quite slow. Furthermore, the output looks horrible, and since regex matching is an opaque thing, you can't debug it very well!

Ultimately, you can recognise a sentence and its structure using regexes, but it's hard to make them parse a sentence just how you like it. Regexes aren't the exactly easy to come up with, and in order to properly define a sentence structure we need recursion (a noun phrase may contains another noun phrase). Unfortunately, regexes don't know about recursion, so we need to cascade them:

```
'a': 'c b'
'b': 'a d'
...
```

2.8 Supertagging

Make me a pull request to fill in this section, do it now!

`http://github.com/Todd-Davies/third-year-notes`

2.9 Deterministic dependency parsing

Using a regex to parse the structure of a sentence isn't widely done. In practice, the way that gets the best results is to use a non-deterministic algorithm. However, (as you may know all to well from `COMP36111`) non- deterministic algorithms don't run very quickly at all on deterministic computing machines, and will often exhibit an exponential runtime.

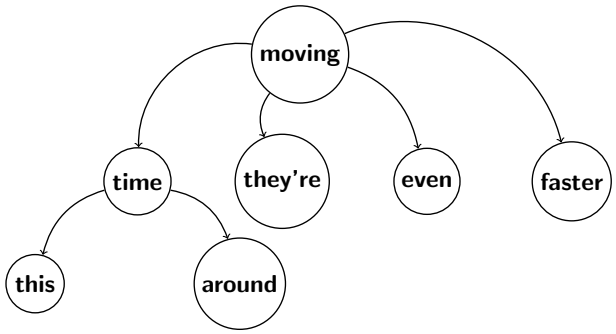
We therefore want to find a deterministic algorithm to extract a sentence structure. When we're thinking about this, we want to optimise for three things:

- Accuracy - get the correct answer
- Robustness - at least *give* an answer
- Speed - try to have a polynomial time complexity

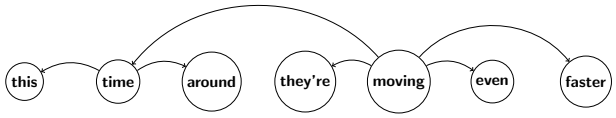
A good approach is to try and make a system that has a good accuracy, then compromise until the other two criteria are satisfied too.

To parse a sentence into a structure, we can recognise the following points:

- Each word has exactly one parent (except the root word)
- You can draw arrows between words in a sentence, and none of the arrows will cross (see Figure 4b for an explanation).



(a)



(b)

Figure 4: (a)A dependency tree for the sentence 'This time around they're moving even faster'. (b) An alternative representation of the dependency tree. Note how none of the lines cross.

2.9.1 MALT

We can come up with a linear time algorithm for comping up with dependency relations (i.e. the arrows on Figure 4b). It uses three data structures:

A **list** of the input words (we treat it like a queue).

A **stack** of words that we've looked at, but haven't yet assigned a parent

A **set** of dependency relations for the output.

The algorithm can do three things:

Shift: Moves the word onto the stack

Left arc: Sets the top item on the stack as a dependent of the head of the input list, and removes it from the stack. This item is now gone.

Right arc: Sets the head of the input list as a dependent of the top item on the stack, throw away that item, but then move the top item from the stack to the head of the input list.

Since *shift* removes an item from the input list, the *arc* operations both remove an item from consideration altogether and at most one dependency is produced at every step, we can see that the algorithm runs in both linear time and space.

So, we have a mechanism for producing dependencies, but we don't actually have any logic to drive it yet; we need some rules to determine what actions we should take for each input item.

There are some rules that constitute 'low hanging fruit' here. First of all, we always need to perform exactly one operation on each iteration of the algorithm (since otherwise the state doesn't change). Secondly, sometimes there's only one thing we *can* do. For example, if there's nothing on the stack, then you can only do a *shift*.

In fact, for any (valid) sequence of operations we perform on the input, we will *always* end up with a dependency tree, so our algorithm is **robust**. The accuracy of our algorithm depends on the rules we give it; though we only have a choice of three operations at each iteration of the algorithm, if we do something wrong, then that error will propogate through the tree and we might end up getting it very wrong.

I can't find what MALT stands for, but it seems to have a webpage: `http://maltparser.org/`