A FINITE-STATE PARSER WITH DEPENDENCY STRUCTURE OUTPUT

David Elworthy
Canon Research Centre Europe Ltd., Occam Road, Guildford GU2 5YJ, UK
dahe@acm.org

Abstract

We show how to augment a finite-state grammar with annotations which allow dependency structures to be extracted. There are some difficulties in determinising the grammar, which is an essential step for computational efficiency, but they can be overcome. The parser also allows syntactically ambiguous structures to be packed into a single representation[1].

1 A finite-state parser with dependency output

Finite-state parsing and dependency grammars both offer advantages to implementers of practical NLP systems. Finite-state parsers have good computational properties, typically $O(n^2)$ in the number of words if the finite-state machines are determinised. Finite-state grammars can also be partly modularised by using cascading (Abney, 1996), and can be implemented so as to tolerate ungrammatical input. Dependency grammars, on the other hand, allow a pseudo-semantic analysis, by extracting relationships such as modification between words. In general, dependency grammars can be implemented with $O(n^3)$ complexity, although ungrammaticality and certain constructs can lead to NP complexity (Neuhaus and Bröker, 1997).

We have developed a simple technique for making a finite-state parser produce dependency structures. As well as outputting the phrase bracketing, the parser also stores words in "variables", in response to annotations in the grammar rules. Variables may be unindexed, storing lists of words, or indexed, providing a mapping from words to lists of words. Indexed variables stand for dependency relations; unindexed ones contain non-dependent elements such as the overall head. Each variable has a name, corresponding to the type of dependency relationship. Thus the dependencies for *a big brown dog barked at the man* could be represented as

```
head = barked; agent[barked] = dog; nomMod[dog] = big, brown
spec[dog] = a; pmod[barked] = at; pcomp[at] = man; spec[man] = the
```

Variable assignments are established by augmenting the rules of the finite-state grammar with annotations which indicate the variable into which the corresponding words are to be stored. In the case of indexed variables, the annotation also includes the name of an unindexed variable, indicating the word at which the dependency link begins. Variables can be set from a lexical item or from an existing variable. Here is an example grammar:

```
NP(nhead, spec[], nomMod[])
     = Det:spec{nhead} (Adj:nomMod{nhead})* (N:nomMod{nhead})* N:nhead
VP(vhead, patient[], spec[], nomMod[])
      = V:vhead NP:(patient{vhead}=head,^)?
S(head, agent[], patient[], spec[], nomMod[])
```

[1] A full version of this paper can be obtained from http://research.microsoft.com/~davidelw/. Author's current address: Microsoft Research Ltd., St. George House, 1 Guildhall Street, Cambridge CB2 3NH, UK. Thanks to Aaron Kotcheff, Tony Rose and Amanda Clare.

```
= NP:(agent{head}=nhead,^) VP:head=vhead^
```

The rules can be glossed as follows. In the first rule, for noun phrases, the final noun is stored in the *nhead* variable, the determiner in the specifier (*spec*) of the head, and the adjectives and nouns as modifiers of the head (*nomMod*). The verb phrase rule stores the verb in the *vhead* variable of the VP. If a noun phrase object is present, its head is copied to *patient*, indexed on the VP's head, thus making a dependency link between the verb and its object. $\wedge$ is shorthand for copying all other variables (*spec*, *mod*) up to the VP. The sentence rule links the subject NP as the agent of the verb, and copies the other dependencies and the head of the VP to the sentence's variables[2].

1.1 Implementation

Most aspects of the implementation are straightforward. We wait until a phrase has been recognised, and then set the unindexed variables followed by the indexed ones. This is implemented by maintaining an agenda of what variables need to be set, during execution of the finite-state machines.

In order to obtain $O(n^2)$ complexity, the finite-state machines must be deterministic. Standard algorithms for determinising finite-state machines will not work here, since the transitions are not uniqely determined by the input symbols. Thus, in `Det:spec (N:nomMod{spec})* N:head`, the `N` input can correspond to different variable outputs. The algorithm described by Roche and Schabes (1995) will solve this in most cases, by deferring output symbols are deferred until enough right-hand context is accumulated to decide which output to produce.

2 Ambiguity and packing

Determinising the finite-state machines only allows a single output, even if the input is ambiguous. Lexical ambiguity can be controlled by a pre-processing step such as tagging, but structural ambiguity is more of a problem. However, the use of variables to represent dependency allows a scheme in which structural ambiguities are packed into a single structure. This will not solve all problems to do with structural ambiguity, but two of the more common ones in English – PP attachment and noun compounding – can easily be handled this way.

To construct packed dependency structures, we place a word in the range (value) of more than one indexed variable, so that it can be accessed via multiple dependencies. The grammar below shows how this is done for prepositional phrase attachment ambiguity.

```
NP(nhead) = Det N:nhead
PP(prep, pcomp[], phead) = Prep:prep NP:(pcomp{prep}=nhead, phead=nhead)
PP(prep, pcomp[], phead) = PP:^ PP:(^,pmod{phead}=prep)
NP(nhead, pmod[], pcomp[]) = NP:^ (PP:pmod{nhead}=prep,pcomp=pcomp,pmod=pmod)*
```

The first PP rule builds simple prepositional phrases, in which *phead* provides access to the head of embedded NP, while *pcomp* provides access to it via the preposition. In the second PP rule, a *pmod* link from this *phead* is then made to the preposition of the second PP. When the phrase *the man in the park on a hill* is parsed, *on a hill* is attached in the final structure to both *man* and *park*. A similar approach can be used for noun compounds.

References

Abney, Steven (1996). Partial Parsing via Finite-State Cascades. In *Proceedings of the ESSLLI '96 Robust Parsing Workshop*. Also available from `http://www.sfs.nphil.uni-tuebingen.de/~abney/`.

Neuhaus, Peter and Norbert Bröker (1997). The Complexity of Recognition of Linguistically Adequate Dependency Grammars. In *Proceedings of the 35th ACL and 8th EACL*, pp. 337–343.

Roche, Emmanuel and Yves Schabes (1995). Deterministic Part-of-Speech Tagging with Finite-State Transducers. *Computational Linguistics*, volume 21, number 2, pp. 227–253.

[2]We are assuming, for the sake of this example, that the subject realises the agent role and the object the patient role