Creating a Finite-State Parser with Application Semantics

Owen Rambow
University of Pennsylvania
Philadelphia, PA 19104
USA

Srinivas Bangalore
AT&T Labs – Research
Florham Park, NJ 07932
USA

Tahir Butt
Johns Hopkins University
Baltimore, MD 21218
USA

Alexis Nasr
Université Paris 7
75005 Paris
France

Richard Sproat
AT&T Labs – Research
Florham Park, NJ 07932
USA

rambow@unagi.cis.upenn.edu

Abstract

Parsli is a finite-state (FS) parser which can be tailored to the lexicon, syntax, and semantics of a particular application using a hand-editable declarative lexicon. The lexicon is defined in terms of a lexicalized Tree Adjoining Grammar, which is subsequently mapped to a FS representation. This approach gives the application designer better and easier control over the natural language understanding component than using an off-the-shelf parser. We present results using Parsli on an application that creates 3D-images from typed input.

1 Parsing and Application-Specific Semantics

One type of Natural Language Understanding (NLU) application is exemplified by the database access problem: the user may type in free source language text, but the NLU component must map this text to a fixed set of actions dictated by the underlying application program. We will call such NLU applications "application-semantic NLU". Other examples of application-semantic NLU include interfaces to command-based applications (such as airline reservation systems), often in the guise of dialog systems.

Several general-purpose off-the-shelf (OTS) parsers have become widely available (Lin, 1994; Collins, 1997). For application-semantic NLU, it is possible to use such an OTS parser in conjunction with a post-processor which transfers the output of the parser (be it phrase structure or dependency) to the domain semantics. In addition to mapping the parser output to application semantics, the post-processor often must also "correct" the output of the parser: the parser may be tailored for a particular domain (such as *Wall Street Journal* (WSJ) text), but the new domain presents linguistic constructions not found in the original domain (such as questions). It may also be the case that the OTS parser consistently misanalyzes certain lexemes because they do not occur in the OTS corpus, or occur there with different syntactic properties. While many of the parsers can be retrained, often an annotated corpus is not available in the application domain (since, for example, the application itself is still under development and there is not yet a user community). The process of retraining may also be quite complex in practice. A further disadvantage of this approach is that the post-processor must typically be written by hand, as procedural code. In addition, the application-semantic NLU may not even exploit the strengths of the OTS parser, because the NLU required for the application is not only different (questions), but generally simpler (the WSJ contains very long and syntactically complex sentences which are not likely to be found as input in interactive systems, including dialog systems).

This discussion suggests that we (i) need an easy way to specify application semantics for a parser and (ii) that we do not usually need the full power of a full recursive parser. In this paper, we suggest that application-semantic NLP may be better served by a **lexicalized finite-state (FS) parser**. We present PARSLI, a FS parser which can be tailored to the application semantics using a hand-editable declarative lexicon. This approach gives the application designer better and easier control over the NLU component. Furthermore, while the finite-state approach may not be sufficient for WSJ text (given its syntactic complexity), it is sufficient for most interactive systems, and the advantage in speed offered by FS approaches in more crucial in interactive appli-

cations. Finally, in speech-based systems, the lattice that is output from the speech recognition component can easily used as input to a FS-based parser.

2 Sample Application: WORDSEYE

WORDSEYE (Coyne and Sproat, 2001) is a system for converting English text into three-dimensional graphical scenes that represent that text. WORDSEYE performs syntactic and semantic analysis on the input text, producing a description of the arrangement of objects in a scene. An image is then generated from this scene description. At the core of WORDSEYE is the notion of a "pose", which can be loosely defined as a figure (e.g. a human figure) in a configuration suggestive of a particular action.

For WORDSEYE, the NLP task is thus to map from an input sentence to a representation that the graphics engine can directly interpret in terms of poses. The graphical component can render a fixed set of situations (as determined by its designer); each situation has several actors in situation-specific poses, and each situation can be described linguistically using a given set of verbs. For example, the graphical component may have a way of depicting a commercial transaction, with two humans in particular poses (the buyer and the seller), the goods being purchased, and the payment amount. In English, we have different verbs that can be used to describe this situation (*buy*, *sell*, *cost*, and so on). These verbs have different mappings of their syntactic arguments to the components in the graphical representation. We assume a mapping from syntax to domain semantics, leaving to lexical semantics the question of how such a mapping is devised and derived. (For many applications, such mappings can be derived by hand, with the semantic representation an *ad-hoc* notation.) We show a sample of such mapping in Figure 1. Here, we assume that the graphics engine of WORDSEYE knows how to depict a TRANSACTION when some of the semantic arguments of a transaction (such as CUSTOMER, ITEM, AMOUNT) are specified.

We show some sample transductions in Figure 2. In the output, syntactic constituents are bracketed. Following each argument is information about its grammatical function ("GF=0" for example) and about its semantic role (ITEM for example). If a lexical item has a semantics of its own, the semantics replaces the lexical item (this is the case for verbs), otherwise the lexical item remains in place. In the case of the transitive *cost*, the verbal semantics in Figure 1 specifies an implicit CUSTOMER argument. This is generated when *cost* is used transitively, as can be seen in Figure 2.

3 Mapping Tree Adjoining Grammar to Finite State Machines

What is crucial for being able to define a mapping from words to application semantics is a very abstract notion of grammatical function: in devising such a mapping, we are not interested in how English realizes certain syntactic arguments, i.e., in the phrase structure of the verbal projection. Instead, we just want to be able to refer to syntactic functions, such as *subject* or *indirect object*. Tree Adjoining Grammar (TAG) represents the entire syntactic projection from a lexeme in its elementary structures in an elementary tree; because of this, each elementary tree can be associated with a lexical item (lexicalization, (Joshi and Schabes, 1991)). Each lexical item can be associated with one or more trees which represent the lexeme's valency; these trees are referred to as its *supertags*. In a derivation, substituting or adjoining the tree of one lexeme into that of another creates a direct dependency between them. The syntactic functions are labeled with integers starting with zero (to avoid discussions about names), and are retained across operations such as topicalization, dative shift and passivization.

A TAG consists of a set of elementary trees of two types, initial trees and auxiliary trees. These trees are then combined using two operations, substitution and adjunction. In substitution, an initial tree is appended to a specially marked node with the same label as the initial tree's root node. In adjunction, a non-substitution node is rewritten by an auxiliary tree, which has a specially marked frontier node called the footnode. The effect is to insert the auxiliary tree into the middle of the other tree.

We distinguish two types of auxiliary trees. **Adjunct auxiliary** trees are used for adjuncts; they have the property that the footnode is al-

Verb	Supertag	Verb semantics	Argument semantics
paid	A_nx0Vnx1	transaction	0=Customer 1=Amount
cost	A_nx0Vnx1	transaction	0=Item 1=Amount Implicit=Customer
cost	A_nx0Vnx2nx1	transaction	0=Item 1=Amount 2=Customer
bought, purchased	A_nx0Vnx1	transaction	0=Customer 1=Item
socks	A_NXN	none	none

Figure 1: Sample entries for a commercial transaction situation

In: I bought socks **Out**: ((I) GF=0 AS=CUSTOMER TRANSACTION (socks) GF=1 AS=ITEM)
In:the pajamas cost my mother-in-law 12 dollars **Out**: (((the) pajamas) GF=0 AS=ITEM TRANSACTION ((my) mother-in-law) GF=2 AS=CUSTOMER ((12) dollars) GF=1 AS=AMOUNT)
In: the pajamas cost 12 dollars **Out**: (((the) pajamas) GF=0 AS=ITEM TRANSACTION IMP:CUSTOMER ((12) dollars) GF=1 AS=AMOUNT)

Figure 2: Sample transductions generated by Parsli ("GF" for grammatical function, "AS" for argument semantics, "Imp" for implicit argument)

ways a daughter node of the root node, and the label on these nodes is not, linguistically speaking, part of the projection of the lexical item of that tree. For example, an adjective will project to AdjP, but the root- and footnode of its tree will be labeled NP, since an adjective adjoins to NP. We will refer to the root- and footnode of an adjunct auxiliary tree as its **passive valency structure**. Note that the tree for an adjective also specifies whether it adjoins from the left (footnode on right) or right (footnode on left). **Predicative auxiliary trees** are projected from verbs which subcategorize for clauses. Since a verb projects to a clausal category, and has a node labeled with a clausal category on its frontier (for the argument), the resulting tree can be interpreted as an auxiliary tree, which is useful in analyzing long-distance *wh*-movement (Frank, 2001).

To derive a finite-state transducer (FST) from a TAG, we do a depth-first traversal of each elementary tree (but excluding the passive valency structure, if present) to obtain a sequence of non-terminal nodes. For predicative auxiliary trees, we stop at the footnode. Each node becomes two states of the FST, one state representing the node on the downward traversal on the left side, the other representing the state on the upward traversal, on the right side. For leaf nodes, the two states are juxtaposed. The states are linearly connected with ϵ-transitions, with the left node state of the root node the start state, and its right node state the final state (except for predicative auxiliary trees – see above). To each non-leaf state, we add one self loop transition for each tree in the grammar that can adjoin at that state from the specified direction (i.e., for a state representing a node on the downward traversal, the auxiliary tree must adjoin from the left), labeled with the tree name. For each pair of adjacent states representing a substitution node, we add transitions between them labeled with the names of the trees that can substitute there. We output the number of the grammatical function, and the argument semantics, if any is specified. For the lexical head, we transition on the head, and output the semantics if defined, or simply the lexeme otherwise. There are no other types of leaf nodes since we do not traverse the passive valency structure of adjunct auxiliary tees. At the beginning of each FST, an ϵ-transition outputs an open-bracket, and at the end, an ϵ-transition outputs a close-bracket. The result of this phase of the conversion is a set of FSTs, one per elementary tree of the grammar. We will refer to them as "elementary FSTs".

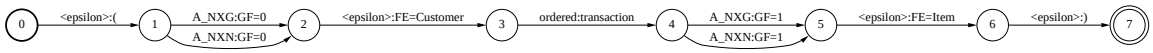

Figure 4: FST corresponding to TAG tree in Figure 3

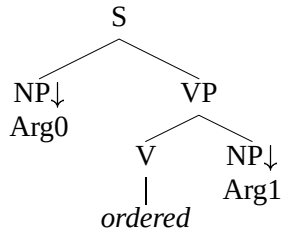

Figure 3: TAG tree for word *ordered*; the downarrow indicates a substitution node for the nominal argument

4 Constructing the Parser

In our approach, each elementary FST describes the syntactic potential of a set of (syntactically similar) words (as explained in Section 3). There are several ways of associating words with FSTs. Since FSTs correspond directly to supertags (i.e., trees in a TAG grammar), the basic way to achieve such a mapping is to list words paired with supertags, along with the desired semantic associated with each argument position (see Figure 1). The parser can also be divided into a lexical machine which transduces words to classes, and a syntactic machine, which transduces classes to semantics. This approach has the advantage of reducing the size of the overall machine since the syntax is factored from the lexicon.

The lexical machine transduces input words to classes. To determine the mapping from word to supertag, we use the lexical probability $p(s|w)$ where w is the word and c the class. These are derived by maximum likelihood estimation from a corpus. Once we have determined for all words which classes we want to pair them with, we create a disjunctive FST for all words associated with a given supertag machine, which transduces the words to the class name. We replaces the class's FST (as determined by its associated supertag(s)) with the disjunctive head FST. The weights on the lexical transitions are the negative logarithm of the emit probability $p(w|c)$ (obtained in the same manner as are the lexical probabilities).

For the syntactic machine, we take each elementary tree machine which corresponds to an initial tree (i.e., a tree which need not be adjoined) and form their union. We then perform a series of iterative replacements; in each iteration, we replace each arc labeled by the name of an elementary tree machine by the lexicalized version of that tree machine. Of course, in each iteration, there are many more replacements than in the previous iteration. We use 5 rounds of iteration; obviously, the number of iterations restrict the syntactic complexity (but not the length) of recognized input. However, because we output brackets in the FSTs, we obtain a parse with full syntactic/lexical semantic (i.e., dependency) structure, not a "shallow parse".

This construction is in many ways similar to similar constructions proposed for CFGs, in particular that of (Nederhof, 2000). One difference is that, since we start from TAG, recursion is already factored, and we need not find cycles in the rules of the grammar.

5 Experimental Results

We present results in which our classes are defined entirely with respect to syntactic behavior. This is because we do not have available an important corpus annotated with semantics. We train on the Wall Street Journal (WSJ) corpus. We evaluate by taking a list of 205 sentences which are chosen at random from entries to WORDSEYE made by the developers (who were testing the graphical component using a different parser). Their average length is 6.3 words. We annotated the sentences by hand for the desired dependency structure, and then compared the structural output of PARSLI to the gold standard (we disregarded the functional and semantic annotations produced by PARSLI). We evaluate performance using **accuracy**, the ration of

n	Correctness	Accuracy	Nb
2	1.00	1.00	12
4	0.83	0.84	30
6	0.70	0.82	121
8	0.62	0.80	178
12	0.59	0.79	202
16	0.58	0.79	204
20	0.58	0.78	205

Figure 5: Results for sentences with n or fewer words; *Nb* refers to the number of sentences in this category

n	Correctness	Accuracy
1	0.58	0.78
2	0.60	0.79
4	0.62	0.81
8	0.69	0.85
12	0.68	0.86
20	0.70	0.87
30	0.73	0.89

Figure 6: Results for n-best analyses

the number of dependency arcs which are correctly found (same head and daughter nodes) in the best parse for each sentence to the number of arcs in the entire test corpus. We also report the percentage of sentences for which we find the correct dependency tree (**correctness**). For our test corpus, we obtain an accuracy of 0.78 and a correctness of 0.58. The average transduction time per sentence (including initialization of the parser) is 0.29 s. Figure 5 shows the dependence of the scores on sentence length. As expected, the longer the sentence, the worse the score.

We can obtain the n-best paths through the FST; the scores for n-best paths are summarized in Figure 6. Since the scores keep increasing, we believe that we can further improve our 1-best results by better choosing the correct path. We intend to adapt the FSTs to use probabilities of attaching particular supertags to other supertags (rather than uniform weights for all attachments) in order to better model the probability of different analyses. Another option, of course, is bilexical probabilities.

6 Discussion and Outlook

We have presented PARSLI, a system that takes a high-level specification of domain lexical semantics and generates a finite-state parser that transduces input to the specified semantics. PARSLI uses Tree Adjoining Grammar as an interface between syntax and lexical semantics. Initial evaluation results are encouraging, and we expect to greatly improve on current 1-best results by using probabilities of syntactic combination. While we have argued that many applications do not need a fully recursive parser, the same approach to using TAG as an intermediate between application semantics and syntax can be used in a chart parser; for a chart parser using the FS machines discussed in this paper, see (Nasr et al., 2002).

References

Michael Collins. 1997. Three generative, lexicalised models for statistical parsing. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics*, Madrid, Spain, July.

Bob Coyne and Richard Sproat. 2001. WordsEye: An automatic text-to-scene conversion system. In *SIGGRAPH 2001*, Los Angeles, CA.

Robert Frank. 2001. *Phrase Structure Composition and Syntactic Dependencies.* MIT Press, Cambridge, Mass.

Aravind K. Joshi and Yves Schabes. 1991. Tree-adjoining grammars and lexicalized grammars. In Maurice Nivat and Andreas Podelski, editors, *Definability and Recognizability of Sets of Trees*. Elsevier.

Dekang Lin. 1994. PRINCIPAR—an efficient, broad-coverage, principle-based parser. In *coling94*, pages 482–488, Kyoto, Japan.

Alexis Nasr, Owen Rambow, John Chen, and Srinivas Bangalore. 2002. Context-free parsing of a tree adjoining grammar using finite-state machines. In *Proceedings of the Sixth International Workshop on tree Adjoining Grammar and related Formalisms (TAG+6)*, Venice, Italy.

Mark-Jan Nederhof. 2000. Practical experiments with regular approximation of context-free languages. *cl*, 26(1):17–44.