FDL: A **F**eature **D**escription **L**anguage for Semantic Role Labeling

Trevor Fountain
doches@mail.utexas.edu
Department of Computer Sciences
The University of Texas at Austin

May 7, 2008

Abstract

In this paper I describe a language for extracting features from parse trees, especially those used in semantic role labelling (SRL). The language, which I call FDL, is designed to be accessible to non-programmers yet powerful enough to express most features found in SRL literature. I argue that such a language is useful both as an engineering platform and as an educational tool. I present the syntax for FDL and describe its implementation, and evaluate its usefulness by investigating how well it expresses novel features used in the CoNLL 2005 shared task.

1 Introduction

A semantic role describes the predicate-argument relationship between a constituent and its predicate. The task of semantic role labelling (SRL), then, is to automatically determine which constituents of a sentence are arguments for a particular predicate, and which roles those arguments fill. For example, the sentence in Figure 1 has two roles for the predicate "eat", "ingestor" and "ingestibles", which are filled by the constituents "I" and "pie", respectively.

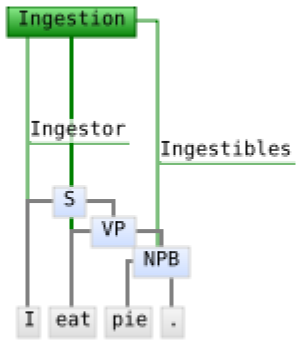

Figure 1: A simple sentence annotated with semantic roles

Typical SRL systems are broken down into three phases: preprocessing, argument recognition, and argument labelling. In the preprocessing step, free (unannotated) text is annotated with syntactic (or semantic) information, which may include anything from a simple syntactic or dependency parse to information about named entities or disambiguated word senses. The latter pair of steps, argument recognition and labelling, are more complicated and generally involve some sort of supervised training. To train an argument recognizer or labeller, features are extracted from a large corpus of data which has been annotated with semantic roles. These features, combined with the gold data from the training corpus, are used as input to a machine learning algorithm.

The task of determining what features are valuable to each task is referred to as feature engineering. Early SRL systems relied on a small number of features (Gildea and Jurafsky [2002]), including voice, constituent phrase type, and the path between constituent and predicate; state-of-the-art systems often utilize much large feature sets (Pradhan et al. [2005]). Each of these features is typically calculate independently of the others; I refer to a method or function which calculates a single feature for a given sentence and constituent-predicate pair as a *feature extractor*.

Feature engineering is a difficult part of building any semantic role labelling (SRL) system. The design of useful features, and the implementation of methods to extract those features from text, has been the focus of a great deal of experimentation in current SRL literature. Indeed, while researchers have settled on a core feature set based around the work of Gildea and Jurafsky [2002] and Pradhan et al. [2004], contemporary systems still rely upon a considerable number of novel features (Carreras and Màrquez [2005]).

This paper presents a description language (FDL) for writing feature extractors in a declarative fashion. Section 2 describes why such a language is useful, section 3 explains why existing tools are poorly suited to this task, and section 7 attempts to evaluate the effectiveness language by determining what percentage of features used during the CoNLL 2005 shared task on SRL can be represented in FDL. An informal description of FDL is included in section 4, along with a detailed example feature in section 5 and a formal specification in the form of a BNF grammar in section 6. Sample features from the Gildea and Jurafsky [2002] and Pradhan et al. [2004] papers are included in appendices A and B, respectively.

2 Motivation

2.1 Engineering

Primarily, FDL is a tool for improving feature engineering. Like any automatic programming system, FDL provides a syntax for writing a program specification (in this case, a feature description) and then uses that specification to generate a more complex program (a feature extractor). Because FDL expressions are so simple and domain-specific, it should be easier for experimenters to write their feature extractors indirectly in FDL than directly in code. Furthermore, FDL is similar in design to other declarative languages like LEX and YACC (tools for generating lexical scanners and parsers, respectively). Like these languages, FDL allows users to specify structures in a high-level fashion, avoiding unnecessary bookkeeping and extraneous code not directly related to the task at hand.

The ease with which FDL can be used to generate complex feature extractors points to another advantage; designing, building, and testing FDL features should be much quicker than manually building equivalent functions. Systems built with FDL features can be prototyped much faster; indeed, it is possible within the Shalmaneser system (Erk and Pado [2006]) in which FDL is embedded to design an entire SRL system with a minimal amount of code. The addition of FDL to the Shalmaneser tools will allow much of the work that would previously have required programming to be completed by non-programmers, expanding the tool's use rbase considerably. A further advantage comes from the portability of FDL expressions; writing feature extractors declaratively alleviates much of the difficulty in porting an SRL system to a new parser or language.

2.2 Education

Many current SRL systems are designed to be as open and flexible as possible, allowing the user to tinker with almost all aspects of the tool chain. These systems are ideal for adaptation into educational tools; the emphasis on open-ness The integration of FDL features into such a system opens up entire new areas of experimentation. Indeed, engineering useful features and determining their effectiveness is often a significant part of the work involved in building a new SRL system; it is only natural to open this avenue of research to non-programmers.

This is analogous to grammar workbenches, which mask the complexity of grammar creation under a friendly (or at least friendlier) user interface. Furthermore, grammar workbenches have been shown to greatly assist users with and without programming experience both in learning new grammar formalisms and in using those formalisms to construct useful grammars (Baldridge et al. [2007]).

2.3 Communication

An additional benefit of using FDL is the ability to communicate the design of feature extractors in a way that is both simple and consistent. Generally, this is done with pseudo-code, but FDL has the additional benefit of being directly compilable. Features described in literature can be run through the FDL compiler to generate a feature extractor without modifying them in any way.

3 Design

FDL must be powerful enough to be useful while remaining simple enough to be easily understandable. If it is too complex then there is no reason not to simply write feature extractors directly; if it is too simple it will be unable to describe common features and again result in requiring experimenters to write feature extractors directly in code. Applying this notion of power versus simplicity illustrates why existing tools are ill-suited to the task of describing feature extractors: Xpath, a language for selecting nodes from an XML document (Clark and DeRose [1999]), and TIGERSearch, a corpus query language, are quite easy to learn but do not allow expressions complex enough to describe common features. Tregex, a tool for applying regular expressions to trees (Levy and Andrew [2006]) comes closer to the necessary level of complexity, but, like TIGERSearch, also doesn't provide a mechanism for extracting data from matched trees.

In order to meet this goal FDL was purposely designed to capture only those features that rely purely on information extracted from the parse tree. More complex information (like named entity recognition or clustered verb senses) can only be used to construct features if it is added to the parse tree by a preprocessing step. This restriction does somewhat limit the possible coverage of FDL, but I believe this is a legitimate line after which it is easier to write extractors directly in code rather than by description. It also provides a useful metric for evaluation; ideally, FDL will be able to describe all features that can be computed using only information from the parse tree.

Obviously, numerous tools exist for extracting information from graphs. These tools, however, tend to have been designed to fill a particular niche, and as such are too domain-specific to be of much use in this particular task. Of the three mentioned above (TIGERSearch, Tregex, and Xpath) only TIGERSearch comes close to describing the necessary information for building feature extractors. TIGERSearch is a query language for extracting information from an annotated corpus – essentially a database query language for graphs (König and Lezius [2000]). However, TIGERSearch expressions can provide only a binary match indicating whether a particular graph matches the expression described. Complex features require additional information; for example, extracting the path between two nodes would be quite impossible within TIGERSearch. FDL's syntax for node descriptors and edge labels is drawn almost directly from TIGERSearch.

FDL expands on TIGERSearch in several ways. Primarily, FDL is a language for extracting data rather than simple matching; it adds a means of retrieving node-specific information from parse trees. Whereas a TIGERSearch expression describing a particular sub-graph will return the entire parse tree of a sentence containing that sub-graph, a comparable FDL expression will return only the requested information from selected nodes within the sub-graph. Furthermore, matching expressions in FDL are significantly more expressive than TIGERSearch: FDL expressions can contain both negation and universally quantified variables, features which are notably absent in TIGERSearch.

4 Informal Language Specification

4.1 Evaluation

FDL expressions are evaluated over directed graphs in which nodes possess various attributes which are drawn from a fixed list. The list of attributes nodes may possess is determined by the pre-processing steps performed.

The search for nodes that match a node descriptor and satisfy the constraints imposed by the current search path (i.e. the tree traversal operators previously encountered while matching a path) proceeds via a left-most depth-first search. If a node descriptor is unbound or bound existentially the search terminates at the first satisfying node; if a descriptor is bound using a universal quantifier, the parse tree is exhaustively searched for matching nodes.

4.2 Definitions

Features are defined using the `feature` keyword, followed by an alphanumeric name, a feature expression, and an optional return vector. If the feature expression can be matched, the variables bound during the matching are used to fill out the return vector, and the feature will return the resulting array. Otherwise, if the expression cannot by matched in the parse tree, the feature returns `false`. If the expression can be matched but no return vector is provided, the feature will simply return `true`.

In general, features are of the form:

Figure 2: General form of features.

```
feature feature_name
    expression
    <return vector>
```

Similarly to features, particular nodes or sets can be defined using the `node` keyword followed by an alphanumeric name, an expression in which at least one variable is bound, and a vector of bound variables to return. Defined nodes can be referenced in features or other node definitions as if they were bound variables, and will be re-computed for different contexts (i.e., when `self` or `target` differ).

Figure 3: Find the least common ancestor between the current and target nodes.

```
node LCA
    [is=self] /* x:[] AND x \* [is=target] AND
    not (x \ y:[] AND y \* [is=self] AND y \* [is=target])
    <x>
```

4.3 Expressions

An expression consists of one or more subgraph expressions joined by logical operators. subgraph expressions descibe individual structures within the parse tree, and are composed of node descriptors joined by tree traversal operators. For example, the expression:

```
[is=self] / [cat='NP'] AND
[is=self] | [cat='PP']
```

represents two subgraphs; one involving the current node and its parent, which has the category 'NP', and one involving the current node and its sibling, which has the category 'PP'. This expression is only true if both of these subgraph expressions can be matched in the current parse tree (i.e. relative to the current assignment of `is=self`).

4.4 Node Descriptors

Node descriptors filter all nodes in the parse tree based on the properties and conditions specified therein. They consist of zero or more `property=value` pairs separated by logical (`and,or`) enclosed by square brackets. `property` is one of `pos,head`, or the special property `is` (described later); `value` is either a quote-delimited string or a regular expression (delimited by forward slashes). `property,value` pairs may be related by either an equals (`=`) or a not-equals (`!=`) sign; properties that are not set have a default value of `nil`. Multiple properties and conjunctions can be grouped with parenthesis; properties and parenthetical expressions may be negated with the `not` keyword. For example:

```
[is=target and (cat="v" or cat="vp")]
```

matches the target node if the target node's part-of-speech is either "v" or "vp". The descriptor returns a boolean indicating whether such a node exists on the current search path. Note that this syntax for matching nodes by their properties is drawn directly from TIGERSearch's feature constraints (König and Lezius [2003]).

The `is` property checks to see if a node matches a pre-defined type; it may take a value of `self`, `root`, or `target` (all of which refer to the corresponding node in the tree) or the name of a bound variable. The constituent under consideration (i.e. the node being considered for role assignment) is assigned the property `is=self`; the node for which this role is being assigned (the predicate) takes the property `is=target`. The root of the parse tree is assigned `is=root`. Obviously, these assignments are relative to each parse tree node.

The properties attached to each node are determined by the choice of parser. For example, a parse tree generated by the Collins parser (Collins [1999]) will have the headword property, whereas a parse generated by a dependency parser (such as Minipar or C&C) may not even have a category property.

The empty descriptor `[]` matches all nodes on the current search path.

4.5 Variable Binding

Nodes matched by descriptors can be bound to named variables by preceding the descriptor with the variable name a' la `varname:[pos = "NP"]`. After binding, variables can be referenced anywhere a node descriptor would be appropriate, or in the return statement. Variables can be bound to sets using the universal quantifier `ALL( x, expression )`, where `x` is the variable to be bound and `x` is bound in `expression`. Variables bound to sets will automatically be expanded if referenced in the return vector. Expanded variables are replaced with a comma-separated list of their member nodes with the requested property; if both nodes `a` and `b` have been bound to the variable `x`, the return vector `<x.cat>` will be expanded into `<a.cat, b.cat>`. For example, consider the feature:

Figure 4: Find the derivation rule for the current node.

```
feature derivation
    parent:[is=self] AND ALL(child, parent \ child:[])
    <parent.cat, "->", child.cat>
```

Because `child` is a set of nodes, this feature may return something like `<NP, ->, NP, PP>`.

4.6 Tree Traversal Operators

Operator	**Name**	**Semantics**
/	child-of	X / Y *if* X is the child of Y.
\	parent-of	X \ Y *if* X is the parent of Y.
/*	dominated-by	X /* Y *if* Y dominates X.
*	dominates	X * Y *if* X dominates Y.
`|`	sibling-of	X `|` Y *if* X and Y share a common parent.
`<|`	left sibling	X `<|` Y *if* X and Y share a common parent and X appears before Y in the parent's list of children.
`>|`	right sibling	X `>|` Y *if* X and Y share a common parent and X appears after Y in the parent's list of children.
`<*`	left-of within tree	X `<*` Y *if* X is dominated by a left child of Y, a left sibling of Y, or a left sibling of any node which dominates Y, or if X dominates Y and all the nodes between X and Y are right children of their parent.
`>*`	right-of within tree	X `>*` Y *if* X is dominated by a right child of Y, a right sibling of Y, or a right sibling of any node which dominates Y, or if X dominates Y and all the nodes between X and Y are left children of their parent.
`cc`	c-command	X cc Y *if* X and Y are siblings or Y is dominated by a sibling of X.

4.7 Return Vectors

A return vector is a comma-separated list of values that are returned by a feature. Each value in the vector must be either a property extracted from a bound variable (`parent.cat`) or a quote-delimited string. Vectors are delimited by `<,>`. Variables bound to sets (using `ALL()`) are expanded, so the vector:

```
<parent.cat,"->",child.cat>
```

would return the array `["S","->","NP","VP"]` if the node bound to `parent` had two children (bound using `ALL()`) with part-of-speech tags "NP" and "VP".

4.8 Summary

Features in FDL have two parts: a required expression (4.3) and an optional return vector (4.7). The expression describes some substructure of the parse tree, and may bind nodes within that structure to variables (4.5). The return vector lists the properties of nodes, bound while matching the expression, that make up the feature. If no return vector is present, the feature returns a boolean value indicating whether the expression could be matched within the parse tree. As an example, consider a feature that extracts the category of the current node's parent:

```
feature parent_cat
    [is=self] / parent:[]
    <parent.cat>
```

The first line declares a feature named `parent_cat` (see section 4.2). The second line is the expression, which in this case consists only of a single path (section 4.3). `[is=self]` is a node descriptor which matches the current node; the forward slash restricts the set of nodes that can be matched by the next descriptor to the parent of the previous node. `parent:[]` binds any node on the current search path to the variable parent (`[]`, the empty node descriptor, matches any node). The final line is the return vector; it is enclosed by angle brackets and consists of a comma-delimited list of elements, either quote-delimited strings or (in this case) `node.property` pairs (section 4.7).

If the return vector was omitted this would be a binary feature indicating whether or not the current node had a parent, which illustrates an important point: unless explicitly specified, bound variables are existential. That is, if there are multiple nodes on the search path that match a node descriptor, only one of them will be bound. In fact, this will be the first node found by a leftmost, depth-first traversal of the parse tree, starting at the root.

5 Path Example

As an in-depth example, consider a feature that represents the path from the current node to the target. Finding a path is particularly difficult because it requires the capture of an indeterminate number of nodes along the tree in a way that maintains the order in which they appear. For convenience, path is defined by first defining a node that represents the lowest common ancestor (LCA) between the current and target nodes. LCA is defined (where N is the set of all nodes) as:

$$\exists lca \in N \wedge (lca\backslash * SELF \wedge lca\backslash * TARGET \wedge (\exists ! x \in N \wedge (x\backslash * SELF \wedge x\backslash * TARGET \wedge lca\backslash * x)))$$

Figure 5: Lowest Common Ancestor

```
node LCA
    [is=self] /* x:[] and x \* [is=target] and
    not (x \ y:[] and y \* [is=self] and y \* [is=target]0
    <x>
```

This collects a node `x` that dominates `self` and also dominates `target`, and that does not have any children that also dominate `self` and `target`. Note the similarity between the FDL expression and the predicate logic, above: FDL can be viewed as nothing more than a logic which operates on trees. In this way, FDL is somewhat similar to logic languages like Prolog.

Using the LCA, the path from `self` to `target` can be easily extracted as the concatenation of the partial paths between `self` up to the LCA and the LCA down to `target`. For convenience, both of these are defined in the same order; as long as the order is consistent across definitions this difference is inconsequential. Also, note that the LCA is included in this path three times: once at in the `up` variable, once as `top`, and once in `down`. This is because the traversal operator (`/*`) behaves like a Kleene star; it captures all nodes that are dominated by zero or more levels, and thus includes the node on each side of the expression. Redundantly including the LCA does not adversely affect the feature, though, since it is consistently appended in the same position each time it appears.

Figure 6: Path feature.

```
feature Path
    ALL(up,[is=self] /* up:[] and up /* LCA) AND
    ALL(down,[is=target] /* down:[] and down /* LCA) AND
    top:LCA
    <up.cat, "<", top.cat, ">", down.cat>
```

6 Formal Grammar (BNF)

⟨file⟩→⟨definition⟩|⟨definition⟩ ⟨file⟩
⟨definition⟩→⟨feature⟩|⟨node⟩
⟨feature⟩→**feature** ⟨string⟩ ⟨expression⟩ ⟨opt-return-vector⟩
⟨node⟩→**node** ⟨string⟩ ⟨expression⟩ ⟨opt-node-return-vector⟩
⟨expression⟩→⟨subexpression⟩|⟨subexpression⟩ ⟨conj⟩ ⟨expression⟩|**NOT** ⟨expression⟩
⟨subexpression⟩→⟨path⟩|⟨quantifier⟩|**(** ⟨expression⟩ **)**
⟨path⟩→⟨node⟩|⟨node⟩ ⟨traversal⟩ ⟨path⟩
⟨quantifier⟩→**ALL(** ⟨string⟩ **,** ⟨expression⟩ **)**
⟨node⟩→⟨string⟩**:**⟨noded⟩|⟨noded⟩
⟨noded⟩→[⟨opt-node-exp⟩]| ⟨string⟩
⟨node-exp⟩→⟨string⟩ = ⟨rvalue⟩|⟨string⟩ **!=** ⟨rvalue⟩|⟨node-exp⟩ ⟨conj⟩ ⟨node-exp⟩|**(** ⟨node-exp⟩ **)**
⟨rvalue⟩→”⟨string⟩”|⟨string⟩|⟨regular-expression⟩|**nil**|⟨string⟩**.**⟨string⟩
⟨return-vector⟩→<⟨list⟩>
⟨node-return-vector⟩→<⟨node-vector-list⟩>
⟨list⟩→⟨listitem⟩|⟨listitem⟩**,**⟨list⟩
⟨listitem⟩→⟨string⟩**.**⟨string⟩|”⟨string⟩”
⟨node-vector-list⟩→⟨string⟩|⟨string⟩**,**⟨node-vector-list⟩
⟨conj⟩→**AND**|**OR**

7 Evaluation

While developing FDL I used the feature set from Gildea and Jurafsky [2002] as my development set; the initial specification for the language was, I believe, the smallest set of rules that could adequately describe that feature set. I then used the feature set from Pradhan et al. [2004] as my initial test set; with no modifications (from the version of FDL used to cover Gildea and Jurafsky [2002]), I was able to describe 18 of the 23 features (out of 25 total; two required additional information not found in the parse tree). By adding additional tree traversal operators to capture sibling information, I was able to describe an additional four. The final remaining feature, constituent tree distance, required only the addition of cardinality as a property of variables and the introduction of basic arithmetic operators in the return vector.

Ultimately, I was able to capture 23 of the 25 features employed in Pradhan et al. [2004]; of the two features which could not be described (dynamic class context and ordinal constituent position), both required extra information not found in the parse tree which would be difficult to introduce in a preprocessing step. Three of the describable features (named entities, verb clusters, and verb senses) required extra preprocessing to add additional information to the parse tree.

To evaluate the expressivity of the language, I examined the features used from 10 papers presented at CoNLL 2005. Of the features used in these papers I identified 56 unique features beyond those used in Gildea and Jurafsky [2002] and Pradhan et al. [2004]. 77% of these features were immediately expressible in FDL; 86% were expressible with additional pre-processing (i.e. after adding named entity information to the parse tree). Those that remain fall into one of two categories: either they are based on dynamic information (i.e. n most recent role assignments) or they require additional post-processing (i.e. converting the path into a set of trigrams).

A Example features from Gildea and Jurafsky [2002].

Figure 7: Phrase type feature.

```
feature Phrase_Type
    x:[is=self]
    <x.cat>
```

Figure 8: Governing category feature.

```
feature Governing_Category
    [is=self and cat="NP"] /* gov:[cat=/s|vp/] AND
    gov \ gov_below:[] AND
    gov_below \* [is=self] AND
    NOT (ALL(also,gov_below \* also:[cat=/s|vp/]) AND
         also \* [is=self])
    <gov.cat>
```

Figure 9: Path feature.

```
node LCA
    [is=self] /* x:[] AND x \* [is=target] AND
    not (x \ y:[] AND y \* [is=self] AND y \* [is=target])
    <x>

feature Path
    ALL(up,[is=self] /* up:[] and up /* LCA) AND
    ALL(down,[is=target] /* down:[] and down /* LCA) AND
    top:LCA
    <up.cat, "<", top.cat, ">", down.cat>
```

Figure 10: Position feature.

```
feature Position
    [is=self] < [is=target]
```

Figure 11: Voice feature.

```
feature ActiveVoice
    [is=target and cat=/v*/] AND
    ([is=target and cat=/(vbg|vbp|vbz|vb)/] OR
        ([is=target] / [] / [cat=/(s|v*)/] AND
        NOT ([is=target] /* x:[] AND
             x \* [lemma=/(be|am|is|are|was|were)/]))

feature PassiveVoice
    [is=target and cat=/v*/] AND
    NOT [is=target and cat=/(vbg|vbp|vbz|vb)/] AND
    [is=target] / [] / gp:[] AND
        (([is=gp and cat=/(s|v*)/] AND
            ([is=target] /* x:[] AND
                 x \* [lemma=/(be|am|is|are|was|were)/])) OR
        [is=target and cat=/(vbn|vbd)/])
```

Figure 12: Head word feature.

```
feature Head_word
    x:[is=self]
    <x.head>
```

B Example Features From Pradhan et al. [2004]

Figure 13: Named entities (with preprocessing).

```
feature named_entity
    x:[is=self]
    <x.entity_type>
```

Figure 14: Partial path.

```
node LCA
    [is=self] /* x:[] AND x \* [is=target] and
    not (x \ y:[] and y \* [is=self] and y \* [is=target]
    <x>

feature partial_path
    ALL(p, [is=self] /* p:[] AND p /* LCA) AND
    top:LCA
    <p.cat,top.cat>
```

Figure 15: Head word of prepositional phrases.

```
feature noun_head
    [is=self and cat='PP'] \ np:[cat='NP']
    <np.head>
```

Figure 16: First word in constituent.

```
node LEAVES
    ALL(x,[is=self] \* x:[word!=nil])
    <x>

feature first_word
    ALL(x,x /* LEAVES AND x <* LEAVES)
    <x.word>
```

References

Jason Baldridge, Sudipta Chatterjee, Alexis Palmer, and Ben Wing. DotCCG and VisCCG: Wiki and programming paradigms for improved grammar engineering with OpenCCG. In Tracy Holloway King and Emily M. Bender, editors, *Proceedings of the GEAF07 Workshop*, pages 5–25, Stanford, CA, 2007. CSLI. URL `http://csli-publications.stanford.edu/GEAF/2007/geaf07-toc.html`.

Figure 17: Parent phrase type.

```
feature parent_cat
    [is=self] / parent:[]
    <parent.cat>
```

Figure 18: Right sibling phrase type.

```
feature right_sibling_cat
    [is=self] >| x:[]
    <x.cat>
```

Figure 19: Head word category.

```
feature head_word_cat
    self:[is=self] AND LEAVES \* head:[word=self.head]
    <head.cat>
```

Figure 20: Constituent tree distance.

```
node PATH_NODES
    ALL(p, [is=self] /* p:[] and p /* LCA) AND
    ALL(d, [is=target] /* d:[] and d/* LCA)
    <p,d>

feature tree_distance
    p:PATH_NODES
    <p.size>
```

Figure 21: Temporal cue words.

```
feature cue_words
    [word="cue_word_1" or word="cue_word_2"]
```

Xavier Carreras and Lluís Màrquez. Introduction to the conll-2005 shared task: Semantic role labeling. *Proceedings of CoNLL-2005*, 2005.

James Clark and Steve DeRose, editors. *XML Path Language (XPath), Version 1.0*, November 1999. W3C. URL `http://www.w3.org/TR/xpath`.

Michael Collins. *Head-Driven Statistical Models for Natural Language Parsing*. PhD thesis, University of Pennsylvania, 1999.

Katrin Erk and Sebastian Pado. Shalmaneser - a flexible toolbox for semantic role assignment. In *Proceedings of LREC 2006*, Genoa, Italy, 2006.

D. Gildea and D. Jurafsky. Automatic labeling of semantic roles. *Computational Linguistics*, 28(3):245–288, 2002.

Esther König and Wolfgang Lezius. A description language for syntactically annotated corpora. In *Proceedings of the COLING Conference*, pages 1056–1060, Saarbrücken, Germany, 2000. URL `http://www.ims.uni-stuttgart.de/projekte/corplex/paper/lezius/coling2000.pdf`.

Esther König and Wolfgang Lezius. The tiger language - a description language for syntax graphs, formal definition. Technical report ims, Universität Stuttgart, Germany, 2003.

Roger Levy and Galen Andrew. Tregex and tsurgeon: tools for querying and manipulating tree data structures. *5th International Conference on Language Resources and Evaluation*, 2006.

Sameer Pradhan, Wayne Ward, Kadri Hacioglu, James Martin, and Dan Jurafsky. Shallow semantic parsing using support vector machines. In *Proceedings of the Human Language Technology Conference/North American chapter of the Association of Computational Linguistics (HLT/NAACL)*, Boston, MA, 2004.

Sameer Pradhan, Kadri Hacioglu, Wayne Ward, James H. Martin, and Daniel Jurafsky. Semantic role chunking combining complementary syntactic views. In *Proceedings of the Ninth Conference on Computational Natural Language Learning (CoNLL-2005)*, pages 217–220, Ann Arbor, Michigan, June 2005. Association for Computational Linguistics.