An Extensible Toolkit for Computational Semantics

Dan Garrette   Ewan Klein

May 16, 2009

1 Introduction

In this paper we focus on the software for computational semantics provided by the Python-based Natural Language Toolkit (NLTK). The semantics modules in NLTK are inspired in large part by the approach developed in Blackburn and Bos (2005) (henceforth referred to as B&B). Since Blackburn and Bos have also provided a software suite to accompany their excellent textbook, one might ask what the justification is for the NLTK offering, which is similarly slanted towards teaching computational semantics.

This question can be answered in a number of ways. First, we believe there is intrinsic merit in the availability of different software tools for semantic analysis, even when there is some duplication of coverage; and this will become more true as computational semantics starts to be as widely studied as computational syntax. For example, one rarely hears the objection that too many implementations of syntactic parsers are available. Moreover, the NLTK software significantly goes beyond B&B in providing an implementation of Glue Semantics.

Second, whatever the relative merits of Prolog vs. Python as programming languages, there is surely an advantage in offering students and instructors a choice in this respect. Given that many students have either already been exposed to Java, or else have had no programming experience at all, Python offers them the option of accomplishing interesting results with only a shallow learning curve.

Third, NLTK is a rapidly developing, open source project[1] with a broad coverage of natural language processing (NLP) tools; see Bird et al. (2008) for a recent overview. This wide functionality has a number of benefits, most notably that lexical, syntactic and semantic processing can be carried out within a uniform computational framework. As a result, NLTK makes it much easier to include some computational semantics subject matter in a broad course on natural language analysis, rather than having to devote a whole course exclusively to the topic.

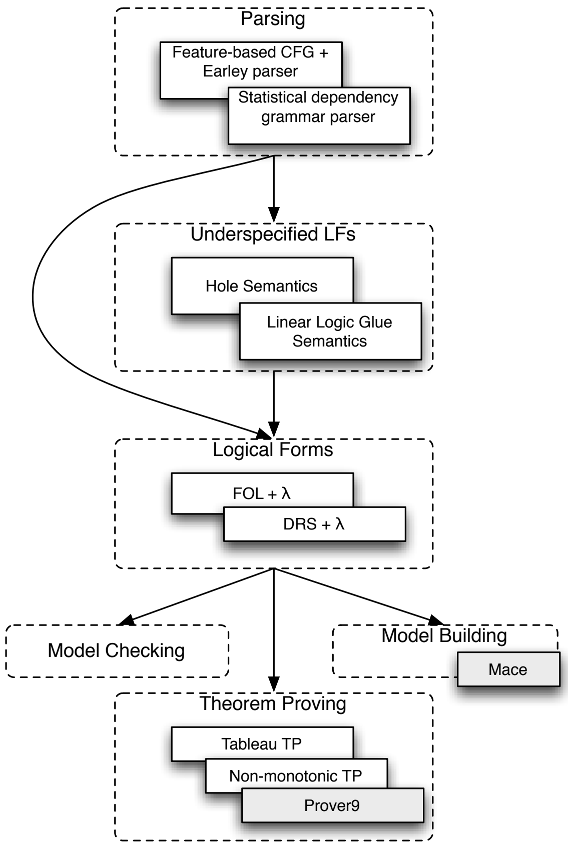
[1]See http://www.nltk.org

Fourth, NLTK is accompanied by a substantial collection of corpora, plus easy-to-use corpus readers. This collection, which currently stands at over 50 corpora and trained models, includes parsed, POS-tagged, plain text, categorized text, and lexicons. The availability of corpora can help encourage students to go beyond writing toy grammars, and instead to start grappling with the complexities of semantically analysing realistic bodies of text.

Fifth, NLTK is not just for students. Although Python is slower than languages like Java and C++, its suitability for rapid prototyping makes it an attractive addition to the researcher's inventory of resources. Building an experimental set-up in NLTK to test a hypothesis or explore some data is straightforward and quick, and the rich variety of existing NLP components in the toolkit allows rapid assembly of quite sophisticated processing pipelines.

2 Overview

Like B&B, we assume that one of the most important tasks for the teacher is to ground students in the basic concepts of first order logic and the lambda calculus, model-theoretic interpretation and inference. This provides a basis for exploring more modern approaches like Discourse Representation Theory (DRT; Kamp and Reyle (1993)) and underspecification.



In the accompanying figure, we give a diagrammatic overview of the main semantics-related functionality that is currently available in NLTK. Logical forms (LFs) can be induced as result of syntactic parsing, using either feature-based grammars that are processed with an Earley chart parser, or else by associating LFs with the output of a broad-coverage dependency parser. Our basic LFs are expressions of first order logic, supplemented with the lambda operator. However, we also admit Discourse Representation Structures (DRSs) as LFs, and underspecified LFs can be built using either Hole Semantics (Blackburn and Bos, 2005) or Glue Semantics (Dalrymple et al., 1999). Once we have constructed LFs, they can be evaluated in a first order model (Klein, 2006), tested for equivalence and validity in a variety of theorem provers, or tested for consistency in a model builder. The latter two tasks are aided by NLTK interfaces to third-party inference tools, currently Prover9 and Mace4 (McCune, 2008).

We do not have space in this paper to discuss all of these components, but will try to present some of the key aspects, and along the way noting certain points of difference *vis-à-vis* B&B.

3 Logical Form

3.1 First Order Predicate Logic with Lambda Calculus

From a pedagogical point of view, it is usually important to ensure that students have some grasp of the language of first order predicate logic (FOL), and can also manipulate λ-abstraction. The `nltk.sem.logic` module contains an object-oriented approach to representing FOL plus λ-abstraction. Logical formulas are typically fed to the `logic` parser as strings, and then represented as instances of various subclasses of `Expression`, as we will see shortly.

An attractive feature of Python is its interactive interpreter, which allows the user to enter Python expressions and statements for evaluation. In the example below and subsequently, `>>>` is the Python interpreter's prompt.

```
1  >>> from nltk.sem import LogicParser
2  >>> lp = LogicParser()
3  >>> e = lp.parse('all x.(girl(x) -> exists y.(dog(y) & chase(x,y)))')
4  >>> e
5  <AllExpression all x.(girl(x) -> exists y.(dog(y) & chase(x,y)))>
```

As illustrated, the result of parsing the formula at line 3 is an object `e` belonging to the class `AllExpression`, itself a subclass of `Expression`. All such subclasses have numerous methods that implement standard logical operations. For example, the `simplify()` method carries out β-conversion; the `free()` method finds all the free variables in an expression; and for quantified expressions (such as `AllExpressions`), there is an `alpha_convert()` method. The `logic` module will α-convert automatically when appropriate to avoid name-clashes in the `replace()` method. Let's illustrate these methods with a formula involving λ-abstraction, namely `\x.P(x)(y)`; we use `\` to represent λ. (Since `\` is a special character in Python, we add the `r` prefix to strings containing it to preclude additional escape characters.)

```
>>> from nltk.sem import Variable
>>> e1 = lp.parse(r'\x.P(x)(y)')
>>> print e1.simplify()
P(y)
>>> e2 = lp.parse('all x.P(x,a,b)')
>>> print e2.free()
set([<Variable('a'), Variable('b')])
>>> print e2.alpha_convert(Variable('z'))
all z.P(z,a,b)
>>> e3 = lp.parse('x')
>>> print e2.replace(Variable('b'), e3)
all z1.P(z1,a,x)
```

Allowing students to build simple first order models, and evaluate expressions in those models, can be useful for helping them clarify their intuitions about quantification. In the next example, we show one of the available methods in NLTK for specifying a model and using it to determine the set of satisfiers of the open formula $\exists x.(girl(y) \wedge chase(x, y))$.[2], [3]

```
>>> from nltk.sem import parse_valuation, Model, Assignment
>>> v = """
... suzie => s
... fido => f
... rover => r
... girl => {s}
... chase => {(f, s), (r, s), (s, f)}
... """
>>> val = parse_valuation(v)   #create a Valuation
>>> m = Model(val.domain, val) #initialize a Model
>>> g = Assignment(val.domain) #initialize an Assignment
>>> e4 = lp.parse('exists y. (girl(y) & chase(x, y))')
>>> m.satisfiers(e4, 'x', g)   #check satisfiers of e4 wrt to x
set(['r', 'f'])
```

In B&B, λ-abstracts are second-class citizens, used exclusively as a 'glue' mechanism for composing meaning representations. Although we use λ-abstracts as glue too, abstracts over individual variables are semantically interpreted in NLTK, namely as characteristic functions.

`Expression`s in NLTK are typed (using Montague-style types). The type can be accessed with the `type` property. A type checking procedure can be invoked with the `typecheck()` method. `typecheck()` will return a dictionary of all variables and their types if the expression is well typed; for

[2]The triple quotes `"""` in Python allow us to break a logical line across several physical lines.

[3]Given a valuation `val`, the property `val.domain` returns the set of all domain individuals specified in the valuation.

non-well typed expressions, an exception will be raised. Additionally, type checking will be automatically invoked by the `LogicParser` if the parameter `type_check=True` is set.

```
>>> a = lp.parse(r'\x.(man(x) & tall(x))')
>>> a.type
<e,t>
>>> a.typecheck()
{'x': e, 'man': <e,t>, 'tall': <e,t>}
>>> tlp = LogicParser(type_check=True)
>>> tlp.parse(r'\x y.-see(x,y)(\x.(man(x) & tall(x)))')
Traceback (most recent call last):
  . . .
TypeException: The function '\x y.-see(x,y)' is of type '<e,<e,t>>'
and cannot be applied to '\x.(man(x) & tall(x))' of type '<e,t>'.
Its argument must match type 'e'.
```

3.2 Discourse Representation Theory

As mentioned earlier, NLTK contains an extension to the `logic` module for working with Discourse Representation Theory (DRT) (Kamp and Reyle, 1993). The `nltk.sem.drt` module introduces a classes for working with Discourse Representation Structures (DRSs). In NLTK, a DRS is represented as a pair consisting of a list of discourse of referents and a list of DRS conditions:

(1) `([j,d],[John(j), dog(d), sees(j,d)])`

On top of the functionality available for FOL expressions, DRT expressions have a 'DRS-concatenation' operator, represented as the `+` symbol. The concatenation of two DRSs is a single DRS containing the merged discourse referents and the conditions from both arguments. DRS-concatenation automatically α-converts bound variables to avoid name-clashes. The `+` symbol is overloaded so that DRT expressions can be added together easily. The `DrtParser` allows DRSs to be specified succinctly as strings.

```
>>> from nltk.sem.drt import DrtParser
>>> dp = DrtParser()
>>> d1 = dp.parse('([x],[walk(x)]) + ([y],[run(y)])')
>>> print d1
(([x],[walk(x)]) + ([y],[run(y)]))
>>> print d1.simplify()
([x,y],[walk(x), run(y)])
>>> d2 = dp.parse('([x,y],[Bill(x), Fred(y)])')
>>> d3 = dp.parse("""([],[([u],[Porsche(u), own(x,u)])
...  ->  ([v],[Ferrari(v), own(y,u)])])""")
```

```
>>> d4 = d2 + d3
>>> print d4.simplify()
([x,y],[Bill(x), Fred(y),
(([u],[Porsche(u), own(x,u)]) -> ([v],[Ferrari(v), own(y,u)]))])
```

DRT expressions can be converted to their first order predicate logic equivalents using the `toFol()` method and can be graphically rendered on screen with the `draw()` method.

```
>>> print d1.toFol()
(exists x.walk(x) & exists y.run(y))
>>> d4.simplify().draw()
```

Since the λ operator can be combined with DRT expressions, the `nltk.sem.drt` module can be used as a plug-in replacement for `nltk.sem.logic` in building compositional semantics.

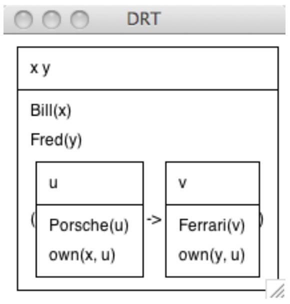


Figure 1: DRS Screenshot

4 Scope Ambiguity and Underspecification

Two key questions in introducing students to computational semantics are:

Q1: How are semantic representations constructed from input sentences?
Q2: What is scope ambiguity and how is it captured?

A standard pedagogical approach is to address (Q1) with a simple syntax-driven induction of logical forms which fails to deal with scope ambiguity, while (Q2) is addressed by introducing underspecified representations which are resolved to produce different readings of ambiguous sentences.

NLTK includes a suite of parsing tools, amongst which is a chart parser for context free grammars augmented with feature structures. A 'semantics' feature `sem` allows us to compose the contributions of constituents to build a logical form for a complete sentence. To illustrate, the following minimal grammar `sem1.fcfg` handles quantification and intransitive verbs (where values such as `?subj` and `?vp` are unification variables, while `P` and `Q` are λ-bound object language variables):

```
S[sem = <?subj(?vp)>] -> NP[sem=?subj] VP[sem=?vp]
VP[sem=?v] -> IV[sem=?v]
NP[sem=<?det(?n)>] -> Det[sem=?det] N[sem=?n]
Det[sem=<\P.\Q.exists x.(P(x) & Q(x))>] -> 'a'
N[sem=<\x.dog(x)>] -> 'dog'
IV[sem=<\x.bark(x)>] -> 'barks'
```

Using `sem1.fcfg`, we can parse *A dog barks* and view its semantics. The `load_earley()` method takes an optional parameter `logic_parser` which specifies the logic-parser for processing the value of the `sem` feature, thus allowing different kinds of logical forms to be constructed.

```
>>> from nltk.parse import load_earley
>>> parser = load_earley('grammars/sem1.fcfg', trace=0)
>>> trees = parser.nbest_parse('a dog barks'.split())
>>> print trees[0].node['sem'].simplify()
exists x.(dog(x) & bark(x))
```

Underspecified logical forms allow us to loosen the relation between syntactic and semantic representations. We consider two approaches to underspecification, namely Hole Semantics and Glue Semantics. Since the former will be familiar from B&B, we devote most of our attention to presenting Glue Semantics.

4.1 Hole Semantics

Hole Semantics in NLTK is handled by the `nltk.sem.hole` module, which uses a context free grammar to generate an underspecified logical form. Since the latter is itself a formula of first order logic, we can continue to use the `sem` feature in the context free grammar:

```
N[sem=<\x h l.(PRED(l,dog,x) & LEQ(l,h) & HOLE(h) & LABEL(l))>] -> 'dog'
```

The Hole Semantics module uses a standard plugging algorithm to derive the sentence's readings from the underspecified LF.

```
>>> from nltk.sem.hole import hole_readings
>>> readings = hole_readings('every girl chases a dog')
>>> for r in readings: print r
exists z1.(dog(z1) & all z2.(girl(z2) -> chase(z1,z2)))
all z2.(girl(z2) -> exists z1.(dog(z1) & chase(z1,z2)))
```

4.2 Glue Semantics

Glue Semantics (Dalrymple et al., 1999), or Glue for short, is an approach to compositionality that tries to handle semantic ambiguity by using resource-sensitive logic to assemble meaning expressions. The approach builds proofs over 'meaning constructors'; these are of the form $\mathcal{M} : \mathcal{G}$, where $\mathcal{M}$ is a meaning representation and $\mathcal{G}$ is a term of linear logic. The linear logic term $\mathcal{G}$ dictates how the meaning expression $\mathcal{M}$ can be combined. Each distinct proof that can be derived reflects a different semantic reading of the entire sentence.

The variant of linear logic that we use has *(linear) implication* (i.e., $\multimap$) as its only operator, so the primary operation during the proof is Modus Ponens. Linear logic is an appropriate logic to serve as 'glue' because it is resource-sensitive. This means that when Modus Ponens combines two terms to create a new one, the two original terms are 'consumed', and cannot be used again in the proof; cf. (2) vs. (3). Additionally, every premise must be used for the proof to be valid; cf. (4). This resource-sensitivity dictates that each word contributes its meaning exactly once to the meaning of the whole.

(2) $A, (A \multimap B) \vdash B$

(3) $A, (A \multimap B) \nvdash A, B$

(4) $A, A, (A \multimap B) \nvdash B$

NLTK's `nltk.gluesemantics.linearlogic` module contains an implementation of linear logic.

The primary rule for composing Glue formulas is (5). Function-argument application of meaning expressions is reflected (*via* the Curry-Howard isomorphism) by the application of Modus Ponens in a linear logic proof. Note that A and B are meta-variables over constants of linear logic; these constants represent 'attachment points' for meaning expressions in some kind of syntactically-derived representation (such as an LFG f-structure). It is (5) which allows Glue to guide the construction of complex meaning expressions.

(5) $\alpha : A,\ \gamma : (A \multimap B) \vdash \gamma(\alpha) : B$

The NLTK modules `gluesemantics.glue` and `gluesemantics.drt_glue` implement Glue for FOL and DRT meaning expressions, respectively.[4] The following example shows how Glue formulas are created and combined to derive a logical form for *John walks*:

```
>>> from nltk.gluesemantics.glue import GlueFormula
>>> john = GlueFormula('john', 'g')
>>> walks = GlueFormula(r'\x.walk(x)', '(g -o f)')
>>> john_walks = walks.applyto(john)
>>> print john_walks.meaning.simplify()
walk(john)
```

Thus, the non-logical constant *john* is associated with the Glue term g, while the meaning expression $\lambda x.walk(x)$ is associated with $(g \multimap f)$ since it is a function that takes g as input and returns the meaning expression f,

[4]See http://nltk.googlecode.com/svn/trunk/doc/contrib/sem/index.html for more details.

corresponding to the whole sentence. Consequently, a proof of f from the premises is a derivation of a meaning representation for the sentence.

Scope ambiguity, resulting, for example, from quantifiers, requires the use of *variables* in the Glue terms. Such variables may be instantiated to any linear logic constant, so long as this is carried out uniformly. Let's assume that the quantified noun phrase *every girl* has the meaning constructor (6) (where G is a linear logic variable):

(6) $\lambda Q.\forall x.(girl(x) \rightarrow Q(x)) : ((g \multimap G) \multimap G)$

Then the Glue derivation shown below correctly generates two readings for the sentence *Every girl chases a dog*:

```
>>> from nltk.gluesemantics.glue import GlueFormula, Glue
>>> a = GlueFormula(r'\Q.all x.(girl(x) -> Q(x))', '((g -o G) -o G)')
>>> b = GlueFormula(r'\x y.chase(x,y)', '(g -o (h -o f))')
>>> c = GlueFormula(r'\Q.exists x.(dog(x)&Q(x))', '((h -o H) -o H)')
>>> glue = Glue()
>>> for reading in glue.get_readings(glue.gfl_to_compiled([a,b,c])):
...     print reading.simplify()
exists x.(dog(x) & all z13.(girl(z13) -> chase(z13,x)))
all x.(girl(x) -> exists z14.(dog(z14) & chase(x,z14)))
```

5 Inference tools

In order to perform inference over semantic representations, NLTK can call both theorem provers and model builders. The library includes a pure Python tableau-based first order theorem prover; this is intended to allow students to study tableau methods for theorem proving, and provides an opportunity for experimentation. In addition, NLTK provides interfaces to two off-the-shelf tools, namely the theorem prover Prover9, and the model builder Mace4 (McCune, 2008).

The `get_prover(G, A)` method by default calls Prover9, and takes as parameters a proof goal `G` and a list `A` of assumptions. Here, we verify that if every dog barks, and Rover is a dog, then it is true that Rover barks:

```
>>> from nltk.inference import Prover9
>>> a = lp.parse('all x.(dog(x) -> bark(x))')
>>> b = lp.parse('dog(rover)')
>>> c = lp.parse('bark(rover)')
>>> prover = Prover9()
>>> prover.prove(c, [a,b])
True
```

A theorem prover can also be used to check the logical equivalence of expressions. For two expressions A and B, we can pass $(A \iff B)$ into a theorem prover and know that the theorem will be proved if and only if the expressions are logically equivalent. NLTK's standard equality operator for `Expressions` (`==`) is able to handle situations where two expressions are identical up to α-conversion. However, it would be impractical for NLTK to invoke a wider range of logic rules every time we checked for equality of two expressions. Consequently, both the `logic` and `drt` modules in NLTK have a separate method, `tp_equals`, for checking 'equality' up to logical equivalence.

```
>>> a = lp.parse('all x.walk(x)')
>>> b = lp.parse('all y.walk(y)')
>>> a == b
True
>>> c = lp.parse('-(P(x) & Q(x))')
>>> d = lp.parse('-P(x) | -Q(x)')
>>> c == d
False
>>> c.tp_equals(d)
True
```

6 Discourse Processing

NLTK contains a discourse processing module, `nltk.inference.discourse`, similar to the CURT program presented in B&B. This module processes sentences incrementally, keeping track of all possible threads when there is ambiguity. For simplicity, the following example ignores scope ambiguity.

```
>>> from nltk.inference.discourse import DiscourseTester as DT
>>> dt = DT(['A student dances', 'Every student is a person'])
>>> dt.readings()
s0 readings:
s0-r0: exists x.(student(x) & dance(x))
s1 readings:
s1-r0: all x.(student(x) -> person(x))
```

When a new sentence is added to the current discourse, setting the parameter `consistchk=True` causes consistency to be checked by invoking the model checker for each 'thread', i.e., discourse sequence of admissible readings. In this case, the user has the option of retracting the sentence in question.

```
>>> dt.add_sentence('No person dances', consistchk=True)
Inconsistent discourse d0 ['s0-r0', 's1-r0', 's2-r0']:
    s0-r0: exists x.(student(x) & dance(x))
    s1-r0: all x.(student(x) -> person(x))
    s2-r0: -exists x.(person(x) & dance(x))
>>> dt.retract_sentence('No person dances', quiet=False)
Current sentences are
s0: A student dances
s1: Every student is a person
```

In a similar manner, we use `informchk=True` to check whether the new sentence is informative relative to the current discourse (by asking the theorem prover to derive it from the discourse).

```
>>> dt.add_sentence('A person dances', informchk=True)
Sentence 'A person dances' under reading 'exists x.(person(x) & dance(x))':
Not informative relative to thread 'd0'
```

It is also possible to pass in an additional set of assumptions as background knowledge and use these to filter out inconsistent readings.

The `discourse` module can accommodate semantic ambiguity and filter out readings that are not admissable. By invoking both Glue Semantics and DRT, the following example processes the two-sentence discourse *Every dog chases a boy. He runs.* As shown, the first sentence has two possible readings, while the second sentence contains an anaphoric pronoun, indicated as `PRO(x)`.

```
>>> from nltk.inference.discourse import DrtGlueReadingCommand as RC
>>> dt = DT(['Every dog chases a boy', 'He runs'], RC())
>>> dt.readings()
s0 readings:
s0-r0: ([],[(([x],[dog(x)]) -> ([z15],[boy(z15), chase(x,z15)]))])
s0-r1: ([z16],[boy(z16), (([x],[dog(x)]) -> ([],[chase(x,z16)]))])
s1 readings:
s1-r0: ([x],[PRO(x), run(x)])
```

When we examine the two threads `d0` and `d1`, we see that that reading `s0-r0`, where *every dog* out-scopes `a boy`, is deemed inadmissable because the pronoun in the second sentence cannot be resolved. By contrast, in thread `d1` the pronoun (relettered to `z24`) has been bound *via* the equation `(z24 = z20)`.

```
>>> dt.readings(show_thread_readings=True)
d0: ['s0-r0', 's1-r0'] : INVALID: AnaphoraResolutionException
d1: ['s0-r1', 's1-r0'] : ([z20,z24],[boy(z20), (([x],[dog(x)]) ->
([],[chase(x,z20)])), (z24 = z20), run(z24)])
```

7 Conclusions and Future Work

NLTK's semantics functionality has been written with extensibility in mind. The `logic` module's `LogicParser` employs a basic parsing template and contains hooks that an extending module can use to supplement or substitute functionality. Moreover, the base `Expression` class in `logic`, as well as any derived classes, can be extended, allowing variants to reuse the existing functionality. For example, the DRT and linear logic modules are implemented as extensions to `logic.py`.

The theorem prover and model builder code has also been carefully architected to allow extensions and the `nltk.inference.api` library exposes the framework for the inference architecture. The library therefore provides a good starting point for creating interfaces with other theorem provers and model builders in addition to Prover9, Mace4, and the tableau prover.

NLTK already includes the beginnings of a framework for 'recognizing textual entailment'; access to the RTE data sets is provided and we are in the course of developing a few simple modules to demonstrate RTE techniques. For example, a Logical Entailment RTE tagger based on Bos and Markert (2005) begins by building a semantic representation of both the text and the hypothesis in DRT. It then runs a theorem prover with the text as the assumption and the hypothesis as the goal in order to check whether the text entails the hypothesis.The tagger is also capable of adding background knowledge *via* an interface to the WordNet dictionary in `nltk.wordnet` as a first step in making the entailment checking more robust.

References

Steven Bird, Ewan Klein, Edward Loper, and Jason Baldridge. Multidisciplinary instruction with the Natural Language Toolkit. In *Proceedings of the Third Workshop on Issues in Teaching Computational Linguistics*, Columbus, Ohio, USA, June 2008.

Patrick Blackburn and Johan Bos. *Representation and Inference for Natural Language: A First Course in Computational Semantics.* CSLI Publications, New York, 2005.

Johan Bos and Katja Markert. Recognising textual entailment with logical inference. In *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*, Vancouver, British Columbia, Canada, 2005.

Mary Dalrymple, V. Gupta, John Lamping, and V. Saraswat. Relating resource-based semantics to categorial semantics. In Mary Dalrymple, editor, *Semantics and syntax in Lexical Functional Grammar: the resource logic approach*, pages 261–280. MIT Press, Cambridge, MA, 1999.

Hans Kamp and Uwe Reyle. *From Discourse to the Lexicon: Introduction to Modeltheoretic Semantics of Natural Language, Formal Logic and Discourse Representation Theory*. Kluwer Academic Publishers, 1993.

Ewan Klein. Computational semantics in the Natural Language Toolkit. In *Proceedings of the Australasian Language Technology Workshop*, pages 26–33, 2006.

William McCune. Prover9: Automated theorem prover for first-order and equational logic, 2008. http://www.cs.unm.edu/~mccune/mace4/manual-examples.html.