Chapter 11

Logical Semantics

11.1 Introduction

There are many NLP applications where it would be useful to have some representation of the *meaning* of a natural language sentence. For instance, current search engine technology can only take us so far in giving concise and correct answers to many questions that we might be interested in. Admittedly, Google does a good job in answering (1a), since its first hit is (1b).

(1) a. What is the population of Saudi Arabia?

b. Saudi Arabia - Population: 26,417,599

By contrast, the result of sending (2) to Google is less helpful:

(2) Which countries border the Mediterranean?

This time, the topmost hit (and the only relevant one in the top ten) presents the relevant information as a map of the Mediterranean basin. Since the map is an image file, it is not easy to extract the required list of countries from the returned page.

Even if Google succeeds in finding documents which contain information relevant to our question, there is no guarantee that it will be in a form which can be easily converted into an appropriate answer. One reason for this is that the information may have to be inferred from more than one source. This is likely to be the case when we seek an answer to more complex questions like (3):

(3) Which Asian countries border the Mediterranean?

Here, we would probably need to combine the results of two subqueries, namely (2) and *Which countries are in Asia?*.

The example queries we have just given are based on a paper dating back to 1982 [Warren & Pereira, 1982]; this describes a system, *Chat-80*, which converts natural language questions into a semantic representation, and uses the latter to retrieve answers from a knowledge base. A knowledge base is usually taken to be a set of sentences in some formal language; in the case of Chat-80, it is a set of Prolog clauses. However, we can encode knowledge in a variety of formats, including relational databases, various kinds of graph, and first-order models. In NLTK, we have used the third of these options to re-implement a limited version of Chat-80:

```
Sentence: which Asian countries border the_Mediterranean
------------------------------
\x.((contain(asia, x) & country (x)) & border (x, mediterranean)
set(['turkey', 'syria', 'israel', 'lebanon'])
```

As we will explain later in this chapter, a semantic representation of the form `\x.P(x)` denotes a set of entities *u* that meet some condition `P(x)`. We then ask our knowledge base to enumerate all the entities in this set.

Let's assume more generally that knowledge is available in some structured fashion, and that it can be interrogated by a suitable query language. Then the challenge for NLP is to find a method for converting natural language questions into the target query language. An alternative paradigm for question answering is to take something like the pages returned by a Google query as our 'knowledge base' and then to carry out further analysis and processing of the textual information contained in the returned pages to see whether it does in fact provide an answer to the question. In either case, it is very useful to be able to build a semantic representation of questions. This NLP challenge intersects in interesting ways with one of the key goals of linguistic theory, namely to provide a systematic correspondence between form and meaning.

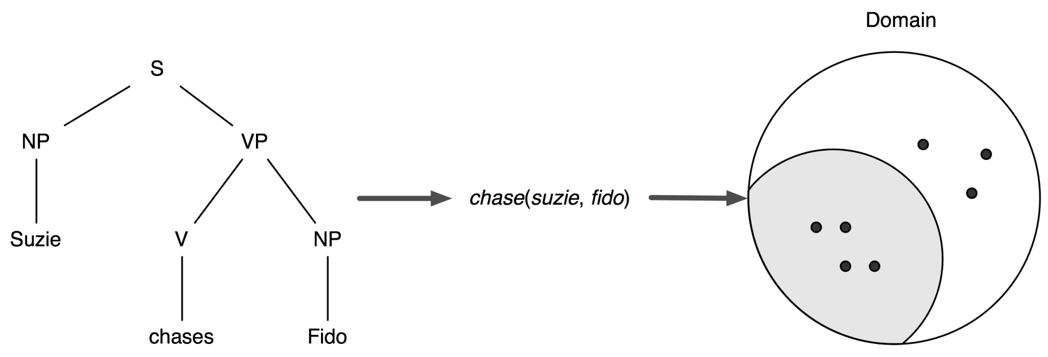
A widely adopted approach to representing meaning — or at least, some aspects of meaning — involves translating expressions of natural language into first-order logic (FOL). From a computational point of view, a strong argument in favor of FOL is that it strikes a reasonable balance between expressiveness and logical tractability. On the one hand, it is flexible enough to represent many aspects of the logical structure of natural language. On the other hand, automated theorem proving for FOL has been well studied, and although inference in FOL is not decidable, in practice many reasoning problems are efficiently solvable using modern theorem provers (cf. [Blackburn & Bos, 2005] for discussion).

While there are numerous subtle and difficult issues about how to translate natural language constructions into FOL, we will largely ignore these. The main focus of our discussion will be on a different issue, namely building semantic representations which conform to some version of the **Principle of Compositionality**. (See [Gleitman & Liberman, 1995] for this formulation.)

Principle of Compositionality: The meaning of a whole is a function of the meanings of the parts and of the way they are syntactically combined.

There is an assumption here that the semantically relevant parts of a complex expression will be determined by a theory of syntax. Within this chapter, we will take it for granted that expressions are parsed against a context-free grammar. However, this is not entailed by the Principle of Compositionality. To summarize, we will be concerned with the task of systematically constructing a semantic representation in a manner that can be smoothly integrated with the process of parsing.

The overall framework we are assuming is illustrated in Figure (4). Given a syntactic analysis of a sentence, we can build one or more semantic representations for the sentence. Once we have a semantic representation, we can also check whether it is true in a model.



(4) syntactic analysis semantic representation truth in a model

A **model** for a logical language is a set-theoretic construction which provides a very simplified picture of how the world is. For example, in this case, the model should contain individuals (indicated in the diagram by small dots) corresponding to Suzie and Fido, and it should also specify that these individuals belong to the *chase* relation.

The order of sections in this chapter is not what you might expect from looking at the diagram. We will start off in the middle of (4) by presenting a logical language that will provide us with semantic representations in NLTK. Next, we will show how formulas in the language can be systematically evaluated in a model. At the end, we will bring everything together and describe a simple method for constructing semantic representations as part of the parse process in NLTK.

11.2 Propositional Logic

The language of propositional logic represents certain aspects of natural language, but at a high level of abstraction. The only structure that is made explicit involves **logical connectives**; these correspond to 'logically interesting' expressions such as *and* and *not*. The basic expressions of the language are **propositional variables**, usually written p, q, r, etc. Let A be a finite set of such variables. There is a disjoint set of logical connectives which contains the unary operator $\neg$ (*not*), and binary operators $\wedge$ (*and*), $\vee$ (*or*), $\rightarrow$ (*implies*) and $\equiv$ (*iff*).

The set of formulas of L_{prop} is described inductively:

1. Every element of A is a formula of L_{prop}.
2. If φ is a formula of L_{prop} , then so is $\neg\varphi$.
3. If φ and ψ are formulas, then so are $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \rightarrow \psi)$ and $(\varphi \equiv \psi)$.
4. Nothing else is a formula of L_{prop}.

Within L_{prop}, we can construct formulas such as

(5) $p \rightarrow q \vee r$

There are many sentences of English which could be taken to have the logical structure shown in (5). Here's an example:

(6) If it is raining, then Kim will take an umbrella or Lee will get wet.

In order to explain the relation between (5) and (6), we need to give a *key* which maps between propositional variables and English sentences:

(7) *p* stands for *it is raining*, *q* for *Kim will take an umbrella* and *q* for *Lee will get wet*.

The Boolean connectives of propositional logic are supported by the `sem` package, and are parsed into various kinds of `Expression`. We use `-`, `&`, `|`, `->`, `<->` to stand, respectively, for *not*, *and*, *or*, *implies* and *iff*. In the following example, we start off by creating a new instance `lp` of the NLTK `LogicParser()`.

```
>>> lp = nltk.LogicParser()
>>> lp.parse('-(p & q)')
<NegatedExpression -(p & q)>
>>> lp.parse('p & q')
<AndExpression (p & q)>
>>> lp.parse('p | (r -> q)')
<OrExpression (p | (r -> q))>
>>> lp.parse('p <-> -- p')
<IffExpression (p <-> --p)>
```

As the name suggests, propositional logic only studies the logical structure of formulas made up of atomic propositions. We saw, for example, that propositional variables stood for whole clauses in English. In order to look at how predicates combine with arguments, we need to look at a more complex language for semantic representation, namely first-order logic. In order to show how this new language interacts with the λ-calculus, it will be useful to introduce the notion of types into our syntactic definition, in departure from the rather simple approach to defining the clauses of L_{prop}.

In the general case, we interpret sentences of a logical language relative to a model, which is a very simplified version of the world. A model for propositional logic needs to assign the values `True` or `False` to every possible formula. We do this inductively: first, every propositional variable is assigned a value, and then we compute the value of complex formulas by consulting the meanings of the Boolean connectives and applying them to the values of the formula's components. Let's create a valuation:

```
>>> val1 = nltk.sem.Valuation([('p', True), ('q', True), ('r', False)])
```

We initialize a `Valuation` with a list of pairs, each of which consists of a semantic symbol and a semantic value. The resulting object is essentially just a dictionary that maps logical expressions (treated as strings) to appropriate values.

```
>>> val1['p']
True
```

The keys of the dictionary (sorted alphabetically) can also be accessed via the property `symbols`:

```
>>> val1.symbols
['p', 'q', 'r']
```

As we will see later, our models need to be somewhat more complicated in order to handle the more complicated expressions discussed in the next section, so for the time being, just ignore the `dom1` and `g1` variables in the following declarations.

```
>>> dom1 = set([])
>>> g = nltk.sem.Assignment(dom1)
```

Now, let's create a model m that uses ``val1:

```
>>> m1 = nltk.sem.Model(dom1, val1, prop=True)
```

The prop=True is just a flag to say that our models are intended for propositional logic.

Every instance of Model defines appropriate truth functions for the Boolean connectives (and in fact they are implemented as functions named AND(), IMPLIES() and so on).

```
>>> m1.AND
<bound method Model.AND of (set([]), {'q': True, 'p': True, 'r': False})>
```

We can use these functions to create truth tables:

```
>>> for first in [True, False]:
...     for second in [True, False]:
...         print "%s %s => %s" % (first, second, m1.AND(first, second))
True True => True
True False => False
False True => False
False False => False
```

11.3 First-Order Logic

11.3.1 Predication

In first-order logic (FOL), propositions are analyzed into predicates and arguments, which takes us a step closer to the structure of natural languages. The standard construction rules for FOL recognize **terms** such as individual variables and individual constants, and **predicates** which take differing numbers of arguments. For example, *Adam walks* might be formalized as walk(adam) and *Adam sees Betty* as see(adam, betty). We will call walk a **unary predicate**, and see a **binary predicate**. Semantically, see is usually modeled as a relation, i.e., a set of pairs, and the proposition is true in a situation just in case the ordered pair pair a, b belongs to this set.

There is an alternative approach in which predication is treated as function application. In this functional style of representation, *Adam sees Betty* can be formalized as see(j)(m). That is, rather than being modeled as a relation, see denotes a function which applies to one argument to yield a new function that is then applied to the second argument. In NLTK, we will in fact treat predications syntactically as function applications, but we use a concrete syntax that allows them to represented as n-ary relations.

```
>>> parsed = lp.parse('see(a, b)')
>>> parsed.argument
<IndividualVariableExpression b>
>>> parsed.function
<ApplicationExpression see(a)>
>>> parsed.function.function
<VariableExpression see>
```

Relations are represented semantically in NLTK in the standard set-theoretic way: as sets of tuples. For example, let's suppose we have a domain of discourse consisting of the individuals Adam, Betty and Fido, where Adam is a boy, Betty is a girl and Fido is a dog. For mnemonic reasons, we use b1, g1 and d1 as the corresponding labels in the model. We can declare the domain as follows:

```
>>> dom2 = set(['b1', 'g1', 'd1'])
```

As before, we are going to initialize a valuation with a list of (symbol, value) pairs:

```
>>> v = [('adam','b1'),('betty','g1'),('fido','d1'),
... ('boy',set(['b1'])),('girl',set(['g1'])), ('dog',set(['d1'])),
... ('walk',set(['g1', 'd1'])),
... ('see', set([('b1','g1'),('d1','b1'),('g1','d1'),]))]
>>> val2 = nltk.sem.Valuation(v)
>>> print val2
{'adam': 'b1',
 'betty': 'g1',
 'boy': set([('b1',)]),
 'dog': set([('d1',)]),
 'fido': 'd1',
 'girl': set([('g1',)]),
 'see': set([('b1', 'g1'), ('d1', 'b1'), ('g1', 'd1')]),
 'walk': set([('d1',), ('g1',)])}
```

So according to this valuation, the value of `see` is a set of tuples such that Adam sees Betty, Fido sees Adam, and Betty sees Fido.

You may have noticed that our unary predicates (i.e, `boy`, `girl`, `dog`) also come out represented as sets of singleton tuples, rather than just sets of individuals. This is a convenience which allows us to have a uniform treatment of relations of any arity. In order to combine a unary relation with an argument, we use the function `app()`. If the input relation is unary, then `app()` returns a Boolean value; if the input is n-ary, for $n > 1$, then `app()` returns an n-1-ary relation.

```
>>> from nltk.sem import app
>>> boy = val2['boy']
>>> app(boy, 'b1')
True
>>> app(boy, 'g1')
False
>>> see = val2['see']
>>> app(see, 'g1')
set([('d1',)])
>>> app(app(see, 'g1'), 'd1')
True
```

11.3.2 Individual Variables and Assignments

In FOL, arguments of predicates can also be individual variables such as `x`, `y` and `z`. These can be thought of as similar to personal pronouns like *he*, *she* and *it*, in that we need to know about the context of use in order to figure out their denotation. In our models, the counterpart of a context of use is a variable **Assignment**. This is a mapping from individual variables to entities in the domain. Assignments are created using the `Assignment` constructor, which also takes the model's domain of discourse as a parameter. We are not required to actually enter any bindings, but if we do, they are in a (variable, value) format similar to what we say earlier for valuations.

```
>>> g = nltk.sem.Assignment(dom2, [('x', 'b1'), ('y', 'd1')])
>>> g
{'y': 'd1', 'x': 'b1'}
```

In addition, there is a `print()` format for assignments which uses a notation closer to that in logic textbooks:

```
>>> print g
g[d1/y][b1/x]
```

Let's now look at how we can evaluate an atomic formula of FOL. First, we create a model, then we use the `evaluate()` method to compute the truth value.

```
>>> m2 = nltk.sem.Model(dom2, val2)
>>> m2.evaluate('see(betty, y)', g)
True
```

What's happening here? Essentially, we are making a call to `app(app(see, 'g1'), 'd1')` just as in our earlier example. However, when the interpretation function encounters the variable `'y'`, rather than checking for a value in `val2`, it asks the variable assignment `g` to come up with a value:

```
>>> g['y']
'd1'
```

Since we already know that 'b1' and 'g1' stand in the see relation, the value `True` is what we expected. In this case, we can say that assignment `g` **satisfies** the formula 'see(adam, y)'. By contrast, the following formula evaluates to `False` relative to `g` — check that you see why this is.

```
>>> m2.evaluate('see(x, y)', g)
False
```

In our approach (though not in standard first-order logic), variable assignments are *partial*. For example, `g` says nothing about any variables apart from `'x'` and `'y''. The method `` purge()` clears all bindings from an assignment.

```
>>> g.purge()
>>> g
{}
```

If we now try to evaluate a formula such as 'see(adam, y)' relative to `g`, it is like trying to interpret a sentence containing a *she* when we don't know what *she* refers to. In this case, the evaluation function fails to deliver a truth value.

```
>>> m2.evaluate('see(adam, y)', g)
'Undefined'
```

11.3.3 Quantification and Scope

First-order logic standardly offers us two quantifiers, *all* (or *every*) and *some*. These are formally written as $\forall$ and , respectively. At the syntactic level, quantifiers are used to bind individual variables like `'x'` and `'y'`. The following two sets of examples show a simple English example, a logical representation, and the encoding which is accepted by the NLTK `logic` module.

(8) a. Every dog barks.

b. $\forall$x.(dog(x) → bark(x))

c. `all x.(dog(x) -> bark(x))`

(9) a. Some girl walks.

b. $x.(\mathsf{girl}(x) \wedge \mathsf{walk}(x))$

c. `some x.(girl(x) & walk(x))`

In the (9c), the quantifier `some` binds both occurences of the variable `'x'`. As a result, (9c) is said to be a **closed formula**. By contrast, if we look at the body of (9c), the variables are unbound:

(10) girl(x) & walk(x)

(10) is said to be an **open formula**. As we saw earlier, the interpretation of open formulas depends on the particular variable assignment that we are using.

One of the crucial insights of modern logic is that the notion of variable satisfaction can be used to provide an interpretation to quantified formulas. Let's continue to use (9c) as an example. When is it true? Let's think about all the individuals in our domain, i.e., in `dom2`. We want to check whether any of these individuals have the property of being a girl and walking. In other words, we want to know if there is some `u` in `dom2` such that `g[u/x]` satisfies the open formula (10). Consider the following:

```
>>> m2.evaluate('some x.(girl(x) & walk(x))', g)
True
```

`evaluate()` returns `True` here because there is some `u` in `dom2` such that (10) is satisfied by an assigment which binds `'x'` to `u`. In fact, `g1` is such a `u`:

```
>>> m2.evaluate('girl(x) & walk(x)', g.add('x', 'g1'))
True
```

One useful tool offered by NLTK is the `satisfiers()` method. This lists all the individuals that satisfy an open formula. The method parameters are a parsed formula, a variable, and an assignment. Here are a few examples:

```
>>> fmla1 = lp.parse('girl(x) | boy(x)')
>>> m2.satisfiers(fmla1, 'x', g)
set(['b1', 'g1'])
>>> fmla2 = lp.parse('girl(x) -> walk(x)')
>>> m2.satisfiers(fmla2, 'x', g)
set(['b1', 'g1', 'd1'])
>>> fmla3 = lp.parse('walk(x) -> girl(x)')
>>> m2.satisfiers(fmla3, 'x', g)
set(['b1', 'g1'])
```

It's useful to think about why `fmla2` and `fmla3` receive the values they do. In particular, recall the truth conditions for `->` (encoded via the function `IMPLIES()` in every model):

```
>>> for first in [True, False]:
...     for second in [True, False]:
...         print "%s %s => %s" % (first, second, m2.IMPLIES(first, second))
True True => True
True False => False
False True => True
False False => True
```

This means that `fmla2` is equivalent to this:

(11) `- girl(x) | walk(x)`

That is, (11) is satisfied by something which either isn't a girl or walks. Since neither `b1` (Adam) nor `d1` (Fido) are girls, according to model `m2`, they both satisfy the whole formula. And of course `g1` satisfies the formula because `g1` satisfies both disjuncts. Now, since every member of the domain of discourse satisfies `fmla2`, the corresponding universally quantified formula is also true.

```
>>> m2.evaluate('all x.(girl(x) -> walk(x))', g)
True
```

In other words, a universally quantified formula $\forall x.\varphi$ is true with respect to `g` just in case for every `u`, φ is true with respect to `g[u/x]`.

11.3.4 Quantificatier Scope Ambiguity

What happens when we want to give a formal representation of a sentence with *two* quantifiers, such as the following?

(12) Every girl chases a dog.

There are (at least) two ways of expressing (12) in FOL:

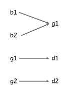
(13) a. $\forall x.((\text{girl } x) \rightarrow y.((\text{dog } y) \wedge (\text{chase } y\ x)))$

b. $y.((\text{dog } y) \wedge \forall x.((\text{every } x) \rightarrow (\text{chase } y\ x)))$

Can we use both of these? Then answer is Yes, but they have different meanings. (13b) is logically stronger than (13a): it claims that there is a unique dog, say Fido, which is chased by every girl. (13a), on the other hand, just requires that for every girl g, we can find some dog which d chases; but this could be a different dog in each case. We distinguish between (13a) and (13b) in terms of the **scope** of the quantifiers. In the first, $\forall$ has wider scope than , while in (13b), the scope ordering is reversed. So now we have two ways of representing the meaning of (12), and they are both quite legitimate. In other words, we are claiming that (12) is *ambiguous* with respect to quantifier scope, and the formulas in (13) give us a formal means of making the two readings explicit. However, we are not just interested in associating two distinct representations with (12). We also want to show in detail how the two representations lead to different conditions for truth in a formal model.
In order to examine the ambiguity more closely, let's fix our valuation as follows:

```
>>> v3 = [('john', 'b1'),
... ('mary', 'g1'),
... ('suzie', 'g2'),
... ('fido', 'd1'),
... ('tess', 'd2'),
... ('noosa', 'n'),
... ('girl', set(['g1', 'g2'])),
... ('boy', set(['b1', 'b2'])),
... ('dog', set(['d1', 'd2'])),
... ('bark', set(['d1', 'd2'])),
... ('walk', set(['b1', 'g2', 'd1'])),
... ('chase', set([('b1', 'g1'), ('b2', 'g1'), ('g1', 'd1'), ('g2', 'd2')])),
... ('see', set([('b1', 'g1'), ('b2', 'd2'), ('g1', 'b1'),
... ('d2', 'b1'), ('g2', 'n')])),
... ('in', set([('b1', 'n'), ('b2', 'n'), ('d2', 'n')])),
... ('with', set([('b1', 'g1'), ('g1', 'b1'), ('d1', 'b1'), ('b1', 'd1')]))]
>>> val3 = nltk.sem.Valuation(v3)
```

We can use the graph in (14) to visualize the chase relation.

(14) 

In (14), an arrow between two individuals *x* and *y* indicates that *x* chases *y*. So b1 and b2 both chase g1, while g1 chases d1 and g2 chases d2. In this model, formula scope2a_ above is true but scope2b_ is false. One way of exploring these results is by using the satisfiers() method of Model objects.

```
>>> dom3 = val3.domain
>>> m3 = nltk.sem.Model(dom3, val3)
>>> g = nltk.sem.Assignment(dom3)
>>> fmla1 = lp.parse('(girl(x) -> exists y.(dog(y) and chase(x, y)))')
>>> m3.satisfiers(fmla1, 'x', g)
set(['g2', 'g1', 'n', 'b1', 'b2', 'd2', 'd1'])
>>>
```

This gives us the set of individuals that can be assigned as the value of x in fmla1. In particular, every girl is included in this set. By contrast, consider the formula fmla2 below; this has no satisfiers for the variable y.

```
>>> fmla2 = lp.parse('(dog(y) & all x.(girl(x) -> chase(x, y)))')
>>> m3.satisfiers(fmla2, 'y', g)
set([])
>>>
```

That is, there is no dog that is chased by both g1 and g2. Taking a slightly different open formula, fmla3, we can verify that there is a girl, namely g1, who is chased by every boy.

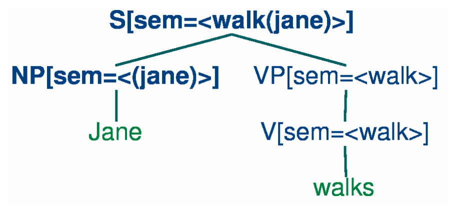
```
>>> fmla3 = lp.parse('(girl(y) & all x.(boy(x) -> chase(x, y)))')
>>> m3.satisfiers(fmla3, 'y', g)
set(['g1'])
```

11.4 Evaluating English Sentences

11.4.1 Using the sem Feature

Until now, we have taken for granted that we have some appropriate logical formulas to interpret. However, ideally we would like to derive these formulas from natural language input. One relatively easy way of achieving this goal is to build on the grammar framework developed in Chapter 10. Our first step is to introduce a new feature, sem. Because values of sem generally need to be treated differently from other feature values, we use the convention of enclosing them in angle brackets. (15) illustrates a first approximation to the kind of analyses we would like to build.

(15)



Thus, the `sem` value at the root node shows a semantic representation for the whole sentence, while the `sem` values at lower nodes show semantic representations for constituents of the sentence. So far, so good, but how do we write grammar rules which will give us this kind of result? To be more specific, suppose we have a NP and VP constituents with appropriate values for their `sem` nodes? If you reflect on the machinery that was introduced in discussing the λ calculus, you might guess that function application will be central to composing semantic values. You will also remember that our feature-based grammar framework gives us the means to refer to *variable* values. Putting this together, we can postulate a rule like (16) for building the `sem` value of an S. (Observe that in the case where the value of `sem` is a variable, we omit the angle brackets.)

(16)
```
S[sem = <app(?vp,?subj)>] -> NP[sem=?subj] VP[sem=?vp]
```

(16) tells us that given some `sem` value `?subj` for the subject NP and some `sem` value `?vp` for the VP, the `sem` value of the S mother is constructed by applying `?vp` as a functor to `?np`. From this, we can conclude that `?vp` has to denote a function which has the denotation of `?np` in its domain; in fact, we are going to assume that `?vp` denotes a curryed characteristic function on individuals. (16) is a nice example of building semantics using **the principle of compositionality**: that is, the principle that the semantics of a complex expression is a function of the semantics of its parts.

To complete the grammar is very straightforward; all we require are the rules shown in (17).

(17)
```
VP[sem=?v] -> IV[sem=?v]
NP[sem=<jane>] -> 'Jane'
IV[sem=<walk>] -> 'walks'
```

The VP rule says that the mother's semantics is the same as the head daughter's. The two lexical rules just introduce non-logical constants to serve as the semantic values of *Jane* and *walks* respectively. This grammar can be parsed using the chart parser in `parse.featurechart`, and the trace in (18) shows how semantic values are derived by feature unification in the process of building a parse tree.

(18)
```
Predictor |> . .| S[sem='(?vp ?subj)'] -> * NP[sem=?subj] VP[sem=?vp]
Scanner   |[-] .| [0:1] 'Jane'
Completer |[-> .| S[sem='(?vp john)'] -> NP[sem='john'] * VP[sem=?vp]
Predictor |. > .| VP[sem=?v] -> * IV[sem=?v]
Scanner   |. [-]| [1:2] 'walks'
Completer |. [-]| VP[sem='walk'] -> IV[sem='walk'] *
Completer |[===]| S[sem='(walk john)'] -> NP[sem='john'] VP[sem='walk'] *
Completer |[===]| [INIT] -> S *
```

11.4.2 Quantified NPs

You might be thinking this is all too easy — surely there is a bit more to building compositional semantics. What about quantifiers, for instance? Right, this is a crucial issue. For example, we want (19a) to be given a semantic representation like (19b). How can this be accomplished?

(19) a. A dog barks.

b. `'exists x.(dog(x) & (bark(x))'`

Let's make the assumption that our *only* operation for building complex semantic representations is function application. Then our problem is this: how do we give a semantic representation to quantified NPs such as *a dog* so that they can be combined with something like `'walk'` to give a result like (19b)? As a first step, let's make the subject's `sem` value act as the functor rather than the argument. Now we are looking for way of instantiating `?np` so that (20a) is equivalent to (20b).

(20) a. [sem=<bark(?np)>]

b. [sem=<exist x.(dog(x) & bark(x))>]

This is where λ abstraction comes to the rescue; doesn't (20) look a bit reminiscent of carrying out β-reduction in the λ-calculus? In other words, we want a λ term M to replace `'?np'` so that applying M to `'bark'` yields (19b). To do this, we replace the occurence of `'bark'` in (19b) by a variable `'P'`, and bind the variable with λ, as shown in (21).

(21) `'\P.exists x.(dog(x) & P(x))'`

As a point of interest, we have used a different style of variable in (21), that is `'P'` rather than `'x'` or `'y'`. This is to signal that we are abstracting over a different kind of thing — not an individual, but a function from **Ind** to **Bool**. So the type of (21) as a whole is ((**Ind** → **Bool**) → **Bool**). We will take this to be the type of NPs in general. To illustrate further, a universally quantified NP will look like (22).

(22) `'\P.all x.(dog(x) -> P(x))'`

We are pretty much done now, except that we also want to carry out a further abstraction plus application for the process of combining the semantics of the determiner *a* with the semantics of *dog*. Applying (21) as a functor to `'bark'` gives us `'(\P.exists x.(dog (x) & P(x)) bark)'`, and carrying out β-reduction yields just what we wanted, namely (19b).

NLTK provides some utilities to make it easier to derive and inspect semantic interpretations. `text_interpret()` is intended for batch interpretation of a list of input sentences. It builds a dictionary `d` where for each sentence `sent` in the input, `d[sent]` is a list of paired trees and semantic representations for `sent`. The value is a list, since `sent` may be syntactically ambiguous; in the following example, we just look at the first member of the list.

```
>>> grammar = nltk.data.load('grammars/sem1.fcfg')
>>> result  = nltk.sem.text_interpret(['a dog barks'], grammar, beta_reduce=0)
>>> (syntree, semrep) = result['a dog barks'][0]
>>> print syntree
(S[sem=<exists x.(dog(x) & bark(x))>]
  (NP[sem=<\P.exists x.(dog(x) & P(x))>]
    (Det[sem=<\Q.\P.exists x.(Q(x) & P(x))>] a)
      (N[sem=<dog>] dog))
   (VP[sem=<\x.bark(x)>] (IV[sem=<\x.bark(x)>] barks)))
```

By default, the semantic representation that is produced by `text_interpret()` has already undergone β-reduction, but in the above example, we have overridden this. Subsequent reduction is possible using the `simplify()` method, and Boolean connectives can be placed in infix position with the `infixify()` method.

```
>>> print semrep.simplify()
exists x.(dog(x) & bark(x))
```

11.4.3 Transitive Verbs

Our next challenge is to deal with sentences containing transitive verbs, such as (23).

(23) Suzie chases a dog.

The output semantics that we want to build is shown in (24).

(24) `'exists x.(dog(x) & chase(suzie, x))'`

Let's look at how we can use λ-abstraction to get this result. A significant constraint on possible solutions is to require that the semantic representation of *a dog* be independent of whether the NP acts as subject or object of the sentence. In other words, we want to get (24) as our output while sticking to (21) as the NP semantics. A second constraint is that VPs should have a uniform type of interpretation regardless of whether they consist of just an intransitive verb or a transitive verb plus object. More specifically, we stipulate that VPs always denote characteristic functions on individuals. Given these constraints, here's a semantic representation for *chases a dog* which does the trick.

(25) `'\y.exists x.(dog(x) & chase(y, x))'`

Think of (25) as the property of being a *y* such that for some dog *x*, *y* chases *x*; or more colloquially, being a *y* who chases a dog. Our task now resolves to designing a semantic representation for *chases* which can combine via `app` with (21) so as to allow (25) to be derived.

Let's carry out a kind of inverse β-reduction on (25), giving rise to (26).

Let Then we are part way to the solution if we can derive (26), where `'X'` is applied to `'\z. chase(y, z)'`.

(26) `'(\P.exists x.(dog(x) and P(x)) \z.chase(y, z))'`

(26) may be slightly hard to read at first; you need to see that it involves applying the quantified NP representation from (21) to `'\z.(chase z y))'`. (26) is of course equivalent to (25).

Now let's replace the functor in (26) by a variable `'X'` of the same type as an NP; that is, of type ((**Ind** → **Bool**) → **Bool**).

(27) `'X(\z.chase(y, z))'`

The representation of a transitive verb will have to apply to an argument of the type of `'X'` to yield a functor of the type of VPs, that is, of type (**Ind** → **Bool**). We can ensure this by abstracting over both the `'X'` variable in (27) and also the subject variable `'y'`. So the full solution is reached by giving *chases* the semantic representation shown in (28).

(28) `'\X y.X(\x.chase(y, x))'`

If (28) is applied to (21), the result after β-reduction is equivalent to (25), which is what we wanted all along:

(29)
```
'(\X y.(X \x.(chase(y, x)) \P.exists x.(dog(x) & P(x))'

'(\y.(\P.exists x.(dog(x) & P(x)) \x.chase(y, x))'

'\y.(exists x.(dog(x) & chase(y, x)))'
```

In order to build a semantic representation for a sentence, we also need to combine in the semantics of the subject NP. If the latter is a quantified expression like *every girl*, everything proceeds in the same way as we showed for *a dog barks* earlier on; the subject is translated as a functor which is applied to the semantic representation of the VP. However, we now seem to have created another problem for ourselves with proper names. So far, these have been treated semantically as individual constants, and these cannot be applied as functors to expressions like (25). Consequently, we need to come up with a different semantic representation for them. What we do in this case is re-interpret proper names so that they too are functors, like quantified NPs. (30) shows the required λ expression for *Suzie*.

(30) `'\P.P(suzie)'`

(30) denotes the characteristic function corresponding to the set of all properties which are true of Suzie. Converting from an individual constant to an expression like (28) is known as **type raising**, and allows us to flip functors with arguments. That is, type raising means that we can replace a Boolean-valued application such as (f a) with an equivalent application (λP.(P a) f).

One important limitation of the approach we have presented here is that it does not attempt to deal with scope ambiguity. Instead, quantifier scope ordering directly reflects scope in the parse tree. As a result, a sentence like (12), repeated here, will always be translated as (32a), not (32b).

(31) Every girl chases a dog.

(32) a. `'all x.((girl x) implies some y.  ((dog y) and (chase y x)))'`

b. `'some y.  (dog y) and all x.  ((girl x) implies (chase y x)))'`

This limitation can be overcome, for example using the hole semantics described in [Blackburn & Bos, 2005], but discussing the details would take us outside the scope of the current chapter.

Now that we have looked at some slightly more complex constructions, we can evaluate them in a model. In the following example, we derive two parses for the sentence *every boy chases a girl in Noosa*, and evaluate each of the corresponding semantic representations in the model `model0.py` which we have imported.

```
>>> grammar = nltk.data.load('grammars/sem2.fcfg')
>>> val4 = nltk.data.load('grammars/valuation1.val')
>>> dom4 = val4.domain
>>> m4 = nltk.sem.Model(dom4, val4)
>>> g = nltk.sem.Assignment(dom4)
>>> sent = 'every boy chases a girl in Noosa'
>>> result = nltk.sem.text_evaluate([sent], grammar, m4, g)
>>> for (syntree, semrep, value) in result[sent]:
...         print "'%s' is %s in Model m\n" % (semrep, value)
'all x.(boy(x) -> (exists z2.(girl(z2) & chase(x,z2)) & in(x,noosa)))' is True in M

'all x.(boy(x) -> (exists z2.(girl(z2) & chase(x,z2)) & in(x,noosa)))' is True in M

'all x.(boy(x) -> exists z3.((girl(z3) & in(z3,noosa)) & chase(x,z3)))' is False in

'all x.(boy(x) -> exists z3.((girl(z3) & in(z3,noosa)) & chase(x,z3)))' is False in
```

11.5 Case Study: Extracting Valuations from Chat-80

Building `Valuation` objects by hand becomes rather tedious once we consider larger examples. This raises the question of whether the relation data in a `Valuation` could be extracted from some pre-existing source. The `corpora.chat80` module provides an example of extracting data from the Chat-80 Prolog knowledge base (which included as part of the NLTK `corpora` distribution).

Chat-80 data is organized into collections of clauses, where each collection functions as a table in a relational database. The predicate of the clause provides the name of the table; the first element of the tuple acts as the 'key'; and subsequent elements are further columns in the table.

In general, the name of the table provides a label for a unary relation whose extension is all the keys. For example, the table in `cities.pl` contains triples such as (33).

(33) `'city(athens,greece,1368).'`

Here, `'athens'` is the key, and will be mapped to a member of the unary relation city.

The other two columns in the table are mapped to binary relations, where the first argument of the relation is filled by the table key, and the second argument is filled by the data in the relevant column. Thus, in the `city` table illustrated by the tuple in (33), the data from the third column is extracted into a binary predicate population_of, whose extension is a set of pairs such as `'(athens, 1368)'`.

In order to encapsulate the results of the extraction, a class of `Concepts` is introduced. A `Concept` object has a number of attributes, in particular a `prefLabel` and `extension`, which make it easier to inspect the output of the extraction. The `extension` of a `Concept` object is incorporated into a `Valuation` object.

As well as deriving unary and binary relations from the Chat-80 data, we also create a set of individual constants, one for each entity in the domain. The individual constants are string-identical to the entities. For example, given a data item such as `'zloty'`, we add to the valuation a pair `('zloty', 'zloty')`. In order to parse English sentences that refer to these entities, we also create a lexical item such as the following for each individual constant:

(34) `PropN[num=sg, sem=<\P.P(zloty)>] -> 'Zloty'`

The `chat80` module can be found in the `corpora` package. The attribute `chat80.items` gives us a list of Chat-80 relations:

```
>>> from nltk.corpus import chat80
>>> chat80.items
 ('borders', 'circle_of_lat', 'circle_of_long', 'city', ...)
```

The `concepts()` method shows the list of `Concepts` that can be extracted from a `chat80` relation, and we can then inspect their extensions.

```
>>> concepts = chat80.concepts('city')
>>> concepts
[Concept('city'), Concept('country_of'), Concept('population_of')]
>>> rel = concepts[1].extension
>>> list(rel)[:5]
[('chungking', 'china'), ('karachi', 'pakistan'),
('singapore_city', 'singapore'), ('athens', 'greece'),
('birmingham', 'united_kingdom')]
```

In order to convert such an extension into a valuation, we use the `make_valuation()` method; setting `read=True` creates and returns a new `Valuation` object which contains the results.

```
>>> val = nltk.corpus.chat80.make_valuation(concepts, read=True)
>>> val['city']['calcutta']
True
>>> val['country_of']['india']
{'hyderabad': True, 'delhi': True, 'bombay': True,
'madras': True, 'calcutta': True}
>>> dom = val.domain
>>> g = nltk.sem.Assignment(dom)
>>> m = nltk.sem.Model(dom, val)
>>> m.evaluate(r'\x .population_of(jakarta, x)', g)
{'533': True}
```

Note

Population figures are given in thousands. Bear in mind that the geographical data used in these examples dates back at least to the 1980s, and was already somewhat out of date at the point when [Warren & Pereira, 1982] was published.

11.6 Summary

- Semantic Representations (SRs) for English are constructed using a language based on the λ-calculus, together with Boolean connectives, equality, and first-order quantifiers.
- β-reduction in the λ-calculus corresponds semantically to application of a function to an argument. Syntactically, it involves replacing a variable bound by λ in the functor with the expression that provides the argument in the function application.
- If two λ-abstracts differ only in the label of the variable bound by λ, they are said to be α equivalents. Relabeling a variable bound by a λ is called α-conversion.
- Currying of a binary function turns it into a unary function whose value is again a unary function.
- FSRL has both a syntax and a semantics. The semantics is determined by recursively evaluating expressions in a model.
- A key part of constructing a model lies in building a valuation which assigns interpretations to non-logical constants. These are interpreted as either curried characteristic functions or as individual constants.
- The interpretation of Boolean connectives is handled by the model; these are interpreted as characteristic functions.
- An open expression is an expression containing one or more free variables. Open expressions only receive an interpretation when their free variables receive values from a variable assignment.
- Quantifiers are interpreted by constructing, for a formula φ[x] open in variable x, the set of individuals which make φ[x] true when an assignment *g* assigns them as the value of x. The quantifier then places constraints on that set.
- A closed expression is one that has no free variables; that is, the variables are all bound. A closed sentence is true or false with respect to all variable assignments.

- Given a formula with two nested quantifiers Q_1 and Q_2, the outermost quantifier Q_1 is said to have wide scope (or scope over Q_2). English sentences are frequently ambiguous with respect to the scope of the quantifiers they contain.

- English sentences can be associated with an SR by treating `sem` as a feature. The `sem` value of a complex expressions typically involves functional application of the `sem` values of the component expressions.

- Model valuations need not be built by hand, but can also be extracted from relational tables, as in the Chat-80 example.

11.7 Exercises

1. ◑ Modify the `sem.evaluate` code so that it will give a helpful error message if an expression is not in the domain of a model's valuation function.

2. ★ Specify and implement a typed functional language with quantifiers, Boolean connectives and equality. Modify `sem.evaluate` to interpret expressions of this language.

3. ★ Extend the `chat80` code so that it will extract data from a relational database using SQL queries.

4. ★ Taking [WarrenPereira1982] as a starting point, develop a technique for converting a natural language query into a form that can be evaluated more efficiently in a model. For example, given a query of the form `'(P(x) & Q(x)'`, convert it to `'(Q(x) & P(x)'` if the extension of `'Q'` is smaller than the extension of `'P'`.

11.8 Further Reading

For more examples of semantic analysis with NLTK, please see the guides at `http://nltk.org/doc/guides/sem.html` and `http://nltk.org/doc/guides/logic.html`.

The use of characteristic functions for interpreting expressions of natural language was primarily due to Richard Montague. [Dowty, Wall, & Peters, 1981] gives a comprehensive and reasonably approachable introduction to Montague's grammatical framework.

A more recent and wide-reaching study of the use of a λ based approach to natural language can be found in [Carpenter, 1997].

[Heim & Kratzer, 1998] is a thorough application of formal semantics to transformational grammars in the Government-Binding model.

[Blackburn & Bos, 2005] is the first textbook devoted to computational semantics, and provides an excellent introduction to the area.