Chapter 11

Semantic Interpretation

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, as we pointed out in Chapter 1, 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).

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

(1b) 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 x asia) and (country x)) and (border mediterranean x))
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.

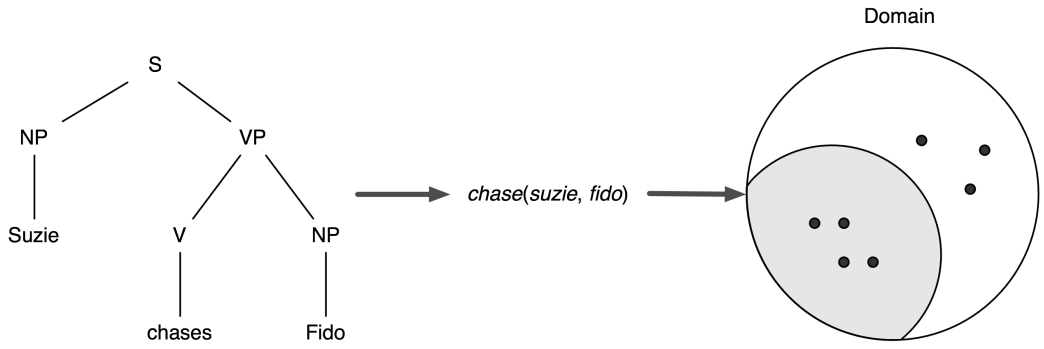
A widely adopted approach to representing meaning — or at least, some aspects of meaning — involves translating expressions of natural language into first-order logic (FOL). From a computational point of view, a strong argument in favour of FOL is that it strikes a reasonable balance between expressiveness 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 FSRL 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 The Lambda Calculus

In a functional programming language, computation can be carried out by reducing an expression E according to specified rewrite rules. This reduction is carried out on subparts of E, and terminates when no further subexpressions can be reduced. The resulting expression $E*$ is called the **Normal Form** of E. Table 11.1 gives an example of reduction involving a simple Python expression (where ' ' means 'reduces to'):

	`len(max(['cat', 'zebra', 'rabbit'] + ['gopher']))`
	`len(max(['cat', 'zebra', 'rabbit', 'gopher']))`
	`len('zebra')`
	`5`

Table 11.1: Reduction of functions

Thus, working from the inside outwards, we first reduce list concatenation to the normal form shown in the second row, we then take the `max()` element of the list (under alphabetic ordering), and then compute the length of that string. The final expression, `5`, is considered to be the output of the program. This fundamental notion of computation is modeled in an abstract way by something called the λ-calculus (λ is a Greek letter pronounced 'lambda').

The first basic concept in the λ-calculus is **application**, represented by an expression of the form (F A), where F is considered to be a function, and A is considered to be an argument (or input) for

F. For example, (walk x) is an application. Moreover, application expressions can be applied to other expressions. So in a functional framework, binary addition might represented as ((+ x) y) rather than (x + y). Note that + is being treated as a function which is applied to its first argument x to yield a function (+ x) that is then applied to the second argument y.

The second basic concept in the λ-calculus is **abstraction**. If M[x] is an expression containing the variable x, then λx.M[x] denotes the function x → M[x]. Abstraction and application are combined in the expression (λx.((+ x) 3) 4), which denotes the function x → x + 3 applied to 4, giving 4 + 3, which is 7. In general, we have

(5) (λx.M[x] N) = M[N],

where M[N] is the result of replacing all occurences of x in M by N. This axiom of the lambda calculus is known as **β-conversion**. β-conversion is the primary form of reduction in the λ-calculus.

The module `nltk_lite.semantics.logic` can parse expressions of the λ-calculus. The λ symbol is represented as `'\'`. In order to avoid having to escape this with a second `'\'`, we use raw strings in parsable expressions.

```
>>> from nltk_lite.semantics import logic
>>> lp = logic.Parser()
>>> lp.parse(r'(walk x)')
ApplicationExpression('walk', 'x')
>>> lp.parse(r'\x.(walk x)')
LambdaExpression('x', '(walk x)')
```

An `ApplicationExpression` has subparts consisting of the function and the argument; a `LambdaExpression` has subparts consisting of the variable (e.g., `x`) that is bound by the λ and the body of the expression (e.g., `walk`).

The λ-calculus is a calculus of functions; by itself, it says nothing about logical structure. Although it is possible to define logical operators within the λ-calculus, it is much more convenient to adopt a hybrid approach which supplements the λ-calculus with logical and non-logical constants as primitives. In order to show how this is done, we turn first to the language of propositional logic.

11.3 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 $p \rightarrow q \vee r$, which might represent the logical structure of an English sentence such as *if it is raining, then Kim will take an umbrella or Lee will get wet*. 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 `nltk_lite.semantics.logic`, and are parsed as objects of the class `ApplicationExpression` (i.e., function expressions). However, infix notation is also allowed as an input format. The connectives themselves belong to the `Operator` class of expressions.

```
>>> lp.parse('(and p q)')
ApplicationExpression('(and p)', 'q')
>>> lp.parse('(p and q)')
ApplicationExpression('(and p)', 'q')
>>> lp.parse('and')
Operator('and')
>>>
```

Since a negated proposition is syntactically an application, the unary operator `not` and its argument must be surrounded by parentheses.

```
>>> lp.parse('(not (p and q))')
ApplicationExpression('not', '(and p q)')
>>>
```

To make the `print` output easier to read, we can invoke the `infixify()` method, which places binary Boolean operators in infix position.

```
>>> e = lp.parse('(and p (not a))')
>>> e
ApplicationExpression('(and p)', '(not a)')
>>> print e
(and p (not a))
>>> print e.infixify()
(p and (not a))
```

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}.

11.4 First-Order Logic

11.4.1 Predication

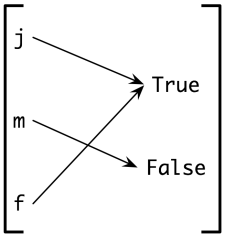
In first-order logic (FOL), propositions are analysed into predicates and arguments, which takes us a step closer to the structure of natural languages. The standard construction rules for FOL recognize **terms** such as individual variables and individual constants, and **predicates** which take differing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

pair j, m belongs to this set. In order to make it explicit that we are treating *see* as a relation, we'll use the symbol see_R as its semantic representation, and we'll call $\mathsf{see}_R(\mathsf{jane}, \mathsf{mike})$ an instance of the 'relational style' of representing predication.

Within the framework of the λ-calculus, there is an alternative approach in which predication is treated as function application. In this functional style of representation, *Jane sees Mike* is formalized as $((\ \mathsf{see}_f\ \mathsf{m})\ \mathsf{j})$ or — a shorthand with less brackets — as $(\ \mathsf{see}_f\ \mathsf{m}\ \mathsf{j})$. Rather than being modeled as a relation, see_f denotes a function. Before going into detail about this function, let's first look at a simpler case, namely the different styles of interpreting a unary predicate such as walk.

In the relational approach, walk_R denotes some set W of individuals. The formula $\mathsf{walk}_R(\mathsf{j})$ is true in a situation if and only if the individual denoted by j belongs to W. As we saw in Chapter 10, corresponding to every set S is the *characteristic function* f_S of that set. To be specific, suppose in some situation our domain of discourse D is the set containing the individuals j (Jane), m (Mike) and f (Fido); and the set of individuals that walk is $W = \{j, f\}$. So in this situation, the formulas $\mathsf{walk}_R(\mathsf{j})$ and $\mathsf{walk}_R(\mathsf{f})$ are both true, while $\mathsf{walk}_R(\mathsf{m})$ is false. Now we can use the characteristic function f_W as the interpretation of walk_f in the functional style. The diagram (6) gives a graphical representation of the mapping f_W.

(6)

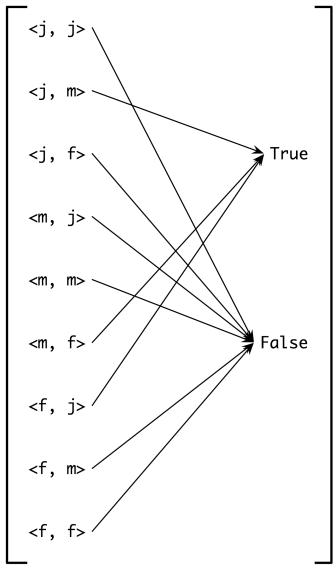


Binary relations can be converted into functions in a very similar fashion. Suppose for example that on the relational style of interpretation, see_R denotes the following set of pairs:

(7) $\{\quad j, m\quad ,\quad m, f\quad ,\quad f, j\quad \}$

That is, Jane sees Mike, Mike sees Fido, and Fido sees Jane. One option on the functional style would be to treat see_f as the expected characteristic function of this set, i.e., a function $f_S\colon D \times D \rightarrow \{True, False\}$ (i.e., from pairs of individuals to truth values). This mapping is illustrated in (8).

(8)



However, recall that we are trying to build up our semantic analysis compositionally; i.e., the meaning of a complex expression is a function of the meaning of its parts. In the case of a sentence, what are its parts? Presumably they are the subject NP and the VP. So let's consider what would be a suitable value for the VP *sees Fido*. It cannot be see_f denoting a function $D \times D \rightarrow \{True, False\}$, since this is looking for a *pair* of arguments. A better meaning representation would be $\lambda x.\mathsf{see}_R(\mathsf{x}, \mathsf{Fido})$, which is a function of a single argument, and can thus be applied to semantic representation of the subject *np*:gc. This invites the question: how should we represent the meaning of the transitive verb *see*? A possible answer is shown in (9).

(9) $\mathsf{see}_f = \lambda \mathsf{y}.\lambda x.\mathsf{see}_R(\mathsf{x}, \mathsf{y})$.

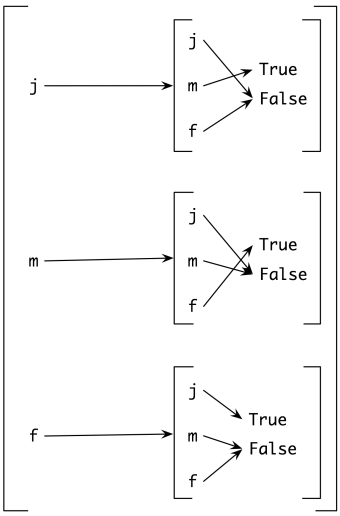
This defines see_f to be a function expression which can be applied first to the argument coming from the NP object and then to the argument comig from the NP subject. In (10), we show how the application of (9) to f and then to m gets reduced.

(10) $(\lambda \mathsf{y}.\lambda x.\mathsf{see}_R(\mathsf{x}, \mathsf{y})\ \mathsf{f}\ \mathsf{m})$ $(\lambda x.\mathsf{see}_R(\mathsf{x}, \mathsf{f})\ \mathsf{m})$ $\mathsf{see}_R(\mathsf{m}, \mathsf{f})$

(9) adopts a technique known as 'currying' (named after Haskell B. Curry), in which a binary function is converted into a function of one argument. As you can see, when we apply see_f to an argument such as f, the value is another function, namely the function denoted by $\lambda x.\mathsf{see}_R(\mathsf{x}, \mathsf{f})$.

Diagram (11) shows the curryed counterpart of (8). It presents a function *F* such that given the argument *j*, *F*(*j)* is a characteristic function that maps m to *True* and *j* and *f* to *False*. (While there are $2^3 = 8$ characteristic functions from our domain of three individuals into {*True*, *False*}, we have only shown the functions which are in the range of the function denoted by see_f.)

(11)



Now, rather than define see_f by abstracting over a formula containing see_R, we can interpret it directly as the function f: (**Ind** → (**Ind** → **Bool**)), as illustrated in (11). Table 11.2 summarizes the different approaches to predication that we have just examined.

English	Relational	Functional
Jane walks	walk(j)	(walk j)
Mike sees Fido	see(m, f)	((see f) m), (see f m)

Table 11.2: Representing Predication

In particular, one has to be careful to remember that in (see f m), the order of arguments is the reverse of what is found in see(m, f).

In order to be slightly more formal about how we are treating the syntax of first-order logic, it is helpful to look first at the **typed lambda calculus**. We will take as our basic types **Ind** and **Bool**, corresponding to the domain of individuals and {*True*, *False*} respectively. We define the set of types recursively. First, every basic type is a type. Second, If σ and τ are types, then (σ → τ) is also a type; this corresponds to the set of functions from things of type σ to things of type τ. We omit the parentheses around σ → τ if there is no ambiguity. For any type τ, we have a set **Var**(τ) of variables of type τ and **Con**(τ) of constants of type τ. We now define the set **Term**(τ) of λ-terms of type τ.

1. **Var**(τ) ⊆ **Term**(τ).
2. **Con**(τ) ⊆ **Term**(τ).
3. If α ∈ **Term**(σ → τ) and β ∈ **Term**(σ), then (α β) ∈ **Term**(τ) (function application).
4. If x ∈ **Var**(σ) and α ∈ **Term**(ϱ), then λx.α ∈ **Term**(τ), where τ = (σ → ϱ) (λ-abstraction).

We replace our earlier definition of formulas containing Boolean connectives (that is, in L_{prop}) by adding the following clause:

5. not ∈ **Con**(**Bool** → **Bool**), and and, or, implies and iff ∈ **Con**(**Bool** → (**Bool** → **Bool**)).

We also add a clause for equality between individual terms.

6. If $\alpha, \beta \in$ **Term**(**Ind**), then $\alpha = \beta \in$ **Term**(**Bool**).

If we return now to NLTK, we can see that our previous implementation of function application already does service for predication. We also note that λ-abstraction can be combined with terms that are conjoined by Boolean operators. For example, the following can be thought of as the property of being an *x* who walks and talks:

```
>>> from nltk_lite.semantics import logic
>>> lp = logic.Parser()
>>> lp.parse(r'\x.((walk x) and (talk x))')
LambdaExpression('x', '(and (walk x) (talk x))')
```

β-conversion can be invoked with the `simplify()` method of `ApplicationExpressions`. As we noted earlier, the"infixify()" method will place binary Boolean connectives in infix position.

```
>>> e = lp.parse(r'(\x.((walk x) and (talk x)) john)')
>>> e
ApplicationExpression('\x.(and (walk x) (talk x))', 'john')
>>> print e.simplify()
(and (walk john) (talk john))
>>> print e.simplify().infixify()
((walk john) and (talk john))
```

Up to this point, we have restricted ourselves to looking at formulas where all the arguments are individual constants (i.e., expressions in **Term**(**Ind**)), corresponding to proper names such as *Jane*, *Mike* and *Fido*. Yet a crucial ingredient of first-order logic is the ability to make general statements involving quantified expressions such as *all dogs* and *some cats*. We turn to this topic in the next section.

11.4.2 Quantification and Scope

First-order logic standardly offers us two quantifiers, *every* (or *all*) and *some*. These are formally written as $\forall$ and , respectively. 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.

(12a) Every dog barks.

(12b) $\forall$x.((dog x) $\rightarrow$ (bark x))

(12c) `all x.((dog x) implies (bark x))`

(13a) Some cat sleeps.

(13b) x.((cat x) $\wedge$ (sleep x))

(13c) `some x.((cat x) and (sleep x))`

The inclusion of first-order quantifiers motivates the final clause of the definition of our version of first-order logic.

7. If x $\in$ **Var**(**Ind**) and $\varphi \in$ **Term**(**Bool**), then $\forall$x.φ, x.$\varphi \in$ **Term**(**Bool**).

One important property of (12b) often trips people up. The logical rendering in effect says that *if* something is a dog, then it barks, but makes no commitment to the existence of dogs. So in a situation where nothing is a dog, (12b) will still come out true. (Remember that `'(p implies q)'` is true when `'p'` is false.) Now you might argue that (12b) does presuppose the existence of dogs, and that the logic formalization is wrong. But it is possible to find other examples which lack such a presupposition. For instance, we might explain that the value of the Python expression `re.sub('ate', '8', astring)` is the result of replacing all occurrences of `'ate'` in `astring` by `'8'`, even though there may in fact be no such occurrences.

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

(14) Every girl chases a dog.

There are (at least) two ways of expressing (14) in FOL:

(15a) $\forall$x.((girl x) $\rightarrow$ y.((dog y) $\wedge$ (chase y x)))

(15b) y.((dog y) $\wedge$ $\forall$x.((every x) $\rightarrow$ (chase y x)))

Can we use both of these? Then answer is Yes, but they have different meanings. (15b) is logically stronger than (15a): it claims that there is a unique dog, say Fido, which is chased by every girl. (15a), 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 (15a) and (15b) in terms of the **scope** of the quantifiers. In the first, $\forall$ has wider scope than , while in (15b), the scope ordering is reversed. So now we have two ways of representing the meaning of (14), and they are both quite legitimate. In other words, we are claiming that (14) is *ambiguous* with respect to quantifier scope, and the formulas in (15) give us a formal means of making the two readings explicit. However, we are not just interested in associating two distinct representations with (14). We also want to show in detail how the two representations lead to different conditions for truth in a formal model. This will be taken up in the next section.

11.4.3 Alphabetic Variants

When carrying out β-reduction, some care has to be taken with variables. Consider, for example, the λ terms (16a) and (16b), which differ only in the identity of a free variable.

(16a) λy.(see x y)

(16b) λy.(see z y)

Suppose now that we apply the λ-term λP.x.(P x) to each of these terms:

(17a) (λP.x.(P x) λy.(see x y))

(17b) (λP.x.(P x) λy.(see z y))

In principle, the results of the application should be semantically equivalent. But if we let the free variable x in (16a) be 'captured' by the existential quantifier in (17b), then after reduction, the results will be different:

(18a) x.(see x x)

(18b) x.(see z x)

(18a) means there is some x that sees him/herself, whereas (18b) means that there is some x that sees an unspecified individual y. What has gone wrong here? Clearly, we want to forbid the kind of variable capture shown in (18a), and it seems that we have been too literal about the label of the particular variable bound by the existential quantifier in the functor expression of (17a). In fact, given any variable-binding expression (involving $\forall$, or λ), the particular name chosen for the bound variable is completely arbitrary. For example, (19a) and (19b) are equivalent; they are called α **equivalents** (or **alphabetic variants**).

(19a) x.(P x)

(19b) z_0.(P z_0)

The process of relabeling bound variables (which takes us from (19a) to (19b)) is known as α-**conversion**. When we test for equality of `VariableBinderExpressions` in the `logic` module (i.e., using ==), we are in fact testing for α-equivalence:

```
>>> from nltk_lite.semantics import *
>>> lp = Parser()
>>> e1 = lp.parse('some x.(P x)')
>>> print e1
some x.(P x)
>>> e2 = e1.alpha_convert(Variable('z'))
>>> print e2
some z.(P z)
>>> e1 == e2
True
```

When β-reduction is carried out on an application (M N), we check whether there are free variables in N which also occur as bound variables in any subterms of M. Suppose, as in the example discussed above, that x is free in N, and that M contains the subterm x.(P x). In this case, we produce an alphabetic variant of x.(P x), say, z.(P z), and then carry on with the reduction. This relabeling is carried out automatically by the β-reduction code in `logic`, and the results can be seen in the following example.

```
>>> e3 = lp.parse('(\P.some x.(P x) \y.(see x y))')
>>> print e3
(\P.some x.(P x) \y.(see x y))
>>> print e3.simplify()
some z2.(see x z2)
```

11.4.4 Types and the Untyped Lambda Calculus

For convenience, let's give a name to language for semantic representations that we are using in `nltk_lite.semantics.logic`: FSRL (for Functional Semantic Representation Language). So far, we have glossed over the fact that the FSRL is based on an implementation of the *untyped* lambda calculus. That is, although we have introduced typing in order to aid exposition, FSRL is not constrained to honour that typing. In particular, there is no formal distinction between predicate expressions and individual expressions; anything can be applied to anything. Indeed, functions can be applied to themselves:

```
>>> lp.parse('(walk walk)')
ApplicationExpression('walk', 'walk')
```

By contrast, most standard approaches to natural language semantics forbid self-application (e.g., applications such as (walk walk)) by adopting a typed language of the kind presented above.

It is also standard to allow constants as basic expressions of the language, as indicated by our use **Con**(τ) is our earlier definitions. Correspondingly, we have used a mixture of convention and supplementary stipulations to bring FSRL closer to this more standard framework for natural language semantics. In particular, we use expressions like `x`, `y`, `z` or `x0`, `x1`, `x2` to indicate individual variables. In FSRL, we assign such strings to the class `IndVariableExpression`.

```
>>> lp.parse('x')
IndVariableExpression('x')
>>> lp.parse('x01')
IndVariableExpression('x01')
```

English-like expressions such as *dog*, *walk* and *john* will be non-logical constants (non-logical in contrast to logical constants such as *not* and *and*). In order to force `logic.Parser()` to recognize non-logical constants, we can initialize the parser with a list of identifiers.

```
>>> lp = Parser(constants=['dog', 'walk', 'see'])
>>> lp.parse('walk')
ConstantExpression('walk')
```

To sum up, while the untyped λ-calculus only recognizes one kind of basic expression other than λ, namely the class of variables (the class `VariableExpression` in `logic`), FSRL adds three further classes of basic expression: `IndVariableExpression`, `ConstantExpression` and `Operator` (Boolean connectives plus the equality relation =).

This completes our discussion of using a first-order language as a basis for semantic representation in NLTK. In the next section, we will study how FSRL is interpreted.

11.5 Formal Semantics

In the preceding sections, we presented some basic ideas about defining a semantic representation FSRL. We also showed informally how expressions of FSRL are paired up with natural language expressions. Later on, we will investigate a more systematic method for carrying out that pairing. But let's suppose for a moment that for any sentence S of English, we have a method of building a corresponding expression of first-order logic that represents the meaning of S (still a fairly distant goal, unfortunately). Would this be enough? Within the tradition of formal semantics, the answer would be No. To be concrete, consider (20a) and (20b).

(20a) Melbourne is an Australian city.

(20b) (((in australia) melbourne) ∧ (city melbourne))

(20a) makes a claim about the world. To know the meaning of (20a), we at least have to know the conditions under which it is true. Translating (20a) into (20b) may clarify some aspects of the structure of (20a), but we can still ask what the meaning of (20b) is. So we want to take the further step of

[1] When combined with logic, unrestricted self-application leads to Russell's Paradox.

giving truth conditions for (20b). To know the conditions under which a sentence is true or false is an essential component of knowing the meaning of that sentence. To be sure, truth conditions do not exhaust meaning. But if we can find some situation in which sentence A is true while sentence B is false, then we can be certain that A and B do not have the same meaning.

Now there are infinitely many sentences in **Term**(**Bool**) and consequently it is not possible to simply list the truth conditions. Instead, we give a *recursive* definition of truth. For instance, one of the clauses in the definition might look roughly like this:

(21) $(\varphi \wedge \psi)$ is True iff φ is True and ψ is True.

(21) is applicable to (20b); it allows us to decompose it into its conjuncts, and then proceed further with each of these, until we reach expressions — constants and variables — that cannot be broken down any further.

As we have already seen, all of our non-logical constants are interpreted either as individuals or as curried functions. What we are now going to do is make this notion of interpretation more precise by defining a **valuation** for non-logical constants, building on a set of predefined individuals in a **domain of discourse**. Together, the valuation and domain of discourse make up the main components of a *model* for sentences in our semantic representation language. The framework of model-theoretic semantics provides the tools for making the recursive definition of truth both formally and computationally explicit.

Our models stand in for possible worlds — or ways that the world could actually be. Within these models, we adopt the fiction that our knowledge is completely clearcut: sentences are either true or false, rather than probably true or true to some degree. (The only exception is that there may be expressions which do not receive any interpretation.)

More formally, a **model** for a first-order language L is a pair $<D, V>$, where D is a domain of discourse and V is a valuation function for the non-logical constants of L. Non-logical constants are interpreted by V as follows (recall that **Ind** is the type of entities and **Bool** is the type of truth values):

- if α is an individual constant, then $V(\alpha) \in D$.
- If γ is an expression of type (**Ind** $\rightarrow$ … (**Ind** $\rightarrow$ **Bool**)…), then $V(\gamma)$ is a function f : $D \rightarrow \ldots (D \rightarrow \{True, False\})\ldots)$.

As explained earlier, expressions of FSRL are not in fact explicitly typed. We leave it to you, the grammar writer, to assign 'sensible' values to expressions rather than enforcing any type-to-denotation consistency.

11.5.1 Characteristic Functions

Within the `semantics` package, curryed characteristic functions are implemented as a subclass of dictionaries, using the `CharFun` constructor.

```
>>> from nltk_lite.semantics import *
>>> cf = CharFun({'d1': CharFun({'d2': True}), 'd2': CharFun({'d1': True})})
```

Values of a `CharFun` are accessed by indexing in the usual way:

```
>>> cf['d1']
{'d2': True}
>>> cf['d1']['d2']
True
```

CharFuns are 'abbreviated' data structures in the sense that they omit key-value pairs of the form (e : False). In fact, they behave just like ordinary dictionaries on keys which are out of their domain, rather than yielding the value False:

```
>>> cf['not in domain']
Traceback (most recent call last):
...
KeyError: 'not in domain'
```

The assignment of False values is delegated to a wrapper method app() of the Model class. app`()` embodies the Closed World assumption; i.e., where ``m is an instance of Model:

```
>>> m.app(cf,'not in domain')
False
```

In practise, it is often more convenient to specify interpretations as n-ary relations (i.e., sets of n-tuples) rather than as n-ary functions. A CharFun object has a read() method which will convert such relations into curried characteristic functions, and a tuples() method which will perform the inverse conversion.

```
>>> s = set([('d1', 'd2'), ('d3', 'd4')])
>>> cf = CharFun()
>>> cf.read(s)
>>> cf
{'d2': {'d1': True}, 'd4': {'d3': True}}
>>> cf.tuples()
set([('d1', 'd2'), ('d3', 'd4')])
```

The function flatten() returns a set of the entities used as keys in a CharFun instance. The same information can be accessed via the domain attribute of CharFun.

```
>>> cf = CharFun({'d1' : {'d2': True}, 'd2' : {'d1': True}})
>>> flatten(cf)
set(['d2', 'd1'])
>>> cf.domain
set(['d2', 'd1'])
```

11.5.2 Valuations

A **Valuation** is a mapping from non-logical constants to appropriate semantic values in the model. Valuations are created using the Valuation constructor.

```
>>> val = Valuation({'Fido': 'd1', 'dog': {'d1': True, 'd2': True}})
>>> val['dog']
{'d2': True, 'd1': True}
>>> val['dog']['d1']
True
```

As with CharFun, an instance of Valuation has a read() method that allows valuations to be specified as relations rather than characteristic functions.

```
>>> setval = [('adam', 'b1'), ('betty', 'g1'),
... ('girl', set(['g2', 'g1'])), ('boy', set(['b1', 'b2'])),
... ('see', set([('b1', 'g1'), ('b2', 'g2'), ('g1', 'b1'), ('g2', 'b1')]))]
>>> val = Valuation()
>>> val.read(setval)
>>> print val
{'adam': 'b1',
'betty': 'g1',
'boy': {'b1': True, 'b2': True},
'girl': {'g2': True, 'g1': True},
'see': {'b1': {'g2': True, 'g1': True},
        'g1': {'b1': True},
        'g2': {'b2': True}}}
```

Valuations have a `domain` attribute, like `CharFun`, and also a `symbols` attribute.

```
>>> val.domain
set(['g1', 'g2', 'b2', 'b1'])
>>> val.symbols
['boy', 'girl', 'see, 'adam', 'betty']
```

11.5.3 Assignments

A variable **Assignment** is a mapping from individual variables to entities in the domain. As indicated earlier, individual variables are written with the letters `'x'`, `'y'`, `'w'` and `'z'`, optionally followed by an integer (e.g., `'x0'`, `'y332'`). Assignments are created using the `Assignment` constructor, which also takes the model's domain of discourse as a parameter.

```
>>> dom = set(['u1', 'u2', 'u3', 'u4'])
>>> g = Assignment(dom, {'x': 'u1', 'y': 'u2'})
>>> g
{'y': 'u2', 'x': 'u1'}
```

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

```
>>> print g
g[u2/y][u1/x]
```

It is possible to update an assignment using the `add()` method; this checks that the variable really is an individual variable, and also checks that the new value belongs to the domain of discourse.

```
>>> dom = set(['u1', 'u2', 'u3', 'u4'])
>>> g = Assignment(dom, {})
>>> g.add('u1', 'x')
{'x': 'u1'}
>>> g.add('u1', 'xyz')
Traceback (most recent call last):
...
AssertionError: Wrong format for an Individual Variable: 'xyz'
>>> g.add('u2', 'x').add('u3', 'y').add('u4', 'x0')
{'y': 'u3', 'x': 'u2', 'x0': 'u4'}
>>> g.add('u5', 'x')
Traceback (most recent call last):
...
AssertionError: u5 is not in the domain set(['u4', 'u1', 'u3', 'u2'])
```

11.5.4 `evaluate()` and `satisfy()`

The `Model` constructor takes two parameters, of type `set` and `Valuation` respectively. Assuming that we have already defined a `Valuation` `val`, it is convenient to use `val`'s domain as the domain for the model constructor.

```
>>> dom = val.domain
>>> m = Model(dom, val)
>>> g = Assignment(dom, {})
```

The top-level method of a `Model` instance is `evaluate()`, which assigns a semantic value to expressions of the `logic` module, under an assignment `g`:

```
>>> m.evaluate('all x. ((boy x) implies (not (girl x)))', g)
True
```

The function `evaluate()` is essentially a convenience for handling expressions whose interpretation yields the `Undefined` value. It then calls the recursive function `satisfy()`. Since most of the interesting work is carried out by `satisfy()`, we shall concentrate on the latter.

The `satisfy()` function needs to deal with the following kinds of expression:

- non-logical constants and variables;
- Boolean connectives;
- function applications;
- quantified formulas;
- lambda-abstracts.

We shall look at each of these in turn.

11.5.5 Evaluating Non-Logical Constants and Variables

When it encounters expressions which cannot be analysed into smaller components, `satisfy()` calls two subsdiary functions. The function `i()` is used to interpret non-logical constants and individual variables, while the variable assignment `g` is used to assign values to individual variables, as seen above.

Any atomic expression which cannot be assigned a value by `i()` or `g` raises an `Undefined` exception; this is caught by `evaluate()`, which returns the string `'Undefined'`. In the following examples, we have set tracing to `2` to give a verbose analysis of the processing steps.

```
>>> m.evaluate('(boy adam)', g, trace=2)
    i, g('boy') = {'b1': True, 'b2': True}
    i, g('adam') = b1
 '(boy adam)': {'b1': True, 'b2': True} applied to b1 yields True
 '(boy adam)' evaluates to True under M, g
 True
 >>> m.evaluate('(girl adam)', g, trace=2)
    i, g('girl') = {'g2': True, 'g1': True}
    i, g('adam') = b1
 '(girl adam)': {'g2': True, 'g1': True} applied to b1 yields False
 '(girl adam)' evaluates to False under M, g
 False
```

```
>>> m.evaluate('(walk adam)', g, trace=2)
   ... checking whether 'walk' is an individual variable
Expression 'walk' can't be evaluated by i and g[b1/x].
'Undefined'
```

11.5.6 Evaluating Boolean Connectives

The `satisfy()` function assigns semantic values to complex expressions according to their syntactic structure, as determined by the method `decompose()`; this calls the parser from the `logic` module to return a 'normalized' parse structure for the expression. In the case of a Boolean connectives, `decompose()` produces a pair consisting of the connective and a list of arguments:

```
>>> m.decompose('((boy adam) and (dog fido))')
('and', ['(boy adam)', '(dog fido)'])
```

Following the functional style of interpretation, Boolean connectives are interpreted quite literally as truth functions; for example, the connective `and` can be interpreted as the function `AND`:

```
>>> AND = {True:  {True: True,
...                 False: False},
...        False: {True: False,
...                 False: False}}
```

We define `OPS` as a mapping between the Boolean connectives and their associated truth functions. Then the simplified clause for the satisfaction of Boolean formulas looks as follows:

```
>>> def satisfy(expr, g):
...   if parsed(expr) == (op, args):
...      if args == (phi, psi):
...         val1 = self.satisfy(phi, g)
...         val2 = self.satisfy(psi, g)
...         return OPS[op][val1][val2]
```

A formula such as `(and p q)` is interpreted by indexing the value of `and` with the values of the two propositional arguments, in the following manner:

```
>>> m.AND[m.evaluate('p', g)][m.evaluate('q', g)]
```

We can use these definitions to generate **truth tables** for the Boolean connectives:

```
>>> from nltk_lite.semantics import Model
>>> ops = ['and', 'or', 'implies', 'iff']
>>> pairs = [(p, q) for p in [True,False] for q in [True,False]]
>>> for o in ops:
...    print "%8s %8s |  p %s q" % ('p', 'q', o)
...    print "-" * 30
...    for (p, q) in pairs:
...        value = Model.OPS[o][p][q]
...        print "%8s %8s | %8s" % (p, q, value)
...    print
...
```

The output is as follows:

```
       p        q |  p and q
-----------------------------
    True     True |     True
    True    False |    False
   False     True |    False
   False    False |    False

       p        q |  p or q
-----------------------------
    True     True |     True
    True    False |     True
   False     True |     True
   False    False |    False

       p        q |  p implies q
-----------------------------
    True     True |     True
    True    False |    False
   False     True |     True
   False    False |     True

       p        q |  p iff q
-----------------------------
    True     True |     True
    True    False |    False
   False     True |    False
   False    False |     True
```

Although these interpretations are close to the informal understanding of the connectives, there are some differences. Thus, `'(p or q)'` is true even when both `'p'` and `'q'` are true. `'(p implies q)'` is true even when `'p'` is false; it only excludes the situation where *'p'* is true and `'q'` is false. `'(p iff q)'` is true if `'p'` and `'q'` have the same truth value, and false otherwise.

11.5.7 Evaluating Function Application

The `satisfy()` clause for function application is similar to that for the connectives. In order to handle type errors, application is delegated to a wrapper function `app()` rather than by directly indexing the curried characteristic function as described earlier. The definition of `satisfy()` started above continues as follows:

```
...  elif parsed(expr) == (fun, arg):
...      funval = self.satisfy(fun, g)
...      argval = self.satisfy(psi, g)
...      return app(funval, argval)
```

11.5.8 Evaluating Quantified Formulas

Let's consider now how to interpret quantified formulas, such as (22).

(22) `some x.(see x betty)`

We decompose (22) into two parts, the quantifier prefix `some x` and the body of the formula, (23).

(23) `(see x betty)`

Although the variable `x` in (22) is **bound** by the quantifier `some`, `x` is not bound by any quantifiers within (23); in other words, it is **free**. A formula containing at least one free variable is said to be **open**. How should open formulas be interpreted? We can think of `x` as being similar to a variable in Python, in the sense that we cannot evaluate an expression containing a variable unless it has already been assigned a value. As mentioned earlier, the task of assigning values to individual variables is undertaken by an `Assignment` object `g`. However, our variable assignments are partial: `g` may well not give a value to `x`.

```
>>> dom = val.domain
>>> g = Assignment(dom)
>>> m.evaluate('(see x betty)', g)
'Undefined'
```

We can use the `add()` method to explicitly add a binding to an assignment, and thereby ensure that `g` gives `x` a value.

```
>>> g.add('b1', 'x')
{'x': 'b1'}
>>> m.evaluate('(see x betty)', g)
True
```

In a case like this, we say that the entity `b1` **satisfies** the open formula `(see x betty)`, or that `(see x betty)` is **satisfied under the assignment** `g['b1'/'x']`.

When we interpret a quantified formula, we depend on the notion of an open subformula being satisfied under a variable assignment. However, to capture the force of the quantifier, we need to abstract away from arbitrary specific assignments. The first step is to define the set of **satisfiers** of a formula that is open in some variable. Formally, given an open formula φ[x] dependent on x and a model with domain D, we define the set *sat*(φ[x], g) of **satisfiers** of φ[x] to be:

(24) $\{u \in D \mid satisfy(\varphi[x], g[u/x]) = True\}$

We use $g[u/x]$ to mean that assignment which is just like g except that $g(x) = u$. Here is a Python definition of `satisfiers()`:

```
>>> def satisfiers(expr, var, g):
...     candidates = []
...     if freevar(var, expr):
...         for u in domain:
...             g.add(u, var)
...             if satisfy(expr, g):
...                 candidates.append(u)
...     return set(candidates)
```

The satisfiers of an arbitrary open formula can be inspected using the `satisfiers()` method.

```
>>> m.satisfiers('some y.((girl y) and (see x y))', 'x', g)
set(['b1'])
>>> m.satisfiers('some y.((girl y) and (see y x))', 'x', g)
set(['b1', 'b2'])
>>> m.satisfiers('(((girl x) and (boy x)) or (dog x))', 'x', g)
set(['d1'])
>>> m.satisfiers('((girl x) and ((boy x) or (dog x)))', 'x', g)
set([])
```

Now that we have put the notion of satisfiers in place, we can use this to determine a truth value for quantified expressions. An existentially quantified formula $\mathsf{x}.\varphi[\mathsf{x}]$ is held to be true if and only if $sat(\varphi[x], g)$ is nonempty. We use the length function `len()` to return the cardinality of a set.

```
...    elif parsed(expr) == (binder, body):
...       if binder = ('some', var):
...          sat = self.satisfiers(body, var, g)
...          return len(sat) > 0
```

In other words, a formula $\mathsf{x}.\varphi[\mathsf{x}]$ has the same value in model M as the statement that the number of satisfiers in M of $\varphi[\mathsf{x}]$ is greater than 0.

A universally quantified formula $\forall \mathsf{x}.\varphi[\mathsf{x}]$ is held to be true if and only if every u in the model's domain D belongs to $sat(\varphi[x], g)$; equivalently, if $\mathsf{D} \subseteq sat(\varphi[x], g)$. The `satisfy()` clause above for existentials can therefore be extended with the clause:

```
...      elif binder == ('all', var):
...          sat = self.satisfiers(body,var,g)
...          return domain.issubset(sat)
```

Although our approach to interpreting quantified formulas has the advantage of being transparent and conformant to classical logic, it is not computationally efficient. To verify an existentially quantified formula, it suffices to find just one satisfying individual and then return `True`. But the method just presented requires us to test satisfaction for every individual in the domain of discourse for each quantifier. This requires m^n evaluations, where m is the cardinality of the domain and n is the number of nested quantifiers.

11.5.9 Evaluating lambda abstracts

Finally, we can also evaluate λ-abstracts; not surprisingly, these are interpreted as `CharFuns`. To illustrate, we can construct the binary relation of individuals who see each other, or the ternary relation of distinct individuals a and b such for some c, a sees c and c sees b.

```
>>> m.evaluate(r'\x y. ((see x y) and (see y x))', g)
{'b1': {'g1': True}, 'g1': {'b1': True}}
>>> r = m.evaluate(r"""\x z y. (((see x z) and (see z y))
...                             and (not (x = y)))""", g)
>>> r.tuples()
set([('g2', 'b1', 'g1'), ('b2', 'g2', 'b1')])
```

Note that λ-abstracts can only be explicitly evaluated when the bound variable is an individual variable. Variables which range over functions, such as the `'P'` in `'\x.  (P suzie)'`, are called **higher-order** variables, and quantification over higher-order variables lies outside first-order logic.

If you attempt to evaluate an expression such as `'\x.  (P suzie)'`, the `semantics` package will raise an error. Since we only allow ourselves to quantify over individuals in FSRL, a variable assignment only give values to individual variables, and variable assignment is crucial for interpreting λ-abstraction. So though we do allow abstracts with higher-order variables in the language, they are not 'first-class citizens': they are only used as a stepping stone on the way to building up semantic representations in a compositional manner, and are eliminated prior to evaluation by β-reduction.

11.5.10 Exercises

1. ☼ Define a denotation for exclusive `or` (i.e., `'(p xor q)'` is equivalent to `'((p or q) and (not (p and q)))'`.)

2. ☼ Evaluate the expressions `'\x.(boy adam)'` and `'\x.(boy fido)'` in the model given above. Explain your results.

3. ◑ Use the `satisfiers()` method for determing the set of satisfiers of the open formula `'((dog x) implies (x = fido))'` in the model given above. Explain why the result is the way that it is.

4. ◑ Develop a set of around 10 sentences, using FSRL. Build a model for the sentences which makes then all true, and verify the results.

5. ◑ Build a model for a relation `rel` which is **transitively closed** and **reflexive**. That is, it satisfies the following two sentences:

   a.) `all x y z.(((rel y x) and (rel z y)) implies (rel z x))`

   b.) `all x.(rel x x)`

11.6 Quantifier Scope Revisited

You may recall that we discussed earlier an example of quantifier scope ambiguity, repeated here as (25).

(25) Every girl chases a dog.

The two readings are represented as follows.

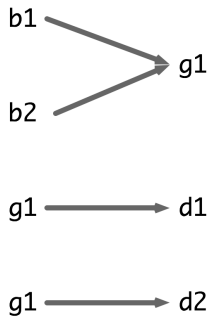
```
>>> sr1 = 'all x.((girl x) implies some z.((dog z) and (chase z x)))'
>>> sr2 = 'some z.((dog z) and all x.((girl x) implies (chase z x)))'
```

In order to examine the ambiguity more closely, let's fix our valuation as follows:

```
>>> val = Valuation()
>>> v = [('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')]))]
>>> val.read(v)
```

Using a slightly different graph from before, we can also visualise the chase relation as in (26).

(26)



In (26), 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 `sr1` above is true but `sr2` is false. One way of exploring these results is by using the `satisfiers()` method of `Model` objects.

```
>>> dom = val.domain
>>> m = Model(dom, val)
>>> g = Assignment(dom)
>>> fmla1 = '((girl x) implies some y.((dog y) and (chase y x)))'
>>> m.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 = '((dog y) and all x.((girl x) implies (chase y x)))'
>>> m.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.

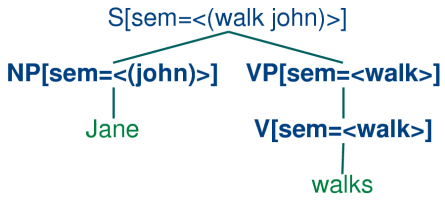
```
>>> fmla3 = '((girl y) and all x.((boy x) implies (chase y x)))'
>>> m.satisfiers(fmla3, 'y', g)
set(['g1'])
>>>
```

11.7 Evaluating English Sentences

11.7.1 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 9. 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. (27) illustrates a first approximation to the kind of analyses we would like to build.

(27)



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 (28) 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.)

(28)
```
S[sem = <app(?vp,?subj)>] -> NP[sem=?subj] VP[sem=?vp]
```

(28) 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. (28) 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 straightford; all we require are the rules shown in (29).

(29)
```
VP[sem=?v] -> IV[sem=?v]
NP[sem=<john>] -> '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 `nltk_lite.parse.featurechart`, and the trace in (30) shows how semantic values are derived by feature unification in the process of building a parse tree.

(30)
```
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 *
```

11.7.2 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 (31a) to be given a semantic representation like (31b). How can this be accomplished?

(31a) A dog barks.

(31b) `'some x.((dog x) and (bark x))'`

Let's make the assumption that our *only* operation for building complex semantic representations is `'app()'` (corresponding to 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 (31b)? As a first step, let's make the subject's `sem` value act as the functor rather than the argument in `'app()'`. Now we are looking for way of instantiating `?np` so that (32a) is equivalent to (32b).

(32a) [sem=<app(?np, bark)>]

(32b) [sem=<some x.((dog x) and (bark x))>]

This is where λ abstraction comes to the rescue; doesn't (32) 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 (31b). To do this, we replace the occurence of `'bark'` in (31b) by a variable `'P'`, and bind the variable with λ, as shown in (33).

(33) `'\P.some x.((dog x) and (P x))'`

As a point of interest, we have used a different style of variable in (33), 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 (33) 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 (34).

(34) `'\P.all x.((dog x) implies (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 (33) as a functor to `'bark'` gives us `'(\P.some x.((dog x) and (P x)) bark)'`, and carrying out β-reduction yields just what we wanted, namely (31b).

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.

```
>>> from nltk_lite.semantics import *
>>> grammar = GrammarFile.read_file('sem1.cfg')
>>> result  = text_interpret(['a dog barks'], grammar, beta_reduce=0)
>>> (syntree, semrep) = result['a dog barks'][0]
>>> print syntree
([INIT][]:
  (S[ sem = ApplicationExpression('(\Q P.some x.(and (Q x) (P x)) dog)', 'bark') ]:
    (NP[ sem = ApplicationExpression('\Q P.some x.(and (Q x) (P x))', 'dog') ]:
      (Det[ sem = LambdaExpression('Q', '\P.some x.(and (Q x) (P x))') ]: 'a')
      (N[ sem = VariableExpression('dog') ]: 'dog'))
    (VP[ sem = VariableExpression('bark') ]:
      (IV[ sem = VariableExpression('bark') ]: 'barks'))))
>>> print semrep
some x.(and (dog x) (bark x))
>>>
```

By default, the semantic representation that is produced by `text_interpret()` has already undergone β-reduction, but in the above example, we have overriden 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()
some x.(and (dog x) (bark x))
>>> print semrep.simplify().infixify()
some x.((dog x) and (bark x))
```

11.7.3 Transitive Verbs

Our next challenge is to deal with sentences containing transitive verbs, such as (35).

(35) Suzie chases a dog.

The output semantics that we want to build is shown in (36).

(36) `'some x.((dog x) and (chase x suzie))'`

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 (36) as our output while sticking to (33) 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.

(37) `'\y.some x.((dog x) and (chase x y))'`

Think of (37) 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 (33) so as to allow (37) to be derived.

Let's carry out a kind of inverse β-reduction on (37), giving rise to (38).

Let Then we are part way to the solution if we can derive (38), where `'X'` is applied to `'\z.(chase z y)'`.

(38) `'(\P.some x.((dog x) and (P x)) \z.(chase z y))'`

(38) may be slightly hard to read at first; you need to see that it involves applying the quantified NP representation from (33) to `'\z.(chase z y))'`. (38) is of course equivalent to (37).

Now let's replace the functor in (38) by a variable `'X'` of the same type as an NP; that is, of type ((**Ind** → **Bool**) → **Bool**).

(39) `'(X \z.(chase z y))'`

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 (39) and also the subject variable `'y'`. So the full solution is reached by giving *chases* the semantic representation shown in (40).

(40) `'\X y.(X \x.(chase x y))'`

If (40) is applied to (33), the result after β-reduction is equivalent to (37), which is what we wanted all along:

(41) `'(\X y.(X \x.(chase x y)) \P.some x.((dog x) and (P x))'`

`'(\y.(\P.some x.((dog x) and (P x)) \x.(chase x y))'`

`'\y.(some x.((dog x) and (chase x y)))'`

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 semanntically as individual constants, and these cannot be applied as functors to expressions like (37). 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. (42) shows the required λ expression for *Suzie*.

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

(42) 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 (40) 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 (14), repeated here, will always be translated as (44a), not (44b).

(43) Every girl chases a dog.

(44a) `'all x.((girl x) implies some y.  ((dog y) and (chase y x)))'`

(44b) `'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.

```
>>> from nltk_lite.semantics import *
>>> from model0.py import *
>>> grammar = GrammarFile.read_file('sem2.cfg')
>>> sent = 'every boy chases a girl in Noosa'
>>> result = text_evaluate([sent], grammar, m, g)
>>> for (syntree, semrep, value) in result[sent]:
...  print "'%s' is %s in Model m\n" % (semrep.infixify(), value)
...
'all x.((boy x) implies (some z4.((girl z4) and (chase z4 x)) and
```

```
(in noosa x)))' is True in Model m
...
'all x.((boy x) implies some z5.(((girl z5) and (in noosa z5)) and
(chase z5 x)))' is False in Model m
```

11.8 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 `chat80` module in `ntlk_lite.corpora` 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 (45).

(45) `'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 (45), 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:

(46) `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_lite.corpora import chat80
>>> chat80.items
['borders', 'contains', 'city', 'country', 'circle_of_lat',
'circle_of_long', 'continent', 'region', 'ocean', 'sea']
```

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 = chat80.make_valuation(concepts, read=True)
>>> val['city']['calcutta']
True
>>> val['country_of']['india']
{'hyderabad': True, 'delhi': True, 'bombay': True,
'madras': True, 'calcutta': True}
>>> from nltk_lite.semantics import *
>>> g = Assignment(dom)
>>> m = Model(dom, val)
>>> m.evaluate(r'\x . (population_of x jakarta)', 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.

11.9 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.

11.10 Exercises

1. ◑ Modify the `nltk_lite.semantics.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 `nltk_lite.semantics.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) and (Q x)'`, convert it to `'((Q x) and (P x)'` if the extension of `'Q'` is smaller than the extension of `'P'`.

11.11 Further Reading

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.

About this document...