12

Semantics and Predicate Logic

12.1 Introduction

Semantics deals with the meaning of words, phrases, and sentences. It is a wide and open subject intricately interwoven with the structure of the mind. The potential domain of semantics is immense and covers many of the human cognitive activities. It has naturally spurred a great number of theories. From the philosophers of ancient and medieval times, to logicians of the 19th century, psychologists and linguists of the 20th century, and now computer scientists, a huge effort has been made on this subject.

Semantics is quite subtle to handle or even to define comprehensively. It would be a reckless challenge to claim to introduce an exhaustive view of the topic. It would be even more difficult to build a unified point of view of all the concepts that are attached to it. In this chapter, we will cover formal semantics. This approach to semantics is based on logic and is the brainchild of both linguists and mathematicians. It addresses the representation of phrases and sentences, the definition of truth, the determination of reference (linking words to the world's entities), and some reasoning. In the next chapter, we will review lexical semantics.

12.2 Language Meaning and Logic: An Illustrative Example

Roughly defined, formal semantics techniques attempt to map sentences onto logical formulas. They cover areas of sentence representation, reference, and reasoning. Let us take an example to outline and illustrate layers involved in such a semantic processing. Let us suppose that we want to build a robot to serve us a dinner. To be really handy, we want to address and control our beast using natural language. So far, we need to implement a linguistic interface that will understand and process our orders and a mechanical device that will carry out the actions in the real world. Given the limits of this book, we set aside the mechanical topics and we concentrate on the linguistic part.

To avoid a complex description, we confine the scope of the robot's understanding to a couple of orders and questions. The robot will be able to bring meals to the table, to answer a few questions from the patrons, and to clear the table once the meals have been eaten. Now, let us imagine a quick dialogue between the two diners, Socrates and Pierre, and the robot (Table 12.1).

Table 12.1. A dialogue between diners and the robot.

Dialogue turns	**Sentences**
Socrates orders the dinner from the robot	*Bring the meal to the table*
The robot, after it has brought the meal, warns the diners	*The meal is on the table. It is hot*
Pierre, who was not listening	*Is this meal cold?*
...	*Miam miam*
Socrates, after the dinner is finished	*Clear the table*

Processing the sentences' meaning from a logical viewpoint requires a set of steps that we can organize in operating modules making parts of a semantic interpretation system. The final organization of the modules may vary, depending on the final application.

- The first part has to **represent** the state of the world. There is a table, diners around the table, a meal somewhere, and a robot. A condition to any further processing is to have them all in a knowledge base. We represent real entities, persons, and things using symbols that we store in a Prolog database. The database should reflect at any moment the current state of the world and the properties of the entities. That is, any change in the world should update the Prolog database correspondingly. When the robot mechanically modifies the world or when it asserts new properties on objects, a corresponding event has to appear in the database.
- The second part has to **translate** phrases or sentences such as *The robot brought the meal* or *the meal on the table* into formulas a computer can process. This also involves a representation. Let us consider the phrase *the meal on the table.* There are two objects, x and y, with x being a meal and y being a table. In addition, both objects are linked together by the relation that x is on y. A semantic module based on formal logic will translate such a phrase into a **logical form** compatible with the representation of objects into the database. This module has also to assert it into the database.
- A third part has to **reference** the logical forms to real objects represented in the database. Let us suppose that the robot asserts: *The meal is on the table. It is hot.* Referencing a word consists of associating it to an object from the real world (more accurately, to its corresponding symbol in the database). Referencing is sometimes ambiguous. There might be two meals: one being served and another one in the refrigerator. The referencing module must associate the word *meal* to the right object. In addition, referencing has also to keep track of entities men-

tioned in the discourse and to relate them. *It* in the second sentence refers to the same object as *the meal on the table* in the first sentence, and not to another meal in the refrigerator.

- A fourth part has to **reason** about the world and the sentences. Consider the utterance *The meal is on the table. Is it cold?* Is the latter assertion true? Is it false? To answer this question, the semantic interpreter must determine whether there is really a meal on the table and whether it is cold. To check it, the interpreter needs either to look up whether this fact is in the database or to have external devices such as a temperature sensor and a definition of cold. In addition, if a fact describes the meal as hot, a reasoning process must be able to tell us that if something is hot, it is not cold. We can implement such reasoning in Prolog using rules and an inference mechanism.

12.3 Formal Semantics

Of the many branches of semantics, formal semantics is one of the best-established in the linguistic community. The main assumption behind it is that logic can model language and, by extension, human thought. This has many practical consequences because, at hand, there is an impressive set of mathematical models and tools to exploit. The most numerous ones resort to the first-order predicate calculus. Such tools were built and refined throughout the 20th century by logicians such as Jacques Herbrand, Bertrand Russell, and Alfred Tarski.

The formal semantics approach is also based on assumptions linking a sentence to its semantic representation and most notably the principle of compositionality. This principle assumes that a sentence's meaning depends on the meaning of the phrases that compose it: "the meaning of the whole is a function of the meaning of its parts." A complementary – and maybe more disputable – assumption is that the phrases carrying meaning can be mapped onto syntactic units: the constituents. As a result, the principle of compositionality ties syntax and semantics together. Though there are many utterances in English, French, or German that are not compositional, these techniques have proved of interest in some applications.

12.4 First-Order Predicate Calculus to Represent the State of Affairs

The first concrete step of semantics is to represent the **state of affairs**: objects, animals, people, and observable facts together with properties of things and relations between them. A common way to do this is to use **predicate-argument structures**. The role of a semantic module will then be to map words, phrases, and sentences onto symbols and structures characterizing things or properties in a given context: the **universe of discourse**.

First-order predicate calculus (FOPC) is a convenient tool to represent things and relations. FOPC has been created by logicians and is a proven tool to express

and handle knowledge. It features constants, variables, and terms that correspond exactly to predicate-argument structures. We examine here these properties with Prolog, which is based on FOPC.

12.4.1 Variables and Constants

We can map things, either real or abstract, onto constants – or atoms – and subsequently identify the symbols to the things. Let us imagine a world consisting of a table and two chairs with two persons in it. This could be represented by five constants stored in a Prolog database. Then, the state of affairs is restrained to the database:

```
% The people:
  'Socrates'.
  'Pierre'.

% The chairs:
  chair1.    % chair #1
  chair2.    % chair #2

% The unique table:
  table1.    % table #1
```

A second kind of device, Prolog's variables such as `X`, `Y`, or `Z`, can unify with any entity of the universe and hold its value. And variable `X` can stand for any of the five constants.

12.4.2 Predicates

Predicates to Encode Properties. Predicates are symbols representing properties or relations. Predicates indicate, for instance, that `'Pierre'` has the property of being a person and that other things have the property of being objects. We state this simply using the `person` and `object` symbols as functors (predicate names) and `'Pierre'` and `table1` as their respective arguments. We add these facts to the Prolog database to reflect the state of the world:

```
person('Pierre').
person('Socrates').

object(table1).
object(chair1).
object(chair2).
```

We can be more specific and use other predicates describing that `table1` is a table, and that `chair1` and `chair2` are chairs. We assert this using the `table/1` and `chair/1` predicates:

```
table(table1).

chair(chair1).
chair(chair2).
```

Predicates to Encode Relations. Predicates can also describe relations between objects. Let us imagine that chair `chair1` is in front of table `table1`, and that `Pierre` is on `table1`. We can assert these relative positions using functors, such as `in_front_of/2` or `on/2`, linking respectively arguments `chair1` and `table1`, and `'Pierre'` and `table1`:

```
in_front_of(chair1, table1).

on('Pierre', table1).
```

So far, we have only used constants (atoms) as arguments in the properties and in the predicates representing them. If we want to describe more accurately three-dimensional scenes such as that in Fig. 12.1, we need more elaborate structures.

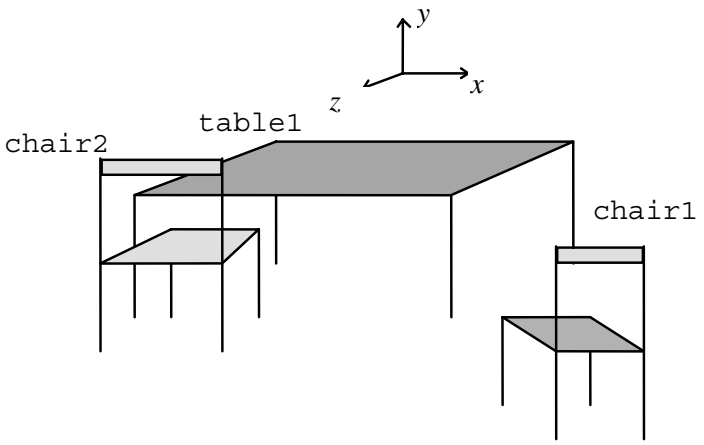

Fig. 12.1. A three-dimensional scene.

In such a scene, a coordinate system is necessary to locate precisely entities of the world. Since we are in a 3D space, 3D vectors give the position of objects that we can represent using the `v/3` predicate. `v(?x, ?y, ?z)` indicates the coordinate values of a point on x, y, and z axes. To locate objects, we will make use of `v/3`.

For sake of simplicity here, we approximate an object's position to its gravity center. We locate it with the `position/2` predicate. Position facts are compound terms that take the name of an object and the vector reflecting its gravity center as arguments:

```
position(table1, v(0, 0, 0)).
```

```
position(chair1, v(1, 1, 0)).
position(chair2, v(10, -10, 0)).
```

12.5 Querying the Universe of Discourse

Now, we have a database containing facts, i.e., properties and relations unconditionally true that describe the state of affairs. Using queries, the Prolog interpreter can check whether a fact is true or false:

```
?- table(chair1).
No

?- chair(chair2).
Yes
```

In addition, unification enables Prolog to determine subsets covering certain properties:

```
?- chair(X).
X = chair1;
X = chair2
```

We can get the whole subset in one shot using `bagof/3`. The alternate query yields:

```
?- bagof(X, chair(X), L).
L = [chair1, chair2]
```

The built-in `bagof/3` predicate has a cousin: `setof/3`. The difference is that `setof/3` sorts the elements of the answer and removes possible duplicates.

We may want to intersect properties and determine the set of the corresponding matching objects. Prolog can easily do this using conjunctions and shared variables. For instance, we may want to select from the set of chairs those that have the property of being in front of a table. The corresponding query is:

```
?- chair(X), in_front_of(X, Y), table(Y).
X = chair1, Y = table1
```

12.6 Mapping Phrases onto Logical Formulas

Using predicate-argument structures, we can map words, phrases, and sentences onto logical formulas. Simplifying a bit, nouns, adjectives, or verbs describe properties and relations that we can associate to predicates. Having said this, we have solved one part of the problem. We need also to determine the arguments that we will represent as logical variables.

Arguments refer to real-world entities, and the state of affair should define their value. We then need a second process to have a complete representation that will replace – unify – each variable with a logical constant. We will first concentrate on the representation of words or phrases and leave the arguments uninstantiated for now.

As a notation, we use λ-expressions that provide an abstraction of properties or relations. The λ symbol denotes variables that we can substitute with an entity of the real world, such as:

$$\lambda x.property(x)$$

or

$$\lambda y.\lambda x.property(x, y)$$

where λx indicates that we may supply an expression or a value for x.

Supplying such a value is called a β-reduction. It replaces all the occurrences of x in the expression and eliminates λx:

$$(\lambda x.property(x))entity\#1$$

yields

$$property(entity\#1)$$

λ is a right-associative operator that we cannot get with Western keyboards. We use the symbol `^` to denote it in Prolog. And `X^property(X)` is equivalent to $\lambda x.property(x)$.

12.6.1 Representing Nouns and Adjectives

Nouns or adjectives such as *waiter*, *patron*, *yellow*, or *hot* are properties that we map onto predicates of arity 1. For example, we represent the noun *chair* by:

$$\lambda x.chair(x)$$

whose equivalent notation in Prolog is `X^chair(X)`. Let us suppose that *chair1* is an entity in the state of affairs. We can supply it to this λ-expression:

$$(\lambda x.chair(x))chair1$$

and carry out a β-reduction that yields:

$$chair(chair1).$$

Table 12.2 shows some examples of representation of nouns and adjectives.

We can consider proper nouns as well as common nouns. In this case, we will have predicates such as `X^pierre(X)` and `X^socrates(X)`. This means that there are several Pierres and Socrates that can be unified with variable `X`. We can also make a nice distinction between them and treat proper nouns as constants like we have done before. In this case, there would be one single Pierre and one single Socrates in the world. Such a choice depends on the application.

Table 12.2. Representation of nouns and adjectives.

Lexical representations	Sentences	Semantic representations
Nouns		
`X^chair(X)`	*chair1 is a chair*	`chair(chair1)`
`X^patron(X)`	*Socrates is a patron*	`patron('Socrates')`
Adjectives		
`X^yellow(X)`	*table1 is yellow*	`yellow(table1)`
`X^hot(X)`	*meal2 is hot*	`hot(meal2)`

12.6.2 Representing Noun Groups

Noun groups may consist of a sequence of adjectives and a head noun. We form their semantic representation by combining each representation in a conjunction of properties (Table 12.3).

Table 12.3. Noun groups.

Noun groups	Semantic representation
hot meal	`X^(hot(X), meal(X))`
fast server	`X^(fast(X), server(X))`
yellow big table	`X^(yellow(X), big(X), table(X))`

The case is trickier when we have compounded nouns such as:

computer room
city restaurant
night flight

Noun compounds are notoriously ambiguous and require an additional interpretation. Some compounds should be considered as unique lexical entities such a *computer room*. Others can be rephrased with prepositions. *A city restaurant* is similar to *a restaurant in the city*. Others can be transformed using an adjective. A *night flight* could have the same interpretation as a *late flight*.

12.6.3 Representing Verbs and Prepositions

Verbs such as *run*, *bring*, or *serve* are relations. We map them onto predicates of arity 1 or 2, depending on whether they are intransitive or transitive, respectively (Table 12.4).

Prepositions usually link two noun groups, and like transitive verbs, we map them onto predicates of arity 2 (Table 12.5).

Table 12.4. Representation of verbs.

Lexical representations	Sentences	Sentence representations
Intransitive verbs		
`X^ran(X)`	*Pierre ran*	`ran('Pierre')`
`X^sleeping(X)`	*Socrates is sleeping*	`sleeping('Socrates')`
Transitive verbs		
`Y^X^brought(X,Y)`	*Roby served a meal*	`served('Roby', Z^meal(Z))`
`Y^X^served(X, Y)`	*Roby brought a plate*	`brought('Roby',Z^plate(Z))`

Table 12.5. Preposition representation.

Lexical representations	Phrases	Phrase representations
`Y^X^in(X, Y)`	*The fish in the plate*	`in(Z^fish(Z), T^plate(T))`
`Y^X^from(X,Y)`	*Pierre from Normandy*	`from('Pierre', 'Normandy')`
`Y^X^with(X,Y)`	*The table with a napkin*	`with(Z^table(Z), T^napkin(T))`

12.7 The Case of Determiners

12.7.1 Determiners and Logic Quantifiers

So far, we have dealt with adjectives, nouns, verbs, and prepositions, but we have not taken determiners into account. Yet, they are critical in certain sentences. Compare:

1. *A waiter ran*
2. *Every waiter ran*
3. *The waiter ran*

These three sentences have a completely different meaning, although they differ only by their determiners. The first sentence states that there is a waiter and that s/he ran. We can rephrase it as there is an x that has a conjunction of properties: $waiter(x)$ and $ran(x)$. The second sentence asserts that all x having the property $waiter(x)$ also have the property $ran(x)$.

Predicate logic uses two quantifiers to transcribe these statements into formulas:

- The existential quantifier, denoted $\exists$, and read *there exists*, and
- The universal quantifier, denoted $\forall$, and read *for all*

that we roughly associate to determiners *a* and *every*, respectively.

The definite determiner *the* refers to an object supposedly unique over the whole universe of discourse. We can connect it to the restricted existential quantifier denoted $\exists!$ and read *there exists exactly one*. *The waiter ran* should then be related to a unique waiter.

We can also use the definite article to designate a specific waiter even if there are two or more in the restaurant. Strictly speaking, *the* is ambiguous in this case

because it matches several waiters. *The* refers then to an object unique in the mind of the speaker as s/he mentions it, for instance, the waiter s/he can see at the very moment s/he is saying it or the waiter taking care of her/his table. The universe of discourse is then restricted to some pragmatic conditions. We should be aware that these conditions may bring ambiguity in the mind of the hearer – and maybe in that of the speaker.

12.7.2 Translating Sentences Using Quantifiers

Let us now consider determiners when translating sentences and let us introduce quantifiers. For that, we associate determiner *a* with quantifier $\exists$ and *every* with $\forall$. Then, we make the quantifier the head of a logical formula that consists either of a conjunction of predicates for determiner *a* or of an implication with *every*. The arguments are different depending on whether the verb is transitive or intransitive.

With intransitive verbs, the logical conjunctions or implications link the subject to the verb. Table 12.6 shows a summary of this with an alternate notation using Prolog terms. Predicates – principal functors – are then the quantifiers' names: `all/3`, `exists/3`, and `the/3`.

Table 12.6. Representation of sentences with intransitive verbs using determiners.

Sentences	Logical representations
A waiter ran	$\exists x(waiter(x) \wedge ran(x))$
	`exists(X, waiter(X), ran(X))`
Every waiter ran	$\forall x(waiter(x) \Rightarrow ran(x))$
	`all(X, waiter(X), ran(X))`
The waiter ran	$\exists! x(waiter(x) \wedge ran(x))$
	`the(X, waiter(X), ran(X))`

When sentences contain a transitive verb like:

A waiter brought a meal
Every waiter brought a meal
The waiter brought a meal

we must take the object into account. In the previous paragraph, we have represented subject noun phrases with a quantified logical statement. Processing the object is similar. In our examples, we map the object *a meal* onto the formula:

$$\exists y(meal(y))$$

Then, we link the object's variable y to the subject's variable x using the main verb as a relation predicate:

$$brought(x, y)$$

Finally, sentence *A waiter brought a meal* is represented by:

$$\exists x(waiter(x) \wedge \exists y(meal(y) \wedge brought(x, y)))$$

Table 12.7 recapitulates the representation of the examples.

Table 12.7. Logical representation of sentences with transitive verbs using determiners.

Sentences	Logical representation
A waiter brought a meal	$\exists x(waiter(x) \wedge \exists y(meal(y) \wedge brought(x, y)))$
	`exists(X, waiter(X),`
	`  exists(Y, meal(Y), brought(X, Y))`
Every waiter brought a meal	$\forall x(waiter(x) \Rightarrow \exists y(meal(y) \wedge brought(x, y)))$
	`all(X, waiter(X),`
	`  exists(Y, meal(Y), brought(X, Y))`
The waiter brought a meal	$\exists! x(waiter(x) \wedge \exists y(meal(y) \wedge brought(x, y)))$
	`the(X, waiter(X),`
	`  exists(Y, meal(Y), brought(X, Y))`

12.7.3 A General Representation of Sentences

The quantifiers we have used so far are the classical ones of logic. Yet, in addition to *a*, *every*, and *the*, there are other determiners such as numbers: *two*, *three*, *four*; indefinite adjectives: *several*, *many*, *few*; possessive pronouns: *my*, *your*; demonstratives: *this*, *that*; etc. These determiners have no exact counterpart in the world of logic quantifiers.

A more general representation uses determiners themselves as functors of Prolog terms instead of logic quantifier names. The subject noun phrase's determiner will be the principal functor of term mapping the whole sentence. Subsequent determiners will be the functors of inner terms. For example,

Two waiters brought our meals

is translated into

```
two(X, waiter(X), our(Y, meal(Y), brought(X, Y)))
```

Figure 12.2 depicts this term graphically.

Such a formalism can be extended to other types of sentences that involve more complex combinations of phrases (Colmerauer 1982). The basic idea remains the same: we map sentences and phrases onto trees – Prolog terms – whose functor names are phrases' determiners and whose arity is 3. Such terms are also called ternary trees. The top node of the tree corresponds to the sentence's first determiner (Fig. 12.3). The three arguments are:

- a variable that the determiner introduces into the semantic representation, say `X`

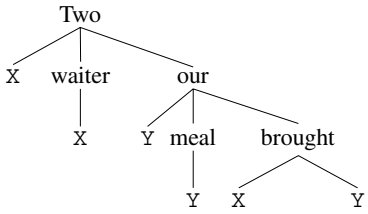

Fig. 12.2. Semantic representation of *Two waiters brought our meals*.

- the representation of the first noun phrase bound to the latter variable, that is `X` here
- the representation of the rest of the sentence, which we give the same recursive structure

As a result, a sentence is transformed into the Prolog predicate: `determiner(X, SemNP, SemRest)` (Fig. 12.3).

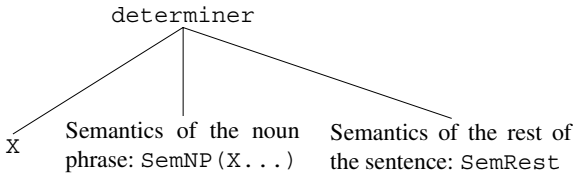

Fig. 12.3. Semantic representation using ternary trees.

This representation also enables us to process relative clauses and adjuncts. We represent them as a conjunction of properties. For example,

The waiter who has a cap

is translated into

```
the(X, (waiter(X), a(Y, cap(X), has(X, Y))), P)
```

where the second argument corresponds to the relative clause, the comma `(,)` between `waiter(X)` and `a(Y, cap(X), has(X, Y))` stands for a conjunction of these properties, and where `P` is linked with a possible rest of the sentence. If we complement this phrase with a verb phrase:

The waiter who has a cap brought a meal

we can give a value to `P` and the complete sentence representation will be (Fig. 12.4):

```
the(X,
    (waiter(X), a(Y, cap(X), has(X, Y))),
    a(Z, meal(Z), brought(X, Z))).
```

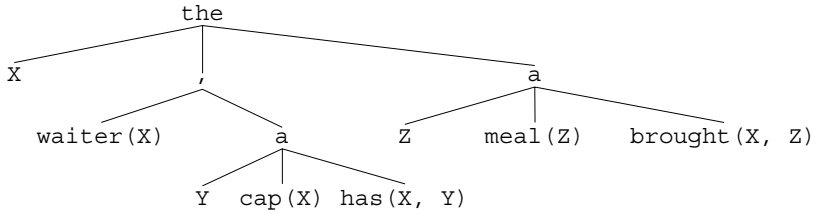

Fig. 12.4. Representation of *The waiter who has a cap brought a meal.*

12.8 Compositionality to Translate Phrases to Logical Forms

In Chap. 8, we used λ-calculus and compositionality to build a logical form out of a sentence. We will resort to these techniques again to incorporate the representation of determiners. Just like the case for nouns and verbs, we will process determiners using arguments in the DCG rules that will carry their partial semantic representation. The construction of the logical form will proceed incrementally using Prolog's unification while parsing the phrases and the sentence. The semantic composition of a sentence involves:

1. the translation of the first noun phrase – the subject
2. the translation of the verb phrase – the predicate – that contains a possible second noun phrase – the object

From the representation we provided in Chap. 8, the main change lies in the noun phrase translation. We approximated its semantics to the noun itself. Now we refine it into:

```
determiner(X, SemNP, SemRest).
```

12.8.1 Translating the Noun Phrase

To obtain `SemNP`, we have to compose the semantics of the determiner and of the noun, knowing that the noun's representation is:

```
noun(X^waiter(X)) --> [waiter].
```

Since the determiner must form the top node of the semantic tree, it has to embed an incomplete representation of the whole phrase. If we go back to the principles of λ-calculus, we know that the λ-variable indicated roughly that we request a missing value. In this case, the determiner needs the noun representation to reduce it. In consequence, variables in the noun phrase rule must be:

```
np(Sem) --> det((X^SemNP)^Sem), noun(X^SemNP).
```

We need to specify a variable `X` in the λ-expression of this rule, because it unifies with the `Sem` term, that is, with its first argument, as well as with `SemNP` and `SemRest`.

To write the determiner's lexical rule, we have now to proceed down into the structure details of `Sem`. The term `Sem` reflects a logical form of arity 3. It obtained its second argument `SemNP` from the subject – it did this in the `np` rule. It has to get its third argument, `SemRest`, from the verb and the object. `SemRest` will be built by the verb phrase `vp`, and since it is not complete at the moment, we denote it with a λ-expression. So, variables in the determiner rules are:

```
det((X^SemNP)^(Y^SemRest)^a(X, SemNP, SemRest)) -->
  [a].
```

Again, we must specify the `Y` variable that is to be bound in `SemRest`.

Using these rules, let us process *a waiter*. They yield the logical form:

```
(Y^SemRest)^a(X, waiter(X), SemRest)
```

whose λ-variable `(Y^SemRest)^` requests the semantic value of the verb phrase. The sentence rule `s` provides it and builds the complete representation, where `vp` brings `SemRest`:

```
s(Sem) --> np((Y^SemRest)^Sem), vp(SemRest).
```

12.8.2 Translating the Verb Phrase

Now, let the verb phrase rules compose the semantics of the rest (`SemRest`). The representation of the verbs remains unchanged. The verbs feature a single variable when intransitive, as in:

```
verb(X^rushed(X)) --> [rushed].
```

and two variables when transitive:

```
verb(Y^X^ordered(X, Y)) --> [ordered].
```

Verb phrase semantics is simple with an intransitive verb:

```
vp(X^SemRest) --> verb(X^SemRest).
```

It is a slightly more complicated when there is an object. As for the subject, the object's determiner embeds a ternary tree as a representation (Fig. 12.2). It introduces a new variable `Y` and contains a λ-expression that requests the representation of the verb. This λ-expression surfaces at the verb phrase level to bind the verb semantics to the third argument in the ternary tree. Let us name it `(Y^SemVerb)^`. It enables us to write the `vp` rule:

```
vp(X^SemRest) -->
  verb(Y^X^SemVerb),
  np((Y^SemVerb)^SemRest).
```

Finally, the whole program consists of these rules put together:

```
s(Sem) --> np((X^SemRest)^Sem), vp(X^SemRest).

np((X^SemRest)^Sem) -->
  determiner((X^SemNP)^(X^SemRest)^Sem),
  noun(X^SemNP).

vp(X^SemRest) --> verb(X^SemRest).
vp(X^SemRest) -->
  verb(Y^X^SemVerb),
  np((Y^SemVerb)^SemRest).
```

Let us also write a couple of vocabulary rules:

```
noun(X^waiter(X)) --> [waiter].
noun(X^patron(X)) --> [patron].
noun(X^meal(X)) --> [meal].

verb(X^rushed(X)) --> [rushed].
verb(Y^X^ordered(X, Y)) --> [ordered].
verb(Y^X^brought(X, Y)) --> [brought].

determiner((X^SemNP)^(X^SemRest)^a(X, SemNP,
SemRest)) --> [a].
determiner((X^SemNP)^(X^SemRest)^the(X, SemNP,
SemRest)) --> [the].
```

These rules applied to the sentence *The patron ordered a meal* yield:

```
?- s(Sem, [the, patron, ordered, a, meal], []).

Sem =
  the(_4,patron(_4),a(_32,meal(_32),ordered(_4,_32)))
```

where `_4` and `_32` are Prolog internal variables. Let us rename them `X` and `Y` to provide an easier and equivalent reading:

```
Sem = the(X, patron(X), a(Y, meal(Y), ordered(X, Y)))
```

Similarly, *the waiter rushed* produces

```
Sem = the(X, waiter(X), rushed(X))
```

12.9 Augmenting the Database and Answering Questions

Now that we have built a semantic representation of a sentence, what do we do with it? This has two answers, depending on whether it is a declaration or a question.

We must keep in mind that the state of affairs – here the Prolog database – reflects the total knowledge available to the interpretation system. If it is a declaration – a statement from the user – we must add something because it corresponds to new information. Conversely, if the user asks a question, we must query the database to find a response. In this section, we will review some straightforward techniques to implement it.

12.9.1 Declarations

When the user utters a declaration, the system must add its semantic representation to the description of the state of affairs. With a Prolog interpreter, the resulting semantic fact – corresponding, for example, to `determiner(X, NP, Rest)` – will have to be asserted to the database.

We can carry this out using one of the `asserta` or `assertz` predicates. The system builds the semantic representation while parsing and asserts the new fact when it has finished, that is, after the `sentence` rule. Since `asserta` is a Prolog predicate and we are using DCG rules, we enclose it within curly brackets (braces). The rule

```
sentence(Sem) -->
  np(...), vp(...), {asserta(Sem), ...}.
```

will result into a new `Sem` predicate asserted in the database once the sentence has been parsed.

12.9.2 Questions with Existential and Universal Quantifiers

In the case of a question, the parser must also build a representation. But the resulting semantic formula should be interpreted using inference rules that query the system to find an answer. Questions may receive *yes* or *no* as an answer. They may also provide the value of a fact from the database.

Yes/no questions generally correspond to sentences beginning with an auxiliary verb such as *do*, *is*, *have* in English, with *Est-ce que* in spoken French, and with a verb in German. Other types of questions begin with *wh*-words such as *what*, *who*, *which* in English, with *qu*-words in French such as *quel*, *qui*, with *w*-words in German such as *wer*, *wen*.

We must bring some modifications to the parser's rules to accept questions, although basically the sentence structure remains the same. Let us suppose that we deal with very simple *yes/no* questions beginning with auxiliary *do*. The rule structure after the auxiliary is that of a declaration. Once the question has been parsed, the system must "call" the semantic fact resulting from the parsing to answer it. We do this using the `call` predicate at the end of rules describing the `sentence` structure. The system will thus succeed and report a yes, or fail and report a no:

```
sentence(Sem) -->
  [do], np(...), vp(...), {call(Sem), ...}.
```

If the sentence contains determiners, the `Sem` fact will include them. Notably, the subject noun phrase's determiner will be the predicate functor: `determiner(X, Y, Z)`. For example,

Did a waiter rush

will produce `Sem = a(X, waiter(X), rushed(X))`

To `call` such predicates, we must write inference rules corresponding to the determiner values. The most general cases correspond to the logical quantifiers `exists`, which roughly maps *a*, *some*, *certain*, ..., and to the universal quantifier `all`.

Intuitively, a formula such as:

```
exists(X, waiter(X), rushed(X)),
```

corresponding to the sentence:

A waiter rushed

should be mapped onto to the query:

```
?- waiter(X), rushed(X).
```

and

```
a(X, waiter(X), a(Y, meal(Y), brought(X, Y)).
```

should lead to the recursive call:

```
?- waiter(X), a(Y, meal(Y), brought(X, Y)).
```

In consequence, `exists` can be written in Prolog as simply as:

```
exists(X, Property1, Property2) :-
  Property1,
  Property2,
  !.
```

We could have replaced `exists/3` with `a/3` or `some/3` as well.

The universal quantifier corresponds to logical forms such as:

```
all(X, waiter(X), rushed(X))
```

and

```
all(X, waiter(X), a(Y, meal(Y), brought(X, Y)).
```

We map these forms onto Prolog queries using a double negation, which produces equivalent statements. The first negation creates an existential quantifier corresponding to

There is a waiter who didn't rush
and
There is a waiter who didn't brought a meal

And the second one is interpreted as:

There is no waiter who didn't rush
and
There is no waiter who didn't brought a meal

Using the same process, we translate the double negation in Prolog by the rule:

```
all(X, Property1, Property2) :-
  \+ (Property1, \+ Property2),
  !.
```

We may use an extra call to `Property1` before the negation to ensure that there are *waiters*.

12.9.3 Prolog and Unknown Predicates

To handle questions, we want Prolog to retrieve the properties that are in the database and instantiate the corresponding variables. If no facts matching these properties have been asserted before, we want the predicate call to fail. With compiled Prologs, supporting ISO exception handling, a call will raise an exception if the predicate is not in the database. Fortunately, there are workarounds. If you want that the unknown predicates fail silently, just add:

```
:- unknown(_, fail).
```

in the beginning of your code.

If you know the predicate representing the property in advance, you may define it as `dynamic`:

```
:- dynamic(predicate/arity).
```

Finally, instead of calling the predicate using

```
Property
```

or

```
call(Property)
```

you can also use

```
catch(Property,
  error(existence_error(procedure, _Proc), _),
  fail)
```

which behaves like `call(Property)` except that if the predicate is undefined it will fail.

12.9.4 Other Determiners and Questions

Other rules corresponding to determiners such as *many*, *most*, and *more* are not so easy to write as the previous ones. They involve different translations depending on the context and application. The reader can examine some of them in the exercise list.

Questions beginning with *wh*-words are also more difficult to process. Sometimes, they can be treated in a way similar to yes/no questions. This is the case for *which* or *who*, which request the list of the possible solutions to predicate `exists`. Other *wh*-words, such as *where* or *when*, involve a deeper understanding of the context, possibly spatial or time reasoning. These cases are out of the scope of this book.

From this chapter, the reader should also be aware that the presentation has been simplified. In "real" natural language, many sentences are very difficult to translate. Notably, ambiguity is ubiquitous, even in benign sentences such as

Every caterpillar is eating a hedgehog

where two interpretations are possible.

Mapping an object must also take the context into account. If a patron says *This meal*, pointing to it with his/her finger, no ambiguity is possible. But, we then need a camera or tracking means to spot what is the user's gesture.

12.10 Application: The Spoken Language Translator

12.10.1 Translating Spoken Sentences

The Core Language Engine (CLE, Alshawi 1992) is a workbench aimed at processing natural languages such as English, Swedish, French, and Spanish. The CLE has a comprehensive set of modules to deal with morphology, syntax, and semantics. It provides a framework for mapping any kind of sentence onto logical forms. The CLE, which was designed at the Stanford Research Institute in Cambridge, England, is implemented in Prolog.

CLE has been used in applications, the most dramatic of which is definitely the Spoken Language Translator (SLT, Agnäs et al. 1994). This system translates spoken sentences from one language into another for language pairs such as English/Swedish, English/French, and Swedish/Danish.

Translation operates nearly in real time and has reached promising quality levels. Although SLT never went beyond the demonstration stage, it was reported that it could translate more than 70% of the sentences correctly for certain language pairs. Table 12.8 shows examples from English into French (Rayner and Carter 1995). SLT is limited to air travel information, but it is based on principles general enough to envision an extension to any other domain.

Table 12.8. Examples of French–English translations provided by the SLT. After Rayner and Carter (1995).

English	*What is the earliest flight from Boston to Atlanta?*
French	*Quel est le premier vol Boston–Atlanta?*
English	*Show me the round trip tickets from Baltimore to Atlanta*
French	*Indiquez-moi les billets aller-retour Baltimore–Atlanta*
English	*I would like to go about 9 am*
French	*Je voudrais aller aux environs de 9 heures*
English	*Show me the fares for Eastern Airlines flight one forty seven*
French	*Indiquez-moi les tarifs pour le vol Eastern Airlines cent quarante sept*

12.10.2 Compositional Semantics

The CLE's semantic component maps sentences onto logical forms. It uses unification and compositionality as a fundamental computation mechanism. This technique makes it easy to produce a representation while parsing and to generate the corresponding sentence in the target language.

Agnäs et al. (1994, pp. 42–43) give an example of the linguistic analysis of the sentence

I would like to book a late flight to Boston

whose semantic structure corresponds to the Prolog term:

```
would(like_to(i,
              book(i,
                   np_pp(a(late(flight)),
                         X^to(X, boston)))))
```

The parse rule notation is close to that of DCGs, but instead of the rule

```
Head --> Body_1, Body_2, ..., Body_n.
```

CLE uses the equivalent Prolog term

```
rule(<RuleId>,
  Head,
  [Body_1,
   Body_2,
    ...
   Body_n])
```

Table 12.9 shows the rules involved to parse this sentence. For example, rule 1 describes the sentence structure and is equivalent to

```
s --> np, vp.
```

Table 12.9. Rules in the CLE formalism. After Agnäs et al. (1994, p. 42).

#	Rules

1	`rule(s_np_vp,`
	`  s([sem=VP]),`
	`  [np([sem=NP,agr=Ag]),`
	`  vp([sem=VP,subjsem=NP,aspect=fin,agr=Ag])]).`

2	`rule(vp_v_np,`
	`  vp([sem=V,subjsem=Subj,aspect=Asp,agr=Ag]),`
	`  [v([sem=V,subjsem=Subj,aspect=Asp,agr=Ag,`
	`  subcat=[np([sem=NP])]]),`
	`  np([sem=NP,agr=_])]).`

3	`rule(vp_v_vp,`
	`  vp([sem=V,subjsem=Subj,aspect=Asp,agr=Ag]),`
	`  [v([sem=V,subjsem=Subj,aspect=Asp,agr=Ag,`
	`  subcat=[vp([sem=VP,subjsem=Subj])]]),`
	`  vp([sem=VP,subjsem=Subj,aspect=ini,agr=])]).`

4	`rule(vp_v_to_vp,`
	`  vp([sem=V,subjsem=Subj,aspect=Asp,agr=Ag]),`
	`  [v([sem=V,subjsem=Subj,aspect=Asp,agr=Ag,`
	`  subcat=[inf([]),vp([sem=VP,subjsem=Subj])]]),`
	`  inf([]),`
	`  vp([sem=VP,subjsem=Subj,aspect=inf,agr=])]).`

5	`rule(np_det_nbar,`
	`  np([sem=DET,agr=(3-Num)]),`
	`  [(det([sem=DET,nbarsem=NBAR,num=Num]),`
	`  nbar([sem=NBAR,num=Num])]).`

6	`rule(nbar_adj_nbar,`
	`  nbar([sem=ADJ,num=Num])`
	`  [adj([sem=ADJ,nbarsem=NBAR]),`
	`  nbar([sem=NBAR,num=Num])]).`

7	`rule(np_np_pp,`
	`  np([sem=np_pp(NP,PP),agr=Ag]),`
	`  [np([sem=NP,agr=Ag]),`
	`  pp([sem=PP])]).`

8	`rule(pp_prep_np,`
	`  pp([sem=PREP]),`
	`  [prep([sem=PREP,npsem=NP]),`
	`  np([sem=NP,agr=_])]).`

Rules embed variables with values under the form of pairs `Feature = Value` to implement syntactic constraints and semantic composition.

The lexicon entries follow a similar principle and map words onto Prolog terms:

```
lex(<Wordform>, <Category> (Features>))
```

Table 12.10 shows lexical entries of the sentence *I would like to book a late flight to Boston*, and Fig. 12.5. shows its parse tree

Table 12.10. Lexicon entries in the CLE formalism. After Agnäs et al. (1994, p. 42).

#	Lexicon entries
1	`lex(boston,np([sem=boston,agr=(3-s)])).`
2	`lex(i,np([sem,agr=(1-s)])).`
3	`lex(flight,n([sem=flight,num=s])).`
4	`lex(late,adj([sem=late(NBAR),nbarsem=NBAR])).`
5	`lex(a,det([sem=a(NBAR),nbarsem=NBAR,num=s])).`
6	`lex(to,prep([sem=X^to(X,NP),npsem=NP])).`
7	`lex(to,inf([])).`
8	`lex(book,v([sem=have(Subj,Obj),subjsem=Subj,aspect=ini, agr=_,subcat=[np([sem=Obj])]])).`
9	`lex(would,v([sem=would(VP),subjsem=Subj,aspect=fin, agr=_,subcat=[vp([sem=VP,aubjsem=Subj])]])).`
10	`lex(like,v([sem=like_to(Subj,VP),subjsem=Subj,aspect=ini, agr=_,subcat=[inf([]),vp([sem=VP,subjsem=Subj])]])).`

The semantic value of words or phrases is denoted with the `sem` constant in the rules. For instance, *flight* has the semantic value `flight` (Table 12.10, line 3) and *a* has the value `a(NBAR)` (Table 12.10, line 5), where `NBAR` is the semantic value of the adjective/noun sequence following the determiner.

The parser composes the semantic value of the noun phrase *a flight* applying the `np_det_nbar` rule (Table 12.9, line 5) equivalent to

```
np --> det, nbar.
```

in the DCG notation. It results in `sem = a(flight).`

All the semantic values are unified compositionally and concurrently with the parse in an upward movement, yielding the sentence's logical form.

12.10.3 Semantic Representation Transfer

The complete CLE's semantic layer relies on two stages. The first one maps a sentence onto a so-called quasi-logical form. Quasi-logical forms are basic predicate-argument structures, as we saw in this chapter, where variables representing real objects remain uninstantiated. The second layer links these variables to values, taking the context into account and so constructing fully resolved logical forms.

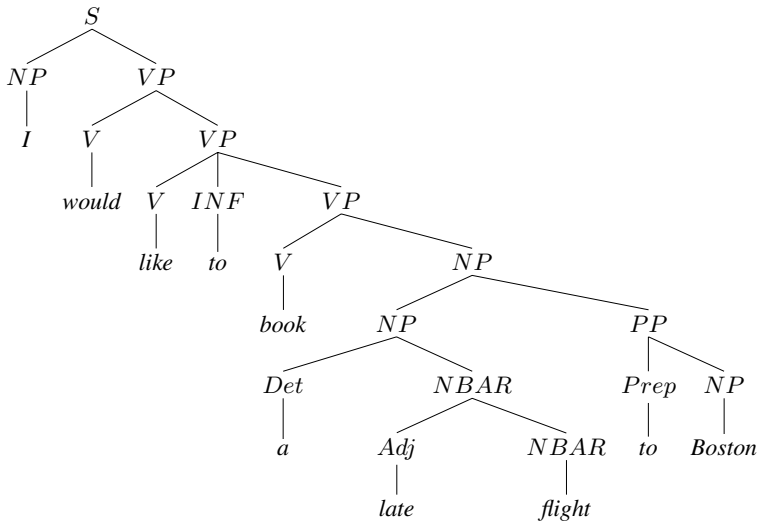

Fig. 12.5. Parse tree for *I would like to book a late flight to Boston*. After Agnäs et al. (1994, p. 43).

Translation from one language to another need not resolve variables. So the SLT builds a quasi-logical form from the source sentence and transfers it into the target language at the same representation level. SLT uses then a set of recursive transfer rules to match patterns in the source sentence and to replace them with their equivalent in the target language. Rules have the following format (Rayner et al. 1996):

```
trule(<Comment>
  <QLF pattern 1>
  <Operator>
  <QLF pattern 2>).
```

where `Operator` describes whether the rule is applicable from source to target (`>=`), the reverse (`=<`), or bidirectional (`==`).

Some rules are lexical, such as

```
trule([eng, fre],
      flight1 >= vol1).
```

which states that *flight* is translated as *vol*, but not the reverse. Others involve syntactic information such as:

```
trule([eng, fre],
      form(tr(relation,nn),
           tr(noun1),
           tr(noun2))
      >=
```

```
[and, tr(noun2),
      form(prep(tr(relation)),
           tr(noun1))]).
```

which transfers English compound nouns like *arrival time* – `noun1 noun2`. These nouns are rendered in French as: *heure d'arrivée* with a reversed noun order – `noun2 noun1` and with a preposition in-between *d'* – `prep(tr(relation))`.

12.11 Further Reading

Relations between logic and language have been a core concern for logicians, linguists, and philosophers. For a brief presentation and a critical discussion on philosophical issues, you may read Habermas (1988, Chap. 5). The reader can also find good and readable introductions in *Encyclopédie philosophique universelle* (Jacob 1989) and in Morton (2003).

Modern logic settings stem from foundational works of Herbrand (1930) and Tarski (1944). Later, Robinson (1965) proposed algorithms to implement logic programs. Robinson's work eventually gave birth to Prolog (Colmerauer 1970, 1978). Burke and Foxley (1996) provide a good introductory textbook on logic and notably on Herbrand bases. Sterling and Shapiro (1994) also give some insights on relations between Prolog and logic.

Some books attribute the compositionality principle to Frege (1892). In fact, Frege said exactly the opposite. The investigation of rational ways to map sentences onto logical formulas dates back to the ancient Greeks and the Middle Ages. Later, Montague (1974) extended this work and developed it systematically to English. Montague has had a considerable influence on modern developments of research in this area. For a short history of compositionality, see Godart-Wendling et al. (1998). The *Handbook of Logic and Language* (van Benthem and Ter Meulen (eds) 1997) provides a comprehensive treatment on current theories in the field. A shorter and very readable introduction on the philosophy of language is that of Taylor (1998).

Exercises

12.1. Write facts to represent
Tony is a hedgehog
A hedgehog likes caterpillars
Tony likes caterpillars
All hedgehogs likes caterpillars

12.2. Write DCG rules to get the semantic structure out of sentences of Exercise 12.1.

12.3. Write DCG rules to obtain the semantic representation of noun phrases made of one noun and one and more adjectives such as *The nice hedgehog*, *the nice little hedgehog*.

12.4. Write rules accepting sentences with embedded relative clauses, such as *The waiter that ran brought a meal* and producing a logical form out of them:
`the(X, (waiter(X), ran(X)), a(Y, meal(Y), brought(X, Y))`

12.5. Write rules to carry out the semantic interpretation of determiner *two*, as in the sentence *Two waiters rushed*.

12.6. Write rules to carry out the semantic interpretation of determiner *No*, as in *No waiter rushed.*

12.7. Write rules to carry out the semantic interpretation of *how many*, as in *how many waiters rushed.*

12.8. Write rules to parse questions beginning with relative pronouns *who* and *what* in sentences, such as *Who brought the meal?* and *What did the waiter bring?* and build logical forms out of them.

12.9. Write a small dialogue system accepting assertions and questions and answering them. A transcript of a session could be:
User: *the patron ordered the meal*
System: *OK*
User: *who brought the meal*
System: *I don't know*
User: *who ordered the meal*
System: *the patron*
User: *the waiter brought the meal*
System: *OK*
User: *who brought the meal*
System: *the waiter*

12.10. Some sentences such as *all the patrons ordered a meal* may have several readings. Cite two possible interpretations of this sentence and elaborate on them.