8 Natural Language Processing in Prolog

Chapter Objectives

Natural language processing representations were presented
- Semantic relationships
  - Conceptual graphs
  - Verb-based case frames

Prolog was used to build a series of parsers
- Context free parsers
  - Deterministic
- Probabilistic Parsers
  - Probabilistic measures for sentence structures and words
  - Lexicalized probabilistic parsers capture word combination plausibility
- Context sensitive parsers
  - Deterministic

Recursive descent semantic net parsers
- Enforce word-based case frame constraints

Chapter Contents

8.1 Natural Language Understanding in Prolog

Because of its declarative semantics, built-in search, and pattern matching, Prolog provides an important tool for programs that process natural language. Indeed, natural language understanding was one of Prolog's earliest applications. As we will see with many examples in this chapter, we can write natural language grammars directly in Prolog, for example, context-free, context-sensitive, recursive descent semantic network, as well as stochastic parsers. Semantic representations are also easy to create in Prolog, as we see for conceptual graphs and case frames in Section 8.2. Semantic relationships may be captured either using the first-order predicate calculus or by a meta-interpreter for another representation, as suggested by semantic networks (Section 2.4.1) or frames (Sections 2.4.2 and 8.1). This not only simplifies programming, but also keeps a close connection between theories and their implementation.

In Section 8.3 we present a context-free parser and later add context sensitivity to the parse Section 8.5. We accomplish many of the same justifications for context sensitivity in parsing, e.g., noun-verb agreement, with the various probabilistic parsers of Section 8.4. Finally, semantic inference, using graph techniques including `join`, `restrict`, and `inheritance` in conceptual graphs, can be done directly in Prolog as we see in Section 8.5.

Many of the approaches to parsing presented in this chapter have been suggested by several generations of colleagues and students.

8.2 Prolog-Based Semantic Representations

Following on the early work in AI developing representational schemes such as semantic networks, scripts, and frames (Luger 2009, Section 7.1) a number of *network languages* were developed to model the semantics of natural language and other domains. In this section, we examine a particular formalism to show how, in this situation, the problems of representing meaning were addressed. John Sowa's *conceptual graphs* (Sowa 1984) is an example of a network representation language. We briefly introduce conceptual graphs and show how they may be implemented in Prolog. A more complete introduction to this representational formalism may be found in Sowa (1984) and Luger (2009, Section 7.2).

A *conceptual graph* is a finite, connected, bipartite graph. The nodes of the graph are either *concepts* or *conceptual relations*. Conceptual graphs do not use labeled arcs; instead the conceptual relation nodes represent relations between concepts. Because conceptual graphs are bipartite, concepts only have arcs to relations, and vice versa. In Figure 8.1 `dog` and `brown` are concept nodes and `color` a conceptual relation. To distinguish these types of nodes, we represent concepts as boxes and conceptual relations as ellipses.

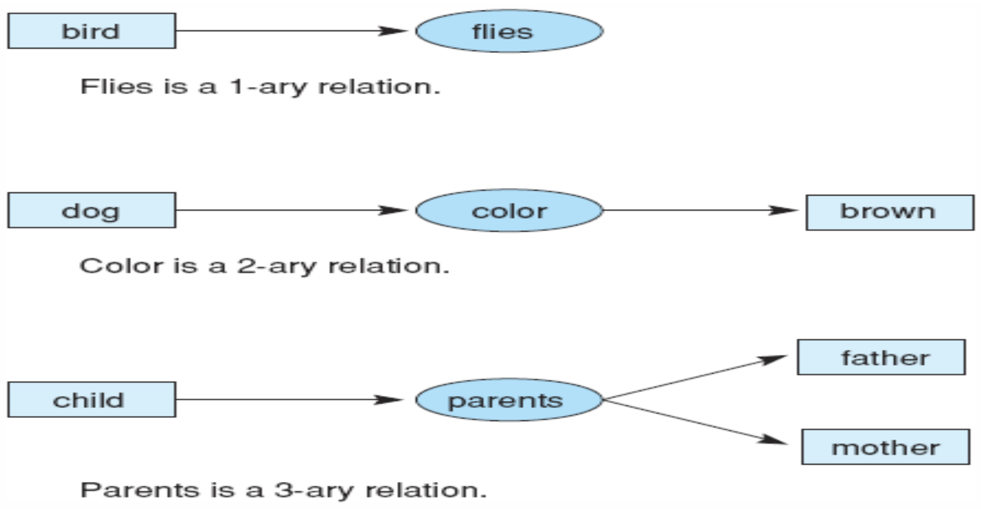
bird
flies
Flies is a 1-ary relation.
dog
color
brown
Color is a 2-ary relation.
child
parents
father
mother
Parents is a 3-ary relation.

Figure 8.1. Conceptual graph relations with different arities.

In conceptual graphs, concept nodes represent either concrete or abstract objects in the world of discourse. Concrete concepts, such as a cat, telephone, or restaurant, are characterized by our ability to form an image of them in our minds. Note that concrete concepts include generic concepts such as cat or restaurant along with concepts of specific cats and restaurants. We can still form an image of a generic cat. Abstract concepts include things such as love, beauty, and loyalty that do not correspond to images in our minds.

Conceptual relation nodes indicate a relation involving one or more concepts. One advantage of formulating conceptual graphs as bipartite graphs rather than using labeled arcs is that it simplifies the representation of relations of any number of arcs (arity). A relation of arity `n` is represented by a conceptual relation node having `n` arcs, as shown in Figure 8.1.

Each conceptual graph represents a single proposition. A typical knowledge base will contain a number of such graphs. Graphs may be arbitrarily complex but must be finite. For example, one graph in Figure 8.1 represents the proposition "A dog has a color of brown." Figure 8.2 is a graph of somewhat greater complexity that represents the sentence "Mary gave John the book." This graph uses conceptual relations to represent the cases of the verb "to give" and indicates the way in which conceptual graphs are used to model the semantics of natural language.

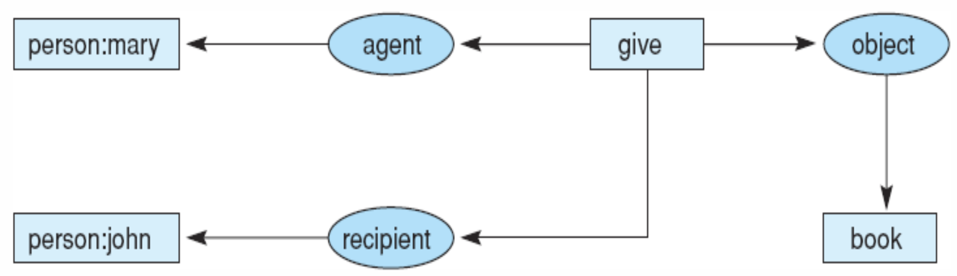

Figure 8.2. Conceptual graph of "Mary gave John the book."

Conceptual graphs can be translated directly into predicate calculus and hence into Prolog. The conceptual relation nodes become the predicate name, and the arity of the relation indicates the number of arguments of the predicate. Each Prolog predicate, as with each conceptual graph, represents a single proposition.

The conceptual graphs of Figure 8.1 may be rendered in Prolog as:

```
bird(X), flies(X).
dog(X), color (X, Y), brown(Y).
child(X), parents(X, Y, Z), father(Y), mother(Z).
```

where `X`, `Y`, and `Z` are bound to the appropriate individuals. Type information can be added to parameters as indicated in Section 5.2. We can also define the type hierarchy through a variation of `isa` predicates.

In addition to concepts, we define the relations to be used in conceptual graphs. For this example, we use the following concepts:

agent links an act with a concept of type animate. agent defines the relation between an action and the animate object causing the action.

experiencer links a state with a concept of type animate. It defines the relation between a mental state and its experiencer.

instrument links an act with an entity and defines the instrument used in an action.

object links an event or state with an entity and represents the verb–object relation.

part links concepts of type physobj and defines the relation between whole and part.

The verb plays a particularly important role in building an interpretation, as it defines the relationships between the subject, object, and other components of the sentence. We can represent each verb using a *case frame* that specifies:

The linguistic relationships (agent, object, instrument, and so on) appropriate to that particular verb. Transitive verbs, for example, can have a direct object; intransitive verbs do not.

Constraints on the values that may be assigned to any component of the case frame. For example, in the case frame for the verb bites, we have asserted that the agent of biting must be of the type dog. This causes "Man bites dog" to be rejected as semantically incorrect.

Default values on components of the case frame. In the "bites" frame, we have a default value of teeth for the concept linked to the instrument relation.

The case frames for the verbs like and bite appear in Figure 8.3.

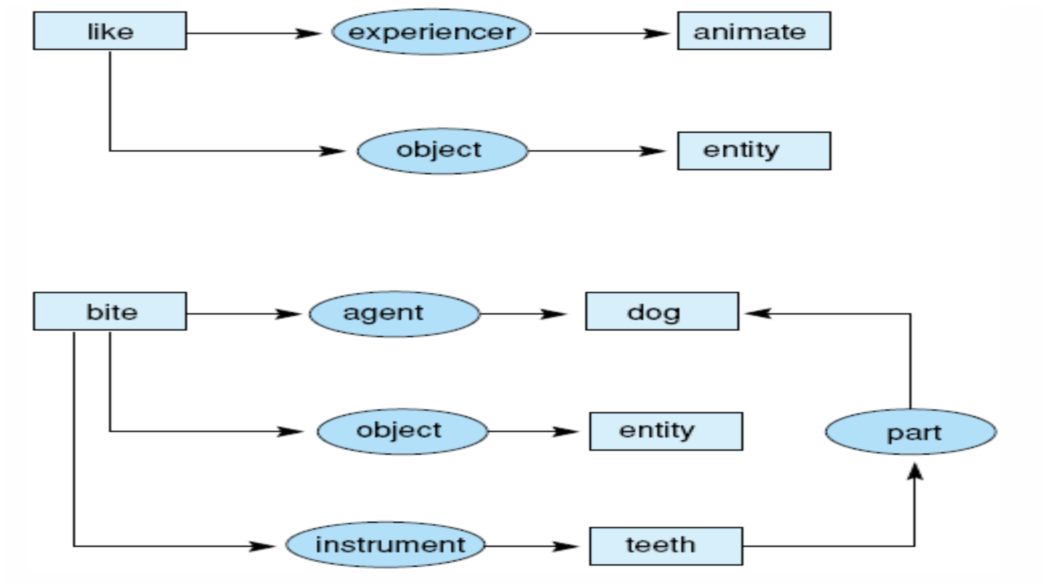

Figure 8.3. Case frames for the verbs "like" and "bite."

These verb-based case frames are also easily built in Prolog. Each verb is paired with a list of the semantic relations assumed to be part of the verb. These may include agents, instruments, and objects. We next offer examples of the verbs give and bite from Figure 8.3. For example, the verb give requires a subject, object, and indirect object. In the English sentence "John gives Mary the book," this structure takes on the obvious assignments. We can define defaults in a case frame by binding the appropriate variable values. For example, we could give bite a default instrument of teeth, and, indeed indicate that the instrument for biting, teeth, belong to the agent! Case frames for these two verbs might be:

```
verb(give,
       [human (Subject),
              agent (Subject, give),
              act_of_giving (give),
              object (Object, give),
              inanimate (Object),
              recipient (Ind_obj, give),
              human (Ind_obj) ] ).
verb(bite,
       [animate (Subject),
              agent (Subject, Action),
              act_of_biting (Action),
              object (Object, Action),
              animate (Object),
              instrument (teeth, Action),
              part_of (teeth, Subject) ] ).
```

Logic programming also offers a powerful medium for building grammars as well as representations for semantic meanings. We next build recursive descent parsers in Prolog, and then add syntactic and semantic constraints to these parsers.

8.3 A Context-Free Parser in Prolog

Consider the subset of English grammar rules below. These rules are "declarative" in the sense that they simply define relationships among parts of speech. With this subset of rules a large number of simple sentences can be judged as well formed or not. The "`<->`" indicate that the symbol on the left hand side can be replaced by the symbol or symbols on the right. For example, a `Sentence` can be replaced by a `NounPhrase` followed by a `VerbPhrase`.

```
Sentence <-> NounPhrase VerbPhrase
NounPhrase <-> Noun
NounPhrase <-> Article Noun
VerbPhrase <-> Verb
VerbPhrase <-> Verb NounPhrase
```

Adding some vocabulary to the grammar rules:

```
Article(a)
Article(the)
Noun(man)
Noun(dog)
Verb(likes)
Verb(bites)
```

These grammar rules have a natural fit to Prolog, for example, a `sentence` is a `nounphrase` followed by a `verbphrase`:

```
sentence(Start, End) :-
     nounphrase(Start, Rest), verbphrase(Rest, End).
```

This `sentence` Prolog rule takes two parameters, each a list; the first list, `Start`, is a sequence of words. The rule attempts to determine whether some initial part of this list is a `NounPhrase`. Any remaining tail of the `NounPhrase` list will match the second parameter and be passed to the first parameter of the `verbphrase` predicate. Any symbols that remain after the `verbphrase` check are passed back as the second argument of `sentence`. If the original list is a `sentence`, the second argument of `sentence` must be empty, `[ ]`.Two alternative Prolog descriptions of `nounphrase` and `verbphrase` parses follow.

Figure 8.4 is the parse tree of "the man bites the dog," with `and` constraints in the grammar reflected by `and` links in the tree.

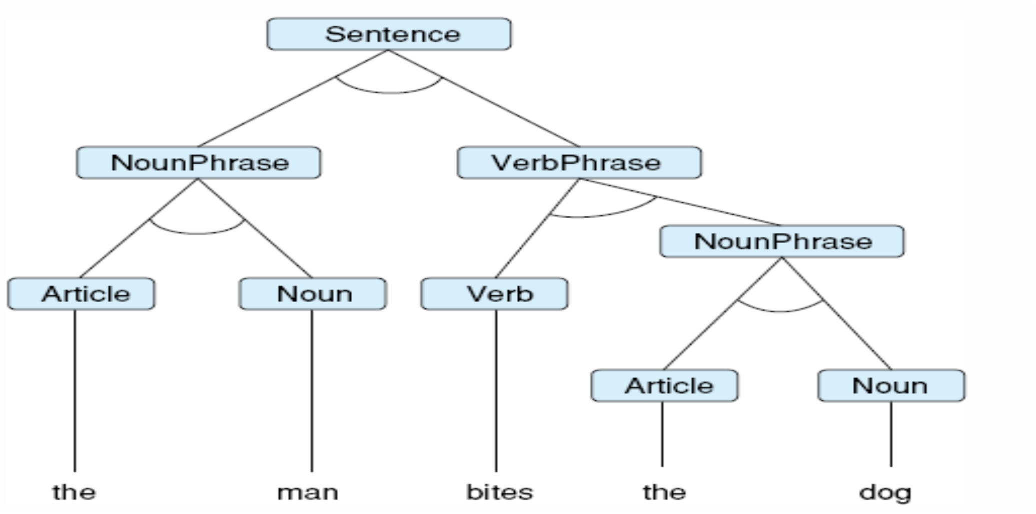

Figure 8.4. The and/or parse tree for "The man bites the dog."

To simplify our Prolog code, we present the sentence as a list: `[the, man, likes, the, dog]`. This list is broken up by, and passed to, the various grammar rules to be examined for syntactic correctness. Note how the "pattern matching" works on the list in question: pulling off the head, or the head and second element; passing on what is left over; and so on. The `utterance` predicate takes the list to be parsed as its argument and calls the `sentence` rule, initializing the second parameter of sentence to `[ ]`. The complete grammar is defined:

```
utterance(X) :- sentence(X, [ ]).
sentence(Start, End) :-
     nounphrase(Start, Rest), verbphrase(Rest, End).
nounphrase([Noun | End], End) :-
     noun(Noun).
nounphrase([Article, Noun | End], End) :-
     article(Article), noun(Noun).
verbphrase([Verb | End], End) :-
     verb(Verb).
```

```
verbphrase([Verb | Rest], End) :-
     verb(Verb), nounphrase(Rest, End).
article(a).
article(the).
noun(man).
noun(dog).
verb(likes).
verb(bites).
```

Example sentences may be tested for correctness:

```
?- utterance([the, man, bites, the, dog]).
Yes
?- utterance([the, man, bites, the]).
no
```

The interpreter can also fill in possible legitimate words to incomplete sentences:

```
?- utterance([the, man, likes, X]).
X = man
;
X = dog
;
no
```

Finally, the same code may be used to generate the set of all well-formed sentences using this limited dictionary and set of grammar rules:

```
?- utterance(X).
[man, likes]
;
[man, bites]
;
[man, likes, man]
;
[man, likes, dog]
etc.
```

If the user continues asking for more solutions, eventually all possible well-formed sentences that can be generated from the grammar rules and our vocabulary are returned as values for X. Note that the Prolog search is left-to-right and depth-first.

The grammar rules specify a subset of legitimate sentences of English. The Prolog grammar code represents these specifications. The interpreter is asked questions about them and the answer is a function of the specifications and the question asked. Since there are no constraints enforced across the subtrees that make up the full parse of a sentence, see Figure 8.4, the parser/generator for this grammar is said to be *context free*. In Section 8.3 we use probabilistic measures to add constraints both to particular word combinations and to the structures of the grammar.

8.4 Probabilistic Parsers in Prolog

In this section we extend the context-free grammar of Section 8.2 to include further syntactic and semantic constraints. For example, we may want some grammatical structures to be less likely than others, such as a noun by itself being less likely than an article followed by a noun. Further, we may want the sentence "The dog bites the widget" to be less likely than the sentence "The dog bites the man." Finally, if our vocabulary includes the verb like (as well as likes), we want "The man likes the dog" to be acceptable, but "The man like the dog" to fail. The parsers for Sections 8.3.1 and 8.3.2 were suggested by Professor Mark Steedman of the University of Edinburgh and transformed to the syntax of this book by Dr. Monique Morin of the University of New Mexico.

We next create two probabilistic parsers in Prolog, first a context free parser and second, a lexicalized context free parser.

Probabilistic Context-Free Parser

Our first extension is to build a *probabilistic context-free parser*. To do this, we add a probabilistic parameter, `Prob`, to each grammar rule. Note that the probability that a sentence will be a noun phrase followed by a verb phrase is 1.0, while the probability that a noun phrase is simply a noun is less than the probability of it being an article followed by a noun. These probabilities are reflected in `pr` facts that are related to each grammar rule, `r1`, `r2`, …, `r5`.

The full probability of a particular sentence, `Prob`, however, is calculated by combining a number of probabilities: that of the rule itself together with the probabilities of each of its constituents. Thus, the full probability `Prob` of `r1` is a product of the probabilities that a particular noun phrase is combined with a particular verb phrase. Further, the probability for the third rule, `r3`, will be the product of that type noun phrase occurring (`r3`) times the probabilities of the particular article and noun that make up the noun phrase. These noun/article probabilities are given in the two argument dictionary "fact" predicates. These probabilities for particular words might be determined by sampling some corpus of collected sentences. In the examples that follow we simply made-up these probabilistic measures.

```
utterance(Prob, X) :- sentence(Prob, X, [ ]).
sentence(Prob, Start, End) :-
     nounphrase(P1, Start, Rest),
     verbphrase(P2, Rest, End),
     pr(r1, P), Prob is P*P1*P2.
nounphrase(Prob, [Noun | End], End) :-
     noun(P1, Noun), pr(r2, P), Prob is P*P1.
nounphrase(Prob, [Article, Noun | End], End) :-
     article(P1, Article), noun(P2,Noun), pr(r3, P),
     Prob is P*P1*P2.
verbphrase(Prob, [Verb | End], End) :-
     verb(P1, Verb), pr(r4, P), Prob is P*P1.
```

```
verbphrase(Prob, [Verb | Rest], End) :-
     verb(P1, Verb),
     nounphrase(P2, Rest, End), pr(r5, P),
     Prob is P*P1*P2.
pr(r1, 1.0).
pr(r2, 0.3).
pr(r3, 0.7).
pr(r4, 0.2).
pr(r5, 0.8).
article(0.25, a).
article(0.75, the).
noun(0.65, man).
noun(0.35, dog).
verb(0.9, likes).
verb(0.1, bites).
```

We now run several example sentences as well as offer general patterns of sentences, i.e., sentences beginning with specific patterns of words such as "The dog bites…" Finally, we ask for all possible sentences that can be generated under these constraints.

```
?- utterance(Prob, [the, man, likes, the, dog]).
Prob = 0.0451474
Yes
?- utterance(Prob, [bites, dog])
No
?- utterance(Prob, [the, man, dog]).
No
?- utterance(Prob, [the, dog, bites, X]).
Prob = 0.0028665
X = man
;
Prob = 0.0015435
X = dog
;
No
?- utterance(Prob, [the, dog, bites, XY]).
Prob = 0.0028665
X = man
Y = [ ]
;
Prob = 0.0015435
X = dog
Y = [ ]
;
```

```
Prob = 0.00167212
X = a
Y = [man] ;
etc.
?- utterance(Prob, X).
Prob = 0.0351
X = [man, likes]
;
Prob = 0.0039
X = [man, bites]
;
Prob = 0.027378
X = [man, likes, man]
;
Prob = 0.014742
X = [man, likes, dog]
```

etc.

A Probabilistic Lexicalized Context Free Parser

We next demonstrate a *probabilistic lexicalized context-free parser*. This is a much more constrained system in which the probabilities, besides giving measures for the various grammatical structures and individual words as in the previous section, also describe the possible combinations of words (thus, it is a probabilistic *lexicalized* parser). For example, we now measure the likelihood of both noun-verb and verb-object word combinations. Constraining noun-verb combinations gives us much of the power of the context-sensitive parsing that we see next in Section 8.4, where noun-verb agreement is enforced by the constraints across the subtrees of the parse.

There are a number of goals here, including "measuring" the "quality" of utterances in the language by determining a probabilistic measure for their occurring. Thus, we can determine that a possible sentence fails for syntactic or semantic reasons by seeing that it produces a very low or zero probability measure, rather than by the interpreter simply saying "no."

In the following grammar we have hard coded the probabilities of various structure and word combinations. In a real system, lexical information could be better obtained by sampling appropriate corpora with noun-verb or verb-object *bigrams*. We discuss the *n-gram* approach to language analysis in Luger (2009, Section 15.4) where the probability of word combinations was described (two words—*bigrams*, three words—*trigrams*, etc.). These probabilities are usually determined by sampling over a large collection of sentences, called a *corpus*. The result was the ability to assess the likelihood of these word combinations, e.g., to determine the probability of the verb "bite" following the noun "dogs."

In the following examples the `Prob` value is made up of the probabilities of the particular sentence structure, the probabilities of the verb-noun and verb-object combinations, and the probabilities of individual words.

```
utterance(Prob, X) :-
     sentence(Prob, Verb, Noun, X, [ ]).
sentence(Prob, Verb, Noun, Start, End) :-
     nounphrase(P1, Noun, Start, Rest),
     verbphrase(P2, Verb, Rest, End),
     pr(r1, P),      % Probability of this structure
     pr([r1, Verb, Noun], PrDep),
          % Probability of this noun/verb combo
     pr(shead, Verb, Pshead),
          % Probability this verb heads the sentence
     Prob is Pshead*P*PrDep*P1*P2.
nounphrase(Prob, Noun, [Noun | End], End) :-
     noun(P1, Noun), pr(r2, P), Prob is P*P1.
nounphrase(Prob, Noun, [Article,Noun | End], End) :-
     article(P1, Article), noun(P2,Noun), pr(r3, P),
     pr([r3, Noun, Article], PrDep),
          % Probability of art/noun combo
     Prob is P*PrDep*P1*P2.
verbphrase(Prob, Verb, [Verb | End], End) :-
     verb(P1, Verb), pr(r4, P), Prob is P*P1.
verbphrase(Prob, Verb, [Verb,Object | Rest], End) :-
     verb(P1, Verb), nounphrase(P2, Object,
          Rest, End).
     pr([r5, Verb, Object], PrDep),
          % Probability of verb/object combo
     pr(r5, P), Prob is P*PrDep*P1*P2.
pr(r1, 1.0).
pr(r2, 0.3).
pr(r3, 0.7).
pr(r4, 0.2).
pr(r5, 0.8).
article(1.0, a).
article(1.0, the).
article(1.0, these).
noun(1.0, man).
noun(1.0, dogs).
verb(1.0, likes).
verb(1.0, bite).
pr(shead, likes, 0.5).
pr(shead, bite, 0.5).
pr([r1, likes, man], 1.0).
pr([r1, likes, dogs], 0.0).
pr([r1, bite, man], 0.0).
pr([r1, bite, dogs], 1.0).
pr([r3, man, a], 0.5).
pr([r3, man, the], 0.5).
pr([r3, man, these], 0.0).
pr([r3, dogs, a], 0.0).
pr([r3, dogs, the], 0.6).
pr([r3, dogs, these], 0.4).
pr([r5, likes, man], 0.2).
pr([r5, likes, dogs], 0.8).
pr([r5, bite, man], 0.8).
pr([r5, bite, dogs], 0.2).
```

The `Prob` measure gives the likelihood of the utterance; words that aren't sentences return `No`.

```
?- utterance(Prob, [a, man, likes, these, dogs]).
Prob = 0.03136
?- utterance(Prob, [a, man, likes, a, man]).
Prob = 0.0098
?- utterance(Prob, [a, man, likes, a, man]).
Prob = 0.0098
?- utterance(Prob, [the, dogs, likes, these, man]).
Prob = 0
?- utterance(Prob, [the, dogs]).
No
?- utterance(Prob, [the, dogs, X | Y])
Prob = 0
X = likes Y = []
;
Prob = 0.042
X = bite Y = []
;
Prob = 0
X = likes Y = [man]
;
Prob = 0.04032
X = bite Y = [man]
;
Prob = 0.01008
X = bite Y = [dogs]
;
Prob = 0.04704
X = bite Y = [a, man]
Etc
?- utterance(Prob, X).
```

```
Prob = 0.03
X = [man, likes]
;
Prob = 0
X = [man, bite]
;
Prob = 0.0072
X = [man, likes, man]
;
Prob = 0.0288
X = [man, likes, dogs]
;
Prob = 0.0084
X = [man, likes, a, man]
etc
```

We next enforce many of the same syntax/semantic relationships seen in this section by imposing constraints (context sensitivity) across the subtrees of the parse. Context sensitivity can be used to constrain subtrees to support relationships within a sentence such as article-noun and noun-verb number agreement.

8.5 A Context-Sensitive Parser in Prolog

A *context-sensitive* parser addresses the issues of the previous section in a different manner. Suppose we desire to have proper noun–verb agreement enforced by the grammar rules themselves. In the dictionary entry for each word its singular or plural form can be noted as such. Then in the grammar specifications for `nounphrase` and `verbphrase` a further parameter is used to signify the `Number` of each phrase. This enforces the constraint that a singular noun has to be associated with a singular verb. Similar constraints for article–noun combinations can also be enforced. The technique we are using is constraining sentence components by enforcing variable bindings across the subtrees of the parse of the sentence (note the and links in the parse tree of Figure 8.4).

Context sensitivity increases the power of a context-free grammar considerably. These additions are made by directly extending the Prolog code of Section 8.2:

```
utterance(X) :- sentence(X, [ ]).
sentence(Start, End) :-
     nounphrase(Start, Rest, Number),
     verbphrase(Rest, End, Number).
nounphrase([Noun | End], End, Number) :-
     noun(Noun, Number).
nounphrase([Article, Noun | End], End, Number) :-
     noun(Noun, Number), article(Article, Number).
```

```
verbphrase([Verb | End], End, Number) :-
     verb(Verb, Number).
verbphrase([Verb | Rest], End, Number) :-
     verb(Verb, Number), nounphrase(Rest, End, _).
article(a, singular).
article(these, plural).
article(the, singular).
article(the, plural).
noun(man, singular).
noun(men, plural).
noun(dog, singular).
noun(dogs, plural).
verb(likes, singular).
verb(like, plural).
verb(bites, singular).
verb(bite, plural).
```

We next test some sentences. The answer to the second query is `no`, because the subject (`men`) and the verb (`likes`) do not agree in number.

```
?- utterance([the, men, like, the, dog]).
Yes
?- utterance([the, men, likes, the, dog]).
no
```

If we enter the following goal, `X` returns all verb phrases that complete the plural "`the men` …" with all verb phrases with noun–verb number agreement. The final query returns all sentences with article–noun as well as noun–verb agreement.

```
?- utterance([the, men  X]).
?- utterance(X).
```

In the context-sensitive example we use the parameters of dictionary entries to introduce more information on the meanings of each of the words that make up the sentence. This approach may be generalized to a powerful parser for natural language. More and more information may be included in the dictionary of the word components used in the sentences, implementing a knowledge base of the meaning of English words. For example, men are animate and human. Similarly, dogs may be described as animate and nonhuman. With these descriptions new rules may be added for parsing, such as "humans do not bite animate nonhumans" to eliminate sentences such as [the, man, bites, the, dog]. We add these constraints in the following section.

8.6 A Recursive Descent Semantic Net Parser

We next extend the set of context-sensitive grammar rules to include some possibilities of semantic consistency. We do this by matching case frames, Section 8.1, for the verbs of sentences to semantic descriptions of subjects and objects. After each match, we constrain these semantic net subgraphs to be consistent with each other. We do this by performing graph operations, such as `join` and `restrict`, to each piece of the graph as it is returned up the parse tree.

We first present the grammar rules where the top-level `utterance`, returns not just a sentence but also a `Sentence_graph`. Each component of the grammar, e.g., `nounphrase` and `verbphrase`, call `join` to merge together the constraints of their respective graphs.

```
utterance(X, Sentence_graph) :-
     sentence(X, [ ], Sentence_graph).
sentence(Start, End, Sentence_graph) :-
     nounphrase(Start, Rest, Subject_graph),
     verbphrase(Rest, End, Predicate_graph),
     join([agent(Subject_graph)], Predicate_graph,
          Sentence_graph).
nounphrase([Noun | End], End, Noun_phrase_graph) :-
     noun(Noun, Noun_phrase_graph).
nounphrase([Article, Noun | End], End,
          Noun_phrase_graph) :-
     article(Article),
     noun(Noun, Noun_phrase_graph).
verbphrase([Verb | End], End, Verb_phrase_graph) :-
     verb(Verb, Verb_phrase_graph).
verbphrase([Verb | Rest], End, Verb_phrase_graph) :-
     verb(Verb, Verb_graph),
     nounphrase(Rest, End, Noun_phrase_graph),
     join([object(Noun_phrase_graph)], Verb_graph,
          Verb_phrase_graph).
```

We next present the graph `join` and `restriction` operations. These are meta-predicates since their domain is other Prolog structures. These utilities propagate constraints across pieces of semantic nets they combine.

```
join(X, X, X).
join(A, B, C) :-
     isframe(A), isframe(B), !,
     join_frames(A, B, C, not_joined).
join(A, B, C) :-
     isframe(A), is_slot((B), !,
     join_slot_to_frame(B, A, C).
join(A, B, C) :-
     isframe(B), is_slot(A), !,
     join_slot_to_frame(A, B, C).
join(A, B, C) :-
     is_slot(A), is_slot(B), !,
     join_slots(A, B, C).
```

`join_frames` recursively matches each slot (property) of the first frame to matching slots of the second frame. `join_slot_to_frame` takes a slot and a frame and searches the frame for matching slots. `join_slots,` once slots are matched, unites the two slots, taking the type hierarchy into account:

```
join_frames([A | B], C, D, OK) :-
     join_slot_to_frame(A, C, E) , !,
     join_frames(B, E, D, ok).
join_frames([ A | B], C, [A | D], OK) :-
     join_frames(B, C, D, OK), !.
join_frames([], A, A, ok).
join_slot_to_frame(A, [B | C], [D | C]) :-
     join_slots(A, B, D).
join_slot_to_frame(A, [B | C], [B | D]) :-
     join_slot_to_frame(A, C, D).
join_slots(A, B, D) :-
     functor(A, FA, _), functor(B, FB, _),
     match_with_inheritance(FA, FB, FN),
     arg(1, A, Value_a), arg(1, B, Value_b),
     join(Value_a, Value_b, New_value),
     D =.. [FN | [New_value]].
isframe([_ | _]).
isframe([ ]).
is_slot(A) :- functor(A, _, 1).
```

Finally, we create dictionary entries, the inheritance hierarchy, and verb case frames. In this example, we use a simple hierarchy that lists all valid specializations; the third argument to `match_with_inheritance` is the common specialization of the first two. A more realistic approach might maintain a graph of the hierarchies and search it for common specializations. Implementation of this is left as an exercise.

```
match_with_inheritance(X, X, X).
match_with_inheritance(dog, animate, dog).
match_with_inheritance(animate, dog, dog).
match_with_inheritance(man, animate, man).
match_with_inheritance(animate, man, man).
article(a).
article(the).
noun(fido, [dog(fido)]).
noun(man, [man(X)]).
noun(dog, [dog(X)]).
verb(likes, [action([liking(X)]),
        agent([animate(X)]), object(animate(Y)])]).
verb(bites, [action([biting(Y)]),
        agent([dog(X)]), object(animate(Z)])]).
```

We now parse several sentences and print out their `Sentence_graph`:

```
?- utterance([the, man, likes, the, dog], X).
X = [action([liking(_54)]), agent([man(_23)]),
        object([dog(_52)])].
?- utterance([fido, likes, the, man], X).
X = [action([liking(_62)]), agent([dog(fido)]),
        object([man(_70)])].
?- utterance([the, man, bites, fido], Z).
no
```

The first sentence states that some man, with name unknown, likes an unnamed dog. The last sentence, although it was syntactically correct, did not meet the semantic constraints, where a dog had to be the agent of bites. In the second sentence, a particular dog, Fido, likes an unnamed man. Next we ask whether Fido can bite an unnamed man:

```
?- utterance([fido, bites, the, man], X).
X = [action([biting(_12)]), agent([dog(fido)]),
        object([man(_17)])].
```

This parser may be extended in many interesting directions, for instance, by adding adjectives, adverbs, and prepositional phrases, or by allowing compound sentences. These additions must be both matched and constrained as they are merged into the sentence graph for the full sentence. Each dictionary item may also have multiple meanings that are only accepted as they meet the general requirements of the sentence. In the next chapter we present the Earley parser for language structures.

Exercises

1. Create a predicate calculus and a Prolog representation for the Conceptual Graph presented in Figure 8.2, "Mary gave John the book." Take this same example and create a general Prolog rule, "`X` gave `Y` the `Z`" along with a number of constraints, such as "`object(Z)`." Also create a number of Prolog facts, such as "`object(book)`" and show how this conceptual graph can be constrained by using the Prolog interpreter on your simple program.

2. Figure 8.3 presents case frames for the verbs `like` and `bite`. Write Prolog specifications that captures the constraints of these representations. Add other related fact and rules in Prolog and then use the Prolog interpreter to instantiate the constraints that are implicit in these two verb case frames.

3. Create a predicate calculus and a Prolog representation for the two Conceptual Graphs presented in Figure 8.5.

4. Describe an algorithm that could be used to impose graph constraints across the structures of Figure 8.5. You will have to address the nesting issue to handle sentences like "Mary believes that John does not like soup."

5. Create Prolog case frames, similar to those of Section 8.1 for five other verbs, including like, trade, and pardon.

6. Write the Prolog code for a subset of English grammar rules, as in the context-free and context-sensitive parsers in Sections 8.2 and 8.4, adding:

   Adjectives and adverbs that modify verbs and nouns, respectively.

   Prepositional phrases. (Can you do this with a recursive call?)

   Compound sentences (two sentences joined by a conjunction).

7. Extend the stochastic context-free parser of Section 8.3 to include probabilities for the new sentence structures of Exercise 8. Explore obtaining probabilities for these sentence structures from a treebank for natural language processing. Examples may be found on the www.

8. Add probabilities for more word pair relationships as in the lexicalized context-free parser of Section 8.3.2. Explore the possibility of obtaining the probabilistic *bigram* values for the noun–verb, verb–object, and other word pairs from actual *corpus linguistics*. These may be found on the www.

9. Many of the simple natural language parsers presented in Chapter 8 will accept grammatically correct sentences that may not have a commonsense meaning, such as "the man bites the dog." These sentences may be eliminated from the grammar by augmenting the parser to include some notion of what is semantically plausible. Design a small "semantic network" (Section 2.4.1) in Prolog to allow you to reason about some aspect of the possible interpretations of the English grammar rules, such as when it is reasonable for the man to bite a dog.

10. Rework the semantic net parser of Section 14.3.2 to support richer class hierarchies. Specifically, rewrite `match_with_inheritance` so that instead of enumerating the common specializations of two items, it computes this by searching a type hierarchy.

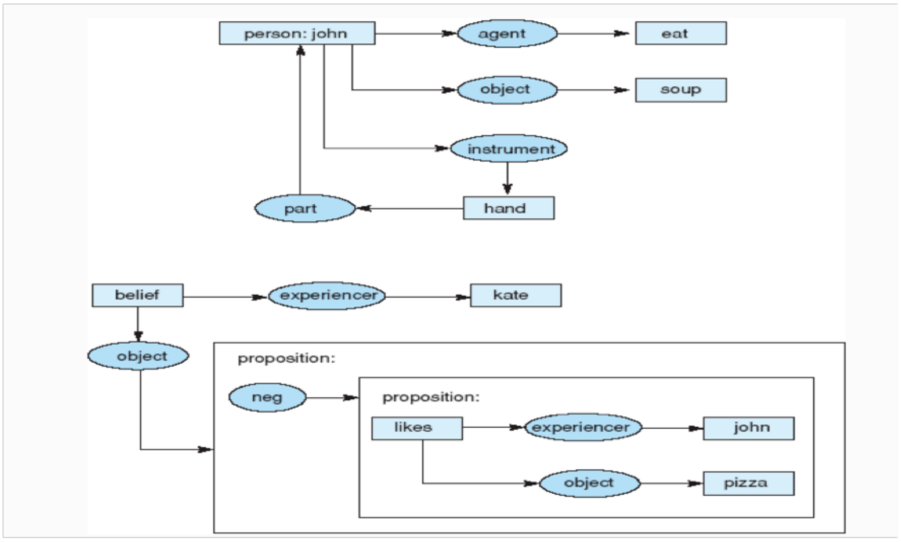

Figure 8.5. Conceptual Graphs to be translated into predicate calculus and into Prolog.