Chapter 1

Tasks and representations

In this chapter we will introduce the tasks that Shalmaneser was made to perform: **Word Sense Disambiguation** (**wsd**) and **Semantic Role Labeling** (**srl**). §1.1 gives a practical example to motivate the importance of these tasks; the remainder of the chapter describes the tasks themselves.

1.1 Why semantic parsing?

Suppose for a minute that we wanted to build a question-answering system. Users could enter any question they were interested in, and it would search through a set of documents for ones that appeared to contain an answer.

One way to build this sort of system would be by matching individual words. If you asked "What's the population of Albuquerque?" it might search for pages containing, say, *population* and *Albuquerque*. But this approach, while it might work okay for some kinds of questions, will get in trouble with others.

First of all, **polysemous** words (ones with several meanings) will cause trouble. If a user asks for directions to the bank, is he looking for a financial institution or a riverside picnic spot?[1] And polysemy is a very widespread phenomenon. Often, a word that seems to have one meaning at first glance will be revealed to have several on closer inspection. The verb *drop*, for instance, may seem straightforward. But consider this: when fruit *drops* from a tree, there's physical movement involved; when prices or temperatures *drop*, there isn't. Even small differences like these can cause trouble. If a user asks when apples tend to drop, does he want to know when the fruit falls from the tree, or when fruit prices

[1]Since the system is taking written input and searching over text files, homographs will also cause trouble. If a user asks how to get more bass, does he mean /bæs/ or /beis/? Does he want advice on fishing or on buying speakers?

are likely to go down? (And those are just uses of *drop* as a verb. Things get even more complicated when we consider its noun senses as well. When we use the phrase *rain drops*, are we talking about droplets of rainwater? Or claiming that rain falls towards the ground? Or that the rate of rainfall goes down over time? As English speakers, we know the answers to these questions intuitively, but a computer won't know unless we program it to figure them out.)

Sometimes, too, we will run into the opposite problem: not one word with many meanings, but one meaning that can be expressed many ways in different words. Suppose a user asks, "When did the Dutch buy Manhattan?" Simply searching for keywords (*Dutch*, *bought* and *Manhattan*, say) will not find the answer if it's expressed with a verb other than *buy* – say, "The Dutch took ownership of Manhattan in 1626" or "The Lenape sold Manhattan to the Dutch in 1626" Because the same facts can be expressed in terms of *buying*, *selling* or *owning*, searching for just one word won't find all of the relevant statements.

This, too, is a widespread problem. We'll encounter it with words like *buy* and *sell* that describe different perspectives on the same situation. But we'll also encounter it with antonyms (ask whether a certain mushroom is *edible*, and you'll miss a document stating that it's *poisonous*) and with synonyms or near-synonyms (ask when the #7 bus *arrives*, and you'll miss a document that tells you when it *shows up*, *reaches* or *gets to* the station).

Finally, there's the risk of finding the right words, with the right meanings, and discovering that they appear in the wrong combination. Suppose a user wants to know about the natural predators of the giant panda. "Are there any animals that eat pandas?" would be a reasonable way to phrase the question — but searching the web for these words returns a lot of irrelevant answers, even without any ambiguity over the word *eat*:

(1.1) Pandas feed mainly on bamboo, but they also eat other plants and even small animals.

(1.2) Since the giant panda is an animal, and therefore "meat," you could technically eat it.

(1.3) What animals eat bamboo other than panda bears?

The problem here is that *eat* describes a situation in which there are two different **ROLES** — there is an INGESTOR (the one doing the eating) and there are some INGESTIBLES (the one getting eaten). A document where the panda is the INGESTOR won't help us answer a question about pandas as INGESTIBLES.

Clearly, there's a problem with the word-matching approach. It's tempting to say that the word-matcher won't work because it wonn't "understand" the meanings of the words it's juggling: it won't "understand" that there are two different kinds of bank, or several ways that something can be said to *drop*. It won't know that buying something is the same as having it sold to you, or that eating is not the same as being eaten. If only the system could "understood" all these things, it would give better answers.

Of course, that's putting it too strongly. Who knows what it would take for a computer to *understand* anything? Here's a less drastic way of putting the same intuition: It would be nice if the system could at least *represent* the meanings of sentences in some way. If we had a good enough way of generating these **MEANING REPRESENTATIONS**, we might be able to tell relevant answers from irrelevant ones. The relevant answers would have meaning representations similar to those of the question, even if they looked very different on the surface. The irrelevant answers, even if they had similar surface forms, would have different meaning representations.

The idea isn't so far-fetched. We already have **PHONETIC** and **PHONEMIC REPRESENTATIONS** like the International Phonetic Alphabet, and **SYNTACTIC REPRESENTATIONS** such as those you see in parse trees. We can even build these representations automatically. A syntactic parser, for instance, takes in text and returns parse trees. Why shouldn't we be able to build our question-answering system on a **SEMANTIC PARSER** that takes in text and returns meaning representations?

1.2 One approach to semantic parsing: Shalmaneser

Shalmaneser is just such a semantic parser. In fact, there are a number of approaches to semantic parsing; Shalmaneser exemplifies one approach. We'll learn about that approach in detail over the course of this guide, but the rest of this chapter gives a rough outline.

Shalmaneser's approach to semantic parsing involves two major tasks and a few minor ones. The major tasks are **WORD SENSE DISAMBIGUATION** (WSD) and **SEMANTIC ROLE LABELING** (SRL). We've already hinted at these tasks up above. Distinguishing the physical-movement sense of the word *drop* apart from the price-changing sense of the same word was an example of WSD. And SRL gives the information we need to recognize that *buying* and *selling* are two sides of the same coin, or that what happens to you when you *eat* is different from what happens when you *are eaten*. Shalmaneser also does some tasks that aren't really semantic at all: for instance, before it does WSD or SRL on a piece of text, it runs the text through a part-of-speech tagger and a syntactic parser. These non-semantic tasks we'll collectively call **PREPROCESSING**.

This chapter and the next will be devoted to describing these tasks in more detail. Later chapters will focus on how to use Shalmaneser, and how to perform experiments aimed at improving its performance. This will include some information on the implementation of these tasks Ñ that is, information on *how* Shalmaneser does them. But in this chapter and the next we will focus on *what* Shalmaneser does, and merely hint at the *how*.

1.2.1 Preprocessing: tagging and syntactic parsing

Before we do any *semantic* parsing, we will want a *syntactic* analysis of the text. There are at least two reasons for this:

The first is that syntactic analysis will solve a few of the problems that we saw in Chapter 1.1. For instance, there are polysemous words whose senses belong to different syntactic categories. We discussed all the different senses that *drop* can have as a verb — but it has nominal senses too, as in (1.4 – 1.5). If we apply part-of-speech tags to a text, they will enable us to distinguish these nominal senses from the verbal ones in (1.6 – 1.7)

(1.4) He wiped a **drop** of sweat from his brow.

(1.5) There has been a sharp **drop** in consumer confidence during the third and fourth quarters.

(1.6) The dry desert air holds little heat, and temperatures **drop** below freezing at night even in the summer.

(1.7) Commodity prices often **drop** in a recession as production slows and demand decreases.

The second reason to do syntactic analysis is that it will help us with semantic analysis. Before we can make sensible predictions about the meanings of words, we have to know how those words are related to one another. And we cannot make *any* predictions about the meanings of phrases unless we know *what those phrases are*.

So how do we represent the results of our syntactic analysis? The standard way is with a **PARSE TREE**.

$< illustration >$

A parse tree is an acyclic directed graph representing the phrase structure of a sentence. The terminal nodes (shown at the bottom) represent individual words. The nonterminal nodes represent phrase-level constituents. Each one represents the phrase that you get by concatenating its children together: the `NP-B` at the left of the tree here represents the noun phrase built by concatenating *its* and *shadow*; the `PP` represents the prepositional phrase built by concatenating *in* and [*its shadow*]; and so on.

These labels on the nonterminal nodes — `NP-B`, `PP` and so on — tell you what type of phrase that node's terminal descendants make up. `NP-B` tells you that that node's terminal descendants — the words *its shadow* — make up a basic noun phrase. `PP` tells you that the words *in its shadow* make up a prepositional phrase.

Although they are not visible in this picture, terminal nodes can also be labeled with a part of speech tag. If the part of speech tags were visible, the node corresponding to *its* would be labeled `P$P` for ("possessive pronoun"), and so on.

As we'll see later in this guide, Shalmaneser does not do its own tagging and syntactic parsing. There are many good taggers and parsers already in existence, and Shalmaneser is set up to use the parses that they create. The parse in FIG TK was created by the Collins parser, and the tags in it (`NP-B`, `PP`, `P$P`, *etc.*) are the ones the Collins parser uses — other parsers might use different abbreviations, or even make more or fewer part of speech distinctions.

1.2.2 Word sense disambiguation

After syntactic analysis, the next task we will need to perform is **WORD SENSE DISAMBIGUATION** (**WSD**). WSD is the task of identifying which sense of a polysemous word is used in any given sentence. For instance, consider the following two senses that exist for the word *drop*

1. to move physically downward
2. to reduce in price, temperature, speed, *etc.*

and the following two sentences:

(1.8) Realizing that he'd suddenly **dropped** to his knees and was about to pop the question, DeAnna quickly blurted, "No, I can't."

(1.9) Shares of department stores **dropped** Thursday after retailers reported largely worse-than-expected same-store sales for June.

To a human reader, it's easy to decide that (1.8) uses sense 1 of the word *drop* and (1.9) uses sense 2. A WSD algorithm must replicate this decision.

How does such an algorithm work? Well, we'll come back to this question in more detail later — but the short answer is that it works by considering lexical and syntactic clues found in the sentence and the surrounding context.

An example of a lexical clue is the presence of the words *shares* and *sales* in (1.9). These words are more often associated with discussion of changes in price, and less often associated with discussion of physical movement. An example of a syntactic clue is the presence of the prepositional phrase *to his knees* in (1.8). Prepositional phrases can appear in discussion of price movement, but they are more common in discussion of physical movement. (Remember our second reason for doing a syntactic parse? Here's an example — without parsing the sentence, we wouldn't have access to the information that *to its knees* comprised a prepositional phrase, and we'd end up missing out on this clue.) By aggregating information from a number of clues like these, the WSD algorithm can generally make a reasonable guess.

So how do we represent the results of word sense disambiguation? There are a number of possibilities; the way Shalmaneser does it is by taking the parse tree obtained in the last step, and applying word sense labels to some of its nodes.

< illustration >

Here we have the same parse tree we saw in the last section. But now some of the terminal nodes — representing individual words in the sentence — have been labeled with word senses. *Creeping* is labeled with SELF_MOTION, *reached* with ARRIVING, and *look* with PERCEPTION_ACTIVE. These labels are meant to be relatively self-explanatory — *creeping* here describes an agent moving under his own power, *reached* describes a place where he arrives, *look* describes him actively investigating his surroundings. But as we'll see in the next chapter, there is also a formal definition for each one.

Ex. 1 — Think of other possible senses for the words in the previous example.

1. When is *creeping* not a form of self-motion?
2. What can *reached* describe other than the shape of a path?
3. What can *look* describe other than active perception?

Ex. 2 — The word sense labels in the previous example are applied to terminal nodes. Can you think of a situation where you would want a word sense label on a non-terminal node?

Note that these word sense labels are not *part* of the parse tree. Rather, they are an additional layer of information represented *on top of* the parse tree. And remember that, as with the parse tree, the picture we are looking at is just a way of visualizing a data structure stored in a computer's memory. What is important is the information at each node and the way the nodes are connected, rather than, say, the color and shape of the lines they are drawn with.

In fact, sometimes we will represent word sense labels differently — using subscripts, as seen here:

(1.10) Creeping$_{\text{SELF\_MOTION}}$ in its shadow I reached$_{\text{ARRIVING}}$ a point whence I could look$_{\text{PERCEPTION\_ACTI}}$ straight through the uncurtained window.

Now let's think back to our question-answering scenario. We saw earlier that our question-answerer would have problems with polysemous words on the one hand and with synonyms on the other hand. We will be able to solve these problems by letting users specify senses for words in their questions.

For instance, look at the question *How far did gold drop in 2008?* Intuitively, we recognize that this is a question about falling commodity prices, that the relevant sense of the word *drop* is CHANGE_POSITION_ON_A_SCALE, and that other senses of the word do not need to be considered. But as we saw earlier, a simple Google search for the keywords *gold*, *drop* and *2008* also returns irrelevant results involving other senses of the word.

Word sense disambiguation could solve this problem. Let us suppose we have parsed the question and applied word sense labels to it, giving the following representation:

< *illustration* >

Note the CHANGE_POSITION_ON_A_SCALE label on the word *drop*. Let us suppose, too, that we have parsed and performed word sense disambiguation on the candidate answers, giving representations like the following:

It is now easy to determine that (?), containing a matching word sense label for *drop*, could be a relevant answer, but that (?) and (?) can not be.

1.2.3 Semantic role labeling

The last task Shalmaneser performs in building a meaning representation is **SEMANTIC ROLE LABELING** (**SRL**).

In section 1.1 we saw that *eat* describes an asymmetric relationship — if *a* eats *b*, that is not the same as *b* eating *a*. We described the situation by saying that *eat* describes a situation with two roles, INGESTOR and INGESTIBLES. Each of its arguments fills one of the roles. Changing which argument fills the INGESTOR role and which fills the INGESTIBLES role will change the meaning of a sentence.

(1.11) Bamboo$_{\text{INGESTOR}}$ **eats** pandas$_{\text{INGESTIBLES}}$ [false]

(1.12) Pandas$_{\text{INGESTOR}}$ **eat** bamboo$_{\text{INGESTIBLES}}$ [true]

You might object to this description, "Wait a minute, you didn't just change the *semantic role labels*. You changed the *word order!* 1.12 and 1.12 mean different things because the words are in a different order."

Well, you'd be right, but that's not the whole story. It's true, English often uses word order to distinguish between the arguments playing different semantic roles. But there are other changes we could make to 1.12 that would preserve the order of the nouns in the sentence, but change the semantic roles they play.

(1.13) Bamboo$_{\text{INGESTIBLES}}$ is **eaten** by pandas$_{\text{INGESTOR}}$ [true]

(1.14) Bamboo$_{\text{INGESTIBLES}}$ **nourishes** pandas$_{\text{INGESTOR}}$ [true]

(1.15) Bamboo$_{\text{INGESTIBLES}}$ is what pandas$_{\text{INGESTOR}}$**eat** [true]

So, as there was with WSD, there are a number of different clues that we look at as English speakers in order to decide which argument fills which semantic role. The SRL task is to make that decision automatically, using many of the same clues that human speakers use. Some of these are lexical and syntactic clues found elsewhere in the sentence. We also can use word sense labels as clues. (As a general rule, we want to use all the information available to us; since we've *got* word sense labels at this point, we may as well *use* them.)

For an example of a syntactic clue, look again at (1.12) and (1.12). In each sentence, the syntactic subject is the INGESTOR, and the syntactic object is the INGESIBLES. In fact, this is reliably true of English sentences with an active form of *eat* as their predicate. So by looking at the voice of the verb and the syntactic position of its arguments, Shalmaneser can tell which argument should be the INGESTOR.

For an example of how word sense labels can act as clues, look at (1.16) and (1.17).

(1.16) I_{AGENT} $\textbf{broke}_{\text{CAUSE\_TO\_FRAGMENT}}$ the $\text{wineglass}_{\text{WHOLE}}$.

(1.17) The $\text{wineglass}_{\text{WHOLE}}$ $\textbf{broke}_{\text{FRAGMENTATION\_SCENARIO}}$.

Each example shows a different sense of the word *break*. In the first, it describes a situation where an agent *causes* a WHOLE item to fragment into pieces. In the second, it describes a situation where a WHOLE item fragments with no cause or agent specified.

Notice that in the first sense, the thing that breaks is the syntactic object of the word *break*. In the second sense, the thing that breaks is the syntactic subject. So word order alone is unhelpful — but by looking at word order *and* the word sense label on the word *break*, we can assign the WHOLE semantic role label to the correct argument.

More generally, each word sense comes with an assortment of semantic roles that it makes available. The INGESTION sense of the word *have* — as in *I'll have three breakfast tacos and a cup of coffee* — makes the INGESTOR and INGESTIBLES roles available. The POSESSION sense of *have* doesn't make those roles available, but instead makes available OWNER and POSESSION. The BIRTH sense — as in *They're having triplets* — makes available MOTHER, FATHER and CHILD. This is the reason we chose to do word sense disambiguation before doing semantic role labeling. Until we know which sense of a word is being used, we don't know which set of role labels to assign to its arguments.

The semantic roles that a word sense makes available are called **ELEMENTS**. So we can say that INGESTOR and INGESTIBLES are elements of INGESTION; MOTHER, FATHER and CHILD are elements of BIRTH; and so on. We'll look closer at frames and their elements in the next chapter.

So how do we represent the results of semantic role labeling? Again, we will do it by overlaying another layer of information on our parse tree.

< *illustration* >

Here we see that, of the elements of SELF_MOTION, the MOVER role is filled in this sentence by *I*. To represent this visually, a line is drawn from SELF_MOTION to *I*, and that line is labeled MOVER.

A semantic role can also be filled by an entire phrase rather than a word. Of the elements of ARRIVING, the GOAL role is filled by prepositional phrase — *straight through the uncurtained window*. To represent this visually, a line is drawn from SELF_MOTION to the PP node corresponding to that phrase, and that line is labeled GOAL.

Semantic role labels can help with asymmetric relationships like that described by the verb *eat*. If we can parse the question "What animals eat pandas?" and determine that *pandas* fills the INGESTIBLES role, we can be sure to provide answers in which *pandas* plays the same role, as in 1.18, rather than unhelpful answers in which it plays the INGESTOR role.

(1.18) Jaguars eat young giant pandas, and if there are a pack of wild dogs they can attack a panda.

Semantic role labels can also help when the same situation is described using different predicates. For instance, if the user were to ask *Who did the Dutch buy Manhattan from?* we would want to be able to retrieve (1.19) as an answer.

(1.19) The Lenape sold Manhattan to the Dutch in 1626.

If we compare the meaning representations for each sentence, we will see that despite containing different verbs — verbs that are in no way synonymous — the two sentences have the same semantic roles filled in the same way. That gives us a hint that we are looking at the same situation: one in which the Dutch are BUYERs, the Lenape are SELLERs, and Manhattan is the GOODS for sale.

< *illustration* >

< *illustration* >

Chapter 2

Frame Semantics

In Chapter 1, we saw how semantic parsing could help us with a natural language processing task — answering questions — and how we could break the task of semantic parsing down into subtasks: syntactic parsing, word sense disambiguation (WSD) and semantic role labeling (SRL). But in the course of discussing those tasks, some additional questions may have come up: discussing WSD, you might have found yourself wondering *where do you draw the line between word senses? How do you know how many senses a word has?* and *how do you describe the connections between senses?* And when it comes to SRL, you may have wondered, *how many roles should we distinguish? How do you describe the connections between roles?*

FRAME SEMANTICS, developed by Charles Fillmore and the FrameNet team at Berkeley, gives one set of answers to these questions. It gives us a way of describing and differentiating word senses and semantic roles, and of drawing the connections between them. The FrameNet project has also created a freely available database that lists the possible word senses and semantic roles for thousands of English words. By applying word sense and semantic role labels drawn from the FrameNet database, we can be confident that the set of labels we're using is coherent and reasonable.

In this chapter, we will explain the basic concepts of Frame Semantics, discuss their relevance to WSD and SRL, and describe how they show up in Shalmaneser's meaning representations. We will also introduce you to the useful information available on the FrameNet project's website.

2.1 What are frames?

Suppose you're teaching English to a visitor from Mars. He's got no clue about human habits, customs or concepts — he needs everything explained to him. And suppose one day he asks you what the word *buy* means. What do you say?

Your first answer might be, "Well, when you *buy* something, you give somebody money for it, and then you own the thing that you bought." But that answer just raises more questions. "What's *money*?" says the Martian. "And what's *owning*?" It seems like the concepts of *buying*, *money* and *owning* are connected, so that you can't understand the first without understanding the other two.

And here's the tricky part: Suppose instead that the Martian had started out by asking "What's *money*?" A natural answer might have been, "*Money*'s what you use to buy things you want to own." Suppose he had started by asking "How do you know when someone *owns* something?" At least in the modern world, the answer's likely to involve money and buying too.

So it looks like *whichever* of the concepts you start with, you'll need to use the other two to explain it. You have to understand the whole system of commerce in order to make sense of the words *buy, money* and *own*. Without the whole system of concepts, the individual words are meaningless.

Frame Semantics begins with the observation that human knowledge (and, as a result, human language) are full of interconnected systems of concepts like this. Each system of concepts is called a **FRAME**. In this case, we might talk about a COMMERCE_BUY frame, containing all the background knowledge you need to understand buying. Sometimes frames overlap. There's another frame, called COMMERCE_SELL, containing all the background knowledge you need to understand selling. Much of the same knowledge falls under both frames (what money is, what ownership is, what goods are, how to exchange things); as we'll see later, some of the details are different.

When you need the knowledge in a frame to understand the meaning of a particular word, we say the word **BELONGS** to that frame. *Buy, purchase* and *rent* belong to the COMMERCE_BUY frame. *Sell, vend, auction, hawk, lease* and *retail* belong to the COMMERCE_SELL frame.

Ex. 3 Renting and leasing aren't the same as buying and selling — what are they doing in the same frame?

When you use a word that belongs to a frame, you **EVOKE** all the background knowledge in the frame itself — that is to say, you call all that knowledge up in the listener's mind. If I tell you *Bob bought an apple*, you'll probably assume that someone sold him the apple, that he used money to buy it, that the money and the apple changed hands, and so on. I haven't told you any of that information outright; I didn't need to. It was **EVOKED** as part of the COMMERCE_BUY frame when I used the word *bought*.

2.2 Frames and word senses

You have probably noticed already that we're using the same small-caps convention for writing frame names as we did for writing word senses. And you've probably already guessed what this means — that we'll be using the names of frames as word sense labels. If that's what you're thinking, you're right. But let's take a minute to see *why* frame names make good word sense labels.

To see that, let's turn again to our Martian learner. And let's imagine now that we're trying to teach him to deal with polysemous words. We'll see that for each sense of the word, we'll need to teach him a brand-new set of background knowledge.

For instance, take the word *drop*, which can refer to a change in some measured quantity (as in "The Dow Jones dropped"), to a change in physical position (as in "His jaw dropped"), or to a literal or metaphorical act of letting go (as in "She dropped her keys on the table"). To understand the changing measurement sense, the Martian will need to know about numbers, units of measurement and rates of change. This background knowledge constitutes one frame. To understand the physical movement sense, he will need to know about movement in general, and about directions, and he'll need to know why we single out one direction as *down*. This background knowledge constitutes a different frame, distinct from the last one. The letting-go sense of the word requires all that knowledge about movement, *plus* an understanding of control and causation — this larger set of background knowledge constitutes yet another frame.

In fact, it will turn out to be true in general of polysemous words that each sense belongs to a different frame. This is what makes frame labels useful for WSD. If we label each instance of a word with the frame it evokes, that will specify which sense of the word we're dealing with.

Table 2.1 shows some of the frames that different senses of the word *drop* belong to.[1]

Ex. 3 — Here are some sentences using the word *drop*. Which frame does each sentence evoke? Do any of them contain a sense of the word we haven't seen so far?

1. Shares of department stores dropped Thursday after retailers reported largely worse-than-expected same-store sales for June.
2. Realizing that he'd suddenly dropped to his knees and was about to pop the question, DeAnna quickly blurted, "No, I can't."
3. The road winds around and eventually drops down to a peaceful spot in the grove.
4. A storm that passed through the southern Red River Valley earlier to-

[1]These summaries and examples are adapted from FrameNet frame reports; see §2.5.2 to learn how to get more information.

Frame	Summary	Example
CHANGE_POSTURE	A protagonist changes the overall position and posture of the body.	He **dropped** to his knees.
MOTION_DIRECTIONAL	A theme moves in a certain direction which is often determined by gravity or other natural, physical forces.	In the kitchen, the towel **dropped** onto the floor.
PATH_SHAPE	A stationary road provides a path that a moving entity could travel along.	The road quickly **drops** after passing the final turn.
BODY_MOVEMENT	An agent uses part of his/her body to perform an action or movement.	Pat **dropped** her hands to her lap.
CHANGE_POSITION_ON_A_SCALE	An item changes its position on a scale measuring some attribute.	Microsoft shares **dropped** from 12 3/8 to 7 5/8.

Table 2.1: Frames that *drop* can belong to

day dropped hail as large as golf balls near Wilkin, Minnesota.

2.3 Frames and SRL

We mentioned in the last section that a frame contains all the background knowledge required to understand a predicate. It turns out that a great deal of that background knowledge can be expressed in terms of semantic roles.

Let's look again at the background knowledge evoked by *Bob bought an apple*. We know that someone sold him the apple. Another way to put this is to say that someone is playing the SELLER role. We know that money was used to buy the apple, and that the money and the apple changed hands — or in other words, that something is playing the MONEY role. The relationship that Bob and the apple have to the predicate *buy* can also be expressed in terms of their semantic roles. Bob is the BUYER, the apple is the GOODS.

Now recall that predicates belonging to the same frame evoke the same background knowledge. This means in particular that when two predicates belong to same frame, they should involve the same semantic roles. We say that these roles are **ELEMENTS** of the frame. We mentioned earlier that *purchase* was in the COMMERCE_BUY frame along with *buy* — and sure enough, *purchasing* also invovles a BUYER, a SELLER, MONEY and GOODS. This reinforces our decision to put *buy* and *purchase* in the same frame, and it suggests that BUYER, SELLER, MONEY and GOODS are elements of COMMERCE_BUY.

We can make a distinction between the **CORE** elements of a frame and the **NON-CORE** elements. The core elements of a frame are at the foreground of the situation described by the predicates in that frame. For instance, BUYER is a core element of COMMERCE_BUY, because predicates like *buy* and *purchase* focus on the buyer's perspective; compare verbs from the COMMERCE_SELL frame like *sell* or *vend*, which focus on the seller's perspective. Non-core elements, on the other hand, provide background information. The non-core elements of COMMERCE_BUY include some roles that we've already seen, SELLER and MONEY, as well as some we haven't considered yet: MEANS, TIME, PURPOSE, MANNER, RATE and so on.

This means that Frame Semantics can be helpful in performing a SRL task. If we know which frame a predicate evokes, that tells us which semantic roles we can assign for that predicate. And it also tells us which of those roles are most likely to be present — the core roles. In some cases, it even tells us where to look for the phrase that fills a particular role. For instance, predicates in the COMMERCE_BUY frame are likely to have the BUYER as their subject, while those in the COMMERCE_SELL frame are likely to have the SELLER as theirs. If a SRL algorithm has information about which frames the predicates in a sentence belong to, it can exploit this information to assign semantic roles more accurately.

2.4 The FrameNet project

Let's sum up what we've learned so far. By labeling each predicate in a sentence with the frame it belongs to, and each argument with the frame element describing its semantic role, we can put together a useful representation of that sentence's meaning. This Frame Semantic representation will

- distinguish between the meanings of ambiguous predicates,
- show when similar meanings are conveyed by sentences with different surface forms, and
- link each sentence's meaning to a wealth of relevant background information.

But before we can use Frame Semantic representations in a practical system — say, our question-answering system from §1.1 — there's a great deal of preliminary work that needs to be done. For starters, we'll need a list of frames that commonly appear in English texts, and for each frame we'll need a list of the words that can evoke it and the semantic roles that might be involved.

This is a non-trivial task, for two reasons. The first is the sheer size of the task: to get wide coverage, we'll need entries for hundreds of frames, the largest of which might include dozens of evoking words and dozens of semantic roles. The second reason is that there are careful decisions to be made every step of the way. We've already given a little thought to the question of whether *buy* and *sell* belong in the same frame or in separate ones. In the end, we decided they belonged in separate frames, and marshalled a bit of evidence for that decision — but when you consider that there are hundreds of decisions like that to be made, and that the evidence is often subtle or even contradictory, it's clear that compiling a complete and consistent list of frames is a serious job.

The FrameNet project is a group of linguists working on that job. To date, they have compiled a list of over 825 frames. Each frame's entry includes

- a list of words that can evoke it,
- a list of the semantic roles that can be involved, and a description of how those roles are related, and
- a set of annotated example sentences showing how different combinations of semantic roles can be instantiated.

FrameNet is an incremental project; new frames are still being identified and described, and the existing entries are being fleshed out. Occasionally work on a new frame will give the researchers a better understanding of a frame that's already been described, and when that happens the old frame's entry will be edited to incorporate the new information.

The results of the FrameNet project are freely and publicly available. The goal is to keep other researchers from needing to reinvent the wheel. If you're building a tool like a semantic parser, or a whole system like the question-answering system we've been considering, and you want to use Frame Semantics as your meaning representation system, you can incorporate all the preliminary work that the FrameNet researchers have done.

2.4.1 How to use the FrameNet website

The meaning representations that Shalmaneser generates are based on frame and semantic role labels drawn from the FrameNet project's inventory. Often, we'll want to know what the FrameNet researchers have to say about one of these labels — for instance, to understand the label's meaning better, or to see if Shalmaneser is applying it correctly.

There are a few ways to get this information from the FrameNet website:

- We can look up an English word to get a list of the frames it can belong to. From that list we can access information on any of those frames.

- We can look up a frame directly if we know its name.
- To get information on a semantic role, we must first look up the frame in which it is an element. We can do this using either of the above methods.

To look up an English word on FrameNet, do the following:

1. Open up a web browser and navigate to FrameNet's address:

   `http://framenet.icsi.berkeley.edu`

2. At the top of the left-hand column, you'll see a text box marked **Search**. In that box, enter the word you're looking for.
3. Press the **Enter** key or click the orange magnifying-glass icon to execute the search.

The search results will appear in the right-hand column. For instance, the results of searching for *drop* are shown in Figure 2.1.

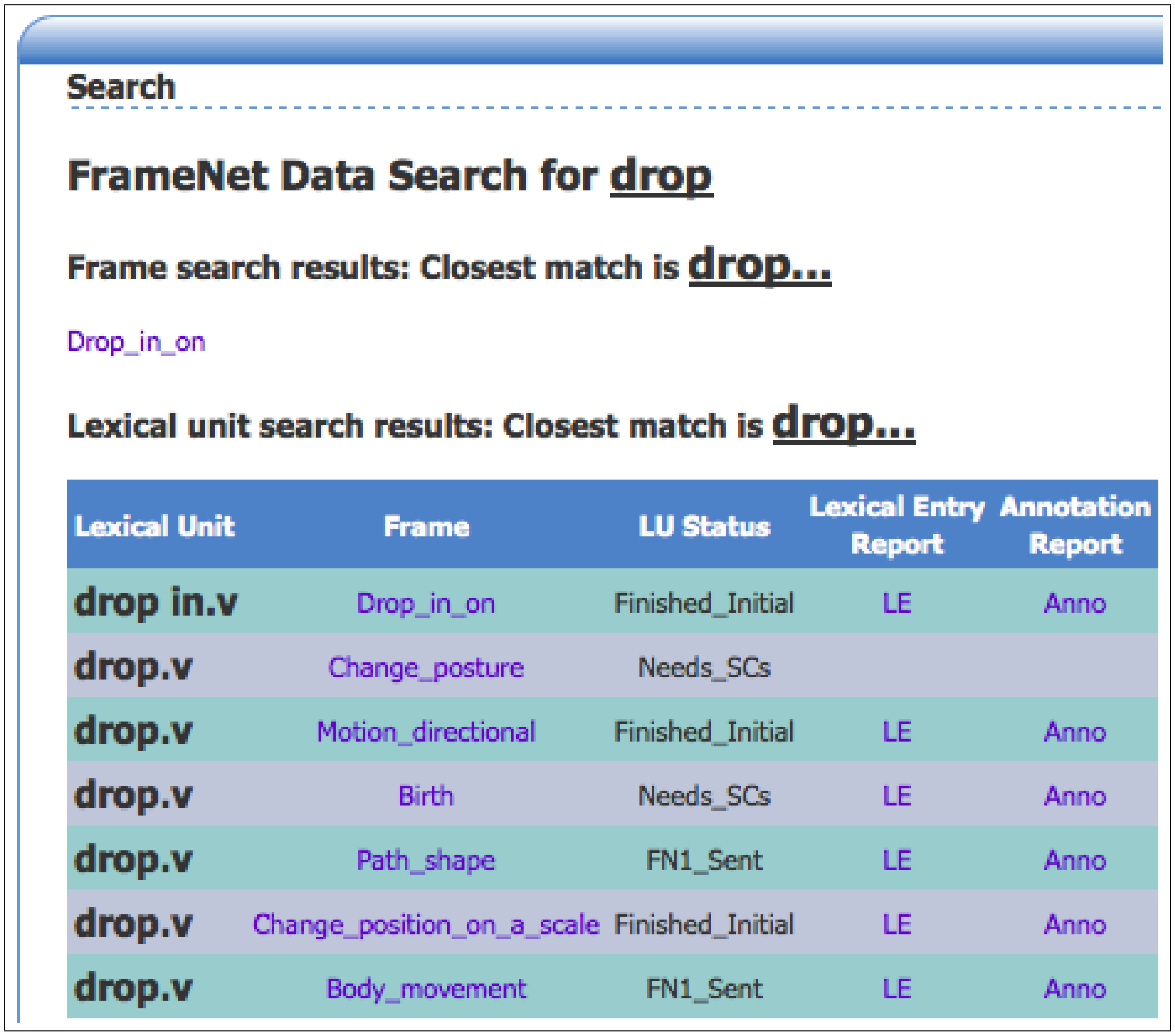
This shows that there is one frame (DROP_IN_ON) and seven different lexical units containing `drop`. When two or more of the lexical units are homonyms, you can distinguish them by the frames they belong to: in this case, CHANGE_POSTURE (as in "She dropped to her knees"), MOTION_DIRECTIONAL (as in "You could hear a pin drop") and so on.

All of the frame names here are links. Clicking on one will open up a new window (or tab, depending on your browser settings) with a **FRAME REPORT** giving basic information on the frame. Figure 2.2 shows the frame report for MOTION_DIRECTIONAL.

The frame report gives a brief definition of the frame and a listing of all the frame elements, both core and (if you scroll down) non-core, with an example sentence or two for each element. The frame elements are color-coded, and these colors are used to indicate which words or phrases in the examples play which semantic role. (For instance, in Figure 2.2, the DIRECTION frame element is highlighted in green, and so are all the words that play the DIRECTION role in the example sentences.) This is a different way of conveying this information than Shalmaneser uses, but the information conveyed is the same.

If you scroll down to the bottom of the frame report, you will see a listing of all the lexical units that belong to that frame, as shown in Figure 2.3.

For most lexical units, you will also see links to a lexical entry (abbreviated LE) and an annotation report (abbreviated ANNO). The annotation report is useful for getting a sense of the different ways in which a lexical unit is used. It includes a summary of the frame elements with which it can occur (Figure 2.4) and more annotated example sentences, organized according to the kinds of argument the lexical unit takes in them. Here, for



Search

FrameNet Data Search for drop

Frame search results: Closest match is drop...

Drop_in_on

Lexical unit search results: Closest match is drop...

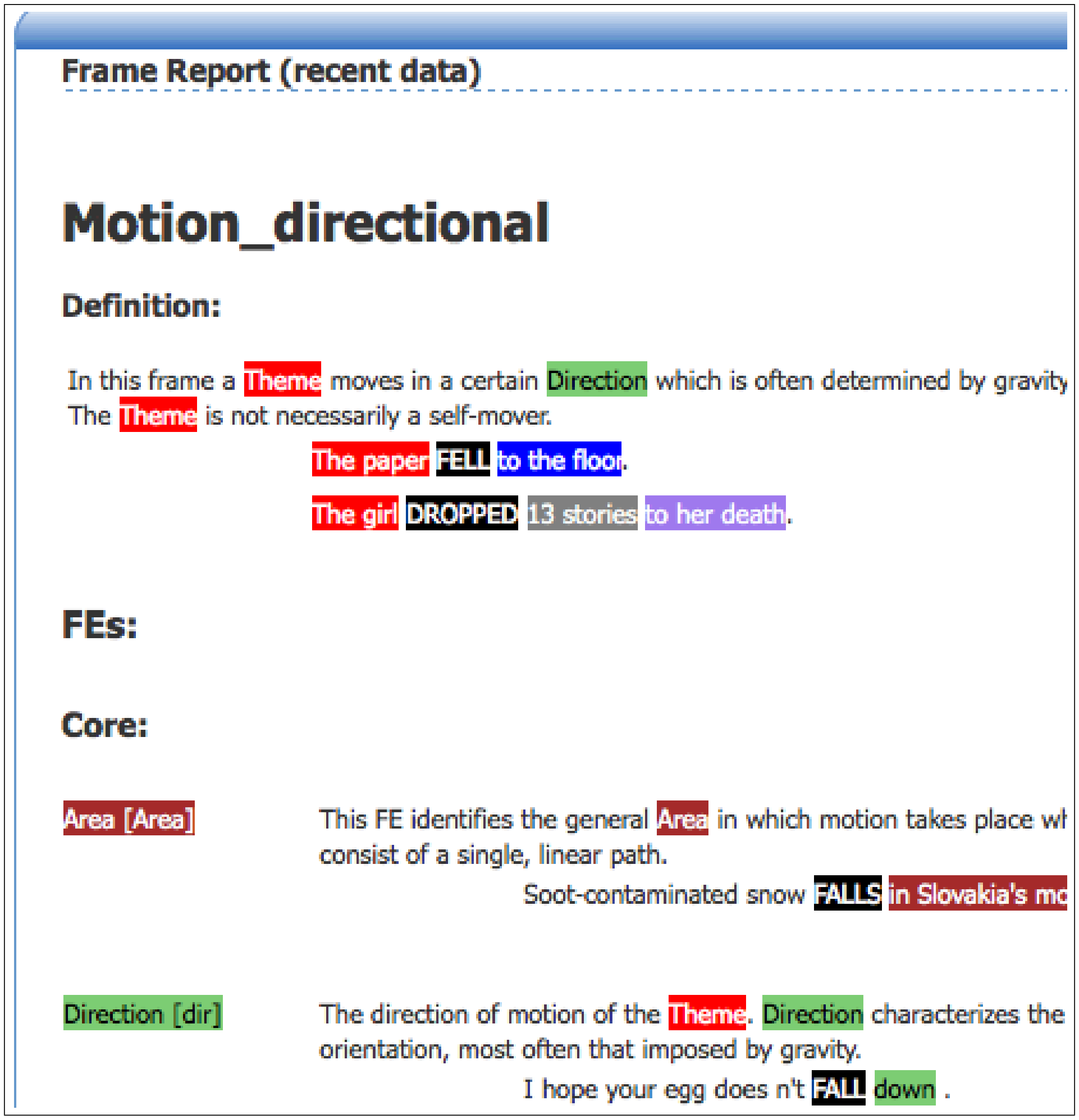
Lexical Unit	Frame	LU Status	Lexical Entry Report	Annotation Report
drop in.v	Drop_in_on	Finished_Initial	LE	Anno
drop.v	Change_posture	Needs_SCs		
drop.v	Motion_directional	Finished_Initial	LE	Anno
drop.v	Birth	Needs_SCs	LE	Anno
drop.v	Path_shape	FN1_Sent	LE	Anno
drop.v	Change_position_on_a_scale	Finished_Initial	LE	Anno
drop.v	Body_movement	FN1_Sent	LE	Anno

Figure 2.1: FrameNet search results for `drop`

instance, we see the different prepositional phrases that the MOTION_DIRECTIONAL sense of *drop* can take as arguments (Figure 2.5).

All those results came from searching for an individual word. You can also search for a frame directly if you know its name:

1. At the top of the left-hand column, in the text box marked **Search**, enter all or part of the frame's name.
2. Press **Enter** or click the orange magnifying glass icon to execute the search.


Frame Report (recent data)

Motion_directional

Definition:

In this frame a Theme moves in a certain Direction which is often determined by gravity
The Theme is not necessarily a self-mover.

The paper FELL to the floor.

The girl DROPPED 13 stories to her death.

FEs:

Core:

Area [Area] This FE identifies the general Area in which motion takes place wl
consist of a single, linear path.

Soot-contaminated snow FALLS in Slovakia's mc

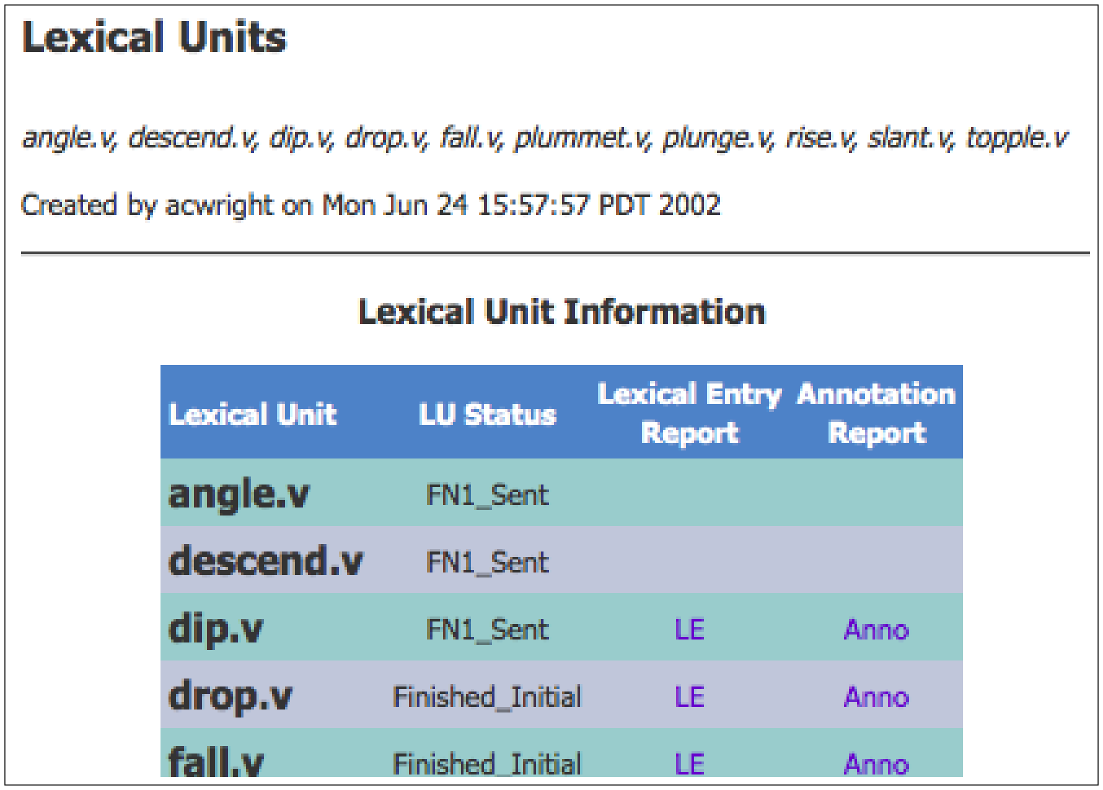
Direction [dir] The direction of motion of the Theme. Direction characterizes the
orientation, most often that imposed by gravity.

I hope your egg does n't FALL down .

Figure 2.2: FrameNet frame report for MOTION_DIRECTIONAL

In the right-hand column, under the heading **Frame search results**, you'll see the frames most closely matching the string you entered. There may be only one (searching for `fill` yields the single frame match FILLING) or many (searching for `motion` yields MOTION, MOTION_DIRECTIONAL, MOTION_NOISE and MOTION_SCENARIO — as well as the verb *motion*, as in "He motioned for me to come closer").

As before, the frame names are all clickable links. Clicking on MOTION_DIRECTIONAL



Lexical Units

angle.v, descend.v, dip.v, drop.v, fall.v, plummet.v, plunge.v, rise.v, slant.v, topple.v

Created by acwright on Mon Jun 24 15:57:57 PDT 2002

Lexical Unit Information

Lexical Unit	LU Status	Lexical Entry Report	Annotation Report
angle.v	FN1_Sent		
descend.v	FN1_Sent		
dip.v	FN1_Sent	LE	Anno
drop.v	Finished_Initial	LE	Anno
fall.v	Finished_Initial	LE	Anno

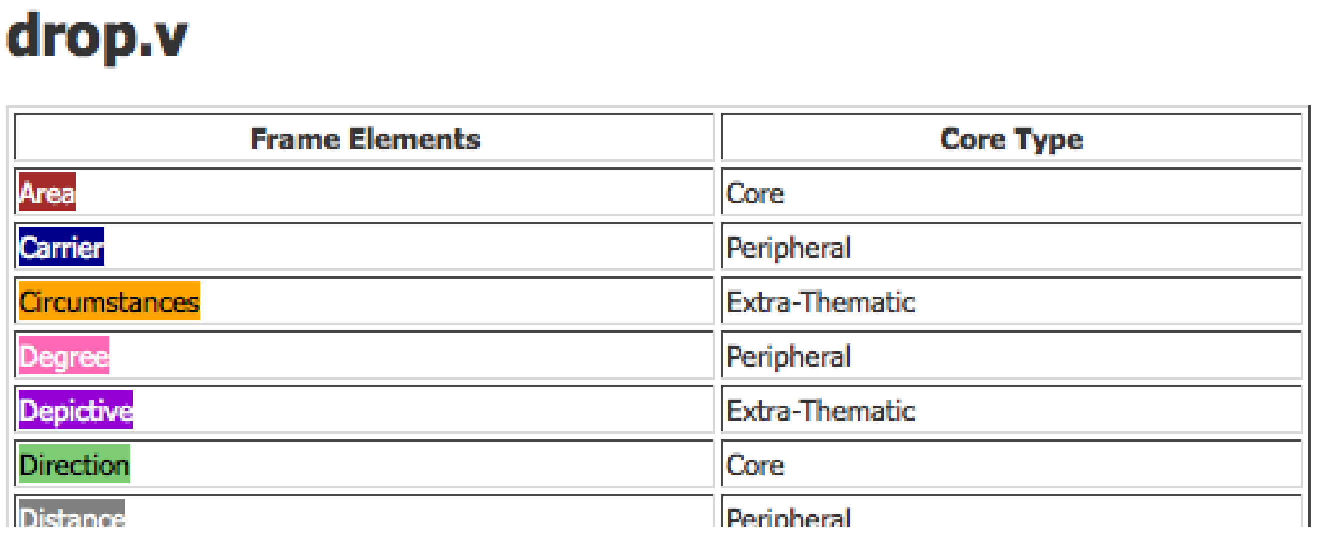
Figure 2.3: FrameNet frame report for MOTION_DIRECTIONAL: lexical units belonging to the frame

will bring you to the same frame report we saw in Figure ??, and clicking on the others will bring you to similar reports for those frames.

Ex. 4 — Look at the following semantic parse:

$< illustration >$

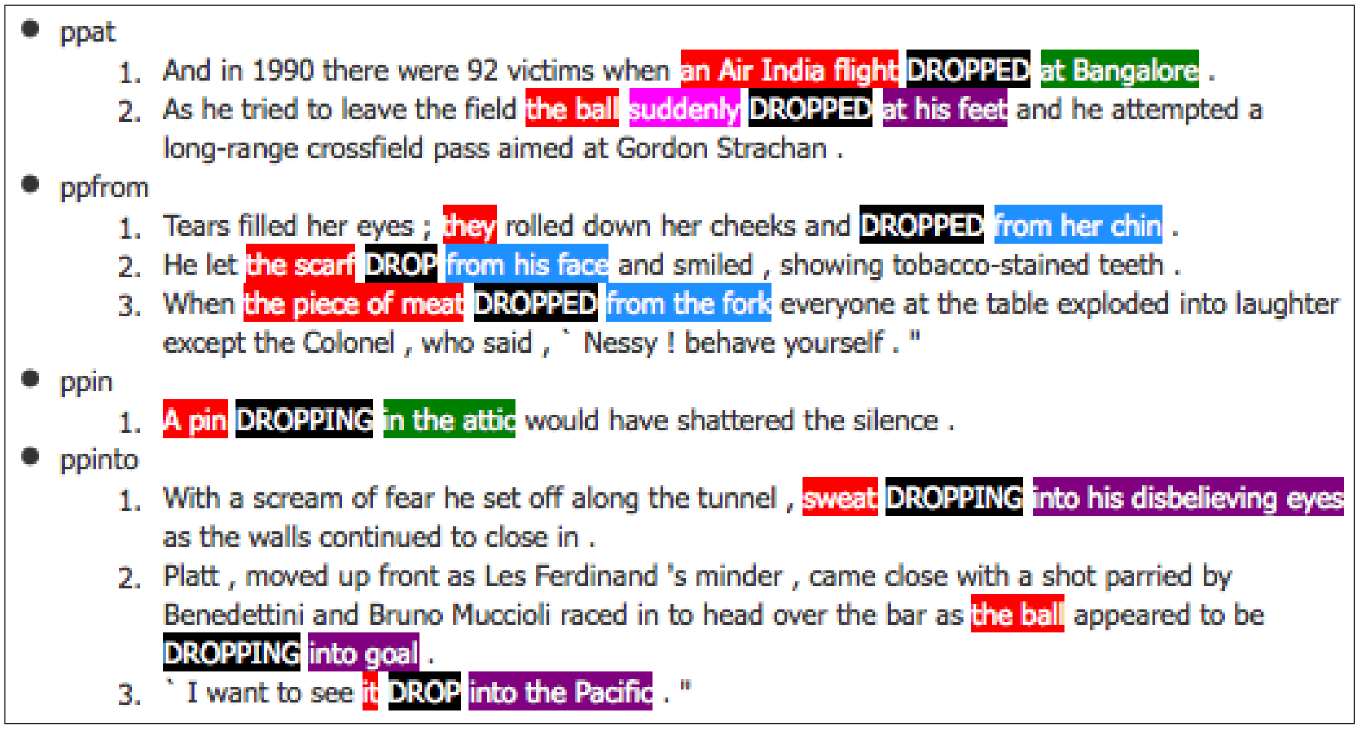
1. What frame is assigned to each predicate?
2. Look these frames up on FrameNet — does the way they've been assigned make sense?
3. Look up the semantic role labels — does the way they've been assigned make sense?



drop.v

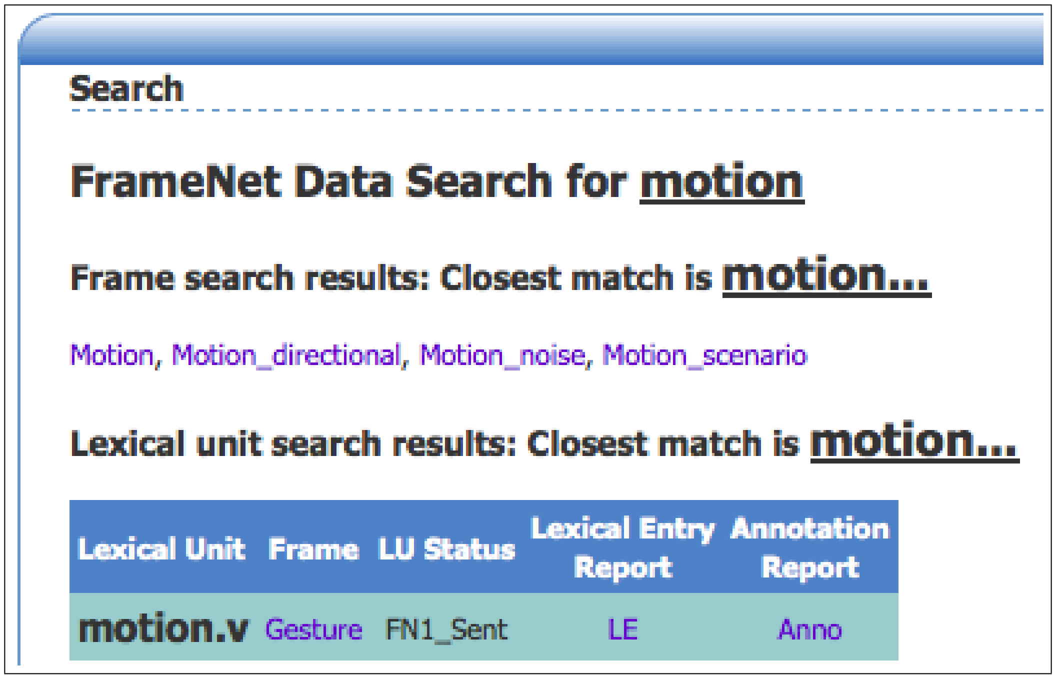
Frame Elements	Core Type
Area	Core
Carrier	Peripheral
Circumstances	Extra-Thematic
Degree	Peripheral
Depictive	Extra-Thematic
Direction	Core
Distance	Peripheral

Figure 2.4: FrameNet annotation report for *drop*



- ppat
    1. And in 1990 there were 92 victims when an Air India flight DROPPED at Bangalore .
    2. As he tried to leave the field the ball suddenly DROPPED at his feet and he attempted a long-range crossfield pass aimed at Gordon Strachan .
- ppfrom
    1. Tears filled her eyes ; they rolled down her cheeks and DROPPED from her chin .
    2. He let the scarf DROP from his face and smiled , showing tobacco-stained teeth .
    3. When the piece of meat DROPPED from the fork everyone at the table exploded into laughter except the Colonel , who said , ` Nessy ! behave yourself . "
- ppin
    1. A pin DROPPING in the attic would have shattered the silence .
- ppinto
    1. With a scream of fear he set off along the tunnel , sweat DROPPING into his disbelieving eyes as the walls continued to close in .
    2. Platt , moved up front as Les Ferdinand 's minder , came close with a shot parried by Benedettini and Bruno Muccioli raced in to head over the bar as the ball appeared to be DROPPING into goal .
    3. ` I want to see it DROP into the Pacific . "

Figure 2.5: Annotation report for *drop*, continued: annotated examples organized by argument structure.


Search

FrameNet Data Search for motion

Frame search results: Closest match is motion...

Motion, Motion_directional, Motion_noise, Motion_scenario

Lexical unit search results: Closest match is motion...

Lexical Unit	Frame	LU Status	Lexical Entry Report	Annotation Report
motion.v	Gesture	FN1_Sent	LE	Anno

Figure 2.6: FrameNet search results for `motion`

Chapter 3

How to Parse Text with Shalmaneser

Chapters 1 and 2 introduced us to the meaning representations that Shalmaneser generates when it parses a sentence, and to the concepts from Frame Semantics that we need to make sense of that representation. In this chapter, we will see how to parse text with Shalmaneser.

3.1 Shalmaneser's two interfaces

There are two different interfaces you can use to control Shalmaneser. One is a web-based interface; you can run it in a browser window without installing anything special on your own computer, and it will work regardless of which operating system you use.

The other is a graphical interface that runs under Linux. You can install it on your own computer or use it at a computer lab that has it installed (such as UT's Computational Linguistics lab). We'll be assuming here that you're using it in a lab. For instructions on installing it yourself, or connecting remotely to a lab machine, see Appendix B.

For parsing individual sentences, you can use either interface. Anything beyond that can't be done with the web interface, and will require the desktop GUI.

3.2 How to parse a sentence

To parse a sentence using the web interface to Shalmaneser, follow these steps:

1. Open up a web browser and navigate to the Shalmaneser web interface.
   Currently, that's at the following address:

```
http://comp.ling.utexas.edu/shalmaneser/shalviz/
```

2. In the box labeled **Text to parse**, enter the sentence you want to parse. If you want, you can enter more than one sentence, but parsing a long text this way can be awkward. See §3.3, "Parsing text from a file," for a better way.

3. Under the **Text to parse** box, you will see some **Options**. For now, select **FrameNet** as your training corpus, **Collins** as your parser and **Timbl** as your machine learning toolkit. We'll look at some other combinations later in this chapter.

4. Press **Parse**.

Shalmaneser will think for a while — you'll see a status bar moving at the top of the screen to let you know it's still working. Then, it will show you a diagram of the sentence or sentences you asked it to parse.

Screenshot

To parse a sentence using the graphical interface to Shalmaneser, you need to launch the program first. Assuming you're using a computer with Shalmaneser installed on it (*e.g.* one of the UT Computational Linguistics lab machines), follow these steps:

1. Open up a terminal window. In the UTCL lab, you can do this by clicking the **Applications** menu, pointing to **Accessories**, and chosing **Terminal** from the submenu.

2. Go to the directory where Shalmaneser is located. In the UTCL lab, you can do this by typing the following command in the terminal window and pressing Enter:

   ```
   cd /groups/projects/shalmaneser/gui/
   ```

   If you're someplace else, ask your lab administrator what command to use.

3. Launch Shalmaneser. On most computers, the command to do this is:

   ```
   ./shalgui_linux32
   ```

   If you're using a computer with a 64-bit processor, you should use this command instead:

   ```
   ./shalgui_linux64
   ```

If you want to keep using the terminal for other things — moving or renaming files, launching other programs, and so on — you can type an ampersand after the command, like so:

```
./shalgui_linux32 &
```

This will let you use the command line even while Shalmaneser is running in the background.

A new window will open up. In that window, follow these steps:

1. In the box labeled **Text to parse**, enter the sentence you want to parse. If you want, you can enter more than one sentence, but parsing a long text this way can be awkward. See §3.3, "Parsing text from a file," for a better way.
2. In the box labeled **Classifiers**, make sure the default classifiers are selected: **Default WSD Classifier** under **Word Sense Disambiguation** and **Default SRL Classifier** under **Semantic Role Labeling**.
3. Press **Go**.

Shalmaneser will think for a while, and then show you a diagram of the sentence or sentences you asked it to parse.

3.3 How to parse text from a file

If you're using the graphical interface, you can also use Shalmaneser to parse text from a file or a whole directory of files. This can be handy, for instance, if you're working with a large corpus.

Files that you want to work with this way need to be uncompressed plain text.

To parse text from a single file:

1. From the pull-down menu in the box labeled **Text to parse**, select **File**.
2. Choose a file to parse. There are two ways to do this.
   - If you know the path to the file, you can enter it directly in the text box provided. In the UTCL lab, you can find interesting text files in `/groups/corpora`. For instance, you could enter

     ```
     /groups/corpora/gutenberg/treas11.txt
     ```

    to parse the text of the Gutenberg Project edition of *Treasure Island*.

  - Otherwise, press **Browse**. A dialog box will open up that will let you select a text file to parse from anywhere on your computer.

3. In the box labeled **Classifiers**, make sure the default classifiers are selected: **Default WSD Classifier** under **Word Sense Disambiguation** and **Default SRL Classifier** under **Semantic Role Labeling**.

4. Press **Go**.

To parse text from a whole directory of files:

1. From the pull-down menu in the box labeled **Text to parse**, select **Directory**.

2. Choose a directory of files to parse. There are two ways to do this.

   - If you know the path to the directory, you can enter it directly in the text box provided.
   - Otherwise, press **Browse** and select a directory from the dialog box that appears.

3. In the box labeled **Classifiers**, make sure the default classifiers are selected: **Default WSD Classifier** under **Word Sense Disambiguation** and **Default SRL Classifier** under **Semantic Role Labeling**.

4. Press **Go**.

Ex. 5 — Try parsing other sentences. Does Shalmaneser give analyses that make sense? Can you find any patterns in what Shalmaneser handles well and what it doesn't? Here are some things to try:

1. Sentences in different styles and registers. (Can Shalmaneser cope with technical writing? Archaic English — maybe Shakespeare or the King James Bible? Slang? Metaphorical or poetic language?)
2. Try sentences with tricky syntax. Can you find cases where the syntactic parser makes mistakes? What happens when it does?
3. Try parsing a long text, and then parsing the sentences individually. Does taking them out of context ever make a difference?

Ex. 6 — Here is one pattern you may notice after parsing a number of sentences: If a parse contains a syntax error, the odds are high that it will contain semantic errors too. But the converse is not true — it is fairly common for a parse to contain a semantic error without containing any syntax errors. Why do you suppose this might be?

Chapter 4

Machine Learning

We said in the introduction that Shalmaneser was a **MACHINE LEARNING** system. In this chapter we will flesh out that statement by explaining what machine learning is and how it underlies Shalmaneser's behavior.

Machine learning approaches came into use for linguistic tasks as a response to the problems with earlier **RULE-BASED** approaches. §4.1 describes some of these problems briefly. §**??** gives an overview of machine learning in general, and §**??** explains how it is used in Shalmaneser.

As we will see, machine learning approaches depend on a sort of abstraction called **FEATURIZATION**. Earlier we talked about using syntactic and semantic 'clues' to do word sense disambiguation (WSD) and semantic role labeling (SRL). Featurization lets us reduce a text with all its complications to a bundle of 'clues' or **FEATURES**. §**??** describes this process.

4.1 Rule-based approaches (and their problems)

Here's one way we could have written a program like Shalmaneser:

- Work out a set of rules for doing word sense disambiguation (WSD) and semantic role labeling (SRL).
- Write a program that follows those rules.

This way of doing things, not surprisingly, is called a **RULE BASED** approach to the problem. Early natural language processing systems of all sorts tended to be rule-based, but it turned out there were several problems with this approach.

Perhaps the easiest problem to see is the sheer number of rules that such an approach requires. To build Shalmaneser we'd first have needed rules for recognizing each of the 800+ frames identified in FrameNet. (*What rules do you follow to find out if a word evokes the* ABANDONMENT *frame? To find out if it evokes the* ABOUNDING_WITH *frame?* ... and so on through *To find out if it evokes the* WORKING_ON *frame?*) And it gets worse from there. Each of those frames has dozens of semantic roles as elements; for each of those roles, we'd have needed rules for recognizing whether a particular word fills it. All in all, it would have been an enormous amount of work.

And now, we've been assuming that it will be possible to find rules for recognizing each of these frames and roles. It turns out that this assumption may be incorrect, and this is another problem with rule-based approaches. In practice, hard-and-fast rules are quire rare when it comes to linguistic phenomena.

4.2 Machine Learning

MACHINE LEARNING is an alternative to this sort of rule-based approach, and one that circumvents both of these problems. A machine learning system works with statistical generalizations rather than hard-and-fast rules. And rather than requiring a human operator to program the statistical generalizations in explicitly, it can "learn" them by examining a set of training data. The next few sections will explain why this sort of learning is possible, and give a broad sketch of how it is done.

Fundamentally, a machine learning algorithm is one that can approximate a function based on examples of its input and output.

This is a very broad definition. In particular, it does not specify *what sort* of function we are trying to approximate. You're probably used to hearing about functions from numbers to numbers, as in mathematics, and it could be one of those that we're trying to approximate. But it could be a function with different inputs and outputs.

4.2.1 Classifier Functions

In particular, when dealing with natural language, we are often interested in **CLASSIFIER FUNCTIONS** — functions that return as output one label out of a finite set of possibilities.

In boxing and wrestling, competitors are grouped by weight, so that nobody is forced to face a much larger opponent. In Olympic boxing, for instance, there are eleven weight classes, from "light flyweight" on up to "super heavyweight." The Olympic rules define a classifier function that takes a single input feature and outputs a weight class label.

Other ways of classifying people by body type use two or more features. For instance, doctors in the US will often use a patient's height and weight to classify the patient as underweight, average or overweight. The doctor's height/weight chart defines a classifier function that takes two input features and outputs a category label. Note that the category label is based on *both* features — neither height or weight alone is enough to label someone as under- or overweight. Most interesting classifier functions are like this — they capture the interaction between two or more features.

Here's an example with five features: the Environmental Protection Agency puts different fuel economy restrictions on different kinds and sizes of vehicle. EPA regulations distinguish sixteen size categories — "two-seater," "compact sedan," "large sedan," "small station wagon," "cargo van" and so on — each one subject to different rules. To assign a vehicle to a size category, the EPA considers five factors: its weight, the volume of its passenger and cargo space, the number of doors it has, the number of seats it has, and the year it was built. These regulations, taken together, define a classifier function with five input features that outputs a vehicle size category label.

In the examples so far, the classifier function has served to define the categories. But we can also use a classifier function to *recognize* and label members of categories that are defined independently. So far, too, we've been looking at examples for which machine learning wouldn't be necessary. It's perfectly practical to use a list of human-defined rules to distinguish small station wagons from large ones — in fact, it's what the EPA does. But because naturally occurring categories are often messier and more complex than the ones humans define, they often require more features with more complicated relationships, and this is where machine learning really comes into its own.

One common example of a classifier function for naturally occurring categories is an email spam filter. The spam filter does not *define* what counts as spam; if it sends a message you wanted to your spam folder, we don't say "Well, it turns out it was spam after all." Rather, the spam filter tries to recognize and label messages that the user won't want. Still, it does this by means of a classifier function. The function takes a (possibly very large) number of features as input. Common ones include the number of instances of various keywords ("viagra," "free," "x-rated," "stock tips"), the number of misspelled words or words in ALL CAPS, and the presence of email attachments. Based on all these input features, the classifier function outputs a label: "spam" or "not spam."

Finally, as we've already mentioned, Shalmaneser is based on a set of classifiers. Each classifier is trained to assign semantic role labels or word sense labels to a word or a small set of words. For instance, there might be a classifier trained to assign word sense labels to individual instances of the word *drop*, and another trained to assign semantic role labels to arguments of the MOTION_DIRECTIONAL frame. Each of these classifiers takes a number of linguistic features as input. (What are those features? See sections 5.2 and 5.3 for details.)

So let's return to our definition of machine learning: "an algorithm that can approximate a function based on examples of its input and output." In the case of spam filters and

City	**Altitude**	**High-altitude recipe needed?**
Albuquerque, NM	4,955	Yes
Billings, MT	3,124	No
Butte, MT	5,549	Yes
Boise, ID	2,730	No
Denver, CO	5,260	Yes
Flagstaff, AZ	6,900	Yes
Las Vegas, NV	2,000	No

Table 4.1: Elevations and brownie recipes for major western cities

Shalmaneser, the function we're trying to approximate is a classifier function, which takes a feature vector as an input and outputs a label. A machine learning algorithm learns to approximate a classification function by looking at **LABELED DATA** — a set of feature vectors with the correct labels already associated with them. (You will also hear these called **TRAINING DATA**, and exposing the machine learning algorithm to them is called **TRAINING**.) The result of this training is an approximate classifier function that can be applied to **UNLABELED DATA** in order to *predict* the correct label.[1]

4.2.2 Training classifiers: a simple example

So how can a classifier function be learned automatically? Let's start with a simple example, a task that can be modeled using a binary function of one variable: deciding whether to use the high-altitude brownie recipe or the regular brownie recipe.

Let's say we've just arrived in Amarillo, Texas, altitude 3,685 feet. We've got two perfectly good brownie recipes — one good for high altitudes where the boiling point is lower due to the low air pressure, and one that's better closer to sea level. There's only one problem: we can't remember what altitude you're supposed to start using the high altitude recipe at. What we do have is experience living (and baking) in the mountains; see Table 4.2 for a summary.

We might reason as follows: "We're at 3,685 feet right now. The closest we've ever lived to this altitude was Billings, at 3,124 feet. And in Billings, the low-altitude recipe worked better. So probably, the low-altitude recipe will work better here too."

This reasoning is a very simple exmple of memory-based learning — what's called the **NEAREST NEIGHBOR** algorithm. The algorithm goes like this: suppose that there is a function mapping inputs (in this case, altitudes) onto outputs (in this case, brownie

[1]You can also apply the resulting classifier function to more labeled data in order to test its performance. See Chapter 7 for more information.

recipes). Take as training data a set of observations of the input and output of that function (in this case, the altitudes and corresponding recipes in Table 4.2). Then, given an input that wasn't in the training data (Amarillo's altitude of 3,685 feet), find the *nearest* input that *was* in the training data, and give the same output.

We can generalize this to the *k*-NEAREST-NEIGHBORS algorithm as follows: rather than taking the single nearest input, take the *k* nearest input for some predetermined value of *k*, and give the output that we observed for the majority of those inputs. The 3-NN solution to the brownie baking task would go as follows: "We're at 3,685 feet right now. The three closest cities we've lived in to this altitude have been Boise (2,730), Billings (3,124) and Albuquerque (4,955). In two of those three cities, Billings and Boise, the low-altitude recipe worked better. So probably, the low-altitude recipe will work better here too."

In theory, we could poll higher numbers of neighbors; in practice, though, there wouldn't be much sense with so few data points. Look at it this way: with seven points of training data, the 7-NN algorithm would always give the same answer — in this case, it would always recommend the high-altitude recipe, since it worked better in four of the seven cities. That's not so useful. But in a real application, with hundreds or thousands of points of training data, higher values of *k* could be appropriate.

4.2.3 Vector space: classifying in two or more dimensions

There's another important difference between the brownie baking task and the real tasks to which machine learning is usually applied. Here, we were only considering a function of a single variable: one mapping altitudes onto brownie recipes. Many applications of machine learning involve modeling a function of dozens or hundreds of variables, and the real strengths of the machine learning approach come out in these multidimensional problems — especially ones where *interactions* between variables are important. It is difficult to visualize examples using more than a few variables (we use computers for these tasks for a reason!) but a few two-variable examples ought to give you an analogy for thinking about the higher-dimensional applications:

It's not just the altitude that affects baking. Temperature and humidity can also play a role. Let's say that we're baking in a house with no air conditioning. If it's cool and moderately humid, making pie crust from scratch is worth the effort. But homemade pie crust is finicky: too humid and it gets soggy; too dry and it gets hard; too hot and the butter melts before it should, ruining the flaky texture. In weather like that, it's easier to save ourselves the frustration and use a frozen crust, even if it won't taste quite as good.

Suppose we're training a classifier to decide whether to use a homemade crust. In this case, our training data are past attempts at pie-making. For each, we have a temperature measurement, a humidity measurement, and a verdict on whether the homemade crust

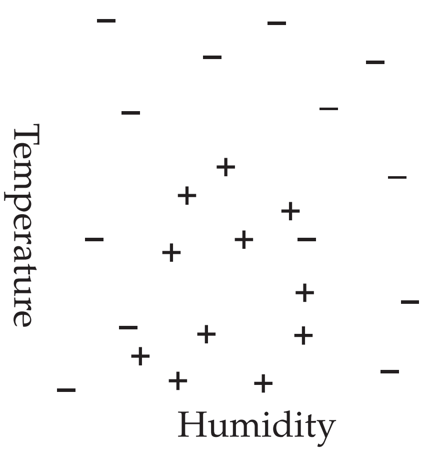


Figure 4.1: Training data

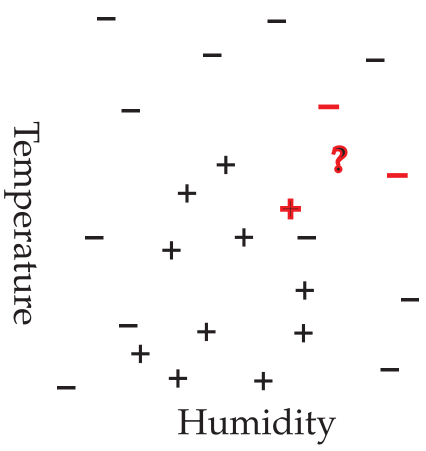


Figure 4.2: A test case: *k*-NN says no

was worth the effort.

As Figure 4.1 shows, we can think of these data points as labeled points in two-dimensional space. One dimension represents temperature, one represents humidity;[2] the label on

[2]Astute readers will have noticed that we've left off the *units* on the axes in Figure 4.1. Especially astute readers may have noticed that there's a larger problem we're sweeping under the rug by doing so: we're comparing incommensurables. The *k*-NN algorithm talks about the distances between data points. Suppose we've got a pair of data points that are 1 °F apart. Are they closer or farther than a pair that differ in relative humidity by 1%? What about a pair with a temperature difference of $\frac{1}{2}$ °F and a humidity difference of $\frac{1}{2}$%?

The answer is, it's up to us — or more properly, up to the particular implementation of *k*-NN that we're working with. Most modern machine learning programs are designed to make reasonable decisions about scaling automatically. The details of how they do this are beyond the scope of this tutorial. For the examples in this chapter, we're assuming that the scaling decisions have already been made, and that they are

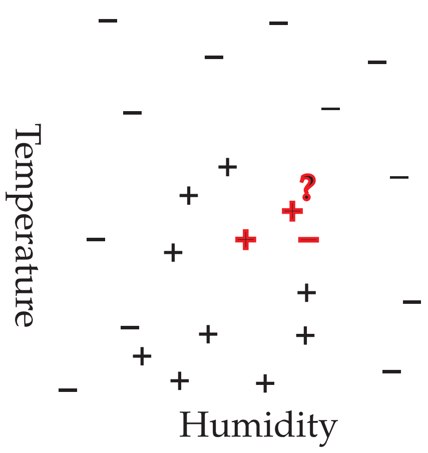


Figure 4.3: Another test case: *k*-NN says yes

each point represents the verdict on the homemade crust, + for 'worth it' and - for 'not worth it.'

Now let's assume as before that there is a function from ⟨*temperature, humidity*⟩ pairs — coordinates in this two-dimensional space — to verdicts on making homemade pie crust. We can use the *k*-NN algorithm to approximate this function for unobserved inputs. Figures 4.2 and 4.3 show two instances of this. The question mark represents the input for which we want to approximate the function's output. The training data points highlighted in red are the three nearest points. In Figure 4.2, the 3-NN algorithm predicts that homemade crust won't be worth it. In Figure 4.3, it predicts that it will.

4.2.4 Non-numeric features

So far we've been discussing non-linguistic examples to get a feel for how machine learning works. Now let's turn to a linguistic example: word sense disambiguation (WSD).

You can probably guess from the last few examples how we'll proceed. As before, we want to approximate a function. The function's input should be a feature vector, where the features encode information about the target word and its context. And the function's output should be a word sense label. So far so good.

But we run into a snag when it comes time to featurize our input. You see, so far the features we've been working with have had numeric values. But the useful features in describing bits of natural language — words, sentences, and so on — are mostly non-numeric. In Chapter 1, we briefly listed some of the linguistic 'clues' that a WSD algorithm

reflected in the figures. A horizontal centimeter of separation between two data points is 'just as far' — and given just as much weight as — a vertical or diagonal centimeter.

might use. These 'clues' are the sorts of things our features should encode:

- the target word;
- the target word's part of speech;
- the target word's subject and object (if it is a verb);
- other words in the vicinity of the target — these are useful for capturing information about the topic of the text in which the target word appears;
- other phrases in the vicinity of the target — does it have a prepositional phrase as a complement? are there any adverbs modifying it? and so on.

Fortunately, most machine learning algorithms — including the ones Shalmaneser relies on — can handle non-numeric features.

At the heart of the *k*-NN algorithm is the idea of distance. If we reduce each data point to a set of numeric features, we can calculate how similar two data points are by measuring the distance between them in multidimensional space. Other machine learning algorithms depend on this same technique of using proximity in vector space to predict similarity.

So what's tricky about non-numeric features is that there's no clearly defined way to measure distance between two values. (How far is `drop` from `climb`? How far is `drop` from `drip`? Is `VP` closer to `NP` or to `S`?) But we can get around this problem by *defining* a **DISTANCE METRIC** for non-numeric values.

One simple example is the **OVERLAP METRIC**. This treats numeric features normally. For non-numeric features, it determines the distance between two values by checking if they are identical. Identical pairs of values have a distance of 0; all other pairs have a distance of 1.

This sort of distance metric opens the door to the use of a number of linguistically useful feature types:

- Boolean values: `true` and `false`. These represent the answers to questions like "Are there any adverbs modifying the target word?"
- Strings of characters, representing words or part-of-speech tags: `NP`, `Adj`, `goldfish`, *etc*. These too can represent the answers to different questions: "What's the target word?" "What part of speech is it?" "What's its subject?" "What's the word before it?" And so on.

- Vectors of words or part-of-speech tags. For instance, we might want a feature that gives the part-of-speech tag of each of the target word's siblings, or one that gives the words immediately before and after the target word.

(There are also some numeric features that are useful for linguistic tasks. Shalmaneser, for instance, uses a few features that give the distance between two nodes in a syntactic tree.)

Ex. 17 You might wonder why we don't just assign numeric labels to all the non-numeric values — for parts of speech, say, convert 'noun' to 1, 'verb' to 2, 'adjective' to 3, and so on. Why is this a bad idea?

Chapter 5

Looking Under the Hood

In the last few chapters, we've treated Shalmaneser as a black box — you input a text, something mysterious happens, and *hey, presto,* out comes a semantic parse. In this chapter, armed with the concepts of **MACHINE LEARNING** and **FEATURIZATION** from Chapter 4, we will look inside the box and see how Shalmaneser does what it does.

On a large scale, Shalmaneser is divided into three modules. In this chapter, we will first introduce the three modules, and then describe each one in detail.

The first module, describes in §5.1, is called the **PREPROCESSOR**, and it handles a number of preliminary tasks that must be taken care of before approaching WSD and SRL. Some of these are basic input/output tasks — reading in text to parse, ensuring it's in the correct format, and so on. Others are lexical or syntactic. Before a text can be semantically parsed, it must be split into words, labeled with lexical information like the root and part-of-speech of each word, and parsed syntactically. We have not considered these tasks so far because Shalmaneser does not perform them itself — it calls on third-party utilities to perform them. But if you're going to be retraining Shalmaneser or running experiments on it, it's helpful to understand how the preprocessor behaves.

The second module, described in §5.2, handles WSD. If we are parsing text, the WSD module takes the parsed sentences output by the preprocessor and featurizes them. It passes the features it has extracted to a machine learning system, which has been trained to predict the correct word sense labels, and then it takes those predicted word sense labels and attaches them to each sentence.

The third module, described in §5.3, handles SRL, and it works much like the second. If we are parsing text, it takes the parsed sentences with word sense labels and extracts another set of features, ones that are useful for SRL. It passes those features to a machine learning system which has been trained to predict the correct semantic role labels, and it takes those predicted semantic role labels and attaches them to each sentence.

5.1 Preprocessing

Each of these tasks is performed by a separate tool — TreeTagger for lemmatization and part-of-speech tagging, the Collins parser for syntactic parsing. See Appendix B for resources giving more information on these tools.

5.1.1 Lemmatization

Lemmatization is the process of finding a **LEMMA** for each word in a piece of text. Intuitively, you can think of a word's lemma as being the dictionary entry you would find it under. The lemma of a noun is its singular form (*cats* is found under the lemma *cat*). The lemma of a verb is its infinitive (*walks* is found under the lemma *walk*; *am* and *were* are found under the lemma *be*). The lemma of an adjective is its positive form — that is, the form that is not comparative or superlative, so *taller* and *tallest* are found under the lemma *tall*.

For most English words, the lemma will be the same as the word's **STEM**. So for instance, *cats* is morphologically *cat-s*, consisting of the suffix *-s* on the stem *cat*. Similarly, *tall-er* and *tall-est* are built on the stem *tall*. But for some irregular words, there is a difference.[1] For instance, *better* and *best* fall under the lemma *good*, but they are not built on the same stem as *good*. Words like this, whose inflected forms are not all built on the same stem as their lemma, are called **SUPPLETIVE**. Another example is the paradigm for *be*, which includes a number of suppletive forms: *am*, *are*, *is*, *was* and *were*, none of them built on the same stem as *be*.

Because Shalmaneser **LEMMATIZES** words rather than **STEMMING** them, it will still label *am* with the lemma *be*, even though they do not have a stem in common. This will be something to keep in mind in later chapters — especially in Chapter 8, where we will learn how to customize Shalmaneser's behavior. We may want to specify that, say, all the forms of the verb *be* should be treated in a certain way, and lemmatization will give us a convenient way to do that.

In any case, the first thing Shalmaneser does when preprocessing a text is to lemmatize it. By default, it uses TreeTagger for this. The sentence in 5.1 will look like 5.2 after lemmatization with TreeTagger.

(5.1) Creeping in its shadow I reached a point whence I could look straight through the uncurtained window.

(5.2)
```
WORD:   Creeping  in  its  shadow  I  reached  a  point  whence
LEMMA:  creep     in  it   shadow  I  reach    a  point  whence
```

[1]Note that only some irregular words are suppletive. *Oxen*, for instance, has the irregular plural suffix *-en* but is still built on the same stem as the singular form *ox*.

```
I   could   look   straight   through   the   uncurtained   window   .
I   could   look   straight   through   the   <unknown>     window   .
```

5.1.2 Part-of-speech tagging

In **PART-OF-SPEECH TAGGING** (**POS TAGGING** for short), each word is marked with a short string that represents its grammatical category. Various **TAGSETS** exist, and different taggers use different sets. For convenience, Shalmaneser converts part of speech tags into a single, simplified tagset, shown in Table 5.1.2. Currently, it uses TreeTagger for POS tagging as well as for lemmatization, but it may be extended in the future to work with other taggers, so it is useful to have a standardized tagset that is independent of the particular tagger used.

After a sentence has been lemmatized, it will be tagged by TreeTagger, and then it will have its tags converted into the simplified tags in Table 5.1.2.

5.3 shows the sentence from 5.1 as it would look after the part-of-speech tags given by TreeTagger have been imported into Shalmaneser.

(5.3)

```
WORD:    Creeping   in     its   shadow   I      reached   a     point
LEMMA:   creep      in     it    shadow   I      reach     a     point
CAT:     verb       prep   det   noun     prep   verb      det   noun

whence   I      could   look   straight   through   the   uncurtained
whence   I      could   look   straight   through   the   <unknown>
adv      prep   verb    verb   adverb     prep      det   adj

window   .
window   .
noun     pun
```

5.1.3 Syntactic parsing

A **PARSE TREE** is a tree structure representing the phrase structure of a sentence. Each terminal node in the tree corresponds to a word, and nonterminal nodes correspond to phrases.

SYNTACTIC PARSING is the task of converting a tagged and lemmatized sentence into a parse tree. By default, Shalmaneser uses the Collins parser for this task. Just as there were various sets of word-level tags used in POS tagging, there are various sets of phrase-level tags used in syntactic parsing. Once again, though, Shalmaneser converts the phrase tags output by the Collins parser into the simplified tagset in Table 5.1.

The Collins parser also gives us one bit of information that goes above and beyond what's

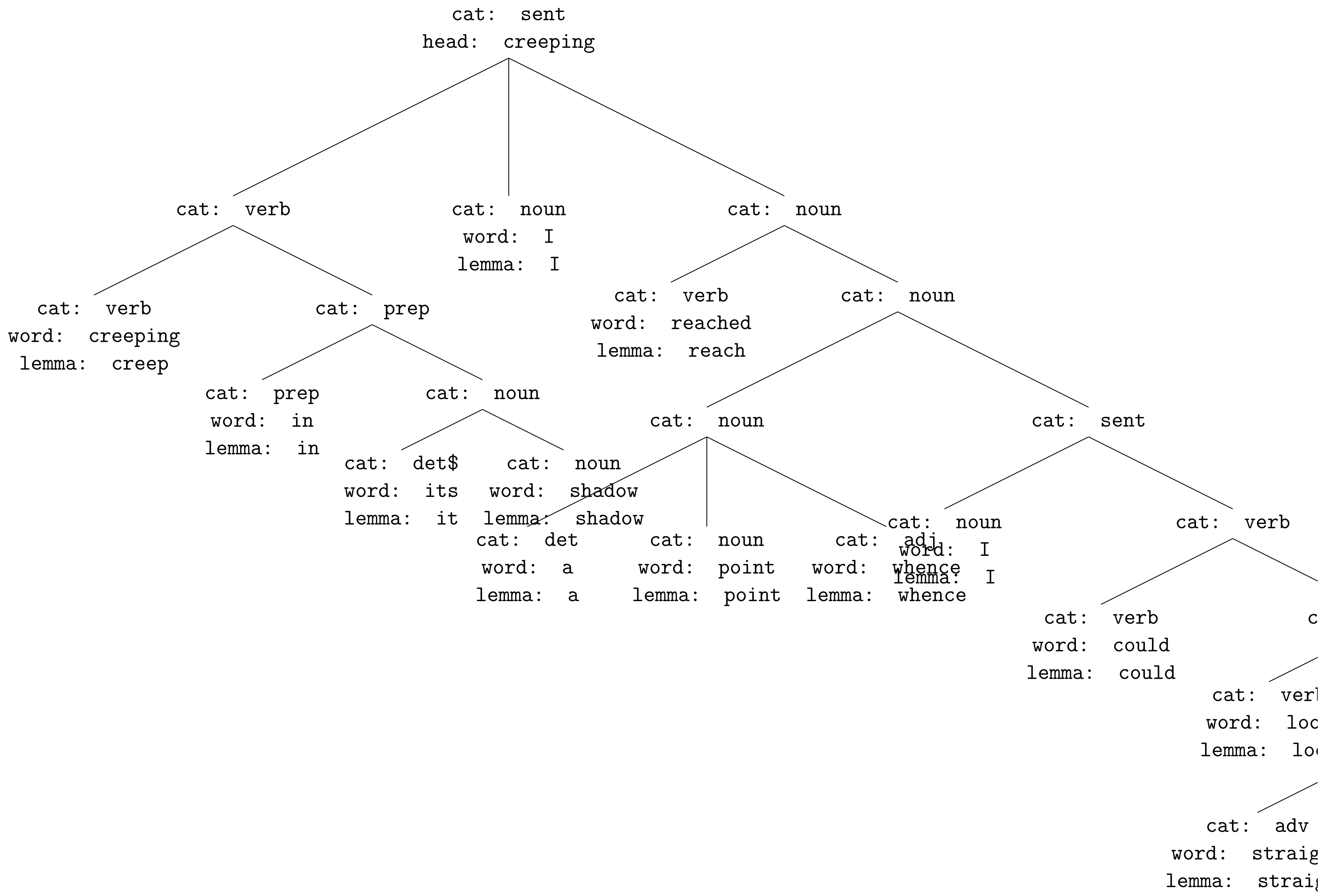
Tag	Meaning	Examples
`adj`	adjectives	*red apples, the tall trees*
	adjective phrases	*a really bright light*
`adv`	adverbs	*quickly drops down*
	adverb phrases	*drops not so quickly*
`card`	numbers	*47, the three musketeers*
`con`	conjunctions	*and, because, although*
`det`	determiners	*the cat, an apple*
	possessive pronouns	*my cat*
	demonstrative pronouns	*those apples*
`for`	foreign-language material	*je ne sais quoi*
`noun`	nouns	*a comfortable chair, the road*
	noun phrases	*blue moon, bend in the road*
	personal pronouns	*him, they*
	expletive *there*	*there was a problem*
`part`	particles	*blown up, sold out, tied down*
`prep`	propositions	*to the store, around the corner*
	preposition phrases	*went to the store*
`sent`	complete sentences	*The cat sat on the mat.*
`verb`	verbs	*go, said, drops*
	verb phrases	*go to the store, drops the ball*
`top`	top node of a tree	
`pun`	punctuation	
`nil`	"something went wrong"	

Table 5.1: The simplified tagset used by Shalmaneser for part of speech and phrase type tags

in the Penn Treebank format for trees. For any nonterminal node with multiple children, it picks out one terminal node descended from it as the head. The parser has syntactic criteria for determining the head of any phrase type. For instance, the head of a verb phrase is its main verb; the head of a sentence is the main verb of its main clause; the head of a simple noun phrase is the noun, and the head of a complex noun phrase is the rightmost noun; the head of a prepositional phrase is the preposition; and so on.

Figure 8.2 shows 5.1 after it has been lemmatized, tagged and parsed. Note that the terminal nodes have the same `cat`, `lemma` and `word` attributes as before. What's been added is the tree structure itself, and the `cat` and `head` attributes of the nonterminal nodes.

(5.4)


cat: sent
head: creeping
cat: verb
cat: noun
word: I
lemma: I
cat: noun
cat: verb
word: creeping
lemma: creep
cat: prep
cat: verb
word: reached
lemma: reach
cat: noun
cat: prep
word: in
lemma: in
cat: noun
cat: noun
cat: sent
cat: det$
word: its
lemma: it
cat: noun
word: shadow
lemma: shadow
cat: det
word: a
lemma: a
cat: noun
word: point
lemma: point
cat: adj
word: whence
lemma: whence
cat: noun
word: I
lemma: I
cat: verb
cat: verb
word: could
lemma: could
cat: adv
word: straig
lemma: strai

5.2 WSD

Shalmaneser does WSD using a set of classifiers. It comes with one set already trained and allows you to train more. Each classifier handles one of the subtasks involved in WSD. (In fact there are a few ways to split the WSD task into subtasks — more on that in a moment.)

The WSD module's job is to produce training data and input for these classifiers. If you are training a new set of classifiers, it takes the labeled training data (in the form of syntax trees with frame labels) converts them into labeled feature vectors, and sends each vector to the appropriate classifier to train itself on. If you are applying a trained set of classifiers to unlabeled data, it takes syntax trees, converts them into unlabeled feature vectors, and sends each one to the appropriate classifier as input; it then collects the label which the classifier outputs.

As we saw in Part I, each frame in the FrameNet inventory carries a list of words that can evoke it. Any instance of any word that appears on one or more of these lists is given a frame label by WSD. (A word that only appears on one list does not pose much of a challenge — there is, strictly speaking, no disambiguation to do. But the frame label will be needed by the SRL module in the next processing step, so WSD applies it.

That means that one sentence can be converted by WSD into more than one feature vector. 5.4, for instance, becomes three feature vectors, one for each frame-bearing word.

Once the feature vectors are compiled, they are sent to a machine learning tool, either to be used for training or to be labeled. Now it is time to consider the division into subtasks, for a different classifier will be trained (or consulted) for each subtask.

There are two ways that WSD can be subdivided: into a smaller set of non-binary tasks, or into a larger set of binary tasks. If you do WSD as a set of non-binary tasks, you train one classifier for each word. That classifier's task is to pick one of the *n* possible word sense labels. So for instance, a single classifier would be trained for *reach*, with the task of choosing between four possible labels: CHANGE_POSITION_ON_A_SCALE, CONTACTING, ARRIVING and PATH_SHAPE. Any feature vector based on an instance of the word *reach* will be sent to that single classifier.

If you do WSD as a set of binary subtasks, you train a classifier for each *sense* of each word. That classifier's task is to decide, for each occurrence of the word: "Does 'my' sense apply here or not?" So for instance, four classifiers would be trained for *reach*: one to decide whether CHANGE_POSITION_ON_A_SCALE applies, one to decide whether CONTACTING applies, one to decide whether ARRIVING applies and one to decide whether PATH_SHAPE applies. Labeled feature vectors used for training will only be sent to the appropriate classifier. (A vector based on an instance of *reach* with the ARRIVING label will only be sent to the ARRIVING classifier, and so on.) Unlabeled feature vectors will be sent to each classifier in turn until one returns `true`.

When you train a set of classifiers, you choose one of these subdivisions. Shalmaneser will remember your choice and send the correct feature vector as input to the correct classifier when you apply them.

5.2.1 Features for WSD

Shalmaneser does not use terribly many features for word sense disambiguation. This reflects the WSD literature, in which only a few useful features have been identified. Fortunately, those few features are very useful indeed, and are widely used with good results.

Context The `context` feature returns the n words before and after the current word, for some value of n.

Different sized context windows capture different sorts of information. A large window picks up information about the general subject matter of a passage. A very small window — perhaps two or three words in each direction — is useful for catching collocations. It's common to train a classifier using several different-sized context windows. Using, for instance, one window of size 3 and one of size 50 will produce a classifier that is sensitive to subject matter but still capable of recognizing collocations.

A context window can cross sentence boundaries. This is often useful, since one sentence can contain clues to the subject matter of other sentences nearby.

Syntax The `syntax` feature captures several pieces of information about the current word's location in the syntactic tree:

- the `cat` attribute of the current node,
- the `cat` attributes of the current node's children (if it has any), and
- the `cat` attributes of the current node's siblings (if it has any).

Synsem The `synsem` feature captures all of the information in the `syntax` feature, plus lexical information about each node in the form of it's headword's lemma. In other words, it returns:

- the `cat` attribute of the current node and the `lemma` attribute of its headword,
- the `cat` attributes of the current node's children (if it has any) and the `lemma` attribtues of their headwords, and

- the `cat` attributes of the current node's siblings (if it has any) and the `lemma` attributes of their headwords.

5.3 SRL

SRL is more or less like WSD, but with a few additional complications. It takes in syntax trees (this time, trees *with* frame labels of the sort generated by WSD) and produces feature vectors (this time using a much wider range of features — perhaps including user-defined features: see Chapter 8). If you are training a new set of classifiers, it sends the vectors to the machine learning tool as training data. If you are applying a trained set of clasifiers, it sends them to the classifiers as input and collects the output.

Like WSD, SRL divides its work into subtasks. And as in WSD, there are several different ways to make the division. The top-level division is always the same:

- **ARGUMENT RECOGNITION** (ARGREC) looks at each frame evoked by the sentence and determines which words or phrases play a semantic role.
- **ARGUMENT LABELING** (ARGLAB) looks at each word or phrase that was determined to play a semantic role in a frame, and determines *which* role it plays.

But there are several ways it can be subdivided from there:

1. By **frame**: For each frame, only one pair of classifiers is trained. So for instance, there would be just one ARGREC and just one ARGLAB classifier to cover all the instances of the ARRIVING frame. These would be used to identify and label the arguments of *reach* in 5.4, and also to identify and label the arguments of other words like *arrival*, *influx*, *come* and *enter* in other sentences where they appear.
2. By **part of speech**: For each frame, a separate pair of classifiers is trained for every part of speech. So for instance, there would be one ARGREC and one ARGLAB classifier for all the *verbs* that evoke the ARRIVING frame: *reach*, *come*, *enter*, and so on. There would be a different ARGREC classifier and a different ARGLAB classifier for all the *nouns* that evoke it: *arrival*, *influx*, and so on.
3. By **lemma**: For each frame, a separate pair of classifiers is trained for every word lemma. So there would be one ARGREC and one ARGLAB

classifier just for *reach* (and its inflected forms *reached*, *reaches* and *reaching*). There would be another pair for *come*, another for *arrival*, another for *enter*, and so on.

However you subdivide the task, the ARGREC classifiers will always be binary (they answer the yes/no question “Is this phrase an argument of that frame-bearing word?”) and the ARGLAB classifiers will always be non-binary (they answer the open-ended question “Which of these many semantic role labels is the right one for this phrase?”)

5.3.1 Features for SRL

A much wider range of features is available for SRL — and, as we’ll see in Chapter 8, its possible to define new features for SRL classifiers to use. Again, this reflects the literature on semantic role labeling. Many different features are used for this task, and reserachers often find new features that can be usefully applied to it.

5.3.1.1 Current and target node

All of these features are defined in terms of the notion of a **CURRENT** and **TARGET NODE**.

To understand these, let’s look again at the process Shalmaneser must go through in completing an argument recognition task. Suppose it’s working on 8.2; it’s already assigned *creeping* to the SELF_MOTION frame, and it’s trying to predict which nodes, if any, should be assigned to this frame as arguments. We can break this task down into a whole series of subtasks. Shalmaneser must predict. . .

- whether the noun *shadow* is an argument of the frame evoked by *creeping*,
- whether the posessive pronoun *its* is an argument of the frame evoked by *creeping*,
- whether the noun phrase *its shadow* is an argument of the frame evoked by *creeping*,

and so on for every node in the tree. In a longer sentence, with more than one constituent that evokes a frame, this series of subtasks will need to be repeated for every frame evoked. We can sum this up as follows: Shalmaneser must predict. . .

- whether **current node** is an argument of the frame evoked by **target node** for every possible value of **current node** and **target node**. Obviously, for some values the correct prediction will be *yes* and for some it will be *no*. In order to generate the correct predictions, we'll need features whose values vary depending on the **current node** and **target node** under consideration.[2]

Of the built-in features Shalmaneser uses for SRL, some return information about the current node, some return information about the target node, and some return information about their relationship to one another.

The remainder of this section lists and defines those features. The list is rather long, and contains some technical details that are not essential to understanding the next few chapters. You may want to skim through it for now, and return to it later if you have specific questions about what a particular feature does.

5.3.1.2 Features of the current node

- `pt` (for "phrase type") returns the `cat` attribute of the current node. The `pt` of *straight through the uncurtained window* is `prep`.
- `father-pt` returns the `cat` attribute of the current node's *parent*. The `father-pt` of *straight through the uncurtained window* is `verb`.
- `prep` (for "preposition") returns `nil` unless the current node is a prepositional phrase. If the current node is a prepositional phrase, it returns the preposition it's headed by. The `prep` of *straight through the uncurtained window* is `through`.
- `const_head` returns the lemma of the headword of the current node. The `const_head` of *straight through the uncurtained window* is `through`.
- `const_head_pos` returns the part of speech the headword of the current node. The `const_head_pos` of *straight through the uncurtained window* is `prep`.
- `firstword` returns the `lemma` and `cat` of the leftmost node descended from the current one. In other words, if the current node represents a phrase, it will return the `lemma` and `cat` of the first word of the phrase; if it represents a single word, it will return the `lemma` and `cat` of that word. The `firstword` of *straight through the uncurtained window* is `straight adv`.

[2] We've used argument recognition as an example here, but the same will turn out to be true of argument labeling: each semantic role label will be appropriate for some current/target pairs, and inappropriate for others. For instance, if the target node is *buy* and the current node is the noun phrase *fifty-seven dollars*, MONEY might be an appropriate label. But if the current node is the noun phrase *Crazy Jim's discount auto*, SELLER is probably more appropriate.

- `lastword` returns the `lemma` and `cat` of the rightmost node descended from the current one. In other words, if the current node represents a phrase, it will return the `lemma` and `cat` of the last word of the phrase; if it represents a single word, it will return the `lemma` and `cat` of that word. The `lastword` of *straight through the uncurtained window* is `window noun`.

- `leftsib` returns the `lemma` and `cat` the current node's left hand sibling, if it has one. The `leftsib` of *straight through the uncurtained window* is `look verb`.

- `rightsib` returns the `lemma` and `cat` the current node's right hand sibling, if it has one. The `rightsib` of *straight through the uncurtained window* is `nil`.

- `icont_node` returns the `cat` and `lemma` of the current node's **INFORMATIVE CONTENT NODE**, defined as follows:

  - If the current node is a prepositional phrase, its informative content node is the head of the noun phrase it contains.
  - If the current node is a clause, its informative content node is the head of the verb phrase it contains.
  - If the current node is a verb phrase, and it contains another verb phrase embedded in it, its informative content node is the head of that embedded verb phrase.

  The informative content node of *straight through the uncurtained window* is *window*, and the `icont_node` feature will return `noun window`.

- `ismaxproj` returns `true` if the current node is a **MAXIMAL PROJECTION**, and `false` otherwise. A maximal projection is defined here as being a node whose parent is not of the same type. So for instance, a `verb` whose parent is another `verb` is not a maximal projection; a `verb` whose parent is a `sent` is. The `ismaxproj` of *straight through the uncurtained window* is `true`.

5.3.1.3 Features of the target node

- `frame` returns the frame that has been assigned to the target word. The `frame` of *look* is `perception_active`.

- `target` returns the `lemma` attribute of the target word. The `target` of *look* is `look`.

- `target_pos` returns the `cat` attribute of the target word. The `target_pos` of *look* is `verb`.
- `target_voice` returns nothing unless the target node is a verb. If it is a verb, `target_voice` returns Shalmaneser's best guess as to its voice, `active` or `passive`, as determined by its position in the syntactic tree. The `voice` of *look* is `active`.
- `gov_verb` returns the `lemma` attribute of the verb governing the target word. The `gov_verb` of *window* is `look`.

5.3.1.4 Features relating the current node to the target

Many of these features rely in some way on the notion of a **LEAST COMMON ANCESTOR** (or **LCA**). If A and B are nodes, their LCA is the unique node that

1. is an ancestor of both A and B, and
2. has no children that are ancestors of both A and B.

One feature is based entirely on the current and target nodes' LCA:

- `ancestor_rule` returns the `cat` attribute of the current and target nodes' LCA, and the `cat` attributes of all its children. The `ancestor_rule` of *I* and *look* is `I -> verb prep verb`.

Several more depend on the derived notion of a **PATH** between two nodes. To find the path between A and B, first find their LCA. Then, imagine climbing through the tree from A to the LCA, naming every node you pass along the way. Then, climb back down from the LCA to B, again naming every node you pass.

Here, for instance, is the path from *I* to *look* in figure N.

illustration

- `pt_path` (for "phrase type path") returns the phrase type of every node on the path from the current word to the target. If the current word is *I* and the target is *look*, `pt_path` will return `prep sent verb verb verb`.
- `pt_combined_path` returns everything that's returned by `pt_path`. It also returns the target's part of speech, and if the target is a verb it returns that verb's voice. If the current word is *point* and the target is *look*, `pt_combined_path` will return `prep sent verb verb verb active`.

- `pt_partial_path` returns the phrase type of every node on the path from the target to the LCA of the current node and target. (To put it another way, it tells you about the nodes on the ascending leg of the path, but not the nodes on the descending leg.) If the current word is *I* and the target is *look*, `pt_partial_path` will return `sent verb verb verb`.

- `pt_gvpath` finds the verb governing the target node, then finds the path from that verb to the current node, and returns the phrase type of every node on that path. If the current word is *I* and the target is *look*, then the target's governing verb is *could*, and `pt_gvpath` will return `verb verb verb verb`.

- `path_length` returns the number of nodes on the path from the current word to the target. If the current word is *I* and the target is *look*, `path_length` will return `5`.

You can also relate the current node to the target without relying on the notion of a path. Shalmaneser comes with three features that do this:

- `relpos` returns `left` if the current node is to the left of the target in the tree, `right` if it's to the right of the target, and `including` if it is one of the target's ancestors.

- `worddistance` returns the number of words intervening between the current and target nodes. If the current node is *I* and the target is *look*, `worddistance` will return `1`.

- `nearest_node` returns `false` if there are terminal nodes intervening between the current and target nodes, and `true` if there aren't. (In other words, it answers the question, *Is the current node as near as possible to the target?*)

5.3.2 More options

There are a few extra techniques that can be used to optimize the training of SRL classifiers. These have been developed to cope with some aspects of the SRL task that make it more difficult than WSD.

Xue/Palmer pruning First of all, a set of SRL classifiers must make more decisions per sentence than a set of WSD classifiers. This is especially true when it comes to the argument recognition portion of the SRL task, in which every constituent of every sentence must be evaluated as a possible argument of every frame-bearing word. In 5.4, there are 28 constituents and 3 frame-bearing words, for a total of $28 * 3 = 84$ decisions that the argument recognition classifiers must make. (In contrast, the WSD classifiers only had to make three decisions on this sentence, one for each frame-bearing word.) Using Xue and Palmer's pruning algorithm, described below, we can cut down dramatically on the number of argument recognition decisions that need to be made. This makes Shalmaneser more efficient on the SRL task, but in some cases it may lead to reduced accuracy.

The idea behind Xue and Palmer's algorithm is as follows: So far, we've been treating every word and phrase in a sentence as a candidate argument in every frame. But in fact, very few of them are *realistic* candidates. For instance, the approach we've been taking requires us to ask whether *window* is an argument of *creep*. But anyone familiar with English syntax can tell, just by looking at the syntax tree, that the answer is "no." The two nodes are part of completely different phrases. They don't have the right sort of syntactic relationship to make a close semantic relationship possible.

If you choose to train a set of classifiers using Xue/Palmer pruning, two things happen. First, the argument recognition classifiers are trained on a reduced set of training data. They are not given *all* of the candidate arguments from the labeled data to train on; they are just given the ones that the Xue/Palmer algorithm considers realistic candidates. (For instance, the classifiers would never be exposed to the fact that *window* is not an argument of *creep* in 5.4.) And second, when the argument recognition classifiers are applied to an unlabeled sentence, they are not asked to make predictions about all the candidate arguments. They are just asked to make predictions about the ones that the Xue/Palmer algorithm considers realistic. (For instance, the classifiers would never be asked whether *window* was an argument of *creep* in 5.4.)

This makes it faster to train new SRL classifiers, and faster to apply the ones we've trained. It also eliminates some simple errors. Without Xue/Palmer pruning, Shalmaneser could wind up labeling *window* as an argument of *creep* in 5.4 — it would be an error, of course, but it's not an impossible error to make. With Xue/Palmer pruning, that error isn't even an option. On the other hand, pruning can sometimes reduce accuracy in other ways. For instance, it reduces the amount of training data that argument recognition classifiers are exposed to, and exposure to less training data can lead to less accurate predictions. Shalmaneser lets you choose whether or not to use Xue/Palmer pruning when you train an SRL classifier. In some cases, you may want to try it both ways and see which way works better.

Data correction The SRL task is also sensitive to errors in other tasks upstream. If the SRL classifiers are given training data that contain parser errors — as often happens when the parsing has been done automatically — they will learn to recreate those errors.

One common instance of this is found in Figure 5.1. Here we have a sentence where a human annotator has correctly applied the STIMULUS role label to the words *this delightful,* *animated musical*, but the automatic parser has incorrectly concluded that the word *will* belongs in the NP along with them. This leads to the STIMULUS label being associated with some, but not all, of the words in the NP.

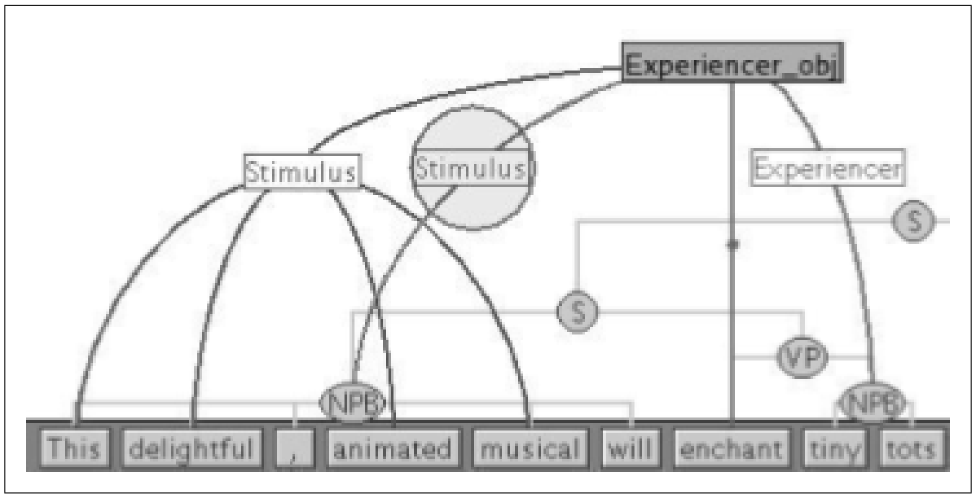


Figure 5.1: A syntax/semantics mismatch

If we give this sentence to Shalmaneser as training data, the SRL classifiers will learn to replicate this mistake, applying the STIMULUS role label to individual words in an NP rather than (correctly) applying it to the NP as a whole. Fortunately, it is often possible to correct this sort of error automatically:

1. If all the children of a node n have role label r, we assign the label r to n instead.
2. If *all but one* of the children of a node n have role label r, we assign the label r to n instead. (This step corrects the error in Figure 5.1: all but one child of the NP has the STIMULUS label, so we apply the label to the whole NP instead.)
3. If a node n has a noun and a relative clause as its children, and only the noun has been given role label r, we assign the label r to n instead.
4. Finally, we repeat the above steps until we find ourselves making no more changes.

This process allows labels that are scattered across a number of child nodes to coalesce onto a single parent node.

If we apply this data correction algorithm to all our labeled sentences before giving them to the SRL classifiers as training data, we will reduce the odds that Shalmaneser will learn to scatter role labels across a number of child nodes like in Figure 5.1, and increase the odds that it will learn to apply each role label to a single parent node instead. So you can determine whether this data correction improves performance, Shalmaneser lets you choose whether or not to apply it.

Training on gold word sense labels There is one last choice to be made when training a set of SRL classifiers. Often, our training data will have not one but two sets of word sense labels — on the one hand, we'll have the gold standard labels applied by a human annotator, and on the other hand, we'll have the labels that our WSD classifier would have predicted on the same data. We must choose which set of labels to train the SRL classifiers on.

There are possible advantages to each approach. In actual use, any set of SRL classifiers that we train is likely to be used in conjunction with a set of WSD classifiers. Given that, it can make sense to train the SRL classifiers on those WSD classifiers' output — essentially teaching it to adapt to the mistakes that those WSD classifiers are likely to make. On the other hand, some WSD classifier errors lead to impossible combinations of labels. For instance, suppose (TK FIGURE) that our WSD classifiers have mistakenly given the word *take* the REMOVING frame label instead of INGESTION. Well, the REMOVING frame doesn't have an element called *ingestibles* — that combination of frame and role is simply nonsensical, and we wouldn't want to train Shalmaneser to reproduce it. Shalmaneser lets you choose whether to train on the gold word sense labels or on the predicted labels given by a set of WSD classifiers.

Chapter 6

How to train new classifiers

Now we have seen enough of how Shalmaneser works that we can begin training new classifiers. There is a great deal of flexibility in the training process; most of this chapter describes various settings that you can tweak in order to get the greatest possible accuracy.

Training new classifiers also requires us to interact with Shalmaneser's modules independently. This means there is no one-click way to train a classifier, as there is to parse a sentence (see Chapter 3). Rather, it is necessarily a several-stage process:

First off, you cannot train classifiers without a set of training data. To make a set of training data available, you must import it by running it through the preprocessor. This process is described in §6.1. You can import as many sets of data as you want — either for training purposes, or, as we'll see in Chapter 7, for development and testing.

Once you have at least one set of data imported, you can begin training classifiers based on it. You can train WSD classifiers, as described in §6.2, or SRL classifiers, as described in §6.3, or both. You can train as many classifiers as you like off of the same set of training data — or, conversely, you can use the same training settings with different sets of training data. (Chapter 7 shows how to evaluate and compare the results of these different combinations.)

6.1 How to preprocess a data set

In the **Preprocess** tab, you specify the location, language and format of the corpus file or files you're using. This shouldn't require you to make any decisions — just report the necessary information.

Also in this tab, you can specify which preprocessing steps need to be performed on the corpus. Up to three steps can be performed: lemmatization, part-of-speech tagging and syntactic parsing. Here there is a decision to be made, albeit a fairly straightforward one.

Basically, for each step, there are three possibilities.

- Your data set has not been through that step. (For instance, you are working with a corpus with no part-of-speech tags, or no lemma information.) In that case, you need to have Shalmaneser perform the step as part of preprocessing.
- Your data set has been through that step, using the same processing tool that Shalmaneser uses. (For instance, you are working with a corpus that's been lemmatized and part-of-speech tagged by TreeTagger.) In that case, do not repeat the step — you can omit it from preprocessing.
- Your data set has been through that step — but it was done automatically with a different tool than Shalmaneser uses. In that case, *do* perform the step, so that your data will be re-tagged or re-parsed in the format Shalmaneser is familiar with.

To set up Shalmaneser to preprocess a data set, do the following:

1. Select the **Preprocess** tab.
2. Using the controls in the **Preprocess** tab, specify how you want preprocessing to be performed. The following paragraphs explain in detail how to do this.
3. Press **Go** to begin preprocessing.

Here is what the controls in the **Preprocess** tab are used for:

Description In this text field, you should enter a descriptive name for the data set. This is the name you will use to select this data set later on; make it something memorable.

Text to parse The controls in this box let you enter the text, or select the text file or files, that will make up your data set.

To use text from a single file:

1. Select **File** from the pull-down menu.
2. Select the file to parse. There are two ways to do this:

   - If you know the path to the file, you can enter it directly in the text box provided. You might, for instance, enter: `/groups/corpora/gutenberg/treas11.txt`
   - Otherwise, press **Browse** and select a file from the dialog box that appears.

To use text from a directory of files:

1. Select **Directory** from the pull-down menu.
2. Select the directory of files to parse. As with single files, there are two ways to do this:
   - If you know the path to the directory, you can enter it directly in the text box provided.
   - Otherwise, press **Browse** and select a directory from the dialog box that appears.

To enter the text yourself that will make up your data set:

1. Select **Text** from the pull-down menu.
2. Enter the text in the text box.

It takes a lot of text to make a good data set. You will probably not want to type that much text yourself if you can help it. More likely, you will find yourself cut-and-pasting text from another source. The keyboard shortcuts **Ctrl-X**, **Ctrl-C** and **Ctrl-V** work for cut, copy and paste respectively.

Format In this box are four pull-down menus. If you've chosen a file or a directory of files as the source of your data, you must use these menus to specify the files' format.

In the **Language** menu are two options: **English** and **German**. Currently, Shalmaneser only supports English, though we have plans to support German in the future. For now, choose **English**.

In the **Encoding** menu you must specify the character encoding used in your file(s). There are three options:

- Choose **ISO** if your source file(s) use the ISO-Latin-1 encoding.
- Choose **UTF-8** if your source file(s) uses the UTF-8 Unicode encoding. (Other Unicode encodings are not supported.)

- Choose **Hex** ???

In the **Format** menu you must specify the annotation format (if any) used in your file(s).

- Choose **Plain text** if your source file(s) are unannotated — that is, if they consist simply of English text with no frame or semantic role information. Files like this are not useable for training, but you may still wish to apply classifiers you have trained to them.
- Otherwise, choose the name of the annotation format used in your source file(s). The supported options are **SalsaTigerXML**, **FNXml** (the format that the FrameNet project uses), **SalsaTab** and **BNC**.

In the **Origin** menu you may specify an annotation format

Steps In this box are three pull-down menus which allow you to specify which preprocessing steps should be performed, and which tools they should be performed with.

At the moment, Treetagger is the only tool that Shalmaneser can use for lemmatizing and POS-tagging English text. In the **Lemmatizer** menu, choose **Treetagger** if your text needs to be lemmatized and **None** if it does not. Likewise, in the **POS-Tagger** menu, choose **Treetagger** if your text needs to be POS-tagged and **None** if it does not.

There are two parsers Shalmaneser can use for English text. The first, the Collins parser, produces phrase-structure trees of the sort we've been seeing in examples so far. The second, Minipar, is a dependency parser. In the **Parser** menu, choose **Collins** or **Minipar** if you want to use one of those parsers, or **None** if your text does not need to be parsed.

Parser In this area are two text fields: **Split input into files of # sentences/file** and **Truncate sentences longer than**. These let you exercise some control over the treatment of large data sets and long sentences respectively.

By default, Shalmaneser produces a *single* data set each time the preprocessor is run, no matter how many files the data is drawn from. You may wish to generate multiple data sets from a single file or directory of files — for instance, in order to create separate sets of training and test data as described in the next chapter. To split your data in to several sets of *n* sentences each, enter the appropriate value of *n* in the first text box. If you do not wish to split the data set, leave this field blank.

The amount of computing time it takes to parse a sentence increases rapidly as the sentence gets longer. Very long sentences may require a prohibitive amount of time to parse, and so it can be worthwhile to simply truncate the longest sentences. This reduces accuracy by a small amount — because the truncated sentences will almost certainly be parsed incorrectly — but can save a great deal of time. To truncate sentences longer than *n* words, enter the appropriate value of *n* in the second text box.

6.2 How to train WSD classifiers

The broad steps in training a new set of WSD classifiers are as follows:

1. Select the **Train** tab.
2. Within the **Train** tab, select the **Word Sense Disambiguation** tab.
3. Using the controls, specify how you want your new classifier to be trained. More on this below.
4. Press **Go**.

The WSD training controls are divided among five areas according to their function.

Description In the text field labeled **Description**, enter a descriptive name for your WSD classifier. In the **Preprocessed Input** pull-down menu, select the data set you want to train this new classifier on.

Training Corpus These controls are only necessary if you are training a WSD classifier using the context window feature. A large enough context window can cross a sentence boundary and include words from neighboring sentences. So if your data set is made up of noncontiguous parts drawn from a longer connected text, Shalmaneser will need access to the full text to correctly generate large context window features.

If this is the case:

1. Select **Part of a larger corpus**.
2. Indicate the location of the larger corpus. As usual, you can do this either by entering the path directly (in the field marked **Path to larger corpus**) or by pressing **Browse** and choosing it from the dialog box.
3. From the pull-down menus labeled **Larger corpus format** and **Larger corpus encoding**, select the appropriate options. See under **Format** in the previous section for a description of these options.

Tools In this area is a single pulldown menu, from which you can choose the machine learning toolkit you want to use for the classifiers you are training. Currently, the available options are **Timbl** and **Maxent**.

Features In this area you can select which features Shalmaneser will use for training. The available features are described in §5.2.1.

It is possible to use several context windows of different sizes. To begin using a context window feature:

1. Select the size of the window by using the arrow buttons or by entering a number in the field.
2. Press **Add context window feature**.

A list of context window features appears to the left of these controls. It starts out empty, but each context window feature you add will appear in it.

To begin using one of the other features, select the appropriate check box: **Subcategorization features** or **Combined subcat + headword features**.

Feature settings This area contains several pulldown menus that let you control some of the details of how Shalmaneser handles WSD.

If a word has three or more senses, there are two ways to treat the WSD task for that word:

- We can treat it as a single non-binary decision. That is, we can have Shalmaneser ask the question *Which sense does this instance of the word have, sense A, sense B or sense C?*
- We can treat it as a series of binary decisions. That is, we can have Shalmaneser first ask *Does this instance of the word have sense A?* then ask *Does this instance of the word have sense B?* and finally ask *Does this instance of the word have sense C?*

To treat three-or-more-way WSD tasks as a series of binary decisions, select **True** from the menu labeled **Binary classifiers?** Otherwise, select **False**.

6.3 How to train SRL classifiers

1. Select the **Train** tab.

2. Within the **Train** tab, select the **Semantic Role Labeling** tab.
3. Using the controls, specify how you want your new classifier to be trained. More on this below.
4. Press **Go**.

The SRL training controls are divided among five areas according to their function.

Description In the text field labeled **Description**, enter a descriptive name for your WSD classifier. In the **Preprocessed Input** pull-down menu, select the data set you want to train this new classifier on.

Options In this are are three check boxes. Each one lets you toggle on and off a particular option that affects how Shalmaneser handles SRL — for more information on these, see §5.3.2.

Tools In this area is a single pulldown menu, from which you can choose the machine learning toolkit you want to use for the classifiers you are training. Currently, the available options are **Timbl** and **Maxent**.

Classification Granularity As we saw in the last chapter, each individual classifier in a set handles a very small sub-task. In the case of SRL, there is room for you to determine exactly *how* small those sub-tasks should be. That is what the controls in this area allow you to do.

There are three possible levels of granularity:

1. **Frame**: At this level of granularity, each frame gets its own classifier. That is, there will be one classifier trained to recognize or label arguments of the ABANDONMENT frame, one for the ABOUNDING_WITH frame, and so on through to the WORKING_ON frame.
2. **Target POS**: At this level of granularity, each combination of frame and part of speech gets its own classifier. That is, there will be one classifier trained to recognize or label arguments of verbs in the ABANDONMENT frame, one for nouns in the ABANDONMENT frame, and so on for the other parts of speech; and then there will be one for verbs in the ABOUNDING_WITH frame, one for nouns, and so on; and this will continue through the WORKING_ON frame, where there will be one classifier for verbs, one for nouns, and so on.

3. **Lemma**: At this level of granularity, every lemma gets its own classifier. That is, there might be one classifier trained to recognize or label arguments of frame-bearing words with the lemma *aardvark*, one for frame-bearing words with the lemma *about*, and so on through the lemma *zymurgy*.

If you wish, you can choose different levels of granularity for argument recognition and argument labeling.

To choose the level of granularity for argument recognition, select the appropriate check box in the row marked **Argrec**. To choose the level of granularity for argument labeling, select the appropriate check box in the row marked **Arglab**.

Features In this area you can select which features Shalmaneser will use for training. The available features are described in §5.3.1. Because there are many more features that can be used for SRL than can be used for WSD, and because a set of SRL classifiers contains both argument recognition and argument labeling classifiers, the controls here are different than the feature controls in the **Word sense disambiguation** tab.

At the left side of this area is a scrolling list of the available features. To begin using a feature for argument recognition:

1. Select the feature you wish to begin using from the list of **Available** features.
2. Press **Add to argrec**.

Similarly, to begin using a feature for argument labeling, select it and press **Add to arglab**. To begin using a feature for both argument recognition and argument labeling, select it and press **Add to all**.

At the right side of this area are three lists, showing the features currently in use for both argument recognition and argument labeling (marked **All**), for argument recognition alone (**Argrec**) and for argument labeling alone (**Arglab**). All three lists start out empty; as you select features to use, they will be added to the appropriate list.

To stop using a feature:

1. Select the feature you wish to stop using from the right-hand list it appears in (either **All**, **Argrec** or **Arglab**).
2. Press **Remove**.

The name of the feature will be removed from the appropriate list, and will reappear in the **Available** list.

Chapter 7

Evaluating and experimenting with new classifiers

7.1 Test and development data

Training a machine learning system takes a large amount of labeled data; and since improvement increases as the amount of training data goes up, it is tempting to use *all* out labeled data for training purposes. But in fact it is important not to do this.

Why not? Because we will eventually want to find out how well our classifier performs on unseen. It will, of course, do very well on "predicting" the labels for the data it was trained on — but that's hardly a difficult or interesting task. The real point is to train a classifier that can make genuine predictions about data it hasn't seen before. And to evaluate how well it's doing *that*, we'll need a separate body of **TEST DATA** — with labels, so we can check if the classifier's predictions are correct.

The usual thing to do is to split off a certain amount of your labeled data, perhaps 10%, and set it aside as test data. Once you have trained a set of classifiers on the remaining 90%, you can test them on the 10% you reserved.

If you are tinkering with a set of classifiers, trying to improve its performance, it is also important not to run them on the test data until you are finished tinkering. Especially in a system like Shalmaneser where there are a lot of parameters you can change, it is possible to over-optimize. You train classifiers, run them on the test data, and get poor results. So you change a parameter, run them again, and get slightly better results. So you change another parameter... and soon you have a set of classifiers that happens to do incredibly well on the particular data you set aside for testing.

But when you try them on other data, they may perform terribly. This is because it may turn out that some of the parameter changes you made weren't really improving its *gen-*

eral performance — they were just achieving a closer fit to the data points in your particular set of test data, at the expense of performance everywhere else.

So if you're planning on doing a lot of tinkering, you should set aside *two* subsets of your labeled data — 10% for test data, as before, and another 10% for **DEVELOPMENT DATA**. While you're tinkering, you should evaluate the classifiers against the development data. Then when you're absolutely finished, you can evaluate them against the test data. They will likely perform *slightly* worse against the test data than they did against the development data — but this is generally a more accurate indication of how well they will perform in the future against unseen data. If they perform *much* worse against the test data, then you will know you have over-optimized.

Shalmaneser makes it easy to divide a set of data into equal partitions when you import it. See the instructions for the **Parser** area of the **Preprocess** tab in §6.1 for details.

7.2 Measuring performance

Once you've set up an experiment, the next question to ask is how we will interpret the results. A common goal in machine learning research is to find new ways of training classifiers that will give more accurate or reliable predictions. In this case the question is, how do we measure accuracy or reliability?

This section describes a few performance metrics that are widely used in machine learning research — not just by computational linguists but by researchers using machine learning techniques for any number of tasks.

7.2.1 Binary classification: precision, recall and F_1-score

For most **BINARY CLASSIFICATION** tasks (tasks where the classifier can assign one of two possible labels) there is a small set of **TARGETS** which we want to pick out from a larger — often *much* larger — set of **CANDIDATES**. You can think of a search engine as performing this sort of binary classification task. The candidates are all the documents it's indexed; the targets are the documents containing a particular word or phrase; and the goal is to pick out all and only the target documents.

As we saw in the last section, Shalmaneser treats argument recognition as this sort of binary classification task. Every word or phrase in a sentence is a candidate; the targets we want to pick out are the words or phrases that are arguments of a particular frame-bearing word. We also have the option of training it to treat word sense disambiguation as a binary classification task. In that case, each classifier is trained to recognize a particular sense of a particular word. For each occurrence of that word, the classifier's task is to

		It's actually…	
		in target set	not in target set
The system…	selects it	*tp*	*fp*
	doesn't select it	*tn*	*fn*

Table 7.1: Two ways to be right, and two kinds of errors

decide: "does 'my' sense apply here or doesn't it?" So the candidates are the occurrences of that word, and the targets for each classifier are those occurrences where 'its' particular sense applies.

When an automatic classifier performs this sort of task, it selects some of the candidates based on the *prediction* that they will belong to the target set. If it performs perfectly, all its predictions will be correct: it will select every single target, and will not select any non-targets. But more often, it will make some errors: sometimes it will select an item that turns out to be a non-target; sometimes it will reject an item that turns out to be a target. In evaluating a binary classifier, then, the question we must ask is how often it makes these two kinds of errors.

The table below sums up the possibilities. Every item is either a *target* or a *non-target*, and every time the classifier considers an item, it will either *select it* or *reject it*. The goal is to build a classifier that will give **TRUE POSITIVES** (*tp*) by selecting the targets, and **TRUE NEGATIVES** (*tn*) by rejecting the non-targets. The two kinds of errors a classifier can make are **FALSE POSITIVES** (*fp*) in which it selects a non-target, and **FALSE NEGATIVES** (*fn*) in which it rejects a target.

PRECISION tells us how many of our classifier's selected items belong to the target set — or, to put it another way, how many of the classifier's selections were correct.

$$precision = \frac{tp}{tp + fp}$$

We can think of precision as measuring the credibility of a classifier's selections. If a classifier has high precision, we can expect most of the items it selects to be real targets. If a classifier has low precision, on the other hand, we should expect many or even most of its selections to be mistakes.

RECALL tells us how many of the target items were selected by our classifier.

$$recall = \frac{tp}{tp + fn}$$

We can think of recall as measuring a classifier's thoroughness. A classifier with high recall tends to select most or all of the targets, even if it selects some non-targets along with them. A classifier with low recall will tend to mistakenly reject many of the targets.

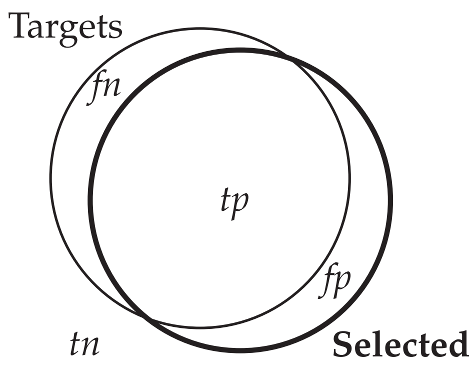


Figure 7.1: Moderate precision, moderate recall

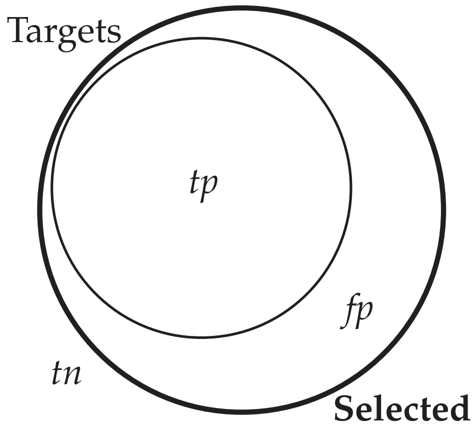


Figure 7.2: Perfect recall, low precision

There is often a tradeoff between precision and recall. At one extreme, we can build a classifier that will only select one item that it is absolutely certain of. If its selection is correct, such a classifier will get a 100% score for precision, but this strategy of selecting one target out of dozens or even hundreds will give it a very low score for recall. At the other extreme, we can build a classifier that gets 100% recall by simply selecting *everything* — all of the targets, and all of the non-targets too — but such a classifier will achieve very low precision. Figures 7.1–7.3 give a (somewhat simplified) picture of this: Figure 7.1 shows the usual situation, with the selection and the target set partly overlapping; 7.2 shows the selection made by a classifier that has achieved perfect recall at the expense of precision; and 7.3 shows the selection made by one that has achieved perfect precision at the expense of recall.

Given this tradeoff, it is often better to use a metric that rewards *both* precision *and* recall. One commonly used metric of this sort is the **F_1-SCORE**, defined as the harmonic mean of

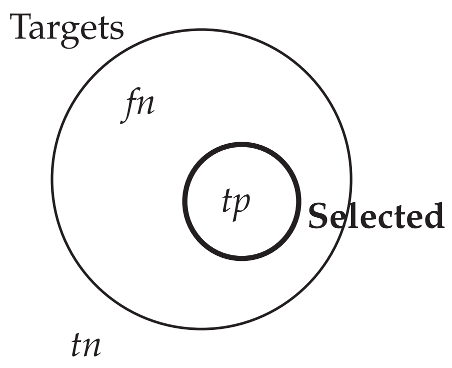


Figure 7.3: Perfect precision, low recall

precision and recall.[1]

$$F_1 = 2\left(\frac{precision \cdot recall}{precision + recall}\right) = \frac{2tp}{2tp + fp + fn}$$

7.2.2 The general case: accuracy

So far we've been talking about metrics for binary classifiers. But when we move to the general case of measuring performance for *any* classifier, precision, recall and F$_1$-score are no longer useful.

As we saw, the definitions for precision, recall and F$_1$-score all depend on the assumption that there are only two ways to be right (true positives and true negatives) and two ways to be wrong (false positives and false negatives). In a non-binary classification task, these assumptions no longer hold. If we've trained a classifier to distinguish all six FrameNet senses of the verb *drop*, there are six ways for it to be right (one per sense), and 30 ways for it to be wrong (five per sense, since any time one word sense label is correct, the other five will be incorrect). With this many possibilities, the equations for precision, recall and F$_1$-score are no longer well-defined.

The metric that is most commonly used for non-binary classifiers is **ACCURACY**, which is simply defined as the percentage of items that were labeled correctly.

$$accuracy = \frac{\#\ correct}{total\ \#}$$

[1]The harmonic mean, defined as $(ab)/(a+b)$, is used for taking the average of two ratios a and b. The subscript 1 in F$_1$ indicates that precision and recall are given equal weight in this calculation. We can also calculate, for instance, an F$_2$ score which weights precision twice as heavily as recall, an F$_{0.5}$ score which weights it half as heavily, and so on. But these metrics are less commonly used — in fact, the F$_1$ score is so much more widespread than the others that you will often see it referred to as the F-score, with no subscript.

Accuracy is guaranteed to be defined no matter how many *ways* there are of being right and wrong, because it doesn't distinguish — it lumps all the ways of being right into a single value and counts *that*. Consequently, we can calculate a classifier's accuracy on an n-way classification task for any n.

So why not use accuracy for $n = 2$ — for binary as well as non-binary classification? Well, it's entirely possible. In fact, the accuracy of a binary classifier is easily calculated:

$$accuracy = \frac{\#\ correct}{total\ \#} = \frac{tp + tn}{tp + tn + fp + fn} \qquad \text{[for binary classification]}$$

But there are problems with accuracy that makes it preferable to avoid when we have other options.

First of all, it is often possible to gain high accuracy scores on a binary classification task using a trivial strategy: *don't select anything*. This strategy exploits a common feature of many interesting binary classification tasks: the number of targets is often very small compared to the number of candidates. When this is the case, selecting nothing leads to an enormous value for *tn*, and thus to a very high accuracy score.

And on the other hand, the small number of targets in these cases means that the best possible value of *tp* is still quite small. The difference in accuracy score between a classifier using this trivial strategy (thus getting a *tp* of zero) and one using a more successful and more interesting strategy (thus getting a small-but-nonzero *tp*) will itself be quite small. But this difference — between doing nothing and doing something worthwhile — is precisely what we are interested in, so it is counterproductive to use a metric that minimizes it.

Significantly, the F_1-score does not suffer from either of these problems. We chose it, remember, specifically to avoid rewarding trivial strategies. And it maximizes the contribution of *tp* rather than minimizing it. For these reasons, it is best to use it — and its components precision and recall — when possible, and to fall back on accuracy only when we are evaluating a non-binary classifier.

7.2.3 What counts as good performance?

Performance metrics don't mean much in a vacuum. An accuracy score of 75% could be impressively good or incredibly bad, depending on the task.

Consider the task of disambiguating the verb *go*. Framenet lists four frames it might evoke: MOTION, which covers most uses of the word, and BECOMING, BEING_NAMED and COMPATIBILITY, which cover less common uses like the following:[2]

[2] I'm leaving out *phrases* containing *go*, some of which evoke frames of their own: *go back and forth* evokes WAVER_BETWEEN_OPTIONS, for instance. But let's **go on**$_{\text{PROCESS\_CONTINUE}}$ ignoring them for the sake of simplicity

(7.1) a. French actress Juliette Binoche plays Michele, an artist from a wealthy family who is **going** blind and lives rough out of choice.

b. Saul Hudson, who **goes** by 'Slash,' was the lead guitarist of Guns 'n Roses.

c. That tie doesn't really **go** with this shirt.

Let's say we've trained a classifier to perform this task, and it gets an accuracy score of 75% on the test data we've set aside. Now suppose we look through that test data and discover that fully 90% of the instances of the word *go* evoke the MOTION frame. That means our classifier is performing 15 percentage points *below* chance! Consider a classifier that always assigned the MOTION label; this classifier would have gotten a 90% accuracy score without doing any real work. If our carefully trained competitor can't defeat a trivial strategy like this one, then that is very bad news.

On the other hand, suppose on inspecting the test data we discovered that the instances of the word *go* were evenly split between the four frames. In that case, the best score a classifier could get by chance alone would be 25%. Against *that* set of test data, an accuracy score of 75% would be a significant improvement over chance, a sign that we'd really done something right.

What we're seeing here is the need to set a standard for every experiment you perform. One way to do this is by comparing your classifier's performance to the **BASELINE** performance that a totally naïve algorithm could achieve simply by chance. On a binary classification task where the two labels are equally likely, the baseline is 50%. On a 4-way classification task with all four labels equally likely, it is 25%. If one label is more likely than others, it sets the baseline: if, for instance, there is one label that appears on 90% of the data points, then the baseline for the task is 90%.

A tougher challenge is to use the result of a previous experiment as a **BENCHMARK** and try to beat that. This is how much of the day-to-day progress in computational linguistics is made: a team of researchers will challenge themselves to beat the benchmark on a popular task, and if they succeed, their performance sets the new benchmark for the next attempt.

7.3 Analyzing problems: confusion matrices

We don't just want to know how well our classifiers perform, though. We want to think about how to make them perform better. And to do this, we need to know what kinds of mistakes they are making, and not just how often they are making mistakes.

One way of visualizing this information is with a **CONFUSION MATRIX** — an *n* x *n* contingency table, where *n* is the number of labels that the classifier must choose between. Each row in the confusion matrix represents the data points to which the classifier assigned a

		Actual label			
		MOTION	BECOMING	BEING_NAMED	COMPATIBILITY
Predicted	MOTION	60	9	3	2
	BECOMING	5	3	2	1
	BEING_NAMED	0	0	0	0
	COMPATIBILITY	3	1	2	9

Table 7.2: A 4 x 4 confusion matrix: predicted *vs* actual senses of the word *go*

particular label; each column represents the data points to which a particular label *should* have been assigned. (You may have realized by now that 7.1 was really just a 2 x 2 confusion matrix. Although we don't have special names for the cells in an $n \times n$ confusion matrix for $n > 2$, the idea is the same.)

For instance, the first column of the confusion matrix above tells you that, out of all the occurrences of the word *go* in the test data,

- 68 evoked the MOTION frame,
- 60 of those 68 were correctly labeled MOTION by Shalmaneser,
- 5 of the 68 were incorrectly labeled BECOMING, and
- 4 of the 68 were incorrectly labeled COMPATIBILITY.

Meanwhile, the first row tells you that, out of all the occurrences of the word *go* in the test data,

- Shalmaneser labeled 74 MOTION,
- 60 of those 74 MOTION labels were correct,
- 9 of the 74 should have been labeled BECOMING instead,
- 3 of the 74 should have been labeled BEING_NAMED, and
- 2 of the 74 should have been labeled COMPATIBILITY.

We can draw a few conclusions from this information. First of all, our classifier is predicting the MOTION label a little too often. And second, it seems to be confusing MOTION and BECOMING with one another especially often; we should look for ways to help it distinguish those two frames.

Chapter 8

Syntax of the feature description language

8.1 Feature detectors

A **FEATURE DETECTOR** is a short program that takes a parse tree as input, and returns a value or a vector of values that can be used for training by a machine learning system. You've already interacted with feature detectors — every feature mentioned in the last few chapters is implemented by a feature detector that extracts the relevant information. In this chapter we'll discuss how to implement new features by writing your own feature detectors in a simple programming language called the Feature Description Language (or FDL).[1]

There are two kinds of feature detectors. **BINARY** feature detectors return either `true` or `false`. You might, for instance, write a binary feature detector that returns `true` if the current sentence is passive, or if the current node is a terminal node. **NON-BINARY** feature detectors can return any value or list of values. You might write a non-binary feature detector that returns the main verb of the current sentence, or a list of all the prepositions in the current sentence, or the path from the root to the current node.

In general, non-binary features are more versatile. Most of the preprogrammed feature detectors that Shalmaneser uses are non-binary, and so are many of the features from the Semantic Role Labeling literature that you might want to implement with a feature detector of your own. But binary feature detectors are easier to write and easier to understand

[1]The FDL might not seem much like the programming languages you're used to. For instance, an FDL program never tells Shalmaneser to *do* anything. Rather, it describes a set of patterns and relationships, and Shalmaneser searches for ways to satisfy the description. If you've used a logic programming language like Prolog, or if you're familiar with regular expressions like those used by the Unix command `grep` or the linguistic tool `tgrep2`, the FDL will probably seem familiar.

— and some of them are quite useful — so we will begin with those.

A binary feature detector is written as follows:

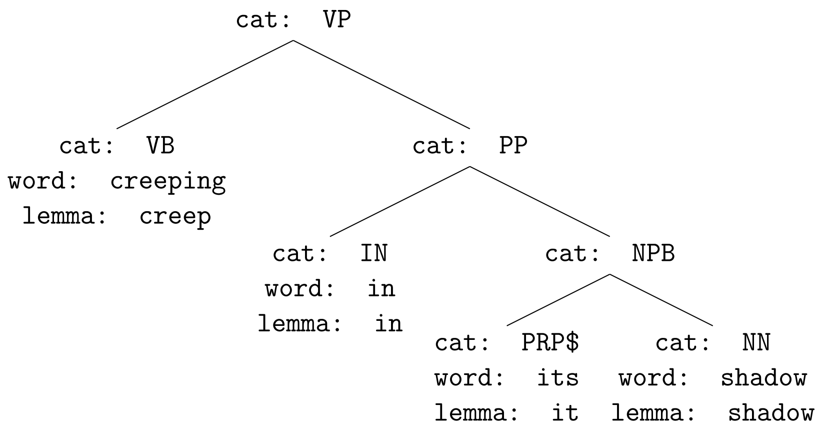
(8.1)
```
 feature FEATURE_NAME
EXPRESSION
```

FEATURE_NAME can be any string made up of letters and the underscore character. The possibilities for EXPRESSION are more complicated; we'll go into them in depth in the following sections. But the basic idea is this: every expression describes a pattern. The feature detector will search through the current sentence looking to see if any part of it matches the pattern. If it finds a match, it will return true; if it does not, it will return false.

Shalmaneser represents a sentence as a tree, where each node contains a list of property-value pairs. Every node has a cat property, representing its syntactic category. Terminal nodes also have word and lemma properties. (There is also one more property — is — whose distribution is more complicated. We will address it later in this chapter.)

Here, for instance, is how Shalmaneser represents the phrase creeping in its shadow:

(8.2)



The EXPRESSION in a feature detector describes a pattern of nodes in a tree such as this. Some expressions describe a single node, others describe a set of nodes. The feature detector will search the current sentence for a node or set of nodes that matches its pattern. As soon as it finds a match, it stops searching and returns true. If it searches the entire tree without finding a match, it returns false.

8.2 Describing a single node

A node description consists of a pair of square brackets, either with or without a **STATEMENT** between them. Simple statements take the form `PROPERTY = VALUE`, where `PROPERTY` can be `cat`, `word`, `lemma` or `in`, and `VALUE` can be any string surrounded by quotation marks[2] or a regular expression surrounded by forward slashes. Compound statements can be built out of simple ones, using the does-not-equal operator `!=`, the boolean operators `and` and `or`, and parentheses for grouping. We will cover this later in the section.

First, let's look at some concrete examples of node descriptions. The simplest one is an empty one:

(8.3) `[ ]`

This will match any node at all. On its own, it's not very useful — a search pattern that always matches, like an alarm that's always ringing or a light that's always on, doesn't convey any information. But as we'll see later, an empty node description *can* be useful in combination with others.

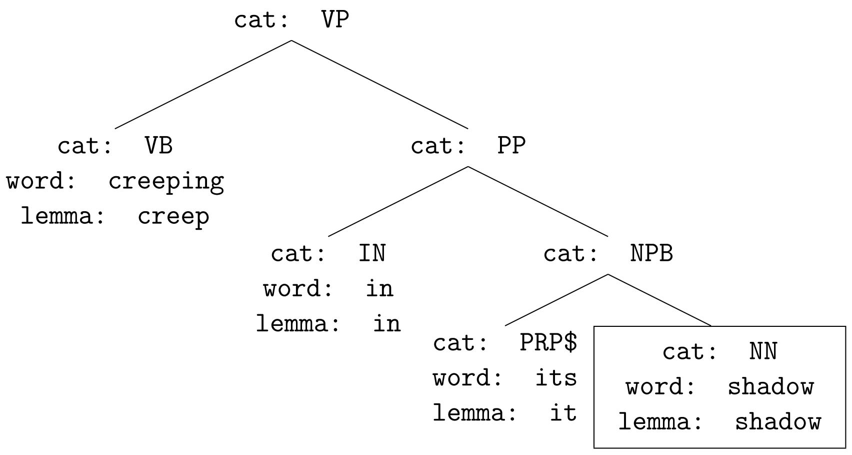
Here's a less trivial example:

(8.4) `[ cat = 'NN' ]`

This will match any node whose `cat` property has the value `'IN'`. As humans, we can tell at a glance that this pattern will find a match in the phrase *creeping in its shadow*. But Shalmaneser is more methodical — it visits every node in the tree in turn, checking to see if the pattern matches or not. Eventually, it will either find one that does match (as in this example; see below) or it will search the entire tree and return without a match.

[2]Either single or double quotes can be used. There's no difference in how they're interpreted, but there is one practical consideration — since the single quote character is also used for apostrophes, you cannot have an apostrophe in a string surrounded by single quotes. `"Bob's"` and `"you're"` will be interpreted correctly; `'Bob's'` and `'you're'` will cause problems

(8.5)



From the node description in 8.4 we can build a feature detector as follows:

(8.6)
```
feature contains_NN
[ cat = 'NN' ]
```

Once again, you may be able to guess at a glance what this will do, but let's go through step-by-step and make sure. If nothing else, it's a good habit to get in for when we begin writing more complicated feature detectors.

A feature detector returns `true` if and only if the expression it contains results in a match. That means that `contains_NN` will return `true` if and only if `[ cat = 'NN' ]` results in a match. And `[ cat = 'NN' ]` will match any node whose `cat` property is `NN`. So if we put it all together, `contains_NN` will return true for any tree with at least one node whose `cat` property is `NN`. In other words, it does exactly what its name says — it returns true for any tree that contains an `NN`.

Ex. 7 — Write feature detectors that return `true` if. . .

1. the sentence contains an adverb.
2. the sentence contains the word *spatula*.

Now let's look at ways of building compound statements out of simple ones. A simple statement can be negated by replacing the = with a `!=`. So `[cat != 'NN']` will match any node that *isn't* of category `NN`.

Any statements can be joined by one of the boolean operators `and` and `or`. These do exactly what you'd expect: `[X and Y]` will match any node that matches both `[X]` and `[Y]`; `[X or Y]` will match any node that matches either `[X]` or `[Y]`. If you use more than one boolean operator in a compound statement, you should use parentheses to clarify which one takes precedence.

Ex. 8 — Find the nodes in (8.2), if any, that will match:

1. `[cat != ’VB’]`
2. `[(cat = ’IN’ or cat = ’NN’) and word = ’elasmosaurus’]`
3. `[cat = ’IN’ or (cat = ’NN and word = ’elasmosaurus’)]`
4. `[cat = ’VB’ and lemma = ’creeping’]`

Ex. 9 — Write a node description that will match any verb with the lemma `go`.

Ex. 10 — Suppose you want to write a feature detector that returns `true` for sentences that contain no nouns, such as *Stop!* or *Don't give up.* Why will the following not work?

```
feature no_nouns [ cat != ’NN’ ]
```

Ex. 11 — The `!=` operator only lets you negate a simple statement. How would you write a node description that will match...

1. nodes that aren't verbs or verb phrases?
2. nodes that aren't verbs with the lemma `go`?

8.2.1 The special property `is`

Let's look again at the process Shalmaneser must go through in completing an argument recognition task. Suppose it's working on 8.2; it's already assigned *creeping* to the SELF_MOTION frame, and it's trying to predict which nodes, if any, should be assigned to this frame as arguments. We can break this task down into a whole series of subtasks. Shalmaneser must predict...

- whether the noun *shadow* is an argument of the frame evoked by *creeping*,
- whether the posessive pronoun *its* is an argument of the frame evoked by *creeping*,
- whether the noun phrase *its shadow* is an argument of the frame evoked by *creeping*,

and so on for every node in the tree. In a longer sentence, with more than one constituent that evokes a frame, this series of subtasks will need to be repeated for every frame evoked. We can sum this up as follows: Shalmaneser must predict...

- whether **current node** is an argument of the frame evoked by **target node**

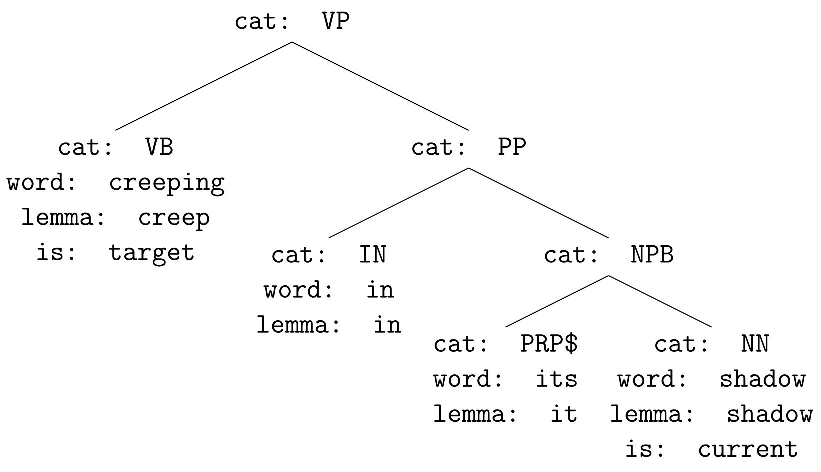
for every possible value of **current node** and **target node**. Obviously, for some values the correct prediction will be *yes* and for some it will be *no*. In order to generate the correct predictions, we'll need features whose values vary depending on the **current node** and **target node** under consideration.

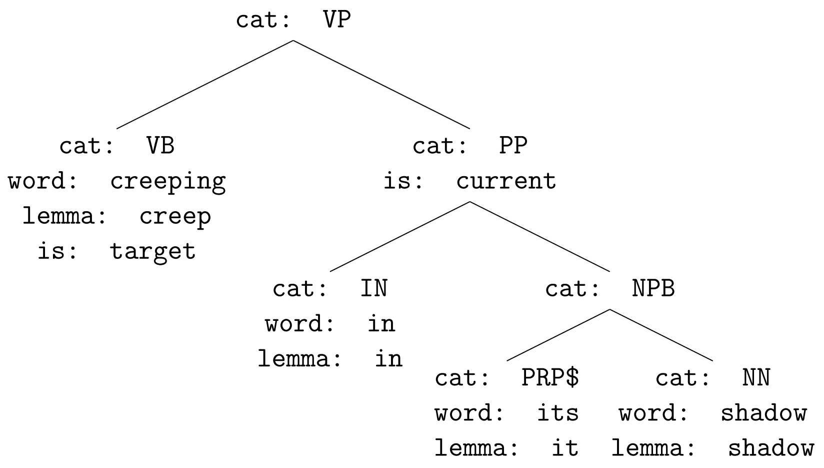
We've used argument recognition as an example here, but the same will turn out to be true of argument labeling: each semantic role label will be appropriate for some current/target pairs, and inappropriate for others. For instance, if the target node is *buy* and the current node is the noun phrase *fifty-seven dollars*, MONEY might be an appropriate label. But if the current node is the noun phrase *Crazy Jim's discount auto*, SELLER is probably more appropriate.

So for either of the tasks involved in semantic role labeling, we need features that are sensitive to the location of the current and target nodes. Here's how Shalmaneser handles this: there is a special property, `is`, used to mark which node is the current node and which node is the target. Every time the question under consideration changes, the current node is marked with the property-value pair `is: current` and the target node is marked with the property-value pair `is: target`.

For instance, if the question under consideration is whether *shadow* is an argument of the frame evoked by *creeping*, the tree in (8.2) will look like (8.7). If the question is whether the `PP` *in its shadow* is one of its arguments, the tree will look like (8.8)

(8.7)



(8.8) 

`Is` has one more function in addition to the ones we've discussed. The root node of a tree is always marked with the property-value pair `is: root`. This does not change with the question under consideration, but is a permanent property of that node.

Ex. 12 — Suppose Shalmaneser is doing argument recognition on (8.2). If the question under consideration is whether *its shadow* is an argument of the frame evoked by *creeping*, which of the following will result in a match?

1. `[word = 'shadow' and is = 'current']`
2. `[is = 'target' and cat = 'VB']`
3. `[is = 'target' and cat = 'VP']`
4. `[is != current and cat = 'NN']`

Ex. 13 — What if the question under consideration is whether *in its shadow* is an argument of the frame evoked by *creeping*?

Ex. 14 — When will this feature detector return true? Why might this be useful?

```
feature mystery
[is = 'root' and cat != 'S']
```

8.2.2 The special value `nil`

Not all attributes are defined at all nodes. Only terminal nodes have a `word` or a `lemma`. Only *non*-terminal nodes have a `head`. `Is` is only defined for three nodes at a time — the current node, the target, and the root.

When you check an attribute at a node where it is not defined, you get a special value back: `nil`. We can take advantage of this fact to construct a few useful node descriptions.

Here, for instance, is one that will match any non-terminal node.

(8.9) `[ word = 'nil' ]`

(Checking whether `lemma = nil` would accomplish the same thing). We can negate it if we want to match terminal nodes instead.

(8.10) `[ word != 'nil' ]`

Ex. 15 — What would `[ is != 'nil' ]` do?

8.3 Describing multiple nodes

So far, we've been dealing with expressions that match a single node in isolation. This is all well and good for writing lexical or morphological features, but for syntactic features we'll need to be able to talk about the *relationships* between nodes.

8.3.1 Boolean relationships: `and` and `or`

The boolean relationships `and` and `or` let you join two subexpressions together into an expression. `[X] and [Y]` results in a match if `[X]` results in a match and `[Y]` results in a match. `[X] or [Y]` results in a match if either `[X]` results in a match or `[Y]` results in a match.

(8.11) `[lemma = 'fish' and cat = 'V']`

(8.12) `[lemma = 'fish'] and [cat = 'V']`

Be careful not to confuse this use of `and` and `or` with their use *inside* a node description. The expression (8.11) describes a *single* search pattern. It will match any *one* node that has *fish* as its lemma and `V` as its category — in other words, any form of the verb *to fish*. By contrast, the expression (8.12) describes *two* search patterns. First Shalmaneser will search for a node that has *fish* as its lemma. If it finds one, it will search for a node — possibly the same one, possibly a different one — that has `V` as its category. If both searches are successful, then the whole expression results in a match.

You can see the difference between the two by considering the sentence *Fish is an important part of a balanced diet.* Expression (8.11) will not match this sentence, because there is no single node that satisfies both conditions. On the ther hand, the expression (8.12) will match it: there is a node that satisfies the first condition (the word *fish*) and there is a node that satisfies the second condition (the word *is*).

Ex. 16 — Write feature detectors that return `true`...

1. if the target node is a verb, but not the only verb in the sentence.
2. if the current node is a subclause.

8.3.2 Syntactic relationships: the tree traversal operators

The **TREE TRAVERSAL OPERATORS** describe the relative position of two nodes within a syntactic tree. Two node descriptions joined together with a tree traversal operator form a **SUBTREE DESCRIPTION**. While node descriptions match single nodes, subtree descriptions match pairs of nodes.

For instance, take the following expression:

(8.13)
```
[ cat = 'VP' ] \ [ cat = 'S' ]
```

The subexpressions to either side of the \ operator match individual nodes: `[ cat = 'VP' ]` matches any verb phrase, `[ cat = 'S' ]` matches any clause or sentence. The entire expression matches a pair of nodes in a particular configuration — a verb phrase with a clause as its child. If we build a feature detector out of the expression, it will return `true` for any sentence containing a verb with a complement clause:

(8.14)
```
feature contains_complement_clause
[ cat = 'VP' ] \ [cat = 'S' ]
```

More often, though, the tree traversal operators are used together with the special property `is` to describe relationships between the current and target nodes, or between either of them and the rest of the tree. A number of useful features can be implemented this way. Here, for instance, is a feature detector that returns true if the target node C-commands the current node:

(8.15)
```
feature c_command
[ is = 'target' ] cc [ is = 'current' ]
```

Table 7.1 gives a full listing of the tree traversal operators and their uses.

Operator	Name	Semantics
`/`	child-of	`X / Y` if `X` is the child of `Y`
`\`	parent-of	`X \ Y` if `X` is the parent of `Y`
`/*`	dominated-by	`X /* Y` if `X` is dominated by `Y`
`\*`	dominates	`X \* Y` if `X` dominates `Y`
`|`	sibling-of	`X | Y` if `X` and `Y` have the same parent
`<|`	left sibling	`X <| Y` if `X` and `Y` have the same parent and `X` appears to the left of `Y`.
`>|`	right sibling	`X >| Y` if `X` and `Y` have the same parent and `X` appears to the right of `Y`.
`<*`	left of	`X <* Y` if `X` appears to the left of `Y`.[1]
`>*`	right of	`X <* Y` if `X` appears to the right of `Y`.[1]
`cc`	C-command	`X cc Y` if `X` and `Y` are siblings or if `Y` is dominated by a sibling of `X`.

Table 8.1: Tree Traversal Operators

Note that syntactic relationships always take precedence over boolean ones. The expression `[is = 'current'] \ [PP] and [is = 'target'] \ [VP]`, for instance, will return a match if two conditions are met: first, if the current node's parent is a PP, and second, if the target noe's parent is a VP.

Ex. 17 — Describe in plain language when the following expressions will result in a match.

1. `[ is = 'current' ] | [ is = 'target' ] and`
   `[ is = 'current' ] / [ cat = 'VP' ]`
2. `[ is = 'current' and cat = 'NP' ] \ [ cat = 'PP' ]`

[1]When `X` and `Y` are not siblings, it's a little tricky defining what it means for `X` to be to the left of `Y`. Here, if you're interested, is the full definition:

`X <* Y` if either

1. `X` is dominated by a left child of `Y`,
2. `X` is dominated by a left sibling of `Y`,
3. `X` is dominated by the left sibling of any node that dominates `Y`, or
4. `X` dominates `Y` and all the nodes between `X` and `Y` are left children of their parent.

Similarly, `X >* Y` if either

1. `X` is dominated by a right child of `Y`,
2. `X` is dominated by a right sibling of `Y`,
3. `X` is dominated by the right sibling of any node that dominates `Y`, or
4. `X` dominates `Y` and all the nodes between `X` and `Y` are right children of their parent.

Ex. 18 — Explain the difference between these two expressions.

1. `[ is = 'current' and cat = 'NP' ] \ [ cat = 'PP' ]`
2. `[ is = 'current' ] and [ cat = 'NP' ] \ [cat = 'PP' ]`

Ex. 19 — Write feature detectors that will return `true`...

1. if the current node *either* C-commands *or* is C-commanded by the target node.
2. if the current node is not a terminal node.
3. if the current node is *in* a subclause.

8.4 Variable binding

A node that has been matched by a node description can be bound to a named variable using the following syntax:

(8.16) `VARNAME:[ DESCRIPTION ]`

For instance, the following code will match any node of category `NPB`, and will assign that node as the value of the variable `thenounphrase`.

(8.17) `thenounphrase:[ cat = 'NPB' ]`

Bound variables can be used later in a feature detector to refer back to the node they have been assigned to. The following code, for instance, will match any noun phrase containing both an adjective and a determiner:

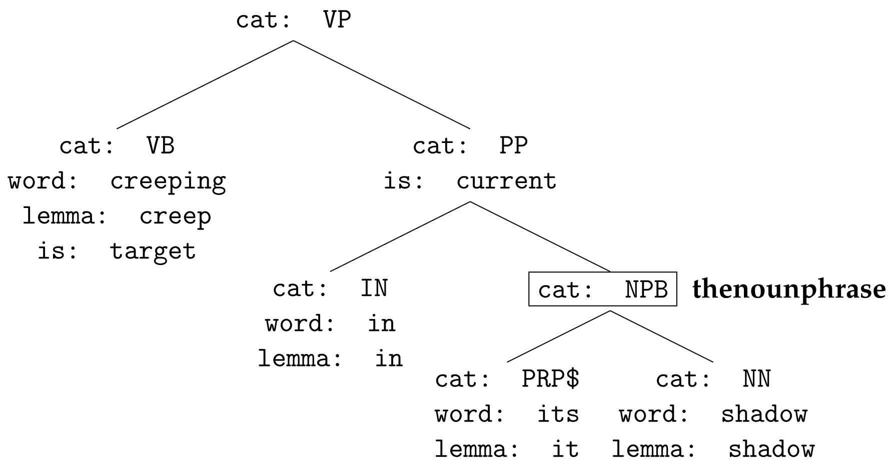
(8.18)
```
thenounphrase:[ cat = 'NPB' ] and (thenounphrase \ [ cat = 'DT' ] and
thenounphrase \ [ cat = 'JJ' ])
```

Let's look at how Shalmaneser searches for a match for the code in 8.22. Because of the boolean operator `and`, 8.22 will result in a match if all three subexpressions result in matches:

(8.19)
a. `thenounphrase:[ cat = 'NPB' ]`
b. `thenounphrase \ [ cat = 'DT' ]`
c. `thenounphrase \ [ cat = 'JJ' ]`

Let's start with the first subexpression. It will match the `NPB` node dominating *its shadow*, and assign that node to the variable `thenounphrase`:

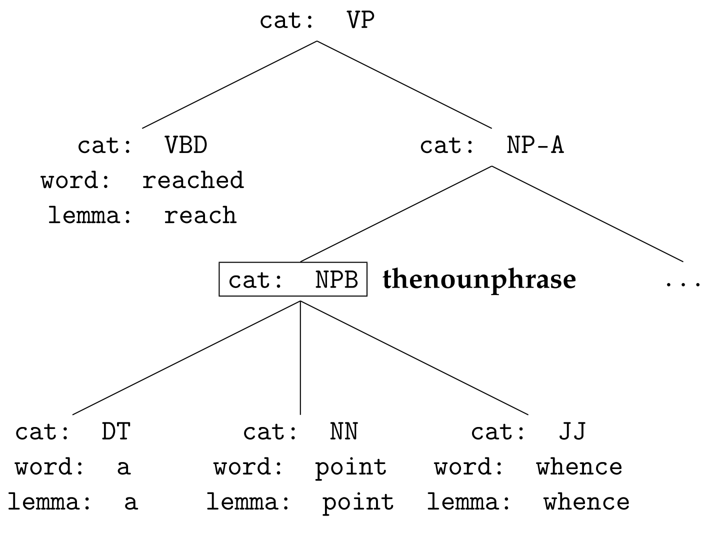
(8.20)



The question now becomes: given the way we matched the first subexpression, can we find compatible matches for the second and third ones? In this case, the answer is no. The node we've assigned to `thenounphrase` does not have any nodes of category `JJ` or `DT` as children.

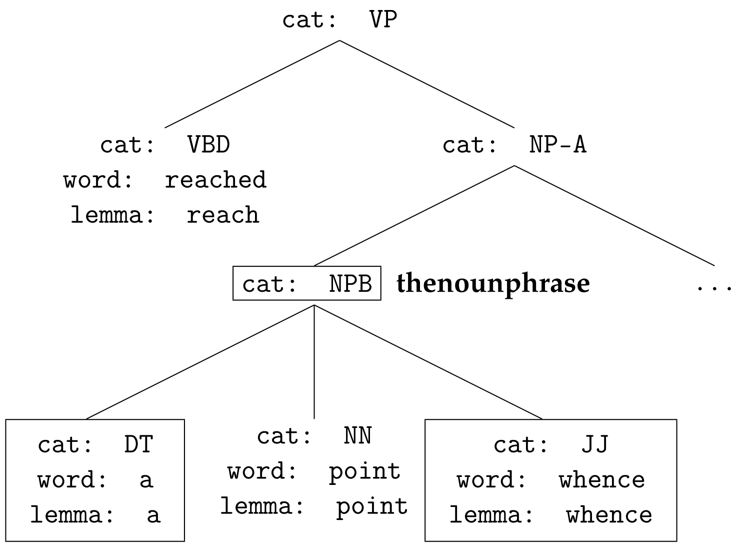
So we backtrack: we look for other ways to match the first subexpression. Here's one — it matches the `NPB` dominating *a point whence*. Now we assign that new match to the variable `thenounphrase`, and proceed as before:

(8.21)



This time the second and third subexpressions match as well. Success!

(8.22)



One especially useful trick is to bind an empty node description to a named variable. An empty node description, remember, — a pair of square brackets with nothing inside — will match any node at all. The following expression lets `X` refer to any node — not terribly useful on its own.

(8.23) `X:[ ]`

But it can be useful in combination with other expressions. Here's an expression that lets `X` refer to any node *so long as* it's the parent of the current node. In other words, it's equivalent to the English instruction "Let `X` be the parent of the current node."

(8.24) `X:[ ] / [ is = "current" ]`

Ex. N Write an expression corresponding to the instruction "Let `X` be an ancestor of the current node."

Ex. N+1 Write one corresponding to the instruction "Let `X` be an ancestor of the current node and a sibling of the target node."

Ex. N+2 Write one corresponding to the instructions "Let `X` and `Y` be two different siblings of the current node." (Hint: make sure that $X \neq Y$.)

8.5 Negation as failure — the `not()` operator

In Chapter 5, we saw that many of the built-in SRL features in Shalmaneser make use of the concept of the **LEAST COMMON ANCESTOR** (or **LCA**) of two nodes. If A and B are nodes, their LCA is the unique node X such that

1. *X* is an ancestor of *A*,
2. *X* is an ancestor of *B*, and
3. any node *Y* that meets conditions 1 and 2 is an ancestor of *X*.

In other words, the least common ancestor of *A* and *B* is the *lowest in the tree* that is still an ancestor of both *A* and *B*, if we visualize the tree in the usual way with its root node at the top.

Suppose we want an expression to find the LCA of the current and target nodes. Here is a first attempt at writing such an expression.

(8.25)
```
( X:[ ] \* [ is = "current" ] and X \* [ is = "target" ] ) and
( ( Y:[ ] \* [ is = "current" ] and Y \* [ is = "current" ] )
    and Y /* X )
```

This is a more complicated expression than we've dealt with it so far, but we can understand it the same way we understand simpler ones — by breaking it down into subexpressions. There are five subexpressions, all joined together using `and`, so all five must match for the full expression to match. Here are rough English translations of the five subexpressions. All five of these conditions must be met before the full expression matches:

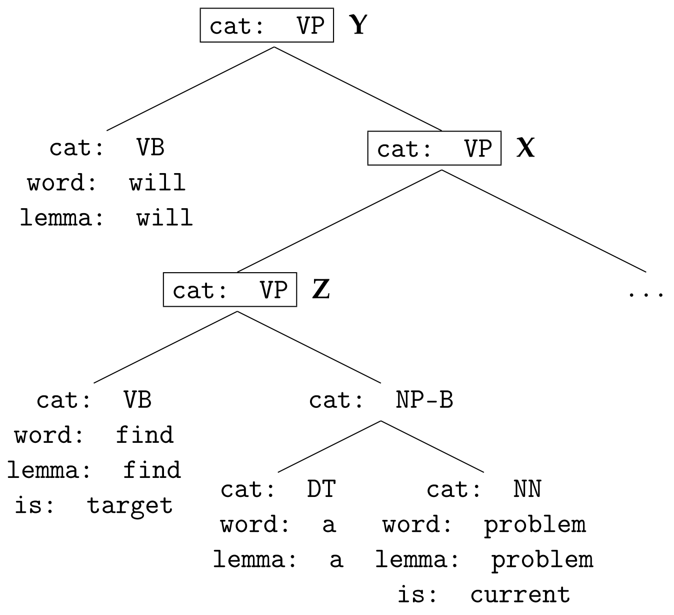
1. `X` is an ancestor of the current node,
2. `X` is an ancestor of the target node,
3. `Y` is an ancestor of the current node,
4. `Y` is an ancestor of the target node, and
5. `Y` is an ancestor of `X`.

At first glance, it might appear that if these conditions are met, `X` will be the LCA of the current and target nodes. But there is a problem.

Ex. What is the problem? Before you read on, see if you can identify it yourself. (Hint: See if you can find a tree and a set of variable bindings where all five conditions are met but `X` is not the current and target nodes' LCA.)

Here is such a tree, that meets all five conditions but doesn't leave `X` as the current and target nodes' LCA:

(8.26)



`X` is an ancestor of the current and target nodes. `Y` is an ancestor of the current and target nodes. And `Y` is an ancestor of `X`. But `X` is not the LCA of the current and target nodes. (That honor belongs to `Z`.)

What went wrong? Well, ordinarily, when Shalmaneser is matching an expression, it just looks for one match. If it finds one, it stops looking. And in particular, when it is matching an expression with variables in it, it just looks for one node to assign to each variable. So in 8.30, it finds a single `Y` that's above `X` in the tree, and it stops looking.

But we don't just want *one* `Y` that's above `X` in the tree. We want *all* the possible `Y`'s to be above `X`. To put it another way, we want to make sure that there are *no* possible `Y`'s *below* `X`.

Here, in plain English, are the real conditions that a node `X` will have to meet to count as the LCA of the current and target nodes:

1. `X` is an ancestor of the current node,
2. `X` is an ancestor of the target nodes, and
3. There is *no* node `Y` such that
   (a) `Y` is an ancestor of the current node,
   (b) `Y` is an ancestor of the target node, and
   (c) `Y` is a descendant of `X`.

Condition 3 can be checked with the following expression

(8.27)
```
not( ( [ Y:[ ] \* [ is = "current" ] and Y \* [ is = "target"] )
       and Y /* X )
```

and to check all three conditions, we can use the following:

(8.28)
```
( X:[ ] \* [ is = "current" ] and X \* [ is = "target" ] ) and
not( ( [ Y:[ ] \* [ is = "current" ] and Y \* [ is = "target"] )
       and Y /* X )
```

8.6 Non-binary features

So far we've been writing binary feature detectors — ones which return `true` if they find a match and `false` if they do not. But we can also write feature detectors with a wider range of return values. These take the following form:

(8.29)
```
feature FEATURE_NAME
EXPRESSION
<RETURN_VECTOR>
```

Note what's changed: we've added a third line, called the **RETURN VECTOR**. This specifies what is to be returned when a match is found.

8.6.1 Return vectors

A return vector contains one or more values that are to be returned by the feature detector. The list of values is surrounded by angle brackets, and if there is more than one value they are separated by commas.

Most of the values that appear in a return vector will be properties of individual nodes in the tree. For instance, we might want to return the target node's `word`, or the `cat` of the current node's parent. In order to return this sort of value, there are two steps. First, we must identify the node we're interested in. And second, we must identify which property we want to return.

We've already seen how to identify a node we're interested in — we bind it to a variable. Suppose we're interested in the current node's parent. We can bind it to the variable `X` using the following expression:

(8.30)
```
X:[ ] / [ is = 'current' ]
```

Now we must identify one of parent's properties to return. We do this using a period followed by the name of the property. So to return the parent node's cat, when the parent node is bound to the variable X, we write the following return vector: < X.cat >

The result is a feature extractor that looks like this:

(8.31)
```
feature parent_cat
X:[ ] / [ is = 'current' ]
< X.cat >
```

We mentioned above that a return vector can contain a comma-separated list of values instead of a single value. To return a vector consisting of the cat of the current node *and* the cat of its parent, we can write the following.

(8.32)
```
feature current_and_parent_cat
X:[ ] / Y:[ is = 'current' ]
< X.cat, Y.cat >
```

Similarly, we can use more than one property of a single node in a return vector. Here's a feature detector that returns the cat, word and lemma properties of the current node.

(8.33)
```
feature current_properties
Y:[ is = 'current' ]
< Y.cat, Y.word, Y.lemma >
```

8.6.2 Lists of nodes with all()

Suppose we want a simple context feature — a vector of all the words that appear in a sentence. We might be tempted to write something like this. (Remember that [word != 'nil'] will match any terminal node — any node for which the word attribute is defined.)

(8.34)
```
feature bag_of_words
X:[ word != 'nil' ]
< X.word >
```

This approach won't work. If we run 8.38, we will only get a single word as a return value, no matter how long a sentence we run it on. This is because Shalmaneser's default behavior is to stop searching once it has found a match. If it is applying 8.38 to a tree, it will search until it finds a single node matching [word != 'nil'], it will bind the variable X to that single node — and then it will stop searching, and return X.word.

If we want *all* the words in a sentence, we need to override this default behavior of Shalmaneser's. The way to do this is with the `all()` operator. `All()` takes two arguments: the name of a variable, and an expression in which that variable is assigned to a node. When Shalmaneser encounters `all()`, it keeps searching for matches for the expression until it's found every possible node that the variable could be assigned to.

So for instance, in 8.41, the expression `all(X, X:[ word != 'nil' ])` tells Shalmaneser, "Find every `X` such that `X` matches `[ word != 'nil' ]`." The return vector `< X.word >` now tells Shalmaneser to return the `word` property of every `X` that was found. And the result is what we wanted — a vector of every word in the sentence.

(8.35)
```
feature bag_of_words
all(X, X:[ word != 'nil' ])
< X.word >
```

It is possible to use `all()` around a part of an expression, not just around the entire thing.

(8.36)
```
feature ancestry
X:[ is = 'current' ] and
all(Y, Y:[ ] \* X)
< X.cat, Y.cat >
```

In evaluating 8.40, Shalmaneser will:

1. Find the current node and assign it to the variable `X`. (This part of the expression is outside the scope of `all()`, so only one match will be found.)
2. Find every node `Y` such that `Y` dominates `X`. (This part of the expression is inside the scope of `all()`, and `Y` is the variable argument of `all()`, so Shalmaneser won't stop until every node `Y` that meets the criteria has been found.)
3. Return the `cat` property of `X` (a single value), followed by the `cat` property of every `Y` (a list of values).

Exercise Write feature detectors that return:

1. The `word` properties of all the current node's children.
2. The `cat` of the current node, followed by the `cat` of each of its children.
3. Every word that appears *in between* the current and target nodes. (Define a node `X` as being between nodes `A` and `B` if `X <* A` and `X >* B` or vice versa. Remember, the current node may come before *or* after the target node.)

8.6.3 Numeric features — measuring lists of nodes with `< X.size >`

In addition to listing the features of multiple nodes using `all()`, we can *count* those lists of nodes. For any variable V, if the expression `all(`V`,` *expression*`)` appears in a feature detector, then the return vector `<` V`.size >` returns the number of times *expression* matched.

Here, for instance, is a feature detector that will return the number of nodes in a sentence. It is the same as 8.41 except for the return vector; it works by matching all the same nodes as 8.41, but then returning the number of nodes matched rather than the `word` property of each node.

(8.37)
```
feature word_count
all(X, X:[ word != 'nil' ])
< X.count >
```

Exercise Based on the last exercise, write feature detectors that return:

1. The number of children of the current node.
2. The number of words that appear in between the current and target nodes. (Define 'between' as before. Remember that there may be non-terminal nodes in between the current and target nodes; we don't want to count these.)

8.6.4 Delimiters

We've seen how the `ALL()` operator lets us return lists of variable length. And we've seen, too, that it's sometimes useful to pack more than one piece of information into the same feature (for example, returning the current word *and* its parent). In fact, there are times when we will want to do both, returning *multiple* lists of variable length. For instance, the *path* features defined in §5.3.1.4 return two lists' worth of information — first, information about the nodes on the path *up* to the LCA, and then, information about the nodes on the path *down*.

But when you try to cram two lists' worth of information into one return vector, a problem arises — how do you know when one list ends and the next begins? For this, we use delimiters. A delimiter is simply a string of text surrounded by quotation marks. Delimiters can appear

(8.38)
```
< X.word, "&", Y.word >
```

If `X` is bound to an expression that only matched the word *apple*, and `Y` is bound to an expression that only matched the word *pear*, this return vector will evaluate to `<apple, &, pear>`.

If `X` is bound (using `ALL()` to an expression that matched the words *apple*, *peach* and *grape-fruit*, this same return vector will evaluate to `<apple, peach, grapefruit, &, pear>`. As these examples show, the location of the delimiter makes it apparent — both to us and to the machine learning tools that will eventually use these features — where the `X` values end and the `Y` values begin.