Artificial Intelligence through Prolog by Neil C. Rowe

Representing facts

If we want computers to act intelligent, we must help them. We must tell them all the common-sense *knowledge* we have that they don't. This can be hard because this knowledge can be so obvious to us that we don't realize that a computer doesn't know it too, but we must try.

Now there are many different kinds of knowledge. Without getting deep into philosophy (or specifically *epistemology*, the theory of knowledge), there are two main kinds: facts and reasoning procedures. Facts are things true about the world, and reasoning procedures (or *inferences*) are ways to follow reasoning chains between facts. Since facts are easier to represent than procedures, we'll consider them first, and postpone procedures to Chapter 4.

Predicates and predicate expressions

To talk about facts we need a "language". Artificial intelligence uses many languages and sub-languages. But in this introductory book we don't want to confuse you. We'll use only one, simple (*first-order*) *predicate logic* (sometimes called *predicate calculus* and sometimes just *logic*). And we'll use a particular notation compatible with the computer programming language Prolog | REFERENCE 1|. .FS | REFERENCE 1| In this book we use a subset of the "standard Prolog" in Clocksin and Mellish, *Programming in Prolog*, second edition, Springer-Verlag, 1984. For a complete description of what we use, see Appendix D. .FE Prolog isn't predicate logic itself; computer languages try to do things, whereas logic just says that certain things are true and false. But Prolog does appear close to the way logic is usually written. That is, its *grammar* or *syntax* or form is that of logic, but its *semantics* or meaning is different.

And what is that grammar? Formally, a *predicate expression* (or *atomic formula*, but that sounds like a nuclear weapons secret) is a name--a *predicate*--followed by zero or more arguments enclosed in parentheses and separated by commas (see Figure 2-1) | REFERENCE 2|. .FS | REFERENCE 2| Several terms closely related to "predicate expression" are used in the logic and artificial-intelligence literature. A *literal* is like a predicate expression only it can have a negation symbol in front of it (negations will be explained in Section 3.6). A *structure* or *compound term* is like a predicate expression only it isn't necessarily only true or false. A *logical formula* is a structure or a set of structures put together with "and"s, "or"s, and "not"s. .FE Predicate names and arguments can be composed of any mixture of letters and numbers, except that predicate names must start with a lower-case letter. (Upper-case letters first in a word have a special meaning in Prolog, as we'll explain shortly.) The underscore symbol "_" also counts as a letter, and we will often use it to make names more readable. So these are all predicate expressions:

```
p(x)
q(y,3)
r(alpha,-2584,beta)
city(monterey,california)
tuvwxy(abc345)
noarguments
pi(3.1416)
long_predicate_name(long_argument_name,3)
```

We can put predicate expressions like these into computers. They can represent facts true about the world.

But what exactly do these expressions *mean* (their *semantics*)? Actually, anything you want--it's up to you to assign reasonable and consistent interpretations to the symbols and the way they're put together, though there are some conventions. The better job you do, the more reasonable the conclusions you'll reach from all these facts.

Predicates indicating types

Predicates can mean many things. But they do fall into categories. We summarize the major categories in Figure 2-2.

One thing they can mean is something like data-type information in a language like Pascal or Ada. Except that in artificial intelligence there are generally a lot more types than there are in most programming, because there must be a type for every category in the world that we want the computer to know about.

For instance, suppose we want the computer to know about some U.S. Navy ships | REFERENCE 3|. We .FS | REFERENCE 3| The occasional use of military examples in this book is deliberate: to serve as a reminder that much artificial intelligence work in the United States has been, and remains, supported by the military. We make no endorsements. .FE could tell it

```
ship(enterprise).
```

to say that the Enterprise is a ship (remember we must use lower case). Or in other words, the Enterprise is an example of the "ship" type. We will put periods at the end of facts because Prolog uses the period to signal the end of a line. We could also tell the computer

```
ship(kennedy).
ship(vinson).
```

to give it the names of two more ships--two more things of the "ship" type. Here **ship** is a *type predicate*. If we knew code numbers for planes we could tell the computer about them too, using the code numbers as names:

```
plane(p54862).
plane(p79313).
```

Similarly, we can label people with types:

```
commodore(r_h_shumaker).
president(r_reagan).
```

and label more abstract things like institutions:

```
university(naval_postgraduate_school).
university(stanford_university).
```

and label concepts:

```
day_of_week(monday).
day_of_week(tuesday).
day_of_week(wednesday).
```

A thing can have more than one type. For instance:

```
ship(enterprise).
american(enterprise).
```

And types can have subtypes:

```
carrier(vinson).
ship(carrier).
```

These are all *type predicates*, and they are all have one argument. The argument is the name of some thing in the world, and the predicate name is the class or category it belongs to. So the predicate name is *more general* than the argument name; this is usual for predicate names in artificial intelligence. So it wouldn't be as good to say

```
enterprise(ship).
kennedy(ship).
```

About types

We've said these predicates are like the types in computer languages, but there are some differences. The main one is that they need never be defined anywhere. If for instance we are using Pascal, we either use the built-in types (integer, real, character, array, and pointer) or define the type we want in terms of those built-in types. But for artificial intelligence, the type (predicate) names are just arbitrary codes used in lookup. This is because you can put integers and characters in a computer, but not a ship. You can't even put in a full representation of a ship, or a full representation of any other real object--real objects have too many complexities, while integers and characters are abstractions.

How then, if we expect the computer to be intelligent, will it ever know what a ship is? Much the way people know. Ships are defined in a dictionary using the concept of a vehicle, the concept of water, the concept of floating, and so on. A dictionary might say a ship is "an oceangoing vessel". But it might define "vessel" as a "craft for travelling on water", and "craft" as an "individual ship"--so the definitions are circular, as all dictionary definitions are sooner or later. But we can indirectly figure out what is being talked about by the secondary words like "oceangoing" and "travelling". So words must be defined in terms of one another.

So we won't expect each type predicate to be *implemented* (that is, understood by a computer) by a separate procedure or processing routine. The same holds for arguments. In fact, we could store all predicate names and arguments the same way in the computer, as characters. This is a bit wasteful of computer storage space--so some Prolog dialects do store numbers differently--but there's nothing wrong philosophically with it.

Good naming

So predicate and argument names can be arbitrary; we just have to remember what they represent. But one name can be better than another, if it is easier to remember what it means. Writing facts for an artificial-intelligence program to use is a kind of programming, and we should follow the usual rules of good programming style. In choosing names, we suggest these guidelines:

1. As much as possible, use everyday English words for names. If you need more than one word, use the underscore character between them for clarity, like in **day_of_week** (though sometimes you can leave out the underscores like in **dayofweek** when the reading is reasonably clear).

2. Choose names that describe their function precisely. For instance, use **day_of_week** instead of **day**, which could describe both **monday** and **october_19_1985**.

3. Avoid names with multiple meanings. For instance, if there is a Commander Kennedy as well as a ship named Kennedy, include the first initial of the person; or if you call the Enterprise a "ship", don't also say that a unit "shipped" somewhere.

4. Avoid numbers in names, with two exceptions: arithmetic (see Chapter 5) and closely related variables and predicates (like **X** and **X2** in Section 5.5 and **iterate** and **iterate2** in Section 10.8).

5. Abbreviate only when absolutely necessary. Since artificial intelligence programs often use many names, abbreviations can be confusing.

6. Predicate names should be more general than their argument names, but not so general that they don't really mean anything (for then facts can't be indexed well).

7. A few predicate names are reserved or "special" to Prolog, so you can't use them for your own predicates.

8. Of course, always use the same name for the same thing.

Property predicates

We can tell the computer (or *assert*):

```
ship(enterprise).
gray(enterprise).
big(enterprise).
```

which could mean "The Enterprise is a ship, it is gray, and it is big." (Never mind that "big" is vague; we could define it as "more than 800 feet long", and "gray" and even "ship" are vague to a lesser extent (is a toy ship a ship? and is an imaginary ship a ship?), and much human knowledge is vague anyway.) Or this could mean the Enterprise is a member of the class of ships, a member of the class of gray things, and a member of the class of big things. But those last two phrases awkward. "Gray" and "big" are adjectives, not nouns like "ship", and they should be treated differently.

So we'll represent properties of objects as *property predicate expressions*, two-argument expressions in which the predicate name is the name of a property, the first argument is the name of an object, and the second argument is the value of the property. The preceding example could be rewritten better as:

```
ship(enterprise).
color(enterprise,gray).
size(enterprise,big).
```

This has the advantage of using predicate names that are more general. It also shows the relation between **gray** and **enterprise**, and that between **big** and **enterprise**: **color** and **size** are the property names for which **gray** and **big** are the values. So we've made some implicit (unstated "common-sense") knowledge explicit (stated), a key goal in artificial intelligence.

Again, the computer won't actually know what **gray** and **big** mean if we type in the preceding three example lines; those are just codes that it uses for comparison. For instance, if the computer also knows

```
color(kennedy,gray).
```

then it knows the Enterprise and the Kennedy have the same color, though it doesn't know what a "color" is (but don't blame it, because most computers don't have eyes).

An important class of property predicates concern space and time. For instance

```
location(enterprise,14n35e).
last_docking(enterprise,16feb85).
```

could mean that the Enterprise is currently at latitude 14N and longitude 35E, and its last docking was on February 16, 1985.

Predicates for relationships

Perhaps the most important predicates of all relate two different things. Such *relationship predicates* are important because a lot of human reasoning seems to use them--people need to relate ideas. For instance, we can use a **part_of** predicate of two arguments which says that its first argument is a component within its second argument. We could give as facts:

```
part_of(enterprise,u_s_navy).
part_of(u_s_navy,u_s_government).
part_of(naval_postgraduate_school,u_s_government).
part_of(propulsion_system,ship).
```

In other words, the Enterprise is part of the U.S. Navy, the Navy is part of the U.S. government, the Naval Postgraduate School is part of the U.S. government, and the propulsion system is part of a ship. An **owns** relationship predicate can say that something is owned by someone:

```
owns(tom,fido).
owns(tom,toms_car).
```

These facts say that Tom owns two things: something called **fido**, and an unnamed car which we can just refer to as **toms_car**.

It's easy to get confused about argument order in relationship predicate expressions. So we'll try to follow this convention: if the predicate name is inserted between the two arguments, the result will be close to an English sentence giving the correct meaning. So if we insert "owns" between "tom" and "fido" we get "Tom owns Fido", and if we insert "part of" between "enterprise" and "u. s. navy" we get "Enterprise part of U. S. Navy".

An important class of relationship predicates relates things in space and time. A real-world object can be north, south, east, west, etc. of another object. Viewed by a fixed observer, an object can also be right, left, above, below, in front, and behind another object. We can describe a picture with these predicates. Similarly, an event in time can be before, after, during, overlapping, or simultaneous with another event, so we can describe history with these predicates.

Relationship predicates can describe relationships between people. For instance, the **boss_of** relationship is important for describing bureaucracies, an important application of artificial intelligence. It says that a person (first argument) is the boss of another person (second argument), and this shows direction of responsibility.

People can also be related by kinship relationship predicates (**father**, **mother**, **child**, **uncle**, **cousin**, **stepfather**, **half-brother**, **grandfather**, etc.). People can also be related with **friend** and **acquaintance** relationship predicates.

Besides all these, another special relationship predicate is frequently used in artificial intelligence. It's called **a_kind_of** or **is_a** (we prefer the first name, because "is" is vague), and it can replace all type predicates. Its first argument is a thing, and its second argument is the type of that thing (the predicate name in the one-argument form considered before). For instance:

```
a_kind_of(enterprise,ship).
a_kind_of(tanker,ship).
a_kind_of(tuesday,day_of_week).
```

which says that the Enterprise is a kind of ship, a tanker is a kind of ship, and Tuesday is a kind of day of the week | REFERENCE 4|. .FS | REFERENCE 4| Some researchers don't agree with this use of **a_kind_of**. They think that the first two facts should have different predicate names since the Enterprise is an individual while tankers are a group of individuals; often they'll use the predicate name **element** for the "Enterprise" fact, and keep **a_kind_of** for the "tanker" fact. But a set whose size is 1 is still a set, and there doesn't seem to be anything fundamentally different between restricting the body type of a ship to be a tanker and restricting the name of a ship to be the word "Enterprise"--it just happens that people try, not always successfully, to make names unique. Researchers who argue against this may be getting this issue confused with the important "extensions vs. intensions" problem which we'll discuss in Section 12.8. .FE Some reasoning is easier with this two-argument form than the equivalent one-argument form.

There are other predicates, but as any psychotherapist will tell you, relationships are the key to a happy life.

Semantic networks

Pictures can make a complicated set of facts a lot clearer. There's a simple pictorial way to show the predicate expressions we've been discussing: the *semantic network*. Unfortunately, there is a major restriction: semantic networks can only directly represent predicates of two arguments (so type predicates must be in the two-argument form) | REFERENCE 5|. .FS | REFERENCE 5| But we can represent predicate expressions with more than two arguments indirectly, as sets of two-argument predicate expressions. .FE

A semantic network is what computer scientists call a a *labeled directed graph* (see Appendix C for a definition). We make every possible fact argument a small named circle (node) in the graph. For each two-argument fact, we draw an arrow (edge) from the circle for its first argument to the circle for its second argument, and label the arrow with the predicate name. So the fact **p(a,b)** is represented as an arrow from a circle labeled "a" to a circle labeled "b", with the arrow itself labeled "p". If for instance our facts are:

```
a_kind_of(enterprise,ship).
a_kind_of(kennedy,ship).
part_of(enterprise,u_s_navy).
part_of(kennedy,u_s_navy).
part_of(u_s_navy,u_s_government).
a_kind_of(u_s_government,government).
color(ship,gray).
location(enterprise,15n35e).
has(u_s_government,civil_service_system).
```

then our semantic network looks like Figure 2-3.

Getting facts from English descriptions

Usually programmers building artificial intelligence programs don't make up facts themselves. Instead, they look up facts in documents and books, or ask people knowledgeable about the subject ("experts") to tell them what they need--the process of *knowledge acquisition*. But English and other "natural languages" are less precise than computer languages (though more flexible), and the programmer must be careful to get the meanings right.

The sorts of facts we've considered so far are usually often signalled by the verb "to be" in English (**is**, "are", "was", "were", "will be", "being", and so on). For instance:

> The Enterprise is a ship.
> A ship is a vehicle.
> The Enterprise is part of the U.S. Navy.
> A ship is gray.

Here "to be" is used for type predicates (the first and second sentences), a **part_of** relationship predicate (the third sentence), and a property predicate (the fourth sentence). Plurals can also be used:

> Ships are gray.
> Ships are vehicles.

English verbs with narrower meanings can be used too:

> The Enterprise belongs to the U.S. Navy.
> The Enterprise has a hull.
> They color ships gray.
> The Enterprise is located at 15N35E.

The first two suggest **part_of** relationship predicates, and the last two are property predicates.

Predicates with three or more arguments

You can have as many arguments to a predicate as you want if you're not concerned about easily representing them in a semantic network. One idea is to include multiple property values in a single fact, much like adjectives and adverbs modifying a noun or verb. So for instance we could put everything we know about a ship together:

```
ship_info(enterprise,15n35e,1200,16feb85,gray,j_kirk).
```

which we could read as "The Enterprise is a ship that was at 15N25E at 12 noon on February 16, 1985, and its color is gray, and its captain is J. Kirk." To interpret such facts we need to keep a description somewhere of the properties and their order within the arguments.

These sort of predicates define a *relational database* of facts. Much research has studied efficient implementation and manipulation of such databases. The information about properties and their order for each such predicate is called a *database schema*.

Another important category of predicates with often many arguments (though they can also have just two) is that representing results of actions--in mathematical terminology, *functions*. Suppose we want to teach a computer about arithmetic. We could use a predicate **sum** of three numerical arguments, which says that the sum of the first two arguments is the third. We could give as facts:

```
sum(1,1,2).
sum(1,3,4).
sum(1,4,5).
sum(1,5,6).
sum(2,1,3).
sum(2,2,4).
sum(2,3,5).
sum(2,4,6).
```

And we could do this for lots of different numbers, and different arithmetic operations. Of course for this to be useful in general, we would need very many facts and this would be unwieldy (we will describe a better way in Chapter 5), but it will suffice to define operations on any finite set of numbers.

We will use function predicates frequently. To avoid confusion, we follow the convention that the last argument always represents the result of (value returned by) the function, with the exception noted in the next section | REFERENCE 6|. .FS | REFERENCE 6| If you're familiar with Lisp, be careful to include the function result as an argument to Prolog predicates. In Lisp, a value is always associated with the whole expression, something you can't do in Prolog. .FE

Functions can also be nonnumeric. An example is a function that gives, for two employees of a company, the name of the lowest-ranking boss over both of them. Since artificial intelligence emphasizes nonnumeric reasoning, you'll see more nonnumeric than numeric functions in this book.

Probabilities

We have assumed so far that facts are always completely certain. In many situations (as when facts are based on reports by people), facts are only probably true. Then we will use the mathematical idea of *probability*, the expected fraction of the time something is true. We will put an approximate probability as an optional last argument to a predicate, after the previously discussed function result if any. So for instance

```
color(enterprise,gray,0.8).
```

says that we're 80 percent sure (or sure with probability 0.8) that the Enterprise is gray. We'll ignore this topic until Chapter 8.

How many facts do we need?

An infinity of facts are true about the world. How then do we decide which to tell a computer? This question has no easy answers. Generally, you must decide what you want the computer to do. Then make sure to tell the computer in advance every fact that might be relevant to that behavior. "Libraries" of useful facts for particular subjects will help. But the smarter you want the computer to be, the more facts you must tell it. The next chapter will discuss the next question, how to get the computer to do things with facts.

Keywords:

knowledge
facts
logic
predicate calculus
Prolog
predicate
predicate expression
arguments
semantics
type predicate
assertion
property predicate
relationship predicate
part_of
a_kind_of
semantic network
knowledge acquisition
relational-database predicate
function predicate
probability predicate

Exercises

(Note: answers to exercises marked with the code "A" are given at the back of the book.)

2-1. (A,E) Which of the following facts is better knowledge representation? Explain. ("Better" means less likely to confuse people.)

```
color(enterprise,gray).
size(enterprise,big).
```

2-2. (R,A,E) Suppose you want to store facts about when and where memos were sent in an organization. Which is the best Prolog format for such facts, and why?

> ***(i) <date>(<name>,<author>,<distribution>).***
> ***(ii) memo(<name>,<date>,<author>,<distribution>).***
> **(iii) fact(memo,<name>,<date>,<author>,<distribution>).**

2-3. Draw a semantic network representing the following facts:

> **Ships are things.**
> **Carriers are ships.**
> **Ships have a position.**
> **Ships have a crew.**
> **Carriers have planes.**
> **Planes are things.**
> **A crew consists of people.**

People are things.

2-4. Represent the nonnumeric meaning of the picture in Figure 2-4 as a set of nonnumeric Prolog facts. (Hint: describe the circles and their relationships.)

2-5. (A) One-argument and two-argument predicates are very common in Prolog knowledge representation. Most of the facts you want to put into computers can be represented with them. Someone might say that this shouldn't be surprising, because most operations in mathematics are either unary (applied to a single thing like the square root operation or the sine operation), or binary (applied to two things, like addition and exponentiation). What's wrong with this comment?

2-6. (R,A) Consider the six-argument ship_info facts in Section 2.9. To represent them in a semantic network, we need to convert each to a set of two-argument facts. Explain how. Assume that six-argument facts only record the most recent position of a ship.

2-7. (A,E) Why might it be a good idea to put falsehoods (statements false in the world) into a computer?

2-8. Suppose you want to write a program that reasons like Sherlock Holmes did, about the facts of some crime to decide who is responsible. You want to represent in Prolog style the facts you find about the crime, and the reports of witnesses. The argument types used for facts will vary. But certain arguments must be included in every fact about the crime--what are they? And certain arguments must be included in every fact giving a report from a witness--what are they?

2-9. (E) Why must the representation of these two facts be fundamentally different?

Clint is mayor.
Someone is mayor.

2-10. (E) Consider the use of the word "boss" in the following facts. Suppose you wanted to represent these facts in Prolog. Would it be a good idea for any two of these to use the same word "boss" as either a predicate name or an argument name?

Mary is boss of Dick.
Dick and Mary boss their children around.
"Boss" has four letters.
A boss has managerial responsibilities.

2-11. (E) Another way to put facts inside computers is with a restricted subset of English. For instance:

The Enterprise is a ship.
The Enterprise is part of the US Navy.
The color of the Enterprise is gray.
The Enterprise is at 15N25A.

(a) Discuss the advantages of storing facts this way instead of with predicate expressions as we have done in the chapter.

(b) Give a disadvantage for efficient use of the facts.

(c) Give a disadvantage for programming errors.

Go to book index