Introduction to AI with Prolog

Files

Alexey Sery
a.seryj@g.nsu.ru

Novosibirsk State University
*THE REAL SCIENCE

НГУ ФИТ

Department of Information Technologies
Novosibirsk State University

August 31, 2022

Overview

Goals of the lesson

- How predicate definitions can be spread across different files
- How to write results to files and how to read input from files

Splitting programs over modules

By now, you have seen and you had to write lots of programs that use miscellaneous predicates you described. What you probably did each time you needed one of them was to go back to the definition and copy it over into the file where you wanted to use it. And maybe, you started thinking that it would be a lot nicer if you could define them somewhere once and for all and then just access that definition whenever you needed it.

That sounds like a pretty sensible thing to ask for and, of course, Prolog offers ways of doing it.

Splitting programs over modules

If you want Prolog to consult an extra code you have to place a list of files at the beginning of your program like this:

```
:- [sourceCode1, sourceCode2, ...].
```

Doing so you are telling Prolog to consult the files in the square brackets before reading in the rest of the file.

On encountering something of such the form, Prolog just goes ahead and consults the files without checking whether the file really needs to be consulted. If the predicate definitions provided by one of the files are already available, because it already was consulted once, Prolog still consults it again, overwriting the definitions in the database.

Splitting programs over modules

If you want Prolog to consult an extra code you have to place a list of files at the beginning of your program like this:

```
:- ensure_loaded([sourceCode1, sourceCode2, ...]).
```

The inbuilt predicate `ensure_loaded/1` behaves more clever in this case and it is what you should usually use to load predicate definitions given in some other file into your program.

On encountering the following directive `:- ensure_loaded([sourceCode1, sourceCode2]).` Prolog checks whether the files `sourceCode1.pl` and `sourceCode2.pl` have already been loaded. If not, Prolog loads them. If they already are loaded in, Prolog checks whether something has changed since last loading them and if that is the case, Prolog loads them, if not, it doesn't do anything and goes on processing the program.

Splitting programs over modules

Imagine that you are writing a program that needs two predicates `f/1` and `g/2`. You described `f/1` and `g/2` in the files `source1.pl` and `source2.pl` respectively. Now you can load this two files as we did before by putting

```
:- ensure_loaded([source1, source2]).
```

at the top of your next program. Now, imagine that each of them depends on predicate `some_helper/2`, which is defined both in `source1.pl` and `source2.pl`.

Unfortunately, there seem to be problems this time. You get a message that looks something like:

```
The procedure some_helper/2 is being redefined
Do you really want to redefine it?  (y, n, p, or ?)
```

So what has happened?

Both files are defining the predicate `some_helper/2`. And what's worse, you can't be sure that the predicate is defined in the same way in both files. So, you can't just say «yes, override», since `f/1` depends on the definition of `some_helper/2` given in file `source1.pl` and `g/2` depends on the definition given in file `source2.pl`.

Furthermore, note that you are not really interested in the definition of `some_helper/2` at all. You don't want to use it. The predicates that you are interested in, that you want to use are `f/1` and `g/2`. They need definitions of `some_helper/2`, but the rest of your program doesn't.

Splitting programs over modules

Instead of consulting the whole file we can turn it into a *module*.

Modules allow you to encapsulate predicate definitions. You are allowed to decide what is public and what is private in the module.

Public predicates are callable from outside of the module, while private are not. You are not allowed to call private predicates from somewhere except of the module itself, but there will be no conflicts if multiple modules internally define the same predicate.

In our example `some_helper/2` is a good candidate for becoming a private predicate.

Splitting programs over modules

You can turn a source file into a module by putting a declaration of the following form at the top of that file:

```
:- module(Name, ListOfPublicPredicates).
```

This declaration specifies the name of the module and the list of predicates that should be declared public. That is, the list of predicates that you want to export.

By putting

```
:- module(source1, [f/1]).  or
:- module(source2, [g/2]).
```

at the top of the corresponding file we define our modules. Predicate `some_helper/2` is now hidden, so there is no clash when loading both modules at the same time.

Splitting programs over modules

Modules are loaded with the built-in predicate `use_module/1`. Putting

```
:- use_module(source1).
:- use_module(source2).
```

at the top of a source file will import all public predicates from both modules.

Moreover, if you don't need all public predicates of a module, you can use the binary predicate `use_module/2`, which takes the list of predicates you want to import from a module.

```
:- use_module(source1, [f/1]).
:- use_module(source2, [g/2]).
```

Splitting programs over modules

Many of the very common predicates are actually predefined in most Prolog implementations in one way or another.

If you have been using SWI Prolog, for example, you will probably have noticed that things like append and member are built in. That's a specialty of SWI, however. Other Prolog implementations, like Sicstus for example, don't have them built in. But they usually come with a set of libraries, i.e. modules defining common predicates.

These libraries can be loaded using the normal commands for importing modules. When specifying the name of the library that you want to use, you have to tell Prolog that this module is a library, so that Prolog knows where to look for it (namely, not in the directory where your other code is, but at the place where Prolog keeps its libraries).

Putting

```
:- use_module(library(libname)).
```

at the top of your file tells Prolog to load a library having a name `libname`.

Note, however, that the way libraries are organized and the inventory of predicates provided by libraries are by no means standardized across different Prolog implementations. In fact, the library systems may differ quite a bit.

So, if you want your program to run with different Prolog implementations, it might be easier and faster to define your own library modules (using the techniques that we saw in the last section) than to try to work around all the incompatibilities between the library systems of different Prolog implementations.

Writing to and reading from files

Now, we know how to load program code from files, and we want to look at reading an input data from files and outputting results to files.

Before we can perform any actions on a file, we have to open it and associate a stream with it. Streams have names, that we can use to specify which stream to write or read from. Prolog assigns these names to streams, and all you need to do is bind them to a variables and then pass these variables around.

The built-in predicate `open/3` opens a files and assign a stream with it.

```
open(+FileName, +Mode, -Stream)
```

The first argument is the name of the file, and in the last argument, Prolog returns the name this it assigns to the stream. Mode is one of *read*, *write* or *append*.

When you are finished working with file, you should close it again. This is done by the built-in predicate `close/1`. It takes the name of the stream associated with file.

Working-with-file session usually has the following structure:

```
open(myfile, write, Stream).
...
write something to the file
...
close(Stream).
```

Writing to and reading from files

The predicates for writing to and reading from a stream are almost the same as the ones we already used. We have `write`, `writeln`, `read`, `nl`, etc. The only thing that is different is that we always give the stream that we want to write to or read from as the first argument.

```
readFile :-   open('in.txt',read,In),
              \+ at_end_of_stream(In),
              read(In,R),
              writeln(R),
              close(In).
writeFile :-  open('../out.txt',write,Out),
              write(Out, 'Some string.'),
              write(Out, 'Skin(Smith,Black).'),
              close(Out).
```