Lecture 12: Working with Files

- This lecture is concerned with various aspects of file handling and modularity
- We will learn three things:
  - How predicate definitions can be spread across different files
  - How to write modular software systems
  - How to write results to files and how read input from files

Splitting programs over files

- Many Prolog predicates make use of the same basic predicates
  - For instance: **member/2**, **append/3**
- Of course you do not want to redefine it each time you need it
  - Prolog offers several way of doing this

Reading in Programs

- The simplest way of telling Prolog to read in predicate definitions that are stored in a file is using the square brackets

```
?- [myFile].
{consulting(myFile.pl)…}
{myFile.pl consulted, 233 bytes}
yes
?-
```

Reading in Programs

- You can also consult more than one file at once

```
?- [myFile1, myFile2, myFile3].
{consulting myFile1.pl…}
{consulting myFile2.pl…}
{consulting myFile3.pl…}
```

Reading in Programs

- You don`t need to do this interactively
- Instead, you can use a directive in the database

```
:- [myFile1, myFile2].
```

Reading in Programs

- Maybe several files, independently, consult the same file
- Extra check whether predicate definitions are known already: ensure_loaded/1

```
:- ensure_loaded([myFile1, myFile2]).
```

Modules

- Imagine you are writing a program that manages a movie database
- You designed two predicates:
  - **printActors/1**
  - **printMovies/1**
- They are stored in different files
- Both use an auxiliary predicate:
  - **displayList/1**

The file printActors.pl

```
% This is the file:  printActors.pl

printActors(Film):-
    setof(Actor,starring(Actor,Film),List),
    displayList(List).

displayList([]):- nl.
displayList([X|L]):-
     write(X), tab(1),
     displayList(L).
```

The file printMovies.pl

```
% This is the file:  printMovies.pl

printMovies(Director):-
    setof(Film,directed(Director,Film),List),
    displayList(List).

displayList([]):- nl.
displayList([X|L]):-
     write(X), nl,
     displayList(L).
```

The file main.pl

```
% This is the file main.pl

:- [printActors].
:- [printMovies].
```

The file main.pl

```
% This is the file main.pl


:- [printActors].
:- [printMovies].
```

```
?- [main].
```

The file main.pl

```
% This is the file main.pl


:- [printActors].
:- [printMovies].
```

```
?- [main].
{consulting main.pl}
```

The file main.pl

```
% This is the file main.pl


:- [printActors].
:- [printMovies].
```

```
?- [main].
{consulting main.pl}
{consulting printActors.pl}
```

The file main.pl

```
% This is the file main.pl


:- [printActors].
:- [printMovies].
```

```
?- [main].
{consulting main.pl}
{consulting printActors.pl}
{printActors.pl consulted}
```

The file main.pl

```
% This is the file main.pl


:- [printActors].
:- [printMovies].
```

```
?- [main].
{consulting main.pl}
{consulting printActors.pl}
{printActors.pl consulted}
{consulting printMovies.pl}
```

The file main.pl

```
% This is the file main.pl


:- [printActors].
:- [printMovies].
```

```
?- [main].
{consulting main.pl}
{consulting printActors.pl}
{printActors.pl consulted}
{consulting printMovies.pl}
The procedure displayList/1 is
    being redefined.
Old file: printActors.pl
New file: printMovies.pl
Do you really want to redefine it?
(y, n, p, or ?)
```

Libraries

- Many of the most common predicates are predefined by Prolog interpreters
- For example, in SWI prolog, **member/2** and **append/3** come as part of a library
- A library is a module defining common predicates, and can be loaded using the normal predicates for importing modules

Importing Libraries

- When specifying the name of a library you want to use, you can tell that this module is a library
- Prolog will look at the right place, namely a directory where all libraries are stored

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

Writing to Files

- In order to write to a file we have to open a stream
- To write the string **'Hogwarts'** to a file with the name **hogwarts.txt** we do:

```
…
open(‘hogwarts.txt‘, write, Stream),
write(Stream, 'Hogwarts').
close(Stream),
…
```

Appending to Files

- To extend an existing file we have to open a stream in the append mode
- To append the string **'Harry'** to the file with the name **hogwarts.txt** we do:

```
...
open(‘hogwarts.txt‘, append, Stream),
write(Stream, 'Harry').
close(Stream),
...
```

Writing to files

- Summary of predicates:
  - open/3
  - write/2
  - close/1
- Other useful predicates:
  - tab/2
  - nl/2
  - format/3

Reading from Files

- Reading information from files is straightforward in Prolog if the information is given in the form of Prolog terms followed by full stops
- Reading information from files is more difficult if the information is not given in Prolog format
- Again we use streams and the open

Example: reading from files

- Consider the file houses.txt:

```
% houses.txt
gryffindor.
hufflepuff.
ravenclaw.
slytherin.
```

- We are going to write a Prolog program that reads this information and displays it on the screen

Example: reading from files

- a Prolog program that reads this information and displays it on the screen:

```
% houses.txt
gryffindor.
hufflepuff.
ravenclaw.
slytherin.
```

```
main:-
    open('houses.txt',read,S),
    read(S,H1),
    read(S,H2),
    read(S,H3),
    read(S,H4),
    close(S),
    write([H1,H2,H3,H4]), nl.
```

Reading from files

- Summary of predicates
  - open/3
  - read/2
  - close/1
- More on read/2
  - The read/2 predicate only works on Prolog terms
  - Also will cause a run-time error when one tries to read at the end of a file

Reaching the end of a stream

- The built-in predicate **at_end_of_stream/1** checks whether the end of a stream has been reached
- It will succeed when the end of the stream (given to it as argument) is reached, otherwise if will fail
- We can modify our code for reading in a file using this predicate

Using at_end_of_stream/1

```
main:-
    open('houses.txt',read,S),
    readHouses(S,Houses),
    close(S),
    write(Houses), nl.

readHouses(S,[]):-
     at_end_of_stream(S).

readHouses(S,[X|L]):-
     \+ at_end_of_stream(S),
     read(S,X),
     readHouses(S, L).
```

With green cuts

```
main:-
    open('houses.txt',read,S),
    readHouses(S,Houses),
    close(S),
    write(Houses), nl.

readHouses(S,[]):-
     at_end_of_stream(S), !.

readHouses(S,[X|L]):-
     \+ at_end_of_stream(S), !,
     read(S,X),
     readHouses(S, L).
```

With a red cut

```
main:-
    open('houses.txt',read,S),
    readHouses(S,Houses),
    close(S),
    write(Houses), nl.

readHouses(S,[]):-
     at_end_of_stream(S), !.

readHouses(S,[X|L]):-
     read(S,X),
     readHouses(S, L).
```

Reading arbitrary input

- The predicate **get_code/2** reads the next available character from the stream
  - First argument: a stream
  - Second argument: the character code

- Example: a predicate **readWord/2** that reads atoms from a file

Using get_code/2

```
readWord(Stream,Word):-
    getcode(Stream,Char),
    checkCharAndReadRest(Char,Chars,Stream),
    atom_codes(Chars,Word).

checkCharAndReadRest(10, [], _):- !.
checkCharAndReadRest(32, [], _):- !.
checkCharAndReadRest(-1, [], _):- !.
checkCharAndRest(Char,[CharlChars],S):-
      get_code(S,NextChar),
      checkCharAndRest(NextChar,Chars,S).
```

Further reading

- Bratko (1990): Prolog Programming for Artificial Intelligence
  - Practical applications
- O`Keefe (1990): The Craft of Prolog
  - For advanced Prolog hackers
- Sterling (1990): The Art of Prolog
  - Theoretically oriented