File Operations (11A)

Young W. Lim
4/2/14

based on the following document:
http://www.learnprolognow.org/ Learn Prolog Now!

Reading in Programs

A Prolog source file is a plain text file containing a Prolog program or part thereof. Prolog source files come in three flavours:

A **traditional : consult, ensure_loaded**
Prolog source file contains Prolog clauses and directives, but no module declaration (see module/1). They are normally loaded using **consult**/1 or **ensure_loaded**/1. Currently, a non-module file can only be loaded into a single module.

A **module : use_module**
Prolog source file starts with a module declaration. The subsequent Prolog code is loaded into the specified module, and only the exported predicates are made available to the context loading the module. Module files are normally loaded with **use_module**/[1,2].

An **include : include**
Prolog source file is loaded using the **include**/1 directive, textually including Prolog text into another Prolog source. A file may be included into multiple source files and is typically used to share declarations such as **multifile** or dynamic between source files.

Consult, List Abbreviation

consult(:File)
Read File as a Prolog source file. Calls to consult/1 may be **abbreviated** by just typing a number of filenames in a list. Examples:

```
?- consult(load).      % consult load or load.pl
?- [library(lists)].   % load library lists
?- [user].        % Type program on the terminal
```

The predicate consult/1 is equivalent to **load_files**(File, []), except for handling the special file user, which reads clauses from the terminal. See also the **stream(Input)** option of **load_files**/2. Abbreviation using ?- [file1,file2]. does not work for the empty list ([]). This facility is implemented by defining the list as a predicate. Applications may only rely on using **the list abbreviation** at the Prolog *toplevel* and in *directives*.

```
[File1,File2,...,FileN].       % the list abbreviation

:- [File1,File2,...,FileN].    % directive
```

load_files

load_files(:Files, +Options)
The predicate load_files/2 is the parent of all the other loading predicates except for include/1. It currently supports a subset of the options of Quintus load_files/2. Files is either a single source file or a list of source files. The specification for a source file is handed to absolute_file_name/2. See this predicate for the supported expansions. Options is a list of options using the format OptionName(OptionValue).

stream(Input)
This SWI-Prolog extension compiles the data from the stream Input. If this option is used, Files must be a single atom which is used to identify the source location of the loaded clauses as well as to remove all clauses if the data is reconsulted.

This option is added to allow compiling from non-file locations such as databases, the web, the user (see consult/1) or other servers. It can be combined with format(qlf) to load QLF data from a stream.

options

autoload(Bool)
derived_from(File)
dialect(+Dialect)
encoding(Encoding)
expand(Bool)
format(+Format)
if(Condition)
imports(Import)
modified(TimeStamp)
must_be_module(Bool)
qcompile(Atom)
redefine_module(+Action)
reexport(Bool)
register(Bool)
sandboxed(Bool)
scope_settings(Bool)
silent(Bool)
stream(Input)

Modules

Modules essentially allow you to hide predicate definitions.
    public predicates
    private predicates

making a file into a module by putting a module declaration at the top.

Module declarations

```
:- module(ModuleName, List_of_Predicates_to_be_Exported).
```

the name of the module
the list of public predicates
  these will be the only predicates that are accessible from outside the module.

Modules

printActors.pl

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

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

redefinition

printMovies.pl

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

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

main.pl

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

Modules

printActors.pl

```
:- module(printActors, [printActors/1]).

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

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

printMovies.pl

```
:- module(printMovies, [printMovies/1]).

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

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

main.pl

```
:- use_module(printActors).
:- use_module(printMovies).
```

Directives

Specifies directives to run at load time

A rule without a head

:- predicates to be executed.

A **directive** is an instruction to the compiler.
Directives are used to set (predicate) properties, set flags and load files.
Directives are terms of the form **:- <term>.** .

Here are some examples:

```
:- use_module(library(lists)).
:- dynamic
      store/2.              % Name, Value
```

store/2 is a built-in predicate that can be modified during program execution

The directive **initialization**/1 can be used to run arbitrary Prolog goals.
The specified goal is started after loading the file in which it appears has completed.

SWI-Prolog compiles code as it is read from the file, and directives are executed as goals.
This implies that directives may call any predicate that has been defined before the point where the directive appears. It also accepts **?- <term>.** as a synonym.

Dynamic Directives

dynamic

In Prolog, a **procedure** is either **static** or **dynamic**.

A **static procedure:**
its facts/rules are predefined at the start of execution,
and do not change during execution.
usually defined in a file which will be loaded

A **dynamic procedure**:
possible to add extra facts/rules to the procedure (**assert**/**asserta**/**assertz**)
possible to remove facts/rules using (**retract**/**retractall**)
during execution of a Prolog query
the procedure must be declared as dynamic.

```
:- dynamic likes/2.
```

Library

Libraries are **modules** defining common predicates,
and can be loaded using the normal commands for **importing modules**.

When specifying the **name of the library** that you want to use,
so that Prolog knows where to look for it (lirary directory)

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

lists.pl

to load a library called lists

Importing Modules by filenames

Predicates can be added to a module by importing them from another module.
Importing adds predicates to the **namespace** of a module.

Note that both directives take **filename(s)** as arguments.
Modules are imported based on their **filename** rather than their module name.

use_module(+Files)

lists.pl

```
:- module(shapes, []).
:- use_module(library(lists)).

flatten(cube, square).
flatten(ball, circle).
```

use_module(+File, +ImportList)

loads member/2 from the lists library and append/2 under the name **list_concat**

lists.pl

```
:- use_module(library(lists), [ member/2,
                                append/2 as list_concat
                              ]).
:- use_module(library(option), except([meta_options/3])).
```

option.pl

loads all exports from library option ***except*** for meta_options/3.

Absolute File Nam

Prolog source files are located using **absolute_file_name**/3 with the following options:

```
locate_prolog_file(Spec, Path) :-
        absolute_file_name(Spec,
                   [ file_type(prolog),
                     access(read)
                   ],
                   Path).
```

The **file_type(prolog)** option is used to determine the **extension** of the file using **prolog_file_type**/2. The default extension is **.pl**.
Spec allows for the **path alias** construct defined by absolute_file_name/3.
The most commonly used **path alias** is **library(LibraryFile)**.

The example below loads the library file **ordsets.pl**

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

Input & Output Stream

open(+**SrcDest**, +**Mode**, -**Stream**, +**Options**)

SrcDest is either an atom specifying a file, or a term `pipe(Command)', like see/1 and tell/1.
Mode is one of **read**, **write**, **append** or **update**.
Mode **append** opens the file for writing, positioning the file pointer at the **end**.
Mode **update** opens the file for writing, positioning the file pointer at the **beginning** of the file **without truncating** the file.
Stream is either a variable, in which case it is bound to an integer identifying the stream, or an atom, in which case this atom will be the stream identifier.

Options
- type(Type)
- alias(Atom)
- encoding(Encoding)
- bom(Bool)
- eof_action(Action)
- buffer(Buffering)
- close_on_abort(Bool)
- locale(+Locale)
- lock(LockingMode)
- wait(Bool)

open(+**SrcDest**, +**Mode**, ?**Stream**)

Equivalent to open/4 with an empty option list.

close(+**Stream**)

Close the specified stream. If Stream is not open, an existence error is raised. However, closing a stream multiple times may crash Prolog. This is particularly true for multithreaded applications.

If the closed stream is the current input or output stream, the terminal is made the current input or output.

Writing to files

```
...
open('hogwarts.txt', write, Stream),

write(Stream, 'Hogwarts'),

nl(Stream),

close(Stream),
...
```

write(+Term)
Write Term to the current output, using brackets and operators where appropriate.

write(+Stream, +Term)
    Write Term to Stream.

Reading to files

```
main:-
    open('houses.txt', read, Str),
    read(Str, House1),
    read(Str, House2),
    read(Str, House3),
    read(Str, House4),
    close(Str),
    write([House1, House2, House3, House4]),  nl.
```

houses.txt

```
gryffindor.
hufflepuff.
ravenclaw.
slytherin.
```

read(-**Term**)

Read the next Prolog term from the **current input stream** and **unify** it with Term. On a syntax error read/1 displays an error message, attempts to skip the erroneous term and fails. On reaching end-of-file Term is unified with the atom end_of_file.

read(+Stream, -**Term**)

Read Term from **Stream**.

Reading to files

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

read_houses(Stream,[]):-
     at_end_of_stream(Stream).

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

It's the 'not provable' operator.
It succeeds if its argument is not provable
and fails if its argument is provable.

comma	[,]	: AND
semicolon	[;]	: OR
backslash +	[\+]	: NOT

Negation, Not, \+ (1)

negation, not, \+

The concept of logical negation

The only method that Prolog can use
to tell if a proposition is false is to try to **prove** it
(from the facts and rules)

if this attempt fails, it concludes that the proposition is false.
: negation as failure

When some critical fact or rule is missing,
it will not be able to prove the proposition.

the negation as failure is only relative to the "mini-world-model"
defined by the **facts** and **rules** known to the Prolog interpreter.
: the closed-world assumption

Also, there is a possibility it takes a very long time to determine
that the proposition cannot be proven.

Negation, Not, \+ (2)

Apart from negation-as-failure, modern Prolog interpreters uses the symbol **\+** (a mnemonic for not provable)
\ : **not** and **+** : **provable**.

```
?- \+ (2 = 4).      not provable
?- not(2 = 4).      negation-as-failure
```

Arithmetic comparison operators having a **negation**
which makes it always possible to determine the falsity of the given proposition

```
?- 2 =\= 4.       negation
```

Reading to files

```
readWord(InStream,W):-
     get_code(InStream, Char),
     checkCharAndReadRest(Char, Chars, InStream),
     atom_codes(W, Chars).


checkCharAndReadRest(10,[],_):-  !.  % Line Feed

checkCharAndReadRest(32,[],_):-  !.  % Space

checkCharAndReadRest(-1,[],_):-  !.  % End of Stream

checkCharAndReadRest(end_of_file,[],_):-  !.

checkCharAndReadRest(Char,  [Char|Chars], InStream):-
     get_code(InStream,NextChar),
     checkCharAndReadRest(NextChar, Chars, InStream).
```

if the read char is Line Feed or Space, or End of Stream, then a complete word has been read, otherwise the next character is read.

Primitive Character I/O (1)

nl, nl(+Stream)   Write a newline character put(10).

put_byte(+Byte), **put_byte**(+Stream, +Byte)
put_char(+Char), **put_char**(+Stream, +Char)
put_code(+Code), **put_code**(+Stream, +Code)

tab(+Amount), **tab**(+Stream, +Amount)

flush_output, **flush_output**(+Stream), **ttyflush**

get_byte(-Byte), **get_byte**(+Stream, -Byte)
get_code(-Code), **get_code**(+Stream, -Code)
get_char(-Char), **get_char**(+Stream, -Char)

peek_byte(-Byte), **peek_byte**(+Stream, -Byte)
peek_code(-Code), **peek_code**(+Stream, -Code)
peek_char(-Char), **peek_char**(+Stream, -Char)

(+Stream, +Len, -String)

skip(+Code), **skip**(+Stream, +Code)
    Read the input until Code or the end of the file is encountered.

Primitive Character I/O (2)

get_single_char(-Code)
Unlike get_code/1, this predicate does not wait for a return. The character is not echoed to the user's terminal. This predicate is meant for keyboard menu selection, etc.

at_end_of_stream, **at_end_of_stream**(+Stream)

set_end_of_stream(+Stream)

copy_stream_data(+StreamIn, +StreamOut, +Len)
Copy Len codes from StreamIn to StreamOut.
copy_stream_data(+StreamIn, +StreamOut)
Copy all (remaining) data from StreamIn to StreamOut.

read_pending_input(+StreamIn, -Codes, ?Tail)
Read input pending in the input buffer of StreamIn and return it in the difference list Codes-Tail. That is, the available characters codes are used to create the list Codes ending in the tail Tail. This predicate is intended for efficient unbuffered copying and filtering of input coming from network connections or devices.

Atom (1)

Predicates to convert between Prolog constants and lists of character codes.

Converting from a constant to a list of character codes

- atom_codes/2
- number_codes/2
- name/2

Converting from a list of character codes to a constant

- atom_codes/2 will generate an atom
- number_codes/2 will generate a number or exception
- name/2 will return a number if possible and an atom otherwise.

The ISO standard defines atom_chars/2 to describe the `broken-up' atom as a list of one-character atoms instead of a list of codes.

Atom (2)

atom_codes	(?Atom, ?String)
atom_chars	(?Atom, ?CharList)
char_code	(?Atom, ?Code)
number_chars	(?Number, ?CharList)
number_codes	(?Number, ?CodeList)
atom_number	(?Atom, ?Number)
name	(?Atomic, ?CodeList)
term_to_atom	(?Term, ?Atom)
atom_to_term	(+Atom, -Term, -Bindings)
atom_concat	(?Atom1, ?Atom2, ?Atom3)
atomic_concat	(+Atomic1, +Atomic2, -Atom)
atomic_list_concat(+List, -Atom)	
atomic_list_concat(+List, +Separator, -Atom)	
atom_length	(+Atom, -Length)
atom_prefix	(+Atom, +Prefix)
sub_atom	(+Atom, ?Before, ?Len, ?After, ?Sub)
sub_atom_icasechk(+Haystack, ?Start, +Needle)	

References

[1] en.wikipedia.org
[2] en.wiktionary.org
[3] U. Endriss, “Lecture Notes : Introduction to Prolog Programming”
[4] http://www.learnprolognow.org/ Learn Prolog Now!
[5] http://www.csupomona.edu/~jrfisher/www/prolog_tutorial
[6] www.cse.unsw.edu.au/~billw/cs9414/notes/prolog/intro.html
[7] www.cse.unsw.edu.au/~billw/dictionaries/prolog/negation.html