Prolog Tutorial-4

Dr A Sahu

Dept of Computer Science & Engineering

IIT Guwahati

Prolog I/O

Prolog I/O Example

```
?write(“Hello world”), nl, write(“lets program).
Hello word
Lets program
yes
```

Prolog I/O Example

```
position(‘Spielberg’, director).
position(‘Allen’, manager).
position(‘Lee’, supervisor).
find_position:-
        write(‘Whose position do you wish to know?’),
        read(Input),
        position(Input, Output),
        write(‘The position of ‘),
        write(Input), write(‘ is ‘),
        write(Output), write(‘.’)
```

Querry: ? Find_position.
Whose position do you wish to know? ‘Lee’.   <enter>

Input/Output in Prolog

- Input/output (I/O) is not a significant part of Prolog.
- Part of the reason
  - purpose of Prolog is *declarative* programming
  - I/O is intrinsically about producing *procedural side-effects*.
- Very hard to state
  - What the logical reading of a Prolog program is when it contains I/O functions.
- The I/O facilities : will present relatively simple.
- As I/O is not a core part of Prolog :
  - will not be examined upon it but
  - you may need to use it in practical exercises.

How I/O works in Prolog.

- At any time during execution of a Prolog program two files are ‘active’:
  - a current input stream, and a current output stream.
- By default these are both set to `user`
  - all input will come from the user terminal and
  - all output will be sent to the user terminal (i.e. write to the screen).
- Multiple I/O streams can be initialised but
  - Only one input and one output can be ‘active’ at any time (i.e. be read from or written to).

File-based I/O: `write/1`

- We have already used Prolog’s default output predicate **`write/1`**.
- This prints a term to the current output stream.

```
?- write(c), write(u_l), write(8),
write(r).
cu_l8r          ← writes to terminal by default.
yes
?- write([a,b,c,d]).
[a,b,c,d]
yes
```

File-based I/O: `write/1`

- It will only accept Prolog terms so strings must be enclosed within single quotation marks.

```
  ?- write(Hello World).
  syntax error
?- write(‘Hello World’).
   Hello World
   yes
```

Formatting Output

- We can use built-in predicates to format the output:
  - nl/0 = write a new line to current O/P stream.
  - tab/1 = write a specified number of white spaces to the current output stream.
    - *this prints single spaces not actual tabs!*

```
|?- write(a), tab(3), write(b), nl, tab(1),
write(c), tab(1), write(d), nl, tab(2), write(e).
a   b
 c d
  e
yes
```

Formatting Output

- We can add syntax by writing string fragments.

```
|?- Day=04, Mth=11, Year=04, write(Day),
write('/'),  write(Mth), write('/'),
write(Year).

4/11/4  Day=4, Mth=11, Year=4, yes
```

Writing ASCII characters

- Instead of writing syntax as strings we can use the corresponding ASCII codes
- An ASCII code is a number between 1 and 127 that refers to a typographical character. A-Z = 65-90
  - a-z = 97-122
- `put/1` takes a single ASCII code as an argument and writes the corresponding character to the current output stream.

```
| ?- put(65),put(44),put(66),put(46).
A,B.
yes
```

Writing ASCII characters

- This can be useful as ASCII codes can have arithmetic operations performed upon them:

```
| ?- X=32, put(65+X), put(44), put(66+X),
put(46).
a,b.
X = 32 ? yes
```

← By adding 32 to each code we can change case.

Writing lists of characters

- Instead of just writing single terms it is often useful to write out the contents of a list.
- We can define a recursive predicate `writelist/1` to do this:

```
writelist([]).
writelist([H|T]):-
   write(H),
   writelist(T).
```

```
|?- X='Bob', writelist(['The',' ',man,' was called
   ',X,'.']).
The man was called Bob.
yes
```

Writing lists of characters

- We can also define a predicate to translate lists of ASCII codes:

```
putlist([]).
putlist([H|T]):-
    put(H),
    putlist(T).
```

```
| ?- putlist([65,44,66,46]).
     A,B.
yes
```

Writing lists of characters (2)

- Either of these could be made to automatically format our output as we wished.

```
writelist2([H]):-
   write(H), put(46), !.
writelist2([H|T]):-
   write(H), tab(1), writelist2(T).
```

```
 | ?- X='Bob',
  writelist2(['The',man,was,called,X]).
 The man was called Bob.
 X = 'Bob' ? ;
 no
```

Writing lists of characters (2)

- Either of these could be made to automatically format our output as we wished.

```
writefacts([]).
       writefacts([[X,Y]|T]):-
       write(X), write('('),
        write(Y), write(')'),
        write('.'), nl,
     writefacts(T).
```

```
|  ?-  writefacts([[big,blue],[tickled,pink]]).
big(blue).
tickled(pink).
yes
```

Changing output stream

- We can redirect our output to a specific file using `tell/1.`

  `tell(Filename).` or

  `tell(‘path/from/current/dir/to/Filename’).`

- This tells Prolog to send all output to the specified file. If the file doesn’t exists it will be created. If the file already exists it will be overwritten.

- The current output stream can be identified using `telling/1.`

Changing output stream

- The current output stream can be identified using `telling/1.`
- This file will remain as the current output stream until either:
  - another output stream is opened using `tell/1`, or
  - the current output stream is closed using `told/0` and the output stream returned to `user.`
- This file remains as the current output stream as long as Prolog remains loaded or it is explicitly closed with `told/0`.

Changing output stream

```
| ?- write('Write to terminal').
Write to terminal
yes
| ?- telling(X).
X = user ?
yes
| ?- tell('demo/test').
yes                            ← file is created or overwritten
| ?- telling(X).
X = 'demo/test' ?
yes
| ?- write('Now where does it go?').
yes                            ← Text doesn’t appear in file until…
| ?- told.
yes                            ← it is closed.
| ?- write('Oh, here it is!').
Oh, here it is!
yes
```

Reading input: `read/1`

- Now that we know how to control our output we need to do the same for our input.
- The default input stream is the user terminal.
- We can read terms from the terminal using the command **`read/1`**.
  - this displays a prompt ‘|:’ and waits for the user to input a term followed by a full-stop.

```
| ?- write('What is your name?'), nl, read(X),
write('Greetings '), write(X).
What is your name?
‘tim’.
Greetings tim
X = tim ?
yes
```

Reading input: `read/1` (2)

- `read/1` can only recognise *Prolog terms* finished with a *full-stop*.

```
|?- read(X).
  ‘hello’
                ← Waits for full-stop to finish term.
     .          ← Finds full-stop and succeeds.
X = hello?
yes
```

Reading input: read/1 (2)

- Therefore, strings of text must be enclosed in single quotes.

```
|?- read(X).         |?- read(X).
 ‘Hi there!’.        |: ‘Hi there!’.
syntax error         X = ‘Hi there!’?
                     yes
```

- Variables are translated into Prolog’s internal representation.

```
|?- read(X).
 blue(Moon).
X = blue(_A)?
yes
```

Different Quotes

- When we are reading strings there are two ways we can input them:
  - Enclose them in single quotes : string read verbatim.

```
| ?- read(X).
 Hi bob!'.
X = 'Hi bob!' ?
yes
```

  - Enclose them in double quotes : string is interpreted into the corresponding list of ASCII codes.

```
| ?- read(X).
"Hi bob!".
X = [72,105,32,98,111,98,33] ?  yes
```

- It is important to use the right quotes as otherwise you won’t be able to process the input correctly.

name/2

- This is not the only way to convert terms into strings of ASCII codes, the built-in predicate name/2 also does this.
- We can translate any Prolog term (except a variable) into a list of corresponding ASCII codes using name/2.

```
|?- name(aAbB,L).
L = [97,65,98,66] ?
yes
|?- X=`Make me ASCII', name(X,L).
L = [77,97,107,101,32,109,101,32,65,83,67,73,73],
yes
```

name/2

- Or convert lists of ASCII codes into Prolog terms.

```
|?- name(C, [72,101,108,108,111,32,87,111,114,108,100]).
C = ‘Hello World’,
yes
```

- These lists are useful as we can use them to segment a sentence and create the input for other purpose

get-ting characters from input

- As well as reading whole terms from the input we can also read individual characters.

- **get0/1** (= get-zero) reads a character from the current input stream and returns the character’s ASCII code.

```
| ?- get0(X).          | ?- get0(X).
`A’.                   `    h’.
X = 65?             X = 32? Top-level options:…space
yes
```

get-ting characters from input

- get/1 has virtually the same function except that it will skip over any spaces to find the next printable character.

```
get(X).        get(X).
`A’             `   h’
X = 65?        X = 104 ?
yes         yes
```

- As both are just reading characters, not terms, they don’t need **to be terminated with a full-stop.**

see-ing an Input file

- `get/1` and `get0/1` are mostly used for processing text files.
  - `read/1` can only read terms so it would be unable to read a file of flowing text.
  - `get/1` and `get0/1` will read each character and return its ASCII code irrespective of its Prolog object status.
- To change our input from a user prompt to a file we use **`see/1`**

  `see(Filename).` or

  `see(‘path/from/curr/dir/to/Filename’).`

`see`-ing an Input file

- We can identify the current input stream using **`seeing/1`**.
- This file will remain as the current input stream until either:
  - another input stream is opened using `see/1`, or
  - the current input stream is closed using **`seen/0`** returning it to `user.`

read-ing input files

- Once the input file is activated using see/1 we can process its content.
- If the input file contains Prolog terms then we can read them one at a time

  Input file ‘colours’ contains:

  ```
  big(blue).
  tickled(pink).
  red_mist.
  ```

  ```
  |?- see(‘demo/colours’),read(X),read(Y),read(Z).
  X = big(blue),
  Y = tickled(pink),
  Z = red_mist ?
  yes
  ```

- The file is processed in order and the interpreter remembers where we were so every new call to read/1 reads the next term.
- This continues until **end_of_file** is reached or input is **seen/0**.

Multiple I/O streams

- Managing multiple I/O streams is difficult using file-based I/O predicates.
- write/1 and read/1 work on the current output and input files respectively.
- You can not specify which file to read from or write to.

Multiple I/O streams

- Output is not written to a file until it is closed (told/0)
  - but told only closes the current output stream. Therefore, each output file must be re-activated (tell/1) before it can be closed.
  - This is a rather verbose way to do it.
- If we want to use multiple input and output files we need to use *stream-based I/O* instead.
- A stream is a interpreter generated pointer for a specific file. It allows us to dynamically access the file and move about within it.

Stream I/O predicates

- Vast number of complex stream handling predicates
- `open/3` opens a file for reading or writing:
  - the file specification (the name of the file);
  - the mode in which the file is to be opened (read/write/append);
  - the stream name (generated by the interpreter). This takes the form '$stream'(2146079208).

  e.g. open('demo/test',append,Stream).
- Stream is initialised when the file is opened, and thereafter the file is referred to using the stream pointer (whatever 'Stream' unified with), *not* using its name.

Stream I/O predicates (2)

- `current_input/1` succeeds if its argument is the current input stream.
- `current_output/1` succeeds if its argument is the current output stream.
- `set_input/1` sets the current input stream to be the stream given as its argument (equivalent of see/1).
- `set_output/1` sets the current output stream to be the stream given as its argument (equivalent of tell/1).
- Once a stream is set as the current input/output then it can be written to using write/1 and read from using read/1.

Stream I/O predicates (3)

- However, using streams you don’t need to set a current I/O as you can refer directly to the streams using their stream pointer.

- `read/2` reads a term from a stream. Its arguments are:
  - the stream to read from;
  - the term to read (or the variable to put the term into).

  e.g. |?- open(file1,read,File1), read(File1,X).

Stream I/O predicates (3)

- `write/2` writes a term to a stream. Its arguments are:
  - the stream to write to;
  - the term to write to the stream.

  e.g. |?- open(file2,write,File2), write(File2,X).

- There are also two argument versions of other file-based I/O predicates that allow you to specify the target stream (e.g. nl/1, tab/2, get/2, get0/2, put/2).

Closing a stream

- As with file-based I/O the output file is not modified until it is closed but now we can refer to it directly using the stream pointer and the command `close/1.`

```
| ?- open('demo/test1',write,Test),
write(Test,'Hello'),          close(Test).
Test = '$stream'(2146079648) ?
yes
```

Built-in I/O Predicates

write/[1,2]	write a term to the current output stream.
nl/[0,1]	write a new line to the current output stream.
tab/[1,2]	write a specified number of white spaces to the current output stream.
put/[1,2]	write a specified ASCII character.
read/[1,2]	read a term from the current input stream.
get/[1,2]	read a **printable** ASCII character from the input stream (i.e. skip over blank spaces).
get0/[1,2]	read an ASCII character from the input stream
see/1	make a specified file the current **input** stream.
seeing/1	determine the current **input** stream.
seen/0	close the current **input** stream and reset it to `user`.
tell/1	make a specified file the current **output** stream.
telling/1	determine the current **output** stream.
told/0	close the current **output** stream and reset it to `user`.
name/2	arg 1 (an atom) is made of the ASCII characters listed in arg 2

Thanks