Input/Output

Artificial Intelligence Programming in Prolog
Lecturer: Tim Smith
Lecture 12
04/11/04
Input/Output in Prolog

- Input/output (I/O) is not a significant part of Prolog.
- Part of the reason for this is that the purpose of Prolog is *declarative* programming, and input/output is intrinsically about producing *procedural side-effects*.
- It is very hard to state what the logical reading of a Prolog program is when it contains I/O functions.
- The I/O facilities I will present here are relatively simple. Each implementation of Prolog has more advanced I/O facilities but these will not be covered in this course.
- As I/O is not a core part of Prolog you 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 which means that
  - all input will come from the user terminal (the shell window in which Sicstus is loaded) 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

- 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 the current output 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
```

- 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 (see http://www.asciitable.com/).

• 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
```

• 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. ← By adding 32 to each code we can change case.
X = 32 ? yes
```
| Dec | Hx | Oct | Char  | HTML | Chr | Dec | Hx | Oct | HTML | Chr | Dec | Hx | Oct | HTML | Chr |
|-----|----|-----|-------|------|-----|-----|----|-----|------|-----|-----|----|-----|------|-----|-----|
| 0   | 00 | 000 | NUL   | `#32` | Space| 32  | 20 | 040 | `#64` | 8   | 96  | 60 | 140 | `#96` | `\` |
| 1   | 00 | 001 | SOH   | `#33` | !    | 33  | 21 | 041 | `#65` | A   | 97  | 61 | 141 | `#97` | a   |
| 2   | 00 | 002 | STX   | `#34` | "    | 34  | 22 | 042 | `#66` | B   | 98  | 62 | 142 | `#98` | b   |
| 3   | 00 | 003 | ETX   | `#35` | #    | 35  | 23 | 043 | `#67` | C   | 99  | 63 | 143 | `#99` | c   |
| 4   | 00 | 004 | EOT   | `#36` | $    | 36  | 24 | 044 | `#68` | D   | 100 | 64 | 144 | `#100` | d   |
| 5   | 00 | 005 | ENQ   | `#37` | %    | 37  | 25 | 045 | `#69` | E   | 101 | 65 | 145 | `#101` | e   |
| 6   | 00 | 006 | ACK   | `#38` | &    | 38  | 26 | 046 | `#70` | F   | 102 | 66 | 146 | `#102` | f   |
| 7   | 00 | 007 | BEL   | `#39` | '    | 39  | 27 | 047 | `#71` | G   | 103 | 67 | 147 | `#103` | g   |
| 8   | 01 | 010 | BS    | `#40` | (    | 40  | 28 | 050 | `#72` | H   | 104 | 68 | 150 | `#104` | h   |
| 9   | 01 | 011 | TAB   | `#41` | )    | 41  | 29 | 051 | `#73` | I   | 105 | 69 | 151 | `#105` | i   |
| 10  | A  | 012 | LF    | `#42` | *    | 42  | 2A | 052 | `#74` | J   | 106 | 70 | 152 | `#106` | j   |
| 11  | B  | 013 | VT    | `#43` | +    | 43  | 2B | 053 | `#75` | K   | 107 | 71 | 153 | `#107` | k   |
| 12  | C  | 014 | FF    | `#44` | ,    | 44  | 2C | 054 | `#76` | L   | 108 | 72 | 154 | `#108` | l   |
| 13  | D  | 015 | CR    | `#45` | -    | 45  | 2D | 055 | `#77` | M   | 109 | 73 | 155 | `#109` | m   |
| 14  | E  | 016 | SO    | `#46` | .    | 46  | 2E | 056 | `#78` | N   | 110 | 74 | 156 | `#110` | n   |
| 15  | F  | 017 | SI    | `#47` | /    | 47  | 2F | 057 | `#79` | O   | 111 | 75 | 157 | `#111` | o   |
| 16  | 10 | 020 | DLE   | `#48` | 0    | 48  | 30 | 060 | `#80` | P   | 112 | 76 | 160 | `#112` | p   |
| 17  | 11 | 021 | DC1   | `#49` | 1    | 49  | 31 | 061 | `#81` | Q   | 113 | 77 | 161 | `#113` | q   |
| 18  | 12 | 022 | DC2   | `#50` | 2    | 50  | 32 | 062 | `#82` | R   | 114 | 78 | 162 | `#114` | r   |
| 19  | 13 | 023 | DC3   | `#51` | 3    | 51  | 33 | 063 | `#83` | S   | 115 | 79 | 163 | `#115` | s   |
| 20  | 14 | 024 | DC4   | `#52` | 4    | 52  | 34 | 064 | `#84` | T   | 116 | 80 | 164 | `#116` | t   |
| 21  | 15 | 025 | NAK   | `#53` | 5    | 53  | 35 | 065 | `#85` | U   | 117 | 81 | 165 | `#117` | u   |
| 22  | 16 | 026 | SYN   | `#54` | 6    | 54  | 36 | 066 | `#86` | V   | 118 | 82 | 166 | `#118` | v   |
| 23  | 17 | 027 | ETB   | `#55` | 7    | 55  | 37 | 067 | `#87` | W   | 119 | 83 | 167 | `#119` | w   |
| 24  | 18 | 030 | CAN   | `#56` | 8    | 56  | 38 | 070 | `#88` | X   | 120 | 84 | 170 | `#120` | x   |
| 25  | 19 | 031 | EM    | `#57` | 9    | 57  | 39 | 071 | `#89` | Y   | 121 | 85 | 171 | `#121` | y   |
| 26  | 1A | 032 | SUB   | `#58` | A    | 58  | 3A | 072 | `#90` | Z   | 122 | 86 | 172 | `#122` | z   |
| 27  | 1B | 033 | ESC   | `#59` | B    | 59  | 3B | 073 | `#91` |  | 123 | 87 | 173 | `#123` | `\` |
| 28  | 1C | 034 | FS    | `#60` | C    | 60  | 3C | 074 | `#92` |  | 124 | 88 | 174 | `#124` | `\` |
| 29  | 1D | 035 | GS    | `#61` | D    | 61  | 3D | 075 | `#93` |  | 125 | 89 | 175 | `#125` | `\` |
| 30  | 1E | 036 | RS    | `#62` | E    | 62  | 3E | 076 | `#94` |  | 126 | 90 | 176 | `#126` | `\` |
| 31  | 1F | 037 | US    | `#63` | F    | 63  | 3F | 077 | `#95` |  | 127 | 91 | 177 | `#127` | `DEL` |

Source: www.asciitable.com
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:

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

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

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

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

  ```prolog
  |?- 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.

```prolog
writelist2([H]):- writefacts([]).
    write(H), put(46), !.
writefacts([X,Y]|T):-
    writelist2([H|T]):-
        write(X), write('('),
        write(H), tab(1),
        write(Y), write(')'),
        write('.',), nl,
        writefacts(T).
| ?- X='Bob', writelist2(['The',man,was,called,X]).
The man was called Bob.
X = 'Bob' ? ;
no
| ?- 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`.
  
  ```prolog
  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 exist, it will be created. If the file already exists, it will be overwritten.

- 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 Sicstus remains loaded or it is explicitly closed with `told/0`. 
| ?- 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
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: \texttt{read/1 (2)}

- \texttt{read/1} can only recognise \texttt{Prolog terms} finished with a \texttt{full-stop}.
  \begin{verbatim}
  \texttt{|?- read(X).}
  \texttt{|: hello}
  \end{verbatim}
  \hspace{1cm} \leftarrow \text{Waits for full-stop to finish term.}
  \begin{verbatim}
  \texttt{|.}
  \end{verbatim}
  \hspace{1cm} \leftarrow \text{Finds full-stop and succeeds.}

  \begin{verbatim}
  X = hello?
  yes
  \end{verbatim}

- Therefore, strings of text must be enclosed in single quotes.
  \begin{verbatim}
  \texttt{|?- read(X).}
  \texttt{|: Hi there!.}
  \end{verbatim}
  \hspace{1cm} \texttt{syntax error}
  \begin{verbatim}
  \texttt{|: ‘Hi there!’}.\texttt{X = ‘Hi there!’?}
  \end{verbatim}
  \hspace{1cm} \texttt{yes}

- Variables are translated into Prolog’s internal representation.
  \begin{verbatim}
  \texttt{|?- read(X).}
  \texttt{|: blue(Moon).}
  \end{verbatim}
  \hspace{1cm} \texttt{X = blue(_A)?}
  \begin{verbatim}
  yes
  \end{verbatim}
Different Quotes

- When we are reading strings there are two ways we can input them:
  - if we enclose them in single quotes (e.g. ‘Hi Bob!’) the string is read verbatim.
    
    ```prolog
    |- ?- read(X).
    |: 'Hi bob!'.
    X = 'Hi bob!' ?
    yes
    ```
  
  - if we enclose them in double quotes (e.g. “Hi Bob!”) the string is interpreted into the corresponding list of ASCII codes.
    
    ```prolog
    |- ?- 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`.

```prolog
|?- 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
```

- Or convert lists of ASCII codes into Prolog terms.

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

- These lists are useful as we can use them to segment a sentence and create the input for our DCG parser (next lecture).
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.
  
  ```prolog
  ?- get0(X).
  X = 65 ?
  yes
  ```

  ```prolog
  ?- get0(X).
  X = 32 ?
  yes
  ```

  White spaces

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

  ```prolog
  get(X).
  X = 65 ?
  yes
  ```

  ```prolog
  get(X).
  X = 104 ?
  yes
  ```

  ASCII code for a space

- 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/current/dir/to/Filename’).

- 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.
- 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

• There are a vast number of complex stream handling predicates (see Sicstus manual). Here are just the basics:

• **open/3** opens a file for reading or writing. Its arguments are:
  – 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).

• The 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).

- **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`.

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

- There are many more stream I/O predicates built-in to Sicstus Prolog but they vary across Prolog implementations so you are not required to know them.
  - Look at the Sicstus manual for more information.
- File-based I/O is the traditional method of performing I/O in Prolog and it is more universally known even though it has serious limitations.
## 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 \texttt{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 \texttt{user}.
- **name/2**: arg 1 (an atom) is made of the ASCII characters listed in arg 2.