Input & Output

York University CSE 3401
Vida Movahedi
Overview

• Read and write terms
• Read and write characters
  – Reading English sentences
• Working with files
• Declaring operators

[ref.: Clocksin- Chap. 5 ]
READ

• read(X)
  – Will read the next term you type
  – The term must be followed by a dot, and a space or newline (enter)
  – The read term will be unified with X
    • If X is not instantiated before, it will be instantiated with the term, and success (e=[X/term])
    • If instantiated before,
      – If X can be matched with term, success.
      – If not, fail.
  – ‘read’ can not be re-satisfied (only once, will fail on backtracking!)
• Examples:
  :- read(X).
  12. entered by user, on keyboard
  X = 12.

  :- X=5, read(X).
  12. false.

  :- read(Y).
  [it, is, a, beautiful, day].
  Y = [it, is, a, beautiful, day].

  :- read(Z).
  1+2
  Z = 1+2.
WRITE

• write(X)
  – If X is instantiated to a term before, the term will be displayed
  – If not instantiated before, a uniquely numbered variable will be displayed
  – ‘write’ can not be re-satisfied (only once!)

• nl
  – Means “new line”
  – Writes a “new line”, all succeeding output appear on the next line of display
WRITE (cont.)

• Examples
  
  :- write ([‘Hello’, world]).
  [Hello, world]
  true.

  :- X is 4+4, write(X).
  8
  X=8.

  :- write(X).
  _G248.
  true.
Vine diagram (pretty print)

• Indentation for nested lists
  pp([1, [2,3], [4, [5]],6], 0)

  spaces(0) :- !.
  spaces(N) :- write(' '), N1 is N -1, spaces(N1).

  pp([H|T], I) :- !, J is I+3, pp(H, J), ppx(T, J), nl.
  pp(X, I) :- spaces(I), write(X), nl.

  ppx([], _).
  ppx([H|T], I) :- pp(H, I), ppx(T, I).
Printing lists

:- write(['Good’, morning, ‘!’]).
[Good, morning, !]

• Write a list w/o the commas and []
  :- phh(['Good’, morning, ‘!’]).
  Good morning!

  phh([]):- nl.
  phh([H|T]) :- write(H), spaces(1), phh(T).
Read/Write characters

• **get_char(X)**
  – Similar to ‘read’, but reads only one character
  – Press ‘Enter’ after input, so it will be available to Prolog

• **put_char(X)**
  – Similar to ‘write’, but writes only one character

• **Example:**
  ```prolog
  :- get_char(X), put_char(X).
  M          entered by user
  M
  X = ‘M’.
  ```
• Read in characters, write them out again, until a ‘.’ is read:
  
go :- do_a_char, go.
do_a_char :- get_char(X), put_char(X), X=‘’, !, fail.
do_a_char .

:- go.
I am feeling great.
I am feeling great.
• Same as previous example, but don’t write out ‘.’:

```prolog
  go :- do_a_char, go.
  do_a_char :- get_char(X), X= '.', !, fail.
  do_a_char :- put_char(X).

:- go.
I am feeling great.

Error! put_char argument not instantiated!
```
How about this code?

go :- do_a_char, go.
do_a_char :- get_char(X), X= '.', !, fail.
do_a_char :- get_char(X), put_char(X).

:- go.
I am feeling great.
mfeigget

Once a character has been read from the terminal, if not saved, it will be gone forever, can never get hold of it again!
• Get hold of the character:
  
  go :- get_char(X), get_more(X).
  get_more(‘.’) :- !, fail.
  get_more(X) :- put_char(X), get_char(Next), get_more(Next).

  :- go.
  I am feeling great.
  I am feeling great
Another Example

- Read in characters, write them out again, until a ‘.’ is read. Convert ‘a’s to ‘A’s.
  
  go :- get_char(X), get_more(X).
  
  get_more(‘.’) :- !, put_char(‘!’), fail.
  
  get_more(a) :- !, put_char(‘A’),
               get_char(Next), get_more(Next).
  
  get_more(X) :- put_char(X), get_char(Next), get_more(Next).

  :- go.

  I am feeling great.

  I Am feeling greAt!
Read/Write Files

- **Input streams**
  - Keyboard
    - Prolog name: ‘user_input’,
    - It is the default input stream
  - A file (opened for reading)

- **Output streams**
  - Display
    - Prolog name: ‘user_output’
    - It is the default output stream
  - A file (opened for writing)

- The same predicates can be used for file streams:
  - read, write, get_char, put_char, nl
Open & Close I/O Streams

• Open a stream
  `open(Filename, Mode, Stream)`
  – Filename: name of the file
  – Mode: one of read, write, append, update
  – Stream: the stream that has been opened

Examples:
  `open('myfile.txt', read, X)`
  `open('output.txt', write, X)`

• Close a stream
  `close(X)`
Current Streams

- Determine what is the current input/output
  
  current_input(Stream)
  current_output(Stream)
  
  - Instantiate their argument to the name of the current input/output stream

- Changing the current input/output
  
  set_input(Stream)
  set_output(Stream)
  
  - Set the current stream to the named stream specified by the argument
  - The argument can be `user_input / user_output`
Templates

program :-
    open(‘input.txt’, read, X),
    current_input(S),
    set_input(X),
    code_reading,
    close(X),
    set_input(S).

program :-
    open(‘output.txt’, write, X),
    current_output(S),
    set_output(X),
    code_writing,
    close(X),
    close(X),
    set_output(S).
program :-
    see('input.txt'),
    code_reading,
    seen.

program :-
    tell('output.txt'),
    code_writing,
    told.

• Question: Does ‘seen’ set the input stream to the previous current stream?
  Try :-help(seen). to find answer.
Example

• Write copyfile(Source File, Destination File) which copies a Source File to Destination File one character at a time:

```prolog
copyfile(Source File, Destination File) :-
  open(Source File, read, X), open(Destination File, write, Y),
  current_input(SI), current_output(SO),
  set_input(X), set_output(Y),
  read_write_code,
  close(X), close(Y),
  set_input(SI), set_output(SO).

read_write_code :- get_char(X), get_more(X).
get_more(end_of_file):- !.
get_more(X):- put_char(X), get_char(X2), get_more(X2).
```

York University - CSE 3401 - V. Movahedi 08_IO
Read program files

- Reading program from a file
  
  ```prolog
  :- consult('mycode.pl').
  or
  :- ['mycode.pl'].
  ```

- Consulting several files:
  
  ```prolog
  :- consult(file1), consult('file2.pl'), consult('c:\pl\file3.txt').
  or
  :- [file1, 'file2.pl', 'c:\pl\file3.txt'].
  ```
More on reading terms

• Examples:

:- read(X).
3 + 4.
X = 3+4.

:- read(X).
3 + .
Error! Unbalanced operator.

How does Prolog know?
Terms (reminder)

- Term
  - Constants
  - Variables
  - Functors applied to arguments
  - Operators and their arguments

- Examples:
  :- read(X).
  We can type in:
  8. a. myatom. ‘GOOD’.
  Myvariable. X.
  +(3,4). 3+4.
Operators (reminder)

- Operators
  - To make some functors easier to use, e.g. instead of +(3,4) we can write 3+4 (Important: it is not the same as 7)

- Position
  - prefix, infix, or postfix, e.g. +(3,4), 2*5, 7!

- Precedence
  - An integer associated with each operator, the closer to 1, the higher the precedence
  - e.g. multiplication has a higher precedence than addition, a-b/c is -(a/(b,c))

- Associativity
  - Left or right
  - All arithmetic operators left associative
  - e.g. 8/4/4 is (8/4)/4
Declaring operators

• An operator is declared by a goal:
  \[ \text{:- op( Precedence, Specifier, Name).} \]

For example:
\[ \text{:- op(1000, xf, myop).} \]
\[ \text{:- op(500, yfx, ‘+’).} \]
\[ \text{:- op(400, yfx, ‘*’).} \]
\[ \text{:- op(900, fy, ‘\+’).} \]

• Precedence:
  an integer between 1 and 1200, lower values, higher priority

• Name:
  the operator’s name

• Specifier:
  specifies position and associativity
  valid specifiers: fx, fy, xfx, xfy, yfx, yfy, xf, yf
Operator specifiers

• Operator position:
  – Prefix: fx, fy
  – Postfix: xf, yf
  – Infix: xfx, xfy, yfx, yfy

• Operator associativity
  – x
    on this position a term with precedence class strictly lower to the precedence of the operator should occur
  – y
    on this position a term with precedence class lower or equal to the precedence of the operator should occur
Example (1)

- Operator + is defined as yfx
  \[ a + b + c \]
  \[ (a + b) + c \text{ or } a + (b + c) \]

  Argument containing an operator with the same precedence

  \( yfx \Rightarrow \) the argument on the right can not have the same precedence!

Therefore \( a + b + c \) is interpreted as \( (a + b) + c \)
(left associative)
Example (2)

- What is the specifier for ‘not’ if we want to allow: not not a

  Prefix → fx or fy
  We want ‘not not a’ to be interpreted as ‘not (not a)’

  Argument containing an operator with the same precedence

  Therefore the specifier is fy
write_canonical

• write_canonical ignores operator declarations:

    :- write(a + b + c).
    a+b+c

    :- write_canonical(a + b + c).
    +(+(a, b), c)