Part 1 The Prolog Language

Chapter 6
Input and Output

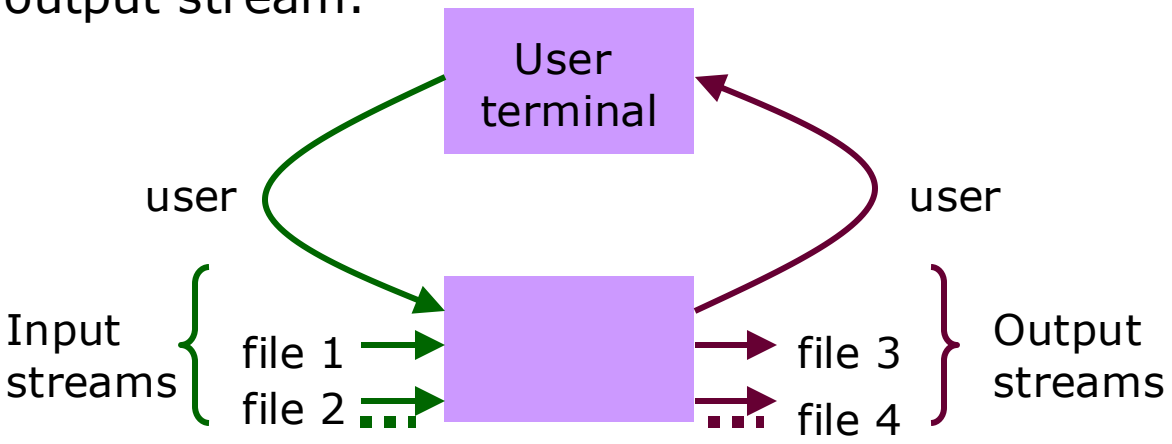
6.1 Communication with files

- The method of communication between the user and the program:
  - User questions to the program
  - Program answers in terms of instantiations of variables

- Extensions to this basic communication method are needed in the following areas:
  - input of data in forms other than questions—for example, in the form of English sentences,
  - output of information in any format desired, and
  - input from and output to any computer file or device and not just the user terminal.

6.1 Communication with files

- We first consider the question of directing input and output to files.
  - In Figure 6.1, the program can read data from several input files (input streams), and output data to several output files (output streams).
  - Data coming from the user's terminal is treated as just another input stream.
  - Data output to the terminal is treated as another output stream.



6.1 Communication with files

- At any time during the execution of a Prolog program, only two files are 'active':
  - one for input (current input stream) and
  - one for output (current output stream).
- The current input stream can be changed to another file, **Filename**, by the goal:

  see(Filename)
- The current output stream can be changed by a goal of the form:

  tell(Filename)
- The goal **seen** closes the current input file.
  - **seen** closes the current input, and resets it to user_input.
- The goal **told** closes the current output file.
  - **told** closes the current output, and resets it to user_output.

6.1 Communication with files

- For examples:

```
…
see( file1),
read_from_file( Information),
see( user),
…
tell( file3),
write_on_file( Information),
tell( user),
…
```

- Another example:

```
readfile( X, Y) :- see('test.txt'), get( X), get( Y), seen.
writefile(X) :- tell('test1.txt'), put( X), told.

| ?- readfile(X, Y).
X = 48
Y = 49
yes
```

test.txt
0123456 …

6.2 Processing files of terms
6.2.1 Read and write

- **read(X):**
  - The built-in predicate read is used for reading terms from the current input stream.
  - The goal **read(X)** will cause the next term, T, to be read, and this term will be matched with X.
  - If X is a variable then, as a result, X will become instantiated to T.
  - If matching does not succeed then the goal **read(X)** fails.
  - The predicate read is deterministic, so in the case of failure there will be no backtracking to input another term.
  - If **read(X)** is executed when the end of the current input file has been reached then X will become instantiated to the atom **end_of_file.**

6.2 Processing files of terms
6.2.1 Read and write

- **write(X):**
  - The built-in predicate **write** outputs a term.
  - The goal **write(X)** will output the term X on the current output file.
  - X will be output in the same standard syntactic form in which Prolog normally displays values of variables.
  - A useful feature of Prolog is that the **write** procedure ‘knows’ to display any term no matter how complicated it may be.

6.2 Processing files of terms
6.2.1 Read and write

- **tab(N):**
  - The built-in predicates for formatting the output.
  - They insert spaces and new lines into the output stream.
  - The goal **tab(N)** causes N space to be output.
  - The predicate **nl** (which has no arguments) causes the start of a new line at output.

6.2 Processing files of terms
6.2.1 Read and write

- An example:

```
cube( N, C) :- C is N * N * N.

| ?- cube(2, X).
X = 8
yes

| ?- cube(5, Y).
Y = 125
yes

| ?- cube(12, Z).
Z = 1728
yes
```

```
cube :-
    read( X), process( X).
process( stop) :- !.
process( N) :-
    C is N * N * N,
    write( C), cube.

| ?- cube.
2.
8
5.
125
12.
1728
stop.
(16 ms) yes
```

6.2 Processing files of terms
6.2.1 Read and write

- An incorrect simplified:

```
cube :- read( stop), !.
cube :- read( N), C is N * N * N, write( C), cube.
```

  - The reason why this is wrong can be seen easily if we trace the program with input data 5.
  - The goal **read(stop)** will fail when the number is read, and this number will be lost forever.
  - The next **read** goal will input the next term.
  - On the other hand, it could happen that the **stop** signal is read by the goal **read(N)**, which would then cause a request to multiply non-numeric data.

6.2 Processing files of terms
6.2.1 Read and write

- Another version:

```
cube :- write( 'Next item, please: '),
        read( X), process( X).
process( stop) :- !.
process( N) :-
        C is N * N * N,
        write( 'Cube of '), write( N),
        write(' is '), write( C), nl, cube.
```

```
| ?- cube.
Next item, please: 5.
Cube of 5 is 125
Next item, please: 12.
Cube of 12 is 1728
Next item, please: stop.
(31 ms) yes
```

6.2 Processing files of terms
6.2.2 Displaying lists

- **writelist(L):**
  - The procedure outputs a list L so the each elements of L is written on a separate line:

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

| ?- writelist( [a, b, c]).
a
b
c
yes
```

6.2 Processing files of terms
6.2.2 Displaying lists

- **writelist2(L):**
  - If we have a list of lists, we can define the procedure:

```
writelist2([]).
writelist2([L|LL]) :- doline( L), nl, writelist2( LL).
doline([]).
doline([X|L]) :- write( X), tab(1), doline( L).
```

```
| ?- writelist2([[a,b,c],[d,e,f],[g,h,i]]).
a b c
d e f
g h i
(15 ms) yes
```

6.2 Processing files of terms
6.2.2 Displaying lists

- **bars(L):**
  - A list of integer numbers can be sometimes conveniently shown as a bar graph.
  - The procedure, bars, can be defined as:

```
bars( []).
bars([N|L]) :- stars( N), nl, bars( L).
stars( N) :- N > 0, write( *), N1 is N-1, stars(N1).
stars( N) :- N =<0.

| ?- bars([3,4,6,5]).
***
****
******
*****
true ?
yes
```

6.2 Processing files of terms
6.2.3 Processing a file of terms

- **processfile:**
  - A typical sequence of goals to process a whole file, F, would look something like this:

    …, see(F), processfile, see(user), …

  - Here **processfile** is a procedure to read and process each term in F, one after another, until the end of the file is encountered.

```
processfile :- read( Term), process( Term).
process( end_of_file) :- !.
process( Term) :- treat( Term), processfile.
```

  - Here treat(Term) represents whatever is to be done with each term.

6.2 Processing files of terms
6.2.3 Processing a file of terms

- An example:
  - This procedure **showfile** can display on the terminal each term together with its consecutive number.

```
showfile( N) :- read( Term), show( Term, N).
show( end_of_file, _) :- !.
show( Term, N) :- write(N), tab( 2), write( Term), nl,
                  N1 is N+ 1, showfile( N1).
```

```
| ?- showfile(3).
a.
3  a
aa.
4  aa
abcdefg.
5  abcdefg
test.
6  test
```

Exercise

- Let **testfile** be a file. Write a procedure
  printfile(‘testfile’)
  That displays on the terminal all the context in **testfile.**

Exercise

- Let **f** be a file of terms. Write a procedure

  findallterms(Term)

  That displays on the terminal all the terms in **f** that match **Term**.

  Make sure that Term is not instantiated in the process(which could prevent its match with terms that occur latter in the file).

6.3 Manipulating characters

- A character is written on the current output stream with the goal
  put(C)
  where C is the ASCII code (0-127) of the character to be output.
- For example:
  ?- put(65), put(66), put(67).
  would cause the following output
  ABC
  - 65 is the ASCII code of 'A', 66 of 'B', 67 of 'C'.

- A single character can be read from the current input stream by the goal
  get0(C)
  - **get0(C)** causes the current character to be read from the input stream.
  - The variable **C** becomes instantiated to the ASCII code of this character.

  get(C)
  - **get(C)** is used to read non-blank characters.
  - **get(C)** causes the skipping over of all non-printable characters.

6.3 Manipulating characters

- Define procedure **squeeze**:
  - **squeeze** can read a sentence from the current input stream, and output the same sentence reformatted so that multiple blanks between works are replaced by single blanks.
  - For example:
    - An acceptable input is then:

      The robot tried to pour wine out of the bottle.
    - The goal squeeze would output:

      The robot tried to pour wine out of the bottle.

```
squeeze :- get0( C), put( C), dorest( C).
dorest( 46) :- !.
dorest( 32) :- !, get( C), put( C), dorest( C).
dorest( Letter) :- squeeze.
```

6.3 Manipulating characters

```
squeeze :- get0( C), put( C), dorest( C).
dorest( 46) :- !.
dorest( 32) :- !, get( C), put( C), dorest( C).
dorest( Letter) :- squeeze.
```

```
| ?- squeeze.
this is   a    test.
this is a test.
(15 ms) yes

| ?- squeeze.
a full stop  ,  a   blank  or  a  letter.
a full stop , a blank or a letter.
yes
```

Here still has a blank, can you improve the program to remove this?

Ps. The ASCII code of ',' is 44.

6.4 Constructing and decomposing atoms

- **name(A, L)**:
  - **name** is a built-in predicate.
  - **name** is true if L is the list of ASCII codes of the characters in A.
  - For example:

    | ?- name(zx232, [122, 120, 50, 51, 50]).
    yes

    | ?- name(ZX232, [122, 120, 50, 51, 50]).
    ZX232 = zx232
    yes
  - There are two typical uses of name:

    (1) given an atom, break it down into single characters.

    (2) given a list of characters, combine them into an atom.

6.4 Constructing and decomposing atoms

- The example of first kind of application:

  taxi(X)

  - **taxi(X)** tests whether an atom X represents a taxi.

  ```
  taxi( X) :- name( X, Xlist),
              name( taxi, Tlist),
              conc( Tlist, _, Xlist).

  | ?- taxi( taxia1).
  yes

  | ?- taxi( taxilux).
  yes

  | ?- taxi( taxtax).
  no
  ```

6.4 Constructing and decomposing atoms

- Another example of second kind of application:

  getsentence(Wordlist)

  - **getsentence** reads a free-form natural language sentence and instantiates **Wordlist** to some internal representation of the sentence.
  - For example:
    - If the current input stream is:

      Mary was pleased to see the robot fail.
    - The goal **getsentence(Wordlist)** will cause the instantiation:

      Sentence = [‘Mary’, was, pleased, to, see, the, robot, fail]

6.4 Constructing and decomposing atoms

```
% Figure 6.2  A procedure to transform a sentence into a list
    of atoms.

getsentence( Wordlist)  :-
          get0( Char), getrest( Char, Wordlist).

getrest( 46, [] )  :-  !.
getrest( 32, Wordlist)  :-  !, getsentence( Wordlist).
getrest( Letter, [Word | Wordlist] )  :-
          getletters( Letter, Letters, Nextchar),
          name( Word, Letters),
          getrest( Nextchar, Wordlist).

getletters( 46, [], 46)  :-  !.
getletters( 32, [], 32)  :-  !.
getletters( Let, [Let | Letters], Nextchar)  :-
          get0( Char), getletters( Char, Letters, Nextchar).
```

6.4 Constructing and decomposing atoms

- The procedure **getsentence** first reads the current input character, **Char**, and then supplies this character to the procedure **getrest** to complete the job.
- **getrest** has to react properly according to three cases:
  (1) **Char** is the full stop: then everything has been read.
  (2) **Char** is the blank: ignore it, **getsentence** form rest of input.
  (3) **Char** is a letter: first read the word, **Word**, which begins with **Char**, and the use **getsentence** to read the rest of the sentence, producing **Wordlist**. The cumulative result is the list **[Word|Wordlist].**

6.4 Constructing and decomposing atoms

- The procedure that reads the characters of one word is:

  getletters(Letter, Letters, Nextchar)

  The three arguments are:

  (1) **Letter** is the current letter (already read) of the word being read.

  (2) **Letters** is the list of letters (starting with **Letter**) of up to the end of the word.

  (3) **Nextchar** is the input character that immediately follows the word read. **Nextchar** must be a non-letter character.

6.4 Constructing and decomposing atoms

```
| ?- getsentence(X).
Mary was pleased to see the robot fail.
X = ['Mary',was,pleased,to,see,the,robot,fail]
(15 ms) yes

| ?- getsentence([X]).
test.
X = test
yes

| ?- getsentence(X).
test.
X = [test]
yes

| ?- getsentence(X).
test and test.
X = [test,and,test]
yes

| ?- getsentence([X]).
test and test.
(16 ms) no
uncaught exception: error(syntax_error('user_input:43 (char:10) . or
    operator expected after expression'),read_term/3)
```

Exercise

- Exercise 6.4
  - Define the relation
    starts(Atom, Character)
    to check whether **Atom** starts with **Character**.

6.5 Reading programs

- **consult(F):**
  - We can tell Prolog to read a program from a file F.
  - For example:

    ?- consult(program3).
    - All the clauses in file **program3** are read and loaded into the memory.
    - They will be used by Prolog when answering further questions from the user.
    - If another file is 'consulted' at some later time during the same session, then the clauses from this new file are added into the memory.
    - However, details depend on the implementation and other circumstances.
    - If the new file contains clauses about a procedure defined in the previously consulted file, then
      - the new clauses may be simply added at the end of the current set of clauses, or
      - The previous definition of this procedure may be entirely replaced by the new one.

6.5 Reading programs

- Several files may be consulted by the same consult goal.
- For example:

  ?- consult([program3, program4, queens]).
- Such a question can also be written more simply as:

  ?- [program3, program4, queens].
- Consulted programs are used by a Prolog interpreter.

```
| ?- consult('C:/GNU-Prolog/Prologcode/programs/fig1_8.pl').
compiling C:\GNU-Prolog\Prologcode\programs\fig1_8.pl for
    byte code...
C:\GNU-Prolog\Prologcode\programs\fig1_8.pl compiled, 42
    lines read - 2851 bytes written, 31 ms
yes
```

6.5 Reading programs

- If a Prolog implementation also features a compiler, then programs can be loaded in a compiled form.
  - For example:

    ?- compiler(program3).

    ?- compiler([program3, program4, queens]).

- The GNU Prolog compiler is a command-line compiler similar in spirit to a Unix C compiler like gcc. To invoke the compiler use the gplc command as follows:

  % gplc [OPTION]… FILE…

  (the % symbol is the operating system shell prompt)

- The simplest way to obtain an executable from a Prolog source file prog.pl is to use:

  % gplc prog.pl