

Chapter 6

Input and output

So far, the only input/output channel we have explored consists of variable instantiation. We might need something else such as input data from the keyboard, output in more general formats, and input/output via external files.

We first study how to direct input/output to files, and then have a look at how to input and output data in other formats.

There are two active files during the execution of a Prolog program: the *current input/output stream*. Initially, they are mapped to the user's terminal. All the files considered here are sequential, and text ones.

This part is implementation dependent, and might not work with SWI-Prolog.

Change the input stream

The current input stream can be changed to another file, `Filename`, with the following goal: `?-see(Filename)`. When succeeding, Prolog gets input from `Filename`.

A typical segment could be the follows:

```
...  
see(file1).  
read_from_file(info).  
see(user)  
...
```

The goal `seen` closes the current input file.

Change the output stream

The current output stream can be changed to another file, `Filename`, with the following goal: `?-tell(Filename)`. When succeeding, Prolog sends output to `Filename`.

A typical segment could be the follows:

```
...  
tell(file2).  
write_on_file(info).  
tell(user)  
...
```

The goal `told` closes the current output file.

Read and write

The built-in relation, `read(X)`, reads in the next term from the current input stream. Similarly, `write(X)` outputs a term to the current output stream. The relation `tab(N)` sends out N spaces to the output, and `nl` starts a new line in the output. For example,

```
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.
```

Let's try this out....

Display lists

The following procedure `writelist(L)` write out each item in `L` in a separate line.

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

If we have a list of lists, we might want to write each list per line.

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

Processing a file

A typical sequence of processing a file of terms is the following:

```
see(F), processfile, see(user),....
```

The procedure `processfile` can be implemented in the following schema:

```
processfile:-  
  read(Term),  
  process(Term).
```

```
process(end_of_file):-!.  
process(Term):-
```

```
  treat(Term),  
  processfile.
```

Here, `treat(Term)` refers to anything to be done with that `Term`, including `write(Term)`.

Another example

A more flexible way to treat a term is to write out the 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).
```

If we start with `showfile(1)`, we will end up with exactly what we wanted.

Homework: Exercises 6.1 and 6.2.

Work with characters

A character can be written out to the current output stream with the goal of `put(C)`, where `C` is the ASCII code of the character. Thus, `?-put(65),put(66)` will send out `AB`.

We can also read in a single character with `?-get0(C)`, which reads in the next character from the current input stream, and `C` is then instantiated with its ASCII code. The `get()` will do the similar thing, except that it only reads in non-blank characters. Thus, it will skip over all the blanks.

```
squeeze:-get0(C),put(C),dorest(C).
```

```
dorest(46):-!.
```

```
dorest(32):-!,get(C),put(C),dorest(C).
```

```
dorest(Letter):-squeeze.
```

Work with atoms

There is a built-in procedure, `name(A, L)`, while is true iff `L` is a list of ASCII codes for the characters in `A`. For example, `name(zx2, [122, 120, 50])` is true.

We can use it to either break an atom to single characters, or combine a bunch of them into an atom. For example, the following code checks if 'taxi' is part of a word.

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

Natural language processing

We now define `getsentence(WordList)`, that will read in a sentence, and initiates `WordList` to represent this sentence. Naturally, each word is represented as an atom, and the sentence is represented as a list of those 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).
```

We finish this component by giving another relation that will process individual letters.

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

We will talk more about natural language processing in a later chapter.

Read in a program

The easy way to read in a program in our case is to double click the program, SWI-Prolog will then automatically read it in, and compile it. If anything wrong, it will complain, and a debugging process will begin.

In general, if the working directory is properly set up, then the built-in predicate `?-consult(filename)` will do the same thing.

Homework: Exercise 6.6.