IKI30820 – Logic Programming
Input and Output
Slide 07

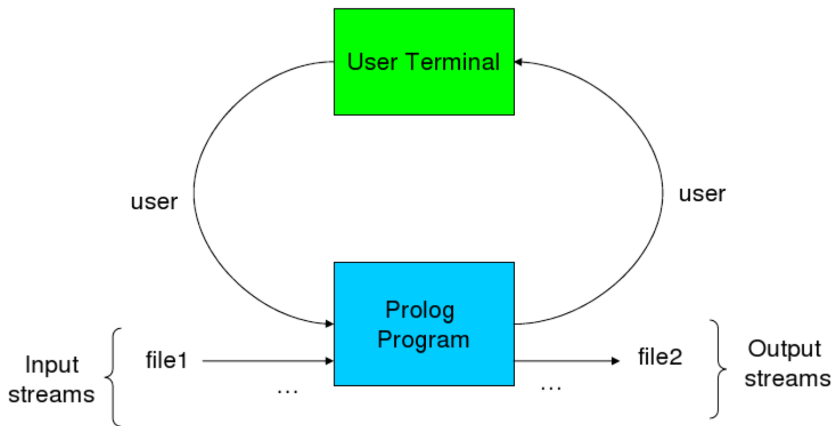
Ari Saptawijaya (slides) Adila A. Krisnadhi (LaTeX adaptation)

Fakultas Ilmu Komputer
Universitas Indonesia

2009/2010 • Semester Gasal

Outline

1. Input and Output
2. Constructing and Decomposing Terms
3. Comparison and Unification of Terms


User Terminal
user
user
Prolog
Program
Input
streams
file1
...
...
file2
Output
streams

Input and Output

- Prolog program:
  - read data from input streams
  - write data to output streams
- Streams are associated with files
- 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
- Both data input from/output to the terminal ("pseudo-files") are referred to by the name **user**
- The name of other files can be chosen by the programmer according to the rules of the OS

Current Input/Output Streams

- At any time during the execution, only two files (streams) are active: one for input and one for output
- At the beginning of execution, these two streams correspond to the user's terminal
- The current input stream can be changed to `Filename` by the goal:

```
see(Filename).
```

- The current output stream can be changed to `Filename` by the goal:

```
tell(Filename).
```

Built-in predicates for I/O

- **see**(+SrcDst)
  Open SrcDest for reading and make it the current input stream
- **tell**(+SrcDest)
  Open SrcDest for writing and make it the current output stream
- append(+**File**)
  Similar to **tell**/1, but positions the file pointer at the end of **File** rather than truncating an existing file.
- **seeing**(?SrcDest)
  Obtain the current input stream
- **telling**(?SrcDest)
  Obtain the current output stream
- **seen**
  Close the current input stream.
- **told**
  Close the current output stream.

ISO compliant I/O predicates

- `open(+SrcDest,+Mode,?Stream)`
  Open a stream.
  - `Mode` is one of read, write, append, or update.
    - Mode append opens the file for writing, positioning the file-pointer at the end.
    - Mode update opens the file for writing, positioning the file-pointer at the beginning of the file without truncating the file.
  - `Stream` is either a variable, in which case it is bound to an integer identifying the stream.
- **`close`**`(+Stream)`
  Close the specified stream.
  - If `Stream` is not open, an error message is displayed
  - If the closed stream is the current input or output stream, the terminal is made the current input or output.

Character I/O

- **nl**
  Write a new line character to the current output stream.
- **nl**(+Stream)
  Write a newline to Stream.
- put(+**Char**)
  Write **Char** to the current output stream, **Char** is either an integer-expression evaluating to a character code or an atom of one character.
- put(+Stream, +**Char**)
  Write **Char** to Stream.

- **tab**(+Amount)
  Writes Amount spaces on the current output stream. Amount should be an expression that evaluates to a positive integer
- **tab**(+Stream, +Amount)
  Writes Amount spaces to Stream.
- get0(-Char)
  Read the current input stream and unify the next character with **Char**. **Char** is unified with -1 on end of file
- get0(+Stream, -**Char**)
  Read the next character from Stream
- get(-**Char**)
  Read the current input stream and unify the next non-blank character with **Char**. **Char** is unified with -1 on end of file.
- get(+Stream, -**Char**)
  Read the next non-blank character from Stream.

Term I/O

- **write**(+Term)
  Write Term to the current output, using brackets and operators where appropriate.
- **write**(+Stream, +Term)
  Write Term to Stream
- **read**(-Term)
  Read the next Prolog term from the current input stream and unify it with Term. On reaching end-of-file Term is unified with the atom end_of_file.
- **read**(+Stream,-Term)
  Read Term from Stream

Please see the SWI-Prolog Reference Manual for other I/O predicates.

Example 1: I/O Interaction

```
cube :-
  write('Input number or stop (end with .): '),
  read(X), process(X).

process(stop) :- !.
process(N) :-
  Result is N*N*N,
  write('The cube of '), write(N),
  write(' is '), write(Result),
  nl, cube.
```

```
?- cube.
Input number or stop (end with .): 2.
The cube of 2 is 8
Input number or stop (end with .): 0.5.
The cube of 0.5 is 0.125
Input number or stop (end with .): sin(pi/6).
The cube of sin(pi/6) is 0.125
Input number or stop (end with .): 4.
The cube of 4 is 64
Input number or stop (end with .): 2*2.
The cube of 2*2 is 64
Input number or stop (end with .): stop.

Yes
```

Example 2: Processing characters

```
/* Read a sentence (which ends with a full stop)
    from the current input stream, and output the
    same sentence reformatted so that multiple
    blanks between words are replaced by single
    blank.
    There are three mutually exclusive cases:
    - the character is a full stop,
    - the character is a blank, or
    - the character is a letter
    Use cut to handle these cases */

squeeze :- get0(C), put(C), dorest(C).
dorest(46) :- !.        % 46 is ASCII code for "."
dorest(32) :- !, get(C),   % skip other blanks
    put(C), dorest(C).
dorest(_) :- squeeze.
```

```
?- squeeze.
|: Learning logic programming is    fun.
Learning logic programming is fun.

Yes
```

Example 3: Displaying List

```
/* Given a list of lists, display the elements of
   each list in one line */

displaylist([]).
displaylist([L|LL]) :- doline(L), nl,
                       displaylist(LL).
doline([]).
doline([H|T]) :- write(H), tab(2),
                 doline(T).

?- displaylist([[a,b,c],[d,e,f],[g,h]]).
a  b  c
d  e  f
g  h
```

Example 4: Processing a file of terms

```
/* Display on the terminal each term from
    a file together with its consecutive
    number */

go :-
    write( Input file name:    ), read(Fin),
    nl,
    see(Fin), showfile(1), seen.

showfile(N) :-
    read(Term), show(Term,N).
show(end_of_file,_) :- !.
show(Term,N) :-
    write(N), tab(3), write(Term),
    nl,
    N1 is N+1,
    showfile(N1).
```

```
?- go.
Input file name: abc.

1   sin(pi)
2   hobbit(frodo)
3   cos(a)
4   suka(unyil, bakso)
5   sin(30)
6   [a, b, c]
7   hello

Yes
```

Constructing and decomposing atoms

- Often it is desirable to have information, input as a sequence of characters, but represented in the program as an atom
- Some related built-in predicates:
  - **name**(?AtomOrInt, ?String)
    - String is a list of ASCII values describing **Atom**. Each of the arguments may be a variable, but not both.
    - When String is bound to an ASCII value list describing an integer and **Atom** is a variable, **Atom** will be unified with the integer value described by String
  - atom_chars(?**Atom**,?CharList)
    - As **name**/2, but CharList is a list of one-character atoms rather than a list of ASCII values.

```
?- name(X,"234").
X = 234
Yes

?- name(234,X).
X = [50, 51, 52]
Yes

?- name(hello,X).
X = [104, 101, 108, 108, 111]
Yes

?- atom_chars(hello,X).
X = [h, e, l, l, o]
Yes

?- atom_chars(X,[104, 101, 108, 108, 111]).
X = hello
Yes

?- name(A,[h,e,l,l,o]).
A = hello
Yes
```

Example: From a sentence to a list of words

```
% Procedure getsentence reads in a sentence and combines
% the words into a list of atoms.
getsentence( Wordlist) :-
 get0( Char),
 getrest( Char, Wordlist).
getrest( 46, [] ) :- !.          % End of sentence:
                                 % 46 = ASCII for '.'
getrest( 32, Wordlist) :- !,     % 32 = ASCII for blank
 getsentence( Wordlist).         % Skip the blank
getrest( Letter, [Word | Wordlist] ) :-
 getletters( Letter, Letters, Nextchar),
                                 % Read letters of current word
 name( Word, Letters),
 getrest( Nextchar, Wordlist).
getletters( 46, [], 46) :- !. % End of word: 46 = full stop
getletters( 32, [], 32) :- !. % End of word: 32 = blank
getletters( Let, [Let | Letters], Nextchar) :-
 get0( Char),
 getletters( Char, Letters, Nextchar).
```

```
?- getsentence(Wordlist).
|: Learning logic programming      is fun.
Wordlist = ['Learning', logic, programming, is, fun]

Yes
```

- The possible use of getsentence program is in natural language processing
- Sentences represented as lists of words are in form that is suitable for further processing
- For example, searching for certain keywords in input sentences

Outline

1 Input and Output

2 Constructing and Decomposing Terms

3 Comparison and Unification of Terms

Constructing and decomposing terms

- **functor**(?Term, ?**Functor**, ?Arity)
  Succeeds if Term is a term with functor **Functor** and arity Arity.
  - If Term is a variable it is unified with a new term holding only variables.
  - If Term is an atom or number, **Functor** will be unified with Term and arity will be unified with the integer 0 (zero).
- **arg**(?N, ?Term, ?Value)
  - Term should be instantiated to a term, N to an integer between 1 and the arity of Term.
  - Value is unified with the N-th argument of Term.
  - N may also be unbound. In this case Value will be unified with the successive arguments of the term. On successful unification, N is unified with the argument number.

- `?Term =.. ?List`
  - The head of the list `List` is the functor of `Term` and the remaining elements of `List` are the arguments of the term.
  - Each of the arguments may be a variable, but not both.
- `copy_term(+In, -Out)`
  - Create a version of `In` with renamed (fresh) variables and unify it to `Out`.

```
?- f(a,b,c) =.. L.
L = [f, a, b, c]

?- D =.. [date, 17, nov, 2006].
D = date(17, nov, 2006)

?- hello =.. List.
List = [hello]

?- functor(f(t, g(X), sin(pi)), F, A)
F=f
A=3

?- arg(2, f(b, sin(0), t(a)), X).
X = sin(0)

?- copy_term(f(X,a),Y).
X = _G378
Y = f(_G458, a) ;
No
```

```
% A procedure for substituting a subterm of a term by
% another subterm.
% substitute( Subterm, Term, Subterm1, Term1):
% if all occurrences of Subterm in Term are substituted
% with Subterm1 then we get Term1.

% Case 1: Substitute whole term
substitute( Term, Term, Term1, Term1) :- !.

% Case 2: Nothing to substitute
substitute( _, Term, _, Term) :- atomic(Term), !.

% Case 3: Do substitution on arguments
substitute( Sub, Term, Sub1, Term1) :-
    Term =.. [F|Args],                        % Get arguments
    substlist( Sub, Args, Sub1, Args1), % Perform substitution
    Term1 =.. [F|Args1].
substlist( _, [], _, []).
substlist( Sub, [Term|Terms], Sub1, [Term1|Terms1]) :-
    substitute( Sub, Term, Sub1, Term1),
    substlist( Sub, Terms, Sub1, Terms1).
```

```
?- substitute(sin(t),f(a+sin(t),g(sin(t))),cos(s),H).
H = f(a+cos(s), g(cos(s)))

?- substitute(f(a), b, g(b), X).
X=b
```

Outline

1 Input and Output

2 Constructing and Decomposing Terms

3 Comparison and Unification of Terms

Standard order of terms

Terms are ordered in the so called “standard order”. This order is defined as follows:

1. Variables ¡ Numbers ¡ Atoms ¡ Strings ¡ Terms
2. Atoms are compared alphabetically.
3. Strings are compared alphabetically.
4. Numbers are compared by value. Integers and floats are treated identically.
5. Compound terms are first checked on their arity, then on their functor-name (alphabetically) and finally recursively on their arguments, leftmost argument first.

Some Built-in Predicates I

- `+Term1 == +Term2`
  Succeeds if Term1 is equivalent to Term2.
- `+Term1 \== +Term2`
  Equivalent to `\+Term1 == Term2.`
- `+Term1 = +Term2`
  Unify Term1 with Term2. Succeeds if the unification succeeds.
  `unify_with_occurs_check(+Term1, +Term2)`
  As =/2, but using sound-unification. That is, a variable only unifies to a term if this term does not contain the variable itself. To illustrate this, consider the two goals below:

```
?- A = f(A).
A = f(f(f(f(f(f(f(f(f(f(...)))))))))).

?- unify_with_occurs_check(A, f(A)).
No.
```

- `+Term1 \= +Term2`
  Equivalent to `\+Term1 = Term2.`

Examples

```
?- f(a,b) == f(a,b).
?- f(a,b) == f(a,X).
?- f(a,X) == f(a,Y).
?- f(a,X) =@= f(a,Y).
?- beauty @< beast.
?- a(b,c) @< f(h).
?- a(b,c) @< a(c,b).
?- g(f(C),d) @< g(h(B),d).
?- g(f(C),d) @< g(a(B,e),d).
?- compare(X,ui,itb).
?- unifiable(sin(X),sin(pi),_).
?- unifiable(sin(X),sin(pi),_),sin(X)=sin(30).
?- sin(X)=sin(pi), sin(X)=sin(30).
```

Some Built-in Predicates I

- `+Term1 =@= +Term2`
  Succeeds if Term1 is 'structurally equal' to Term2. Two terms are structurally equal if their tree representation is identical and they have the same 'pattern' of variables. Examples:

```
a =@= A             false
A =@= B             true
x(A,A) =@= x(B,C)   false
x(A,A) =@= x(B,B)   true
x(A,B) =@= x(C,D)   true
```

- `+Term1 \=@= +Term2`
  Equivalent to `\+Term1 =@= Term2`.
- `+Term1 @< +Term2`
  Succeeds if Term1 is before Term2 in the standard order of terms.
- `+Term1 @=< +Term2`
  Succeeds if both terms are equal (==/2) or Term1 is before Term2 in the standard order of terms.

Some Built-in Predicates II

- `+Term1 @> +Term2`
  Succeeds if Term1 is after Term2 in the standard order of terms.
- `+Term1 @>= +Term2`
  Succeeds if both terms are equal (==/2) or Term1 is after Term2 in the standard order of terms.
- `compare(?Order, +Term1, +Term2)`
  Determine or test the Order between two terms in the standard order of terms. Order is one of ¡, ¿, or =, with the obvious meaning.
- `unifiable(@X, @Y, -Unifier)` If X and Y can unify, unify Unifier with a list of Var = Value, representing the bindings required to make X and Y equivalent.