

Functional and Logic Programming

Bachelor in Informatics and Computing Engineering
2024/2025 - 1st Semester

Prolog

Non-logical Features

Agenda

- Cut
- Input / Output
- Useful Predicates / Libraries

Cut

- Backtracking in Prolog can lead to some inefficiency
 - Branches that lead to no feasible solution are still explored
- Solution: cut (!)
 - Always succeeds as a goal (can be ignored in a declarative reading), binding Prolog to all choices made since the parent goal unified with the clause where the cut is
 - Prunes all clauses for the same predicate below the one where the cut is
 - Prunes all alternative solutions to the goals left of the cut in the clause
 - Does not prune the goals to the right of the cut in the clause
 - They can produce several solutions via backtracking
 - Backtracking to the cut fails and causes backtracking to the last choice point

Cut

- Example: remember the definition of *member* / *memberchk*

```
member( X, [X|_] ).  
member( X, [_|T] ) :-  
    member(X, T) .
```

```
memberchk( X, [X|_] ) .  
memberchk( X, [Y|T] ) :-  
    X \= Y,  
    memberchk(X, T) .
```

```
memberchk( X, [X|_] ) :- ! .  
memberchk( X, [_|T] ) :-  
    memberchk(X, T) .
```

Cut

• Another example

```
a(X, Y) :- b(X), !, b(Y).
a(3, 4).
b(2).
b(3).
```

```
| ?- a(X, Y).
X = 2,
Y = 2 ? ;
X = 2,
Y = 3 ? ;
no
```

```
| ?- a(X, Y).
1      1 Call: a(_1011,_1051) ?
2      2 Call: b(_1011) ?
?      2 Exit: b(2) ?
3      2 Call: b(_1051) ?
?      2 Exit: b(2) ?
?      1 Exit: a(2,2) ?
X = 2,
Y = 2 ? ;
1      1 Redo: a(2,2) ?
3      2 Redo: b(2) ?
3      2 Exit: b(3) ?
1      1 Exit: a(2,3) ?
X = 2,
Y = 3 ? ;
no
```

Cut

- Remember the solution to sum all numbers between 1 and N
 - Now with a cut!

```
sumN(N, Sum) :- sumN(N, Sum, 0).
sumN(0, Sum, Sum) :- !.
```

```
sumN(N, Sum, Acc) :- N > 0,
                     N1 is N-1,
                     Acc1 is Acc + N,
                     sumN(N1, Sum, Acc1).
```

Is $N > 0$ still necessary?

```
| ?- sumN(2, S, 0).
      1      1 Call: sumN(2,_903,0) ?
      2      2 Call: 2>0 ?
      2      2 Exit: 2>0 ?
      3      2 Call: _2081 is 2-1 ?
      3      2 Exit: 1 is 2-1 ?
      4      2 Call: _2099 is 0+2 ?
      4      2 Exit: 2 is 0+2 ?
      5      2 Call: sumN(1,_903,2) ?
      6      3 Call: 1>0 ?
      6      3 Exit: 1>0 ?
      7      3 Call: _9391 is 1-1 ?
      7      3 Exit: 0 is 1-1 ?
      8      3 Call: _9409 is 2+1 ?
      8      3 Exit: 3 is 2+1 ?
      9      3 Call: sumN(0,_903,3) ?
      9      3 Exit: sumN(0,3,3) ?
      5      2 Exit: sumN(1,3,2) ?
      1      1 Exit: sumN(2,3,0) ?

S = 3 ?
yes
```

Red vs Green Cut

- **Red cut** is one that influences the results
 - If we remove the cut, the results will be different

```
a(A, B) :- b(A), !, b(B).
a(3, 4).
b(2).
b(3).
```

```
?- a(X, Y).
X = 2,
Y = 2 ? ;
X = 2,
Y = 3 ? ;
no
```

```
a(A, B) :- b(A), b(B).
a(3, 4).
b(2).
b(3).
```

```
?- a(X, Y).
X = 2,
Y = 2 ? ;
X = 2,
Y = 3 ? ;
X = 3,
Y = 2 ? ;
X = 3,
Y = 3 ? ;
X = 3,
Y = 4 ? ;
no
```

Red vs Green Cut

- **Green cut** is one that does not influence results, but is used to increase efficiency
 - If we remove the cuts, the results will be the same, but Prolog will explore branches that won't lead to any possible solution

```
classify(BMI, 'low weight'):- BMI < 18.5, !.  
classify(BMI, 'normal weight'):- BMI >= 18.5, BMI < 25, !.  
classify(BMI, 'excessive weight'):- BMI >= 25, BMI < 30, !.  
classify(BMI, 'obesity'):- BMI >= 30, !.
```

Trace a call to `classify(20, Class)` to see the differences!

Negation as Failure

- Negation can be attained by using a cut

```
not(X) :- X, !, fail.  
not(_X).
```

Is this cut red or green?

- *Fail* always fails (just as *true* always succeeds)
- The cut is necessary to ensure the second clause is not reached when backtracking

Can we change the order of these clauses?

Negation as Failure

- Negation should be used with ground terms (no variables in the goal), or ‘strange’ results may occur

- Example: determine if a man is not a father

```
not_a_father(X) :- not(parent(X, _)), male(X).
```

- Works well with instantiated values, but what about with a variable?

```
not_a_father(bart) .  
yes
```

```
not_a_father(X) .  
no
```

- Change the order of the goals so that variables in the negated goal are ground (possibly instantiated by other goals in the clause)

```
not_a_father(X) :- male(X), not(parent(X, _)).
```

Conditional as Failure

- We can attain a conditional execution by using two clauses with a mutually exclusive condition verification

```
pred_ite(If, Then, _Else):- If, Then.  
pred_ite(If, _Then, Else):- not(If), Else.
```

Why is `not(If)` necessary?

- Conditional execution can also be attained by using a cut

```
if_then_else(If, Then, _Else):- If, !, Then.  
if_then_else(_If, _Then, Else):- Else.
```

Is this cut red or green?

Cut – Notes on use

- Ensure that the predicates where the cut is used work as intended (including variations of argument instantiation)

```
max(A, B, B) :- B >= A.
max(A, B, A) :- A > B.
```

- No need to backtrack; add a cut to improve efficiency

```
max(A, B, B) :- B >= A, !.
max(A, B, A) :- A > B.
```

- No need for test in second clause; remove it

```
max(A, B, B) :- B >= A, !.
max(A, B, A) .
```

What happens now?

| ?- max(1, 2, 2).

| ?- max(1, 2, 1).

Cut – Notes on use

- Use cuts sparingly, and *only* at proper places
 - A cut should be placed at the exact point that it is known that the current choice is the correct one: no sooner, no later
- Make cuts as local in their effect as possible
 - If a predicate is intended to be determinate, then define it as such; do not rely on its callers to prevent unintended backtracking

See SICStus Manual, section 9 – Writing Efficient Programs

Agenda

- Cut
- **Input / Output**
- Useful Predicates / Libraries

Input / Output

- Input / Output is based on streams, used either for reading or writing, in text (characters and terms) or binary (bytes) mode
 - At any one time there is one current input stream and one current output stream (by default the user's terminal)
 - I/O predicates operate on the corresponding current stream
 - All predicates support additional parameter (as the first one) specifying the stream to read from / write to
- Input and output cannot be undone, but variable binding (from input predicates) is undone when backtracking

Input / Output

- Prolog provides several predicates for input and output
 - ***read/1*** reads a term (by default, from the standard input)
 - Input needs to end with a period
 - If a compound term is being read, input must match term being read
 - ***write/1*** writes a term
 - ***nl/0*** prints a new line

```
| ?- write('Hello World'),nl.  
Hello World  
yes
```

```
| ?- read(_X), read(_Y/_Z), write(_X-_Y), nl, write(_Z-_X).  
|: 3.  
|: 4/a.  
3-4  
a-3  
yes
```

```
| ?- write("Hello World"),nl.  
[72,101,108,108,111,32,87,111,114,108,100]  
yes
```


Input / Output

- *get_char* obtains a single character
- *get_code* obtains the ASCII code of a single character
- *put_char* prints a single character
- *put_code* prints a single character given its ASCII code
- *char_code(?Atom, ?Code)* allows converting between character and corresponding ASCII code
- *get_byte* and *put_byte* read and write binary data
- *peek_char*, *peek_code* and *peek_byte* obtain a single character / code / byte without consuming it from the input stream
- *format* prints terms with specified formatting options

Input / Output

```
| ?- get_code(_X), _Y is _X+3, put_code(_Y).  
|: asd  
d  
yes  
  
.  
! Existence error in user:sd/0  
! procedure user:sd/0 does not exist  
! goal: user:sd  
| ?-
```

- ***skip_line*** skips any input until the end of the line
 - It is OS independent

```
| ?- get_code(_X), skip_line, _Y is _X+3, put_code(_Y).  
|: asd  
d  
yes  
| ?-
```

skip_line can be very useful!

File Input / Output

- There are some useful predicates to work with files
 - ***see/1*** opens a file for reading
 - The file is used for reading instead of the standard input
 - ***seen/0*** closes the file that was opened for reading
 - ***tell/1*** opens a file for writing
 - The file is used for writing instead of the standard output
 - ***told/0*** closes the file that was opened for writing
- Other predicates exist to open, manage and close streams

See section 4.6 of the SICStus Manual for more information on Input and Output

Agenda

- Cut
- Input / Output
- Useful Predicates / Libraries

Code Organization

- You can (should) organize your code in different files, for increased modularity and readability
- Several directives can be used to import files
 - *use_module(library(lib_name))* % for libraries or modules
 - *consult(file_to_load)*
 - *[file_to_load]*
 - *ensure_loaded(file_to_load)*
 - *include(file_to_include)*

See section 4.3 of the SICStus Manual for more information on loading programs

Repeat

- *repeat* always succeeds
 - Can be used to repeat some portion of code until it succeeds

```
read_value(X) :-  
    repeat,  
    write('write hello'),  
    read(X),  
    X = hello.
```

- It may be useful to use a cut after reaching the condition to break the cycle, to avoid undesired backtracking

Between

- ***between(+Lower, +Upper, ?Number)*** can be used both to test and generate integers between given bounds
 - Necessary to include the *between* library

```
| ?- between(1, 6, 4) .  
yes  
| ?- between(1, 6, 9) .  
no  
| ?- between(1, 3, X) .  
X = 1 ? ;  
X = 2 ? ;  
X = 3 ? ;  
no
```

See section 10.6 of the SICStus Manual for more information on generating integers

Hint: you can use *repeat* together with *between* to test for valid coordinate input in the practical assignment

Random

- Random library provides several predicates for generating random numbers
 - *maybe / maybe(+Probability)*
 - *random(+Lower, +Upper, -Value)*
 - *random_member(-Element, +List)*
 - *random_select(?Element, ?List, ?Rest)*
 - *random_permutation(?List, ?Permutation)*

See section 10.38 of the SICStus Manual for more information on random number generation

Q & A

