Cuts and Negation

CS3100 Fall 2019

Review

Previously

- Generate and Test: A design pattern for logic programming

This lecture

- Cuts
  - A mechanism for pruning Prolog search trees
  - Red and Green cuts

Evaluator

Consider a simple evaluator for arithmetic expressions.

In [1]:

```
eval(plus(A,B),C) :- eval(A,VA), eval(B,VB), C is VA + VB.
eval(mult(A,B),C) :- eval(A,VA), eval(B,VB), C is VA * VB.
eval(A,A).
```

```
Added 3 clauses(s).
```

Evaluator

What is the result of evaluating `1 + (4 * 5)` ?

In [2]:

```
?- eval(plus(1,mult(4,5)),X) {1}.
```

```
X = 21 .
```

In [3]:

```
?- eval(plus(1,mult(4,5)),X) {2}.
```

```
ERROR: Caused by: '  eval(plus(1,mult(4,5)),X) '. Returned: 'error(typ
e_error(evaluable, /(mult, 2)), context(:(system, /(is, 2)), _1840))'.
```

Trace `eval(plus(1,mult(4,5)),X)` by hand.

Fixing the evaluator - with wrapper

Wrap the values in a function `value` .

In [4]:

```
eval2(plus(A,B),C) :- eval2(A,VA), eval2(B,VB), C is VA + VB.
eval2(mult(A,B),C) :- eval2(A,VA), eval2(B,VB), C is VA * VB.
eval2(value(A),A).
```

```
Added 3 clauses(s).
```

In [5]:

```
?- eval2(plus(value(1),mult(value(4),value(5))),X).
```

```
X = 21 .
```

Fixing the evaluator - with cut

- The **cut** (!) is an extra-logical (outside pure logic) operator that prunes the search trees.
- When the evaluation cross a cut, it prunes
  - All the subsequent possible branches in the parent.
  - All the subsequent possible branches in the preceeding sub-goals.

In [6]:

```
eval3(plus(A,B),C) :- !, eval3(A,VA), eval3(B,VB), C is VA + VB.
eval3(mult(A,B),C) :- !, eval3(A,VA), eval3(B,VB), C is VA * VB.
eval3(A,A).
```

```
Added 3 clauses(s).
```

In [7]:

```
?- eval3(plus(1,mult(4,5)),X).
```

```
X = 21 .
```

Cut behaviour

In [8]:

```
p(a).
p(b).
r(c).
q(X) :- !, p(X).
q(X) :- r(X).
```

```
Added 5 clauses(s).
```

In [9]:

```
?- q(X).
```

```
X = a ;
X = b .
```

Quiz

What does `split/3` do?

```
split([],[],[]).
split([H|T],[H|L],R) :- H < 5, split(T,L,R).
split([H|T],L,[H|R]) :- H >= 5, split(T,L,R).
```

It splits the given list into elements less than 5 and greater than or equal to 5.

Split

In [10]:

```
split([],[],[]).
split([H|T],[H|L],R) :- H < 5, split(T,L,R).
split([H|T],L,[H|R]) :- H >= 5, split(T,L,R).
```

```
Added 3 clauses(s).
```

In [11]:

```
?- split([1,2,3,4,5,6,7,8,9],L,R).
```

```
R = [ 5, 6, 7, 8, 9 ], L = [ 1, 2, 3, 4 ] .
```

- Observe that the last two cases are mutually exclusive.
  - But Prolog still searches through the third rule, if second rule was successfully matched.

Split with cut

In [12]:

```
split([],[],[]).
split([H|T],[H|L],R) :- H < 5, !, split(T,L,R).
split([H|T],L,[H|R]) :- H >= 5, !, split(T,L,R).
```

```
Added 2 clauses(s).
```

- The second `!` is unnecessary as there are no further choices.
  - In fact, the predicate `H >= 5` is unnecessary since the only way to end up here is if the first rule failed.
  - But better to leave it there for readability.

- Recommendation:
  - Use cut to optimise execution, but retain predicates which help readability.

Quiz

What is the logical meaning of these clauses?

```
p :- a,b.
p :- c.
```

1. $p \leftrightarrow (a \wedge b) \vee c$.
2. $p \leftrightarrow a \wedge b \wedge c$.
3. $p \leftrightarrow (a \wedge b) \vee (\neg a \wedge c)$.
4. $p \leftrightarrow a \wedge (b \vee c)$.

Quiz

What is the logical meaning of these clauses?

```
p :- a,b.
p :- c.
```

1. $p \leftrightarrow (a \wedge b) \vee c$. ✓
2. $p \leftrightarrow a \wedge b \wedge c$.
3. $p \leftrightarrow (a \wedge b) \vee (\neg a \wedge c)$.
4. $p \leftrightarrow a \wedge (b \vee c)$.

Quiz

What is the logical meaning of these clauses?

```
p :- a,!,b.
p :- c.
```

1. $p \leftrightarrow (a \wedge b) \vee c$.
2. $p \leftrightarrow a \wedge b \wedge c$.
3. $p \leftrightarrow (a \wedge b) \vee (\neg a \wedge c)$.
4. $p \leftrightarrow a \wedge (b \vee c)$.

Quiz

What is the logical meaning of these clauses?

```
p :- a,!,b.
p :- c.
```

1. $p \leftrightarrow (a \wedge b) \vee c$.
2. $p \leftrightarrow a \wedge b \wedge c$.

3. $p \leftrightarrow (a \wedge b) \vee (\neg a \wedge c)$. ✓
4. $p \leftrightarrow a \wedge (b \vee c)$.

Red and Green cuts

```
p :- a,!,b.
p :- c.
```

Since the cut above changes the logical meaning of the program, it is known as **Red cut**.

```
split([],[],[]).
split([H|T],[H|L],R) :- H < 5, !, split(T,L,R).
split([H|T],L,[H|R]) :- H >= 5, split(T,L,R).
```

The cut in split does not change the logical meaning of the program. Hence, it is called **Green cut**.

Remove Stutter

In [13]:

```
remove_stutter([],[]).
remove_stutter([H],[H]).
remove_stutter([H,H|T],L) :- !, remove_stutter([H|T],L).
remove_stutter([X,Y|T],[X|L]) :- remove_stutter([Y|T],L).
```

```
Added 4 clauses(s).
```

In [14]:

```
?- remove_stutter([1,1,2,2,2,3,4,1,1],X).
```

```
X = [ 1, 2, 3, 4, 1 ] .
```

Remove Stutter

Can be equivalently written as:

In [15]:

```
remove_stutter2([],[]).
remove_stutter2([H],[H]).
remove_stutter2([H,H|T],L) :- remove_stutter2([H|T],L).
remove_stutter2([X,Y|T],[X|L]) :- not(X=Y), remove_stutter2([Y|T],L).
```

```
Added 4 clauses(s).
```

In [16]:

```
?- remove_stutter2([1,1,2,2,2,3,4,1,1],X).
```

```
X = [ 1, 2, 3, 4, 1 ] .
```

Negation

- What is the relationship between cut and negation?
- With the use of negation, the clause is not longer a definite clause. What's going on here?

Quiz

What does this do?

```
a :- !, 1=2.
?- a.
```

1. successfully unfies 1 with 2
2. throws an exception
3. loops indefinitely
4. fails

Quiz

What does this do?

```
a :- !, 1=2.
?- a.
```

1. successfully unfies 1 with 2
2. throws an exception
3. loops indefinitely
4. fails ✓

- You can give a better name for `a/0` : `fail/0` .
  - Prolog has an built-in predicate `fail/0` , which always fails.

Quiz

What does this do?

```
a(A,A) :- !,fail.
a(_,_).
```

1. unifies the two arguments
2. succeeds if the arguments unify
3. succeeds if the arguments don't unify
4. always fails

Quiz

What does this do?

```
a(A,A) :- !,fail.
a(_,_).
```

1. unifies the two arguments
2. succeeds if the arguments unify
3. succeeds if the arguments don't unify ✓
4. always fails

You can give a better name for `a/2` : `is_different/2` .

Failure on unification

In [17]:

```
is_different(A,A) :- !,fail.
is_different(_,_).
```

```
Added 2 clauses(s).
```

In [18]:

```
?- is_different(m,n).
```

```
true.
```

Behaviour of fail and is_different

- Clauses such as 'fail' and 'isDifferent' can cause us to backtrack in unusual ways.
- This will undo any variable unifications along the way.

Negation by failure

We can now implement `not` using negation-by-failure.

```
not(A) :- A,!,fail.
not(_).
```

In [19]:

```
?- not(1=2).
```

```
true.
```

- `not/2` is a built-in in Prolog.
- You may also write `not(A)` as `\+A` .

Quiz

What sort of a cut is this?

```
not(A) :- A,!,fail.
not(_).
```

1. red.
2. amber.
3. green.

Quiz

What sort of a cut is this?

```
not(A) :- A,!,fail.
not(_).
```

1. red. ✓
2. amber.
3. green.

If the cut were not there, then the first rule would always fail, but the second rule will always succeed.

Closed world assumption

Everything that is true in the "world" is stated (or can be derived from) the clauses in the program.

`not` is based on the closed world assumption.

`not(A)` holds if it cannot be shown from the given clauses that `A` holds.

Example: Buying a phone

In [20]:

```
goodPhone(iphone11pro).
goodPhone(oneplus7tpro).
expensive(iphone11pro).

bargain(X) :- goodPhone(X), not(expensive(X)).
```

```
Added 4 clauses(s).
```

In [21]:

```
?- bargain(X).
```

```
X = oneplus7tpro .
```

Simple mistake with negation

In [22]:

```
goodPhone2(iphone11pro).
goodPhone2(oneplus7tpro).
expensive2(iphone11pro).

bargain2(X) :- not(expensive2(X)), goodPhone2(X).
```

```
Added 4 clauses(s).
```

In [23]:

```
?- bargain2(X).
```

```
false.
```

What went wrong?

Trace through:

```
goodPhone2(iphone11pro).
goodPhone2(oneplus7tpro).
expensive2(iphone11pro).

not(A) :- A,!,fail.
not(_).

bargain2(X) :- not(expensive2(X)), goodPhone2(X).

?- bargain2(X).
```

When using negation remember the quantifier

Our negation is not a logical one.

`expensive2(X)` is $\exists x.\, expensive2(x)$.

Our `not(expensive2(X))` is $\neg(\exists x.\, expensive2(x)) \equiv \forall x.\, \neg expensive2(x))$

Hence, the rule

```
bargain2(X) :- not(expensive2(X)), goodPhone2(X).
```

will only succeed if there are no expensive restaurants, which is not our intention.

- **Recommendation**: Use `not(T)` only when `T` is ground.
  - This was the case in the first example.

Fin.