CPSC 312

Functional and Logic Programming

17 September 2015

Office Hours

- Instructor: (Sara Sagaii) sarams@cs.ubc.ca
  - Tuesdays 10-11am
  - **Thursdays 10:30-11:30am**
  - ICCS 187
- Rui Ge:
  - Wednesdays 10-12am
  - Table 1 at DLC
- Susanne Bradley:
  - Fridays 11:30-1:30
  - Table 1 at DLC

Questions

Any questions left over from last session?

consult vs reconsult vs make

Obsolete meanings:

- `consult/1`: adds all clauses in the program
- `reconsult/1`: adds the new clauses and redefines previously defined clauses based on new ones.

new meanings (in SWI-Prolog):

- `consult/1`: wipes the old clauses and adds all the new ones.
- `make/0`: checks all loaded programs for change and consults or reconsults the ones that are changed and operates a range of other maintenance. make is slower and better suited for large projects with multiple files.
- Therefore, you can use consult to load *and* reload your small single project files.

the curious case of semi-colon

```
parent(tom, bob).
parent(pam, bob).
parent(tom, liz).
parent(bob, ann).
parent(bob, pat).
parent(pat, jim).
```

```
?- parent(bob,ann).
true ;
false.

?- parent(tom,liz).
true.
```

Why is this happening?

Semantics of semi-colon

- what happens when we type in ; ?
- What does ; mean, anyway?

Semantics of semi-colon

- what happens when we type in ; ?
- What does ; mean, anyway?
  - In prolog terms it is like saying “pretend that we failed on that last attempt”.
  - So it forces the system to backtrack and find another answer... (if there is one)

Why is prolog saying the same thing twice?

```
% jealous.pl
likes(al,betty).
likes(al,carol).
likes(betty,david).
likes(carol,david).
jealous(X,Y) :-
likes(X,Z),likes(Z,Y).
```

Why is prolog saying the same thing twice?

```
% jealous.pl
likes(al,betty).
likes(al,carol).
likes(betty,david).
likes(carol,david).
jealous(X,Y) :-
likes(X,Z),likes(Z,Y).
```

```
?- jealous(X,Y).

X = al
Y = david ;

X = al
Y = david ;

false
```

Why is prolog saying the same thing twice?

In the first example all we need is a cut (!) to stop prolog from backtracking (we will learn more about cut and backtracking later).

This is different from the first example, in that in the first example prolog wrongly assumes there is a second answer.

But here, it is correct for prolog to say the same thing twice. In English, it's because Al likes the same two girls who both happen to also like David.

This is an example of how our intended meaning of a predicate is not matching how prolog understands our expression of it.

Intuitively, we expect unique pairs from the jealous relations, but in order to achieve that, we need to write our program differently.

How can I query...

- "Is Fred the father of everyone?"

How can I query...

- "Is Fred the father of everyone?"
- "Can I query about a rule, like: is it true that if f(x) then g(x)?"

How can I query...

- "Is Fred the father of everyone?"
- "Can I query about a rule, like: is it true that if f(x) then g(x)?"
- The Answer is No.
- You can't do any one of these because queries are existentially quantified and facts and rules are both universally quantified.
- the second one you also can't do because queries are conjunctive clauses, and rules are not in conjunctive form.

But you can...

- transform your intended queries into existentially quantified forms.
- Ask instead: is there an X for which f(X) is true but g(x) isn't: (which is a logical equivalent of the negation of the original query, plus brings the query into conjunctive form)
- `?- f(X),not(g(X)).`
- Ask instead: is there an X for whom fred is not the father. But how do you ask that? can you say:
- `?- not(father(fred,X)).`

Is Fred the father of all men?

Let's re-write our father relation in a new way:

```
gender(fred,male).
gender(andy,male).
gender(bob,male).
gender(chuck,male).
parent(fred,andy).
parent(fred,bob).
parent(fred,chuck).
father(X,Y) :- gender(X,male),parent(X,Y).

?- gender(X,male), not(father(fred,X)).
```

Is Fred the father of everyone?

Who is everyone? need to bind X outside `father`. re-write:

```
person(fred)
person(andy).
person(bob).
person(chuck).
father(fred,andy).
father(fred,bob).
father(fred,chuck).

?- person(X), not(father(fred,X)).
```

Is Fred the father of everyone?

Who is everyone? need to bind X outside `father`. re-write:

```
person(fred)
person(andy).
person(bob).
person(chuck).
father(fred,bob).
father(fred,chuck).
```

```
?- person(X), not(father(fred,X)).
```

would it work if I change the order?

```
?- not(father(fred,X)), person(X).
```

changing the order of execution?

```
grandmother(X,Y) :- mother(X,Z),father(Z,Y).

grandmother(X,Y) :- mother(X,Z),mother(Z,Y).
```

changing the order of execution?

```
grandmother(X,Y) :- mother(X,Z),father(Z,Y).

grandmother(X,Y) :- mother(X,Z),mother(Z,Y).
```

```
grandmother(X,Y) :-
mother(X,Z), father(Z,Y) ; mother(X,Z),
mother(Z,Y).
```

changing the order of execution?

```
grandmother(X,Y) :- mother(X,Z),father(Z,Y).

grandmother(X,Y) :- mother(X,Z),mother(Z,Y).
```

```
grandmother(X,Y) :-
mother(X,Z), father(Z,Y) ; mother(X,Z),
mother(Z,Y).
```

```
grandmother(X,Y) :- mother(X,Z),
(father(Z,Y);mother(Z,Y)).
```

Simple Abstract Interpreter

Input: A ground **goal** `G` and a **program** P

Output: *yes* if `G` is a **logical consequence** of P,
no otherwise

Algorithm:

Initialize **resolvent** to goal `G` (the query)
while resolvent not empty do
choose a goal `A` from the resolvent
choose a ground instance of a clause
`A' :- B`$_1$`,…,B`$_n$ from program P
such that `A` and `A'` are identical
(if no such goal and clause exist, exit
the while loop)
replace `A` by `B`$_1$`,…,B`$_n$ in the resolvent
If the resolvent is empty, then output yes,
else output no

On choice

- The choice of goal to reduce is *arbitrary*; it does not matter which is chosen for the computation to succeed. If there is a successful computation by choosing a given goal, then there is a successful computation by choosing any other goal.

- The choice of the clause to effect the reduction is *nondeterministic*. Not every choice will lead to a successful computation. A nondeterministic interpreter "guesses" the clause that leads to a successful computation and chooses it.

Note on nondeterminism

- The simple abstract interpreter we discussed is nondeterministic.
- nondeterminism is a technical conceprt used to define abstract computation model.
- a nondeterministic model can choose its next operation correctly when faced with several alternatives. How? using the magic of being *abstract*.

Prolog vs the abstract interpreter

How does one implement nondeterminism?

- find a psychic
- explore all possible solutions in parallel
- depth-first search with backtracking (SWI-Prolog)

True nondeterminism cannot be realized but can be simulated or approximated.

Back to our abstract interpreter

- We saw how the algorithm works in its simplest case: where program is made up of ground facts and so is the query. Let's go one level up

- What if the program has rules?

Data Objects

Prolog has simple and structured data objects. Simple objects are

- atoms
- numbers
- variables

Atoms as Data Objects

Atoms are constructed in three ways:

- strings of letters, digits and the underscore character, starting with a lower case letter.

- strings of special characters such as + - * / < > = : . & _ ~ as long as they don't shape into character strings with reserved meanings, such as :-

- strings of characters enclosed in single quotes: 'United States'

Numbers

Numbers in a prolog program are treated similarly to atoms and they include integer numbers and real numbers.

Numbers in general are relative rare in symbolic computing. Real numbers are extra rare compared to integer that can have some use.

Variables

- Strings of letters, digits, and underscore characters. They start with an uppercase letter or an underscore:
  - X, Result, Sara, Object, Country, Student, _x, _234
  - the anonymous variable: _
  - the anonymous but named variable: _Student (for documentation purpose)

Structured Data

- Structures that are composed of a collection of simple data objects, or other structured data, connected with a predicate.
- date(27, april, 1987)
- classroom('Hugh Dempster',310)

Structured Data

Nested structured data helps with data organization and readability:

if point/2 represents a point in the two-dimensional space, e.g.

```
point(1,1)
point(2,3)
```

then a line can be represented as:

```
line(point(1,1), point(2,3))
```

or a triangle as:

```
triangle( point(4,2), point(6,4),
point(7,1) ).
```

Structured Data

- Compare this:

```
book(art_of_prolog,
0262193388,3,
sterling,shapiro,none,mit_press
,1994).

book(haskell,0201342758,0,
thompson,none,none,addison_wesl
ey,1999).
```

Structured Data

- With this:

```
book(art_of_prolog,0262193388,3,
authors(sterling,shapiro,none),
pubdata(mit_press,1994)).

book(haskell,0201342758,0,
authors(thompson,none,none),
pubdata(addison_wesley,1999)).
```

Put structure into your facts and rules where appropriate: use relations within relations.

Database Programming

Your textbook divides Prolog programs into two types:

- those that manipulate data structures
- those that define and operate on a logical database

The latter style is called Database Programming.

All programming we have been doing so far with the family trees is Database Programming.

Exercise (No Credit)

This is for you to practice your rule writing skills. Don't submit.

Expand a family tree example that we discussed last time (either the Flintstones, the no-name family tree or write your own family tree): add rules that discover a variety of family relations. For instance a rule that finds out who is a sister of whom:

```
sister(X,Y) :- female(X), female(Y), X\=Y,
parent(P,X), parent(P,Y).
```

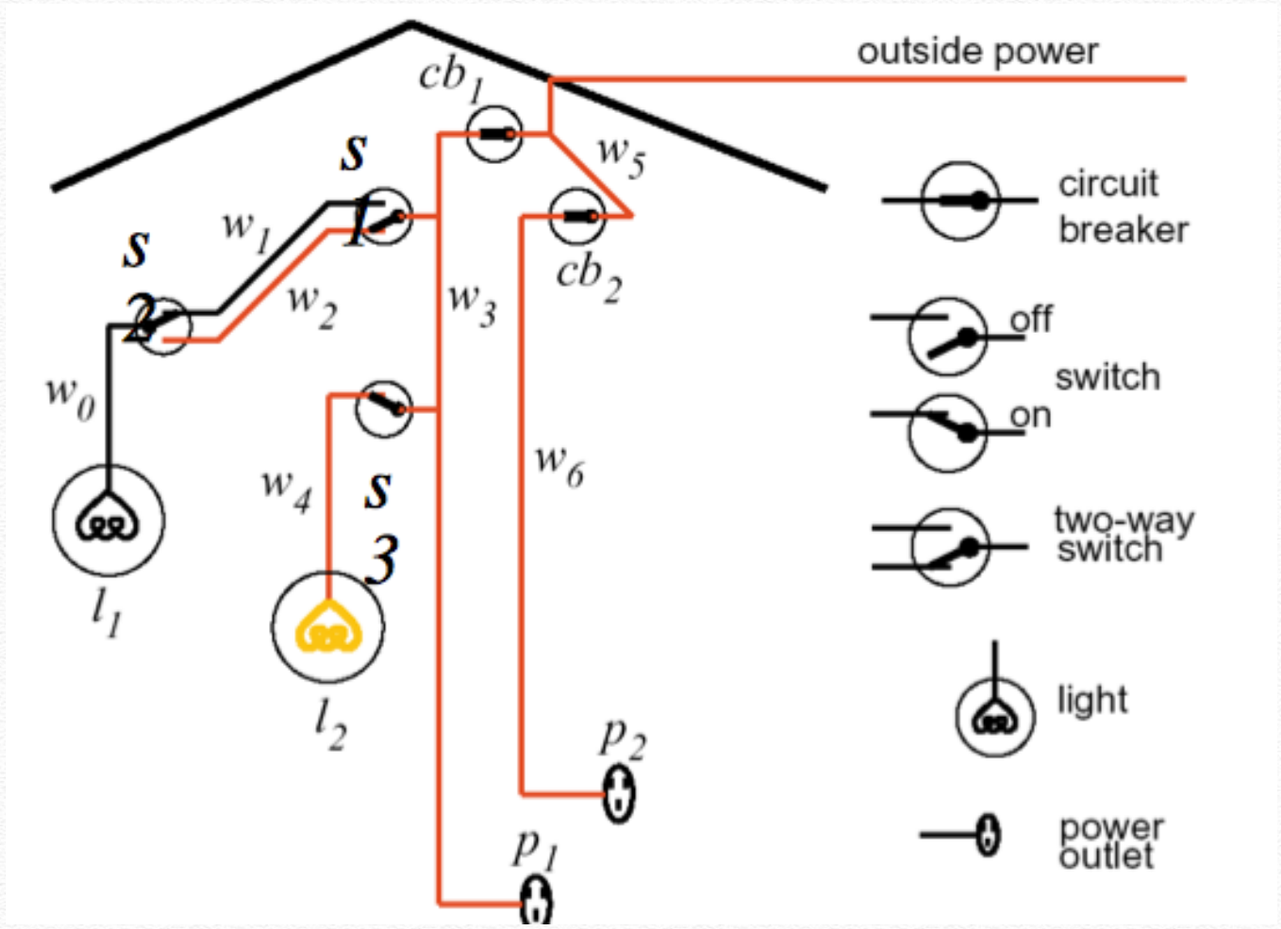
add as many new predicates as you need for your rules. You can similarly define:

- son, daughter, siblings, cousins and second-cousins.
- nephew, aunt, niece, uncle, grandparent.

For more exercises in database programming, see section 2.1 of your textbook.

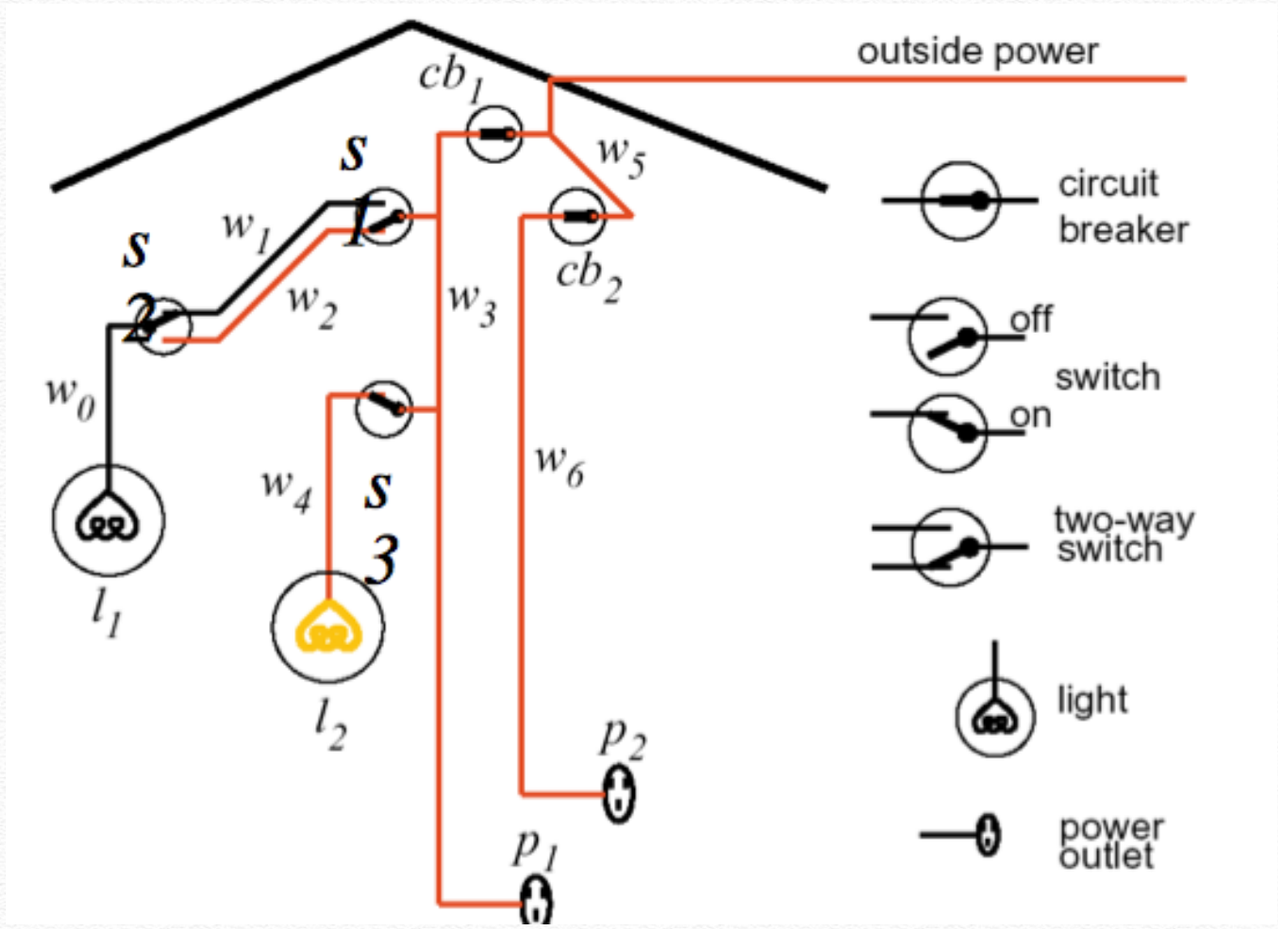
The Wiring Domain

```
light(l1).
light(l2).
down(s1).
up(s2).
up(s3).
ok(l1).
ok(l2).
ok(cb1).
ok(cb2).
connected_to(l1, w0).
connected_to(w0, w1) :- up(s2).
connected_to(w0, w2) :- down(s2).
connected_to(w1, w3) :- up(s1).
connected_to(w2, w3) :- down(s1).
connected_to(l2, w4).
connected_to(w4, w3) :- up(s3).
connected_to(p1, w3).
connected_to(w3, w5) :- ok(cb1).
connected_to(p2, w6).
connected_to(w6, w5) :- ok(cb2).
connected_to(w5, outside).
```



The Wiring Domain

```
light(l1).
light(l2).
down(s1).
up(s2).
up(s3).
ok(l1).
ok(l2).
ok(cb1).
ok(cb2).
connected_to(l1, w0).
connected_to(w0, w1) :- up(s2).
connected_to(w0, w2) :- down(s2).
connected_to(w1, w3) :- up(s1).
connected_to(w2, w3) :- down(s1).
connected_to(l2, w4).
connected_to(w4, w3) :- up(s3).
connected_to(p1, w3).
connected_to(w3, w5) :- ok(cb1).
connected_to(p2, w6).
connected_to(w6, w5) :- ok(cb2).
connected_to(w5, outside).
```

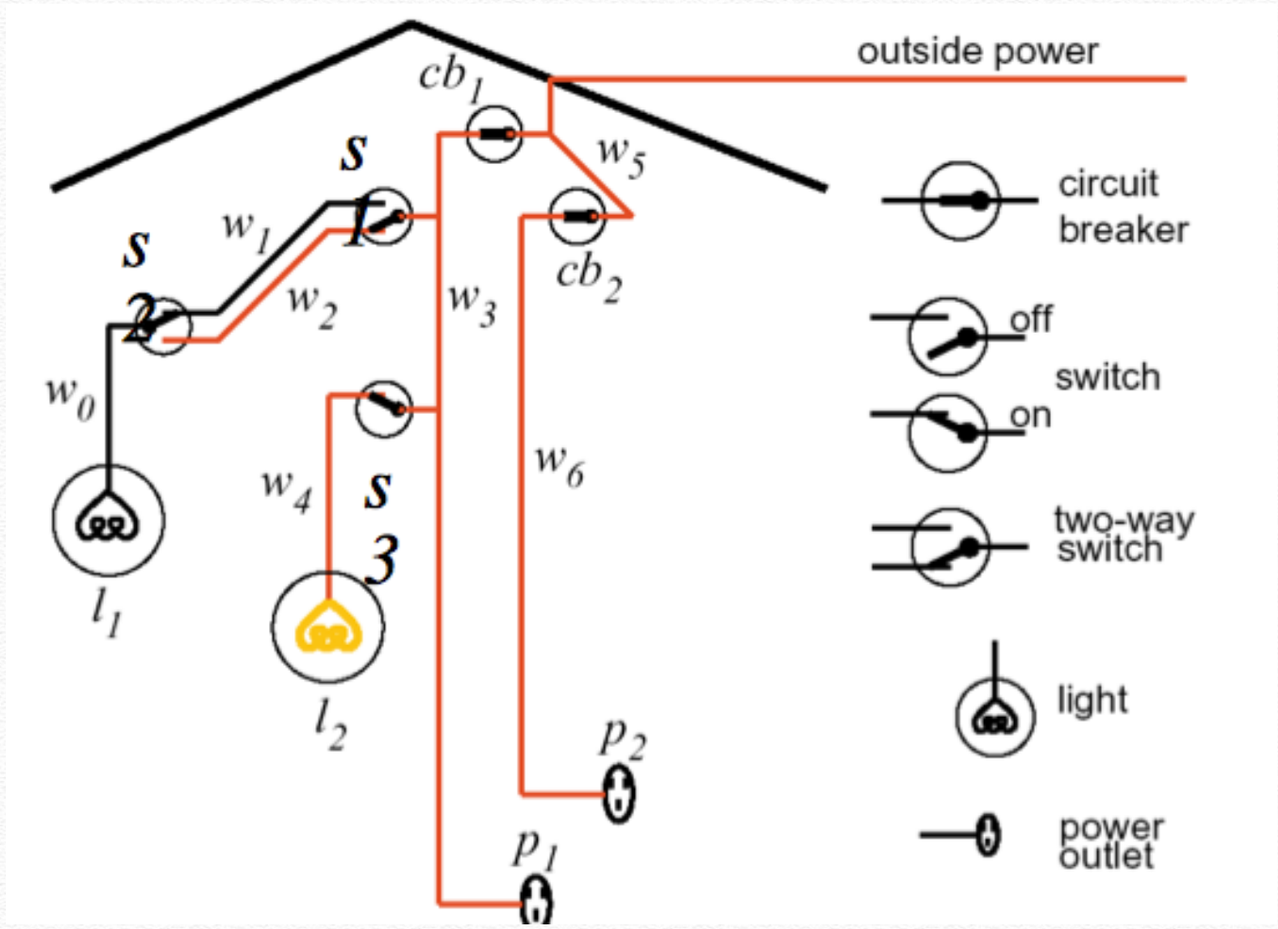


Write a procedure that proves that light l2 is illuminated and light l1 isn't.

The Wiring Domain

```
light(l1).
light(l2).
down(s1).
up(s2).
up(s3).
ok(l1).
ok(l2).
ok(cb1).
ok(cb2).
connected_to(l1, w0).
connected_to(w0, w1) :- up(s2).
connected_to(w0, w2) :- down(s2).
connected_to(w1, w3) :- up(s1).
connected_to(w2, w3) :- down(s1).
connected_to(l2, w4).
connected_to(w4, w3) :- up(s3).
connected_to(p1, w3).
connected_to(w3, w5) :- ok(cb1).
connected_to(p2, w6).
connected_to(w6, w5) :- ok(cb2).
connected_to(w5, outside).

illuminated(L) :-
light(L),ok(L),connection(L,outside).
```

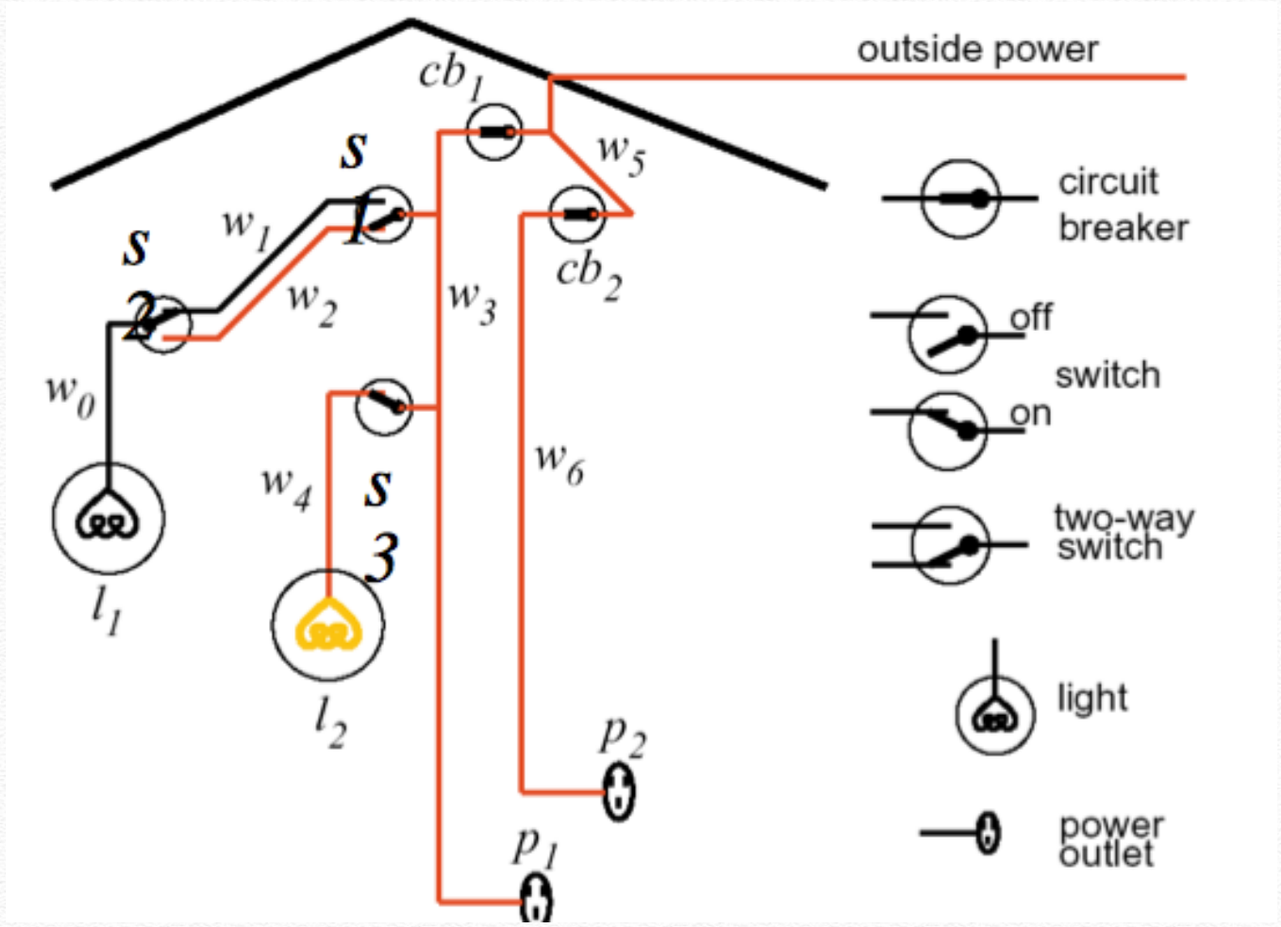


Write a procedure that proves that light l2 is illuminated and light l1 isn't.

The Wiring Domain

```
light(l1).
light(l2).
down(s1).
up(s2).
up(s3).
ok(l1).
ok(l2).
ok(cb1).
ok(cb2).
connected_to(l1, w0).
connected_to(w0, w1) :- up(s2).
connected_to(w0, w2) :- down(s2).
connected_to(w1, w3) :- up(s1).
connected_to(w2, w3) :- down(s1).
connected_to(l2, w4).
connected_to(w4, w3) :- up(s3).
connected_to(p1, w3).
connected_to(w3, w5) :- ok(cb1).
connected_to(p2, w6).
connected_to(w6, w5) :- ok(cb2).
connected_to(w5, outside).

illuminated(L) :-
light(L),ok(L),connection(L,outside).
connection(X,Y) :- connected_to(X,Y).
connection(X,Y) :-
connected_to(X,Z),connection(Z,Y).
```



Write a procedure that proves that light l2 is illuminated and light l1 isn't.

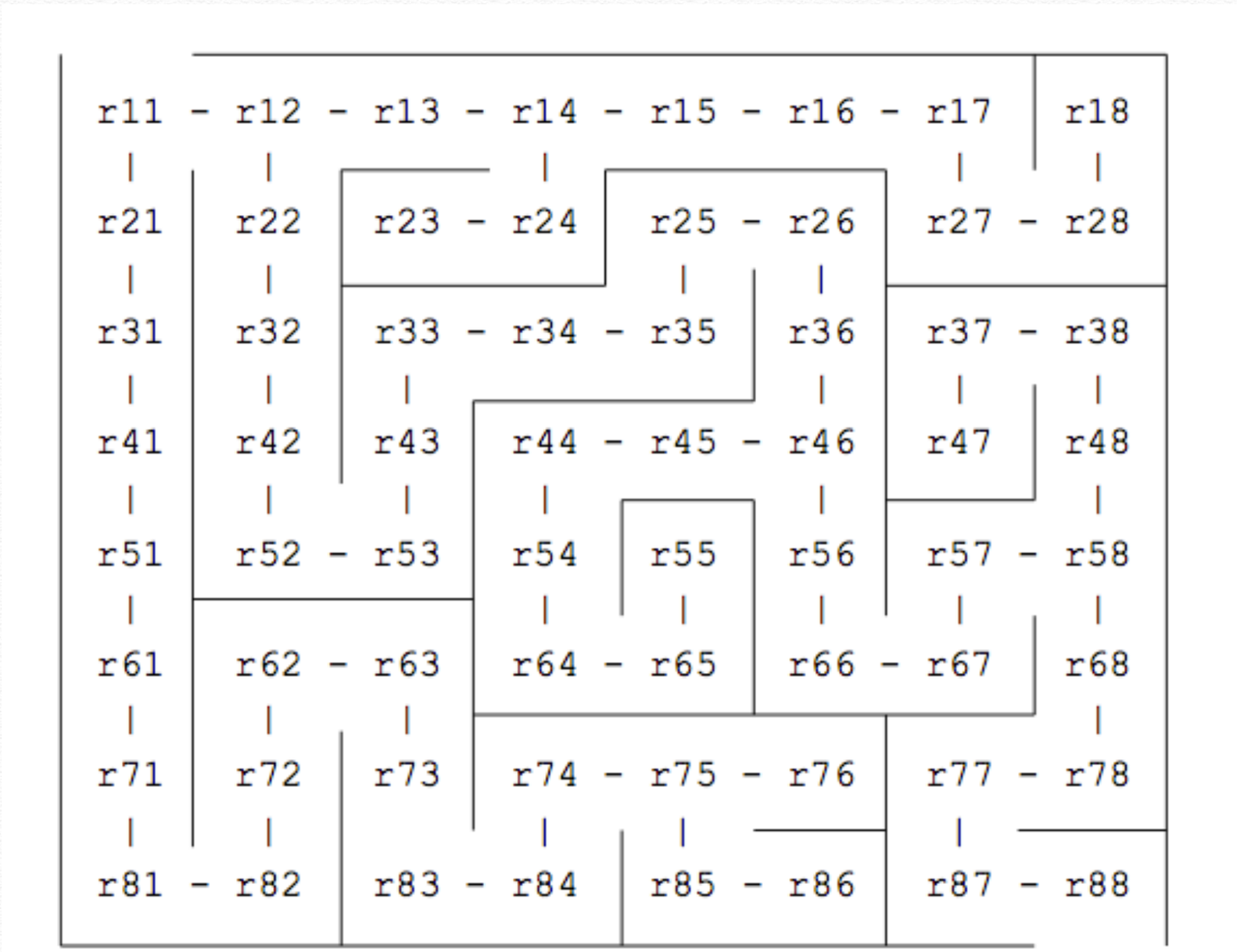
Recursive definition

```
illuminated(L) :- light(L),ok(L),
connection(L,outside).

connection(X,Y) :- connected_to(X,Y).

connection(X,Y) :- connected_to(X,Z),
connection(Z,Y).
```

The Maze


r11 - r12 - r13 - r14 - r15 - r16 - r17 r18
r21 r22 r23 - r24 r25 - r26 r27 - r28
r31 r32 r33 - r34 - r35 r36 r37 - r38
r41 r42 r43 r44 - r45 - r46 r47 r48
r51 r52 - r53 r54 r55 r56 r57 - r58
r61 r62 - r63 r64 - r65 r66 - r67 r68
r71 r72 r73 r74 - r75 - r76 r77 - r78
r81 - r82 r83 - r84 r85 - r86 r87 - r88

The Maze

How do we represent the facts of the maze?

The Maze

```
connects_to(r11,r12).
connects_to(r12,r13).
connects_to(r13,r14).
connects_to(r14,r15).
connects_to(r15,r16).
connects_to(r16,r17).
connects_to(r11,r21).
connects_to(r12,r22).
connects_to(r14,r24).
connects_to(r24,r23).
connects_to(r25,r26).
connects_to(r27,r28).
connects_to(r17,r27).
connects_to(r28,r18).
connects_to(r21,r31).
...
```

How do we represent the facts of the maze?

The Maze

```
connects_to(r11,r12).
connects_to(r12,r13).
connects_to(r13,r14).
connects_to(r14,r15).
connects_to(r15,r16).
connects_to(r16,r17).
connects_to(r11,r21).
connects_to(r12,r22).
connects_to(r14,r24).
connects_to(r24,r23).
connects_to(r25,r26).
connects_to(r27,r28).
connects_to(r17,r27).
connects_to(r28,r18).
connects_to(r21,r31).
...
```

How do we represent the facts of the maze?

How do we write a prolog procedure that can prove the existence of a path from one location in the maze to another?

The Maze

```
connects_to(r11,r12).
connects_to(r12,r13).
connects_to(r13,r14).
connects_to(r14,r15).
connects_to(r15,r16).
connects_to(r16,r17).
connects_to(r11,r21).
connects_to(r12,r22).
connects_to(r14,r24).
connects_to(r24,r23).
connects_to(r25,r26).
connects_to(r27,r28).
connects_to(r17,r27).
connects_to(r28,r18).
connects_to(r21,r31).
...

path(X,Y) :- connects_to(X,Y).
path(X,Y) :- connects_to(X,Z),path(Z,Y).
```

How do we represent the facts of the maze?

How do we write a prolog procedure that can prove the existence of a path from one location in the maze to another?

What about these alternatives?

```
path01(X,Y) :- connects_to(X,Y).
path01(X,Y) :- connects_to(X,Z),path01(Z,Y).

path02(X,Y) :- connects_to(X,Z),path02(Z,Y).
path02(X,Y) :- connects_to(X,Y).
```

```
path03(X,Y) :- connects_to(X,Y).
path03(X,Y) :- path03(Z,Y),connects_to(X,Z).

path04(X,Y) :- path04(Z,Y),connects_to(X,Z).
path04(X,Y) :- connects_to(X,Y).

path05(X,Y) :- connects_to(X,Y).
path05(X,Y) :- path05(X,Z),connects_to(Z,Y).

path06(X,Y) :- path06(X,Z),connects_to(Z,Y).
path06(X,Y) :- connects_to(X,Y).
```

The order of execution: not so nondeterminist

Because your Prolog interpreter isn't nondeterministic, it doesn't behave exactly like the Simple Abstract Interpreter.

When Prolog chooses a goal or conjunct from the resolvent to work on, **it always chooses the leftmost one**.

When Prolog chooses a rule or clause from the program, it **always searches from top to bottom**, selecting the first one whose head unifies with (matches) the previously-selected goal or conjunct. If that rule or clause leads to a dead end, Prolog **backtracks** to that decision point and then continues the search further down the list of rules or clauses. (See Chapter 6)

The order of execution

Consequently, when you construct your Prolog procedures (like path), the ordering of your rules in the program can have big impact on your program's performance.

Similarly, the ordering of goals or conjuncts in the bodies or right-hand sides of those rules can have big impacts too. (See Chapter 7.)

How much impact? And why? That's for you to explore when you do Assignment #1.

Assignment #1

Download `maze.pl` from Piazza. It contains all the facts for the maze shown previously. It also contains twelve different versions of the recursive procedure path.

Part A: Try each version of path. Does that version find a proof that a path from r11 to r88 exists, or does the stack overflow? Use SWI-Prolog's `time` predicate to see how much work Prolog did along the way. Record your observations for each version.

Part B: Now comment out one fact in maze.pl so that there is no path from r11 to r88. Try all twelve versions of path again and record your observations.

Assignment #1

Part C: Now answer these questions:

1) Why do some versions of path use very few inferences while others use many more and take so much longer, while still others overflow the stack before the query can be proven?

2) Why do some versions of the path procedure work in Part A and Part B, while others work only in Part A?

Assignment #1

When answering these questions, think in terms of how your Prolog interpreter really works, and how rule order and goal order in the each path procedure affects how its recursion works (or doesn't).

The results are due by next Thursday, September 24th, before the class begins (so no submissions after 2pm).

Collaboration Policy

The assignment, as said before, will be handed in individually. However, you may choose to discuss your ideas with only one other student from the class, but even then you are not to share or take away anything from the discussion other than what's in your brains.

If you choose to have a discussion partner, you should mutually credit each other in your submissions.

Questions

Next Class

- We will talk about Lists, List Recursion, and Unification.

- Time to finish reading Chapter 3 and 4.