- Project #2 - should be well underway....
- Practice exam questions on web page.

"Once you replace negative thoughts with positive ones, you'll start having positive results."

Willie Nelson, 2006 in "The Tao of Willie"

Plan

Since Midterm

- difference lists, definite clause grammars and natural language interfaces to databases
- computer algebra and calculus
- Triples are universal representations of relations, and are the basis for RDF, and knowledge graphs
- URIs/IRIs provide constants that have standard meanings
- Ontologies define the meaning of symbols
- You should know what the following mean: RDF, IRI, rdf:type, rdfs:subClassOf, rdfs:domain, rdfs:range
- Complete Knowledge Assumption, Negation as failure, unique names assumption

To do:

- Negation-as-failure (cont)
- Extra-logical predicates
- Proofs

Clark Normal Form

The Clark normal form of the clause

$$p(t_1, \ldots, t_k) \texttt{ :- } B.$$

is the clause

$$p(V_1, \ldots, V_k) \texttt{ :- } \exists W_1 \ldots \exists W_m \; V_1 = t_1, \; \ldots, \; V_k = t_k, \; B.$$

where

- $V_1, \ldots, V_k$ are k variables that did not appear in the original clause
- $W_1, \ldots, W_m$ are the original variables in the clause.
- When the clause is an atomic clause, B is *true*.
- Often can be simplified by replacing $\exists W \; V = W \wedge p(W)$ with $P(V)$.

Clark normal form

For the clauses

$$student(mary).$$

$$student(sam).$$

$$student(X) \;\texttt{:-}\; undergrad(X).$$

the Clark normal form is

$$student(V) \;\texttt{:-}\; V = mary.$$

$$student(V) \;\texttt{:-}\; V = sam.$$

$$student(V) \;\texttt{:-}\; \exists X \; V = X \wedge undergrad(X).$$

Clark's Completion

Suppose all of the clauses for p are put into Clark normal form, with the same set of introduced variables, giving

$$p(V_1, \dots, V_k) \text{ :- } B_1.$$

$$\vdots$$

$$p(V_1, \dots, V_k) \text{ :- } B_n.$$

which is equivalent to

$$p(V_1, \dots, V_k) \text{ :- } B_1 \vee \dots \vee B_n.$$

Clark's completion of predicate p is the equivalence

$$\forall V_1 \dots \forall V_k \ p(V_1, \dots, V_k) \leftrightarrow B_1 \vee \dots \vee B_n$$

If there are no clauses for p, the completion results in

$$\forall V_1 \dots \forall V_k \ p(V_1, \dots, V_k) \leftrightarrow \mathit{false}$$

Clark's completion of a knowledge base consists of the completion of every predicate symbol along the unique names assumption.

Clark normal form

For the clauses

$$student(mary).$$
$$student(sam).$$
$$student(X) \text{ :- } undergrad(X).$$

the Clark normal form is

$$student(V) \text{ :- } V = mary.$$
$$student(V) \text{ :- } V = sam.$$
$$student(V) \text{ :- } \exists X \; V = X \wedge undergrad(X).$$

which is equivalent to

$$student(V) \text{ :- } V = mary \vee V = sam \vee \exists X \; V = X \wedge undergrad(X).$$

The completion of the *student* predicate is

$$\forall V \; student(V) \leftrightarrow V = mary \vee V = sam \vee \exists X \; V = X \wedge undergrad(X).$$

Completion Example

Consider the recursive definition:

$$passed\_each([\,], St, MinPass).$$
$$passed\_each([C|R], St, MinPass) \;:\!-\; passed(St, C, MinPass),\; passed\_each(R, St, MinPass).$$

In Clark normal form, this can be written as

$$passed\_each(L, S, M) \;:\!-\; L = [\,].$$
$$passed\_each(L, S, M) \;:\!-\; \exists C\ \exists R\ L = [C|R],\ passed(S, C, M),\ passed\_each(R, S, M).$$

Here we renamed the variables as appropriate. Thus, Clark's completion of *passed_each* is

$$\forall L\ \forall S\ \forall M\ passed\_each(L, S, M) \leftrightarrow L = [\,] \vee \exists C\ \exists R\ L = [C|R],\ passed(S, C, M),\ passed\_each(R, S, M).$$

Clark's Completion of a KB

- Clark's completion of a knowledge base consists of the completion of every predicate.
- The completion of an *n*-ary predicate *p* with no clauses is $p(V_1, \ldots, V_n) \leftrightarrow false$.
- You can interpret negations in the body of clauses. `\+ a` means *a* is false under the complete knowledge assumption. `\+ a` is replaced by $\neg a$ in the completion. This is negation as failure.

Defining *empty_course*

Given database of:

- $course(C)$ that is true if C is a course
- $enrolled(S, C)$ that is true if student S is enrolled in course C.

Define $empty\_course(C)$ that is true if there are no students enrolled in course C.

- Using negation as failure, $empty\_course(C)$ can be defined by

$$empty\_course(C) \texttt{ :- } course(C), \quad \backslash + \ has\_enrollment(C).$$
$$has\_enrollment(C) \texttt{ :- } enrolled(S, C).$$

- The completion of this is:

$$\forall C \ empty\_course(C) \iff course(C), \neg has\_enrollment(C).$$
$$\forall C \ has\_enrollment(C) \iff \exists S \ enrolled(S, C).$$

Problem Cases

- p :- p.
- r :- \+ r.
- a :- \+ b.
  b :- \+ a.
- c :- \+ d.
  d :- c.
- It isn't clear what the semantics should be.
  Prolog goes into an infinite loop.
  Avoid cycles!

Problematic Cases

$p(X)$:- \+ $q(X)$
$q(X)$:- \+ $r(X)$
$r(a)$
?- $p(X)$.

- What is the answer?
- How can this be implemented?

Asserting and retracting clauses

New clauses can be added using

- `assertz(atom)` adds `atom` as the last clause.
  `atom` must be declared dynamic.
- `assertz((h :- b))` adds `h :- b` as the last clause
  (note double parenthases). `h` must be declared dynamic.
- `asserta` adds a clause as the first clause.

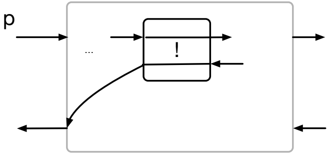
These are not undone by backtracking.

Example: count the number of times `counthis` is called:

```
:- dynamic countn/1.
countn(0).
countthis :-
    retract(countn(N)),
    N1 is N+1,
    assertz(countn(N1)).
```

cut, or commmit, written as !

- when called, exits
- when retried, fails the atom it is used in



Example: implementing negation as failure

```
mynot(A) :- call(A), !, fail.
mynot(A).
```

bagof, setof, findall

```
setof(t(Xs),Ys^foo(Xs,Ys,Zs), L)
```

where $t(Xs)$ is a term containing variables Xs.
Ys is a set of existential variables
Zs is the other variable in *foo*
is true when $L = \{t(Xs) \mid \exists Y \ foo(X, Y, Z)\} \neq \{\}$
there is an answer for each Z.
`bagof(t(Xs),Ys^foo(Xs,Ys,Zs), L)` returns a list not a set
Try from `cs312_2024`:

```
bagof(P, D^S^F^office_hour(P, D, S, F), Bag).
setof(P, D^S^F^office_hour(P, D, S, F), Bag).
bagof(P, S^F^office_hour(P, D, S, F), Bag).
bagof(P, office_hour(P, D, S, F), Bag).
bagof(s(P,S), F^office_hour(P, D, S, F), Bag).
```

`findall(t(Xs),foo(Xs,Ys,Zs), L)` like
`bagof(t(Xs),Ys^Zs^foo(Xs,Ys,Zs), L)`