Logic Programming

The term *logic programming* refers to a programming paradigm that
bodies the rules of *first-order logic* (also known as *predicate logic*
logic, you can express *facts* involving non-logical objects, and *relat*
relations involve facts, standard logical operations, variables, and (u
istential) quantifiers. Logic programming uses *rules of inference* to
a collection of facts and relations.

In the imperative and functional example programming languages w
so far, we wanted the result of applying (or "running") a program wi
In logic programming, we want to determine whether a particular
evaluates to *true* or *false*; or, more generally, to determine all of th
variables in a logic expression that make the expression true.

Prolog remains the dominant logic programming language i
though there exist variants that extend (or restrict) Prolo
ity and purpose: **Datalog** is one such variant. **AbcDatal**
at Harvard University, is a Java-based implementation of Da
use in these notes. AbcDatalog is covered by the BSD
`http://abcdatalog.seas.harvard.edu/license.tx`

Logic Programming

Our implementation of AbcDatalog resides in the `ABC` language s
the `Code` directory. The `README.txt` file in this subdirectory
information about where to find the AbcDatalog source files and de
plementation specifics.

The tokens of the `ABC` language are given here:

```
skip WHITESPACE '\s+'
skip COMMENT '%.*'
PRINT 'Print'
CLEAR 'Clear'
DOT '\.'
NOT 'not'
EQ '='
NE '!='
IF ':-'
QUERY '\?'
DEL '~'
COMMA ','
RPAREN '\)'
LPAREN '\('
US '_'
VAR '[A-Z_][_\w]*'
LIT '[a-z][_\w]*'
```

Logic Programming

Of these, the `PRINT` and `CLEAR` tokens are not part of the Harvard i
of AbcDatalog.

Here is the grammar specification for the `ABC` language, in PLCC fo

```
<program>:DFRQ        ::= <head> <dfrq>
<program>:Print       ::= PRINT
<program>:Clear       ::= CLEAR
<dfrq>:Delete         ::= DEL
<dfrq>:Fact           ::= DOT
<dfrq>:Rule           ::= IF <conjuncts> DOT
<dfrq>:Query          ::= QUERY
<head>                ::= <LIT> LPAREN <args> RPAREN
<args>                **= <vlw> +COMMA
<vlw>:Var             ::= <VAR>
<vlw>:Lit             ::= <LIT>
<vlw>:WC              ::= US # wildcard
<conjuncts>           **= <conjunct> +COMMA
<conjunct>:HeadConj   ::= <head>
<conjunct>:RelConj    ::= <VAR>lh <rel> <vlw>rh
<conjunct>:NotConj    ::= NOT <head>
<rel>:EQRel           ::= EQ
<rel>:NERel           ::= NE
```

Logic Programming

The <dfrq> nonterminal suggests its four variants:

- d stands for *delete*
- f stands for *fact*
- r stands for *rule*
- q stands for *query*

Here are some examples of *facts*, written in the ABC language:

```
planet(earth).
planet(mercury).
planet(jupiter).
larger(jupiter,earth).
larger(earth,mercury).
```

If we want to find all of the planets that the current systems knows about, w
following *query*:

```
planet(X)?
```

The system responds with the following result (the planets may appear in any ord

```
3 matches:
planet(earth)
planet(mercury)
planet(jupiter)
```

Logic Programming

Continuing the above example, the query '`larger(X,Y)?`' produ
ing result:

```
2 matches:
larger(jupiter, earth)
larger(earth, mercury)
```

Can we logically deduce that jupiter is larger than mercury?
is 'no', since there is no fact that says this: if we were to
'`larger(jupiter,mercury)?`', we would get no matches.

To make the `larger` relation transitive, we build a *rule* that says th
than Y, and if Y is larger than Z, then X is larger than Z. We do so as

```
larger(X,Z) :- larger(X,Y), larger(Y,Z).
```

The LHS of this rule (the part to the left of '`:-`') is called an *Atom*,
this rule is a sequence of *Premises* separated by commas. You can r
saying *the LHS is true if all of the Premises on the RHS are true*. T
of the possible substitutions of X, Y, and Z that make the rule true.

Logic Programming

Once we have included this rule, our "larger" query reports this (in s

```
3 matches:
larger(jupiter, earth)
larger(earth, mercury)
larger(jupiter, mercury)
```

Entering facts, rules, and queries can be done interactively at the '–

What if we wanted to include the planet saturn in our list of plane certainly add the fact

```
planet(saturn).
```

We know that saturn is larger than earth, so we might then add the fa

```
larger(saturn,earth).
```

But upon a query of the `larger` relation, we would see no `larg` tween saturn and jupiter. Try it!

The important thing is that the query engine cannot deduce someth explicitly given in facts or explicitly deduced from rules.

Logic Programming

A *binary relation* on a set A is defined to be a subset of the cartesian In the `ABC` language, we can represent a binary relation as a set example, consider the set `A = {a,b,c,d}`. Define a binary relat on `A` in the `ABC` language as follows, which you would enter as facts `Rep` program:

```
rel(a,b).
rel(a,d).
rel(d,a).
rel(c,c).
```

If you were to enter the query `rel(X,Y)?`, you would get a respo sisting of exactly the same four facts (in some order).

We use the term *reflexive* to describe a binary relation `rel` on `A` havi that if `x` is any element in the set `A`, then `rel(x,x)` is true (*i.e.*, is a see that the `rel` binary relation is definitely not reflexive, but we ca `rel` to make it reflexive, as follows:

```
rel(X,X) :- rel(X,_).
rel(X,X) :- rel(_,X).
```

We call the resulting modification to the relation `rel` its *reflexive* that we are using the '_' variable as a place holder that can match an in the `rel` relation.

Logic Programming

We similarly define a relation `rel` on `A` to be *symmetric* if `rel`
`rel(y,x)`. Just like we did for a reflexive closure, we can create
closure of the relation `rel` in the `ABC` language as follows:

```
rel(X,Y) :- rel(Y,X).
```

Finally, we define a relation `rel` on `A` to be *transitive* if `re`
`rel(y,z)` implies `rel(x,z)`. We can create the *transitive clo*
lation `rel` in the `ABC` language as follows:

```
rel(X,Z) :- rel(X,Y), rel(Y,Z).
```

Notice that this is exactly what we did with the `larger` relation on

One can create any one of these closures individually, or all of them,
your resulting relation is reflexive, symmetric, and transitive, it is ca
lence relation, an important construct in first-order logic and set the

(Notice that the rule defining the reflexive closure assumes that ev
set `A` is related to *something*.)

Logic Programming Puzzles

Consider the Fox/Goose/Corn river crossing puzzle (also k
Wolf/Goat/Cabbage puzzle):

A farmer, a fox, a goose, and a bag of corn are on one side of a riv
has a boat that can hold *at most one* other "passenger" – either the
the bag of corn, or nothing at all. The problem is to transport ev
other side of the river using the boat. One constraint is that the farm
the fox and the goose together on one side with the farmer on th
otherwise the fox, unsupervised, will eat the goose. Another constr
farmer cannot leave the goose and the bag of corn together on one
farmer on the other side – otherwise the goose, unsupervised, will e
final constraint is that the farmer and the boat will always be on th
the river together. *Can the farmer transport everything from one side
the other using the boat, while maintaining the constraints?*

We will model this problem by considering all possible *states*, wh
string of the form `sbfgc`. Here, `s` is just the letter 's', which stand
`bfgc` positions will be digits `0` or `1`, referring to which side of the
(and the farmer), the fox, the goose, and the corn are on, respectivel
the digit `0` to mean the starting side and the digit `1` to mean the endi
problem is to move from state `s0000` to state `s1111` by legal river

Notice that each river crossing will toggle the `b` position. (The wor
switching from a `0` to a `1` or *vice versa*.) At most one of the other
will also get toggled, depending on which passenger (or none) the f
the boat.

Some of the states do not satisfy the constraints. For example, `s01`
sible, since otherwise the boat (and the farmer) will be on the starti
fox and goose will be on the ending side. (the `?` just means it doesn'
are all of the impossible states:

```
s011? -- fox and goose unsupervised
s100? -- fox and goose unsupervised
s0?11 -- goose and corn unsupervised
s1?00 -- goose and corn unsupervised
```

Notice that the total number of states (possible and impossible) is
number of impossible states is 6, so there are exactly 10 possible stat
Five of these have the boat (and the farmer) on the start side.

A river *crossing* will consist of a transition from one state to anot
the `b` position (the farmer takes the boat from one side to the other,
passenger) and toggling at most one of the `fgc` positions (the passe

Logic Programming Puzzles

For example, in the first river crossing (starting from state `s0000`), t
the boat and the goose from the starting side to the ending side. W
in the `ABC` language by the following fact:

```
cross(s0000, g, s1010). % carry the goose i
```

You can see that this is the *only* legal crossing starting from the initia
We use the letter `x` to mean a river crossing with no "passenger" in t

Observe that for every legal crossing of the form

```
cross(s0..., ?, s1...).
```

(where the `?` can be any of `fgcx`) with the boat (and farmer) going fr
side to the ending side of the river, there is a corresponding legal
form

```
cross(s1..., ?, s0...).
```

and *vice versa*. This means that we need only consider the legal c
the boat is on the starting side of the river, and use the following r
the others:

```
cross(X, P, Y) :- cross(Y, P, X).
```

Logic Programming Puzzles

Here are all of the legal crossings where the boat is on the starting si

```
cross(s0000,g,s1010).
cross(s0001,g,s1011).
cross(s0001,f,s1101).
cross(s0010,x,s1010). % no passenger
cross(s0010,c,s1011).
cross(s0010,f,s1110).
cross(s0100,c,s1101).
cross(s0100,g,s1110).
cross(s0101,x,s1101). % no passenger
cross(s0101,g,s1111).
```

There are two ways to think of a solution to the fox/goose/corn puzz

- determine *if* there is a solution
- determine an explicit solution, giving the actual crossings

The first is easier, at least in the `ABC` language.

Logic Programming Puzzles

The idea is to define what it means for there to be a *path* from one s
First, we consider every crossing

```
cross(X,P,Y)
```

as a path from `X` to `Y`:

```
path(X,Y) :- cross(X,_,Y).
```

Then we develop the transitive closure of the relation `path`

```
path(X,Z) :- path(X,Y), cross(Y,_,Z).
```

to get all paths. Notice that the second clause in the RHS just nee
premise instead of the more general `path(Y,Z)` (why?).

The query

```
path(s0000, s1111)?
```

will produce a result if and only if there is a path from state `s000`
the starting side) to state `s1111` (everyone on the ending side). You
this is the case!

Logic Programming Puzzles

Here is the complete ABC program, with the solution query:

```
cross(s0000,g,s1010).
cross(s0001,g,s1011).
cross(s0001,f,s1101).
cross(s0010,x,s1010). % no passenger
cross(s0010,c,s1011).
cross(s0010,f,s1110).
cross(s0100,c,s1101).
cross(s0100,g,s1110).
cross(s0101,x,s1101). % no passenger
cross(s0101,g,s1111).
cross(X,P,Y) :- cross(Y,P,X).
path(X,Y) :- cross(X,_,Y).
path(X,Z) :- path(X,Y), cross(Y,_,Z).
path(s0000,s1111)?
```

When you run this program, you will see the following result, showi
a solution!

```
1 match
path(s0000, s1111)
```

Logic Programming Puzzles

However, this doesn't show give an explicit soltuion to the puzzle.
the ABC language doesn't have a way to build the solution using s
structure. So we do it by brute force!

First, observe that a minimal length solution path (with fewest cr
have two crossings that repeat a start side state, otherwise you wou
previous state. Since there are exactly 5 start side states of the fo
can be used for a crossing (see Slide 7.14), a minimum-length path c
nine crossings with different start side states. (Remember that a solu
an odd number of crossings because a solution will always have
ending side.)

Define a premise c1 for a single crossing:

```
c1(X,P1,Z) :- cross(X,P1,Z). % P1 is the pa
```

Then define a premise c2 for two consecutive crossings:

```
c2(X,P1,P2,Z) :- c1(X,P1,Y), cross(Y,P2,Z).
```

Logic Programming Puzzles

Continue in this way to define three and more consecutive crossings

```
c3(X,P1,P2,P3,Z) :- c2(X,P1,P2,Y), cross(Y,P3,Z).
c4(X,P1,P2,P3,P4,Z) :- c3(X,P1,P2,P3,Y), cross(Y,P4,
c5(X,P1,P2,P3,P4,P5,Z) :- c4(X,P1,P2,P3,P4,Y), cros
c6(X,P1,P2,P3,P4,P5,P6,Z) :- c5(X,P1,P2,P3,P4,P5,Y),
c7(X,P1,P2,P3,P4,P5,P6,P7,Z) :- c6(X,P1,P2,P3,P4,P5,
                                  cross(Y,P7,Z).
```

We can determine (by brute force) that less than seven crossings (number) will not get everything to the ending side. (For example, th

```
c5(s0000,P1,P2,P3,P4,P5,s1111)?
```

gives no results.) However, `c7` has two matches:

```
c7(s0000, g, x, f, g, c, x, g, s1111)
c7(s0000, g, x, c, g, f, x, g, s1111)
```

Reading the letters from left to right gives the list of passengers (passenger, etc.) of a solution path from the start to the end state.

Observe that both solution paths have the goose as a passenger thre fox only once.

Logic Programming Puzzles

Consider modifying the puzzle with the additional constraint that has at least one passenger. Is there a solution to this variant? To simply remove the `cross` premises with an `x` in the passenger slot (and see if ‘`path(s0000, s1111)?`’ returns a result.

Another variant would have a boat with two places for passengers, in this case would be trivial to see “by inspection”, requiring no pro

The Prolog language, upon which Datalog (and AbcDatalog) is bas richer collection of data structures and operations that lends itself larger set of problems, including those that require the use of lis operations.