Study Problems for Prolog for the Final Exam

CS 440: Programming Languages and Translators, Spring 2020

4/29: solution added

Chapter and section numbers below refer to the Learn Prolog Now book.

Chapter 1: Facts, Rules, and Queries

1.1. [Like/Extends Ex 1.1] What's are differences between `Big_kahuna_burger`, `'Big kahuna burger'`, `'Big_kahuna_burger'`, `big kahuna burger`, `'big kahuna burger'`, `big(kahuna, burger)`, `17`, `17+0`, `'17'`, and `'17+0'` ? (I.e., atom, variable, number, complex term/structure, none of the above?)

1.2. [Like/Extends Ex 1.3] Is the item below a fact or a rule? If a rule, what is its head? What are its goals? What are the predicates contained in the item?

```
animal(X) :- dog(X), cat(X), platypus(X).
```

1.3. How are the two items below different? Are they both legal?

```
grandparent(X) = Y :- parent(X) = Z, parent(Z) = Y.
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
```

1.4. What is a knowledge base? When is it used?

Chapter 2: Unification

2.1 What's the "occurs check"? Does it apply to `X = f(X)`, and if so, does pass the check? What about `x = f(x)` ? `f(X) = f(f(X))` ? `X = X+0` ? `X = 'X+0'` ?

2.2 Study the `vertical(line(point(X,Y), point(U,V)))` example in section 2.1.

Is `vertical(line(point(X, _), point(X, _))).` represent a true fact?

What about vertical(line(point(_, Y), point(_, Y))).?

2.3 [see section 2.2] Say we have facts `f(a)`, `f(b)`, `g(a)`, `g(b)`, `g(c)`, `h(a)` and `h(c)`. and rules `p(X) :- f(X), g(X)` and `p(X) :- g(X), h(X)`. If we ask the query `p(X)` and repeat until no more answers are produced, what sequence of results do we get?

2.4 [See section 2.4] What does infix `\=` mean? Why are the queries `a \= A` and `f(b) \= f(B)` false?

Chapter 3: Recursion

3.1 [See section 3.1] What are the similarities and differences between the three definitions below of "descendant"? Assume some facts: `child(riley, finley)`, `child(finley, skyler)`. What happens if you ask for descendants of `riley`?


```
d1(X,Y) :- child(X,Y).
d1(X,Y) :- child(X,Z), d1(Z,Y).

d2(X,X).
d2(X,Y) :- child(X,Z), d2(Z,Y).

d3(X,Y) :- child(X,Y).
d3(X,Y) :- d3(X,Z), d3(Z,Y).
```

3.2 What happens if you swap the two rules for each `d` predicate in the previous problem?

3.3 [Like section 3.4] Say we have some link facts, e.g. `link(1,2)`, `link(2,3)`, `link(3,1)` and so on. Complete the definitions below of predicates `path(F,T)` and `has_path(F,T,P)` where we mean there's a path from `F` to `T` (with `has_path` including the path `P`).

`path(F,T) :- link(F,T).`

`path(F,T) :- link(F,N),` ??? <-- finish this definition (replace ??? with code)

`has_path(F,T,[(F,T)]) :- link(F,T).`

`has_path(F,T, ???) :- link(F,N),` ??? <-- finish this definition

Chapter 4: Lists

4.1 What are the differences between `X1 = 17`, `X2 = [17]`, `X3 = [17|[]]`, `X4 = [[17]|[]]`, `X5 =[17|[18]]`, `X6 = [[17]|[18]]`, and `X7 = [17|[[18]]]`?

4.2 Is the first rule below really necessary? Why or why not?

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

4.3 Write a definition for a predicate `prefix(X,Y)` (short for "initial segment") that is true iff list `X` is an initial segment of list `Y`. E.g. `prefix([1,2], [1,2,3])` is true. So are `prefix([], [])`, `prefix([], [1,2])`, and `prefix([1,2],[1,2])`.

4.4 Write a definition for a predicate `stutter(X)` that is true iff list `X` has two occurrences of the same value next to each other. E.g., `stutter([1,2,2,3])` or `stutter([1,1])`. If X has < 2 members, stutter should be false.

Chapter 5: Arithmetic

5.1 What are the differences between `X = 2+2`, `X is 2+2`, `2+2` is `2+2`, and `2+2` is `X`?

5.2 Say we declare `len([],0).` What are the differences between declaring the recursive case as

`len([_| T],N+1) :- len(T,N)`

`len([_| T],M) :- len(T,N),M is N+1`

`len([_| T],M) :- M is N+1, len(T,N)` ?

5.3 Do <, >, >=, and =< (means ≤) behave the way we'd expect? E.g., `2+3 < 6*5` ? What about `X > Y-2`?

5.4 How do = and =:= differ on arithmetic terms? How about \= and =\= ? Give some examples.

Chapter 6: More Lists [Just Section 1: Append]

6.1 The `append` predicate takes three arguments, e.g., `append([1,2],[3,4],[1,2,3,4])`. Which of the following queries succeed, and what values do the variables get? If there multiple proofs, describe their results too.

- `append([1,2],[3,4],Z)`
- `append(X,[3,4],[1,2,3,4])`
- `append([1,2],Y,[1,2,3,4])`
- `append(V,W,[1,2,3,4])`

6.2 `append(P,Q,R)` has an infinite number of solutions. The first is `P = []`, `Q = R` (i.e., `P` is the empty list, `Q` is any list, and `R` is `Q`. If we write `_1` for a system-generated variable, then the second solution is `P = [_1]`, `R = [_1| Q]`.

a. Give an English description of what the second solution is.

b. Repeat, with the next solution, `P = [_1,_2]`, `R = [_1, _2| Q]`.

c. Why do we need names like `_1` and `_2`? Why can't we just write `P = [_]`, `R = [_|Q]` as a solution?

6.3 Write the definition of a predicate `appendAll(List1,List2)` where `List1` is a list of lists and `List2` is the result of concatenating all the lists in `List1`. E.g., `appendAll([ [1,2],[3,4],[[5]] ], [1,2,3,4,[5]])`.

Chapter 7: Definite Clause Grammars (just sections 7.1 and 7.2)

7.1 Consider the CFG $S \to A\ B$, $A \to$ `aa`, $B \to$ `bb`

a. What is the translation of this grammar into Prolog, using `append`?

b. How do we represent the sentence `aa bb` ?

c. How do we represent the notion $S \to^*$ `aa bb`?

d. How do we represent the notion $S \to A\ B \to$ `aa bb`?

e. What problem(s) does the representation of CFGs using `append` have?

7.2 Repeat parts (a) – (d) of the previous problem, this time using the difference list representation, in Prolog.

7.3 Repeat parts (a) – (d) of problem 7.1, this time using the definite clause grammar supported by Prolog.

7.4 If we want to introduce a recursive rule, say $A \rightarrow B\ A$, what do we need to do?

Chapter 10: Cuts and Negation (sections 10.1 and 10.2 on Cuts)

10.1 How do we write a cut? What does a cut do? Why do we use them? Why does using them make code more procedural?

10.2 What's the difference between a green cut and a red cut?

10.3 Give an example of code that uses a green cut.

10.4 Give an example of code that uses a red cut.

Solutions to Study Problems for Prolog for the Final Exam

CS 440: Programming Languages and Translators, Spring 2020

1.1. `Big_kahuna_burger` is a variable, `'Big kahuna burger'`, `'Big_kahuna_burger'`, and `'big kahuna burger'` are atoms, `big(kahuna, burger)` is a complex term a.k.a. structure, `17` is a number, `17+0` is a complex term, `'17'` is an atom (and not the same as `17`), `'17+0'` is an atom (and not the same as `17+0`). `big kahuna burger` is none of the above because of the embedded spaces.

1.2. (fact? rule?)

```
animal(X) :- dog(X), cat(X), X = pat_platypus.
```

is a rule with head `animal(X)` and goals `dog(X)`, `cat(X)`, and `X = pat_platypus`. There are three predicates: `animal`, `dog`, and `cat`.

1.3. (grandparent syntax)

```
grandparent(X) = Y :- parent(X) = Z, parent(Z) = Y. — is illegal.
grandparent(X,Y) :- parent(X,Z), parent(Z,Y). — is legal.
```

1.4 A knowledge base is a collection of facts and rules. Prolog refers to it when trying to prove a query.

2.1 The occurs check is a test that can be done during unification; it looks basically for recursive uses (in a complex term) of a variable: `X = X` passes; `X = f(X)` fails. `x = f(x)` doesn't involve variables (it's just `false`). `f(X) = f(f(X))` fails the test, so does `X = X+0`. Trick question: `X = 'X+0'` instantiates/unifies variable `X` with the atom `'X+0'`. It would be similar to `X = eks_plus_zero`.

2.2. vertical(line(point(X, _), point(X, _))). says that a line is vertical if its two endpoints have the same X coordinate, so it's reasonable. If we replace point(X, _) with point(_, Y), that's a horizontal line.

2.3 X = a, b, and a. The first comes from f(a), g(a), the second from f(b),g(b), and the third from g(a),h(a).

2.4 Infix \= means "does not unify with". a \= A is false because a = A is true. f(b) \= f(B) is false because b = B so f(b) = f(B).

3.1 d1(riley, Y) finds Y = finley and Y = skylar (and then false). d2(riley,Y) finds Y = riley, Y = finley, and Y = skylar (and then false). (I.e., finley is a descendant of finley.) d3(riley, Y) finds Y = finley and Y = skylar and then goes into infinite recursion.

3.2 d1 and d2 find that riley's descendants are skyler and then finley. d3(riley,Y) goes immediately into an infinite loop.

3.3 In both definitions, N is a neighbor node (connected by a link).

```
path(F,T) :- link(F,N), path(N,T).
has_path(F,T,[(F,N)|NTpath]) :- link(F,N), has_path(N,T,NTpath).
```

4.1 `X1` = `17`, `X2` and `X3` = `[17]`, `X4` = `[[17]]`, `X5` = `[17, 18]`, `X6` = `[[17], 18]`, and `X7` = `[17, [18]]`.

4.2 Adding or removing `member(_, []) :- false` doesn't make a difference in that the empty list will have no members. If it's removed, Prolog figures out it's false by failing to unify with the other two rules.

4.3 (initial segment predicate)

```
prefix([], _).
prefix([H|T1], [H|T2]) :- prefix(T1,T2)
```

4.4 (stutter)

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

5.1 (Difference between = and `is` with arithmetic terms)

`X` = `2+2` instantiates/unifies `X` with `2+2`.

- If X was uninstantiated, it gets instantiated (i.e. bound) to the term 2+2
- If X was instantiated, we try to unify its value with the term 2+2 (succeeding iff the value of X is also the term 2+2)
- X is 2+2 evaluates 2+2 and instantiates/unifies X with 4
- 2+2 is 2+2 fails because it evaluates the rhs 2+2, gets 4 and then fails to unify (the term) 2+2 and 4. Similarly, 2+2 is 4 fails.
- 2+2 is X fails if X is uninstantiated; if X is instantiated to some term then we fail by the same reasoning as for 2+2 is 4.

5.2 len([_| T], N+1) :- len(T, N) makes the calculated length a term: 0, 0+1, 0+1+1, etc.

len([_| T], M) :- len(T, N), M is N+1 instantiates/unifies M and a number: 0, 1, 2, etc. It calculates the length of the tail as a number N, calculates N+1, and instantiates/unifies M and the resulting number.

len([_| T], M) :- M is N+1, len(T, N) fails, either because N and possibly M are uninstantiated. (Note len([], 0) succeeds.)

5.3 Yes, <, >, >=, and =< evaluate both sides and compare the resulting numbers. `X > Y-2` fails if either `X` or `Y` is uninstantiated; if both are instantiated, then the expected arithmetic and comparison gets done. E.g., `X = 8, Y = 3, X > Y-2.` succeeds

5.4 The = operator tries to unify its operands; it doesn't try to evaluate arithmetic terms. The `=:=` operator takes two arithmetic terms and evaluates them and checks for equality of the results. `\=` and `=\=` are the negations of = and `=:=`. Examples: `2+2 =:= 3+1`, `2+2 \= 3+1`; `2+8 =\= 4-3`; `2+8 \= 4-3`.

6.1 The first three queries have one solution each: `Z = [1,2,3,4]`, `X = [1,2]`, and `Y = [3,4]`. The `append(V,W,[1,2,3,4])` query has five solutions, one for each different way we can append two lists and get `[1,2,3,4]`: (1) `V = []`, `W = [1,2,3,4]`; (2) `V = [1]`, `W = [2,3,4]`; (3) `V = [1,2]`, `W = [3,4]`; (4) `V = [1,2,3]`, `W = [4]`; and (5) `V = [1,2,3,4]`, `W = []`.

6.2 For `append(P,Q,R)`:

a. The solution `P = [_1]`, `R = [_1|Q]` means that `P` is a list of any one term, `Q` is any list, and `R` is `Q` with the member of `P` prepended ("consed") to its front.

b. The solution `P = [_1, _2]`, `R = [_1, _2|Q]` says that `P` is a list of any two values, `Q` is any list, and `R` is `Q` with the two values of `P` prepended to it.

c. We can't use `P = [_]`, `R = [_|Q]` as a solution because the two uses of _ stand for different unnamed variables. We wrote `P = [_1]`, `R = [_1|Q]` because we do need some otherwise-unnamed values, but they have to be the same on the left and right sides of the equation.

6.3 (Append all the members of a list of list)

```
appendAll([],[]).
appendAll([List1|Lists2_n],Result)
    :- appendAll(Lists2_n,Result2_n),append(List1,Result2_n,Result).
```

7.1 (Context-Free Grammar $S \rightarrow A\ B$, $A \rightarrow$ aa, $B \rightarrow$ bb)

a. The Prolog translation (using `append`) is

```
a([aa]).
b([bb]).
s(Sentence) :- a(Part1),b(Part2),append(Part1,Part2, Sentence).
```

b. The sentence `aa bb` is represented by `[aa,bb]`.

c. $S \rightarrow^*$ `aa bb` is represented as `s([aa,bb])`.

d. $S \rightarrow A\ B \rightarrow^*$ `aa bb` is represented as `s([aa,bb]) :- a([aa]),b([bb]),append([aa], [bb], [aa,bb]).`

e. Parsing for a CFG using `append` can involve a lot of appending, which can sow down parsing. Plus, the append of *N* lists is a bit awkward to write out unless you use something like `appendAll` (see problem 6.3).

7.2 (Representation of CFG using difference lists)

a. The Prolog translation using difference lists is

```
a[[aa|R],R].
b[[bb|R],R].
s(Sentence, Rem2) :- a(Sentence, Rem1), b(Rem1, Rem2).
```

b. `aa bb` is still represented by `[aa, bb]`.

c. $S \to^*$ `aa bb` is represented by `s([aa, bb], [])`. (Or `s([aa, bb|Rem], Rem)`.)

d. $S \to A\ B \to^*$ `aa bb` is represented by `s([aa, bb], []) :- a([aa, bb], [bb]), b([bb], [])`.

7.3 (Representation of CFG using definite clause grammars)

a. The Prolog translation using DCG syntax is

```
a --> [aa].
b --> [bb].
s --> a, b.
```

b. `aa bb` is still represented by `[aa, bb]`.

c. $S \to^*$ `aa bb` is still `s([aa, bb], [])`. (Or `s([aa, bb|Rem], Rem)`.)

d. $S \to A\ B \to^*$ `aa bb` is still `s([aa, bb], []) :- a([aa, bb], [bb]), b([bb], [])`.

7.4 We would want the base case `A --> [aa]` before the recursive case `A --> B A` to avoid infinite recursion.

10.1 A cut is written as the term `!`. When we encounter `!` as a goal to prove (i.e., when going left-to-right through the goals), it behaves like `true`. When we encounter ! as a goal to backtrack through (i.e., when going right-to-left), it behaves like `false` (and eliminates backtracking for this rule instance). We use them to cut down on backtracking, generally to make code faster by removing useless lines of reasoning. Using ! makes code more procedural because code after it (in a backtracking sense) doesn't get to run.

10.2 A green cut doesn't affect the set of results of a program, just its efficiency. A red cut does affect the set of results of a program.

10.3 The green cut version of the `max` predicate (maximum of two values) was

```
maxg(X, Y, Max) :- X =< Y, !, Max = Y.
maxg(X, Y, Max) :- X > Y, !, Max = X.
```

The cuts are green in that removing them doesn't change what value we calculate for `Max`. The cut in the first rule ensures that when $X \le Y$, we won't bother trying to prove $X > Y$. (The cut in the second rule doesn't make a difference.)

10.4 The red cut version of the max predicate was

```
maxr(X, Y, Max) :- X =< Y, !, Max = Y.
maxr(X, Y, X).
```

The cut is red because removing it lets us prove `maxr(X, Y, X)` without requiring $X > Y$. Inserting the cut makes the code faster than the green cut version because (with the cut) we only get to the second rule if $X > Y$, so there's no point in testing for it.

