

COMP 410
Fall 2019
Midterm Practice Exam #2

Unification without Lists

Consider each of the following unification attempts. If the unification succeeds, record any values any variables take. If the unification fails, say so.

1.) $\text{foo}(1, X) = \text{foo}(Y, 2)$

$X = 2, Y = 1$

2.) $\text{foo}(1, X) = \text{foo}(X, 2)$

false

3.) $\text{foo}(1, _) = \text{foo}(X, 2)$

$X = 1$

4.) $\text{foo}(1, _) = \text{foo}(1, _)$

true

5.) $\text{foo}(1, 2, \text{bar}) = \text{foo}(X, _, _, _)$

false

6.) $\text{foo}(\text{bar}(\text{baz}), X) = \text{foo}(Y, Z), Y = \text{bar}(A)$

$X = Z, Y = \text{bar}(\text{baz}), A = \text{baz}$

Unification with Lists

Consider each of the following unification attempts involving lists. If the unification succeeds, record any values any variables take. If the unification fails, say so.

$$7.) [1, 2, _] = [A, B, C|D]$$

$$A = 1, B = 2, D = []$$

$$8.) A = [1, 2|B], B = [4]$$

$$A = [1, 2, 4], B = [4]$$

$$9.) [[A|B], C] = [[1, 2]|D]$$

$$A = 1, B = [2], D = [C]$$

$$10.) X = [A|[2]]$$

$$X = [A, 2]$$

$$11.) [A, [B, [C|D]]] = [1, [2, [3, 4]]]$$

$$A = 1, B = 2, C = 3, D = [4]$$

Consider the following inductive list definition, which makes use of Prolog atoms and structures:

$$e \in ListElement$$

$$l \in List ::= cons(e, l) \mid nil$$

Now consider the following unifications, using Prolog lists. Rewrite these unifications using the above definition.

12.) $X = [1, 2, 3]$

$$X = cons(1, cons(2, cons(3, nil)))$$

13.) $X = [Y|Z]$

$$X = cons(Y, Z)$$

14.) $X = [A|[2]]$

$$X = cons(A, cons(2, nil))$$

15.) $X = [1, [2, [3]]]$

$$X = cons(1, cons(cons(2, cons(cons(3, nil), nil)), nil))$$

More Recursion

16.) Consider the following mathematical definition of a recursive function:

$$f_n = \begin{cases} 2 & \text{if } n = 0 \\ 3 & \text{if } n = 1 \\ (3 \times f_{n-1}) + (4 \times f_{n-2}) & \text{otherwise} \end{cases}$$

Write an equivalent definition in Prolog.

```
f(0, 2).  
f(1, 3).  
f(N, Result) :-  
    N > 1,  
    MinOne is N - 1,  
    MinTwo is N - 2,  
    f(MinOne, T1),  
    f(MinTwo, T2),  
    Result is (3 * T1) + (4 * T2).
```

17.) Write a procedure named `evensBetween`, which will nondeterministically produce all the even numbers within an inclusive range. As a hint, a number `N` is even if and only if the clause `0 is mod(N, 2)` is true. An example query is below:

```
?- evensBetween(1, 4, Even).  
Even = 2 ;  
Even = 4.  
  
evensBetween(Min, Max, Min) :-  
    Min =< Max,  
    0 is mod(Min, 2).  
evensBetween(Min, Max, Result) :-  
    Min < Max,  
    NewMin is Min + 1,  
    evensBetween(NewMin, Max, Result).
```

18.) Consider the following code:

```
proc([], 0).  
proc([_|A], B) :-  
    proc(A, C),  
    B is C + 1.
```

18.a) In your own words, what does this procedure compute?

The length of a given list.

18.b) This procedure is not very efficient when it comes to memory. Why is it inefficient?

It uses $O(N)$ stack space since it is not tail-recursive.

18.c) Rewrite this procedure to be more efficient with memory. You may introduce a helper procedure if desired.

```
proc(List, Len) :-  
    proc(List, 0, Len).  
  
proc([], Accum, Accum).  
proc([_|T], Accum, Len) :-  
    NewAccum is Accum + 1,  
    proc(T, NewAccum, Len).
```

19.) Define a procedure named `isPrime` which will determine if a given input number is prime. You may introduce any helpers you wish. Example queries follow:

```
?- isPrime(1).  
true .  
?- isPrime(2).  
true .  
?- isPrime(3).  
true .  
?- isPrime(4).  
false.
```

As a hint, the following Java-like code:

```
int x = y % z;
```

...is equivalent to the following Prolog code:

```
X is mod(Y, Z)
```

```
isPrime(Num) :-  
    StartNum is Num - 1,  
    isPrime(Num, StartNum).  
  
isPrime(1, 0).  
isPrime(_, 1).  
isPrime(Num, CurNum) :-  
    NonZero is mod(Num, CurNum),  
    NonZero \== 0,  
    NewNum is CurNum - 1,  
    isPrime(Num, NewNum).
```

Test Case Generation

20.) Consider the following grammar-based definition of simplistic SQL queries:

$$c \in \textit{ColumnName} \quad t \in \textit{TableName}$$
$$q \in \textit{SQLQuery} ::= \textit{select } c \textit{ from } t;$$

20.a) Assume the only possible columns are named `c1` and `c2`, and the only possible tables are named `t1` and `t2`. Write a generator of valid SQL query ASTs. An example of a valid AST is `select(c1, t1)`. Do not simply hardcode all possible ASTs.

```
columnName(c1).
columnName(c2).

tableName(t1).
tableName(t2).

sql(select(C, T)) :-
    columnName(C),
    tableName(T).
```

20.b) Bounds or related mechanisms are not necessary for this problem, at least as described. Why?

From the description, there are a reasonably finite number of possible ASTs. (Another possible answer) from the implementation, there is no recursion, which would potentially allow us to “spam” the same rule indefinitely.

20.c) Name a change to this problem which would necessitate adding a bound or a related mechanism, and explain why such a change would add this necessity.

Add a column or table name generator. Another answer is to add support for `while` clauses, which can be chained arbitrarily long with Boolean operators like `AND`. In both cases, these features make the space infinite, requiring us to inject failure somewhere to prevent us from producing repetitive-looking ASTs.