COMP 410
Fall 2023
Midterm Practice Exam #2 Solutions

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.

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

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

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

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

3.) `[[A|B], C] = [[1, 2]|D]`

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

4.) `X = [A|[2]]`

`X = [A, 2]`

5.) `[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:

$$e \in \mathit{ListElement}$$

$$\ell \in \mathit{List} ::= \texttt{cons}(e, \ell) \mid \texttt{nil}$$

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

6.) `X = [1, 2, 3]`

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

7.) `X = [Y|Z]`

`X = cons(Y, Z)`

8.) `X = [A|[2]]`

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

9.) `X = [1, [2, [3]]]`

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

More Recursion

10.) Write a procedure named `allEqual` which will succeed if all list elements are equal to each other according to unification (=). You may introduce any helpers you wish. Example calls are below:

```
?- allEqual([]).
true.
?- allEqual([1, 1, 1]).
true.
?- allEqual([1, 2, 3]).
false.
?- allEqual([1, X, 1]).
X = 1.
?- allEqual([A, B]).
A = B.
?- allEqual([X, 1, 2]).
false.
```

```
allEqual([]).
allEqual([_]).
allEqual([H, H|Rest]) :-
    allEqual([H|Rest]).
```

11.) Write a procedure named `zip`, which takes two lists of the same length, an output list of the same length. The output list is a list of `pair` structures, where each `pair` holds an element from each list, preserving order. If the lists are not the same length, `zip` should fail, though you shouldn't need to explicitly check the length. Example calls are below:

```
?- zip([], [], Output).
Output = [].
?- zip([hello], [goodbye], Output).
Output = [pair(hello, goodbye)].
?- zip([1, 2, 3], [a, b, c], Output).
Output = [pair(1, a), pair(2, b), pair(3, c)].
?- zip([A, B], [C, D], Output).
Output = [pair(A, C), pair(B, D)].
?- zip([foo], [bar, baz], Output).
false.
?- zip([foo, bar], [baz], Output).
false.
```

```
zip([], [], []).
zip([H1|T1], [H2|T2], [pair(H1, H2)|Rest]) :-
    zip(T1, T2, Rest).
```

12.) Consider the following code:

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

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

The length of a given list.

12.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.

12.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).
```

13.) 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(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) :-
    FirstTest is Num - 1,
    isPrime(Num, FirstTest).

% isPrime: Number, CurrentTest
isPrime(_, 1).
isPrime(Num, Test) :-
    Test > 1,
    NonZero is mod(Num, Test),
    NonZero \== 0,
    NewTest is Test - 1,
    isPrime(Num, NewTest).
```

Test Case Generation

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

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

14.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).
```

14.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.

14.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.

15.) Consider the following grammar:

```
tree ::= `node(` tree `,` tree `)` | `leaf`
```

15.a.) Write a generator for `tree` below. It's ok if the generator gets "stuck" generating similar values over and over again.

```
gen(leaf).
gen(node(Left, Right)) :-
    gen(Left),
    gen(Right).
```

15.b.) Write a modified version of your prior generator, which takes an additional depth bound, and will only generate values that are no deeper than this bound.

```
genBound(_, leaf).
genBound(Bound, node(Left, Right)) :-
    Bound > 0,
    NewBound is Bound - 1,
    genBound(NewBound, Left),
    genBound(NewBound, Right).
```