Assignment 5 - COMP 3400

Khalil Van Alphen
Student Number: 100863992
Email: khalil.va@gmail.com

1. [50%] (a) Use Prolog with predicates and variables (not propositional Prolog) to decide for an **arbitrary finite string** φ over the alphabet $\{\wedge,), \neg, (, \vee, \rightarrow, \leftrightarrow, p, q, r, s, t, ...\}$ (the latter stand for propositional variables) is a well-formed propositional formula. Use a unary predicate for this, say $Decide(\cdot)$.

Solution: We will use a boolean algebra with predicate notation to represent our rules. Our set of formula are as follows:

- $formula(atom(-))$.
- $formula(\vee(A, B)) :- formula(A), formula(B)$.
- $formula(\neg(A)) :- formula(A)$.
- $formula(\wedge(A, B)) :- formula(A), formula(B)$.
- $formula(\rightarrow(A, B)) :- formula(A), formula(B)$.
- $formula(\leftrightarrow(A, B)) :- formula(A), formula(B)$.

Our first base case rule will be applied to each term. Effectively, we must first identify atomic elements in our given formula. By this, we are in a sense declaring that a formula must be a composition of atomic elements recursively, or inductively built from atomic elements. Elements such as $p, q, r, s, t...$ are all atomic by definition.
Examples:

φ_1: $formula(\rightarrow(atom(a), atom(b)))$. VALID

φ_2: $formula(\neg(\wedge(atom(a), \vee(atom(b), atom(c)))))$. VALID

φ_3: $formula(\rightarrow(\neg(atom(a))), atom(b))$. VALID

φ_4: $formula((atom(a), atom(b)))$. INVALID

φ_5: $formula(\vee(atom(a)))$. INVALID

φ_6: $formula((atom(b), \vee(atom(a), atom(c))))$. INVALID

Traces:

```
[trace]  ?- formula(->(atom(a),atom(b))).
   Call: (8) formula((atom(a)->atom(b))) ? creep
   Call: (9) formula(atom(a)) ? creep
   Exit: (9) formula(atom(a)) ? creep
   Call: (9) formula(atom(b)) ? creep
   Exit: (9) formula(atom(b)) ? creep
   Exit: (8) formula((atom(a)->atom(b))) ? creep
true.
[trace]  ?- formula(~(^(atom(a),v(atom(b),atom(c))))).
   Call: (8) formula(~(atom(a)^v(atom(b), atom(c)))) ? creep
   Call: (9) formula(atom(a)^v(atom(b), atom(c))) ? creep
   Call: (10) formula(atom(a)) ? creep
   Exit: (10) formula(atom(a)) ? creep
   Call: (10) formula(v(atom(b), atom(c))) ? creep
   Call: (11) formula(atom(b)) ? creep
   Exit: (11) formula(atom(b)) ? creep
   Call: (11) formula(atom(c)) ? creep
   Exit: (11) formula(atom(c)) ? creep
   Exit: (10) formula(v(atom(b), atom(c))) ? creep
   Exit: (9) formula(atom(a)^v(atom(b), atom(c))) ? creep
   Exit: (8) formula(~(atom(a)^v(atom(b), atom(c)))) ? creep
true.
[trace]  ?- formula(->(~(atom(a)),atom(b))).
   Call: (8) formula((~(atom(a))->atom(b))) ? creep
   Call: (9) formula(~(atom(a))) ? creep
   Call: (10) formula(atom(a)) ? creep
   Exit: (10) formula(atom(a)) ? creep
   Exit: (9) formula(~(atom(a))) ? creep
   Call: (9) formula(atom(b)) ? creep
   Exit: (9) formula(atom(b)) ? creep
   Exit: (8) formula((~(atom(a))->atom(b))) ? creep
true.
[trace]  ?- formula((atom(a),atom(b))).
   Call: (8) formula((atom(a), atom(b))) ? creep
   Fail: (8) formula((atom(a), atom(b))) ? creep
false.
[trace]  ?- formula(v(atom(a))).
   Call: (8) formula(v(atom(a))) ? creep
   Fail: (8) formula(v(atom(a))) ? creep
false.
[trace]  ?- formula((atom(b),v(atom(a),atom(c)))).
   Call: (8) formula((atom(b), v(atom(a), atom(c)))) ? creep
   Fail: (8) formula((atom(b), v(atom(a), atom(c)))) ? creep
false.
```

An extension to determine the length of a formula is made with the following extra rules:

- $flength(atom(-), 1).$
- $flength(\vee(A, B)) :- flength(A, C), flength(B, D), LisC + D.$
- $flength(\neg(A)) :- flength(A, C)LisC.$
- $flength(\wedge(A, B)) :- flength(A, C), flength(B, D), LisC + D.$
- $flength(\rightarrow (A, B)) :- flength(A, C), flength(B, D), LisC + D.$
- $flength(\leftrightarrow (A, B)) :- flength(A, C), flength(B, D), LisC + D.$

Using the same potential formula from before: **Traces:**

```
[trace]  ?- flength(->(atom(a),atom(b)), L).
   Call: (8) flength((atom(a)->atom(b)), _5646) ? creep
   Call: (9) flength(atom(a), _5884) ? creep
   Exit: (9) flength(atom(a), 1) ? creep
   Call: (9) flength(atom(b), _5884) ? creep
   Exit: (9) flength(atom(b), 1) ? creep
   Call: (9) _5646 is 1+1 ? creep
   Exit: (9) 2 is 1+1 ? creep
   Exit: (8) flength((atom(a)->atom(b)), 2) ? creep
L = 2.
[trace]  ?- flength(~(^(atom(a),v(atom(b),atom(c)))), L).
   Call: (8) flength(~(atom(a)^v(atom(b), atom(c))), _5660) ? creep
   Call: (9) flength(atom(a)^v(atom(b), atom(c)), _5914) ? creep
   Call: (10) flength(atom(a), _5914) ? creep
   Exit: (10) flength(atom(a), 1) ? creep
   Call: (10) flength(v(atom(b), atom(c)), _5914) ? creep
   Call: (11) flength(atom(b), _5914) ? creep
   Exit: (11) flength(atom(b), 1) ? creep
   Call: (11) flength(atom(c), _5914) ? creep
   Exit: (11) flength(atom(c), 1) ? creep
   Call: (11) _5918 is 1+1 ? creep
   Exit: (11) 2 is 1+1 ? creep
   Exit: (10) flength(v(atom(b), atom(c)), 2) ? creep
   Call: (10) _5924 is 1+2 ? creep
   Exit: (10) 3 is 1+2 ? creep
   Exit: (9) flength(atom(a)^v(atom(b), atom(c)), 3) ? creep
   Call: (9) _5660 is 3 ? creep
   Exit: (9) 3 is 3 ? creep
   Exit: (8) flength(~(atom(a)^v(atom(b), atom(c))), 3) ? creep
L = 3.
[trace]  ?- flength(->(~(atom(a)),atom(b)), L).
```

```
   Call: (8) flength((~(atom(a))->atom(b)), _5650) ? creep
   Call: (9) flength(~(atom(a)), _5902) ? creep
   Call: (10) flength(atom(a), _5902) ? creep
   Exit: (10) flength(atom(a), 1) ? creep
   Call: (10) _5900 is 1 ? creep
   Exit: (10) 1 is 1 ? creep
   Exit: (9) flength(~(atom(a)), 1) ? creep
   Call: (9) flength(atom(b), _5902) ? creep
   Exit: (9) flength(atom(b), 1) ? creep
   Call: (9) _5650 is 1+1 ? creep
   Exit: (9) 2 is 1+1 ? creep
   Exit: (8) flength((~(atom(a))->atom(b)), 2) ? creep
L = 2.
[trace]  ?- flength((atom(a),atom(b)), L).
   Call: (8) flength((atom(a), atom(b)), _5646) ? creep
   Fail: (8) flength((atom(a), atom(b)), _5646) ? creep
false.
```

As we can see in the last example, formulas that do not evaluate to well-formed will not have a length under our database of rules.

For our next extension, we can apply the same inductive logic to construct from scratch a list of variables use in the formula. We know that an atomic formula has only one variable, so the list can be created with that as a base case. We make use of the built in *append*/3 operation to be efficient:

- $lvar(atom(A), [A])$.
- $lvar(\vee(A, B), L) : -lvar(A, C), lvar(B, D), append(C, D, L)$.
- $lvar(\neg(A), L) : -lvar(A, L)$.
- $lvar(\wedge(A, B), L) : -lvar(A, C), lvar(B, D), append(C, D, L)$.
- $lvar(\rightarrow (A, B), L) : -lvar(A, C), lvar(B, D), append(C, D, L)$.
- $\leftrightarrow (< - > (A, B), L) : -lvar(A, C), lvar(B, D), append(C, D, L)$.

Traces:

```
[trace]  ?- lvar(->(atom(a),atom(b)), L).
   Call: (8) lvar((atom(a)->atom(b)), _5646) ? creep
   Call: (9) lvar(atom(a), _5882) ? creep
   Exit: (9) lvar(atom(a), [a]) ? creep
   Call: (9) lvar(atom(b), _5888) ? creep
   Exit: (9) lvar(atom(b), [b]) ? creep
   Call: (9) lists:append([a], [b], _5646) ? creep
   Exit: (9) lists:append([a], [b], [a, b]) ? creep
   Exit: (8) lvar((atom(a)->atom(b)), [a, b]) ? creep
L = [a, b].
```

```
[trace]  ?- lvar(~(^(atom(a),v(atom(b),atom(c)))), L).
   Call: (8) lvar(~(atom(a)^v(atom(b), atom(c))), _5660) ? creep
   Call: (9) lvar(atom(a)^v(atom(b), atom(c)), _5660) ? creep
   Call: (10) lvar(atom(a), _5914) ? creep
   Exit: (10) lvar(atom(a), [a]) ? creep
   Call: (10) lvar(v(atom(b), atom(c)), _5920) ? creep
   Call: (11) lvar(atom(b), _5920) ? creep
   Exit: (11) lvar(atom(b), [b]) ? creep
   Call: (11) lvar(atom(c), _5926) ? creep
   Exit: (11) lvar(atom(c), [c]) ? creep
   Call: (11) lists:append([b], [c], _5934) ? creep
   Exit: (11) lists:append([b], [c], [b, c]) ? creep
   Exit: (10) lvar(v(atom(b), atom(c)), [b, c]) ? creep
   Call: (10) lists:append([a], [b, c], _5660) ? creep
   Exit: (10) lists:append([a], [b, c], [a, b, c]) ? creep
   Exit: (9) lvar(atom(a)^v(atom(b), atom(c)), [a, b, c]) ? creep
   Exit: (8) lvar(~(atom(a)^v(atom(b), atom(c))), [a, b, c]) ? creep
L = [a, b, c].
[trace]  ?- lvar(->(~(atom(a)),atom(b)), L).
   Call: (8) lvar((~(atom(a))->atom(b)), _5710) ? creep
   Call: (9) lvar(~(atom(a)), _5962) ? creep
   Call: (10) lvar(atom(a), _5962) ? creep
   Exit: (10) lvar(atom(a), [a]) ? creep
   Exit: (9) lvar(~(atom(a)), [a]) ? creep
   Call: (9) lvar(atom(b), _5968) ? creep
   Exit: (9) lvar(atom(b), [b]) ? creep
   Call: (9) lists:append([a], [b], _5710) ? creep
   Exit: (9) lists:append([a], [b], [a, b]) ? creep
   Exit: (8) lvar((~(atom(a))->atom(b)), [a, b]) ? creep
L = [a, b].
```

2. [50%]Define lists, i.e. a general predicate $list(\cdot)$ that is true with (finite) lists over the alphabet $A = \{a, b, c, d, e, f, g, h\}$. For example, the list $bdeh$, with first element b and last h, should be represented by the term $cons(b, cons(d, cons(e, cons(h, nil))))$, and $list(cons(b, cons(d, cons(e,cons(h, nil)))))$ should be true. Use Prolog to verify that $cons(b, cons(d,cons(e, cons(h, nil))))$ is a list, but not $cons(b, cons(d, e), cons(h, nil))))$.

Define a binary predicate $sublist(\cdot, \cdot)$, such that $sublist(L_1, L_2)$ becomes true when L_1 is a sublist of L_2.

Use Prolog to verify (with the representation in (a)) that ab is a sublist of $efabde$. But fd is not a sublist of the latter. Use the representation in (a) to define the length of a list. It should be $length(efabde, 6)$ true. Use Prolog to verify this. (You can use Prolog's built-in numbers and arithmetic.)

Solution:

- $list(cons(-, nil))$.
- $list(cons(-, B)) : -list(B)$.

The above rule set defined a list inductively. The first formula is our base case for a list; one that is a single item appended with the empty list. From this, any list item attached to the begin creates a valid list.

Examples:

φ_1: $cons(a, cons(b, cons(c, nil)))$. VALID

φ_2: $cons(b, cons(c, cons(d, cons(h, nil))))$ VALID

φ_3: $cons(cons(a, nil))$. INVALID

φ_4: $cons(b, cons(d, e), cons(h, nil))))$ INVALID

Traces:

```
[trace]  ?- list(cons(a,cons(b,(cons(c,nil))))).
   Call: (8) list(cons(a, cons(b, cons(c, nil)))) ? creep
   Call: (9) list(cons(b, cons(c, nil))) ? creep
   Call: (10) list(cons(c, nil)) ? creep
   Exit: (10) list(cons(c, nil)) ? creep
   Exit: (9) list(cons(b, cons(c, nil))) ? creep
   Exit: (8) list(cons(a, cons(b, cons(c, nil)))) ? creep
true .
[trace]  ?- list(cons(b,cons(c,cons(d,cons(h,nil))))).
   Call: (8) list(cons(b, cons(c, cons(d, cons(h, nil))))) ? creep
   Call: (9) list(cons(c, cons(d, cons(h, nil)))) ? creep
   Call: (10) list(cons(d, cons(h, nil))) ? creep
   Call: (11) list(cons(h, nil)) ? creep
   Exit: (11) list(cons(h, nil)) ? creep
```

```
   Exit: (10) list(cons(d, cons(h, nil))) ? creep
   Exit: (9) list(cons(c, cons(d, cons(h, nil)))) ? creep
   Exit: (8) list(cons(b, cons(c, cons(d, cons(h, nil))))) ? creep
true .
[trace]  ?- list(cons(cons(a,nil))).
   Call: (8) list(cons(cons(a, nil))) ? creep
   Fail: (8) list(cons(cons(a, nil))) ? creep
false.
[trace]  ?- list(cons(b,cons(d,e),cons(h,nil))).
   Call: (8) list(cons(b, cons(d, e), cons(h, nil))) ? creep
   Fail: (8) list(cons(b, cons(d, e), cons(h, nil))) ? creep
false.
```

As an extension, we can define $sublist(L1, L2)$ to determine whether a list is a sublist (L1 exists somewhere in L2). The logic behind this is we check for each level of the list whether the rest of the list is a prefix of the goal list. To navigate through the list we can use the suffix function which cuts the head of the list off.

- $prefix(A, L) :- consAppend(A, , L).$
- $suffix(B, L) :- consAppend(, B, L).$
- $sublist(L1, L2) :- suffix(S, L2), prefix(L1, S).$

Examples:

φ_1: $sublist(cons(1, cons(2, nil)), cons(1, cons(2, cons(3, nil)))).$

φ_2: $sublist(cons(4, cons(5, nil)), cons(1, cons(2, cons(3, cons(4, cons(5, nil)))))).$

Traces:

```
[trace]  ?- sublist(cons(1,cons(2,nil)), cons(1,cons(2,cons(3,nil)))).
   Call: (8) sublist(cons(1, cons(2, nil)), cons(1, cons(2, cons(3, nil)))) ? creep
   Call: (9) suffix(_5884, cons(1, cons(2, cons(3, nil)))) ? creep
   Call: (10) consAppend(_5884, _5886, cons(1, cons(2, cons(3, nil)))) ? creep
   Exit: (10) consAppend(nil, cons(1, cons(2, cons(3, nil))), cons(1, cons(2, cons(3, nil)))
   Exit: (9) suffix(cons(1, cons(2, cons(3, nil))), cons(1, cons(2, cons(3, nil)))) ? creep
   Call: (9) prefix(cons(1, cons(2, nil)), cons(1, cons(2, cons(3, nil)))) ? creep
   Call: (10) consAppend(cons(1, cons(2, nil)), _5886, cons(1, cons(2, cons(3, nil)))) ? cre
   Call: (11) consAppend(cons(2, nil), _5886, cons(2, cons(3, nil))) ? creep
   Call: (12) consAppend(nil, _5886, cons(3, nil)) ? creep
   Exit: (12) consAppend(nil, cons(3, nil), cons(3, nil)) ? creep
   Exit: (11) consAppend(cons(2, nil), cons(3, nil), cons(2, cons(3, nil))) ? creep
   Exit: (10) consAppend(cons(1, cons(2, nil)), cons(3, nil), cons(1, cons(2, cons(3, nil)))
   Exit: (9) prefix(cons(1, cons(2, nil)), cons(1, cons(2, cons(3, nil)))) ? creep
   Exit: (8) sublist(cons(1, cons(2, nil)), cons(1, cons(2, cons(3, nil)))) ? creep
```

```
true .
[trace]  ?- sublist(cons(4,cons(5,nil)),cons(1,cons(2,cons(3,cons(4,cons(5,nil)))))).
   Call: (8) sublist(cons(4, cons(5, nil)), cons(1, cons(2, cons(3, cons(4, cons(5, nil)))))
   Call: (9) suffix(_5926, cons(1, cons(2, cons(3, cons(4, cons(5, nil)))))) ? creep
   Call: (10) consAppend(_5926, _5928, cons(1, cons(2, cons(3, cons(4, cons(5, nil)))))) ? c
   Exit: (10) consAppend(nil, cons(1, cons(2, cons(3, cons(4, cons(5, nil))))), cons(1, cons
   Exit: (9) suffix(cons(1, cons(2, cons(3, cons(4, cons(5, nil))))), cons(1, cons(2, cons(3
   Call: (9) prefix(cons(4, cons(5, nil)), cons(1, cons(2, cons(3, cons(4, cons(5, nil)))))
   Call: (10) consAppend(cons(4, cons(5, nil)), _5928, cons(1, cons(2, cons(3, cons(4, cons(
   Fail: (10) consAppend(cons(4, cons(5, nil)), _5928, cons(1, cons(2, cons(3, cons(4, cons(
   Fail: (9) prefix(cons(4, cons(5, nil)), cons(1, cons(2, cons(3, cons(4, cons(5, nil)))))
   Redo: (10) consAppend(_5926, _5928, cons(1, cons(2, cons(3, cons(4, cons(5, nil)))))) ? c
   Call: (11) consAppend(_5914, _5934, cons(2, cons(3, cons(4, cons(5, nil))))) ? creep
   Exit: (11) consAppend(nil, cons(2, cons(3, cons(4, cons(5, nil)))), cons(2, cons(3, cons(
   Exit: (10) consAppend(cons(1, nil), cons(2, cons(3, cons(4, cons(5, nil)))), cons(1, cons
   Exit: (9) suffix(cons(2, cons(3, cons(4, cons(5, nil)))), cons(1, cons(2, cons(3, cons(4,
   Call: (9) prefix(cons(4, cons(5, nil)), cons(2, cons(3, cons(4, cons(5, nil))))) ? creep
   Call: (10) consAppend(cons(4, cons(5, nil)), _5934, cons(2, cons(3, cons(4, cons(5, nil))
   Fail: (10) consAppend(cons(4, cons(5, nil)), _5934, cons(2, cons(3, cons(4, cons(5, nil))
   Fail: (9) prefix(cons(4, cons(5, nil)), cons(2, cons(3, cons(4, cons(5, nil))))) ? creep
   Redo: (11) consAppend(_5914, _5934, cons(2, cons(3, cons(4, cons(5, nil))))) ? creep
   Call: (12) consAppend(_5920, _5940, cons(3, cons(4, cons(5, nil)))) ? creep
   Exit: (12) consAppend(nil, cons(3, cons(4, cons(5, nil))), cons(3, cons(4, cons(5, nil)))
   Exit: (11) consAppend(cons(2, nil), cons(3, cons(4, cons(5, nil))), cons(2, cons(3, cons(
   Exit: (10) consAppend(cons(1, cons(2, nil)), cons(3, cons(4, cons(5, nil))), cons(1, cons
   Exit: (9) suffix(cons(3, cons(4, cons(5, nil))), cons(1, cons(2, cons(3, cons(4, cons(5,
   Call: (9) prefix(cons(4, cons(5, nil)), cons(3, cons(4, cons(5, nil)))) ? creep
   Call: (10) consAppend(cons(4, cons(5, nil)), _5940, cons(3, cons(4, cons(5, nil)))) ? cre
   Fail: (10) consAppend(cons(4, cons(5, nil)), _5940, cons(3, cons(4, cons(5, nil)))) ? cre
   Fail: (9) prefix(cons(4, cons(5, nil)), cons(3, cons(4, cons(5, nil)))) ? creep
   Redo: (12) consAppend(_5920, _5940, cons(3, cons(4, cons(5, nil)))) ? creep
   Call: (13) consAppend(_5926, _5946, cons(4, cons(5, nil))) ? creep
   Exit: (13) consAppend(nil, cons(4, cons(5, nil)), cons(4, cons(5, nil))) ? creep
   Exit: (12) consAppend(cons(3, nil), cons(4, cons(5, nil)), cons(3, cons(4, cons(5, nil)))
   Exit: (11) consAppend(cons(2, cons(3, nil)), cons(4, cons(5, nil)), cons(2, cons(3, cons(
   Exit: (10) consAppend(cons(1, cons(2, cons(3, nil))), cons(4, cons(5, nil)), cons(1, cons
   Exit: (9) suffix(cons(4, cons(5, nil)), cons(1, cons(2, cons(3, cons(4, cons(5, nil)))))
   Call: (9) prefix(cons(4, cons(5, nil)), cons(4, cons(5, nil))) ? creep
   Call: (10) consAppend(cons(4, cons(5, nil)), _5946, cons(4, cons(5, nil))) ? creep
   Call: (11) consAppend(cons(5, nil), _5946, cons(5, nil)) ? creep
   Call: (12) consAppend(nil, _5946, nil) ? creep
   Exit: (12) consAppend(nil, nil, nil) ? creep
   Exit: (11) consAppend(cons(5, nil), nil, cons(5, nil)) ? creep
   Exit: (10) consAppend(cons(4, cons(5, nil)), nil, cons(4, cons(5, nil))) ? creep
   Exit: (9) prefix(cons(4, cons(5, nil)), cons(4, cons(5, nil))) ? creep
   Exit: (8) sublist(cons(4, cons(5, nil)), cons(1, cons(2, cons(3, cons(4, cons(5, nil)))))
```

```
true .
```

Lastly, we can define $lilength(list, size)$ to check the size of a list:

- $lilength(cons(,nil), 1)$.
- $lilength(cons(A, B), L) : -lilength(B, D), Lis1 + D$.

Examples:

φ_1: $cons(1, cons(2, cons(3, cons(4, nil)))) =$ SIZE 4

φ_2: $cons(4, cons(5, cons(5, cons(4, (cons(3, nil)))))) =$ SIZE 5

Traces:

```
[trace]  ?- lilength(cons(1,cons(2,cons(3,cons(4,nil)))), 4).
   Call: (8) lilength(cons(1, cons(2, cons(3, cons(4, nil)))), 4) ? creep
   Call: (9) lilength(cons(2, cons(3, cons(4, nil))), _6308) ? creep
   Call: (10) lilength(cons(3, cons(4, nil)), _6308) ? creep
   Call: (11) lilength(cons(4, nil), _6308) ? creep
   Exit: (11) lilength(cons(4, nil), 1) ? creep
   Call: (11) _6312 is 1+1 ? creep
   Exit: (11) 2 is 1+1 ? creep
   Exit: (10) lilength(cons(3, cons(4, nil)), 2) ? creep
   Call: (10) _6318 is 1+2 ? creep
   Exit: (10) 3 is 1+2 ? creep
   Exit: (9) lilength(cons(2, cons(3, cons(4, nil))), 3) ? creep
   Call: (9) 4 is 1+3 ? creep
   Exit: (9) 4 is 1+3 ? creep
   Exit: (8) lilength(cons(1, cons(2, cons(3, cons(4, nil)))), 4) ? creep
true .
[trace]  ?- lilength(cons(4,cons(5,cons(5,cons(4,(cons(3,nil)))))), 5).
   Call: (8) lilength(cons(4, cons(5, cons(5, cons(4, cons(3, nil))))), 5) ? creep
   Call: (9) lilength(cons(5, cons(5, cons(4, cons(3, nil)))), _6374) ? creep
   Call: (10) lilength(cons(5, cons(4, cons(3, nil))), _6374) ? creep
   Call: (11) lilength(cons(4, cons(3, nil)), _6374) ? creep
   Call: (12) lilength(cons(3, nil), _6374) ? creep
   Exit: (12) lilength(cons(3, nil), 1) ? creep
   Call: (12) _6378 is 1+1 ? creep
   Exit: (12) 2 is 1+1 ? creep
   Exit: (11) lilength(cons(4, cons(3, nil)), 2) ? creep
   Call: (11) _6384 is 1+2 ? creep
   Exit: (11) 3 is 1+2 ? creep
   Exit: (10) lilength(cons(5, cons(4, cons(3, nil))), 3) ? creep
   Call: (10) _6390 is 1+3 ? creep
   Exit: (10) 4 is 1+3 ? creep
   Exit: (9) lilength(cons(5, cons(5, cons(4, cons(3, nil)))), 4) ? creep
```

```
   Call: (9) 5 is 1+4 ? creep
   Exit: (9) 5 is 1+4 ? creep
   Exit: (8) lilength(cons(4, cons(5, cons(5, cons(4, cons(3, nil))))), 5) ? creep
true .
```

Appendices I:
Below is the full .pl file used for testing. It contains the full definitions and facts as defined by the solutions above.

```
formula(atom(_)).
formula(v(A, B)):- formula(A), formula(B).
formula(~(A)):- formula(A).
formula(^(A, B)):- formula(A), formula(B).
formula(->(A, B)):- formula(A), formula(B).
formula(<->(A, B)):- formula(A), formula(B).

flength(atom(_), 1).
flength(v(A, B), L):-flength(A, C),flength(B, D),L is C+D.
flength(~(A), L):-flength(A, C),L is C.
flength(^(A, B), L):-flength(A, C),flength(B, D),L is C+D.
flength(->(A, B), L):-flength(A, C),flength(B, D),L is C+D.
flength(<->(A, B), L):-flength(A, C),flength(B, D),L is C+D.

lvar(atom(A), [A]).
lvar(v(A, B), L):-lvar(A, C),lvar(B, D),append(C, D, L).
lvar(~(A), L):-lvar(A, L).
lvar(^(A, B), L):-lvar(A, C),lvar(B, D),append(C, D, L).
lvar(->(A, B), L):-lvar(A, C),lvar(B, D),append(C, D, L).
lvar(<->(A, B), L):-lvar(A, C),lvar(B, D),append(C, D, L).

list(cons(_, nil)).
list(cons(_, B)):-list(B).

consAppend(nil,X,X).
consAppend(cons(X,L1),L2,cons(X,L3)):-consAppend(L1,L2,L3).
prefix(A, L) :- consAppend(A, _, L).
suffix(B, L) :- consAppend(_, B, L).
sublist(L1, L2) :- suffix(S, L2), prefix(L1, S).

lilength(cons(_,nil),1).
lilength(cons(A,B), L):-lilength(B, D), L is 1+D.
```