Advanced Logical and Functional Progrmming

Problems

Logic Programming

1. Formalize the following statements as facts and rules in Prolog:

   Every friend of John is a friend of Bill. Bill is student. If Bill has a dog, then John has a cat. Every student knows John.

2. Assume the following predicates are defined with facts in a Prolog program:

```
animal(X). % X is an animal
swims(X).  % X swims
```

   (a) What is the meaning of the predicate `r(X)` defined by

```
r(X) :- animal(X),swims(X)m!,fail.
r(X) :- animal(X).
```

   (b) How many answers can you get to the query

```
?- r(X).
```

   if the program contains the facts

```
animal(horse).
animal(cat).
animal(duck).
animal(hen).
swims(horse).
swims(duck).
```

3. Consider the Prolog program

```
friend(roy, helen).
friend(helen, mike).
friend(zoe, will).
```

   (a) Define, using the cut-fail combination, the relation `not_friend(X,Y)` which holds if the relation `friend(X,Y)` does not hold.

   (b) What will be the answers to the queries:

```
?- not_friend(roy,helen).
?- not_friend(george,zoe).
```

4. Define a predicate for reversing lists, using accumulators.
5. Define a predicate for determining the maximum element from a list of integers.
6. Define a predicate for determining the minimum element from a list of integers.
7. Define a predicate `swap_every_2` for swapping every consecutive pair of elements from a list. Sample run:

   ```
   ?- swap every 2([1, 2, 3, 4, 5, 6], X).
   X=[2,1,4,3,6,5].

   ?- swap every 2([1, 2, 3, 4, 5, 6, 7], X).
   X=[2,1,4,3,6,5,7].
   ```

8. Define a predicate that detects flat lists, that is, lists that do not contain other lists as elements.
9. Define the predicate `(appnd L1 L2 R)` which holds iff `R` is the result of joining lists `L1` and `L2`.
10. Define the predicate `suffixes(L,S)` that binds `R` to the list of all suffixes of list `L`. Sample run:

    ```
    ?- suffixes([a,b],S).
    S=[[],[b],[a,b]]
    ```

11. Define the predicate `sublist(L,R)` which holds iff `R` is sublist of list `L`.
12. Define a ternary predicate `count_occurences(El, List , N)` that gives the number `N` of occurrences of `El` in the list `List`. Sample run:

    ```
    ?- count_occurences(2, [a, b, 2, 3, s, 2, 2], N).
    N=3.

    ?- count_occurences(z, [a, b, 2, 3, s, 2, 2], N).
    N=0.
    ```

13. Define a predicate `eliminate_consecutive` for the elimination of consecutive duplicates in a list. Sample run:

    ```
    ?- eliminate consecutive([a,a,a,a,b,c,c,a,a,d,e,e,e,e],X).
    X=[a,b,c,a,d,e].
    ```

14. Define the predicate `last(L,E)` which holds iff `L` is a nonempty list and `E` is its last element. Sample run:

```
?- last([1,2,3,4],E).
E=4.

?- last([],E).
false.
```

15. Define a predicate `perm(L,R)` that holds iff list `R` is a permutation of list `L`. Sample run:

```
?- perm([3,2,4,1],[3,2,1,4]).
true.
```

16. Define the predicate `isIncreasing(L)` which holds iff `L` is a list of numbers in increasing order.

17. Define a predicate `insert` for the insertion of an element in a sorted list such that the result remains sorted. Sample run:

```
?- insert(3, [1,2,4,5,7,8], X).
X=[1,2,3,4,5,7,8].
```

18. Define a predicate that determines the sum of the squares of the elements in a list of numbers. Use accumulators.

19. Define a predicate that `grel(X,Y)` on real numbers such that `grel(X,Y)` holds iff

$$Y = \begin{cases} X^2 - 1 & \text{if } X < 0, \\ 2 \cdot X - 1 & \text{if } 0 \leq X \leq 6, \\ \frac{X-1}{X-5} & \text{otherwise.} \end{cases}$$

20. Define a predicate `merge2(L1,L2,R)` that takes as inputs two sorted lists of numbers `L1` and `L2`, and binds `R` to the sorted list of elements from `L1` and `L2`. Sample run:

```
?- merge2([1,3,5,9],[2,3,4,9,11],R).
R=[1,2,3,3,4,5,9,9,11].
```

21. Define the predicate `gcdRel(M,N,D)` that holds iff `M,N` are two non-negative integers, and `D` is their greatest common divisor (gcd). You can use the following knowledge about gcd:

$$gcd(\mathtt{M},\mathtt{N}) = \begin{cases} gcd(\mathtt{N},\mathtt{M}) & \text{if } \mathtt{M} < \mathtt{N}, \\ \mathtt{M} & \text{if } \mathtt{N} = \mathtt{0}, \\ gcd(\mathtt{N},\mathtt{mod}(\mathtt{M},\mathtt{N})) & \text{otherwise.} \end{cases}$$

where `mod(M,N)` is the remainder of dividing `M` by `N`. The Prolog predicate that binds a variable `R` to `mod(M,N)` is

`R is mod(M,N).`

22. Define the predicate `three_times(L1,L2)` which holds iff the list `L2` is three times as long as list `L1`.

23. Define the predicate `add_to_end(L,E,R)` which holds if `R` is the list produced by adding element `E` at the end of list `L`.

24. Define the predicate `sorted(L,R)` which holds when `L` is a list of numbers and `R` is the list of elements of `L` in increasing order.

Functional Programming

1. Define recursively the function `gcd(a,b)` which computes the greatest common divisor of non-negative integers `a, b`.

2. Define recursively the function `sumDigits(n)` which computes the sum of digits of non-negative integer `n` in decimal representation. For example, `sumDigits(0)` = `0`, `sumDigits(43209)` = `18`, and `sumDigits(725)` = `17`.

3. Define recursively the function `merge2(L1,L2)` which takes as inputs two lists of numbers in increasing order, and returns the sorted list of elements from `L1` and `L2`. For example, if `L1` = $[1,3,5,9]$ and `L2` = $[2,3,4,9,11]$ then the result should be `[1,2,3,3,4,5,9,9,11]`.

4. A nested list of symbols is defined by the grammar:

   ⟨`Slist`⟩ `::=` ⟨`symbol`⟩ | `(list` ⟨`Slist`⟩ `...` ⟨`Slist`⟩`)`

   (a) Define the predicate `(Slist? L)` which holds if and only if `L` is a nested list of symbols.

   (b) Define the function `(flatten L)` which returns the flattened form of the nested list of symbols `L`. For example, `(flatten '(a '(b (c d) e) f))` should return the list `'(a b c d e f)`.

5. A nested list of numbers is defined by the grammar:

   ⟨`Nlist`⟩ `::=` ⟨`number`⟩ | `(list` ⟨`Nlist`⟩ `...` ⟨`Nlist`⟩`)`

   (a) Define the predicate `(Nlist? L)` which holds if and only if `L` is a nested list of numbers.

   (b) Define the function `(sumN L)` which computes the sum of numbers that occur in the nested list of numbers `L`.

6. Assume `L` is a list of positive numbers $'(a_1\ a_2\ \ldots\ a_n)$. Define the following functions:

(a) `(meanSum L)` which computes the value of $\dfrac{a_1 + a_2 + \ldots + a_n}{n}$

(b) `(hSum L)` which computes the value of $\dfrac{n}{\frac{1}{a_1} + \frac{1}{a_2} + \ldots + \frac{1}{a_n}}$

(c) `(alternatingSum L)` which computes the value of the expression

$$a_1 - a_2 + a_3 - a_4 + \ldots \pm a_n$$

Note: `(alternatingSum '())` should return 0.

7. Write a tail-recursive definition of the function `fact(n)` which computes the factorial of a non-negative integer `n`.

8. Use RACKET to define recursively the function `(downFrom n)` which takes as input a positive integer `n` and returns the list of numbers

   `'(n n − 1 ... 2 1 0)`

9. Use RACKET to define the function `(nRepeat n a)` which computes the list

$$'(\underbrace{a\ a\ \ldots\ a}_{n \text{ times}})$$

10. Write a tail-recursive definition of the function `reverseList(L)` which computes the reverse of a list `L`.

11. Write a tail-recursive definition of the function `fn(n)` which returns the value of a_n defined recursively by

$$a_1 = 4, \quad a_2 = 3, \quad a_n = 2 \cdot a_{n-1} - 3 \cdot a_{n-2} \text{ if } n > 2,$$

12. Use RACKET to write a recursive definition of the function (`join L1 L2`) which computes the result of joining lists `L1` and `L2`. For example, if `L1='(a b c)` and `L2='(x e b)` then the result should be `'(a b c x e b)`.

13. Suppose `strm` is the stream of elements

    $a_1\ a_2\ a_3\ a_4\ \ldots$

    defined in RACKET with nullary functions. Define the function `(dupl strm)` which computes the stream

    $a_1\ a_1\ a_2\ a_2\ a_3\ a_3\ \ a_4\ a_4\ \ldots$

    (That is, every element of the input stream is duplicated.)

14. Define in RACKET the stream of prime numbers. You can assume that the following functions are predefined, and use them in your implementation:

    - `(s-filter p strm)` computes the stream of elements `x` from `strm` for which `(p x)` holds.
    - `(s-map f strm)` computes the stream of elements produced by applying function `f` to every element of `strm`.