Ben-Gurion University of the Negev
Department of Computer Science
Logic programming

Assignment 1, Semester B, 2022

Deadline: April 26 at 23:00

In this assignment you will focus on integers in various representations. Two common number representations are presented, the *unary* and *binary* number representations, as well as the standard representation that you know from other programming languages. This exercise is about (almost) pure Prolog.

1 Before Solving

Some Prolog Notations

The exercise uses a few notations that you have seen in class. We repeat those notations here for convenience:

1. The notation `Pred/N` where `Pred` is a functor name and `N` is a natural number denotes a functor named `Pred` with arity `N`. For example, `append/3` denotes a functor named `append` with arity 3. We use this notation also for terms.

2. In addition, we use the mode notation, which denotes which arguments are instantiated during a predicate call. For example `append(+,+,-)` means that the predicate `append/3` may be called when the first two arguments are instantiated and the last argument may be uninstantiated. Notice that a predicate may support more than one mode. For example, `append/3` also supports mode `append(-,-,+)`.

Restrictions

There are a few restrictions you must follow in this assignment:

1. Make sure that your code is well documented and clear. Pay attention to style (we will when we check your code!). Credit will be given to elegant code writing.

2. **Do not use** constructs that we did not yet cover in class (no cuts, no findall, no if-then-else, no external libraries, and no negation of any kind – which includes the negation operators such as `\+` and the the inequality operators such as `\=`).

3. You may use the following external predicates: `append/3` `reverse/2`, `member/2` and any other predicate which was taught in class. You may use `between/3` and Prolog arithmetics, including the arithmetic inequality operator `=\=`.

Unary Notation: the alphabet consists of `0/0` and `s/1` representing the zero constant and the successor function respectively. Examples are `0`, `s(0)`, `s(s(0))`, etc. The integer value associated with a term of this alphabet is the number of `s` symbols it contains. We have seen in class the predicates `is_unary/1`, `unary_plus/3`, `unary_times/3` and `unary_leq/2` which specify respectively the set of natural numbers and the relations defining addition, multiplication and less-equal of natural numbers.

Task 1: Unary Square Root (10%)

Given two natural numbers n and k, k is the integer square root of n if $k*k \leq n$ and $(k+1)*(k+1) > n$. Write a Prolog predicate `unary_sqrt(N,K)` with mode `unary_pow(+,-)` which succeeds if and only if `K` is the integer square-root of `N`, where `N` and `K`, are unary numbers. You may assume that `N` and `K` are legal representations of natural numbers in unary notation. Your implementation must produce legal representations as well.

```
?- unary_sqrt(s(s(s(s(0)))), K).
 K = s(s(0));
 false.
?- unary_sqrt(s(s(s(s(s(0))))),K).
K = s(s(0)) ;
false.
```

Task 2: Unary Divisor (10%)

For integers n and k, k is a divisor of n if there exists r such that $n = k \cdot r$. Write a Prolog predicate `unary_divisor(N,K)` with mode `unary_divisor(+,-)` which given an integer `N` in unary representation unifies `K` with every possible divisor of `N` upon backtracking. You may assume that `N` is in legal representation of natural numbers in unary notation. Your implementation must produce legal representations as well.

For example, (your solution might return these values in any order).

```
% divisors of 12: 1, 2, 3, 4, 6, 12
?- unary_divisor(s(s(s(s(s(s(s(s(s(s(s(s(0))))))))))), X).
X = s(0) ;
X = s(s(0)) ;
X = s(s(s(0))) ;
X = s(s(s(s(0)))) ;
X = s(s(s(s(s(s(0)))))) ;
X = s(s(s(s(s(s(s(s(s(s(s(s(0)))))))))))) ;
false ;
```

Binary Notation: The alphabet consists of the symbols `[]/0`,`[|]/2`, 0/0 1/0, representing bit sequences as lists of zero and one bits (least significant bit first). The number 0 is represented as `[]`, and the number 1 as [1]. For any bit sequence N representing a **positive** number, the sequence $[0|N]$ represents $2 \times N$. For any bit sequence N representing a number,$[1|N]$ represents $2 \times N + 1$. The following are the first 7 binary numbers:

`[]`, `[1]`, `[0,1]`, `[1,1]`, `[0,0,1]`, `[1,0,1]`, `[0,1,1]`. Notice that this representation of binary numbers is least significant bit first (LSBF). Notice also that that binary numbers have no leading zeros.

Task 3: Binary Addition (15%)

Write a Prolog predicate `binary_plus(X,Y,Z)` with mode `binary_plus(+,+,-)` which succeeds if and only if `X` and `Y` and `Z` are natural numbers in binary notation such that `X+Y=Z`.

There are several ways to implement binary addition, which vary widely in efficiency. Your solution will be graded, in part, on the efficiency of your addition algorithm.

```
?- binary_plus([1,0,1], [1,1,1], C).
 C = [0, 0, 1, 1] ;
 false.
```

Task 4: Binary multiplication (10%)

Write a Prolog predicate `binary_times(X,Y,Z)` with mode `binary_times(+,+,-)` which succeeds if and only if `X` and `Y` and `Z` are natural numbers in binary notation such that `X*Y=Z`.

Notice: There are several ways to implement binary multiplication, which vary widely in efficiency. Your solution will be graded, in part, on the efficiency of your addition algorithm.

```
?- binary_times([0,1,1],[1,0,1],Z).
Z = [0, 1, 1, 1, 1] ;
false.
```

Task 5: Primality Testing (10%)

A prime number $n > 0$ is an integer that has no divisors greater than one and smaller than n. Write a Prolog predicate `is_prime(N)` with mode `is_prime(+)` which succeeds if and only if `N` is a prime number. You do not need to apply a probabilistic algorithm (but you may). Any solution which is correct with probability greater than $1 - 0.5^{100}$ will be accepted. Please provide an explanation in comments about your primality testing method.

```
?- is_prime(2).
true.
?- is_prime(22).
false.
?- is_prime(262337931900843498930963 47).
true.
?- is_prime(28769375361317015373302983).
false.
```

Task 6: Test Right Truncatable Primes(10%)

A prime number n is right truncatable if every prefix of n is also a prime number. For example, the number 7393 is right truncatable since: 7393, 739, 73 and 7 are all prime numbers. Write a Prolog predicate `right_prime(N)` with mode `right_prime(+)` which succeeds if and only if `N` is a right truncatable prime.

Task 7: Generate Right Truncatable Primes(15%)

write a Prolog predicate `right_prime_gen(N)` with mode `right_prime_gen(-)` which unifies its argument with right primes. All alternatives must be reached upon backtracking.

Notice: There is a finite number of right-truncatable primes. A partial list of right truncatable primes can be found in sequence A024770 of OEIS. You may use this list to verify your work, however, your final submission must not memoize these numbers. For example,

```
?- right_prime(7393).
true.
?- right_prime(937).
false.
?- right_prime_gen(N).
 N = 2 ;
 N = 23 ;
 N = 233 ;
 N = 239 ;
 N = 29 ;
 N = 293 ;
 N = 3 ;
 N = 31 ;
 . . .
```

The order in which `right_prime_gen(N)` provides results may be different in your implementation.

Task 8: Generate Binary Tree from Traversals (10%)

Write a Prolog predicate `gentree(Preorder,Inorder,Tree)` with mode `gentree(+,+,-)` which succeeds if and only if the binary tree `Tree` has preorder and inorder traversals that correspond to `Preorder` and `Inorder`.

```
?- gentree([1,2,3],[2,1,3],T).
T = tree(tree(nil, 2, nil), 1, tree(nil, 3, nil)) ;
false.
```

Task 9: Evaluate Expression (without parentheses) (10%)

Write a Prolog predicate `evaluate(Expression,Value)` with mode `evaluate(+,-)` which succeeds if and only if the arithmetic expression `Expression` given as a list with numbers and operators `*, +` evaluates to `Value`. Assume that `Expression` represents a legal and not empty expression. For example,

```
?- evaluate([2,'+',3,'*',4,'*',2,'+',3],Value).
Value = 29 ;
false.
```

2 Grading & Procedures

After Solving :

When grading your work, an emphasis will be given on code efficiency and readability. We appreciate effective code writing. The easier it is to read your code — the more we appreciate it! Even if you submit a partial answer. So please indent your code, add good comments.

Those simple tips can help you write a readable and clear code that will grant you a higher grade -

- Write a modular code. A complex predicate should be composed of several simple predicates.
- Define as many simple predicates as required.
- A short comment (1-3 lines) should describe each predicate you write.
- Enjoy your work, and write a code you would like to test.

Procedure

Submit a single file called `ex1.pl` with the assignment's solution. Please include a header with following statement:

/**** I, Name (ID number) assert that the work I submitted is entirely my own. I have not received any part from any other student in the class (or other source), nor did I give parts of it for use to others. I have clearly marked in the comments of my program any code taken from an external source. *****/

Submission is solo, i.e., you may *not* work in pairs. If you take any parts of your solution from an external source you must acknowledge this source in the comments. Please note that we test your work using a Linux installed SWI-Prolog (as in the CS Labs) – so please make sure your assignment runs on such a configuration.