It is a bird – Is is a plane – No it's!

Super-Prime
Assignment 3

Course: Advanced Programming, Block 1, 2013

Deadline: 20:00, September 29, 2013

1 Objective

The objective of this assignment is to gain hands-on programming experience with Prolog. The assignment consists of 2 parts, and an optional 1 part:

- Part 1: Implement the standard arithmetic functions over natural numbers using pure Prolog.
- Part 2: Use the arithmetic predicates and lists to find primes and super-primes. Again only in pure Prolog.
- Part 3: Suggestions for various other predicates. This part is **optional**, it will have no influence whatsoever on your grading.

1.1 What to hand in?

You should hand in two things:

1. Your code. A skeleton (`nats.pl`) file is provided, which you must use for the code.
2. A short report explaining the code, the question posted, and an assessment of the quality of code including what this assessment is based on. Discuss anything you find relevant to the implementation of the predicates.

1.2 Restrictions

Your program must consist of pure facts and rules only; do not use any built-in Prolog predicates or control operators (including in particular, but not limited to, cuts "`!`" and negation-as-failure `\+`). If you need any auxiliary predicates (like length), include their definitions in the program, but be sure to name them something different from the built-in ones.

Though not a strict requirement for this assignment, it is desirable that your predicates succeed only once with each possible answer. However, you must ensure that predicates fail explicitly (rather than going into an infinite loop) after enumerating all answers. If your implementation does not halt, you must, in your report, point to the cases for which the predicate diverges and explain why.

Remember, that `false` is not a "real" answer, but means that the Prolog backtracking cannot find any more solutions that succeeds. Especially, if the false occurs after Prolog finds succeeding answers.

2 Part 1: Natural Number Arithmetic

For the purpose of this assigment, the natural numbers are defined to be the non-negative integers, i.e., including zero. A natural number n can be (somewhat inefficiently) represented in pure Prolog as the term `s(...(s(z)))`; for example, the number 3 is represented as the term `s(s(s(z)))`.

In the following we will use that a well-defined natural number t_1 represents the integer n_1, t_2 represents n_2, and t_3 represents n_3.

1. Write a predicate `less/2`, such that `less`(t_1, t_2) succeeds iff $n_1 < n_2$.

   For example, `less(s(z), s(s(s(z))))` should succeed, and `less(s(z), s(z))` should fail. If t_1 is an uninstantiated logic variable, while t_2 is a proper representation of a natural number n_2, then the predicate should enumerate all numbers less than n_2. For example, the query "`?- less(X, s(s(z))).`" should succeed with `X = z` and `X = s(z)` (not necessarily in that order).

2. Write a predicate `add/3`, such that `add`(t_1, t_2, t_3) succeeds iff $n_3 = n_1 + n_2$.

   If some terms are uninstantiated logic variable, then the predicate should enumerate all numbers succeeding the predicate, and then halt. For example, the query "`?- add(s(s(z))), s(z), X.`" should succeed with `X = s(s(s(z)))`, and "`?- add(X, Y, s(z).`" should succeed with `X = z, Y = s(z)` and `X = s(z), Y = z` (not necessarily in that order).

   How can you then implement a `minus/3`, where `minus`(t_1, t_2, t_3) succeeds iff $n_3 = n_1 - n_2$.?

3. Write a predicate `mult/3`, such that `mult`(t_1, t_2, t_3) succeeds iff $n_3 = n_1 * n_2$.

   Again, if some terms are uninstantiated logic variable, then the predicate should enumerate all numbers succeeding the predicate, and then halt. For example, the query "`?- mult(X, Y, s(s(z)))`" should succeed with `X = s(z), Y = s(s(z))` and `X = s(s(z)), Y= s(z)` (not necessarily in that order).

   Explain why your solution halt or not halts after querying the above the example.

   How can `mult` be used to make a `pow2/2` and `sqrt/2`?

4. Write a predicate `mod/3` (modolus), such that `mod`(t_1, t_2, t_3) succeeds iff $n_3 = n_1 \bmod n_2$. *Hint: it is possible with the above three predicates.*

3 Part 2: Listing Super-Primes

Super-prime numbers are the subsequence of prime numbers that is located prime-numbered positions within the sequence of all prime numbers.[1] The subsequence begins:

$$3, 5, 11, 17, 31, 41, 59 \ldots$$

To find super-primes in Prolog we first need to be able to find prime numbers. Using our previously defined multiplication (`mult/3`) we can decide if a number n_1 divides n_2. (If you are not convinced start by implementing this.)

Now, a prime number is a number that is *not* dividable by any smaller number larger than 1. Note, that this definition is based on something that is "not" true, which makes it unsuitable for implementation of prime in pure Prolog (without the build-in `not/1`).

1. First, write a predicate `notprime/1`, such that `notprime`(t_1) succeeds iff n_1 is *not* a prime number.

   Querying `?- notprime(X)`", what non-primes are listed and why?

2. Now, write a predicate `prime/1`, such that `prime`(t_1) succeeds iff n_1 is a prime number. *Hint: you can define a `notdividable/2` with your arithmetic predicates.*

   Querying `?- prime(X)`", what non-primes are listed and why?

3. Write a predicate `primeList/2`, where s represents a list l such that `primeList`(s,t_1) succeeds iff l is the list of primes smaller than or equal to n_1. *Hint: you can start by making a list of all natural numbers up to and including n_1 and then filter this for non-primes.*

4. Write a predicate `superprime/1`, such that `superprime`(t_1) succeeds iff n_1 is a super-prime. *Hint: you may need to find the length of a list.*

4 Part 3 (optional): Extensions

Here are some suggestions for extensions to the assignment, in no particular order.

1. Factorial: Write a predicate `fact/2`, such that `fact`(t_1, t_2) succeeds iff $n_2 = n_1!$. *Hint: consider the invariant that you need for `fact/2` to terminate if t_1 is uninstantiated.*

2. Power: Write a predicate `power/3`, such that `power`(t_1, t_2, t_3) succeeds iff $n_3 = n_1^{n_2}$. *Hint: same a factorial, but a bit more tricky.*

3. Third-order primes: Super-primes are sometime called "higher-order primes". To be more specific we can call them 2nd-order prime (a prime of a prime list). Now a 3rd-order prime must then be the subsequence that is located at prime number positions in the list of super-primes. Write a predicate `o3prime/1`, such that `o3prime`(t_1) succeeds iff n_1 is a 3rd-order prime.

[1] Modified copy from Wikipedia: `http://en.wikipedia.org/wiki/Super-prime`.

5 Assessment

Your hand-in should contain a short section giving an overview of your solution, including **an assessment** of how good you think your solution is, and how you have tested you solution. It is also important to document all *relevant* design decisions you have made. Likewise, if you have failed to complete your implementation, describe in detail the implementation strategy and tests you had planned.

6 Helper predicates

Two helper predicates (`i2n/2` and `n2i`) are provided in the skeleton files, which can be used to convert from integers to natural numbers and back, respectably. You are welcome to use to ease your testing.