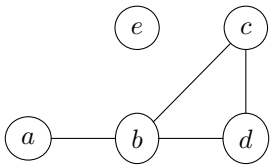
Declarative Languages

Prolog 2

1 Graphs



Address the following tasks using the above graph:

1. Transform the undirected graph in Prolog facts. Come up with an appropriate representation but make sure to explicitly represent both the nodes and edges. Since multiple representations are possible, you should pick a representation that allows you to address the following tasks in a straightforward way.

2. Implement a `neighbor/2` predicate that succeeds when two given nodes are direct neighbors. This predicate should be symmetrical such that in any given graph `neighbor(a,b)` and `neighbor(b,a)` yield the same outcome.

```
?- neighbor(a,b).
true.

?- neighbor(b,a).
true.

?- neighbor(a,e).
false.
```

3. Implement a `path/2` predicate that succeeds when a path exists between two given nodes. The predicate should not make any assumptions about the maximum length of the path, e.g. a path can have length 10. Investigate which nodes are reachable from node `a` by only instantiating the predicate's first argument. What strikes you about these solutions? Can you address this issue?

```
?- path(a,c).
true.

?- path(a,e).
...

?- path(a,X).
...
```

4. Resolve the issue with the `path/2` predicate by maintaining a list of visited nodes. When multiple paths exist between two nodes, we are interested in all possible simple paths (i.e. paths without cycles). Use the built-in predicates `\+/1` and `member/2`. Make sure that the arguments of `member/2` have been sufficiently instantiated at the time of the call.

   ```
   ?- path2(a,X).
   X = b ;
   X = c ;
   X = d ;
   X = d ;
   X = c ;
   false.

   ?- path2(a,e).
   false.
   ```

   Tip: Write a helper predicate `pathHelper/3` which explicitly maintains the list of visited nodes as an extra argument. Use this predicate to implement `path/2`.

2 Fibonacci numbers

The Fibonacci sequence starts with 0 and 1, and each subsequent number is the sum of the previous two numbers. Hence, the first ten elements in the sequence are 0, 1, 1, 2, 3, 5, 8, 13, 21, and 34. More formally, the Fibonacci sequence is defined as follows:

$$fib_n = \begin{cases} 0 & , n = 1 \\ 1 & , n = 2 \\ fib_{n-1} + fib_{n-2} & , otherwise \end{cases}$$

Implement a `fib/2` predicate that represents the relation between n and the n-th Fibonacci number. Compute Fibonacci numbers 25 and 30. Why is the implementation so inefficient? How can it be made more efficient?

3 Calculator

We will represent equations by Prolog terms of the following form:

- `number(5)`: for 5.
- `plus(`B_1`,`B_2`)`: for B_1+B_2
- `min(`B_1`,`B_2`)`: for B_1-B_2
- `neg(`B`)`: for $-B$

Where B, B_1 en B_2 are equations as well.
Write a predicate `eval/2` that evaluates an equation to its value.
An example query and its result:

```
| ?- eval(plus(number(3),number(4)), X).

  X = 7.
```

Extend the evaluator from "Boolean Formulas" so that it can evaluate arithmetic equalities to true and false. Represent the equalities by `=(A,B)` terms.
An example query, and its result:

```
| ?- eval(and(=(plus(number(3),number(4)), number(5)), not(or(fal, fal))), X).

  X = fal.
```

4 Expressive

Implement an `eval/3` predicate that evaluates arithmetic expressions given a list of assignments to the variables that appear in the expression. The following example shows the evaluation of $x + (2 * y)$, where $x = 2$ and $y = 3$.

```
?- eval(plus(var(x),times(int(2),var(y))),[pair(x,2),pair(y,3)],Value).
Value = 8.
```

An arithmetic expression is one of the following Prolog terms, where E, E_1, and E_2 are arithmetic expressions themselves:

- `int(`I`)` where I is an integer: `int(4)` denotes the value 4;
- `var(`A`)` where A is an atom: `var(x)` denotes the variable x;
- `plus(`E_1`,`E_2`)`;
- `times(`E_1`,`E_2`)`;
- `pow(`E_1`,`E_2`)`;
- `min(`E`)` to represent $-E$.

The built-in operator for exponentiation is `(**)`. For example, `8 is 2**3`.

```
?- eval(min(int(3)),[],Value).
Value = -3.

?- eval(plus(int(2),var(x)),[pair(x,3)],Value).
Value = 5.

?- eval(plus(pow(var(x),var(y)),min(plus(times(int(3),var(z)),min(var(y))))),
[pair(x,2),pair(y,3),pair(z,5)],Value).
Value = -4.
```

5 Prime numbers

Implement an `all_primes/2` predicate that computes all prime numbers smaller than a given number using the Sieve of Eratosthenes,[1] and returns them as a list. The first argument gives the upper limit, while the second argument is the output list containing all prime numbers up to that upper limit.

```
?- all_primes(3,L).
L = [2, 3].

?- all_primes(15,L).
L = [2, 3, 5, 7, 11, 13].

?- all_primes(50,L).
L = [2, 3, 5, 7, 11, 13, 17, 19, 23|...].

?- all_primes(500000,L),reverse(L,NL).
L = [2, 3, 5, 7, 11, 13, 17, 19, 23|...],
NL = [499979, 499973, 499969, 499957, 499943, 499927, 499903, 499897, 499883|...].
```

6 Extra: Infinite Turing tape

Devise a finite Prolog representation for the tape and head of a Turing machine. The tape of a Turing machine consists of an infinite string of cells, where each cell is either empty or contains a symbol. The tape contains a finite number of symbols and empty cells are represented as '#'. The head can read and write symbols on the tape and is always positioned above one particular cell.

Assignment

Implement the following predicates, and make sure to represent the tape as **a data structure that supports the above operations in $\mathcal{O}(1)$**:

[1] http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

- The `move/3` predicate moves the head one cell to the right or the left. The first argument denotes the direction represented by the constants 'right' and 'left', the second argument denotes the input tape, and the third argument denotes the output tape.
- The `read_tape/2` predicate reads the symbol in the cell under the head. The first argument denotes the tape and the second argument denotes the symbol in the cell under the head.
- The `write_tape/3` predicate writes a given symbol to the cell under the head. The first argument denotes the symbol, the second argument denotes the input tape, and the third argument denotes the output tape.