Declaratieve Talen
Prolog 2

1 Drilling on lists

In the lectures, we discussed the following implementation of the recursive `listlength/2` predicate to compute the length of a given list.

```
listlength([],0).
listlength([_|Rest],Length) :-
        listlength(Rest,LengthRest),
        Length is LengthRest + 1.
```

Use the **graphical debugger**[1] to investigate what happens when you apply this predicate to a given list. More concretely, investigate the following query:

```
?- listlength([3,1,2],Length).
```

Implement a new version of the `listlength/3` predicate that stores the number of counted elements in an accumulator. What kind of recursion is this? What is the advantage of this version?

Now implement the following predicates:

- A predicate `last/2` that returns the last element of a list.

  ```
  ?- last([1,2,a],X).
  X = a
  ```

- A predicate `next_to/3` that verifies whether the second element immediately follows the first element in a given list.

  ```
  ?- next_to([a,b,c,d],b,c).
  true.

  ?- next_to([a,b,c,d],a,c).
  false.
  ```

[1]Recall to start Prolog via the `swipl` command.

- A predicate `vector_sum/3` that computes a list in which each element is the sum of the corresponding elements in two given lists. Assume that all lists are equally long. What happens if both lists are empty? What happens if both lists are infinite? Hint: The output list can be built in the head of the predicate.

  ```
  ?- vector_sum([1,2,3],[4,5,6],L).
  L = [5,7,9]
  ```

- A predicate `look_up/3` that looks up the value that corresponds to a given key in a list of key-value pairs. This list is represented as a list of `pair/2` terms. Note the difference between terms and predicates!

  ```
  ?- look_up([pair(tom,14),pair(bart,17),pair(daan,19)],bart,Grade).
  Grade = 17
  ```

A common task in Prolog is to verify whether or not a list contains a particular element. To address this task, Prolog provides a built-in[2] predicate `member/2`, which succeeds when a given element occurs in a given list.

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

?- member(4,[1,2,3]).
false.
```

Furthermore, the `member/2` predicate can also be used to enumerate all elements in a list by calling it with an uninstantiated variable as first argument and an instantiated list as second argument.

```
?- member(X,[1,2,3]).
X = 1 ;
X = 2 ;
X = 3.
```

The built-in predicate `\+/1` succeeds when a given goal fails. Hence, this predicate can be used to invert the truth value of a call. For example, to let a predicate succeed when a particular element does not occur in a given list.

```
?- \+ member(4,[1,2,3]).
true.

?- \+ member(X,[1,2,3]).
false.
```

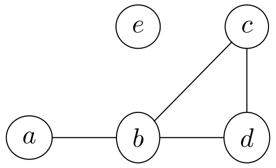
[2]The commands `help` and `apropos` provide help with *built-ins*. For example, `help(member)`.

Make sure that the arguments of the `member/2` predicate have been sufficiently instantiated at the time of the call within the negation. Use the **graphical debugger** to investigate the difference between the calls
`X = 3, \+ member(X,[1,2])` and `\+ member(X,[1,2]), X = 3`.

```
?- X = 3, \+ member(X,[1,2]).
X = 3.

?- \+ member(X,[1,2]), X = 3.
false.
```

2 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. 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 paths. 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.
```

3 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:

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

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?

4 Balanced trees

We use the same representation for trees as in the previous exercise session. A tree can either be empty or exist of a node with a value and two subtrees.

Tree := empty | node(Tree, Value, True)

Hence, the expression `node(node(empty,2,empty),3,empty)` represents a tree. A tree is balanced if the depths of the left and right subtree differ by at most one, and both subtrees are balanced as well.

Implement a predicate `balanced/1` that succeeds if a given tree is balanced and fails in all other cases.

```
?- balanced(empty).
true.

?- balanced(node(empty,3,empty)).
true.

?- balanced(node(empty,3,node(empty,4,empty))).
true.

?- balanced(node(empty,3,node(empty,4,node(empty,2,empty)))).
false.
```

Implement a predicate `add_to/3` that adds an element to a balanced tree, and ensures that the tree remains balanced. The tree should not be sorted.

```
?- add_to(node(node(empty,3,empty),2,empty),4,Tree).
Tree = node(node(empty,3,empty),2,node(empty,4,empty))
```

Extra: Perform all required changes to store in each node both a value and the depth of the tree at that point. Adapt the `add_to/3` predicate such that its complexity decreases.

5 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.
```

6 Prime numbers

Implement an `all_primes/2` predicate that computes all prime numbers smaller than a given number using the Sieve of Eratosthenes,[3] 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|...].
```

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