# Declaratieve Talen
Prolog 1

## 1 Introduction

This first exercise session is intended to familiarise you with some general concepts and practices of Prolog (more specifically the *SWI-Prolog* system), and how it works.

### 1.1 Starting & using SWI-Prolog

In the PC rooms you can start SWI-Prolog with the command `swipl`. After booting the program, you are shown the Prolog prompt:

```
| ?- <command or query>.
```

In this prompt you can ask queries or execute commands. Each command or query ends with a dot to indicate its end.

To start, make a file `.pl` that will contain your Prolog program. We recommend the `gedit` text editor for writing your programs. `gedit` supports Prolog syntax highlighting, which can be activated using `View -> Highlight Mode -> Sources -> Prolog`. We also recommend you to consistently use spaces as tabs for indentation. To activate this, go to `Edit -> Preferences`, navigate to the `Editor` tab and select “`Insert spaces instead of tabs`”. This makes your code easy to read in all editors. Prolog programs are generally stored in files with a `.pl` suffix. We prefer that you also adhere to this practice. For your first program, put the following facts in a file (e.g. `intro.pl`)

```
father(anton,bart).
father(anton,daan).
father(anton,elisa).
father(fabian,anton).

mother(celine,bart).
mother(celine,daan).
mother(celine,gerda).
mother(gerda,hendrik).
```

Your file now contains `father/2` and `mother/2` predicates. The number after "`/`" indicates the number of arguments for that predicate (also known as its *arity*). You can now load in your Prolog program with the following command(s). Note that the directory from which the `swipl` command was called serves as the *working directory* of your session. This means that the file "`intro.pl`" has to be in this directory.

```
| ?- ['intro.pl'].
```

or

```
| ?- [intro].
```

Next, you can ask queries to the prompt. SWI-Prolog will then generate answers for your query using all loaded files. Execute the next command (press `return` once to execute the query).

```
| ?- father(anton,X).
```

You'll immediately see Prolog's first answer to your query, namely `X = bart`. Your Prolog prompt now asks your next desired course of action. In this case, we want all possible answers for your query: press `;` until there are no new answers appearing.

To exit SWI-Prolog, use the command `halt.` or use the hotkey combination `<CTRL> + D`.

During these exercise sessions we encourage asking questions when you're stuck somewhere. You can also always consult the SWI-Prolog manual at:

`http://www.swi-prolog.org/pldoc/refman/`

## 1.2 Debugging

SWI-Prolog's debugger is your best friend when developing Prolog programs. Besides quickly tracing the mistakes in your program, it also allows you to "step" through your program like any other debugger. However, since Prolog's execution scheme is a little more tricky, this also provides valuable insight into how your Prolog program is executed. **This is an essential development tool for Prolog - learn how to use it!** A complete description of the debugger is given in the manual as well. Below we give a list of the most common commands.

To visually execute your query step-by-step, execute these two commands:

```
| ?- guitracer.
| ?- trace.
```

Note that your Prolog prompt is now tagged with the keyword `[trace]`:

```
| [trace]  ?- <command or query>.
```

Next you can execute your query. During each step you can perform a variety of actions:

- `<ENTER>` or `<SPACEBAR>` (step) to go to the next step in Prolog's execution.
- `s` (skip) to 'step over' the current command (= execute the step without going deeper into the execution tree).
- `r` (retry) to re-execute your last query. This is useful when you've performed a `skip` step and noticed your query failed and want to know why.
- `t` (backtrace) print the entire execution trace up until this point.
- `?` show the list of possible actions.

To turn off the tracer, use the command `notrace/0`:

```
| [trace] ?- notrace.
```

If you're not interested in debugging the entire execution of your program, but are interested in how some of your predicates or rules work, you can use `spypoints`. These are somewhat similar to breakpoints in other languages, as they allow you to interrupt normal execution when entering each spy point. To use spypoints, first use the following command to enter debug mode:

```
| ?- debug.
```

Similar to the `trace` command, your prompt is again tagged. Next, use

```
| ?- spy(predicate_name).
```

or

```
| ?- spy(predicate_name/arity).
```

to place a spypoint. Deleting spypoints is done with

```
| ?- nospy(predicate_name).
```

or

```
| ?- nospy(predicate_name/arity).
```

The command `nospyall/0` deletes all spypoints.

You can disable debug mode using `nodebug/0`. At any point you can request all debugging or trace information using the `debugging/0` predicate. For further debugging tips we refer to the SWI-Prolog manual.

## 1.3 Some practical stuff

- Active attendance is mandatory on all DT exercises.
- We very strongly recommend to first try to solve each exercise **individually**, and not to collaborate on a single computer. Designing solutions (instead of just programming them) is also part of this course, try to acquire this skill by thinking of your solutions yourself!

- The exam for this course has to be taken on the departmental computers, so use these exercise sessions to get some experience developing Prolog programs on these machines. There may not always be sufficient computers for everyone to use, in this case, try to use your own laptop.
- Structure your code! Use the layout shown below for your Prolog clauses to increase readability of your program. This layout is mandatory - if you write poorly formatted Prolog programs, points may be deducted from your grade.
  - For clauses with just a single body term:

    ```
    head_predicate :- body.
    ```

    or

    ```
    head_predicate :-
        body.
    ```

  - For clauses with multiple body terms:

    ```
    head_predicate :-
        body_1,
        body_2,
        ...,
        body_n.
    ```

  Further style advice can be found in the article with guidelines on Toledo (under "`Documents`").

## 2 Sibling

Given the following family structure, where father(anton, bart) means that **Anton is the father of Bart**.

```
father(anton,bart).
father(anton,daan).
father(anton,elisa).
father(fabian,anton).

mother(celine,bart).
mother(celine,daan).
mother(celine,gerda).
mother(gerda,hendrik).
```

Write a predicate `sibling(X,Y)` such that `X` and `Y` have the same father and mother. Construct a few queries and **study how they are executed using the debugger**.

Also test your program with the query `?- sibling(X,X)`. This should not return any result (since you're not a sibling of yourself). Filter out these undesired answers using the `\==/2` built-in that succeeds if two terms are not (provably) equal. E.g.

```
| ?- daan \== bart.

  true.

| ?- daan \== daan.

  false.
```

Afterwards, confirm that all undesired solutions are indeed filtered out with the query `?- sibling(X,Y)`. If this is not the case, reflect on the placement of your `\==/2` built-in.

## 3 Investigating your family tree safely

We want to construct a predicate that represents the "ancestor" relation. A solution for this might be:

```
ancestor(X,Y) :-
     ancestor(X,Z),
     father(Z,Y).
ancestor(X,Y) :- father(X,Y).
```

Use the facts from the previous exercise and the query shown below to experiment with. Note that we still hold the convention that father(anton, bart) means that **Anton is the father of Bart**.

```
| ?- ancestor(anton,X).
```

The desired answers are: `X = bart`, `X = daan`, and `X = elisa`. However, something's going awfully wrong here. Use the debugger to check why your program was not responding. Improve your program such that it produces the desired answers.

## 4 Peano

Prolog supports regular arithmetic of the form "`X is` *calculation*", e.g.:

```
| ?- X is ((3 + 2) - 1) / 2.

  X = 2.
```

For this exercise we will use a different representation of numbers. We represent 0 using the atom `zero`. The successor of a number `X` is represented using `s(X)`. E.g. `s(zero)` is 1, `s(s(zero))` is 2, etc. Using this representation we can define `peano_plus/3` in the manner: [1]

```
peano_plus(zero,X,X).
peano_plus(s(X),Y,s(Z)) :- peano_plus(X,Y,Z).
```

An example query:

```
| ?- peano_plus(s(s(zero)),s(zero),X).

  X = s(s(s(zero)))
```

Note the difference between **predicates** (e.g. `peano_plus`/3) and **terms** that are used as data or input (e.g. `s(s(zero))`). Use the debugger to study the example query and inspect how the answer `X = s(s(s(zero)))` is constructed using unification.

Now write the following predicates:

- `min/3` such that `min(X,Y,Z)` represents $X - Y = Z$.
- `greater_than/2` such that `greater_than(X,Y)` succeeds only if `X` is greater than `Y`.
- maximum/3 such that for `maximum(X,Y,Z)`, `Z` is unified with the largest of `X` and `Y`.

**Tip**: Start with a base case (such as $0+x = x$) and next write a clause reducing the general case stepwise until this base case (e.g. $(x+1)+y = (z+1) \iff x+y = z$).
**Extra**: Write `min/3` in terms of `peano_plus/3`.
**Extra**: Write the predicate `div/4` such that for `div(X,Y,D,R)` `D` equals the result of dividing a given `X` by a given `Y`, with `R` as remainder.

## 5 Depth

In this exercise, we represent trees as Prolog terms. An empty tree is represented using the atom `nil`. A non-empty tree is represented using a term `node(`$L$`,`$V$`,`$R$`)`, where $L$ and $R$ stand for the **l**eft and **r**ight subtree, and $V$ the **v**alue in the node.
Write a predicate `depth/2` that determines the depth of a given tree. Don't use the Peano numbers from the previous exercise. Use the built-in arithmetic of SWI-Prolog (e.g. `max/2`) and `is/2`.

Some example queries and results:

[1] The predicate `plus/3` is predefined, so we'll have to use a different name.

```
| ?- depth(node(node(nil,2,nil),1,nil),D).

  D = 2

| ?- depth(node(nil,1,node(node(nil,3,nil),2,node(nil,4,nil))),D).

  D = 3
```

## 6 Boolean formulas

We will represent Boolean formulas by Prolog terms of the following form:

- `tru`: for true
- `fal`: for false
- `and(`$B_1$`,`$B_2$`)`: for $B_1$ and $B_2$
- `or(`$B_1$`,`$B_2$`)`: for $B_1$ or $B_2$
- `not(`$B$`)`: for not $B$

Where $B$, $B_1$ en $B_2$ are also Boolean formulas.
Write a predicate `eval/2` that evaluates a Boolean formula to `tru` or `fal` according to the rules of Boolean algebra.
An example query and its result:

```
| ?- eval(and(or(not(tru),tru),fal),X).

  X = fal
```

**Tip**: Make sure you know the difference between Prolog **predicates** and Prolog **terms** (see the Peano exercise). Which parts of the example query are Prolog predicates, and which are Prolog terms?

## 7 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.
```

## 8 sumtree

Working further on the Depth exercise, we represent trees using the same kinds of prolog terms.
Write a predicate `sumtree/2` that finds the sum of all numbers in the tree. Don’t use Peano numbers, use the built-in arithmetic of SWI-Prolog.
Some example queries and results:

```
| ?- sumtree(node(node(nil,2,nil),1,nil),D).

  D = 3

| ?- sumtree(node(nil,1,node(node(nil,3,nil),2,node(nil,4,nil))),D).

  D = 10
```