Introduction to AI
First-order Logic

Project 2

21CLC05 – University of Science – VNUHCM

21127004 – 21127081 – 21127135 – 21127325

✦

1. Report

This will be a long read so please make use of bookmarks for ease of navigation.

Priority	No.	Task	Status
Required	1.1	Learn the Prolog language, write report on main features and give examples.	Done
	1.2	Learn a Prolog environment, write report on how to implement Prolog and give examples.	
	1.3	Solving deductive problems using SWI-Prolog, build a Knowledge Base from the British Royal Family Tree, give at least 20 questions, and compile the outputs.	
	2	Build a Knowledge Base from any topic containing at least 50 predicates, give at least 20 questions, and compile the outputs.	
	3	Build a logical inference program using one of the deductive methods, verify the results with 1.3 and 2.	
Bonus	a		
	b		
	c		

Project Completion Rate: 100%.

Student ID	Full name	Tasks	Contribution
21127004	Trần Nguyễn An Phong	1.1	100%
21127081	Nguyễn Minh Khôi	1.1	100%
21127135	Diệp Hữu Phúc	1.1	100%
21127325	Phan Đặng Anh Khôi	1.1	100%

2. Working with the Prolog tool (40%)

Learn the Prolog language

"Prolog is a logic programming language associated with artificial intelligence and computational linguistics." – Wikipedia

Unlike many other programming languages, Prolog is intended primarily as a declarative programming language. In Prolog, program logic is expressed in terms of relations, and a computation is carried out by running a query over these relations.

A typical Prolog program requires a Knowledge Base, which is composed of predefined clauses, i.e., facts and rules. The Knowledge Base can, then, be queried against to generate outputs. If our query is already in the Knowledge Base or it is implied by Knowledge Base, the result might be true or a list of all satisfied terms, otherwise we get false.

Query	Result
?- sister('zara_phillips', 'peter_phillips').	true.
?- nephew(X, 'princess_anne').	X = prince_william ; X = prince_harry ; X = james_viscount_severn ; false.
?- granddaughter('isla_phillips', 'timothy_laurence').	false.

Terms

- The sole data type of Prolog.
- Terms are either atoms, numbers, variables, or compound terms.

Terms	Definition	Example
Atoms	General-purpose name with no inherent meaning.	x, red, 'Hello world'
Numbers	Floats or integers.	6.9, 420, 727
Variables	Strings consisting of letters, numbers, and underscores. Must begin with an upper-case letter or underscore.	X, _360, Hello_world
Compound terms	- Composed of an atom called a *functor* and several *arguments*, which are terms. - The number of arguments is called the *arity*. An atom can be seen as a compound term with arity zero. - Lists and Strings are special cases of compound terms.	foo(bar), point(X, Y, Z), "Hello world", .(1,.(2,.(3,[]))), [1, 2, 3]

Facts and rules

Prolog programs describe relations, which are defined by means of clauses. There are two types of clauses: facts and rules. A **rule** is of the form,

$$Head :- Body.$$

which is read as "Head is true if Body is true". A rule's body consists of calls to predicates, which are called the rule's goals. The built-in logical operator ,/2 denotes conjunction of goals, and ;/2 denotes disjunction. Conjunctions and disjunctions can only appear in the body, not in the head of a rule.

Facts can be interpreted as clauses with empty bodies or $true/0$ as the bodies. Below are some examples of both,

Facts	program. hello_world. hello(world) :- true. hello(program, world).
Rules	learn(Stud, Subj) :- is_student(Stud), is_subject(Subj). execute(Program) :- is_program(Program). rule(X, Y) :- head(X), body(Y). grandparent(GP, GC) :- parent(GP, X), parent(X, GC).

Execution

"Prolog's execution strategy can be thought of as a generalization of function calls in other languages, one difference being that multiple clause heads can match a given call. In that case, the system creates a choice-point, unifies the goal with the clause head of the first alternative, and continues with the goals of that first alternative. If any goal fails in the course of executing the program, all variable bindings that were made since the most recent choice-point was created are undone, and execution continues with the next alternative of that choice-point. This execution strategy is called chronological backtracking." – Wikipedia

Well, that sure was a lot of texts! Fortunately, we are going to meticulously walk through an example, so all you need to retain from that information dump is the strategy being coined **chronological backtracking**.

Knowledge Base	```music(pulse).```
	```music(tantantan).```
	```vocal(tantantan).```
	```song(X) :- music(X), vocal(X).```

Query	```?- song(JSong).``` ```JSong = tantantan.```

1. Firstly, Prolog searches for all clauses containing the predicate **song/1**.

$$song(X) :- music(X), vocal(X).$$

2. It will then try to unify **song(X)** and **song(JSong)** by generating a common variable, e.g., **Var**. This could also be depicted as the binding of two variables **X** and **JSong**. Therefore, the query will be evaluated as,

$$song(Var) :- music(Var), vocal(Var).$$

3. Proving the query is equivalent to proving the body of that clause with the appropriate variable bindings in place, i.e., the conjunction. In this case, it is **music(Var), vocal(Var)**, and the first goal needs to be proven is **music(Var)**. Prolog proceeds to look up in the Knowledge Base all clauses with the predicate **music/1** and find,

$$music(pulse).$$

4. Similar to before, it will attempt to unify **music(Var)** and **music(pulse)**. In this case, the unification is a binding of a variable **Var** and a constant, i.e., a fact, **pulse/0**, which gives us the new body,

$$music(pulse), vocal(pulse).$$

5. Since **music(pulse)** is a known fact in the Knowledge Base, the next goal to be proven is **vocal(pulse)**. And as we can expect, **vocal(pulse)** doesn't exist in the Knowledge Base so the query for it fails. Thus, Prolog backtracks to the previous body,

$$music(Var), vocal(Var).$$

6. It continues with the look up for the predicate **music** and discovers **music(tantantan)**. Likewise, a unification for **music(Var)** and **music(tantantan)** takes place and we have,

$$music(tantantan), vocal(tantantan).$$

7. Once again, Prolog seeks **music(tantantan)** and succeeds. Hence, it moves on to **vocal(tantantan)**, thankfully this time, **vocal(tantantan)** is also a recognized fact.
8. Because all goals, i.e., the body, could be proven, the query for **song(X)** succeeds. As the query contains a variable **X**, all bindings are reported to the user, thus, we are presented with,

$$JSong = tantantan.$$

Loops and recursion

"Iterative algorithms can be implemented by means of recursive predicates." − Wikipedia

This concept is best explained through concrete examples and that is exactly what we are going to do. Nonetheless, we won't be bothered re-explaining some of the trivial steps that had already been done on the **Execution** section above.

Knowledge Base	```route(home, park).
route(park, bus).
route(bus, hcmus).
goto(Src, Dest) :- route(Src, Dest).
goto(Src, Dest) :- route(Src, X), goto(X, Dest).```
Query	```?- goto(home, hcmus).
true ;
false.```

1. Firstly, Prolog searches for all clauses containing the predicate **goto/2**. It catches the first instance with the body of just **route/2** and a unification takes place.

$$goto(home, hcmus) :- route(home, hcmus).$$

2. Expectedly, this will fail since **route(home, hcmus)** is not a fact and the body, therefore, can't be proven. Consequently, Prolog will try the next instance of **goto/2**.

$$goto(home, hcmus) :- route(home, X), goto(X, hcmus).$$

3. Now to prove the first goal of the body, a search for all clauses with predicate **route/2** begins. We discovers **route(home, park)** which is a known fact, and so Prolog binds $X = park$, and the goal is successfully proven. Additionally, we obtain a new body.

$$route(home, park), goto(park, hcmus).$$

4. While retaining the binding, Prolog moves on to the next goal of the new body. *This is where our recursion will occur*.

5. Firstly, Prolog searches for all clauses containing the predicate **goto/2**. It catches the first instance with the body of just **route/2** and a unification takes place.

$$goto(park, hcmus) :- route(park, hcmus).$$

6. Expectedly, this will fail since **route(park, hcmus)** is not a fact and the body, therefore, can't be proven. Consequently, Prolog will try the next instance of **goto/2**.

$$goto(park, hcmus) :- route(park, X), goto(X, hcmus).$$

7. Now to prove the first goal of the body, a search for all clauses with predicate **route/2** begins. We discovers **route(park, bus)** which is a known fact, and so Prolog binds $X = bus$, and the goal is successfully proven. Additionally, we obtain a new body.

$$route(park, bus), goto(bus, hcmus).$$

8. While retaining the binding, Prolog moves on to the next goal of the new body. *This is where our recursion will occur*.

9. Firstly, Prolog searches for all clauses containing the predicate **goto/2**. It catches the first instance with the body of just **route/2** and a unification takes place.

$$goto(bus, hcmus) :- route(bus, hcmus).$$

10. *Unexpectedly*, this will succeed since **route(bus, hcmus)** is a known fact and the body, therefore, can be proven. Consequently, this means the last goal of our body at step 3 has been successfully shown to be true.

$$goto(home, hcmus) : - route(home, park), route(park, hcmus).$$

11. Since all goals could be proven, our initial query **goto(home, hcmus)** succeeds. As the query contains no variables, no bindings are reported to the user and Prolog only outputs **true**.

We are perfectly aware of the **false.** which is often returned on the last line. To loosely explain, the contents above the **false.** are everything satisfying the query that Prolog could find. The **false.** appears this way can be interpreted as Prolog telling you it cannot find anything else.

Learn a Prolog programming environment

Our environment of choice is SWI-Prolog, specifically the stable version for 64-bit Windows. We won't delve further into the installation process since the installer is already very intuitive. Below is the download link,

- https://www.swi-prolog.org/download/stable

Query Interface

Although SWI-Prolog has a main interface for writing queries, this environment can essentially be initiated on any command prompt by inputting **swipl**. Additionally, a session can be ended by doing **halt.** query.
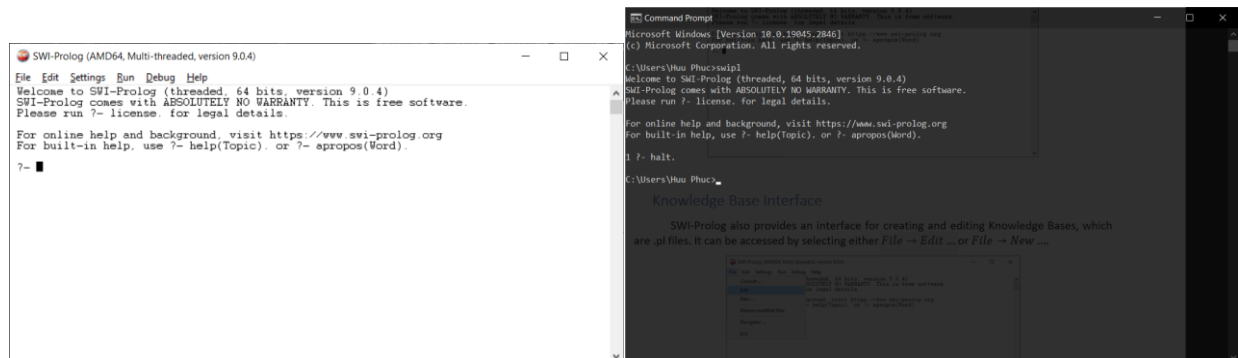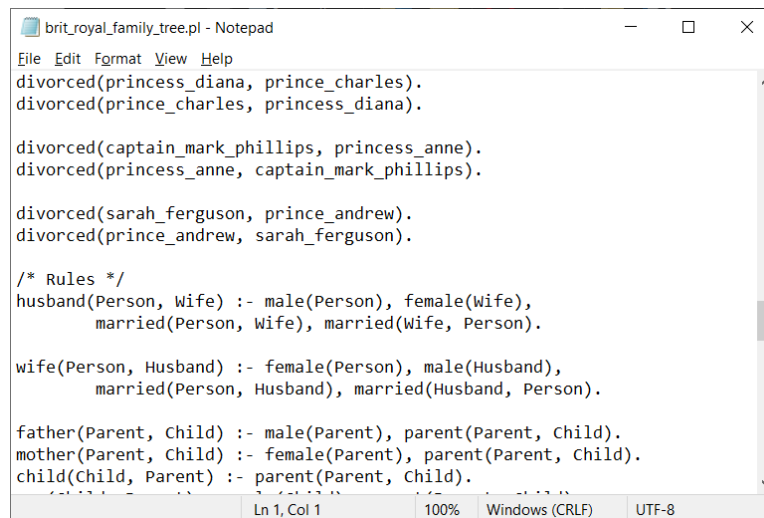

Knowledge Base Interface

SWI-Prolog also provides an interface for creating and editing Knowledge Bases, which are .pl files. It can be accessed by selecting either $File \rightarrow Edit \dots$ or $File \rightarrow New \dots$.
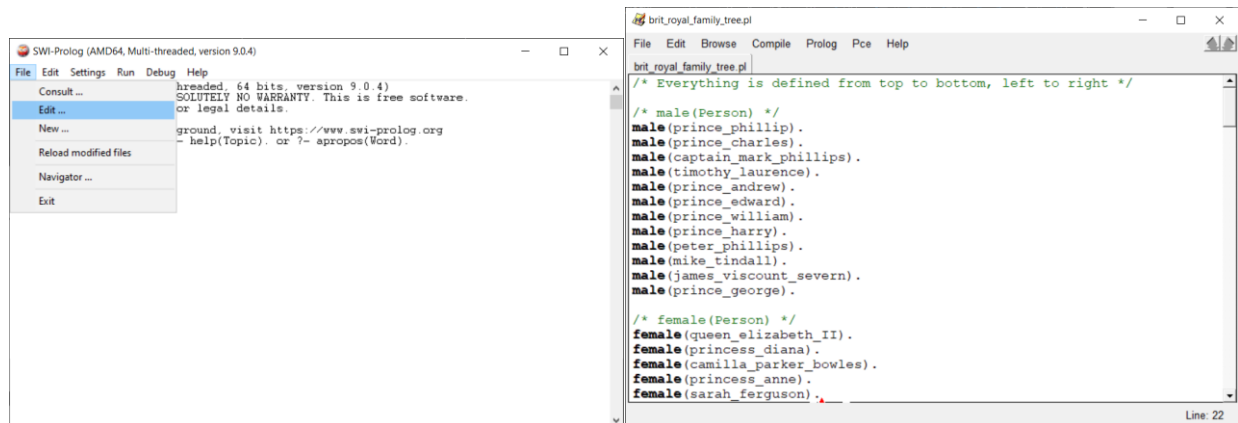
However, the .pl files can be modified with pretty much any text editors in existence. As for the format, each line denotes a clause, i.e., a rule or a fact, that must end with a dot.


Apart from using the interface and text editors, it is possible to construct or modify a Knowledge Base with queries. Considering it will be quite a hefty write for us, we will just leave the link to the source documentation here for your perusal.
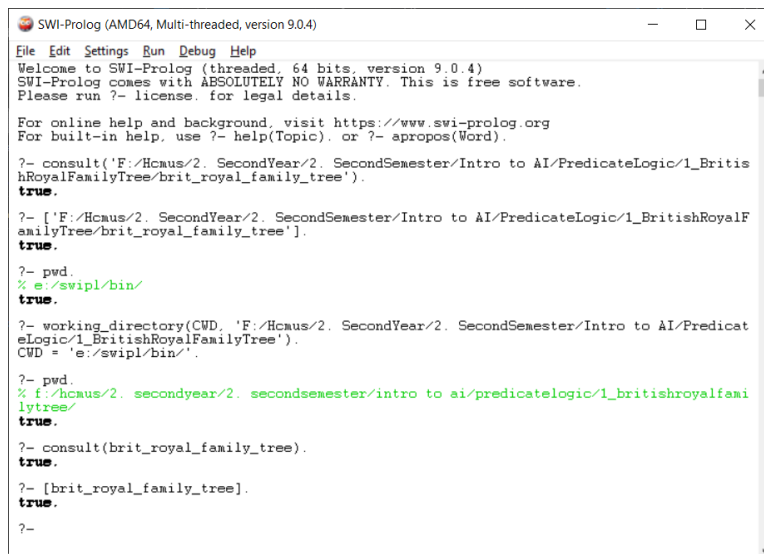
- https://www.swi-prolog.org/pldoc/man?section=dynpreds

Consulting a Knowledge Base

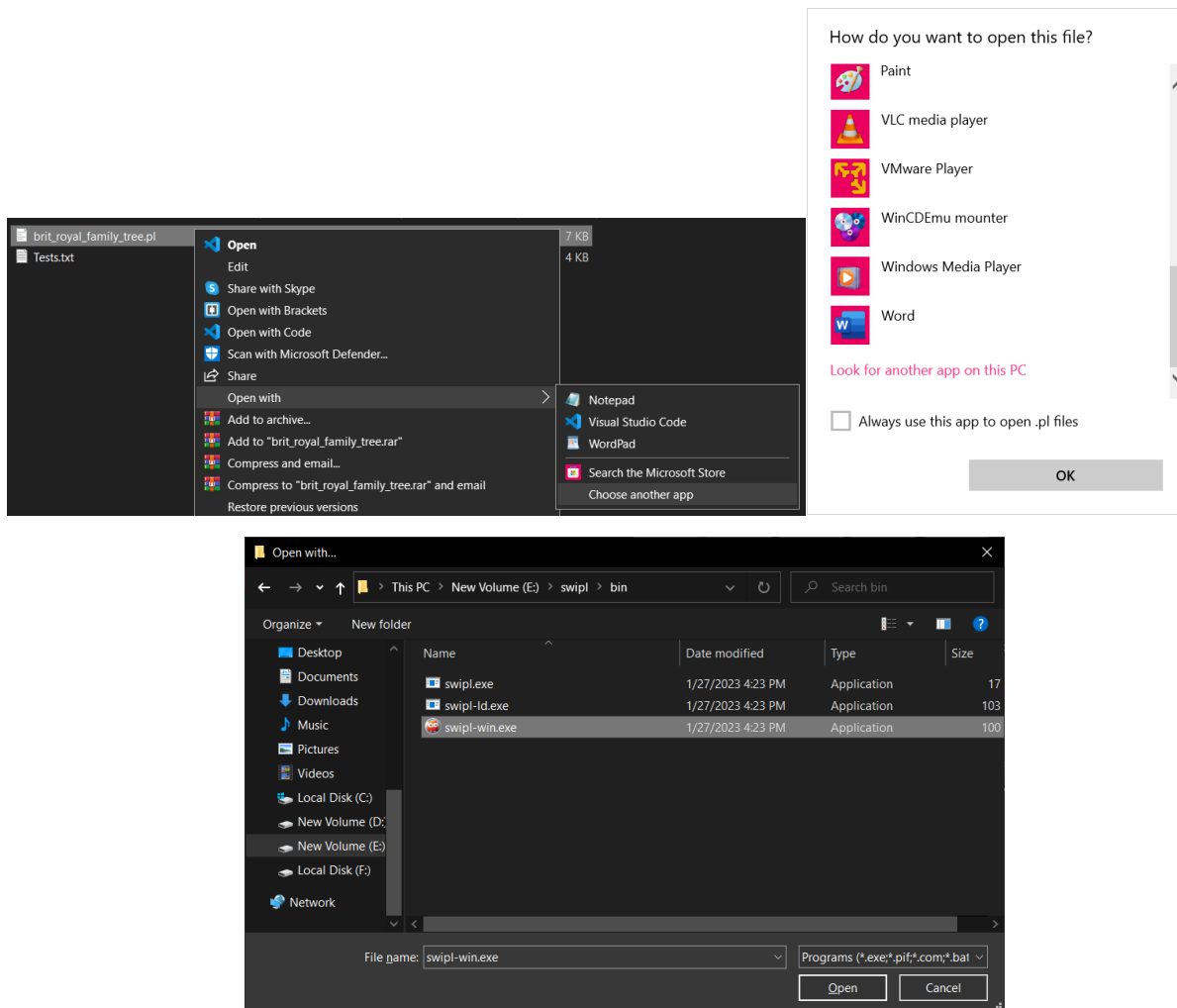There are multiple methods to load a Knowledge Base.

- **Method 1:** Using *File → Consult ….*
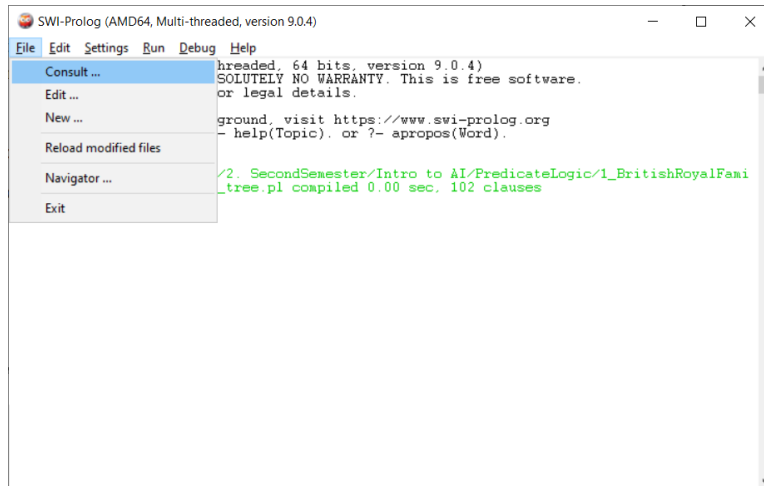
- **Method 2:** Using **consult('path:/to/pl/file')** or **['path:/to/pl/file']**.
- **Method 3:** Changing directory with **working_directory(CWD, 'path:/to/dir/with/pl/file')** then **consult(plFile)** or **[plFile]**.


- **Method 4:** Opening the .pl file directly with **swipl-win.exe**.
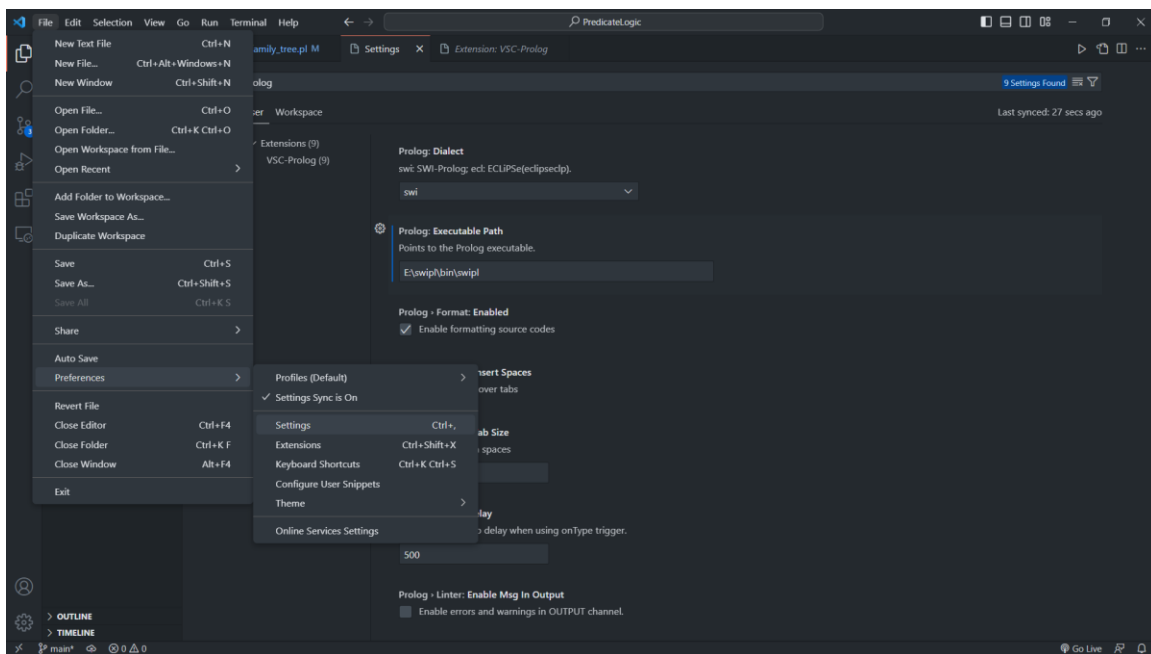
Integrating with Visual Code

In our humble opinion, the base interfaces of SWI-Prolog are not very aesthetically pleasing. It is that, combined with the annoying act of having to jump between multiple windows during test compilation or debugging, gives enough reasons to seek out the reliable VSCode.

All we need to do for the integration is a decent Prolog extension. We recommend VSC-Prolog since it has the nicest-looking syntax highlighting.
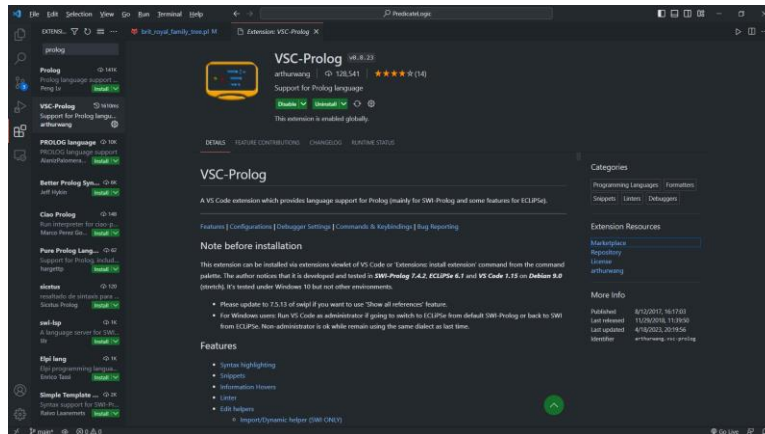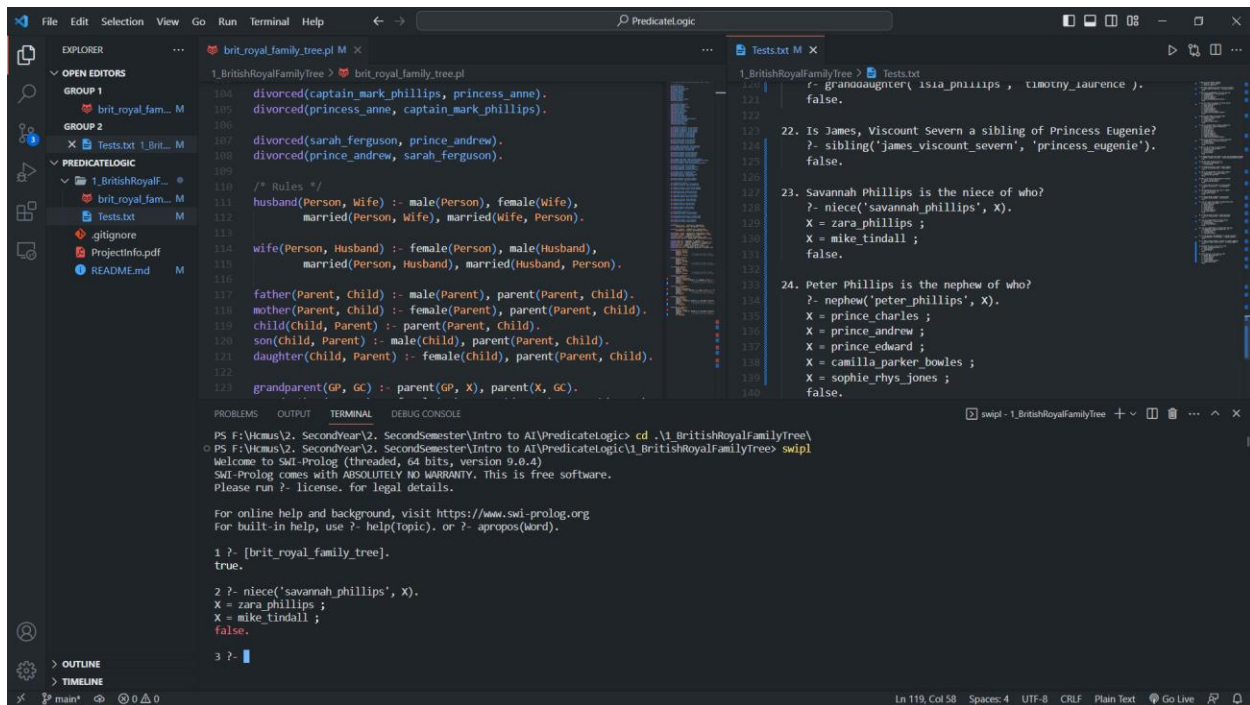
If you proceed with this extension, for the linters to work properly,

1. Go into $File \rightarrow Preferences \rightarrow Settings$.
2. Search for **prolog**.
3. Change the **Executable Path** to where your **swipl.exe** is located.

Writing some queries

Below is a picture depicting our workflow.

Type	Query
| **True/False** | ```
1 ?- [brit_royal_family_tree].
true.

2 ?- female('queen_elizabeth_II').
true.

3 ?- male('queen_elizabeth_II').
false.
``` |
| **Multiple results** | ```
1 ?- [brit_royal_family_tree].
true.

2 ?- nephew('peter_phillips', X).
X = prince_charles ;
X = prince_andrew ;
X = prince_edward ;
X = camilla_parker_bowles ;
X = sophie_rhys_jones ;
false.
``` |
| **Multiple variables** | ```
1 ?- [brit_royal_family_tree].
true.

2 ?- divorced(X, Y).
X = princess_diana,
Y = prince_charles ;
X = prince_charles,
Y = princess_diana ;
X = captain_mark_phillips,
Y = princess_anne ;
X = princess_anne,
Y = captain_mark_phillips ;
X = sarah_ferguson,
Y = prince_andrew ;
X = prince_andrew,
Y = sarah_ferguson.
``` |

| **Multiple predicates** | ```
1 ?- [brit_royal_family_tree].
true.

2 ?- female(X), uncle('prince_harry', X).
X = princess_charlotte ;
false.
``` |
| **Recursion** | ```
1 ?- assertz(parent(albert, bob)).
true.

2 ?- assertz(parent(albert, betsy)).
true.

3 ?- assertz(parent(alice, bob)).
true.

4 ?- assertz(parent(alice, betsy)).
true.

5 ?- assertz(parent(bob, carl)).
true.

6 ?- assertz(parent(bob, charlie)).
true.

7 ?- assertz((related(X, Y) :- parent(X, Y))).
true.

8 ?- assertz((related(X, Y) :- parent(X, Z), related(Z, Y))).
true.

9 ?- related(X, carl).
X = bob ;
X = albert ;
X = alice ;
false.
``` |

British Royal Family Tree

The relevant folder is **1_BritishRoyalFamilyTree**.

- **brit_royal_family_tree.pl:** The Knowledge Base constructed from the British Royal Family.
- **Tests.txt:** Records every single pair of performed query and its output. Due to their lengthy nature, we chose not to include them in this report.

For most of the predicates are very straightforward, we will only elaborate on the ones that, we think, are noteworthy.

Sibling, brother, sister

Since they share basically the same implementation, let us examine **brother**.

```prolog
brother(Person, Sibling) :-
    male(Person),
    parent(X, Person),
    parent(X, Sibling),
    female(X),              % The gender here doesn't matter,
    Person \= Sibling.      % we choose only one to avoid duplication.
```

Since we only care about whether **Person** and **Sibling** share one parent **X**, the parent's gender doesn't matter. Hence, we arbitrarily picked **female(X)**, we could also have gone with

male(X). One last thing is that we must make sure that two different people are really being checked, thus, **Person \= Sibling**.

Here is an example of the duplication when we removed **female(X)**.

```
?- brother('prince_george', X).
X = princess_charlotte ;
X = princess_charlotte.
```

Aunt, uncle, niece, nephew

Since they share basically the same implementation, let us examine **aunt**.

```
aunt(Person, NieceNephew) :-
      female(Person),
      parent(X, NieceNephew),
      (parent(Y, Person); (parent(Y, Z), husband(Z, Person), X \= Z)),
      female(Y),              % The gender here doesn't matter,
      parent(Y, X),           % we choose only one to avoid duplication.
      X \= Person.
```

We consider an aunt to be either a parent's sister or uncle's wife. As we have already written the logic for the **sister** predicate, we will skip them here. On to the uncle's wife, instead of checking for a parent **Y** of both **X** and **Person**, we need a parent **Y** of both **X** and the husband **Z** of **Person**, and to also ensure that we don't check someone multiple times with **X \= Z**.

Once again, here is an example of the duplication when we removed **female(X)**.

```
?- aunt('autumn_kelly', X).
X = mia_grace_tindall ;
X = mia_grace_tindall ;
false.
```

And here is an example of what will happen if the **X \= Z** check is neglected.

```
?- aunt('autumn_kelly', X).
X = savannah_phillips ;
X = isla_phillips ;
X = mia_grace_tindall ;
false.
```

2. Build a Knowledge Base with Prolog (30%)

3. Implement logic deductive system in the programming language (30%)

4. Reference

Working with the Prolog tool

- https://en.wikipedia.org/wiki/Prolog
- https://sicstus.sics.se/sicstus/docs/3.12.9/html/sicstus/Compound-Terms.html
- https://youtu.be/SykxWpFwMGs
- https://www.geeksforgeeks.org/prolog-an-introduction/
- https://www.swi-prolog.org/pldoc/man?section=dynpreds

Build a Knowledge Base with Prolog

Implement logic deductive system in the programming language