# Draft 1

I would like to design a project whose ultimate goal is to produce a highly functional implementation of the Prolog Language. My plan is unique in several ways. I wish to develop the modules as a process that incorporates the use of the two languages of Python and RUST, as well as YAML. The way these 3 languages will interact is summarized this way. YAML will be used to define and keep track of the entire project as well as the architecture, including directory structure, module names as well as what they do and how they interact.

The development of the Prolog system will be performed in major steps that we will be described as a sequential series of “Prolog Lite” languages with increasing complexity and functionality. These “Prolog Lite” languages will be referred to using the notation: P1, P2,..Px…Pn where Pn is the final version of our Prolog system; And each step, Px, a fully defined languages to be implemented in both Python and RUST.

At each major step ( Px ), there will be a both a Python version, called Yx, and a corresponding RUST version, called Rx. Once Px and Yx are complete, they will have an equivalent (not including performance) functionality. The goal will be to, as much as possible, putting performance and resource uses aside an equivalent set of tests Tx, consisting of the pair (TYx,TRx) which will have a one to one correspondence and all will pass.

There will be a YAML document {P1…Pn}.YAML that contains all the information required to generate a language reference for each Px : x >=1 & x <= n. There will be a directory root for each of the Python (Yx) an RUST (Rx) versions of the source code that can be run and all tests will pass.

# Draft 2

I aim to design a project with the ultimate objective of crafting a robust implementation of the Prolog language. My approach is distinctive in several aspects. Specifically, I plan to build the modules via a process that integrates the use of Python, RUST, and YAML languages.

The interaction of these three languages can be summarized as follows: YAML will be responsible for defining and monitoring the entire project and its architecture, which includes the directory structure, module names, their functions, and their interactions.

The Prolog system's development will occur in a series of significant steps, presented as an ascending series of "Prolog Lite" languages, each with increased complexity and functionality. These "Prolog Lite" languages will be denoted as P1, P2,... Px... Pn, where Pn represents the final version of our Prolog system. Each step, Px, signifies a fully defined language, set to be implemented in both Python and RUST.

At each significant step (Px), there will be a Python version (Yx) and a corresponding RUST version (Rx). Upon completion of both Px and Yx, they will demonstrate equivalent functionality, ignoring differences in performance and resource usage. Our objective is to create an identical set of tests Tx, comprised of the pair (TYx, TRx), which will correlate one-to-one and pass consistently.

A YAML document, denoted as {P1...Pn}.YAML, will encapsulate all the necessary information for generating a language reference for each Px, where x is greater than or equal to 1 and less than or equal to n. Each Python (Yx) and RUST (Rx) version of the source code will have a designated root directory, from where it can be executed, and all tests are expected to pass.

# Draft 3

YPR – Yaml, Python, Rust compiler development framework.

The YPR framework defines a set of languages and protocols for driving a project that delivers a robust implementation of the Prolog language. By using YPR, this Prolog implementation stands out due to its unique approach, which involves building modules via a process that blends the use of three programming languages: Python, RUST, and YAML along with a tight correspondence among these three languages.

YAML sits at the top of the pyramid, to define and monitor the entire project and its architecture. It manages the directory structure, module names, their functions, and how they interact.

The project's execution entails developing the Prolog system in a series of stages, each represented as "Prolog Lite" languages of escalating complexity and functionality. These "Prolog Lite" languages are denoted as P1, P2,... Px... Pn, where Pn symbolises the ultimate version of the Prolog system. Each stage, Px, corresponds to a fully defined language that is implemented in both Python and RUST.

At each significant stage (Px), a Python version (Yx) and a corresponding RUST version (Rx) are created. These versions demonstrate equivalent functionality upon completion, setting aside differences in performance and resource usage. The aim is to generate an identical set of tests Tx, comprising the pair (TYx, TRx), which have one-to-one correspondence and consistently pass all tests.

A YAML document, represented as {P1...Pn}.YAML, encompasses all the requisite information to generate a language reference for each Px, where x is greater than or equal to 1 and less than or equal to n. For each version of the source code, in both Python (Yx) and RUST (Rx), a designated root directory is established, enabling its execution, and facilitating all tests to pass.

# Roadmap

This roadmap closely follows the evolution and unique aspects of logic programming, would be more appropriate.

1. **P1: "BooleanLogic"**: A language that expresses and evaluates boolean logic, incorporating only constants and the operators: [“,”, “;”, “\+] (and or and not). NOT (\+). This lays the foundation for basic parsing and evaluation.
2. **P2: "PropositionalLogic"**: An extension of BooleanLogic to include variables, thus introducing the concept of an environment or variable assignment.
3. **P3: "PredicateLogic"**: Extends PropositionalLogic to include predicates and quantifiers, which introduces variable scoping.
4. **P4: "FirstOrderLogic"**: Builds upon PredicateLogic to include functions, transitioning from propositional to first-order logic.
5. **P5: "HornClausalLogic"**: Extends FirstOrderLogic to restrict it to Horn clauses, which are crucial for Prolog's computational model.
6. **P6: "SimpleProlog"**: Extends HornClausalLogic to include the basic features of Prolog: facts, rules, and queries. The key addition here is the execution model that aims to satisfy a query by finding a valid assignment for its variables.
7. **P7: "PrologWithBacktracking"**: Enhances SimpleProlog with support for backtracking, a quintessential feature of Prolog's execution model.
8. **P8: "PrologWithBuiltinPredicates"**: Builds upon PrologWithBacktracking by adding support for built-in predicates like arithmetic operations, list operations, I/O operations, etc.
9. **P9: "PrologWithCut"**: Extends PrologWithBuiltinPredicates with support for the cut operator, which influences the backtracking mechanism.
10. **Pn: "FullProlog"**: The final stage that extends PrologWithCut with all remaining features of the Prolog standard, such as modules, operator definitions, exceptions, etc.

This sequence provides a thorough and detailed roadmap for your project, closely aligned with the principles of logic programming and the unique features of the Prolog language.

Here is a refined specification for "P1: BooleanLogic".

1. **Language Constructs**: The language should support three types of constructs:
   * **Facts**: These are predicates that are either true or false, e.g., p. which implies that 'p' is a fact and it's true.
   * **Logical Operators**: The operators , (AND), ; (OR), and \+ (NOT).
   * **Queries**: These are expressions that ask if a given predicate or a combination of predicates is true.
2. **Parsing**: The system must be capable of correctly parsing input strings to form boolean expressions. The parser should be able to handle:
   * Whitespace: The parser should ignore any whitespace characters.
   * Syntax errors: If the input string does not form a valid Prolog query, the system should report a syntax error.
3. **Evaluation**: Once parsed, the system should be able to evaluate a boolean query to a single boolean value (true or false). The evaluation should follow these rules:
   * A, B (A AND B) returns true if both A and B are true facts, and false otherwise.
   * A ; B (A OR B) returns true if either A or B is a true fact, and false otherwise.
   * \+ A (NOT A) returns true if A is not a true fact, and false if A is a true fact.
4. **Fact Declaration and Storage**: The system should be able to take input lines that declare facts. Once declared, these facts are stored and can be referenced in later queries. Example:
   * p. (Declare 'p' as a fact)
   * q. (Declare 'q' as a fact)
   * p, q. (Query: Is both 'p' and 'q' true? The system should return 'true'.)
5. **Execution**: The system should execute line by line. Facts are stored as they're declared. Queries are evaluated as they're encountered.

This basic specification should give you a good starting point for implementing the first version of your Prolog system, "P1: BooleanLogic".

When we say that P1 is a subset of Prolog, it means that it includes a certain part of the complete functionality of Prolog. In this case, it implies that P1 should include certain constructs of the Prolog language but not necessarily all.

In the context of your roadmap, P1 or "BooleanLogic" seems to be a very basic subset of Prolog. It's designed to express and evaluate Boolean logic using the Prolog logical operators AND (represented as ,), OR (represented as ;), and NOT (represented as \+). This implies that P1 should support predicates but in a very basic form, i.e., as facts that are either true or false.

Here is how predicates come into play:

1. **Predicates as Facts**: In P1, predicates will be used in their simplest form: as facts. A fact in Prolog is a simple, standalone predicate that is unconditionally true. For example, the statement p. means that 'p' is a true fact.
2. **Predicates in Queries**: Once facts are declared, they can be used in queries to evaluate boolean expressions. For example, if 'p' and 'q' are declared as true facts, you can form a query like p, q. which will evaluate to true.

However, P1 does not support predicates in the sense of a more advanced usage found in full-fledged Prolog, like defining rules with variables and clauses. That functionality is introduced gradually in your roadmap in stages P3 ("PredicateLogic") and onwards. In P1, you're laying down the very basics of parsing and evaluation using facts and boolean logic operators.

Alright, in order to interact with the P1 system, you'll need a shell, which will allow users to load P1 programs, declare facts, and evaluate queries. Here's a basic specification for such a shell:

1. **Interactive Mode**: The shell should offer an interactive mode where users can input P1 programs (fact declarations) and queries line by line, with the shell providing immediate feedback on each line.
2. **Loading Programs**: Users should be able to load complete P1 programs from a file. The program will contain a series of facts that will be stored in the system. The shell should provide a command to load programs, something like LOAD filename.p1. Upon executing this command, all facts in the given file should be loaded into the system.
3. **Fact Declaration**: After a program is loaded, or in the interactive mode, users should be able to declare new facts. Each fact is declared as a predicate followed by a period (e.g., p.). Upon declaration, the fact should be stored in the system and can be used in subsequent queries.
4. **Query Evaluation**: Users should be able to input queries to check the truth value of certain conditions based on the facts stored in the system. Each query is a boolean expression formed using facts and logical operators, followed by a period (e.g., p, q.). Upon inputting a query, the system should evaluate it and return the result (true or false).
5. **Error Handling**: If a user inputs an incorrect command, an unrecognized fact, or an invalid query, the shell should return an appropriate error message.
6. **Exit Command**: There should be a command to exit the shell, such as EXIT.
7. **Help Command**: There should be a command to display help information, such as HELP. This command should list all possible commands with a brief description of each.

Here's an example interaction with the shell:

css

> LOAD my\_program.p1

Program loaded successfully.

> p.

Fact 'p' declared.

> q.

Fact 'q' declared.

> p, q.

true

> \+ r.

true

> EXIT

Goodbye!

In this example, my\_program.p1 is a file containing P1 program, p. and q. declare new facts, p, q. is a query that returns true, \+ r. is another query that returns true (because 'r' is not a declared fact), and EXIT exits the shell.

How are asserting facts different than making queries?

ChatGPT

2 / 2

In the context of Prolog and its subset P1, asserting facts and making queries are two distinct operations that serve different purposes. Here's how they are different:

**Asserting Facts**

1. **Purpose**: The purpose of asserting facts is to inform the system about some piece of knowledge that is unconditionally true. Facts represent the base knowledge that the system has about the world.
2. **Syntax**: In P1, a fact is asserted simply by stating it followed by a period. For example, p. asserts that 'p' is a fact.
3. **Effect**: Once a fact is asserted, it's stored in the system's knowledge base. It can then be used in later queries to determine the truth value of certain expressions.

**Making Queries**

1. **Purpose**: The purpose of making a query is to ask the system about the truth value of a certain expression based on the facts it knows. Queries are used to extract or infer information from the knowledge that the system has.
2. **Syntax**: In P1, a query is a boolean expression formed using facts and logical operators, followed by a period. For example, p, q. is a query that asks whether both 'p' and 'q' are true.
3. **Effect**: Upon making a query, the system evaluates the expression based on the facts in its knowledge base and returns the result. The system's knowledge base is not modified by making queries.

In summary, asserting facts is about telling the system what is true, while making queries is about asking the system what is true based on what it knows. They are two sides of the same coin: facts are the input to the system, and queries are the output.

P1/

│

├── src/

│ ├── \_\_init\_\_.py

│ ├── lexer.py

│ ├── parser.py

│ ├── evaluator.py

│ ├── knowledge\_base.py

│ ├── shell.py

│ └── file\_loader.py

│

├── tests/

│ ├── \_\_init\_\_.py

│ ├── test\_lexer.py

│ ├── test\_parser.py

│ ├── test\_evaluator.py

│ ├── test\_knowledge\_base.py

│ ├── test\_shell.py

│ ├── test\_file\_loader.py

│ └── test\_integration.py

│

├── programs/

│ └── example\_program.p1

│

├── main.py

└── README.md

Here's what each part of the directory does:

* src/: This directory contains the Python source code for the implementation. Each module is represented by a separate .py file.
* tests/: This directory contains the tests for the project. Like the source code, each module has a corresponding test file. There's also a file for integration tests.
* programs/: This directory contains example P1 programs that you can use to test the system.
* main.py: This is the main entry point of the application. It starts the shell and allows users to interact with the system.
* README.md: This file contains information about the project, such as how to install and run it, and how to use the shell.

This structure allows for a clean separation of different parts of the project. The src/ and tests/ directories allow for a clear distinction between the implementation and the tests. The programs/ directory provides a place to store example programs, which can be useful for testing and demonstrating the system.

Alright, let's define the grammar and AST for P1, in line with your specification.

**Grammar**

1. **Program**: A program in P1 consists of zero or more **statements**. Each statement ends with a period (.) and they can be either **fact declarations** or **queries**.
2. **Fact Declarations**: A fact declaration is a string of alphanumeric characters that represents a predicate. For example, p. declares 'p' as a fact.
3. **Queries**: A query is an **expression** followed by a period. For example, p, q. is a query that checks if both 'p' and 'q' are true.
4. **Expression**: An expression can be either a fact or a logical **operation**.
5. **Operation**: An operation is either:
   * An AND operation, represented as A, B, which checks if both A and B are true.
   * An OR operation, represented as A ; B, which checks if either A or B is true.
   * A NOT operation, represented as \+ A, which checks if A is not true.

In EBNF, the grammar could be represented as:

<program> ::= { <statement> '.' }

<statement> ::= <fact\_declaration> | <query>

<fact\_declaration> ::= <fact>

<query> ::= <expression>

<expression> ::= <fact> | <operation>

<operation> ::= <expression> ',' <expression>

| <expression> ';' <expression>

| '\+' <expression>

### AST

The AST for P1 should represent the structure of the program. For simplicity, it could be a list of statements, where each statement is either a fact declaration or a query. A fact declaration is represented as a string, and a query is represented as an expression.

An expression is either a fact or an operation. An operation is a tuple where the first element is a logical operator and the second element is a list of one or more expressions. An operator is one of the strings 'AND', 'OR', or 'NOT'.

Using Python data structures, the AST could look something like this:

* Fact declaration: 'p'
* AND operation: ('AND', ['p', 'q'])
* OR operation: ('OR', ['p', 'q'])
* NOT operation: ('NOT', ['p'])
* Query: ('QUERY', ('AND', ['p', 'q']))

So, for example, the P1 program:

p.

q.

p, q.

Could be represented in the AST as:

['p', 'q', ('QUERY', ('AND', ['p', 'q']))]

The Warren Abstract Machine (WAM) is a virtual machine designed to efficiently execute Prolog programs. It's a crucial aspect of any Prolog interpreter, as it introduces and utilizes various techniques and structures (like choice points and environments) to handle Prolog's specific execution model, including backtracking and unification.

However, introducing WAM might be considered an advanced step. In your project roadmap, you gradually introduce various Prolog features, starting with basic Boolean logic in P1, and moving towards more complex aspects of the language.

Given that structure, a good point to start introducing WAM concepts might be at the stage where you start dealing with more complex execution models. For instance, when you implement "P6: SimpleProlog", you introduce facts, rules, and queries, and the execution model that aims to satisfy a query by finding a valid assignment for its variables. This stage could be an appropriate time to start leveraging WAM to handle this efficiently.

However, the actual introduction of WAM in your implementation can depend on various factors, including how you decide to structure your implementation, your familiarity with WAM, and the particular features of the Prolog language you wish to focus on. Some may even introduce WAM at an earlier stage to facilitate certain parts of the implementation. It's really a design choice that depends on the specifics of your project.

Keep in mind that once introduced, WAM will most likely become a central part of your interpreter's implementation, and as such, understanding it thoroughly is paramount to successfully executing your project.

# Draft 4

Based on the provided summary, it appears that you are developing a framework called YPR (Yaml, Python, Rust) for implementing a robust Prolog language implementation. YAML plays a central role in defining and monitoring the project and its architecture, managing directory structures, module names, their functions, and their interactions.

The project involves developing the Prolog system in stages represented as "Prolog Lite" languages (P1, P2, ..., Px, ..., Pn) of increasing complexity and functionality. Each stage corresponds to a fully defined language implemented in both Python and Rust. At each significant stage (Px), both a Python version (Yx) and a corresponding Rust version (Rx) are created to demonstrate equivalent functionality, with performance and resource usage differences set aside. The goal is to generate an identical set of tests (Tx) consisting of pairs (TYx, TRx) that consistently pass all tests.

A YAML document, represented as {P1...Pn}.YAML, contains all the necessary information to generate a language reference for each stage (Px). For each version of the source code in Python (Yx) and Rust (Rx), a designated root directory is established to enable execution and facilitate passing all tests.

It seems that you are utilizing YAML to define the project structure, architecture, and configurations, while Python and Rust are the primary languages used for implementing the Prolog system at different stages. YAML serves as a coordinating language in this framework, providing a structured representation of the project's components and facilitating the generation of language references.

# Draft 5

YPR – Yaml, Python, Rust compiler development framework.

The YPR framework is designed to deliver a robust implementation of the Prolog language through a unique approach that combines three programming languages: Python, Rust, and YAML. At the top of the framework sits YAML, which defines and monitors the entire project and its architecture, including directory structures, module names, functions, and their interactions.

The Prolog system is developed in stages, represented as "Prolog Lite" languages (P1, P2, ..., Px, ..., Pn), each progressively increasing in complexity and functionality. Every stage (Px) is implemented in both Python (Yx) and Rust (Rx) to ensure equivalent functionality, with performance and resource usage differences set aside.

The YAML structure within the YPR framework includes Yx instances for each stage (Px), capturing the necessary information for generating language references. Additionally, there is a YRoot document that provides an overview of the P1...Pn series and their relationships, facilitating iterative development and refinement. Each Px represents a distinct set of feature enhancements over the previous stage (P(x-1)), allowing for iterative development and refinement.

A key aspect of the YPR framework is the 100% syntax and semantic compatibility between each stage (Px) and the ultimate version (Pn) of the Prolog system. This property, known as the "Px Compatibility Rule," is crucial. It allows us to assert that a specific stage (Px) is incorrect if it fails to pass the Px Compatibility Rule, indicating that its code does not run in all higher-level stages.

The YPR framework incorporates a JSON schema and a set of semantic rules to establish the validity of YPR YAML documents. These documents define the project structure, architecture, and configurations. While the YPR YAML documents play a critical role in driving the project, they are not standalone languages in their own right. Instead, they leverage the expressive power of YAML to define and manage the evolving Prolog system. The possibility of developing a DSL (Domain-Specific Language) may arise in the future, but for now, the focus remains on leveraging YAML.

In summary, the YPR framework combines YAML, Python, and Rust to develop a Prolog implementation. YAML serves as the orchestrating language, while Python and Rust implement each stage of the Prolog system. The framework ensures syntax and semantic compatibility, provides language references, and utilizes JSON schema and semantic rules to validate YPR YAML documents. While a DSL may be considered in the future, YAML remains the primary choice for leveraging its capabilities within the YPR framework√

# Draft 6

YPR – Yaml, Python, Rust compiler development framework.

The YPR framework is designed to deliver a robust implementation of the Prolog language through a unique approach that combines three programming languages: Python, Rust, and YAML. At the top of the framework sits YAML, which defines and monitors the entire project and its architecture, including directory structures, module names, functions, and their interactions.

The Prolog system is developed in stages, represented as "Prolog Lite" languages (P1, P2, ..., Px, ..., Pn), each progressively increasing in complexity and functionality. Python and Rust independently implement functionally equivalent units for each stage of the Prolog-lite sequence, while YAML serves as the orchestrating language, defining and monitoring the project's architecture and facilitating the coordination between Python and Rust implementations.

The YAML structure within the YPR framework includes Yx instances for each stage (Px), capturing the necessary information for generating language references. Additionally, there is a YRoot document that provides an overview of the P1...Pn series and their relationships, facilitating iterative development and refinement. Each Px represents a distinct set of feature enhancements over the previous stage (P(x-1)), allowing for iterative development and refinement.

A key aspect of the YPR framework is the 100% syntax and semantic compatibility between each stage (Px) and the ultimate version (Pn) of the Prolog system. This property, known as the "Px Compatibility Rule," is crucial. It allows us to assert that a specific stage (Px) is incorrect if it fails to pass the Px Compatibility Rule, indicating that its code does not run in all higher-level stages.

The YPR framework incorporates a JSON schema and a set of semantic rules to establish the validity of YPR YAML documents. These documents define the project structure, architecture, and configurations. While the YPR YAML documents play a critical role in driving the project, they are not standalone languages in their own right. Instead, they leverage the expressive power of YAML to define and manage the evolving Prolog system. The possibility of developing a DSL (Domain-Specific Language) may arise in the future, but for now, the focus remains on leveraging YAML.

In summary, the YPR framework combines YAML, Python, and Rust to develop a Prolog implementation. Python and Rust independently implement functionally equivalent units for each stage of the Prolog-lite sequence, while YAML serves as the orchestrating language, defining and monitoring the project's architecture and facilitating the coordination between Python and Rust implementations. The framework ensures syntax and semantic compatibility, provides language references, and utilizes JSON schema and semantic rules to validate YPR YAML documents. While a DSL may be considered in the future, YAML remains the primary choice for leveraging its capabilities within the YPR framework.