Imperial College London
Department of Computing

# NoLog: Declarative Logic Programming for Python

Dragoş-Cristian Dumitrache

June 2017

Supervised by: Professor Susan Eisenbach

Second Marker: Doctor Fariba Sadri

To the Godfather and my uncle, Cristian Stănescu, for introducing me to computers so long ago, and without whom none of this would have been possible

# Abstract

Logic programming requires a higher level of formalism and training to be performed efficiently. For a competent software engineer, picking up a language should be a matter of days thanks to the shared similarities in programming language design. However, logic programming comes with different restrictions, behaviour or challenges, and abstracting most of the language is not as straightforward. Furthermore, due to its highly specialised design, tasks that may appear simple in common languages can be quite difficult to achieve.

One popular solution is represented by APIs between Prolog and other languages, also known as bridges. However, these only provide a less than ideal solution. The feedback loop caused by passing a request, as well as the state and necessary context information to obtain a result will lead to increased overhead due to changing processes mid-computation. For Python, there are a few frameworks that perform logic programming, but they are either as complicated as Prolog to use, or they simply have not been maintained for current versions of Python.

NoLog is a Python implementation of Prolog whose purpose is to add support for the logic programming paradigm to Python, strengthening the language without the overhead offered by current solutions. Facts, rules and queries can be introduced, including the globally and existentially quantified variables required by unification. Resolution and unification algorithms are implemented from scratch, adapted to a more Pythonic way and with several optimisations for performance, while preserving the expected behaviour.

# Acknowledgements

I would like to thank:

- Professor Susan Eisenbach, my supervisor and tutor, for all her help and guidance, both during the development of NoLog as well as during my academic years.
- Doctor Fariba Sadri, my second marker, for her feedback and assistance developing NoLog, as well as for her patience when I stubbornly kept implementing my unusual ideas.
- Sorina-Marlis Ştefan, my girlfriend, for her support and feedback and for putting up with me throughout this project.
- My grandparents, Dumitru and Maria Stănescu, and parents, Cristian and Antoanetta Dumitrache, for encouraging me to study mathematics early on, opening up the way for reasoning and logic.
- My uncle for his support, for introducing me to my first computer, teaching me to regard it as a tool, not a toy, and without whom none of this would have been possible.

# Contents

# List of Illustrations

## Figures



## Tables

# Listings

# 1. Introduction

There are two types of programmers: Prolog programmers and everyone else. This statement aims not to belittle Prolog's power, nor its users, but rather to separate two different aspects of software development.

In a research focused environment based on a strong logical background, most developers will be familiar with Prolog. In an industrial environment, on the other hand, Prolog has very few use cases, not because the language is lacking in feature or power, but because it provides a unique, yet specialised approach to solving problems. Prolog is a declarative logic programming language, initially a product of first order logic, meaning it does not require explicit instructions to reach a solution. It only requires a representation of the problem in logic and the necessary assumptions to be able to infer the correct answer. By contrast, Python needs to be explicitly told how to reach a solution. Both approaches have with their own benefits, but also their own shortcomings. While Prolog is quicker at generating new information from existing rules, Python is better suited for handling mathematical and algorithmic tasks and problems.

## 1.1. Motivation and use cases

Over the past years, Prolog has evolved to allow for more than just logic programming. It is even the case that a software engineer may choose to build a web application entirely in Prolog. However, usage numbers have not increased dramatically due to the plethora of alternative languages that people can use without changing the way they reason about software.

Similar to Prolog, Python is a high-level programming language featuring a dynamic type system, imperative loops and recursion as well as various functional components. It was built to enable programmers to create shorter and

**Listing 1.1** Path Prolog code example

```
1 arc(a, b).
2 arc(b, c).
3 path(X, Y) :- arc(X, Y).
4 path(X, Y) :- arc(X, Z), path(Z, Y).
```

more readable code than other languages while still offering support for multiple programming paradigms. However, despite being paradigm agnostic, Python offers no out-of-the box support for logic programming. Its popularity has remained rather constant between 2012 and 2014 and is currently the main language in approximately 8.5% of the public GitHub repositories, ranking in $3^{rd}$ after JavaScript and Java, whereas Prolog is not even in the top 30% [11].

Without any familiarity with Prolog, one can infer the meaning of the code in listing 1.1. It defines two facts in lines 1 and 2, stating that there are two arcs between nodes a and b and respectively b and c. Lines 3 and 4 define two recursive clauses. More specifically, line 3 declares that there exists a path from X to Y if there is an arc connecting X and Y. This is equivalent to the base case of a recursive function. Line 4 describes the recursive case of the predicate path, declaring that a path exists between the variables X and Y if there exists a node Z such that there exists an arc between X and Z, which in turn describes that there is a path from X to Z via the basecase, and furthermore, that there is also a path between Z and Y.

In the listing 1.1 example, X and Y are the equivalent of universally quantified variables from first-order logic, while Z is an existentially quantified variable. Values are assigned to them using Prolog's unification algorithm, which searches the database; if the body of the clause is satisfied, the new fact represented by the head of the clause is considered a provable fact.

I have previously mentioned that Python is better suited for mathematical and numerical problems due to its design. As such, when working with what is virtually a graph, the most straightforward solution is be to represent the graph using an adjacency matrix. An adjacency matrix is a matrix filled with zeroes and ones, such that if the element at indexes i and j is 1, there is an arc

**Listing 1.2** Python path code example

```
1  def get_path(start, end, path):
2      path += [start]
3      if start == end:
4          return path
5      connections = getpossiblepaths(start)
6      for c in connections:
7          if c not in path:
8              newpath = getpath(c, end, path)
9              if newpath:
10                 return newpath
11         return None
12
13 def get_possible_paths(start):
14     return [i for i in range(len(paths[start]))
15             if paths[start][i] == 1]
```

between the two nodes. The equivalent Python algorithm solving the same problem as listing 1.1 is in listing 1.2 and the implementation is very simple, albeit slightly more verbose. However, it provides a good example of the main difference between Prolog and Python. If in a procedural programming language we have to specify a series of steps that the program has to follow in order to reach a solution (as seen in listing 1.2), a declarative language allows us to merely state which solution we want, but not necessarily how to get it, as was shown in listing 1.1. This allows for an increased level of freedom and allows developers to focus more on solving the problem rather than teaching the computer how to do so.

## 1.2. Contributions

NoLog's aim is to add native Python support for the logic programming paradigm in a way which appeals to both Python and Prolog developers alike. To do so, several different concepts need to be implemented and added into Python to bridge the gap between the two languages and their users. All subtasks are quite challenging on their own and they require a large amount

of background research into some well-known algorithms with the addition of some novel ones for managing and running logic code in a Pythonic way as well as for parsing and converting Prolog code into its Python equivalent. The final result aims to be a framework capable of performing querying and solving logical problems through the following components:

**Knowledge base** Inspired from the concept of knowledge bases, when clauses with no constraints are declared, they are immediately added into the knowledge base, a map structure with keys represented by predicate names and values as lists of all possible facts gained. This ensures that the lookup time of all values satisfying a clause is always negligible as it is simply a lookup in a map structure [24].

**Rules and facts** Predicates are one of the building blocks of logic programming and they represent the way a developer describes a program. They can be split into facts and rules. Facts are clauses with no constraints and no variables, and they immediately get stored into the knowledge base. Rules define more general behaviour and have to be dealt with differently. There may not be a default instantiation of variables in a rule to compute a fact, or a rule may simply be true for an infinite number of possibilities. Both types of predicates are necessary to improve expressivity and computational capabilities [28].

**Negation as failure** Rules generally have some constraints. The ability to have negated predicates as constraints greatly increases the number of scenarios that can be expressed into rules, making the development process smoother. They allow for natural translation of specifications into the logic program that NoLog will execute [25].

**Extended list pattern matching** Prolog uses the same head-tail approach when representing lists as functional programming languages. A list is made up of a head and tail, and recursing down the tail, as long as it is not empty, there will be a head and another tail. Unlike Prolog, which uses recursion instead of loops, Python is an imperative language and a list is represented as a collection of items rather than a head and a tail. While Python does allow a list to be unpacked into a head and tail form, the syntax is verbose, unlike the implicit pattern matching featured in Prolog. Leveraging that ability allowed me to write a wrapper around `list` and use it to effectively

unpack a list into head-tail or more complex representations [13].

**Dynamic variables** In order to compute unification effectively, Prolog variables are immutable, which means that once assigned a value, it cannot be modified, only accessed. To be able to do that, Prolog assigns a unique identifier to each variable upon declaration i.e. _G65, _G1805, identifier which is later used to know which variable the program is referring to. To be able to do this, NoLog aims to store variable values in a dictionary, where the key will be the unique identifier generated upon variable initialisation. For each query, the dictionary maps the readable variable names i.e. X, Y, Z to their unique identifiers to ensure the proper value is used [6, 17].

**Prolog unification** When performing unification, Prolog employs a left-to-right strategy. If the query is a fact, it checks if it exists in the knowledge base and attempts to unify all variables in it. If the query is represented by a clause, the head of it is unified with its constraints, and they in turn are unified in order to find a solution that validates all of them. If a solution is found, the program execution is paused and the solution is returned. Should another solution be desired, the program can resume, backtrack up the resolution tree to the last decision point where there exists another unexplored option, and compute the following one. If no other backtracking branches are left, the program terminates and no more solutions can be found.

Unification is at the heart of logic programming as it is what sets it apart from all other paradigms and makes it possible to declare programs in a more natural way than Python currently allows [29].

**Answer Set inspired unification without backtracking** Personally, I believe that backtracking has a history of inefficiency and suffers from scaling difficulties. I believe that a unification approach inspired by answer set programming, combined with the functional power offered by Python, might be more efficient. Answer Set Programming is a different paradigm of logic programming. Its aim is to compute an entire model for an entire defined program. The syntax is similar to that of Prolog, but imposes a few constraints. It computes the solution for all queries and rather than validating a single one, it aims to validate all of them at runtime.

Since the purpose of this project is not to implement an Answer Set Solver

like Clingo [1], but rather Prolog, it would not make sense to implement it just like that. However, intuition has led me to believe that trying a similar approach, but have it limited to a single query might improve efficiency. Therefore, instead of looking for a single solution to a certain query, our aim is to compute all the possible solutions for that query, leaving as a later design decision the manner in which they will be presented to the user [26].

**Knowledge engine** While it was not a priority initially, as statements could still be evaluated sequentially, when dealing with general rules, the need for delayed execution became evident. Predicates could no longer be executed once and forgotten, as they might not necessarily offer any facts to be used in the remaining computations. The knowledge engine solved that issue, together with the constraint that predicates had to be defined prior to usage by storing rules and delaying the evaluation until convenient for the developer, usually at the end of the program definition.

## 1.3. Report structure

This document is structured into 7 chapters, detailing the motivation and current solutions, the implementation and design processes as well as the evaluation of those results. Chapter 1 expresses the motivation and the contributions to NoLog. Continuous to it, Chapter 2 and Chapter 3 detail the relevant background information relevant to NoLog as well as current solutions with their advantages and disadvantages.

The next chapters present the requirements and the reasoning behind design and implementation decisions. The aim is to provide insight into the challenges faced while avoiding the shortcomings of current solutions. Thus, Chapter 4 sets apart requirements and describes the design process and overall evolution of NoLog while Chapter 5 offers implementation details and discusses the algorithms used. The goal of these two chapters is to showcase the evolution of NoLog as more and more features were added.

Finally, Chapter 6 discusses evaluation from both the perspective of users and that of future developers, as well as the testing performed during the development process. Chapter 7 concludes the report by summing up the achievements and providing some interesting ideas for further development.

# 2. Background and Research

## 2.1. Introduction into logic programming

### 2.1.1. Components of a logic program

We have already seen an example of what Prolog looks like in 1.1. At the most basic level, Prolog uses two different concepts to represent everything: statements and terms. On their own, statements are categorised into facts, predicates and queries which, in turn, are combinations of terms. A collection of those three constructs represents a knowledge base that links facts together using predicates, while queries are used for interacting with the knowledge base and extracting new information from it [17, 24].

Terms are used to represent atoms, numbers and variables and can be combined to make up more complex terms. A more complex term is made up of a functor and a series of arguments. For example, `father(X, Y)` is a functor stating that `X` is the father of Y [28].

### 2.1.2. Queries

The sample Prolog code in listing 2.1 offers an example of what a query looks like. The definition of our program is from lines 1 to 5, while from line 7 onward our queries and their results are visible. If for instance we query the program with `son(thomas, patrick)`, we will get as result true, since `son(thomas, patrick)` can be proved via unification from the rule defining son. However, if we perform a more general query such as `son(X, Y)`, via unification, the first result will be `X = bruce`, `Y = nora`, since that is already defined in the knowledge base, while the second result will consist of the unified solution `X = thomas`, `Y = patrick`.

**Listing 2.1** Example of a Prolog query

```
1  son(bruce, nora).
2  male(bruce).
3  male(thomas).
4  parent(patrick, thomas).
5  son(X, Y) :- parent(Y, X), male(X).
6
7  ? - son(thomas, patrick).
8  true.
9  ? - son(X, Y).
10 X = bruce,
11 Y = nora;
12 X = thomas,
13 Y = patrick.
```

### 2.1.3. Negation as failure

Negation as failure is a powerful computational feature in Logic programming. It refers to the event where all attempts to prove a predicate `p` will fail finitely, making `not p` valid [25]. A simple example is presented aiming to prove goal `p` is offered in Figure 2.1.

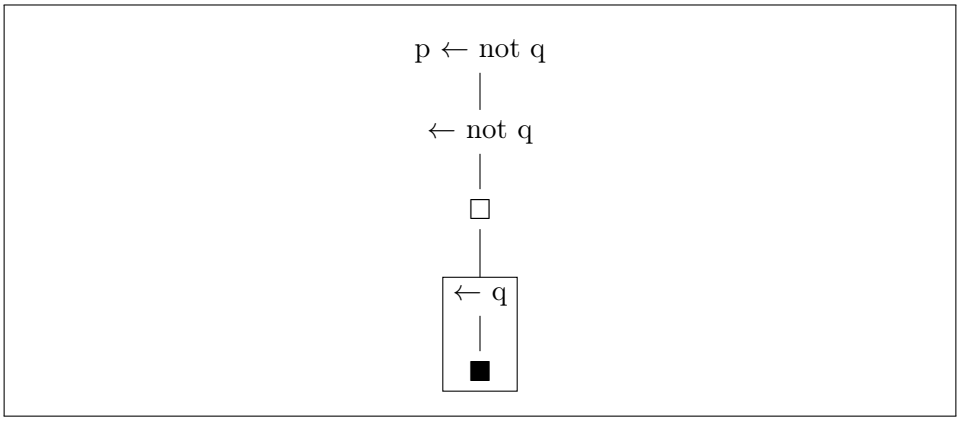


Figure 2.1.: Negation as failure example

## 2.2. Unification algorithm

We have briefly offered an explanation of unification, but we will now go into more detail. Taking the simplest definition of unification in the field of Knowledge Representation and logic, $t_1$ is a common instance of $t_2$ if there exists substitutions $\theta_1$ and $\theta_2$ such that t = $t_1\theta_1 = t_2\theta_2$. [29]. If that is the case, we say that $t_1$ unifies or is unifiable with $t_2$.

The Prolog implementation of the unification algorithm computes the most general unifier or mgu of two terms and is based on solving equations.

**Definition. Most general unifier**
$\theta$ is a most general unifier of $\alpha$ and $\beta$ if and only if it satisfies the following constraints:
1. $\theta$ is a unifier of formulas $\alpha$ and $\beta$, meaning $\alpha\theta = \beta\theta$
2. For any other unifier $\sigma$ of $\alpha$ and $\beta$, $\alpha\sigma$ is an instance of $\alpha\theta$ and similarly, $\beta\sigma$ is an instance of $\beta\theta$. [30]

The unification algorithm takes as input two terms, $T_1$ and $T_2$ and outputs the most general unifier if they unify, or returns failure otherwise. It uses a LIFO stack to store the equations that need to be solved, as well as an initially empty substitution list $\theta$. The stack is initialised to contain the equation $T_1 = T_2$. The algorithm then loops and pops each value from the stack, processes it and terminates if the stack becomes empty or a failure occurs during the processing phase.

Take `S = T` to be a popped equation from the stack. There are several cases for dealing with it. The first one is if S and T are identical constants or variables, which means the equation has been satisfied and nothing needs to be done for it, continuing the computation by popping the next equation from the stack.

The next case is if S is a variable and T is a term with no occurrences of S. Then, the stack is searched for all occurrences of S and they are replaced with T. Similarly, all occurrences of S in $\theta$ are also replaced by T. Then the substitution S = T is added to $\theta$. Note that T must not contain S, otherwise the algorithm would loop. Checking for this is known as the occurs check.

**Listing 2.2** Unification algorithm pseudocode

```
1  Input: Two terms, T_1 and T_2
2  Output: θ, the mgu of T_1 and T_2
3  Algorithm: Initialize θ to be empty, the stack to
       contain T_1 = T_2  and failure to false
4  while stack not empty and not failure do:
5    pop X = Y from the stack
6    case
7      X is a variable not occurring in Y:
8        substitute Y for X in the stack and
9        add X = Y to θ
10     Y is a variable not occurring in X:
11       substitute X for Y in the stack and
12       add Y = X to θ
13     X and Y are identical constants or variables:
14       continue
15     X is f(X_1,...,X_n) and Y is f(Y_1,...,Y_n)
16       for some functor f and n > 0:
17         push X_i = Y_i, i = 1...n, on the stack
18     otherwise:
19       failure is true
20 if failure:
21   return failure
22 else:
23   return θ
```

An example of such looping would be if we were to try to solve `X = father(X)`, where the predicate `father` represents the idea that X is a father. As such, we would assign `X` the value `father(X)`, but then we would need to solve `father(X) = father(father(X))`.

If however T is a variable and S a term with no occurrences of T, the symmetric set of steps as above is executed.

If S and T are compound terms, they unify if they have the same functor and arity. Calling our functor f, with arity of n, the two terms unify by pushing into the stack n pairs of equations $S_n = T_n$ and solving those. If they all unify, a solution is found and S and T unify. In any other cases, failure is reported and it terminates.

Given the code in listing 2.1, the unification for that query is represented as a graph in figure 2.2. Since our query is `son(X, Y)`, it unifies that with the head of the clause that defines it and assigns hidden IDs to the variables `X` and `Y`. As such, we get a solution `X = _G5 = Bruce`, `Y = _G6 = Nora`. To get the other solution, the algorithm backtracks and replaces the goal with the conditions from the body, the query now becoming `parent(_G6, _G5), male(_G5)`. From this step, it unifies them left-right to right, but since the second unification term, `male(_G5)`, has no other variables, `_G5` will have a value assigned and it will only need to be evaluated to true. The solution it finds is `_G5 = thomas`, `_G6 = patrick` and because no further solutions can be found, the execution terminates.

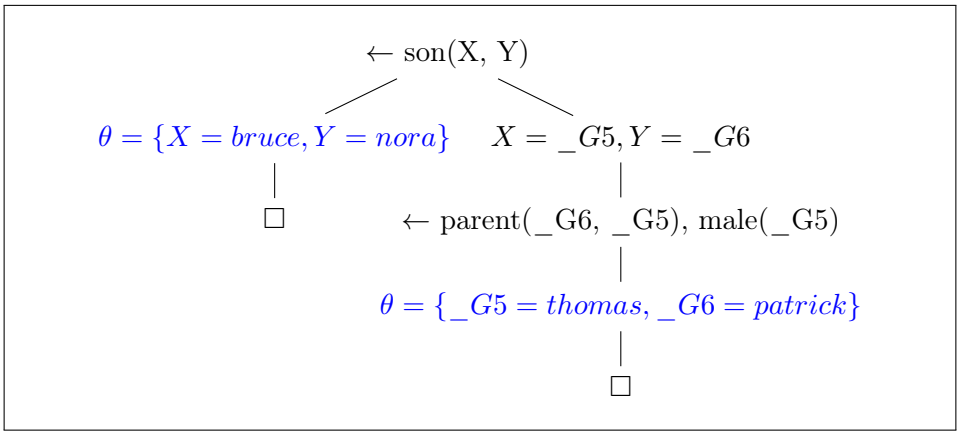


Figure 2.2.: Unification Example

## 2.3. Answer Set Programming

A program `P` is a collection of facts and predicates. Given one such program, the grounding of P, notated as `ground(P)`, is constructed by replacing each rule `R` in `P` with its grounded equivalent. In logic, to ground a term means to instantiate all the variables it contains. As you may imagine, a program `P` may have infinite groundings, and computing all of them is, if not impossible, at least unnecessary. Answer Set Programming (ASP) solvers will not generate all groundings at random. They will instead choose to ground a rule R incrementally, such that all atoms in the positive elements of the

**Listing 2.3** Grounding of a program

```
1 ground(P) = { arc(a, b), arc(b, c),
2               path(a, b), path(b, c), path(a, c) }
```

**Listing 2.4** Example of mutually exclusive rules

```
1 p :- not q.
2 q :- not p.
```

body of R[1] already represent the head of another ground rule. Given the example Prolog code in listing 1.1, by gradually grounding the rules, we obtain `path(a, b), path(b, c), path(a, c)`. Adding the already grounded facts to that, our program's grounding is the one represented in 2.3. That represents P's answer set or stable model. In this case it was simple as there were no mutually exclusive rules and our answer set contained a single entry. However, consider the program in 2.4. It has two stable models, `p` and `q` and without any more information, either one is admissible. By now, you should have an intuition at least that answer sets represent all possible solutions of a program. [18]

Going back to our example code in 1.1, if our query was `path(X, Y)`, the algorithm would simply have to retrieve all possible solutions by looking up the list of values in the knowledge base, using as key the goal clause name, `path`. No other computation is required as when the rules in line 3 and 4 have been defined, all the other necessary computations have been run. Initially, when Line 3 was executed, the body of it would be run before, such that all possible solutions for `arc(X, Y)` are retrieved. Afterwards, if the found `path(X, Y)` pairs have not been previously found, they will be added as facts in the knowledge base, considering that they have just been proven. For the second example, the same is valid for `arc(X, Z)` and `path(Z, Y)`. However, the resulting two lists would have then been filtered to find the common elements around Z, and for any solution found, the instantiated

[1] Elements not defined using negation by failure, a topic that will be discussed in more detail further on

pair `(X, Y)`, will be added under the path knowledge base. The advantage of this approach was the lack of backtracking which I hoped would lead to better efficiency. That was the original plan, but filtering the resulting two lists proved a less elegant solution requiring extra computation than initially anticipated, so that course of action has been discarded in favour of a more Prolog-like approach.

# 3. Current solutions

Before we can dive into the requirements and design of NoLog, we should first have a look at current solutions available. As may be expected, this is quite a popular problem, since many developers would like to leverage Prolog's capabilities while still holding on to Python's. As such, there are numerous solutions out there from bridges passing information back and forth across the two languages to already existing Python implementations of Prolog. We will go over some of the most popular solutions, aim to learn from mistakes of the past and to extract a list of requirements showcasing the most fundamental features that NoLog must offer.

## 3.1. Jitting VM for Prolog

One current solution available at the moment is the use of PyPy [20], an alternative implementation of Python offering a just-in-time compiler. The JIT compiler identifies and optimises instructions which get executed multiple times by converting them to an architecture-dependent Assembly code. A group of researchers at the Universität Düsseldorf in Germany has implemented a continuation-based Prolog interpreter using the PyPy JIT RPython compiler. The aim of this approach is to optimise repeating Prolog instructions and convert them to Assembly code. It also features the introduction of garbage collection techniques all the while yielding a higher performance than traditional ahead-of-time compilers [14].

Regarding implementation details, it makes use of the object-oriented paradigm, having defined Prolog objects by instances and subclasses of the PrologObject base class. A representation of simple non-variable terms is also offered in classes of their own, i.e. Atom, Number, Float, which simply encapsulate the existing behaviour of the String, Integer and Float classes. The point

of interest and novelty from a Pythonic perspective is represented by the Var class which features a binding field initially set to NULL. If a variable becomes bound during the execution of the program, the binding field is set to the bound value.

Unification follows the same object-oriented paradigm, where each PrologObject has a unify method which takes another PrologObject as an argument and a Trail. The Trail is simply a linked-list structure, where each trail point is linked to its following oldest predecessor. If a variable becomes bounded during the current execution, it is added to a list within the current Trail object containing all variables whose binding requires undoing in the event of backtracking.

In order to implement a continuation-based interpreter, the current state is represented as two Continuation objects, one for success and one for failure. While the success continuation contains the remaining program to be executed, the failure continuation represents the code that needs to be re-executed should backtracking occur.

In order to evaluate the efficiency of their interpreter, testing has been performed both on the interpreter as is, as well as with the JIT compiler, and the JIT compiler solution has a clear advantage over the simple interpreter as can be seen in the two tables below. However, while the JIT compilation offers significantly better results to the simple interpreter, it seems to still fall short compared to the classic Prolog implementations available, perhaps due to the optimisations performed by the JIT compiler. Furthermore, the original development team had to implement most of Prolog's built-ins, meaning that if a built-in function was missing, developers would have no other choice than to implement it themselves. Since it is essentially a standalone platform, leveraging Python's built-in functions is nearly impossible without altering the execution.

As a standalone platform, its syntax is identical to Prolog. For all intents and purposes, it actually is Prolog with a different backend implementation aiming to offer out of the box optimisations. Therefore, it does not do much to address the issue of performing logical programming with Python. A syntax example for checking whether a value represents a natural number is shown in Listing 3.1.

**Listing 3.1** Example of iteration code using the JVM Prolog implementation

```
1 iterate(0).
2 iterate(X) :- Y is X - 1, iterate(Y).
```

Table 3.1.: Benchmark times for classical Prolog benchmarks

| **Benchmark** | SICStus interpreter | SICStus compiled | SWI |
|---|---|---|---|
| **arithmetic** | 3630 ms | 490 ms | 1080 ms |
| **boyer** | 490 ms | 40 ms | 100 ms |
| **chat_parser** | 20930 ms | 5050 ms | 9030 ms |
| **crypt** | 1810 ms | 70ms | 470 ms |
| **deriv** | 1200 ms | 420 ms | 190 ms |
| **nrev** | 820 ms | 60 ms | 180 ms |
| **primes** | 2230 ms | 190 ms | 380 ms |
| **qsort** | 1320 ms | 160 ms | 440 ms |
| **queens** | 8930 ms | 460 ms | 1880 ms |
| **reducer** | 6610 ms | 930 ms | 2710 ms |
| **tak** | 720 ms | 20 ms | 80 ms |
| **zebra** | 2480 ms | 1050 ms | 2400 ms |

Table 3.2.: Benchmark times for classical Prolog benchmarks

| **Benchmark** | Simple interpreter | JIT-cold | JIT-warm |
|---|---|---|---|
| **arithmetic** | 2990 ms | 260 ms | 250 ms |
| **boyer** | 980 ms | 20340 ms | 17040 ms |
| **chat_parser** | 33940 ms | 15900 ms | 15830ms |
| **crypt** | 740 ms | 70 ms | 770 ms |
| **deriv** | 2720 ms | 880 ms | 880 ms |
| **nrev** | 760 ms | 350 ms | 360 ms |
| **primes** | 1730 ms | 830 ms | 830 ms |
| **qsort** | 2040 ms | 3350 ms | 340 ms |
| **queens** | 4100 ms | 310 ms | 310 ms |
| **reducer** | 29650 ms | 12150 ms | 5320 ms |
| **tak** | 610 ms | 230 ms | 230 ms |
| **zebra** | 5720 ms | 3020 ms | 3000 ms |

## 3.2. Bridge implementations

### 3.2.1. InterProlog for SWI-Prolog

InterProlog is a Java-Prolog bridge, offering an SDK that allows both-ways communication between the two languages. As such, it allows Java developers to run Prolog code in their applications, while allowing Prolog programmers to use any JVM available. The code in 3.2 is a Prolog-sided call to Java, passing as input arguments the String `"InterProlog Java bridge"`, the variable in which it should be stored in the form of a String as well, Middle, and the desired output, which is the a substring of the input. The returned value would be Middle = Java [12].

For the Java side of things, performing Hello World in Prolog would look something similar to this, omitting imports and Prolog Engine declarations [5]. As can be clearly noticed, this is a lot of code for something as simple as `hello :- write('Hello world!')`.

While the Prolog side of it shown in 3.2 looks simple enough to use, that does not address the issue of adding the declarative features of a logic programming language to an imperative one. On the Java side of things, the code seen at 3.3 looks unnecessarily complicated compared to its Prolog counterpart, regardless of the name parameter introduced into the Hello World string. Another aspect of why bridges do not represent an adequate solution consists the overhead of calls, since jumping from Java to Prolog and vice versa takes more time as the size of the query grows.

**Listing 3.2** Prolog-side Java call

```
1 java( string('InterProlog Java bridge'),
2       string(Middle),
3       substring(int(12),int(16)) ).
```

**Listing 3.3** Java-side Prolog call

```
public static void main(String args[]) {
    PrologEngine engine = new NativeEngine();
    // Only for XSB Prolog
    engine.command("import append/3 from basics");
    Object[] bindings = engine.deterministicGoal(
                    "name(User,UL),
                    append(\"Hello,\", UL, ML),
                    name(Message,ML)",
                    "[string(User)]",

   System.getProperty("user.name"),
                    "[string(Message)]");
    String message = (String)bindings[0];
    System.out.println("\nMessage:"+message);
    // the above demonstrates object passing both
   ways;
    // since we may simply concatenate strings, an
   alternative coding would be:
    bindings = engine.deterministicGoal(
                 "name('" +
   System.getProperty("user.name")
                  + "',UL),append(\"Hello,\", UL,
   ML),"
                  + " name(Message,ML)",
                  "[string(Message)]"
              );
    // (notice the ' surrounding the user name,
   unnecessary in the first case)
    System.out.println("Same:"+bindings[0]);
    engine.shutdown();
}
```

**Listing 3.4** Python-side Prolog query

```
1  from pyswip import Prolog, registerForeign
2  def hello():
3      print "Hello, World!"
4
5  hello.arity = 0
6
7  registerForeign(hello)
8
9  prolog = Prolog()
10 list(prolog.query(hello()))
```

### 3.2.2. PySWIP

Having looked at the Java bridge, we can now better appreciate Python bridges. PySWIP is an implementation of such a bridge allowing for less verbose code to print "Hello World!". However, it was never fully finished and features only a partial implementation SWI-Prolog's foreign language interface. The foreign language interface aims to provide the basic behaviour of executing Prolog queries and converting functions, the constructs most languages are based on, into clauses. That incompleteness combined with the fact that it has not been updated to support current versions of Python makes it a less than ideal choice. A counter example of the Java to Prolog Hello World can be observed in 3.4. As you can see, it uses a Python function into the CTypes library, which adds it into C dynamic library. The C dynamic library is then used by the SWI-Prolog's foreign language interface to convert some of the functions into their Prolog equivalent. However, for more complex cases where predicates will need to contain variables, the arity of the predicate will always have to explicitly stated, adding unnecessary requirement steps for developers [8].

## 3.3. Current Python Implementations

As we are approaching the topic at hand, there are various Python implementations of engines that can process Prolog code. Since they are quite similar, we will only briefly analyse them to see their pros and cons.

### 3.3.1. PyDataLog

PyDataLog [7] makes use of a JIT compiler as discussed previously. However, we have also seen that performance-wise, it is still not matching the efficiency of native Prolog implementations. Its pros include the ease of use and similarity to Prolog, making it a viable option for both target groups of developers, despite obvious performance drawbacks.

### 3.3.2. Pyke

Pyke is another Python implementation of Prolog, more complete than PyDataLog as it defines its own knowledge engine. It is only meant to complete Python, not compete with it. It uses partial functions to be able to define Prolog-looking queries and supports both forward and backward chaining rules. However, unlike Prolog, it splits facts, clauses and queries into different knowledge bases and stores them separately, being less dynamic when compared to Prolog, as queries cannot be typed at run-time, but rather have to be defined ahead of time. An example similar our Prolog family code from listing 2.1 would look something similar to listing 3.5.

While its look is the one most closely matching Prolog, it functions noticeably different, which may not appeal to our target base. Furthermore, it has

**Listing 3.5** Pyke declarations

```
1 male(bruce)
2 male(thomas)
3 son_of(bruce, thomas, nora)
4 daughter_of(shirley, david_r, sarah_r)
```

not been maintained in several years and support is sparse. It defines multiple concepts, including the knowledge engine, to handle all the computation, making it more difficult to scale with future versions of Python. In terms of evaluation, no information was offered and given the scarce documentation available, the library was counter-intuitive to use, as it was defining its own syntactic rules.

## 3.4. Conclusion

Throughout this chapter we have reviewed some of the most popular options available and are now in a better position to understand the challenges of the task. There are a number of requirements we should aim to satisfy, which we can split into two different categories: aesthetic and functional. Some of the solutions did a good job at addressing aesthetic requirements, such as PyDataLog [7] or Pyke [21]. Other solutions focused solely on the functionality, among which we can count PyDataLog [7] or PySWIP [8]. However, none of the solutions successfully addressed the issue of bridging the gap between Prolog and Python. Some were too verbose, like InterProlog [12], others too similar to Python, such as PySWIP [8]. Finally, there some which defined their own computation engine, but were simply too different to either Prolog or Python, as was the case with Pyke [21].

One of the main aesthetic requirements includes syntax similarity to both Python and Prolog, to appeal to both our target user bases. The ability to preserve one of Python's best features, its lack of verbosity, also falls within that category. Retaining the ability to write short, elegant, code means requiring as little boilerplate code as possible. In terms of functionality, it is imperative not to alter the Python interpreter or its execution, to allow for easy porting of NoLog to future versions. From a functional perspective, we need to implement several features which are at the root of logic programming: predicates, unification, negation as failure, pattern matching, rules and querying. Last, but not least, we aim to be able to integrate Python with NoLog seamlessly, so that we may build applications using logic programming for obtaining new information in the backend, and build a front-end in Python.

# 4. Requirements and design

This chapter aims to deal with the list of requirements as outlined at the end of the previous one, as well as the different decisions that affected NoLog's design. The design and development process was split and driven by various features of Prolog, and we will cover them in the order in which they were implemented, across the multiple versions.

## 4.1. Predicates

So far, we have seen several examples of predicates in the Prolog code showcased in 1.1, 2.1. Analysing them, we identify several components. At the highest level of abstraction, a predicate is made up of a head and an optional body. The body represents constraints on the values for which the head holds. It can be thought of as the logical implication `body` $\rightarrow$ `head`. Delving deeper, a head is a complex Prolog term, meaning it is made up of a functor and its arguments. There are no constraints on the number of arguments, in this case resembling function definitions in Python.

The main challenge was finding an accurate representation of clauses that Python could efficiently work with, but that also included all the components of a Prolog predicate. Naturally, Python has no such concept, the closest representation being that of a function. While a predicate is a construct that adds some logical meaning to a Prolog program, a Python function is a container for a block of code and aims to add reusability. Based on that definition, an empty function will not be in any way useful, and Python will return a syntax error for it. To preserve similar behaviour and be useful, NoLog needs to be capable of doing the same. The original design of clauses can be observed in 4.1. However, a single function trying to perform the entire computation did not scale very well, so a different solution had to be

**Listing 4.1** Example of NoLog Python Code

```
1  def clause(*args, **kwargs):
2      ...
3
4  clause('arc', ('a', 'b'))
5  clause('arc', ('b', 'c'))
6  clause('path', (Var('X'), Var('Y')),
7          body=[clause('arc', (Var('X'), Var('Y')))])
8  clause('path', (Var('X'), Var('Y')),
9          body=[clause('arc', (Var('X'), Var('Z'))),
10               clause('path', (Var('Z'), Var('Y'))])
```

found. Another issue was the clause taking a dynamic number of arguments passed into `*args` and `**kwargs`, which made the parsing of those arguments difficult. There was no way of telling apart the string that represented the predicate name from a parameter without cluttering the code. Furthermore, those two sets of arguments played different roles. Where `*args` was used to pass in the predicate name and its arguments, `**kwargs` was used to pass in the body of the predicate, as can be seen in 4.1.

In order to address the issue of parsing arguments and distinguishing between the predicate name and the rest of the parameters, for any given number of arguments, the solution agreed upon was the passing of a function instead of a string. Doing so meant that we would require a function representing our predicate to be defined. The body of the function was irrelevant for the purpose of its definitions, as the function itself was never called. What mattered was the pointer to the function, as can be seen in 4.2. However, while this approach offered a similar syntax to Prolog's predicate definitions, it resulted in boilerplate code, thus breaking one of our requirements. Furthermore, it did nothing to improve scalability.

Soon after, that approach was discarded entirely and instead, a predicate function was implemented to perform the responsibilities of any general clause. The definition of arcs and paths from 4.1 can now be written as in 4.3, essentially removing the need for unnecessary code. Working on general clauses as opposed to specific ones greatly improved robustness and

**Listing 4.2** Clause definition example

```
# General clause skeleton definition
def clause_name(*args, **kwargs):
    pass

def son(X, Y, **kwargs):
    pass

clause('son', ['Bruce', 'Nora'])
clause('son', [Var('X'), Var('Y')],
        body=[clause('male', [Var('X')]),
              clause(parent, [Var('Y'), Var('X')])
    ])
clause('path', [Var('X'), Var('Y')],
        body=[clause('arc', [Var('X'), Var('Y')])])
clause('path', [Var('X'), Var('Y')],
        body=[clause('arc', [Var('X'), Var('Z')],
              clause('path', [Var('Z'), Var('Y')])
    ])
```

scalability. From this point onward, the definition of a predicate will include the predicate name, optional arguments, and an optional body.

When I initially started developing in Python, the lack of verbosity dealt a heavy blow. Once I got more comfortable with the language, I recognised what I initially perceived as out of place syntactic sugar to in fact represent elegant features. Similarly, Prolog's syntax is even more minimalistic. No apostrophes or quotation marks have to be added around literals and predicates do not need to be called as functions. Prolog succeeds in enabling developers to write logic as if they were sketching a proof. Aiming for NoLog to be as similar to Prolog as possible, from the point of view of Prolog developers, I believe that always calling the predicate method for each definition adds unnecessary noise to a program. Therefore, to add some syntactic sugar and to aid with the visualisation of a program, partial functions were used to define predicate functions of certain names.

**Listing 4.3** Predicate header definition and example

```
1 def predicate(p_name, p_args=None, p_body=None):
2     ...
3
4 predicate('arc', ('a', 'b'))
5 predicate('arc', ('b', 'c'))
6 predicate('path', ('X', 'Y'), [('arc', ('X', 'Y'))
     ])
7 predicate('path', ('X', 'Y'), [('arc', ('X', 'Z')),
8                                ('path', ('Z', 'Y'))
     ])
```

**Listing 4.4** Partial functions example

```
1 arc = partial(predicate, 'arc')
2 path = partial(predicate, 'path')
3
4 arc(('a', 'b'))
5 arc(('b', 'c'))
6 path(('X', 'Y'), [('arc', ('X', 'Y'))])
7 path(('X', 'Y'), [('arc', ('X', 'Z')),
8                   ('path', ('Z', 'Y'))])
```

A partial function takes as input a function together with a subset of its arguments and returns another function which will receive as input the remaining arguments. Thanks to this feature, the code in listing 4.3 can be represented in a cleaner way in 4.4. Unfortunately, the body still has to be defined using the longer representation `(predicate_name, arguments, optional_body)` in order to defer its execution until it gets called. Once it does, it can be passed into the predicate function to get the necessary information. It is, like any syntax sugar, entirely up to the developer to use one way or another. Personally, I am biased towards using partial functions for overall clarity, especially when working with predicates of a high arity. For the remainder of the project, the structure of a predicate will remain mostly the same, so we can now deal with the other requirements.

**Listing 4.5** Variables defined prior to usage

```
1 X = Var('X')
2 Y = Var('Y')
3 Z = Var('Z')
4
5 # the definition of predicates using only X, Y, Z
    in their head and body.
```

## 4.2. Variables

Although NoLog aims to be as Pythonic as possible, the Variable class had to be implemented, as a consequence of Python's lack of support for dynamic variables. At the time, it was a wrapper class defining variables on demand. The constructor received a string representing the name of the variable, and it would get assigned a unique id of the form _G1, _G2 etc., based on the length of the list containing the global variables already defined.

However, creating new objects to represent variables meant that they either had to be defined prior to their usage, or that they would always be defined in place. Overall, the Variable class raised more issues than it was solving. Assuming all variables would be defined prior to their usage, a code example would look similar to 4.5. If on the other hand they were to be defined in place, like in 4.1, despite wrapping the same variable name, they would be different objects, requiring the hash to be modified. A design decision led to upper case strings representing variables, while lower case strings represent atoms, making it a familiar concept for anyone who has used Prolog in the past.

## 4.3. Unification

We have covered the basic functionality of unification and offered the pseudocode in [29] and with a representation of predicates that satisfied our necessary constraints, we are in a position to start designing our unification algorithm.

**Listing 4.6** Original unification header

```
1 def unify(t1, t1_args, kb, t2=None, t2_args=None):
2     ...
```

**Listing 4.7** Current unification header

```
1 def unify(term_1, kb, rules=None, term_2=None):
2     ...
```

While we were initially designing the predicates, an open-source unification library for Python was used. However, it was soon revealed that, despite the fact that it was working most of the time, the results were unexpected and non-deterministic, unification failing all-together occasionally [22]. Such a problem enforced the decision to write my own unification method and, as with predicates, designing it took several iterations.

Unification was done via a single function that unified according to the algorithm outlined in 2.2. Although the algorithm was a standard implementation, some additional type checking had to be performed to deal with the various cases. It is also worth mentioning that at the point when the main unification feature was added and could unify most things, in version 0.0.7, the unification method received a different input than it currently does.

Looking at the header definition in the snippet 4.6, `t1` and `t2` represent the two predicate names, `t1_args` and `t2_args` represent the arguments for the two predicates to be unified, and `kb` represents the knowledge base, a repository of all known facts stored in a dictionary using predicate names as keys. Furthermore, `t2` and `t2_args` are optional arguments. If they are present, it tries to unify them with the arguments of the first predicate, assuming the predicate names unify successfully. However, if they are missing, the function tries to unify the first term and its arguments with facts stored in the knowledge base. It returns those mappings, if successful, or false if the process fails at any point.

The current header implementation showcased in 4.7 is more robust than the original one, which failed for cases where we attempted to unify lists. Instead of taking two terms and their arguments separately, unification now more closely resembles the standard algorithm as it receives two terms and attempts to unify them. Type checking is still performed to deal with different cases accurately. The parameters are almost the same, with the exception of the newly introduced *rules*. We will discuss rules on their own as the problem they posed is worthy of its own section.

## 4.4. Negation as failure

Having defined predicates, a basic unification algorithm and resolution, the next feature to be implemented was negation as failure.

While I originally believed Python's negation might be versatile enough to be used, due to the information returned by the unification algorithm, that was not the case. To add support for negating predicates, the initial approach was to parse the name of the predicate to see if *not* was present. However, manipulating the predicate name and performing that check regardless was just an extra computation that I wanted to avoid. It made the overall execution seem less elegant. A more elegant approach was to extend the predicate function to receive another optional parameter *p_naf* as in 4.8. Its default value is *False*, making no changes to the normal execution of a predicate. However, if p_naf is explicitly passed as *True* it means the negation as failure is activated. Then, the *p_naf* argument acts as the enabling point of a seam, changing the execution in several places [15].

**Listing 4.8** Predicate header with negation flag

```
1 def predicate(p_name, p_args=None, p_body=None,
      p_naf=False):
2     ...
```

## 4.5. Pattern matching on lists

Prolog's representation of lists takes a page from functional programming. Lists are represented as [H|T], which is simply syntactic sugar representing .(H, T), where T is a list in its own right. So .(H, T) = .(H, .(H2, T2)). This can go on until the whole list has been unpacked, at which point unification is performed on it. Although Python offers no support for pattern matching on a list using the head-tail representation, it does offer a way of unpacking a list. The initial drawback was the inability of dynamically creating variables corresponding to the string arguments. A workaround was found using a dictionary to simulate the local scope of unpacking, where each string argument represents a key, as in 4.9.

Python's unpacking mechanism supports functionality beyond the head-tail syntax of Prolog as is demonstrated in 4.10. While in Prolog it does not have any use-cases yet, adding the support for it might introduce uses in the future.

With an elegant solution in mind, all that was left was the implementation. I thought that only creating an alias around lists would be the way to go. The latest versions of Python have added support for aliasing types. Lists can take arguments of any type and the idea was to alias the general list to a wrapper. However, after fiddling with the new type annotations and aliasing introduced in Python 3.6, it seemed I was mistaken. The new features would not solve the dynamic unpacking requirement, nor were they fully functional.

**Listing 4.9** List unpacking example

```
1  # We can unpack head and tail specifically
2  H, *T = [1, 2, 3] ⇒ H = 1, *T = [2, 3].
3
4  # But we cannot unpack them without defining the
       variables first.
5  'H', '*T' = [1, 2, 3] ⇒ error
6
7  dictionary['H'], dictionary['*T'] = [1, 2, 3]
8      ⇒  {'H': 1, '*T': [2, 3]}
```

**Listing 4.10** Various examples of list unpacking

```
1 X, Y = [1, 2] ⇒ X = 1, Y = 2
2
3 H, *T = [1, 2, 3, 4, 5] ⇒ H = 1, T = [2, 3, 4, 5]
4
5 H1, H2, *T, L = [1, 2, 3, 4, 5, 6, 7]
6     ⇒ H1 = 1, H2 = 2, L = 7, T = [3, 4, 5, 6]
```

**Listing 4.11** ListWrap Unpacking

```
1 wrapped_list = ListWrap(['H', '*T'])
2 wrapped_list.unpack([1, 2, 3]) ⇒ H = 1, *T = [2, 3]
3
4 wrapped_list = ListWrap(['H', '*T', 'T2', 'T3'])
5 wrapped_list.unpack([1, 2, 3, 4, 5, 6]) ⇒ H = 1, *T
      = [2, 3, 4], T2 = 5, T3 = 6
6
7 wrapped_list = ListWrap(['H', '*T', 3])
8 wrapped_list.unpack([1, 2, 3, 4,]) ⇒ False
```

The better idea was to implement a wrapper class around the standard list. The constructor of ListWrapper takes as input the list you want to wrap around, and provides a way to unpack an iterable based on the arguments of the wrapped list. It is capable of performing all the possible combinations of unpacking for a list. The only constraint is that there can only be one starred argument. Even so, it should more than satisfy our requirements.

## 4.6. Rules

We have covered predicates. We have covered variables and the constraints they impose. We have covered unification and negation as failure. Covering rules at this point may seem counter-intuitive, and so it was. Predicates are rules, and we have already designed them. However, due to a mistake on my part, I only thought of predicates as having a body definition. I had not

**Listing 4.12** Prolog's implementation for member and append

```
member(X, [X|T]).
member(X, [H|T]):- member(X, T).

append([], X, X).
append([H|T], L2, [H|T2]):- append(T, L2, T2).
```

**Listing 4.13** NoLog's implementation for member and append

```
# Predicate header with seam for recursive rules
def predicate(self, p_name, p_args=None, p_body=
    None, p_naf=False, recursive=False):

member = partial(predicate, 'member')
append = partial(predicate, 'member')

member(('X', ListWrap(['X', '*T']))).
member(('X', ListWrap(['H', '*T']),
       [('member', ('X', ListWrap(['*T'])))]))

append(([], 'X', 'X'))
append((ListWrap(['H', '*T']),
        ListWrap(['*L2']),
        ListWrap(['H', '*T2'])),
        [('append', ('*T', '*L2', '*T2'))])
```

considered being able to write more general rules, such as member or append. While they do not alter the syntax, they are an improvement to the overall design, and we will spend more time going through their implementation, as they have definitely altered the implementation of predicates. The header of the predicate function was also changed, and it now contains another seam's enabling point. A NoLog definition of member and append from 4.12 is given in 4.13. It preserves a Python appearance, but also resembles Prolog's definition once you look past the helper function calls.

**Listing 4.14** Query function header

```
1 def query(self, p_name, p_args, display_flag=False,
      recursive=False):
2     ...
```

## 4.7. Queries

Probably one of the most important requirements, queries are the mechanism that allows the developer to interact with the knowledge base. We have already seen most of the arguments that the query function takes in 4.14 in previous predicate definitions 4.13. The only different one is the `display_flag`, which has the simple task of choosing whether to display the output of a query to the console the same way Prolog does or skip that and simply return the answer.

## 4.8. Granularity

Moving away from the technical requirements, comes the question of usability. Would developers be required to write a huge block of code detailing the entire NoLog program, or would they be able to finely intermix NoLog and Python? Ideally, we chose the latter to allow for development of finer points. Since NoLog is still very much Python, intertwining the execution of either is a natural development. Once the knowledge engine is declared, developers can write a single predicate and evaluate it, or perform queries, and use those results to perform some Python computation. They can perform this gradually until they have a working solution. They can also use pythonic features like list comprehensions when defining arguments, as well as declaring arguments into variables and passing the reference to the variable instead. The advantage is evident. Large queries can be contained in variables and used repeatedly. A very brief example is showed in listing 5.7, where we contain a query's arguments to preserve the appearance of the code.

**Listing 4.15** Prolog's tracing mechanism

```
1  arc(a, b).
2  arc(b, c).
3
4  ? - trace.
5
6  [trace] ?- arc(X, Y).
7    Call: (7) arc(_G1569, _G1570) ? creep
8    Exit: (7) arc(a, b) ? creep
9  X = a,
10 Y = b;
11   Redo: (7) arc(_G1569, _G1570) ? creep
12   Exit: (7) arc(b, c) ? creep
13 X = b,
14 Y = c.
```

## 4.9. Missing features

It is worth noting that some of the constraints introduced by design decisions limit some core Prolog functionality. For example, in Prolog, variables can be used in the body of a predicate for assignments as seen in 3.1. NoLog, on the other hand, does not support this behaviour yet, as our variables are only identified by strings and while most of the constructs are in place, the syntax does not allow for it yet. This means that unfortunately, arithmetic operations are not supported.

Furthermore, Prolog developers are used to being able to perform traces of their queries for debugging purposes. While NoLog offers no explicit tracing mechanism, the idea has been considered and marked as a possible enhancement. The recently added Knowledge Engine should prove extremely useful in offering a usable trace, given its ability to delay predicate evaluation. An example of Prolog's tracing mechanism is visible in listing 4.15. Consider we only have knowledge of two facts, `arc(a, b)` and `arc(b, c)` and that we want to query `arc` while tracing the execution. For debugging purposes, Prolog allows us to peek at the execution of each query, aiding developers in identifying bugs.

Less important than the two core Prolog features outline earlier is the way of execution. Prolog developers tend to define a program in a `program.pl` file which will the be loaded from the Prolog prompt. Queries can then be performed inside the prompt regarding facts and predicates defined in the `program.pl` file. Python is rarely used from the prompt, as most developers prefer the power of a text editor and the terminal. However, I believe this behaviour should still be preserved, as it provides a familiar ground to Prolog developers.

## 4.10. General overview

Throughout this chapter we have looked at various features and requirements that NoLog must support and showcased their evolution. However, it was mostly syntax related, but having done so, we can now move onto some of their technical aspects.

# 5. Implementation

## 5.1. Implementation details

The table in 5.1 showcases the overall progress of NoLog. It describes the semantic versioning [9] used to keep track of the evolution of the library as new features have been added, bugs fixed and overall improved design decisions have been taken. Semantic versioning provides a way of quantifying the evolution of a project. A version number is of the form **Major**.**Minor**.**Patch**, where each field gets increments in certain cases. **Major** values get incremented when making incompatible API changes, **Minor** when adding backwards-compatible functionality and **Patch** when fixing bugs. For example, Predicates and Unification were the main features that development began with. Queries, on the other hand, have been implemented after unification and their syntax updated until the latest version. Then came negation and extension of pattern matching, finally followed by general rules. A complete change log is offered in table D.1. With that knowledge in mind, we can now proceed to analyse the inner workings of NoLog.

Table 5.1.: Semantic versioning table

| **Feature** | **Semantic version** |
|---|---|
| Predicates | 0.0.1-0.0.3 |
| Unification | 0.0.3-0.0.8 |
| Queries | 0.1.0-0.5.8 |
| Negation | 0.3.0-0.3.5 |
| List pattern matching | 0.4.0-0.4.5 |
| Rules | 0.5.0-0.5.8 |

NoLog draws inspiration from the Knowledge Representation [27], Introduction to Artificial Intelligence [31] and Logic-based learning [23] courses. Initial development followed an evolutionary prototyping, where the aim

**Listing 5.1** Knowledge base

```
1 { 'parent': [('patrick', 'thomas')],
2   'male'  : ['bruce', 'thomas'] }
```

was to build a robust system and as time goes on, add new features and improvements. However, towards the end it became a mixture of evolutionary prototyping and rapid or throwaway prototyping. One of the implementation aims of NoLog was to avoid reinventing the wheel whenever possible. The approach taken demonstrates that in the attempts to leverage existing Python features and using them outside their normal scope. As such, the decision to represent known facts, i.e. clauses with no constraints and no body, into a dictionary structure, seemed natural to me. Furthermore, the dictionary provides efficient lookup time and makes unification easier. The keys represent the name of the rule, and the values are possible values structured in a list. For the code in our family example 2.1, the knowledge base would look something like listing 5.1.

The overall architecture of NoLog is reasonably simple. The codebase was split into multiple modules, with the main interaction point being the Knowledge Engine. The engine offers access to predicates, queries and evaluation, which, in turn, use various other private methods to perform their tasks. The unification module holds the unification algorithm and can be used independently of the engine, provided the same syntax is used when calling it.

## 5.2. Predicates

A simplified NoLog representation of a predicate adheres to the pattern in 4.3, where `p_body` is optional, due to some predicates representing general rules or facts. The NoLog code in 4.3 is already less verbose and easier to read than its equivalent in 4.1 and a more similar syntax to Prolog is offered in 4.4. We will traverse the guided execution of this example first to see what has been accomplished and what edge cases were not dealt with appropriately.

Remember the latest predicate definition. It takes up to five input arguments: the predicate name `p_name`, the arguments `p_args`, the definition of the body `p_body` and two enabling points for various behaviours used by negation as failure and general rules. We will begin by ignoring the two enabling points and focus on the basic behaviour.

A predicate can be executed without any arguments or body definitions, but the only computation that is performed is the addition of the predicate name as a key into the knowledge base and rule base, unless already present. Once the arguments have been passed in, if there is no body definition, we have to check whether we may be dealing with a fact. A fact is a predicate without a body which has no variables as arguments. Therefore, we need to perform a check to assess whether there are any variables in the arguments. There are two executions paths possible at this point. The first one deals with situations where all the parameters are constants. In such cases, the fact becomes stored in the knowledge base and the predicate returns **true**. The second one handles predicates with at least some variables among their parameters. If that is the case, then the predicate is considered a rule and added into the rule base. Prior to the implementation of the knowledge engine, the next step was the execution of the code to compute all admissible facts and store them in the knowledge base.

That explanation of how and when facts become part of the knowledge base flags up an edge case. Prolog supports both predicate and propositional logic, and NoLog seems to offer no support for the latter. To avoid having to rewrite the code entirely, a workaround was offered. Defining propositional atoms is done by using the actual atoms as parameters to the 'atoms' predicate, preserving the overall behaviour of the system.

To illustrate another edge case, let us assume that we do not know about the engine for the time being and that the code is still executed. Our aim is to find all the possible paths in the arc/path system 4.4.

### 5.2.1. Direct evaluation

Take line 4 of listing 4.4. The predicate name 'arc' is not in knowledge base as it is the first time NoLog encounters it, so an entry is made and the value

**Listing 5.2** Arc knowledge base

```
1 {'arc': [('a', 'b'), ('b', 'c')]}
```

**Listing 5.3** Incomplete Arc-path knowledge base

```
1 {
2   'arc': [('a', 'b'), ('b', 'c')],
3   'path': [('a', 'b'), ('b', 'c')]
4 }
```

initialized to the empty list. No body definition exists so NoLog go through the checks to asses whether it is dealing with a fact. Having decided that it is, the entry `('a', 'b')` is added to the list corresponding of known arcs. Similar computation is performed for line 5, adding `('b', 'c')` as an entry to the 'arc' facts. Our knowledge base looks like listing 5.2 and now that facts are working we can perform a query for arcs, the result of which would contain both possible answers.

That was pretty straightforward. Line 6 becomes more interesting, since it represents a rule, and a body is provided. The current information from the predicate is passed into a backwards chaining function. Backwards chaining is a method that finds a solution starting from the goal. Our current goal is the head of the predicate, `path('X', 'Y')`. Since the body represents constraints that have to be met in order for the goal to be satisfied, we begin by looking at the body. We take the first entry in the body, the predicate `arc('X', 'Y')`. The actual representation is different, but for brevity we will abstract the tuple-style declaration. We now can state that if `arc('X', 'Y')` is satisfied and if the rest of the body is also satisfied, our goal is satisfied. We begin by unifying `arc('X', 'Y')` with current known arc facts. Possible solutions will be represented by X: [a, b] and Y: [b, c]. There are no other entries in the body, meaning that for the two results, the head is satisfied and both entries are added into the knowledge base 5.3.

The last line of the program is even more interesting since not only does it define a rule, it defines a recursive rule where the path head is dependent

**Listing 5.4** Arc-path knowledge base

```
1 {
2   'arc': [('a', 'b'), ('b', 'c')],
3   'path': [('a', 'b'), ('b', 'c'), ('a', 'c')]
4 }
```

on the path in the body. The same evaluation as with the previous case is performed, `arc('X', 'Z')` giving us the two entries `'X': ['a', 'b']` and `'Y': ['b', 'c']`. Moving towards the right in the body, we encounter `path('Z', 'Y')`. Since one of the arguments, Z, has two possible instantiations, 'b' and 'c', we use those two values to compute the following queries for path: `('b', 'Y')` and `('c', 'Y')`. Our next task is then to see which query satisfies the `path` predicate. Searching the knowledge base, we have a possible solution for the former by instantiating `'Y'` to 'c', while the latter fails finitely. Therefore, the second possible answer is thrown away as there is no instantiation for `'Y'` which would satisfy `path('Z', 'Y')`. We are now left with the following instantiation: `'X': ['a']`, `'Z': ['b']` and `'Y': ['c']`. Those values are then used to construct a solution for `('X', 'Y')`, namely `('a', 'c')`, which gets added to the list of possible answers. In the end, the backward chaining method returns the found solutions that have not been previously discovered, and they are being stored in the knowledge base. Those are the fundamentals of evaluating predicates and storing the solutions into the knowledge base.

At this point, there is one issue worth mentioning. The overall program would fail to find all solutions if they were based on generated facts other than the original ones. Let us assume we have knowledge of `arc('c', 'd')` on top of the two already defined arcs. The backwards chaining method used to compute the solutions would require two executions before all paths would correctly be discovered. The reason behind this lies within the overall design. The first execution finds all answers one step away from the original program definition. However, the facts generated then do not get used to compute the rest of the answers. Since we do not know how many executions it takes to find all the answers, we choose to continue evaluating as long as new information is discovered. If something has already been found, it does

not count as new information, and when no new information is returned, the execution terminates.

## 5.3. Unification implementation

Unification is more complex than the predicate evaluation, but better defined as a problem, as there is a limited number of cases that must be supported. Development was performed incrementally, the different scenarios being tackled as NoLog evolved. The initial goal was to be able to unify two variables, while the next step aimed to make unification scalable for dynamic numbers of variables. However, predicates often receive more than just variables, atoms or numbers. They may receive a list or a tuple and the different cases required different approaches and conditions.

The most recent version of unification supports all of that. Since in Prolog, the name term refers to anything that can be unified, I realised Python's dynamic type system will be a liability in the implementation of unification. Polymorphic function definitions would have been the most elegant solution for it. However, Python offers no support for polymorphism, despite recent attempts at implementing it. The only solution was to perform the type checking manually and define helper methods to deal with each and every case. The execution trace is showcased in a simplified manner in listing 5.5.

Type checking was required as a consequence of my decision to not implement wrappers around the data types that Prolog works with. It has been stated that PyDataLog [7] implements its own data types that extend PrologObjects. The object-oriented approach simply adds wrappers around various types that Python already supports. While it allowed the developer of PyDataLog to not worry about any type checking, its results were far from ideal. Users now had to actively define objects in their predicates: Atom('a'), Number(5), Variable('X'). Our initial approach was taking a similar stance with regards to variables and we saw first-hand the issues it raised and the amount of noise it introduced into the syntax. Therefore, in an attempt to improve the overall user experience, I decided to perform type checking in the backend, rather than shifting the responsibility to the user.

**Listing 5.5** Unification trace

```
unify(term_1, term_2)
  if term_2 is None:
    unify_single_term(term_1)
      compute list of possible answers from the
    knowledge base
      attempt unification of term_1 with every
    possible answer
      store all possible solutions and return them
  otherwise:
    unify_multiple_terms(term_1, term_2)
    if both terms are either strings or integers:
      if both atoms or numbers:
        return result of equality check
      if one term is a variable:
        instantiate it to the other term
      if both are variables and
          no clashes exist
            with previous instantiations:
        instantiate each to the other
    if one term is a list wrapper:
      unpack the other term
      return the possible instantiations
    otherwise:
        for each t1, t2 in zip(term_1, term_2):
          unify(t1, t2)
```

**Listing 5.6** Complex queries and unification in Prolog

```
1  father(john, [tom, david, samantha]).
2  father(richard, []).
3  father(mark, [rodney, jenny]).
4  father(tony, [michelle, olivia]).
5  father(matt, [primus, secundus, tertius, quartus,
       quintus, septimus]).
6
7  father(X, [Y, Z]).
8  father(X, [Y|Z]).
9  father(X, [Y, T|Z]).
10 father(X, []).
11 father(X, [primus|T]).
```

Unification for simple cases was showcased in 2.2 and during the walkthrough of our arc-path example in the previous section. However, Prolog is capable of performing more complex unifications. Take for example the code in 5.6.

We can perform various queries to suit our needs. For example, line 7 asks for all fathers X who have exactly two children, Y and Z. The result is $\theta$ = `{X: [mark, tony], Y: [rodney, michelle], Z: [jenny, olivia]}`. Line 8 queries all fathers with at least one child and similarly, line 9 asks for all fathers with at least two children. Line 10 wants to know if there are any parents that do not have any children. An argument could be made that you cannot be a parent without having children, so let us assume that the predicate `father` only models the biological relationship. The last query asks for all parents who have named their first child Primus. The only answer is $\theta$ = `{X: [matt], T: [secundus, tertius, quartus, quintus, septimus]}`. As can be seen, only Matt seems to have been hung up on his Latin lessons.

The equivalent NoLog computation in 5.7 leverages both Python lists and ListWraps, and the unification is performed term by term. The first five queries are the equivalent of the queries defined above. The last one is a result of the side-effects of the unpacking operator in Python, and aims to add expressivity beyond what Prolog is capable of.

**Listing 5.7** NoLog queries containing iterables

```
1  father = partial(predicate, 'father')
2
3  father(('john', ['tom', 'david', 'samantha']))
4  father(('richard', []))
5  father(('mark', ['rodney', 'jenny']))
6  father(('tony', ['michelle', 'olivia']))
7  father(('matt', ['primus', 'secundus', 'tertius', '
       quartus', 'quintus', 'septimus']))
8
9  query('father', ('X', ['Y', 'Z']))
10 query('father', ('X', ListWrap(['Y', '*Z'])))
11 query('father', ('X', ListWrap(['Y', 'T', '*Z'])))
12 query('father', ('X', []))
13 query('father', ('X', ListWrap(['primus', '*T'])))
14 q_unify = ('X', ListWrap(['*H', 'quintus', 'T']))
15 query('father', ('X', q_unify))
```

## 5.4. Negation by failure

We have described the manner in which negation as failure has been implemented previously in 4.8. The predicate function now takes an optional parameter `p_naf`. It plays the role of an enabling point for a seam, meaning it alters the evaluation in certain places. A constraint of Prolog states that the head of any predicate must be a positive term, meaning negation cannot be used there. That only leaves the terms in the body as a possible place for negation. While it is originally passed into the predicate method, it is only used in the helper method performing backward chaining.

If you remember, for the example in listing 4.4, no element of the body was defined using negation as failure. The execution first computes all possible answers of the first element of the body. Then, for the remainder elements, it attempts to create queries by using either instantiated arguments or the arguments themselves if no variables had been instantiated. The execution terminates with all the queries that satisfy each predicate. If a predicate is not satisfied, the execution fails finitely.

Let us now look towards an example where negation is used and compare the execution paths. Take the code defined in 5.8. The `sad` predicate states that X is sad if X is a `student` and `X` does not have a grant. The example is so simple we can immediately see the answer. Mary is the only student that we know of who does not have a grant, meaning Mary is sad. While the human brain has the capability to infer the result almost immediately, NoLog needs to perform a few checks. We will skip the facts as they introduce nothing new, and instead, focus on the predicate sad. We know X is sad if X is a student and X does not have a grant. Since it is dealing with a rule, NoLog will try to satisfy the body in order to prove the head. The first step is the unification of `student(X)`, resulting the the unifier $\theta$ = {X: ['john', 'mary']}. The next step is now computing the queries for `not grant(X)`. Since we have all the possible values of X already, we can simply check them against `not grant(X)`. Remember, NoLog does not view the predicate as a negation, the only difference is that the negation flag will be set to true. For the first instantiation of X, we want to check whether `grant('john')` is true, and we can see that it is. However, due to the negation flag being set to true, the result is false, so 'john' is not sad. Given the second possible value of X, we want to check whether `grant('mary')` is true. This time, the computation will fail and `grant('mary')` will return false. Once again, however, the negation flag is set and the result becomes true. In this situation, both entries of the body are satisfied, meaning 'mary' is a sad student. The current example was a simple one, since in both cases the results were boolean and in order to get the actual value, all we did was perform an exclusive-or between the result and the value of the negation flag.

That was a simple example, however. Looking at something more complex, we can check whether an individual likes a certain burger. Let us assume we live in a small, sad, world in which only three types of burgers exist: a taxi driver, a big kahuna and a whooper, as in listings 5.9 and 5.10. Besides the facts, Prolog would tell us that `vincent` dislikes `big kahuna` burgers and `whoopers`. Furthermore, it would also tell us that Vincent likes the Taxi Driver. The tricky part comes when we define the `not_like` predicate. You may imagine we should be able to obtain the same information we have for dislike, since that is how the human brain is wired to interpret it. However,

**Listing 5.8** Negation as failure NoLog example

```
1 student = partial(predicate, 'student')
2 sad = partial(predicate, 'sad')
3 grant = partial(predicate, 'grant')
4 grant('john')
5 student('john')
6 student('mary')
7 sad('X', [('student', 'X'), ('grant', 'X', True)])
```

Prolog works differently. If we were to perform a query `not_like(X, Y)`, we would only be able to say that X does not like `big kahuna` burgers and X does not like `whoopers` for any X. That is because Prolog queries are using unification and we have mentioned previously that unification returns the most general unifier [30].

While implementing negation as failure, that was the same output NoLog would have returned. However, that seemed, if not wrong, at least incomplete. Yes, it would return a less general rule than the original predicate, but in my opinion, it added no real value to the program. That reasoning led me to try a different approach. During the evaluation of `not_like`, we implicitly have to evaluate the body, meaning that for the positive representation of each body predicate we have the following unifiers. $\theta$ = {Y: [taxi_driver, big_kahuna, whooper]}, and $\theta$ = {X: [vincent], Y: [taxi_driver]}. We can also clearly observe that the likes predicate is in its negated form. While, the likes predicate unifies for a single instantiation of Y, namely `taxi_driver`, in that single case we get a possible value for X. And because there are no other instantiations of X for any other values of Y, we choose to use that one instantiation to be able to essentially generate the same facts as `dislikes`. While it is not standard Prolog behaviour, I believe it might improve the overall expressivity. In the event that it does not, the block of code responsible for this behaviour consists of only four lines that can be discarded.

It is worth mentioning that while Prolog would not compute these answers directly, there is a way of reaching the same facts by instantiating X individually in the body definition of `not_likes`, by declaring vincent as a person

**Listing 5.9** Burger world facts

```
burger = partial(predicate, 'burger')
taxi_driver = partial(predicate, 'taxi_driver')
big_kahuna = partial(predicate, 'big_kahuna')
whooper = partial(predicate, 'whooper')
likes = partial(predicate, 'likes')
dislikes = partial(predicate, 'dislikes')
not_like = partial(predicate, 'not_like')

taxi_driver('taxi_driver')
big_kahuna('big_kahuna')
whooper('whooper')

burger('X', [('taxi_driver', 'X')])
burger('X', [('big_kahuna', 'X')])
burger('X', [('whooper', 'X')])
```

and querying `person(X)`.

**Listing 5.10** Burger world rules

```
1  dislikes(('vincent', 'X'), [('burger', 'X'),
2                              ('big_kahuna', 'X')])
3  dislikes(('vincent', 'X'), [('burger', 'X'),
4                              ('whooper', 'X')])
5
6  likes(('vincent', 'X'), [('burger', 'X'),
7                           ('big_kahuna', 'X', True),
8               ('dislikes', ('vincent', 'X'), True)])
9
10 not_like(('X', 'Y'), [('burger', 'Y'),
11                       ('likes', ('X', 'Y'), True)])
```

## 5.5. Knowledge Engine and Rules

While predicates are rules, they did not initially highlight some prevalent issues. NoLog needed to be able to support body-less, general rules, and the overall initial design highlighted the need for deferred execution. Without changing the Python interpreter, the only alternative was storing rules until they are required. This allows for querying of general rules as well, as no facts need to be added for a certain query to be valid. Recall the definition of member from 4.13. No facts can be computed from it, but for the right instantiation of variables, we may still get a valid query. On the other hand, recall the path example from 4.4. In that case, we require the `arc` facts to be able to obtain relevant information.

Furthermore, delaying evaluation solves one of the constraints that the initial design of NoLog introduced. A base case for a predicate had to be defined prior to the recursive cases. Predicates from the body of a clause had to be defined before their use. Not respecting those conditions will cause NoLog to either return an incorrect result or complain about not knowing anything about a certain predicate. All of these constraints are a direct consequence of Python's sequential evaluation, much like the way you would not use a variable prior to declaring it. Delaying the evaluation partially solves these restrictions. The only remaining constraint is the one stating that base cases have to be defined prior to recursive ones. An attempt could be made to

**Listing 5.11** Rule storage

```
{
path: [
        [('X', 'Y'), [('arc', ('X', 'Y'))]],
        [('X', 'Y'), [('arc', ('X', 'Z')),
                      ('path', ('Z', 'Y'))]]
      ],
member: [
          [('X', ListWrap(['X', '*T']))],
          [('X', ListWrap(['H', '*T'])),
                  [('X', ListWrap(['*T']))]]
        ]
}
```

remove it as well, but I feel that preserving a level of required organisation will result in cleaner code.

Having explained the why, now comes the how of the Knowledge Engine. The overall computation was already in place, inside the predicate method. The Knowledge Engine came as a simple wrapper around the predicate method. The evaluation section was removed from the predicate method and placed inside a standalone method. The block of code was initially designed to only handle a single predicate, so we had to iterate through all the rules and evaluate each one.

In turn, that requires all rules to be stored in a data structure so they may be accessed repeatedly and individually. Initial representation drew inspiration from the design of our knowledge base. Each predicate would be a key in the rule base and mapped to a list of possible rules. The consensus was that each rule would follow the representation of predicates. Each rule would be a tuple where the first element represents the arguments of the head, while the body is represented inside a list, as if it were passed into the predicate function.

Notice how we have not mentioned facts at all. Nothing has changed when dealing with facts. However, the overall execution now offers a hybrid approach between the knowledge base model and general rules. But dealing

**Listing 5.12** Pseudocode for checking rule

```
1  valid_entry = False
2  for each rule:
3    unify head of rule
4    rule_SAT = True
5    for each possible body definition:
6      construct query
7      if not predicate(body, query)
8        rule_SAT = False
9        break
10     if rule_SAT:
11       valid_entry = True
```

with rules changed the way the predicate function worked. While initially it was a **void type** function, the implementation was changed to return more information. I have, mistakenly, attempted to solve this problem by writing more code to deal with the definition of rules. However, the solution proved to be flawed. It did not scale well and any attempt at changing that resulted in increased cyclomatic complexity of the code. It then dawned on me that I was storing rules using the already existing representation of predicates. If so, I could simply pass them up as arguments to the predicate function, or query them. Of course, that changed nothing as they would still use the variables with which they were defined. However, a solution was already in place for dealing with such a scenario. In the backwards chaining method we used to compute new queries for body elements based on possible instantiations of variables in the scope of that predicate. Querying the knowledge base would use any defined rules for predicates, if they are there. If no rules are defined, it performs the query using previously known facts.

The pseudocode in listing 5.12 shows the execution trace of checking a query against a rule. It aims to prove the goal by proving at least one of the bodies. Essentially, if there is a rule that makes the head goal true, the execution terminates. Otherwise, it keeps trying as long as there are different possible definitions. If nothing succeeds, it terminates and returns False.

# 6. Evaluation and testing

This chapter focuses on the evaluation of NoLog, considering a number of different design aspects as well as the overall code quality. We will discuss qualitative and quantitative evaluation, as well as testing. Ever since I have begun developing NoLog, a personal aim has been to be able to release it as an Open-source toolkit, as one of Python's advantages is the community and the open-source approach to new developments. Therefore, evaluation will take into account both future developers of the framework as well as future users. Firstly, the future users will expect to be able to easily write code if they are familiar with Prolog, and if not, to be provided with examples and good documentation, as well as a short guide on how to get started, for the more practically inclined / practical learners. Secondly, future developers should be able to obtain an overview of the toolkit as well as what drove different design decisions. Steve McConnell once said that "Good code is its own best documentation". However, I believe that while good code is indeed readable, it is also well-documented for future developers.

## 6.1. Qualitative evaluation

### 6.1.1. User focused evaluation

In the beginning, NoLog was just two functions dealing with predicates and unification. There was very little evaluation that could have been done. The main requirement at that time was to be able to write code that looked similar to Prolog while integrating seamlessly in Python. As more features were added and the overall code complexity increased, the syntax remained an important focal point.

Consider one of our earlier examples from listing 1.1. For better showcasing the aesthetic evolution of the syntax, a few examples can be seen in listing 6.1. The syntax is clear and concise, and anyone can grasp the meaning. On the other hand, take the original code from listing 4.1. The NoLog syntax introduced a few elements such as clause to describe a predicate or Variable to actively declare a variable. Furthermore, the body of a predicate had to be explicitly defined, increasing the amount of noise in the code.

While the extra constructs do not alter the code beyond recognition, the NoLog implementation can get rather messy when larger clauses would have to be defined. For example, imagine having to define something like the Prolog code in 6.4. The snippet below tests a strategy against an opponent N times, keeping track of how many wins each side accumulates.

The base case, when there are no more games to be played displays the results, while the second definition makes sure that all N games are played and records the results. Recursively, the two cases can be handled easily, but the solution would look rather messy due to the definition of each clause, arguments and body constraints. However, the translation to NoLog would benefit from recursive features of Python as well as a more common if-else syntax and easier variable assignment, leading to a better combined code from an aesthetic point of view. And with more research, we can hope to reach an even more elegant solution, one that would better mimic the cleanliness offered by Prolog and bridge the gap between the two code styles even further.

Fast forward to the latest version, listings 4.3 and 4.4 showcase the current syntax available. The quantity of boilerplate code has been reduced, resembling Prolog even more, while at the same time preserving a pythonic appearance.

Another relevant example showcasing a difference between Prolog and NoLog can be observed in listing 6.2. Prolog's syntax is less verbose as no wrapper class needs to be instantiated. However, any developer should be capable of simply abstracting away the `ListWrap` call as they would a print statement. No one would get hung up on the print, everyone would simply parse the printed message, as that was the useful information. Since Python does not allow users to define their own operators, the only way a new syntactic

**Listing 6.1** Comparison between syntax

```
1  # Prolog syntax
2  path(X, Y) :- arc(X, Z), path(Z, Y).
3
4  # Original NoLog syntax
5  clause('path', [Var('X'), Var('Y')],
6          body=[clause('arc', [Var('X'), Var('Z')],
7                clause('path', [Var('Z'), Var('Y')])
8                ])
9
10 # Possible declarations of path in current NoLog
11 predicate('path', ('X', 'Y'), [('arc', ('X', 'Z')),
12                                 ('path', ('Z', 'Y'))
13                                 ])
14
15 path = partial(predicate, 'path')
16 path(('X', 'Y'), [('arc', ('X', 'Z')),
17                   ('path', ('Z', 'Y'))
18                   ])
```

construct to unpack a list would be to alter the interpreter. For maintenance and to preserve NoLog's compatibility to future versions of Python, we will refrain from doing so. Under such circumstances, the list wrapper is an adequate if not elegant solution.

As Prolog is a complex language, some of the capabilities have not been currently implemented. Such an example was offered in listing 3.1. Another example can also be found in listing 6.4 code. Notice how line 2 instantiates `Draws` to be `N - WinsRed - WinsBlue`. Design decisions have constrained variables to be simple upper case strings and not standalone objects, meaning that we cannot perform instantiations like that in the body of a clause. A solution will be found to address this issue, while in the meantime, NoLog currently supports predicate definitions, delayed execution courtesy of the Knowledge Engine implementation as well as unification and querying capabilities. The use of the knowledge base still allows for hybrid computation of answers based on the actual content. A possible example of what the syntax might look like in the future can be observed in listing 6.5. The timing is

**Listing 6.2** Pattern matching on lists

```
1  # Prolog recursive case of member
2  member(X, [H|T]):- member(X, T).
3
4  # NoLog recursive case of member
5  member = partial(predicate, 'member')
6  member(('X', ListWrap(['H', '*T']),
7         [('member', ('X', ListWrap(['*T'])))]))
```

**Listing 6.3** Granularity example

```
1  # assume arc is defined, now let us store some
       facts first
2  arc(('a', 'b'))
3  arc(('b', 'c'))
4  arc(('c', 'b'))
5
6  answers = query('arc', ('X', 'b'))
7  print(answers)
```

not performed within the body as we can perform it outside both queries or within the smaller constraints, like `play`. Furthermore, the increased granularity can be observed easily, Python code can simply be executed within the body definition and the answers used to keep computing the right solutions.

Moving away from syntax for a moment, since NoLog offers essentially a different Python execution using the Python interpreter, the fine granularity of the project should be mentioned. Users will not be required to write their program all at once. Incremental updates are more than common as there is no shifting between the two languages. An example can be observed in listing 6.3.

**Listing 6.4** War of Life Prolog code snippet

```
1  test_strategy(N, 0, StrategyP1, StrategyP2,
       WinsBlue, WinsRed, Longest, Shortest, MovesAcc,
       TimeAcc) :-
2    Draws is N - WinsBlue - WinsRed,
3    write('Number of draws: '), write(Draws), nl,
4    write('Number of blue wins: '), write(WinsBlue),
       nl,
5    write('Number of red wins: '), write(WinsRed),
       nl,
6    write('Longest game: '), write(Longest), nl,
7    write('Shortest game: '), write(Shortest), nl,
8    AvgMoves is (MovesAcc / N),
9    write('Average game length: '), write(AvgMoves),
       nl,
10   AvgTime is (TimeAcc / (N*1000)),
11   write('Average game time: '), write(AvgTime), nl.
12
13 test_strategy(N, GamesLeft, StrategyP1,
       StrategyP2, WinsBlue, WinsRed, Longest,
       Shortest, MovesAcc, TimeAcc) :-
14   statistics(walltime, _),
15   play(quiet, StrategyP1, StrategyP2, NumMoves,
       WinningPlayer),
16   statistics(walltime, [_ | Time]),
17   NewTimeAcc is (TimeAcc + Time),
18   NewGamesLeft is (GamesLeft - 1),
19   NewMovesAcc is (MovesAcc + NumMoves),
20
21   (WinningPlayer == 'b' -> NewWinsBlue is
       (WinsBlue + 1), NewWinsRed is WinsRed;
22    NewWinsBlue = WinsBlue, (WinningPlayer == 'r'
        -> NewWinsRed is (WinsRed + 1) ;
23    NewWinsRed = WinsRed)),
24
25   (NumMoves < Shortest -> NewShortest = NumMoves ;
26    NewShortest = Shortest),
27   (NumMoves > Longest -> NewLongest = NumMoves ;
28    NewLongest = Longest),
```

**Listing 6.5** War of Life NoLog code snippet

```
test_strategy = partial(predicate, 'test_strategy')

test_strategy(('N', 0, 'StrategyP1', 'StrategyP2',
        'WinsBlue', 'WinsRed', 'Longest',
        'Shortest', 'MovesAcc'),
        [ 'Draws' = 'N' - 'WinsBlue' - 'WinsRed',
          print('No of draws: ' + 'Draws'),
          print('No of blue wins: ' + 'WinsBlue'),
          print('No of red wins: ' + 'WinsRed'),
          print('Longest game: ' + 'Longest'),
          print('Shortest game: ' + 'Shortest'),
          'AvgMoves' = 'MovesAcc' / 'N',
          print('Avg game length: ' + 'AvgMoves'),
        ])

test_strategy(('N', 'GamesLeft', 'StrategyP1',
        'StrategyP2', 'WinsBlue', 'WinsRed',
        'Longest', 'Shortest', 'MovesAcc'),
        [('play', ('quiet', 'StrategyP1',
                   'StrategyP2', 'NumMoves',
                   'WinningPlayer')),
          'NewGamesLeft' = 'GamesLeft' - 1
          'NewMovesAcc' = 'MovesAcc' + 'NumMoves',
          if 'WinningPlayer' == 'b':
            'NewWinsBlue' += 1
          if 'WinningPlayer' == 'r':
            'NewWinsRed' += 1
          if 'NewMoves' < 'Shortest':
            'NewShortest' = 'NewMoves'
          if 'NewMoves' > 'Longest':
            'NewLongest' = 'NewMoves'
        ])
```

### 6.1.2. Developer focused evaluation

One of the aims of NoLog has always been open-sourcing in an attempt to fill a void in Python's capabilities. One of Python's greatest advantages over other languages has always been the excellent open-source community surrounding it, offering a wealth of documentation, as well as numerous tools. Furthermore, there is always someone willing to contribute and help move things forward. It is, after all, what makes the open-source movement such a success.

To be able to be a part of that, you have to begin by assuming people will be willing to contribute. All open-source projects have begun by attracting some users who had need for them. As time went by, some of those users became the developers in an attempt to expand the original capabilities of the project. However, in order to get people excited about contributing, they need to be able to understand how the project works, as well as what design decisions have been made and the reasons behind them. As developers, we can appreciate elegant code and we look down on big balls of mud, deeming them difficult to maintain, test or expand safely. Thus, we understand the need for documentation and elegance in design.

While it began in a chaotic way, there has always been a focus in maintaining a high-standard for the code. Linters have been used to ensure compliance with Python's official style guide [32]. Documentation strings as showcased in listing C.2 have been added for most major methods, encompassing the functionality provided by helper methods, as suggested by Python's DocString guidelines [16]. To preserve the overall flow of the code, in-line comments have been added sparsely and only where design decisions were disputed or raised some concerns, as a warning to future developers C.1.

Rapid prototyping sometimes results in large, complex and inelegant code, and can potentially lead to serious issues down the line. The technical debt introduced by our approach resulted in the use of multiple online code quality tools, which have been integrated into the version-control platform, to target different issues. For example, CodeClimate [3] focused mostly on cyclomatic complexity, a measure describing the number of individual execution paths in a block of code [19]. Refactoring had to be performed in order to split

code into smaller methods, making the overall design more manageable, as larger blocks of code tend to be error-prone and harder to follow. Codacy [2] and CodeFactor [4] have been used to analyse the overall codebase for duplication, linting errors or error-prone code. All the information provided by these tools aided the development process by providing insight into all of these aspects and ensuring good coding standards have been met. As a result, all the technical debt has been completely paid off and duplication removed altogether.

## 6.2. Quantitative evaluation

One of NoLog's requirements has been to resemble Prolog as much as possible, therefore, analysing the verbosity of Prolog and NoLog yields our measure for quantitative evaluation. NoLog's syntax is only slightly more verbose than Prolog's, both offering similar lines of code for representing the same predicates. Take our example in listing 6.1. Disregarding the partial definition as it is not required, we can have a `path` rule definition in one line. Looking at something more complex in listings 6.4 and 6.5, the main difference again comes from the definition of the partial predicate. The Prolog code takes 28 lines of code and has not been formatted in any way, whereas, with formatting and including the partial definition, the NoLog equivalent code would require 32 lines, a negligible difference.

## 6.3. Testing

The development process worked in a test driven manner. We were focusing on a single example at a time, while ensuring no previous functionality had been broken. Testing was introduced in version 0.0.3, while implementing the basic behaviour for predicates and unification. Due to the small size of the implementation, at that moment in time,testing was performed manually and only included several methods that checked the overall behaviour of minuscule examples to ensure backwards compatibility. However, once I have turned my attention towards negation, it became apparent that larger examples had to be used, and relying on manual testing was no longer viable,

marking the introduction of unit testing in version 0.2.5.

Since then, unit testing has been implemented for each use case that we have covered. Testing does not directly cover private module methods aimed to support the main functionality of individual modules. Instead, it focuses on the higher-level execution, which cascades through the helper functions. Should that fail during development, we would be able to point the fault to one of the helper methods using the error trace.

# 7. Conclusions and further work

This chapter covers some of the insights that I have gained into the computational models of Prolog and Python and how to better bridge the gap between the two languages. Furthermore, it offers some insight into my own self evaluation, as well as future improvements to be performed.

Having looked at current solutions, we were able to identify the benefits and disadvantages of various approaches and take the optimal development route for NoLog. In spite of all the differences between Prolog and Python, we have managed to add logic programming functionality into Python in a natural way while still preserving a degree of similarity to Prolog.

I continue to believe that the use of backtracking in Prolog is less than efficient, and that, computationally, Prolog might benefit from at least a hybrid approach between backtracking and fact storage, balancing the space-time trade-off. NoLog's original goal was to perform all computation iteratively and in a single pass, thus leveraging the knowledge base model and avoiding backtracking altogether. The idea was to use filtering on instantiation lists to compute all possible facts. However, the algorithm was difficult to scale for more complex predicates and, due to time constraints, postponed. Instead, we chose to implement a hybrid approach, where we still use the model approach to store facts. Furthermore, it has been extended to support general rules for querying, and in those instances, facts are not being currently stored.

NoLog's simple design allowed for a logical structure of its architecture. The Knowledge Engine is the entry point for every computation, be it a query or a predicate definition. The ListWrap class simply extends Python native behaviour to support Prolog's list pattern matching and the Unification module handles all unification cases internally. With all of the components clearly delimited in these 3 modules, debugging was swift and efficient.

## 7.1. Self evaluation

This section aims to cover some of the mistakes made during the development of NoLog and the different approaches that I would take should I start again.

While the overall architecture functions as expected, design decisions taken throughout the development process have imposed several constraints and highlighted some suboptimal development phases of the project. The original system was approached from a predicate logic perspective, with no thought being offered to propositional clauses, leading to issues down the line. Despite the fact that a workaround was found, development should have begun with the most basic case, rather than skipping it for the most common use case. To that end, before implementing any code, a comprehensive list of use cases covering all necessary features should have been defined, rather than focusing on a single feature and use case at a time.

Furthermore, while I still favour the model approach over the backtracking implementation of Prolog, I have naively not realised the complexity of it, spending too much time on it and essentially taking away from other features. It would have been a far better approach to implement Prolog's unification and resolution before working on a different idea. The complexity of the challenge was masked by my internal representation of predicates, which altered the way I was tackling unification. Having fixed that, unification was functioning as expected and the resolution was much easier to implement using a hybrid approach.

Focusing on the hybrid approach and the poorly defined list of use cases had another, more serious, consequence. NoLog was not able to deal with general predicates and queries. A general predicate could potentially define an infinite number of facts, and the knowledge base was not built to deal with such a case. A solution was found that combines the two types of predicates and the different approaches. However, instead of having two different approaches, a unified solution should have covered both cases from the beginning.

Ultimately, the technical debt has been paid off and the implemented features behave as expected. However, there is still work to be done and potential improvements which we will discuss in the next chapter.

## 7.2. Further work

This section discusses some of the features of NoLog which would benefit from further development or design improvements.

NoLog's variables are defined using upper case strings, as we have previously seen. Python has no way of assigning a value to a string outside NoLog, and as far as the Python interpreter is concerned, those variables will really only be just strings. However, we have seen an example in listing 6.4 where the **`is`** operator is used to alter or instantiate variables to different values. NoLog's syntax does not currently support this, but an example of the possible syntax is offered in listing 6.5. A possible solution to this issue is to make the scope of instantiations inside a predicate's body accessible to the user programmatically, rather than keeping it only in the backend functionality.

NoLog does not use full-on backtracking the way Prolog does, nor does it offer a fully iterative solution based on filtering as initially hoped. However, I still believe that it is a problem worth looking into. A basic filtering approach was defined for the initial predicate representation, but scaling proved to be more difficult. The new predicate representation and more robust unification, together with the delayed evaluation provided by the Knowledge Engine, may very well represent, if not everything required, at least a step in the right direction to support this approach.

The computational model of NoLog currently offers two separate execution paths when dealing with predicates, as a direct consequence of the current hybrid approach, complicating debugging and design briefly. A unified solution must be discovered which aims to better integrate with the knowledge base model as well as general rule definitions. I believe this must be done quite carefully to fully take advantage of NoLog's unusual evaluation and ensure optimal performance. It may well require some severe refactoring and redesign.

To offer some more familiar ground to the Prolog user base, NoLog could try to support query execution inside the Python prompt. Developers could define their NoLog code inside a Python file which may then be imported as a module inside the prompt. This should then allow developers to pose queries directly inside the prompt. However, while this is the standard Prolog way,

the Python prompt is rarely used, making this feature less of a priority when compared to the other ones.

Finally, we have previously mentioned that NoLog currently does not support tracing, meaning that developers have no way of peeking at the execution of a query. While it is quite an important feature (a lot of developers are using it for catching bugs), it has been marked as an enhancement, rather than a core requirement. There are two possible alternatives at the moment. Firstly, developers could access the backend code of NoLog and perform manual debugging and tracing using prints or logging. However, this is not desirable, as they may unintentionally end up altering the functionality of NoLog. Secondly, Python's native debugger [10] could be used and an example of the Python trace for the same `arcs` as in listing 4.15 is offered in listing A.3. Unfortunately it does not offer too much detailed information, thus supporting the need for a dedicated NoLog tracing mechanism.

These are the most important features that either require some work or need to added entirely. Some, like the syntax for instantiating variables using the **`is`** operator, are missing and are considered core Prolog features, with numerous use cases, therefore taking priority over updates to the knowledge engine or supporting prompt execution of queries.

## 7.3. Closing remarks

NoLog allows developers to leverage the power of logic programming in Python without the need for using bridges or learning a new syntax and using an external interpreter on top of Python's own. NoLog provides a Pythonic syntax and feel for performing all of the required computation, reusing some of Python's core features, while preserving a syntactic resemblance to Prolog. Thus, NoLog successfully bridges the gap between two different paradigms and expands Python's native capabilities.

# A. User Guide

While we have showcased examples of how different concepts look like, this appendix aims to represent a more compact user guide. First of all, to be able to use NoLog in your Python module, you must import the Knowledge Engine and instantiate it. Afterwards, you may begin to declare predicate and perform queries. If a predicate takes more than on argument, the arguments need to be passed in as a tuple. An example is offered in listing A.1.

Rules are defined the same way as predicates. Variables are represented by upper case strings, or in the case of ListWraps, also as upper case strings beginning with a *.

More complex queries using list pattern matching can be performed as in listing A.2. Besides the common Prolog ones, we can also perform something more unusual, courtesy of Python's unpacking operator. The last listing showcases such an example, for which in Prolog we would have to recursively traverse the list until it unified finitely.

Using Python's debugger can be done by importing the `pdb` module and calling `pdb.set_trace()` at the position in the code where you want tracing to commence A.3. More information can be found in Python's documentation regarding tracing [10].

**Listing A.1** Basic predicate operations

```
from functools import partial
from knowledge_engine import KnowledgeEngine

ke = KnowledgeEngine()

arc = partial(ke.predicate, 'arc')

arc(('a', 'b'))
arc(('X', 'Y'), [('arc', ('Y', 'X'))])
# => we now know both arc('a', 'b') and arc('b', 'a')

# will not print possible instantiations out, but will be equal to X: ['a', 'b'] and Y: ['b', 'c']
ke.query('arc', ('X', 'Y'))
# will also print possible instantiations one by one. ';' asks for more, '.' terminates

ke.query('arc', ('X', 'Y'), True)
```

**Listing A.2** ListWrap examples

```
from functools import partial
from knowledge_engine import KnowledgeEngine

ke = KnowledgeEngine()

parent = partial(ke.predicate, 'parent')

parent('john', ['a', 'b', 'c'])
parent('mandy', [])
parent('groot', ['baby groot'])
parent('lily', ['a', 'b'])

# Find all parents X with exactly two children, Y
    and Z
ke.query('parent', ('X', ['Y', 'Z']))

# Find all parents X with any children
ke.query('parent', ('X', ListWrap(['*Y'])))

# Find all parents with at least 1 child
ke.query('parent', ('X', ListWrap(['Y', '*Z'])))

# Find all parents with at least 2 children, where
    the second one is 'b'
ke.query('parent', ('X',
                    ListWrap(['Y', 'b', '*Z'])))

# Find all parents X whose youngest child is 'c'
ke.query('parent', ('X',
                    ListWrap(['Y', '*Z', 'c'])))
```

**Listing A.3** Python tracing mechanism

```
arc = partial(knowledge_engine.predicate, 'arc')
arc(('a', 'b'))
arc(('b', 'c'))

knowledge_engine.eval()
pdb.set_trace()
print(knowledge_engine.query('arc', ('X', 'Y'),
    False))

# tracing output
-> knowledge_engine.query('arc', ('X', 'Y'), False)
OrderedDict([('X', ['a', 'b']), ('Y', ['b', 'c'])])
--Return--
```

# B. Test code example for negation as failure

The code in listings B.1, B.2 and B.3 represents one of the test cases for negation as failure. After execution terminates, it should compute that both 'bill' and 'sam' can sing while only 'bill' can fly. It also showcases the use of negation as failure and how the syntax differs from positive body predicates.

**Listing B.1** Negation as failure test case part 1

```
1  ke = KnowledgeEngine()
2  ke.knowledge_base = OrderedDict()
3  ke.rules = OrderedDict()
4
5  can_fly = partial(ke.predicate, 'can_fly')
6  bird = partial(ke.predicate, 'bird')
7  ab_bird = partial(ke.predicate, 'ab_bird')
8  ostrich = partial(ke.predicate, 'ostrich')
9  penguin = partial(ke.predicate, 'penguin')
10 ab_ostrich = partial(ke.predicate, 'ab_ostrich')
11 ab_penguin = partial(ke.predicate, 'ab_penguin')
12 magic_ostrich = partial(ke.predicate, '
       magic_ostrich')
13 ab_magic_ostrich = partial(ke.predicate, '
       ab_magic_ostrich')
14 feathers = partial(ke.predicate, 'feathers')
15 ab_bird_feathers = partial(ke.predicate, '
       ab_bird_feathers')
```

**Listing B.2** Negation as failure test case part 2

```
1  can_sing = partial(ke.predicate, 'can_sing')
2  ab_bird_singing = partial(ke.predicate, '
       ab_bird_singing')
3  ab_ostrich_singing = partial(ke.predicate, '
       ab_ostrich_singing')
4  ab_it_os_sin = partial(ke.predicate, 'ab_it_os_sin'
       )
5  italian_ostrich = partial(ke.predicate, '
       italian_ostrich')
6  # These empty predicates have to be called to add
       each entry to the rules and knowledge base
7  # If we took more care in ordering the predicate
       definitions, they would no longer be required
8  can_fly()
9  bird()
10 ab_bird()
11 ostrich()
12 penguin()
13 ab_ostrich()
14 ab_penguin()
15 magic_ostrich()
16 ab_magic_ostrich()
17 feathers()
18 ab_bird_feathers()
19 can_sing()
20 ab_bird_singing()
21 ab_ostrich_singing()
22 ab_it_os_sin()
23 italian_ostrich()
```

**Listing B.3** Negation as failure test case part 3

```
bird('bill')
ostrich('bill')
penguin('sam')
magic_ostrich('bill')
ab_bird_feathers('bill')
italian_ostrich('bill')

bird('X', [('ostrich', 'X')])
bird('X', [('penguin', 'X')])

ostrich('X', [('magic_ostrich', 'X')])
ab_ostrich('X', [('magic_ostrich', 'X'),
             ('ab_magic_ostrich', 'X', True)])
feathers('X', [('bird', 'X'),
          ('ab_bird_feathers', 'X', True)])

ab_bird('X', [('ostrich', 'X'),
              ('ab_ostrich', 'X', True)])
ab_bird('X', [('penguin', 'X'),
              ('ab_penguin', 'X', True)])

ab_ostrich_singing('X', [('italian_ostrich', 'X'),
                    ('ab_it_os_sin', 'X', True)])
ab_bird_singing('X', [('ostrich', 'X'),
              ('ab_ostrich_singing', 'X', True)])

can_sing('X', [('bird', 'X'),
          ('ab_bird_singing', 'X', True)])
can_fly('X', [('bird', 'X'),
          ('ab_bird', 'X', True)])
```

# C. Documentation

Examples of documentation for developers in the form of docstrings or inline comments can be seen in listings C.2 and C.1.

**Listing C.1** Inline comment for disputed implementation decision

```
1 ...
2 if isinstance(values, OrderedDict):
3     other_variables[k] = values[k]
4 # This bit may be WRONG! Testing at the end
5 # Leave it like this for now
6 elif isinstance(complete_values, OrderedDict):
7     sols = []
8 ...
```

**Listing C.2** DocString for Predicate function

```
1  '''
2  Function representing a predicate
3  predicate(string, list/tuple, *body) -> Dictionary
4
5  Usage:
6  Can be directly used as a standard function.
      However, to preserve
7  some similarity to Prolog, I tend to create a
      variable called
8  p_name = partial(predicate, p_name). To add a rule
      for predicate func
9  then, one simply calls p_name(p_args, *body).
10
```

```
11 Parameters:
12 -----------
13 p_name : str
14           The name of the predicate you want to
      represent
15 p_args : singleton list, tuple, list[lists]
16           The arguments the predicate takes
17 p_body : list of tuples, optional
18           Takes the form [(p_name, p_args)]
19
20 Returns:
21 --------
22 A dictionary where the keys are the variables from
      func_args and the
23 values are lists of values that satisfy all
      conditions if predicate
24 represents a query
25
26 Note:
27 If the predicate represents an atom, func = "atoms"
       and
28 func_args = [atom].
29
30 The idea is to define the body as a list [(
      pred_name, pred_args)]
31 abiding by the same syntax as the previous
      predicates.
32 '''
```

# D. Change Log

The table D.1 describes the most relevant commits that impacted the semantic versioning, since the beginning of the project. While initial development began prior to the initial commit, the initial commit contains all the prototyping done until then. The description covers the most important changes performed in each commit, but does not necessarily mean they are the only ones that have taken place.

Table D.1.: Semantic versioning change log

| Version | Commit | Date | Description |
|---|---|---|---|
| v0.5.8 | b2610c4 | 06-06-2017 | Bug fixes in rules |
| v0.5.7 | 3649b72 | 04-06-2017 | Robustness of unification & querying |
| v0.5.6 | 5fab657 | 03-06-2017 | Bug fixes with unification |
| v0.5.5 | 71bbf16 | 29-05-2017 | Fixed rules, tests passing |
| v0.5.4 | a4bbcf5 | 29-05-2017 | KE finished, starting on rules |
| v0.5.3 | bfa1b09 | 24-05-2017 | Bug fixes with unification |
| v0.5.2 | bc51bfb | 20-05-2017 | Work on body-less rules |
| v0.5.1 | fe038a2 | 18-05-2017 | Work on knowledge engine |
| v0.5.0 | fd5e6f9 | 18-05-2017 | Support for general rules |
| v0.4.8 | 7f728ca | 16-05-2017 | Cleaning repo and bug fixing |
| v0.4.7 | c10d398 | 16-05-2017 | Bug fixing and complexity reduct |
| v0.4.6 | bb9cd2b | 15-05-2017 | Refactoring of predicates |
| v0.4.5 | e095e87 | 09-05-2017 | Fixed pattern match and list unif |
| v0.4.4 | 3b6e649 | 08-05-2017 | Fixed new unification. TODO: Data rep |
| v0.4.3 | fe8914a | 08-05-2017 | More bug fixes on pattern matching |
| v0.4.2 | b8aed7a | 05-05-2017 | Bug fixes on list pattern matching |
| v0.4.1 | 4f2a016 | 30-04-2017 | PM almost done. Still buggy |
| v0.4.0 | f7849e9 | 28-04-2017 | Work on pattern matching |
| v0.3.6 | d09e428 | 26-04-2017 | Minor bug fixes, added more tests |
| v0.3.5 | 3f899dc | 25-04-2017 | Negation as failure working |
| v0.3.4 | e6532e0 | 15-04-2017 | Bug fixes on NAF pattern matching |
| v0.3.3 | 0e78a40 | 12-04-2017 | Fixed minor NAF bugs |
| v0.3.2 | 5f280fd | 12-04-2017 | Fixed Vincent's preferences example |
| v0.3.1 | d787e7a | 11-04-2017 | Minor instantiation bug fixes |
| v0.3.0 | 374bd48 | 11-04-2017 | Started work on NAF |
| v0.2.5 | 42fbcae | 02-04-2017 | Bug fixes and hacking |
| v0.2.4 | 2caae6a | 02-04-2017 | Supporting multiple term types |
| v0.2.3 | 40b4727 | 26-03-2017 | Added more docs |
| v0.2.2 | a83766b | 19-03-2017 | Added initial documentation |
| v0.2.1 | 2c46070 | 14-03-2017 | Bug fixes and cleaning before NAF |
| v0.2.0 | df2489e | 14-03-2017 | Finished adding queries |
| v0.1.2 | 16baf41 | 12-03-2017 | Minor unification bugs |
| v0.1.1 | 19a58a7 | 08-03-2017 | Backward chaining and resolution |
| v0.1.0 | 39dd36c | 06-03-2017 | Began working on resolution |
| v0.0.8 | e081f09 | 02-03-2017 | Fixed unification, working |
| v0.0.7 | 8b78290 | 23-02-2017 | Unification bug fixes and cleaning up |
| v0.0.6 | afa2906 | 23-02-2017 | Partial unification working |
| v0.0.5 | a401823 | 21-02-2017 | Began work on unification |
| v0.0.4 | bc74ef2 | 14-02-2017 | Updated tests, Travis CI |
| v0.0.3 | 3668a55 | 14-02-2017 | Testing fact add and basic unif |
| v0.0.2 | b4b38fa | 15-01-2017 | Added report, working on unification |
| v0.0.2-alpha | 4c47e92 | 01-12-2016 | Initial unification prototype |
| v0.0.1 | 2a3ab2d | 16-11-2016 | Initial prototyping of kb and pred |

# Bibliography


[1] Clingo.
https://github.com/potassco/clingo.

[2] Codacy nolog review.
https://www.codacy.com/app/DragosDumitrache/NoLog/dashboard.

[3] Code climate.
https://codeclimate.com/.

[4] Codefactor nolog review.
https://www.codefactor.io/repository/github/dragosdumitrache/nolog/overview/master.

[5] InterProlog code examples.
https://github.com/SWI-Prolog/contrib-InterProlog/blob/master/com/declarativa/interprolog/examples/HelloWindow.java.
Date Accessed: 10-01-2017.

[6] Learn prolog now, variables and terms.
http://www.learnprolognow.org/lpnpage.php?pagetype=html&pageid=lpn-htmlse2.
Date Accessed: 10-11-2016.

[7] PyDataLog.
https://sites.google.com/site/pydatalog/home.
Date Accessed: 21-11-2016.

[8] PySWIP bridge.
https://github.com/yuce/pyswip.
Date Accessed: 03-01-2017.

[9] Semantic versioning.
http://semver.org/. Date Accessed: 26-03-2017.

[10] The Python Debugger.
https://docs.python.org/3/library/pdb.html.

[11] GitHub programming language information.
https://en.wikipedia.org/wiki/Prolog, 2012-2014.
Date Accessed: 05-01-2017.

[12] InterProlog documentation page.
http://interprolog.com/wiki/index.php?title=Java-Prolog_bridge, 2014-2016.
Date Accessed: 10-01-2017.

[13] Python unpacking operator.
https://docs.python.org/3/tutorial/controlflow.html#unpacking-argument-lists, 2017.
Date Accessed: 07-05-2017.

[14] Carl Friedrich Bolz, Michael Leuschel, and David Schneider. Towards a Jitting VM for Prolog Execution.
Date Accessed: 13-01-2017.

[15] Michael C. Feathers. Working effectively with legacy code. pages 88–89, September 2004.

[16] David Goodger and Guido van Rossum. Pep257 - docstring conventions.
https://www.python.org/dev/peps/pep-0257/.
Date Accessed: 09-03-2017.

[17] Chris Hogger, Keith Clark, Fariba Sadri, and Murray Shanahan. Introduction to Prolog. 2014. Lecture 1.

[18] Mark Law. Logic-based learning in ASP. 2016. Lecture 6.

[19] Thomas J. McCabe. A complexity measure. 1976.
Date Accessed: 11-06-2017.

[20] Armin Rigo, Maciej Fijalkowski, and Carl Friedrich Bolz-Tereick. PyPy.
https://pypy.org/.
Date Accessed: 12-06-2017.

[21] Matthew Rocklin. Pyke.
http://pyke.sourceforge.net/index.html.
Date Accessed: 21-11-2016.

[22] Matthew Rocklin. Python unification.
https://github.com/mrocklin/unification.
Date Accessed: 17-11-2016.

[23] Alessandra Russo and Mark Law. Logic-based learning. 2016.

[24] Marek Sergot. Knowledge Representation. 2016.
Lecture 2 - Logic databases.

[25] Marek Sergot. Knowledge Representation. 2017.
Lecture 3 - Negation as failure.

[26] Marek Sergot. Knowledge Representation. 2017.
Lecture 6 - Stable Models and ASP.

[27] Marek Sergot. Knowledge Representation. 2017.

[28] Leon Sterling and Ehud Shapiro. The Art of Prolog. pages 12–13, 1986.

[29] Leon Sterling and Ehud Shapiro. The Art of Prolog. page 30, 1986.
Date Accessed: 17-01-2017.

[30] Francesca Toni and Marek Sergot. Introduction to AI. 2014.
Lecture 8 - Knowledge Representation and Reasoning.

[31] Francesca Toni and Marek Sergot. Introduction to AI. 2014.

[32] Guido van Rossum, Barry Warsaw, and Nick Coghlan. Pep8 - style guide for python code.
https://www.python.org/dev/peps/pep-0008/.
Date Accessed: 09-03-2017.