Introduction:

This report will detail the design decisions undergone when implementing Featherweight Prolog, as well as discussing certain strengths and limitations throughout the implementation process.

Featherweight Prolog is a logic programming language that uses resolution and unification to answer queries, given some knowledge base. Keeping this in mind, this report will first touch on the main classes that are used to define the various objects that are used throughout the program, then touch on the choice and implementation of Robinson's unification algorithm, followed by the choice and breakdown of implementation of SLDResolver, and finally the backtracking and tracing logic.

Design and Implementation:

Overview

In a logic programming language like Featherweight Prolog, there exist sequences of facts, rules and queries, of which facts and rules are added to a knowledge base. As a result, utilising the Abstract Syntax Tree (AST) that was given, each component that make up facts, rules and queries, namely, a FPTerm, FPHead and FPBody, were given their own individual classes. These were given along with basic methods like toString to aid with parsing of the language for the program. However, during the implementation of the program, the tedious handling of FPHead and FPBody, especially when passed through the unification and SLD resolution functions, necessitated a need for a helper method to convert them into respective FPTerms that can be handled together, hence toTerm() method was designed to convert any FPHead and FPBody into a valid FPTerm. Moreover, as Robinson's unification algorithm required the comparison of various FPTerms, a custom method to check the equality of 2 FPTerms was created as well, considering the differences in structure between the various FPTerms, such as complex terms, atoms/constants and rules. Such considerations were especially crucial as it covers many of the edge cases, such as atoms/constants having a null args attribute.

On a higher level, the codebase was split into the various stages that is required to interpret and output within the program, namely, Unifier, SLDResolver. Moreover, helper classes such as Substitution was created to better handle the unification algorithm which will be detailed further below in the report. The knowledge base was also implemented separately, with basic helper methods like adding and getting rules and facts. Facts and rules were stored in hashmaps with their names as the keys to facilitate the unification algorithm with more ease. A utils class was created as well to facilitate any additional features that the program required, which includes the renaming of programmer-given variables to avoid variable capture.

Unification

Overall, Robinson's algorithm was chosen due to its soundness and completeness. More importantly, the algorithm will find the most general unifier (MGU) for first-order syntactic unification, which is crucial as there only exists one unique MGU for every solvable unification, emphasising on its completeness property. Before the unification process, any substitutions will be initially applied recursively. The substitution is stored in a mapping that will be populated. An important point to note about the substitution class is the duplication of the substitution maps, which will be touched on further when discussing on backtracking. As for the implementation, different cases were considered, such as unifying constants with another term versus unifying compound terms. Importantly, an occursCheck method was added to ensure that a variable does not unify with a term containing itself, potentially resulting in an infinite structure.

Resolution

In general, SLD resolution was chosen for its high efficiency due to it using Horn clauses. This simplifies our search strategy as well as we perform a linear search. Moreover, the linearity of SLD resolution allows us to backtrack in a systematic manner to find alternate paths and search for all possible solutions to a query. This can be seen in my implementation of SLD which is done through a recursive process, and with multiple saved states of the substitution mapping. Should unification succeed, the recursion will then continue with the updated substitutions. A write function is handled as well to facilitate usability.

Main codebase

The main codebase houses the main interpret function, which will handle the creation of knowledge base and the parsing of the clauses. As it is handled in a top-down manner, the clauses are handled through an iterative manner. The trace logic is also handled here, where it can be inputted by the user for debugging purposes, where every call and fail lines will be printed for the user to outline the logic behind the program. Another important point to note is the utils.resetUniqueNamePool call, to avoid variable capture.

Testing:

My program was run with the sample given test cases and the outputs were checked to ensure its accuracy.

Difficulties:

The implementation of Featherweight Prolog was an arduous one, due to the handling of the different classes that had to be ran through the various algorithms (Robinson's Algorithm, SLD Resolution etc). Many helper functions were created to alleviate the issue, but I believe that there was a more efficient way to handle them and integrate them more smoothly and effectively with the algorithms. Moreover, the edge cases were extremely hard to come up with, and it would be difficult for me to ensure that all grounds were covered to ensure that the program would not bug out.

Conclusion:

Overall, the practical was an extremely fruitful one as it has exposed me to the workings behind a logic programming language such as Prolog and heightened my curiosity and interest for them. Moreover, I now have a more profound understanding of the unification and resolution process that are fundamental to a logic programming language.