**PROJECT IMPLEMENTATION REPORT**

**Contents**

|  |  |  |
| --- | --- | --- |
|  | **Introduction** |  |
|  | **Motivation** |  |
|  | **CoDalog High Level Design** |  |
|  | **Parser** |  |
|  | Parser Tool |  |
|  | Grammar |  |
|  | Safety Rules |  |
|  | Parsed Input |  |
|  | **Engine** |  |
|  | Algorithmic Flow |  |
|  | Naïve Evaluation |  |
|  | Semi Naïve Evaluation |  |
|  | **Query Processing** |  |
|  | **Built-in Predicates** |  |
|  | **Handling Errors** |  |
|  | **Performance Comparison** |  |
|  | **Conclusion** |  |
|  | **References** |  |

**Introduction**

In this project, you and your team are required to design, implement, and test a running prototype for CoDalog, a Concordia deductive database system. In addition to parse, the CoDalog has two other main components for the naive and semi-naive evaluation methods, and ONE additional component which your team could pick among four options suggested for extensions. We recommend to consider the Magi Sets technique which complements and improves the evaluation methods.

**CoDalog High Level Design:**

**Architecture:**

**Tools:**

a user-defined parameter to enforce bottom-up naive evaluation method

parameter to enforce bottom-up semi-naive evaluation method

support magic-sets, built-ins, function symbols, or aggregation operations, whichever one you picked as extension

newly derived facts should be highlighted when displayed (using say different color or adding special symbols like \*)

**Parser**

The parser component is responsible for the following:

* reads the input (a datalog program, say P) already stored in a file with extension (P.cdl).
* checks the syntax of the input and reports possible warnings or errors for any violations including the safety rules, by indicating the positions of those in command line.
* It also stores these warnings and errors to file P.err

A fully implemented parser should have the following items:

* The grammar (Context-free). (can re-use ready-made components)
* supports built-in predicates.
* Options to print the results to file (p.res), warning/error reports, code listings.
* Parser should provide all the errors and warnings at once and not incrementally, that is, it should not stop at every error.

The parser output should be ready to be fed to the evaluation module/component, with options to print to file.

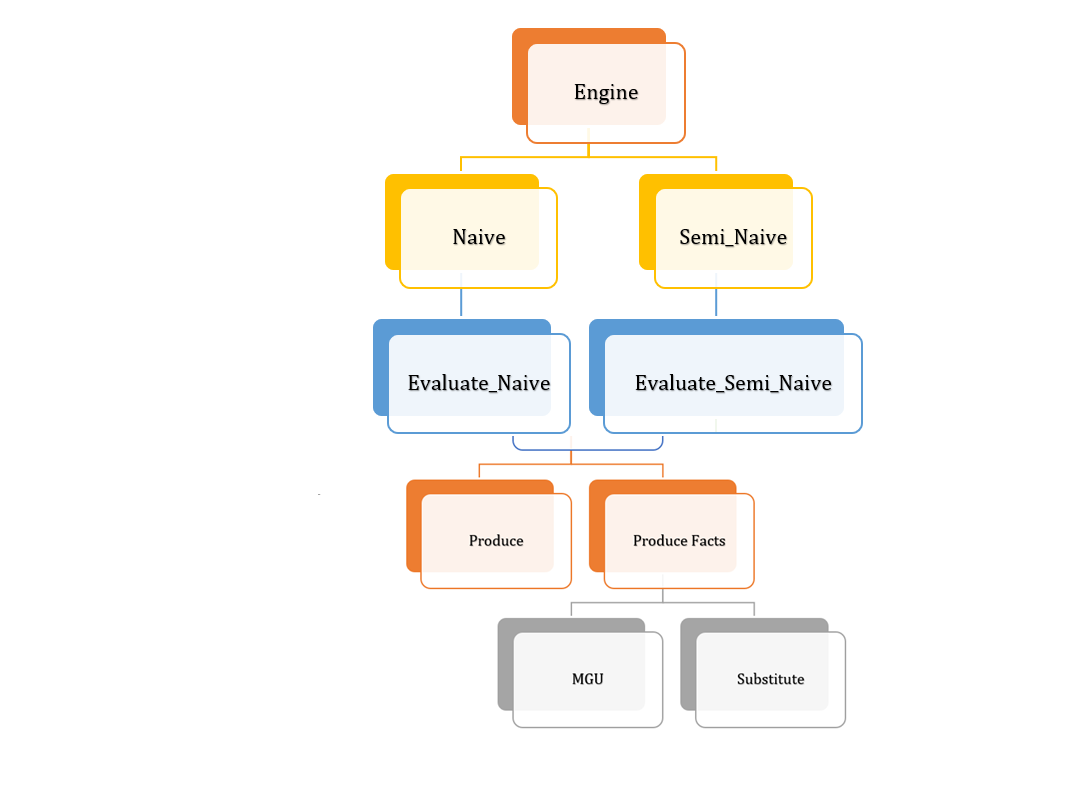
semantic errors (binding errors, not defined predicates, etc)

**Engine**

The CoDalog engine is implemented in the file CoDalog.py.

The parser input is the output to the evaluation engine and is responsible for following tasks.

The engine is divided into total seven functions called in the following sequence in the following manner:



The input to evaluation engine is the parser output, and is responsible for the following:

* take a user query/command
* produce query results (perform inferencing)
* help trace run-time compilations
* handle semantic errors
* return correct answer tuples
* Naïve
* Semi Naïve
* Evaluation Naive
* Evaluation Semi Naïve
* Produce
* Produce Facts
* MGU
* Substitute

**Query Processing:**

Query processing should:

* return answer tuples to user queries.
* be parametric, i.e., the user indicates which run-time environment to pick for query processing/evaluation – the query processing (parsing) should use the parser used for parsing.
* options to print the results to file. options to trace the run-time evaluation (e.g., unification, step-by-step evaluation progress).

Extensions: Built-in Predicates

**Performance Evaluation**

run-time efficiency

scalability

memory utilization

Performance evaluation of the system is the process of running numerous different datalog programs

and sizes to measure various performance criteria related to efficiency and effectiveness of the

system. It helps monitor and study the following issues:

* Error handling: Usefulness of parsing to users through meaningful messages by pointing to the concerned lines and position.
* Correctness: Thorough testing of the results returned for correctness by running least 5 non-trivial programs including linear and double recursion, in particular consider computing transitive closure programs, both linear and non-linear versions.
* Scalability: use different sizes of data and measure the run-time and efficiency of the engine.
* Run-time: include tabulated reports of run-time with different programs, and comment on the results.
* Improvements: show the performance improvement (in percent) for semi-naive over the naive evaluation. Report the same if you are considering the magic sets as an extension.