

# Milestone 5: The Knowledge Base

| 10/3/2022

## 100 Possible Points

Offline Score:  
N/A

 Add Comment

### ▼ Details

Recall from the project overview that a knowledge-base is a set of clauses: facts and rules described by prolog expressions. The purpose of this milestone is to develop the data structures to represent a prolog clause and define the API for the KnowledgeBase class with a partial implementation.

## Clause Structure

A *clause* in prolog represents either a fact or a rule and is composed of two parts: a required *head*, consisting of a single expression, and an optional *body*, consisting of an expression list. This can be simply defined as:

```
struct Clause{
    ExpressionTreeNode head;
    ExpressionTreeNode body;
};
```

## Knowledgebase Class

The class `KnowledgeBase` will be used to hold the clauses that make up our vt-prolog program. The programmer can add a clause to the knowledge-base using a member function call `tell`, and make queries using a member function `ask`. The class also has members to determine the number of clauses, called `size` and the ability to iterate through the clauses in arbitrary order using a constant iterator.

The C++ interface should be:

```
class KnowledgeBase {
public:

    typedef TODO Iterator;

    /// add a clause to the database
    void tell(const Clause & clause);

    /// return the number of clauses in the database
    std::size_t size() const;

    /// get an iterator to the first clause
    Iterator begin() const;

    /// get an iterator to one-past the last clause
```

```
    Iterator end() const;
};
```

where the `TODO` will be determined by what type you use internally to store the clauses but must support forward iteration at a minimum. **Note:** the ask member will be implemented in a later milestone.

To demonstrate how a programmer should be able to use the class, here is example code that iterates through every clause in a knowledge-base `kb` and prints out the expression trees that make up the clauses:

```
KnowledgeBase kb;

for(KnowledgeBase::Iterator it = kb.begin();
    it != kb.end();
    ++it){
    std::cout << it->head.toString() << " :- " << it->body.toString() << ".\n";
}
```

## Instructions

Your task in this milestone is to create the initial `KnowledgeBase` module for VT-Prolog as described above.

- `KnowledgeBase.hpp` should define the classes `Clause` and `KnowledgeBase`, described above, in the namespace `vtp1`.
- These classes and functions should be implemented in the file `KnowledgeBase.cpp`.
- The file `KnowledgeBaseTests.cpp` should contain Catch based unit tests for your `KnowledgeBase` module.

As in previous milestones, you can define additional units of code (classes, helper functions) in `KnowledgeBase.cpp`, but do not change the public interface of `KnowledgeBase.hpp` beyond that specified. Remember to add these files in the appropriate place in the `CMakeLists.txt` file.

## Steps to build and run the tests in the reference environment (after vagrant ssh).

Note: you should be working primarily on your host system for development. These steps are just to check the code in the reference environment, as well as test code coverage and memory safety.

- To configure the build

```
cmake /vagrant
```

- To run the build

```
make
```

or

```
cmake --build .
```

- To run the unit tests

```
make test
```

or

```
cmake --build . --target test
```

The test output is placed in ``Testing/Temporary/LastTest.log. You can also just run the tests directly:

```
./unit_tests
```

- To run the memory checks on your unit tests.

```
cmake -DMEMTEST=TRUE /vagrant
```

followed by

```
make memtest
```

or

```
cmake --build . --target memtest
```

This should report no “definitely lost” leaks and no access errors. Look for

```
LEAK SUMMARY:  
definitely lost: 0 bytes in 0 blocks
```

or

```
All heap blocks were freed -- no leaks are possible
```

and

```
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

- To run your tests and check code coverage.

```
cmake -DCOVERAGE=TRUE /vagrant
```

followed by

```
make coverage
```

or

```
cmake --build . --target coverage
```

This will create a directory called `Coverage_Report` in the build directory. You can copy it back to your host by doing `cp -r Coverage_Report /vagrant`. Open the `index.html` file inside it to examine the detailed report.

- To configure and run the build in strict mode (increased warnings, warnings become errors)

```
cmake -DSTRIC=True /vagrant
cmake --build . --target clean
cmake --build .
```

Note: you can also combine all the options into one build configuration, i.e.

```
cmake -DSTRIC=True -DCOVERAGE=TRUE -DMEMTEST=TRUE /vagrant
cmake --build .
cmake --build . --target test
cmake --build . --target memtest
cmake --build . --target coverage
```

## Submission

You should add changes to your repository and push them to GitHub regularly to demonstrate progress. When you are ready to submit your assignment:

1. Tag the git commit that you wish to be considered for grading as “m5”.

```
git tag m5
```

or

```
git tag m5 COMMIT
```

where COMMIT is the commit id you wish to tag.

2. Push this change to GitHub

```
git push origin m4
```

If you need to tag a different version of your code simply create and push a new tag appending a full stop followed by a monotonically increasing number, e.g. m5.1, m5.2, etc.

Be sure you have committed all the changes you intend to. Before the milestone due date it is a good idea to re-clone your repository into a separate directory and double check it is what you intend to submit. **Failure to complete these steps by the due date will result in a failed submission.**

## Milestone Assessment

Milestone 5 will be assessed according to the following criteria.

Your code compiles in the grader/reference environment	1 point
Your tests pass	1 points
Your tests show no memory errors	1 points
Your code coverage	5 points
Your code compiles in strict mode	1 points
Your code compiles with instructor tests	1 points
Your code passes instructor tests	20 points
Instructor tests show no memory errors	5 points

Note, if your code does not build in the reference environment you will receive **no points**. Correctness is determined by the proportion of instructor unit tests that pass. Code quality will be assessed in this assignment by ensuring your code compiles cleanly, with no warnings, using the strict mode specified above, there are no memory errors (leaks, use before initialize, etc), and proportion of your tests that cover your code (code coverage), which should exceed 98% of lines.

Your milestone can be **checked in the auto-grader (<https://grader.ece.vt.edu>)**. The auto-grader uses the exact same environment as the reference so if your code compiles there, it should in the auto-grader as well (but check anyway!). You are rate-limited to only four submissions every hour to the auto-grader so as to prevent you from using it as your development environment and encourage proper debugging skills.