

Milestone 6: Parsing the Knowledge Base

| 10/11/2022

100 Possible Points

 Add Comment

▼ Details

The purpose of this milestone is to finish the parser by adding the ability to parse the knowledge base grammar.

Parsing Knowledge-bases

Recall the full BNF grammar for our vt-prolog language:

```
<knowledgebase> ::= <clause list>
<clause list> ::= <clause> { <clause> }
<clause> ::= <expression> '.' | <expression> ':-' <expression list> '.'
<query> ::= <expression list> '.'
<expression list> ::= <expression> { ',' <expression> }
<expression> ::= <atom> | <atom> '(' <arg list> ')'
<arg list> ::= <arg> { ',' <arg> }
<arg> ::= <atom> | <variable> | <expression>
<atom> ::= <lowercase letter> { <character> }
<variable> ::= <uppercase letter> { <character> }
<lowercase letter> ::= a | b | c | ... | x | y | z
<uppercase letter> ::= A | B | C | ... | X | Y | Z | _
<character> ::= <lowercase letter> | <uppercase letter> | <numeral>
<numeral> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

This describes a knowledge-base as an unordered list of clauses. Each clause is either a fact or a rule. An expression followed by an **END** token is a fact. An expression followed by an **IMP** token, followed by an expression list, followed by and **END** token is a rule. As in milestone 5 you will be able to use much of your existing parser code.

parseKnowledgeBase functions

This should be a free function with the following definition.

```
/// Given a TokenList from lexer, attempt to parse as a vt-prolog knowledge base.
///
/// If returned ParseError object is set, then an error occurred and the KnowledgeBase
/// may be incomplete.
std::tuple<ParseError, KnowledgeBase> parseKnowledgeBase(const TokenList& tokens);
```

Also write a function to parse queries directly from a string, using the `tokenize` function from milestone 1 internally and calling the above function. It should have the definition:

```
/// convenience function to parse a knowledgebase directly from string
std::tuple<ParseError, KnowledgeBase> parseKnowledgeBase(const std::string& input);
```

The above class and function definitions should be placed in the file `Parser.hpp` and implemented in `Parser.cpp`. You can (and should) implement any additional classes or functions inside `Parser.cpp`, but do not expose them in the `Parser.hpp` header.

Unit Tests

You should modify `ParserTests.cpp` to hold the unit tests for your module additions. You should use these tests to help drive your development process as described in meeting 5. These should cover as close to 100% of your Parser module code as possible.

Note the tests directory in the repository contains several examples of knowledge-bases you should be able to parse.

Instructions

Your task in this milestone is to finish the `Parser` module for VT-Prolog as described above.

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

As in previous milestones, you can define additional units of code (classes, helper functions) in any implementation (.cpp) file, but do not change the public interface of any headers (.hpp) beyond that specified.

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 “m6”.

```
git tag m6
```

or

```
git tag m6 COMMIT
```

where COMMIT is the commit id you wish to tag.

2. Push this change to GitHub

```
git push origin m6
```

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. m6.1, m6.2, 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 6 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 in strict mode	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.