Code review for "g-dosa":
using "codesample1.cpp" from folder "d".

- setUpConfigurationFromJSON function (line ~4):
    - takes in a Config* pointer, but immediately dereferences and uses it without checking if it is NULL/nullptr. Perhaps taking in a mutable object reference might be a better idea? As it stands right now the function looks ready to cause a segfault unless everyone is VERY careful.

    - There is a code snippet on lines 10 through 17 that set booleans in an if/else statement. That seems wasteful to me... Since we are comparing item.value() to false, we know item.value() is a boolean. Why not `gameConfiguration->setAudienceEnabled(item->value())`?

    - since setUpConfigurationFromJSON modifies lots of intrinsic attributes to the hidden Config class, why not make setUpConfigurationFromJSON a member function of that class rather than passing a pointer into the independent function?

- InitializeConfigWithParams test (line ~52):
    - calls a hidden function InitializeConfig and then manually sets a bunch of parameters on the gameConfiguration pointer. I'm not sure what InitializeConfig does if those params need to be set after the fact. Maybe take in the parameters and make those setter calls in initializeConfig?

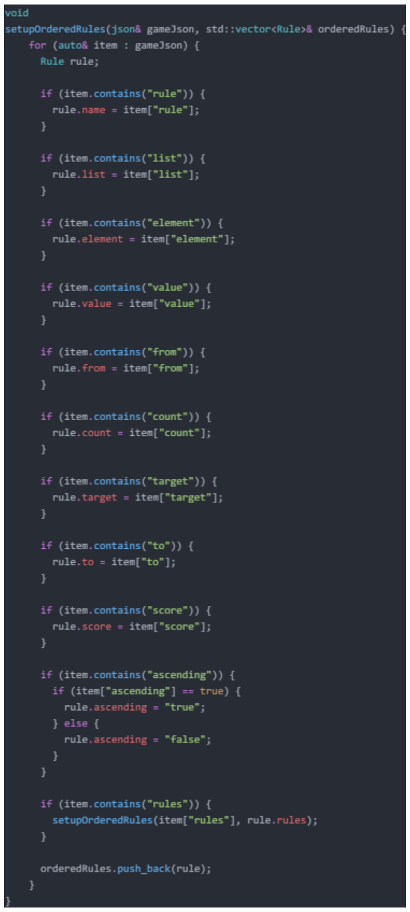
    - Once again, a Config pointer is used without first checking it is not NULL/nullptr. I'm starting to get worried I will be seeing Config pointers more and more…

    - Call to ASSERT_TRUE(gameConfiguration) is useless, rendering the test uninteresting. If gameConfiguration is null the assert would be false, but if gameConfiguration is null the prior calls to setters will result in dereferencing and using a null ptr (segfault time). Therefore ASSERT_TRUE(gameConfiguration) will always be true, so long as the code reaches that line.

- TEST_F test (line ~64):
    - OK, from this test I can see that you have an InitializeConfigWithParams function. It seems ambiguous as to why both InitializeConfig and InitializeConfigWithParams could be useful, but that's up to your implementation. You may consider renaming InitializeConfigWithParams to be an overload of InitializeConfig; but that's a matter of preference.

    - Call to ASSERT_NE for some value compared against false. Wouldn't ASSERT_FALSE and just the value be a more fitting call?

- createGame function (line ~149):
    - at the end, the server goes into a while loop of some sort performing serverTicks and updating a window. I did not expect the main server loop to be located in a function called "createGame" - maybe consider renaming the function to better expose its functionality...

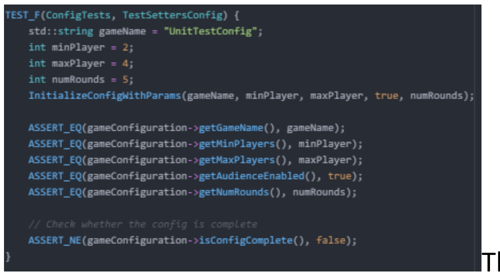


```cpp
void
setupOrderedRules(json& gameJson, std::vector<Rule>& orderedRules) {
    for (auto& item : gameJson) {
      Rule rule;

      if (item.contains("rule")) {
        rule.name = item["rule"];
      }

      if (item.contains("list")) {
        rule.list = item["list"];
      }

      if (item.contains("element")) {
        rule.element = item["element"];
      }

      if (item.contains("value")) {
        rule.value = item["value"];
      }

      if (item.contains("from")) {
        rule.from = item["from"];
      }

      if (item.contains("count")) {
        rule.count = item["count"];
      }

      if (item.contains("target")) {
        rule.target = item["target"];
      }

      if (item.contains("to")) {
        rule.to = item["to"];
      }

      if (item.contains("score")) {
        rule.score = item["score"];
      }

      if (item.contains("ascending")) {
        if (item["ascending"] == true) {
          rule.ascending = "true";
        } else {
          rule.ascending = "false";
        }
      }

      if (item.contains("rules")) {
        setupOrderedRules(item["rules"], rule.rules);
      }

      orderedRules.push_back(rule);
    }
}
```

There are a conditionals if's which are fine since your code is just

setting up the ordered rules. The prof did mention a way of using mapping to help condense

the amount of code you have.

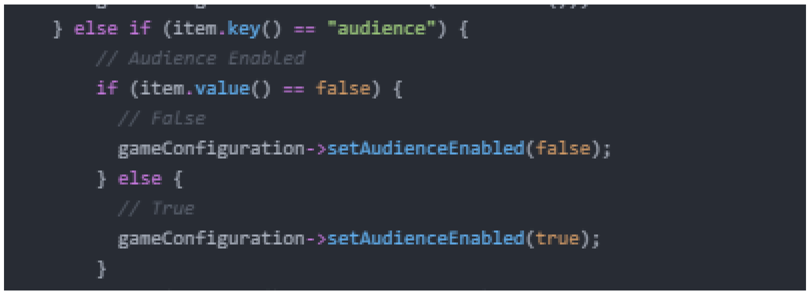


```cpp
TEST_F(ConfigTests, TestSettersConfig) {
    std::string gameName = "UnitTestConfig";
    int minPlayer = 2;
    int maxPlayer = 4;
    int numRounds = 5;
    InitializeConfigWithParams(gameName, minPlayer, maxPlayer, true, numRounds);

    ASSERT_EQ(gameConfiguration->getGameName(), gameName);
    ASSERT_EQ(gameConfiguration->getMinPlayers(), minPlayer);
    ASSERT_EQ(gameConfiguration->getMaxPlayers(), maxPlayer);
    ASSERT_EQ(gameConfiguration->getAudienceEnabled(), true);
    ASSERT_EQ(gameConfiguration->getNumRounds(), numRounds);

    // Check whether the config is complete
    ASSERT_NE(gameConfiguration->isConfigComplete(), false);
}
```

The testing case here is a good typical case but you should try some more special cases

for example if minPlayer = -5. This could be good if down the line your project wants

users to input min/max players



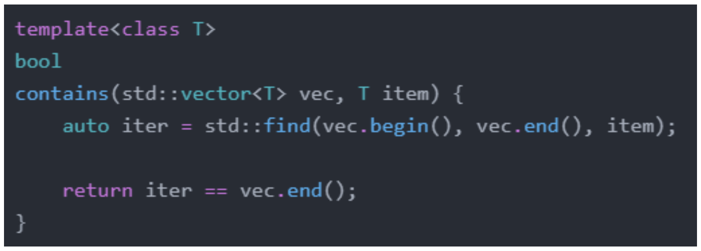
```
    } else if (item.key() == "audience") {
        // Audience Enabled
        if (item.value() == false) {
          // False
          gameConfiguration->setAudienceEnabled(false);
        } else {
          // True
          gameConfiguration->setAudienceEnabled(true);
        }
```

A way you can remove the nested if would be if replaced lines 10 - 17 with

else if(item.key() == "audience") {

**gameConfiguration->setAudienceEnabled(item.value());**

}



```
template<class T>
bool
contains(std::vector<T> vec, T item) {
    auto iter = std::find(vec.begin(), vec.end(), item);

    return iter == vec.end();
}
```

I think you are returning the wrong value for the function contains

what you want is return **iter != vec.end()** rather than return iter == vec.end()

what you have is when your iterator is at the end of the vector return true. This means that

the item does not exist in the vector

Review for Team Dosa:

1. Deeply nested if/else statements
   - in function: void setUpConfigurationFromJSON(const json& gameJson, Config* gameConfiguration). Lines 9 - 17 don't seem like they should be nested for sure. Potential solution: could refactor into its own function.

2. code redundancy.
   - line 9. instead of if (item.value() == false), this could be converted to if (!item.value()). It's already a boolean so there's no need to check whether it equals false.

3. redundant implementation.
   - when counting the player count from line 20 - 28, they use an if/else statement to get the min and max count of players. You don't need an if / else to check for this. the json library (assuming they're using nhlomman) handles it.
   - I don't understand the need for the for loops at line 20 and 31. There is only one instance of "player count" and "setup" so I genuinely don't understand the need for the loops.
   - for both points mentioned above, using the json parser library properly can fix them. right now, the code just looks cluttered because of the for loops and nested if/else

4. Testing
   - The testing is a very smart thing to implement ! My group completely forgot about this and the tests are a good sanity check so good for team Dosa. The tests seem like basic unit tests and nothing fancy which makes sense. Maybe using google tests and CI (from our previous exercise) could benefit them more.

5. Rule interface
   - Having a rule interface makes sense and ensures uniformity amongst the rules. It would've been nice to see what was in the interface.

6. lots of stringly typed information
   - The rule strings (ex. "list", "element", etc.) could be placed in a structure like an enum. They're constants and having them as strings just leaves room for coding / spelling errors.
   - This is said specifically for the function void setupOrderedRules(json& gameJson, std::vector<Rule>& orderedRules)

## Code review for dosa

**1) Lines 4-39**

The ifs and for loops being deeply nested, with the additional placement of comments that just echo what the code already shows, makes the code hard to read. First of all, some of the code is not indented properly - such as the if statement within the for loop on line 20, or the bracket on line 38, making it harder to understand which for loop is within what if, or what if is within what for loop, or even where an if statement or for loop ends. Even if they were indented properly, because of the deep nesting, the code would still be hard to read. Moreover, the code seems clear enough that the comments seem unnecessary, and only provide more clutter to it all. For instance, on line 9, item.key() == "audience" already shows that it is dealing with the audience portion of the game, and the comments on lines 12 and 15 stating "True" and "False" is highly unnecessary, as the code clearly shows that the audience is being set to true and false directly below that comment. Additionally, although the "setupConfigurationFromJSON" function is clearly setting up the game configuration from the JSON, there is too much going on within it. It is setting up the game name, the audience, etc.

My suggestion for improvements in readability is as follows:
break the "setUpConfigurationFromJSON" function into smaller functions, one to 'setup' each portion of the game. This would remove the deep nesting, make the code clear enough to forego comments, and would make each function (including the "setupConfigurationFromJSON" function) have one, overall clear purpose. This would improve readability significantly. See the following example of what that could look like below:

```
void setUpConfigurationFromJSON(const json& gameJson, Config* gameConfiguration) {
        for (auto& item : gameJson["configuration"].items()) {
                if (item.key() == "name") {
                         setupGameName(item);
                } else if (item.key() == "audience") {
                        setupAudience(item);
                } else if (item.key() == "player count") {
                        setupPlayerCount(item);
                } else if (item.key() == "setup") {
                        setupRounds(item);
                }
        }
}
```

**2) Lines 148-180**

This "createGame" function does not have one clear purpose, unlike what its simple name implies. It has too many responsibilities. It seems to be parsing a json object, creating a game object, opening up a chat window, and then running the game object it created on that chat window. Even on lines 161-164, the place of the comment stating that code was omitted because it was too long for a code review shows that this function is doing way too much. It needs to be broken apart into several functions to make it easier to understand, as well as to make refactoring and changes easier in the future. Having to focus on changing one smaller function with a clear purpose would be much easier than trying to figure out what a giant function with many purposes is up to, and would be less prone to error. I would suggest breaking it into the following functions: "createGameObject" and "runGame". Then, possibly break those functions into even smaller functions such as "parseGame" within "createGameObject" or "setupChatWindow" in "runGame", for better readability and maintainability, overall. I would also recommend creating booleans to use on line 160, as something like "inputNotEmpty && inputEndsWithJson" would be much clearer to read than trying to figure out what that line of code does, and would eliminate the need for the comment above it as well.

**3) Lines 85-140**

Although there are multiple if statements within the function "setupOrderedRules,” the design of the function is amazing, overall. The if statements are organized in such a way that the code is easy to understand, would be easy to modify, and differ enough from each other that you can justify creating the multiple if statements rather than by using a single function or switch statement, for example, to reduce repetitive code. Since the if statements are all structured the same way, the code is easy to follow. If you want to add in or delete another game variable, you can easily just add in or delete an if statement, making refactoring easy. My only suggestion for improvement would be to perhaps to use constants instead of hard-coded strings. Since each string is used at least twice for each if statement, the use of string constants would reduce the possibility of accidentally spelling a string wrong, for instance, thus reducing the possibility for errors.

**4) Lines 186-202**

The design and code construction of these two functions "contains" and "collects" are not only very clear, with each function giving a single purpose, but it is very flexible, as well. The use of templates for a list class makes sense, as different lists can store different types. Moreover, the use of iterator and vector functions already provided in given C++ libraries reduce the chance of errors there would be if the code has been implemented by the programmers themselves. The use of variable names "iter" and "result" are clear enough to be read without comments. Overall, I can offer no suggestions on improvement, as the code is concise, clear, each function has a clear purpose, and the use of templates/vectors/iterators actually adds purpose to the code, simplifying it and reducing the possibility of errors, overall.

**5) Lines 45 - 79**

The variables described in lines 48-50 seem to have no clear purpose, or even worse, have a purpose that is unnecessary and confusing if the variables were actually used within the test cases. For instance, if you were to call the variables "DEFAULT_NAME" or "DEFAULT_PLAYERS" within your code as arguments within the "InitializeConfigWithParams" function, or even just to use with basic setters, it would be highly unclear that those "DEFAULT" variables are actually assigning invalid values to the gameConfig object. It would be misleading, as "DEFAULT" could make you think of valid values being used - like 0 for "DEFAULT_PLAYERS", for instance - instead of invalid values like -1. Moreover, these "DEFAULT" variables seem like they may be used to create a dummy gameConfig object, so in that case, why not add another function within the class ConfigTests instead? Having a function called "InitializeDummyConfig" would be much clearer than using those "DEFAULT" variables throughout the test cases. Plus, you get the additional benefit of saving typing in more repetitive code by only having to call one function with no parameters, rather than calling the "InitializeWithConfigParams" and passing in all those "DEFAULT" variable as arguments. Thus, my recommendation for improvement is to use another function rather than those variables directly.

## Code review on sample 1

### Mixed formatting

I'm not talking about weird indentations as it may cause by copying. The code has no line limit, no consistent return type on a separate line or not, mixing tabs and spaces, some code indented by step of 2 spaces and some indented by step of 4, etc. Without a clear formatting standard, over time, the code will become hard to read, affecting maintainability. I would suggest to enforce a standard using `clang-format` and make formatting check a part of CI.

### Code duplication

```
void setUpConfigurationFromJSON(const json& gameJson, Config* gameConfiguration) {
    for (auto& item : gameJson["configuration"].items()) {
        ///...
        } else if (item.key() == "player count") {
        // Player Min & Max Counts
        for (auto& item_player_count : gameJson["configuration"]["player
count"].items()) {
            ///...
        }
        } else if (item.key() == "setup") {
            // Setup
            for (auto & item_set_up : gameJson["configuration"]["setup"].items()) {
                ///...
            }
        }
    }
}
```

In this function, `configuration` is being used multiple time. It's a good idea to make it a local variable. This avoids updating errors in the future. For example, if json structure is changed, you have to change all `gameJson["configuration"]`, which is error prone.

### Stringly types

```
void
setupOrderedRules(json& gameJson, std::vector<Rule>& orderedRules) {
    //...
    if (item.contains("ascending")) {
        if (item["ascending"] == true) {
          rule.ascending = "true";
        } else {
          rule.ascending = "false";
        }
      }
    //...
}
```

Not sure how `Rule` is implemented but it is storing stringly types. As we learned from the lecture, stringly type is never the answer.

### Rule trying to encode too many things

From the usage in `setupOrderedRules()`, I think `Rule` class is trying to encode all possible rules in a single class. This is can become hard to maintain in my opinion. Rules may have an attribute with the same name but different value types which can break this pattern. I suggest having a base common `Rule` struct and extend it as needed.

### Method trying to do too much (not single purpose)

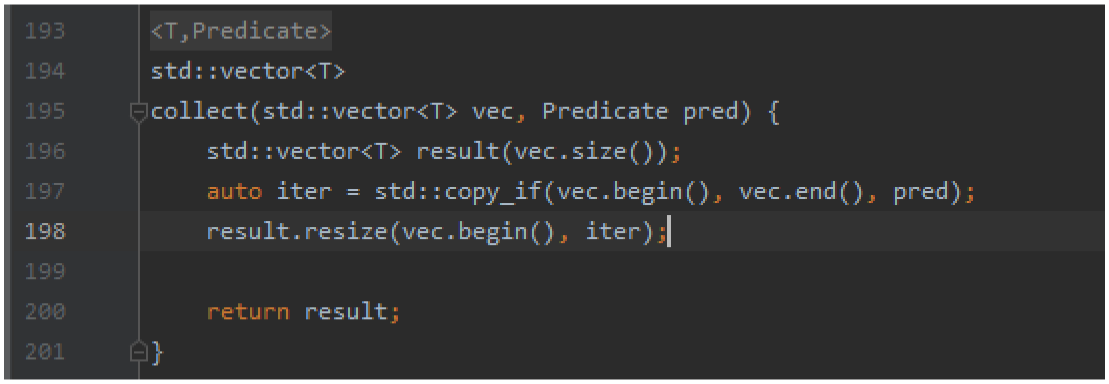
In method `createGame`, the method doesn't just create game, it creates a lambda to populate game json, check for user behavior (check if user has pressed `q`), then passes that whole lambda in to a `ChatWindow` which is a UI related object, then forces server to update until that window is done. Even the lambda is doing input matching and json parsing at the same time(?). This method needs to be broken down into many smaller, single-purpose methods.

### Nitpick

```
class GlobalMessageRule : public Rules
```

Class name should `Rule` instead of `Rules`. `Rules` implies a collection instead of a single object.

1.



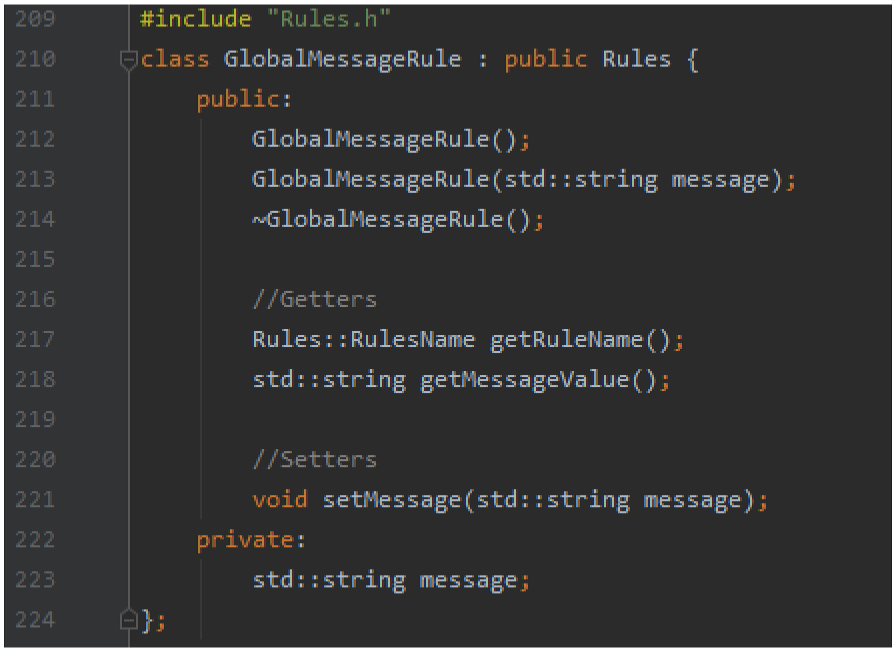
```
<T,Predicate>
std::vector<T>
collect(std::vector<T> vec, Predicate pred) {
    std::vector<T> result(vec.size());
    auto iter = std::copy_if(vec.begin(), vec.end(), pred);
    result.resize(vec.begin(), iter);

    return result;
}
```

Using context clues, it seems that this function is intended to collect all values in a given vector that meet a certain condition *pred*, however the code doesn't achieve that. The code performs the following:

a. creates a new vector that is the same size as the argument vector
b. copies all values from the argument into the new vector *that wasn't passed as an argument*
c. resizes *result* to contain *vec.begin()* elements, and then adds *iter* if there is more room allocated.

This appears to be an attempt to implement the example found at: http://www.cplusplus.com/reference/algorithm/copy_if/ , but with various errors in the implementation.

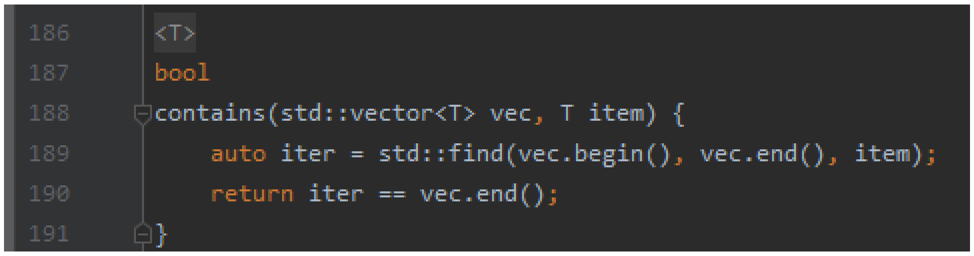
2.



```
#include "Rules.h"
class GlobalMessageRule : public Rules {
    public:
        GlobalMessageRule();
        GlobalMessageRule(std::string message);
        ~GlobalMessageRule();

        //Getters
        Rules::RulesName getRuleName();
        std::string getMessageValue();

        //Setters
        void setMessage(std::string message);
    private:
        std::string message;
};
```

getters/setters can simply be implemented in the .h file rather than in the .c file.

3.

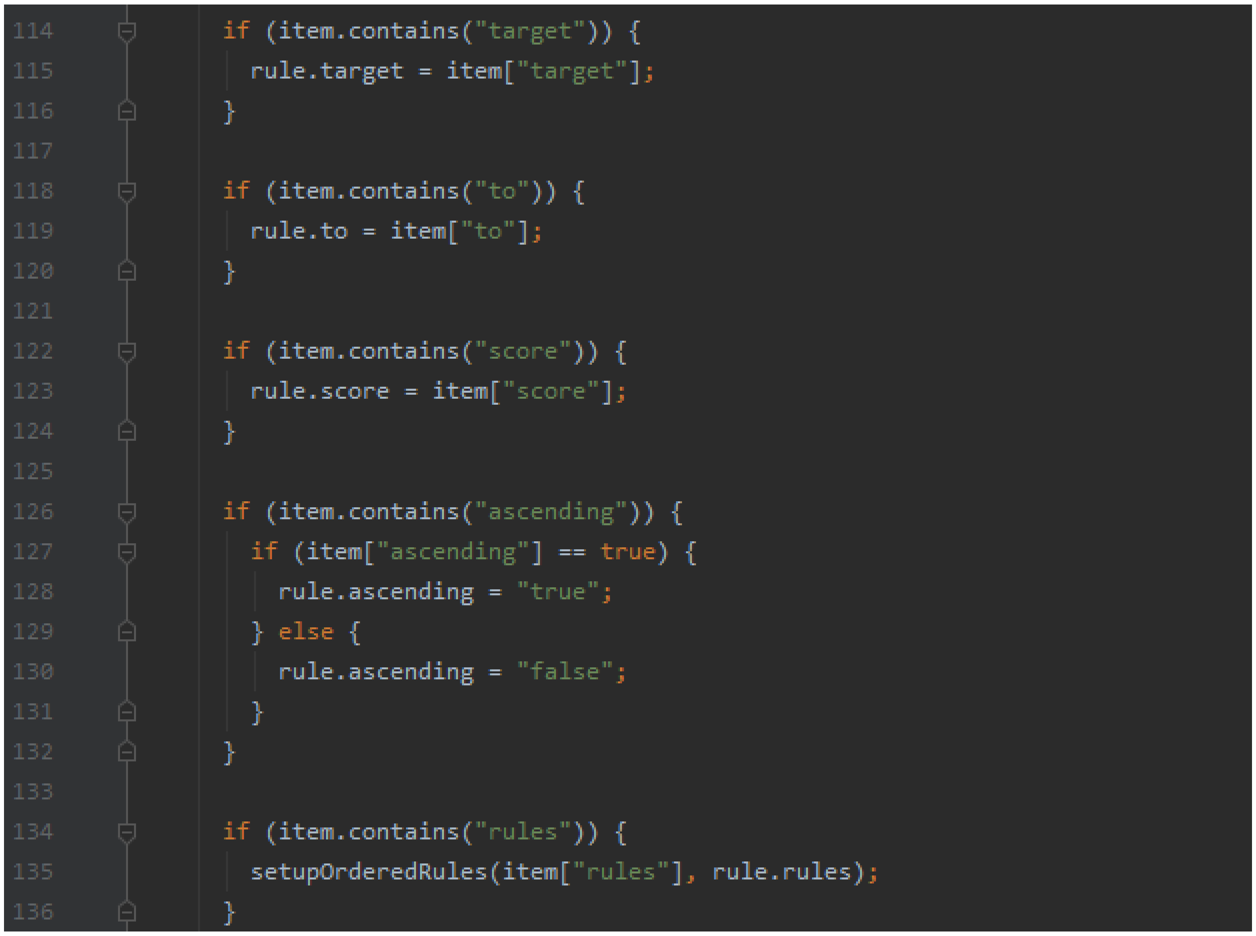


```
<T>
bool
contains(std::vector<T> vec, T item) {
    auto iter = std::find(vec.begin(), vec.end(), item);
    return iter == vec.end();
}
```

can be reworded into one line.

```
return std::find(vec.begin(),vec.end(),item) == vec.end();
```

4.



```
if (item.contains("target")) {
  rule.target = item["target"];
}

if (item.contains("to")) {
  rule.to = item["to"];
}

if (item.contains("score")) {
  rule.score = item["score"];
}

if (item.contains("ascending")) {
  if (item["ascending"] == true) {
    rule.ascending = "true";
  } else {
    rule.ascending = "false";
  }
}

if (item.contains("rules")) {
  setupOrderedRules(item["rules"], rule.rules);
}
```

It looks like the idea is to implement some kind of Mega-Rule object, that has attributes for every possible field. The Mega-Rule would perform functions based on the value located in rule.name, and hypothetically it would never access the irrelevant fields.
i.e.

```
{ "rule": "add",

"to": target,

"value": 2,

}
```

would have default values for its *target , score* fields etc. However, they wouldn't need to be, since the rule-parser will only access the rule/to/value fields.

While this implementation could definitely work, it has the potential to be ridiculously inefficient memory wise, with every single rule allocating memory for over 10 attributes, but never using more than 4 at any given time on average.

5. Going back to the Mega Rule implementation, there are attributes missing for:
    a. until
    b. while
    c. cases
    d. condition

   and more….