CITS5501 Project 2024

Version:	0.1
Date:	26 Sep, 2024

1 Introduction

- This project contributes **35%** towards your final mark this semester, and is to be completed as individual work.
- The project is marked out of 55.

- The deadline for this assignment is **11:59 pm Thu 17 October**
- You are expected to have read and understood the University academic integrity guidelines. In accordance with this policy, you may discuss with other students the general principles required to understand this project, but the work you submit must be the result of your own effort.
- You must submit your project before the submission deadline above. There are significant penalties for late submission (click the link for details).

2 Clarifications and changes to the project specification

You are encouraged to start reading through this project specification and planning your work as soon as it is released. Any queries regarding the project should be posted to the Help5501 forum with the "project" tag.

Any clarifications or amendments that need to be made will be posted by teaching staff in the Help5501 forum.

3 Format and submission of your work

All code and answers to questions should be submitted by making a Moodle submission.

For any long English answers:

- Please structure your answers using numbered headings where appropriate.
- You can use any of the available Moodle editors (Atto, TinyMCE or marked up plain text) by setting your Moodle preferences – more information about this will be provided together with the Week 10 labsheet.

- If you include any diagrams, charts or tables, they must be clear, legible and large enough to read when viewed on-screen at 100% magnification.
- If you give scholarly references, you may use any standard citation style you wish, as long as it is consistent. However, a recommended citation style is "AMS short alpha-numeric" (see AMS style guide, sec 10.3).
- Diagrams, charts, tables, bibliographies and reference lists do not count towards any word-count maximums.

Submitted code should meet the usual guidelines for CITS5501 code:

- code should be clearly written, well-formatted, and easy for others to understand
- function bodies should not contain excessive inline comments
- code should compile without errors or warnings
- code should follow sound programming practices, including:
  - the use of meaningful comments
  - well chosen identifier names
  - appropriate choice of basic data-structures, data-types, and functions
  - appropriate choice of control-flow constructs
  - proper error-checking of any library functions called, and
  - cleaning up/closing any files or resources used.

Any methods (including JUnit `@Test` methods) that you write should include a Javadoc documentation block which follows the guidelines at https://www.oracle.com/au/technical-resources/articles/java/javadoc-tool.html under "Writing Doc Comments".

Note that code which does not compile will be awarded (at most) a very small number of marks. You can check that your code compiles using Moodle; teaching staff will not fix your code if it does not compile.

Your code should never emit any output to `System.out` or `System.err` unless specified in the question. Doing so is extremely poor practice, and will typically result in your code being awarded a low number of marks (and in any case, will likely result in your code failing any checks or tests applied by Moodle).

3.1 Documenting assumptions

If, after posting questions to the Help5501 forum and having received answers, you believe there is still insufficient information for you to complete part of this project, you should document any *reasonable assumptions* you had to make in order to answer a question or write a portion of code.

Moodle will include a question entitled "Assumptions" in which you can list these. Make sure you give each assumption a number, and explain why you think it's reasonable.

Then in your code or other questions, you can briefly refer to these assumptions (e.g. "This test case assumes that Assumption 1 holds, so that we can ...").

An assumption which contradicts anything in the project specification, the Java class specifications or postings made by staff in the Help5501 forum is *prima facie* not reasonable.

4 Background

You are part of the Testing and Quality Assurance (QA) Department at Lensherr, Connors, Doom, and Associates, a leading company in crafting innovative security and automation systems tailored for high-end and niche customers.

Your current project involves extending the **Domotopia** software system. Domotopia is designed to streamline the control and integration of various smart devices and appliances within a home. At the heart of this system is a specialized command and query language (Domolect), that allows users to interact seamlessly with their devices.

4.1 Domolect language

The current version of Domotopia (1.7) uses a language defined by the following EBNF grammar to control smart devices, appliances, and other aspects of a customer's premises.

It allows users to control *lighting devices*, *thermal devices*, *barriers* and *appliances*, each with their own specific options. For instance, a thermostat is a type of thermal device, and can be controlled with a command like

```
set thermostat to 297 K
```

(Temperatures are always specified in Kelvin, to allow users to conveniently and precisely specify a wide range of temperatures.)

Commands can optionally start with a *location*, which isn't defined in the grammar, but will be specific to each customer (e.g. `main-bedroom`, `kitchen`, `laboratory`, `armory`, etc.). If a command is unambiguous, it will not need a location (e.g. if just one thermostat controls the entire premises).

Words in the commands are separated by whitespace (any non-zero-length sequence of whitespace characters). To simplify our grammar, this whitespace isn't included in the EBNF, and we also don't include the rules for constructing *decimal numbers*. (If you need to represent those in a grammar for any reason, you can just assume that the non-terminal `<number>` is available, which for testing purposes is considered to have just one production.) Locations must *not* contain whitespace. If a location consists of multiple words (e.g. "master bedroom"), then in Domolect the words have to be separated by hyphens ("`master-bedroom`").

Domolect 1.7

```
<command> ::= <location>? ( <lighting_command> | <barrier_command> | <appliance_command>
                | <thermal_device_command> )

<lighting_command> ::= "turn" <light_source> <state>
<thermal_device_command> ::= "set" <thermal_device> "to" <temperature>
<barrier_command> ::= <barrier_action> <barrier>
<appliance_command> ::= "turn" <appliance> <state>

<barrier_action> ::= "lock" | "unlock" | "open" | "close"
<state> ::= "on" | "off"
<temperature> ::= <number> "K"

<light_source> ::= "lamp" | "bulb" | "neon" | "sconce" | "brazier"

<barrier> ::= "gate" | "curtains" | "garage-door" | "blinds" | "window" | "shutter"
              | "trapdoor" | "portcullis" | "drawbridge" | "blast-door" | "airlock"

<thermal_device> ::= "oven" | "thermostat" | "electric-blanket" | "incinerator"
                     | "reactor-core"

<appliance> ::= "coffee-maker" | "oven" | "air-conditioner" | "centrifuge"
                "synchrotron" | "laser-cannon"
```

Example command:

If "`master-bedroom`" is a valid location at a customer's premises, then

```
master-bedroom set electric-blanket to 400 K
```

would be a syntactically correct Domolect command.

4.2 Proposed additions to the Domolect language

A new version of Domotopia is being developed, version 2.0, which extends the language above. It allows commands to be augmented with conditions that specify when the command will come into effect ("when" conditions), and conditions that specify when the command will stop being in effect ("until" conditions). Conditions can involve times or temperatures.

As an example, version 2.0 of the language would allow a user to write

```
master-bedroom set electric-blanket to 400 K when current-temperature less-than
300 K until 10:00 pm
```

A command augmented by optional conditions is called an *augmented command*. An augmented command contains a normal command (as defined in version 1.7 of the language, in section 4.1), followed by an optional "when" condition and an optional "until" condition (in that order).

Conditions are either temperature conditions or time conditions. A temperature condition starts with the word "`current-temperature`", is followed by a comparator (either "`less-than`", "`greater-than`", or "`equal-to`"), and then a target temperature in Kelvin (as defined by version 1.7 of the language, in section 4.1). (The implemented system will monitor its sensors to find out when the condition is satisfied.)

Time conditions consist of 2 digits, a colon, 2 digits, and then the words "`am`" or "`pm`". For example, the following are all syntactically valid times:

- `04:39 am`
- `12:54 pm`
- `76:32 am`

4.3 Supplied files

You are supplied with Java code representing the Domotopia system – see the `system-code.zip` file.

This code should compile with any current recent Java compiler or IDE (such as BlueJ, Eclipse or IntelliJ IDEA).

A brief description of some of the classes in the Java code is given below:

Enumerated types: The code includes `BarrierAction` and `State` enum types. The former represents actions (such as "lock", "unlock", "open", and "close") which can be sent to barrier devices, and the latter represents the desired states ("on" or "off") of appliances or light sources.

Command classes: The base class `Command` serves as the abstract foundation for specific command types. It includes a `Location`, but that can be null in cases where the command is unambiguous. For instance, a command to close the curtains is unambiguous if there is only one set of curtains on a premises; but if there are more than one set of curtains, a location needs to be specified.

Subclasses like `LightCommand`, `BarrierCommand`, and `ThermalDeviceCommand` provide implementations for different types of commands that can be executed in the system.

Condition classes: The abstract class `Condition` defines a blueprint for conditions that can be checked for satisfaction. Subclasses such as `TemperatureCondition` and `TimeCondition` allow the system to check e.g. whether a target temperature has been reached or exceeded, or whether a time of day has occurred. Conditions have a `boolean isSatisfied()` method, which gets repeatedly queried by the system; once a condition is satisfied, any `Command`s containing that `Condition` are executed.

Device classes: Several classes represent devices on a customer's premises. For instance, `LightSource` represents a light source (and includes methods to turn the light on or off), `Appliance` represents an appliance (such as an oven or coffee maker), and `Barrier` represents doors, gates and similar devices that can be opened and closed (and, if the relevant device supports this, locked or unlocked).

Detailed specifications for each class and method are provided as JavaDoc comments in the supplied code.

You should **not submit** any of the code currently in the `src` directory. The Moodle test servers will have their own versions of classes in that directory.

5 Tasks

Answers to the following questions should be submitted via Moodle. Some of the questions require you to write English answers; others require you to write code.

Question 1 [10 marks]

Give a BNF or EBNF grammar that will parse (or generate) valid Domolect 2.0 augmented commands, including "when" conditions and "until" conditions.

You do *not* need to (and should not) define non-terminals that already exist in version 1.7 of the language, as defined in section 4.1, but can make use of them if you wish.

You *do* need to define an `augmented_command` non-terminal, and it should correctly include "when" conditions and "until" conditions, as defined in section 4.2.

Your grammar should use the notation accepted by the BNF playground. It must be in plain text which could be pasted into the BNF playground and compiled without error.

(If using the BNF playground to experiment with a grammar, you may find it useful to temporarily put spaces at the end of each word – e.g. `"oven "` instead of `"oven"` – to make commands more readable, but your submitted grammar should not include these spaces.)

Question 2 [4 marks]

Write a suite of test cases for your grammar from question 1 that have *production* coverage. You may assume there is already a suite of test cases for version 1.7 of the Domolect language, and that it is already known to have production coverage.

Your tests will simply be a list of Domolect 2.0 commands (one command per line) that represent augmented commands: this is the input to the test. (The expected result in each case is that the system correctly accepts that command. Since the expected result is the same in all cases, there's no need for you to specify it.)

A thorough set of test suites would test not just that Domolect 2.0 commands *are* parsed correctly, but also that invalid command are correctly rejected. However, for the purposes of this project, we will focus on just the former.

Question 3 [4 marks]

Explain the reasoning behind your choice of tests in question 3, and why it is that those test cases have production coverage. Explain also: are there semantic constraints not covered by your grammar, that would need to be tested before the new Domolect was used

in a production system?

max 1000 words

Question 3 [4 marks]

Would it be possible to exhaustively test the syntax of Domotopia commands (that is, to write tests which have derivation coverage)? Give reasons for your conclusions. (Maximum 500 word answer.)

Question 4 [4 marks]

Describe in detail the preconditions and postconditions of the constructor for the `LightingCommand` class, justifying your answer. (Max 500 words)

Question 5 [4 marks]

Some of the JavaDoc documentation for the project includes example code – for example, the `BarrierCommand` class. Do you think it's important to make sure the JavaDoc examples contain valid, compilable code? Explain why or why not. If we wanted to ensure the examples compiled, how could we do that? Explain what tools you would use.

(Max 500 words)

Question 6 [10 marks]

Apply Input Space Partitioning (ISP) to the constructor for the `LightingCommand` class, explaining the steps you take and what characteristics and partitions you would use.

You should describe at least 3 characteristics.

Describe three test cases in detail, including all fixtures, test values and expected values. Include a test ID for each test case, so you can refer to it in question 7.

(Max 1000 words)

Question 7 [15 marks]

Write a test class using JUnit 5 called `LightingCommandTest`, which contains `@Test` methods which implement the three test cases you described in question 6.

Your test cases should implement all appropriate best practices for unit tests.

Note that to obtain any marks, it is a requirement of the question that:

- you implement three of your question 6 test cases – for each test, include Javadoc documentation providing the test ID from question 6 which it corresponds to. If you implement something else instead, no marks will be awarded.
- your tests properly "Arrange, Act and Assert" your test case. Tests that (for instance) contain no assertions will not be awarded marks.

Some code checks will be made available via Moodle that can be useful to highlight possible problems in your code, but passing them is not a guarantee of any marks being awarded.

5.1 Extension tasks

You may submit an answer to either of the following questions for 3 bonus marks, awarded at the discretion of the unit coordinator based on the coherence and quality of the answer. These 3 bonus marks cannot take your final mark higher than 55.

- How could mutation testing be used to evaluate the quality of your team's tests, and what empirical evidence exists that mutation testing is a sound approach to evaluating test quality? If you conclude mutation testing is not a sound approach, suggest alternatives, and explain your reasoning. Your answer should include appropriate academic references.
- Write a parser and read–eval–print (REPL) loop for the command syntax described in this project specification, in any language of your choosing that can be compiled and run on an Ubuntu 20.04 distribution running on AMD64 processors. Your REPL loop may return canned responses to commands.

  Commands for building and running your program should be uploaded as an image to the Docker Hub or any other publicly available Docker repository, so that the program can be run using a `docker run` command.

  Your Moodle answer should explain what testing approaches you adopted while writing the program, give a URL on a repository for the source code, and include the details of the `docker run` command needed to run it. (Note: ensure you keep your source code repository private until the marker requests access, as this is an individual project.)