Report on the solution of Oblig 4 in IN2031

Student Aazan Nisar (aazann@ifi.uio.no)

November 13, 2025

1 Introduction

This report documents the reference solution of the Oblig4 assignment. I will describe how I arrived at the complete solution for all four assignments but with a focus on this one, which parts were unclear, eventual problems and the tests that were created.

The objective of the assignment was to implement a static type checker, that runs before execution and after parsing, if a program does not comply with the the rules we set we stop execution and return an error message.

To complete this task, i used Gradle, ANTLR, and JUnit. It was essential to understand how each of these tools worked and how they could be combined to solve the assignment effectively.

All requirements of the assignment were met. In the following sections, I will present my approach, discuss limitations, and provide conclusions.

2 Implementation

In the first part of the project i started implementing the first simple mathematical operations, point and range operations in the grammar file, using ANTLR semanthics. I also added the first classes for Execution, Statement, and the numerical operations, i then proceeded to implement the logic for the interpreter, wich visits a file with some simple operations, parses them, tokenizes them, passes to the lexer and feeds to the interpreter. In the interpreter we use the data from the lexer to make the 'rules' for how to translate the grammar.

For the second part the project the task was to create more expanded grammer rules and interpreter, this consisted in implementing support for Actions, a stack for the executions and a heap to save data about initial and final position and battery. I started implementing all the different actions, with their respective Execute function, were we create the logic for how much battery is consumed and update the positions in the heap, it was also important to remember that some of the actions could have a point or range as an argument, and execute accordingly. In the interpreter i implemented a function to visit Executions, Statement and Actions, and save them in a table, wich we can use later to add a Statement to the stack and execute all the actions, and so on. It was important here for the interpreter to be able to recursivly navigate the table with Executions, since some of the Executions gave us the next to Execute, but i also needed to check if this was not going to create an infinte loop.

For the third part of the assignment I had the the task to finish the the grammar and support for Events and Reactions,

2.1 Event Types

1. Low Battery Event

The low battery event triggers automatically when the battery level drops below 20%. The implementation checks the battery level before executing each statement in the execution loop:

2. Obstacle Event The obstacle event works similarly to low battery but checks if a flag. The interpreter checks this flag before each statement execution and triggers the obstacle reaction if present.

3. Message Event Message events are triggered manually from the REPL. During program parsing, the interpreter registers message listeners for all message reactions found in the code. When a message is received via the REPL, the corresponding listener executes and transitions to the target mode.

2.1.1 Reaction Implementation

Reactions are defined in the grammar as:

2.1.2 Event Handling Flow

The event handling follows this pattern:

1. Program starts executing the start mode
2. Before each statement execution, check for events:
   - Check obstacle flag
   - Check battery level
3. If an event is triggered:
   - Find the corresponding reaction in the current execution
   - Clear the execution stack (discard remaining actions)
   - Transition to the target mode
   - Reset event flags if applicable
4. If no event occurs, execute the statement normally
5. If the execution completes and has a next execution we continue with that recursively.

2.1.3 REPL Implementation

The REPL (Read-Eval-Print Loop) allows user interaction after the initial program execution. **5. Listener Registration** Message listeners are registered during the program parsing phase in the Interpreter using a specific funtion that goes trough all Statements and creates a runnable with all the Message events itfinds. This allows the REPL to access all available message handlers without needing to search through executions at runtime.

2.2 Static Type checker

To implement the static type checker i made a new class TypeCheck, and defined the typing rules the following weer used in addition to the rules given in the assignment: xspace

2.2.1 Addition Rules

Rule 1: Num + Num

$$\frac{\vdash_{exp} e_1 : \textsc{Num} \quad \vdash_{exp} e_2 : \textsc{Num}}{\vdash_{exp} e_1 + e_2 : \textsc{Num}}$$

Rule 2: Point + Point

$$\frac{\vdash_{exp} e_1 : \textsc{Point} \quad \vdash_{exp} e_2 : \textsc{Point}}{\vdash_{exp} e_1 + e_2 : \textsc{Point}}$$

2.2.2 Subtraction Rules

Rule 3: Num - Num

$$\frac{\vdash_{exp} e_1 : \textsc{Num} \quad \vdash_{exp} e_2 : \textsc{Num}}{\vdash_{exp} e_1 - e_2 : \textsc{Num}}$$

Rule 4: Point - Point

$$\frac{\vdash_{exp} e_1 : \textsc{Point} \quad \vdash_{exp} e_2 : \textsc{Point}}{\vdash_{exp} e_1 - e_2 : \textsc{Point}}$$

2.2.3 Multiplication Rules

Rule 5: Num * Num

$$\frac{\vdash_{exp} e_1 : \textsc{Num} \quad \vdash_{exp} e_2 : \textsc{Num}}{\vdash_{exp} e_1 * e_2 : \textsc{Num}}$$

Rule6: Point * Point

$$\frac{\vdash_{exp} e_1 : \textsc{Point} \quad \vdash_{exp} e_2 : \textsc{Point}}{\vdash_{exp} e_1 * e_2 : \textsc{Point}}$$

2.2.4 Apply rules

To apply the rules we, as done with the interpreter, we parse through the proram, separating the executions,statements, actions and expressions.

For each we define what they should return and accept:

- Action: for these we decide which Type they should accept, for example a Ascend action should only accept a Type NUM or RANGE.
- Expressions: for the operations, we should accept the Type defined for them in our rules. for example in point expression we should expect for the two internal expressions to be of Type NUM or RANGE, and not POINT.
- Program: if everything is correct and no error is encountered our program returns nothing, and the program is correctly typed, if a section is wrongly typed, we return an error with information about the wrong Type.

3 Testing

3.1 Test from assignment 1 to 3

To test if executions are correctly sorted, the battery and position are coorectly calculated, and all executions are present when vistiting the program the following tests where implemented :

- `getFirstExecution`: Tests that the interpreter correctly orders executions in the program. It creates two `Execution` objects, runs them through `ExcuteProgram`, and then verifies via the key iterator of the `LinkedHashMap` that the first execution is `"start"` and the next is `"next"`.
- `getPosition`: Checks that the initial position stored in `heap` is correctly initialized to `(0.0f, 0.0f)` for both X and Y coordinates.
- `getDistanceTravelled`: Verifies that the distance traveled in the `heap` can be correctly stored and retrieved. Here, the test sets `Memory.DISTANCE_TRAVELED` to `100.0f` and asserts that the value is correctly returned.
- `getBatteryLevel`: Ensures that the initial battery level is correctly set in the `heap` and can be retrieved. The test checks that it equals `100.0f`.
- `visitProgram`: Validates that the interpreter can read and parse a program file (`program.aero`), convert it into `Execution` objects, and produce the correct number of executions. It reads the file, creates a lexer and parser, visits the program, and asserts that the number of executions matches the expected count (`5` in this case). Any exceptions in reading or parsing result in a test failure.

3.2 Test for assignment 4

Many tests were made for assignment4 here are some of them:

- **TestPlusType:** Verifies that the type checker correctly infers the type of a simple arithmetic addition expression. It parses `"2 + 2"`, visits the expression using the `TypeCheck` visitor, and asserts that the resulting type is numeric (`Type.NUM`).
- **TestPlusTypeError:** Ensures that the type checker properly throws an error when an invalid addition is performed between incompatible types. It tests the expression `"2 + point(2,1)"` and asserts that a runtime `Error` is raised, since a number cannot be added to a point.
- **TestMinusWithRangeType:** Checks that subtraction between a numeric literal and a random range expression produces a numeric result. It parses `"2 - random[1,10]"`, visits the expression, and verifies that the resulting type is `Type.NUM`.
- **TestMinusTypeWithPointError:** Tests type error handling for subtraction between incompatible types. The expression `"2 - point(2,1)"` should trigger an error because a number cannot be subtracted from a point. The test asserts that an `Error` is thrown.
- **TestMultiWithRangesType:** Confirms that multiplication between two range-based random expressions results in a numeric type. It parses `"random[1,19] * random[1,10]"`, visits the expression, and asserts that the type checker infers `Type.NUM`.

- **TestActions:** Validates that the type checker correctly handles and accepts valid drone action commands. It tests four different actions — `"ascend by 50"`, `"descend by 100"`, `"move by 1"`, and `"turn by 66"` — ensuring that each is successfully parsed and visited without producing any errors (`visit` returns `null` for all).

3.3 Test Limitations

The automated tests primarily cover the *happy path*, i.e., cases where the input is valid and no errors occur. While trivial edge cases have been manually tested, there are many additional cases that could be added, such as:

- Messages that don't have corresponding listeners
- Obstacle detection with no obstacle reaction defined

Including such tests would make the test suite more comprehensive and robust.

4 Limitations and Potential Problems

4.1 assignment 1 to 3

In the first two assignments, most limitations arose from unclear requirements, such as handling multiple executions and compatibility issues with Gradle 9, which required modifying `MethodTable` to a `LinkedHashMap` and updating `visitExpression`. For the third assignment, major challenges included undefined obstacle detection logic, lack of automatic obstacle generation, and ambiguity in obstacle behavior. I chose not to store reactions and messages in the heap, instead managing them directly through the REPL for efficiency and clarity. Additionally, I avoided making `Execution` extend `Statement`, as I found it unnecessary and preferred handling execution logic within the Interpreter. Overall, the main limitations stemmed from specification gaps and structural design decisions to maintain clarity and modularity.

4.2 assignment 4

Some of the limitaions for assignment 4 were the contraditions between assignment and grammar, for example operations between points and numbers, is not defyined in the grammar given and it is not stated in the assignment to change the grammar. The assignment defines RANGE as type, even if this can be simplyfied to NUM since all range expressions always return a number of type NUM.

5 Conclusions

I have implemented a full DSL, with an expanded interpreter, with wider support for drone movements, events and reactions and a REPL and a type checker. I judge the implementation to be for the most part complete, all the given test are passed and the files are correctly read.