

Oblig 2: The AeroScript Language

Deadline: October 6th 2026 at 23:59

1 Objective

For your second assignment, you will extend the abstract syntax tree (AST) that forms the structure of AeroScript. You will take on the challenge of implementing both a lexer and a parser to analyze and execute AeroScript code. Following this, you will develop a complete interpreter by expanding on the expression evaluator you made in Assignment 1.

2 Syntax

For easy reference, you can find the concrete syntax of AeroScript in [Figure 1](#).

$n \in \mathbb{Q}^p$	
$id \in \text{NAME}$	
<i>program</i>	$:= (\text{execution})^+$
<i>execution</i>	$:= \rightarrow^? id \{ (statement)^* \} (\rightarrow id)^?$
<i>statement</i>	$:= action$
<i>action</i>	$:= (acDock \mid acMove \mid acTurn \mid acAscend \mid acDescend) (\text{for } expression \text{ seconds} \mid \text{at speed } expression)^?$
<i>acDock</i>	$:= \text{return to base}$
<i>acMove</i>	$:= \text{move} (\text{to } point \mid \text{by } expression)$
<i>acTurn</i>	$:= \text{turn} (\text{right} \mid \text{left})^? \text{by } expression$
<i>acAscend</i>	$:= \text{ascend by } expression$
<i>acDescend</i>	$:= \text{descend by } expression \mid \text{descend to ground}$
<i>expression</i>	$:= \text{neg } expression \mid expression \text{ op } expression \mid \text{random } range^? \mid \text{point } point \mid n \mid (expression)$
<i>range</i>	$:= [expression , expression]$
<i>point</i>	$:= (expression , expression)$
<i>op</i>	$:= + \mid - \mid *$

Figure 1. The syntax of AeroScript. New constructs for this exercise are marked with a purple background. $\mathbb{Q}^p$ denotes positive rational numbers. The star ($\odot^*$) denotes a sequence containing 0 or more occurrences. and the question mark ($\odot^?$) denotes choice (zero or one occurrence), the plus ($\odot^+$) denotes a sequence containing one or more occurrences. NAME is the set of all possible identifiers.

Intuition: An AeroScript program consists of one or more *executions*, which have some similarity to methods in Java. Each execution is defined by an identifier and enclosed actions within braces. An execution may be prefixed by an optional arrow and may end with an optional arrow and a named execution. If an execution is prefixed by an arrow, this execution should be automatically started as part of the program's run block. If an execution ends with an arrow to another execution, program flow should transition to this execution and execute the corresponding instructions, before continuing with the rest of the main block. The following are some example executions:

```

-> MissionControl {} -> TakeOffLand

TakeOffLand {
    ascend by 20
    move to point (50, 50)
} -> ScanArea

ScanArea {
    move to point (100, 100) at speed 10
} -> EndMission

EndMission { return to base }

-> RandomTour {} -> TakeOff

Explore {
    move to point (random[0, 100], random[0, 100]) at speed 10
} -> ReturnToBase

ReturnToBase { return to base }

TakeOff {
    ascend by 20
    move to point (random[0, 100], random[0, 100])
} -> Explore

```

When executing an AeroScript program, your program should read the content of the file `program.aero` (can be found under `src/test/resources/`) and parse it into an AST. Your program should then execute the AeroScript program by traversing the AST and executing the statements.

Note: the executions that are defined (e.g. `TakeOffLand`, `TakeOff`, etc.) **must not** be executed when not invoked.

3 Semantics of AeroScript

The syntax and semantics of AeroScript expressions were covered (and hopefully implemented) in your previous assignment. As part of the handout for this exercise, you were given code that implements expression evaluation. This might differ somewhat from what you have implemented yourself previously. That is OK. You can choose either to continue working from your own code from the previous assignment or to use the provided code. Whatever you choose in that regard, the code to parse and evaluate expressions should now be easy to understand for you.

We will in this exercise focus on how to evaluate executions. Every execution is given by a sequence of actions. When an execution is entered, its actions are performed in order. Thus, the program transitions to the next statement in the sequence. At all times the program needs to keep track at the current

- position,
- altitude,
- distance traveled, and
- battery level.

If the level of the battery drops to zero, the program should trigger an exception.

Each action needs to be printed out to the command line as a log, and will affect the variables of the program (be it the position as a coordinate in X and Y or the altitude if ascending or descending) and will deplete the battery level.

3.1 Detailed semantics

Executions and Transitions In AeroScript, **executions** are like different states or scenarios in which a drone can operate—such as taking off, flying, or landing and are defined in the grammar by the `execution`. In Assignment 2, an execution is similar to a method and a new method can only be called after executing the method body of the currently active method.

Action	Description	Battery cost
<i>acDock</i>	Terminal action that takes no arguments. The program should print the distance travelled and the battery level before it terminates.	altitude + (time * 0.1) + (speed * 1). A successful return to dock restores the battery level to its initial value.
<i>acMove</i>	Move to a point or by n meters. If point, we need to compute the cartesian difference (in 2 dimension of point, do not consider the altitude), otherwise we move by n meters into either the x or y directions (choose one).	(distance * 0.7 if point, distance * 0.5 otherwise) + (time * 0.1) + (speed * 1)
<i>acTurn</i>	Turn by n degrees.	angle * 0.3 + (time * 0.1) + (speed * 1)
<i>acAscend</i>	Go up by n meters.	distance * 0.6 + (time * 0.1) + (speed * 1)
<i>acDescend</i>	Go to the ground (altitude goes to 0).	altitude * 0.2 + (time * 0.1) + (speed * 1)

Table 1. Table of costs for battery depletion. Note that for time and speed they might be not present at all – set the value to 0 if the parameter is not present.

Each execution contains a set of operations that will be performed when the drone is in that state. For instance, a `TakeOff` execution might include instructions to power up the rotors and ascend to a specific altitude. Note that in order to have a program that makes the drone *do* something, it must have at least one execution with a leading transition arrow—the *start execution*.

Actions and Commands Actions are the building blocks of what your drone can actually do. These include moving to a specific (2D) coordinate, ascending or descending, and turning. Each action in `AeroScript` directly corresponds to a command sent to the drone’s control system.

- Commands like `move to point (x, y)` tell the drone to fly to a specific location in the two-dimensional plane at its current altitude. The semantics of this command involve calculating the flight path and adjusting the rotors to navigate to these coordinates.

Expressions and Calculations Expressions and calculations were covered in the last assignment (Oblig 1); no changes are warranted here!

Battery Depletion Each action (except printing to the command line) will deplete the battery level as given in [Table 1](#).

Distance travelled Each action should update the distance travelled as described in [Table 1](#).

3.2 Putting It All Together

After parsing the source into an AST, your interpreter should traverse it and perform the instructions. More specifically, it should identify the start execution, perform actions, and evaluate expressions. The interpreter continually checks which execution the drone is in, and which actions to perform based on the current execution.

Note: Moreover, the executions that are defined (e.g., `TakeOffLand`, `TakeOff`, etc.) **must not** be executed when not invoked (e.g. `TakeOffLand` gets executed after the body of `MissionControl` has been completed.) Executions that start with ‘`->`’ should be invoked when reached. Executions that do not start with ‘`->`’ should only be invoked when called from another execution.

4 Your task

The assignment includes a template for the implementation, similar to Oblig 1. You should do the following:

- Extend the AST to cover the entire syntax of `AeroScript`.

- Extend the lexer and parser to cover the language given in [Figure 1](#) using ANTLR.
- Extend the interpreter to cover the language given in [Figure 1](#).
- We have provided some `JUnit` tests. **Discuss the coverage of the tests**, and provide your own. In particular, make sure that exceptions are thrown correctly.
- Write a report (2-3 pages) where you discuss the challenges you faced and choices you made during implementation. Be sure to read the report guide on the [course page](#) beforehand.

5 Implementation Hints

- When adding elements to the heap, you should use these variables
 - `altitude` (initially set to 0)
 - `battery level` (initial value can be provided by the user on the command line)
 - `distance travelled` (initially set to 0)
 - `initial execution` (initially set to null)
- When implementing the interpretation of executions:
 - Think about what should happen when the drone runs out of battery...
 - make sure your code can run recursive executions (until they run out of battery!)

6 How and what to Submit

See the Norwegian version of this document for more updated info about this.

You should hand in a `zip` file containing your report (as a PDF, about 2-3 pages) along with the project containing your implementation. Make sure to clean (run `./gradlew clean`) any unnecessary garbage files before zipping. If your zip-file is very large, you likely did not clean.

7 Where to Submit

In IN2031, we use Devilry, not Canvas, for assignments. Go to

<https://devilry.ifi.uio.no>

and log in with your user UiO-name and password. When you are logged in, it should be fairly obvious how to upload the report and code.