

Oblig 3: The AeroScript Language with actions and reactions

Deadline: October 27th 2026 23:59

1 The Objective

For your third assignment, you will extend AeroScript with an event-mechanism that makes AeroScript more expressive. Regular program flow can be interrupted by *events* that can trigger global executions. This assignment entails adapting both the grammar, the abstract syntax tree and program interpretation accordingly.

2 Syntax

For easy reference, you can find the full concrete syntax of AeroScript in [Figure 1](#). The main difference to the concrete syntax in Assignment 2 is that the production for *statement* has changed, and can now be either an *action* (as before), or a *reaction*. The new part of the concrete syntax is highlighted.

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

Figure 1. The syntax of AeroScript. The star denotes a sequence containing 0 or more occurrences, the question mark denotes choice (zero or one occurrence), and the plus 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 optionally starting with a transition arrow, followed by an identifier and enclosed actions, reactions, or nested executions 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. If an execution

ends with an arrow to another execution, program interpretation should transition to this execution and perform the corresponding instructions. The new event-mechanism allows a program to transition between different executions based on specified events. When program flow is evented, any remaining actions of the current execution should be discarded. The following are example executions:

```
-> MissionControl {
  on message [launch] -> TakeOffLand
}

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

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

EndMission { return to base }

EmergencyLanding { descend to ground }
```

And another example; note how the Explore execution does not call another execution, and the interpreted should thus wait for events to happen:

```
-> RandomTour {
  on message [start_tour] -> TakeOff
}

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

Explore {
  move to point (random[0, 100], random[0, 100]) at speed 10
  on low battery -> EmergencyLanding
  on message [end_tour] -> ReturnToBase
}

ReturnToBase { return to base }

EmergencyLanding { descend to ground }
```

We can also make use of random movements and reactions to events in our AeroScript programs, to make for perhaps a more interesting example:

```
-> MissionControl {
  on message [launch] -> TakeOffLand
}

ScanAreaUntilOutOfBattery {
  move to point (random [1, 100], random [1, 100]) at speed random [10, 50]
  on low battery -> EmergencyLanding
  on obstacle -> EmergencyLanding
} -> ScanAreaUntilOutOfBattery

TakeOffLand {
  ascend by 20
```

```

    move to point (50, 50)
} -> ScanAreaUntilOutOfBattery

EmergencyLanding { descend to ground }

```

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

Note: reactions (obstacle, message and battery) **must not** be performed unless they are triggered by an event. Moreover, the named executions (e.g., `TakeOffLand`, `TakeOff`, etc.) **must not** be performed unless they are invoked (e.g., `TakeOffLand` starts when we receive the message `launch`).

After all the statements are interpreted, your program needs to enter the REPL (Read-Eval-Print Loop) and wait for triggers (messages) from the user. When a trigger is received, your program should perform the corresponding reaction. To end the simulation, you must type `exit` in the REPL. Until then, the program must interpret all the statements that have to be executed and then wait.

3 Semantics of AeroScript

The syntax and semantics of AeroScript expressions, actions and executions were covered (and hopefully implemented) in your previous assignments; we now focus on how to evaluate reactions.

Every execution is a sequence of actions and reactions. When an execution is entered, the actions are performed in order. If a reaction is triggered, remaining actions of the current execution are discarded and the program transitions to the execution specified in the reaction. If no reaction is triggered, 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 goes below 20, the program should trigger the `low battery` event if present, or exit with a failure otherwise.

Each action needs to be printed on the command line as a log, and will affect the variables of the program (be it the position as 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*. Each execution contains a set of operations that can 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 outer execution with a leading transition arrow—the *start execution*.

- **Transitions** between executions are triggered by events or conditions. For example, once a drone reaches the desired altitude in `TakeOff` execution, it might automatically switch to a `Hover` or `Navigate` execution.

Events and Reactions Events are specific occurrences like detecting an obstacle (supplied as a message from the user in the REPL), receiving a command to return to base, or having low battery. In AeroScript, you define what the drone should do when these events happen through **reactions**.

- If an event like `on low battery` is triggered, the drone might switch to a `ReturnToBase` execution, which contains a predefined set of actions to safely return and land the drone.

Actions, Expressions and Calculations Actions, expressions and calculations were covered in the previous assignments and work as before; in particular, the computation of current position, battery depletion and distance travelled works as before. No changes are warranted here!

3.2 Putting It All Together

After parsing the source into an AST, your interpreter should traverse it and execute the instructions. More specifically, it should identify the start execution, perform actions, and wait for events.

The program needs to enter the REPL and wait for triggers (messages) from the user. When a trigger is received, the program should perform the corresponding reaction. The program should continue to wait for triggers until the user types `exit` in the REPL.

Note: The reactions (message and battery) **do not** have to be performed if they are not triggered. Moreover, the functions that are defined (e.g. `TakeOffLand`, `TakeOff`, etc.) **must not** be executed when not invoked (e.g. `TakeOffLand` gets executed when we receive the message `launch`).

4 Your task

For this assignment you must extend your work from the last assignment or use the given expression semantics. In particular you are expected to do the following:

- Extend the AST to cover the entire syntax of `AeroScript`.
- Extend the interpreter accordingly, and ensure it handles all new events and reactions.
- Extend the lexer and parser to cover the whole language using ANTLR.
- Build a REPL so the user can interact with your interpreter.
- Make sure that all the programs under `src/test/resources/` run correctly.
- Write some unit tests that cover the functionality of your interpreter. Discuss your test coverage.
- Write a report (3-4 pages) where you discuss the challenges you faced and choices you made during implementation. The report should cover the entire work (that is also the most important part of assignments 1 and 2), but should have a focus on this work. Be sure to read the report guide on the [course page](#) beforehand.

5 Implementation Hints

- If you want to use listeners for event handling, you can see the REPL class for an example of how to implement them and use the provided code. However, you can also use a different approach if you prefer. You may choose to implement real parallelism using threads, but be aware this is probably more challenging (and perhaps fun?) than using a cooperative approach that checks for events between interpretation of statements.
- After you `exit` the REPL, the program should print the initial position, initial battery capacity, final position, final battery level and distance travelled
- When implementing the events, make sure to add the correct logic for each event:
 - The `obstacle` message should trigger the function corresponding to the event immediately
 - `low battery` should trigger the function corresponding to the event if the battery level is below 20
 - `message` should trigger the function corresponding to the event if the message is received from the REPL
- When writing the grammar remember to always have the correct mapping for the keywords (e.g. `ON` for `'on'`, `BATTERY` for `'low battery'`, etc.) so that you can then check for their existence in the `Interpreter`

6 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 3-4 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; go to

<https://devilry.ifl.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.