

Oblig 3: Språket AeroScript med handlinger og reaksjoner

Frist: 27. oktober 2026 kl. 23:59

1 Mål

I den tredje obligatoriske oppgaven skal du utvide AeroScript med en hendelsesmekanisme som gjør AeroScript mer uttrykksfullt. Den ordinære programflyten kan avbrytes av *hendelser* som kan utløse globale executions. Oppgaven innebærer å tilpasse både grammatikken, det abstrakte syntakstreet og tolkningen av programmet.

2 Syntaks

Den fullstendige konkrete syntaksen til AeroScript finnes i [Figure 1](#). Hovedforskjellen fra den konkrete syntaksen i Oblig 2 er at produksjonen for *statement* er endret og nå kan være enten en *action* (som før) eller en *reaction*. Den nye delen av den konkrete syntaksen er fremhevet.

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

Figure 1. Syntaksen til AeroScript. Stjernen angir en sekvens med 0 eller flere forekomster, spørsmålsteget angir et valg (null eller én forekomst), og plussteget angir en sekvens med én eller flere forekomster. NAME er mengden av alle mulige identifikatorer.

Intuisjon: Et AeroScript-program består av én eller flere *executions*, som har enkelte likhetstrekk med metoder i Java. Hver execution defineres av en valgfri innledende overgangspil, etterfulgt av en identifikator og handlinger, reaksjoner eller nøstede executions omsluttet av klammeparenteser. En execution kan ha en valgfri pil foran seg og kan avsluttes med en valgfri pil og navnet på en execution. Hvis en execution har en pil foran seg, skal denne execution startes automatisk. Hvis en execution

avsluttes med en pil til en annen execution, skal programtolkningen gå videre til denne execution og utføre de tilhørende instruksjonene. Den nye hendelsesmekanismen lar programmet gå mellom ulike executions basert på bestemte hendelser. Når en hendelse avbryter programflyten, skal gjenstående handlinger i gjeldende execution forkastes. Følgende er eksempler på 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 }
```

Her er et annet eksempel. Merk at Explore-execution ikke kaller noen annen execution, og at interpreteren derfor skal vente på at hendelser inntreffer:

```
-> 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 }
```

Vi kan også bruke tilfeldige bevegelser og reaksjoner på hendelser i AeroScript-programmene våre for å lage et kanskje mer interessant eksempel:

```
-> 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 }

```

Når et AeroScript-program tolkes, skal programmet ditt lese innholdet i en `.aero`-fil (som finnes under `src/test/resources/`) og parse det til et AST. Deretter skal programmet tolke AeroScript-programmet ved å traversere AST-et og utføre setningene.

Merk: Reaksjoner (obstacle, message og battery) **må ikke** utføres med mindre de utløses av en hendelse. Navngitte executions (f.eks. `TakeOffLand`, `TakeOff` osv.) **må heller ikke** utføres med mindre de kalles (f.eks. starter `TakeOffLand` når meldingen `launch` mottas).

Etter at alle setningene er tolket, skal programmet gå inn i REPL (Read-Eval-Print Loop) og vente på utløsende meldinger fra brukeren. Når en slik melding mottas, skal programmet utføre den tilhørende reaksjonen. For å avslutte simuleringen må du skrive `exit` i REPL-en. Frem til da skal programmet tolke alle setninger som skal utføres, og deretter vente.

3 Semantikken til AeroScript

Syntaksen og semantikken for uttrykk, handler og executions i AeroScript ble gjennomgått (og forhåpentligvis implementert) i tidligere oppgaver. Nå fokuserer vi på hvordan reaksjoner skal evalueres.

Hver execution er en sekvens av handler og reaksjoner. Når en execution startes, utføres handlingene i rekkefølge. Hvis en reaksjon utløses, forkastes de gjenværende handlingene i gjeldende execution, og programmet går videre til execution som er angitt i reaksjonen. Hvis ingen reaksjon utløses, går programmet videre til neste setning i sekvensen. Programmet må til enhver tid holde oversikt over gjeldende

- posisjon,
- høyde,
- tilbakelagt avstand og
- batterinivå.

Hvis batterinivået synker under 20, skal programmet utløse hendelsen `low battery` dersom den finnes, eller avslutte med en feil ellers.

Hver handling skal loggføres på kommandolinjen og påvirker programmets variabler (enten posisjonen som X- og Y-koordinater eller høyden ved opp- eller nedstigning), samtidig som batterinivået reduseres.

3.1 Detaljert semantikk

Executions og overganger I AeroScript tilsvarer **executions** ulike tilstander eller scenarioer en drone kan operere i—som avgang, flyging eller landing—og defineres i grammatikken av `execution`. Hver execution inneholder et sett med operasjoner som kan utføres når dronen er i denne tilstanden. En `TakeOff`-execution kan for eksempel inneholde instruksjoner om å starte rotorene og stige til en bestemt høyde. Merk at et program som skal få dronen til å *gjøre* noe, må ha minst én ytre execution med en innledende overgangspil—*start-execution*.

- **Overganger** mellom executions utløses av hendelser eller betingelser. Når en drone for eksempel når ønsket høyde i en `TakeOff`-execution, kan den automatisk gå videre til en `Hover`- eller `Navigate`-execution.

Hendelser og reaksjoner Hendelser er bestemte situasjoner, som å oppdage en hindring (formidlet som en melding fra brukeren i REPL-en), motta en kommando om å returnere til basen eller ha lavt batterinivå. I AeroScript definerer du hva dronen skal gjøre når slike hendelser inntreffer, ved hjelp av **reaksjoner**.

- Hvis en hendelse som `on low battery` utløses, kan dronen gå videre til en `ReturnToBase`-execution som inneholder et forhåndsdefinert sett med handler for å returnere og lande trygt.

Handler, uttrykk og beregninger Handler, uttrykk og beregninger ble gjennomgått i de tidligere oppgavene og fungerer som før. Spesielt beregnes gjeldende posisjon, batteriforbruk og tilbakelagt avstand på samme måte som tidligere. Ingen endringer er nødvendige her!

3.2 Vi setter det hele sammen

Etter at kildekoden er parset til et AST, skal interpreteren traversere det og utføre instruksjonene. Mer spesifikt skal den identifisere start-execution, utføre handlinger og vente på hendelser.

Programmet skal gå inn i REPL-en og vente på utløsende meldinger fra brukeren. Når en melding mottas, skal programmet utføre den tilhørende reaksjonen. Programmet skal fortsette å vente på meldinger til brukeren skriver `exit` i REPL-en.

Merk: Reaksjonene (message og battery) skal **ikke** utføres hvis de ikke utløses. De definerte funksjonene (f.eks. `TakeOffLand`, `TakeOff` osv.) **må ikke** utføres med mindre de kalles (f.eks. kjøres `TakeOffLand` når meldingen `launch` mottas).

4 Din oppgave

I denne oppgaven skal du bygge videre på arbeidet fra forrige oppgave eller bruke den utdelte semantikken for uttrykk. Du skal spesielt gjøre følgende:

- Utvid AST-et slik at det dekker hele syntaksen til `AeroScript`.
- Utvid interpreteren tilsvarende, og sørg for at den håndterer alle nye hendelser og reaksjoner.
- Utvid lekseren og parseren slik at de dekker hele språket, ved hjelp av ANTLR.
- Lag en REPL slik at brukeren kan samhandle med interpreteren.
- Sørg for at alle programmene under `src/test/resources/` kjører korrekt.
- Skriv enhetstester som dekker funksjonaliteten til interpreteren. Diskuter testdekningen.
- Skriv en rapport (3–4 sider) der du diskuterer utfordringene du møtte og valgene du tok under implementasjonen. Rapporten skal dekke hele arbeidet (også de viktigste delene av Oblig 1 og 2), men ha hovedvekt på denne oppgaven. Sørg for å lese rapportveiledningen på [emnesiden](#) på forhånd.

5 Implementasjonstips

- Hvis du vil bruke lyttere til hendelseshåndtering, kan du se REPL-klassen for et eksempel på hvordan de implementeres, og bruke den utdelte koden. Du kan også velge en annen tilnærming. Du kan implementere reell parallellitet ved hjelp av tråder, men vær klar over at dette sannsynligvis er mer utfordrende (og kanskje morsommere?) enn en kooperativ tilnærming som sjekker etter hendelser mellom tolkningen av setninger.
- Etter at du avslutter REPL-en med `exit`, skal programmet skrive ut startposisjon, opprinnelig batterikapasitet, sluttposisjon, endelig batterinivå og tilbakelagt avstand
- Når du implementerer hendelsene, må du sørge for riktig logikk for hver hendelse:
 - Meldingen `obstacle` skal umiddelbart utløse funksjonen som hører til hendelsen
 - `low battery` skal utløse funksjonen som hører til hendelsen dersom batterinivået er under 20
 - `message` skal utløse funksjonen som hører til hendelsen dersom meldingen mottas fra REPL-en
- Når du skriver grammatikken, må du sørge for riktig kobling mellom nøkkelord og token (f.eks. `ON` for `'on'`, `BATTERY` for `'low battery'` osv.), slik at du kan kontrollere om de finnes i `Interpreter`

6 Hvordan og hva skal leveres

Ved samretting før fristen bestemmer retter i samråd med gruppa hva som skal leveres i Devilry for å få godkjent. Se forøvrig [prosjektsida](#).

Du skal levere en zip-fil som inneholder prosjektet. Sørg for å rydde bort unødvendige filer (kjør `./gradlew clean`) før du lager zip-filen. Hvis zip-filen er svært stor, har du sannsynligvis ikke ryddet.

Ved tradisjonell retting med skriftlig tilbakemelding skal rapporten inkluderes i zip-filen.