Imperial College London

MEng Individual Project Report

Imperial College London

Department of Computing

xDrone Programming Language

Author:
Kai Zhu

Supervisor:
Dr. Robert Chatley

Second Marker:
Dr. Antonio Filieri

June 14, 2021

Abstract

We present *xDrone*, a new domain-specific language that is sophisticated in drone controlling. *xDrone* targets drone operators who are not professional programmers but still want to create sophisticated control patterns. It is straightforward, versatile, elegant, and powerful.

In recent years, there has been an increasing trend towards the professional use of programmable drones. However, existing drone programming languages are primarily targeting students or hobbyists for educational purposes, and have some limitations to a greater or lesser extent.

Programs for concurrent flying of multiple drones can be written easily in *xDrone*, with a simulator and highly adjustable safety checks and collision checks to guarantee flight safety. The collision check can also take the deviation of drones from their intended flight path that is inevitable due to unreliability of hardware into account, and provide a report about the probability of potential collisions.

Acknowledgements

I would like to express my gratitude to Robert Chatley for agreeing to take over this project after the leaving of its original supervisor, and for supervising this project and guiding the writing of this report, for all the time he has devoted to our weekly meetings.

I would also like to thank Madasar Shah for introducing *xDrone*, helping me find the main direction to investigate, as the original supervisor of this project.

I would like to thank Haoxiang Zhou for proofreading my report, as well as all the other friends I have made at Imperial for the greatest memory we made together over the past four years.

To those who worked hard during the pandemic, thank you for devoting yourselves during this unusual time and maintaining the order of the society.

Finally, I would like to thank my parents for their support throughout my entire life.

Contents

Chapter 1

Introduction

1.1 Motivation

Over the last decade, drones have become easily accessible and more affordable to hobbyists and businesses. Many drone producers, going with this stream, have launched programmable drones. Most of these programmable drones are designed for educational purposes to aid students and hobbyists in learning programming, usually in languages like Python or Scratch, as there does not exist a specific programming language for drone controlling in the market. However, there is actually an increasing application of professional use of programmable drones, like in parcel delivery [1], drone displays [2], etc.
In the past few years, students [3, 4, 5, 6] at Oxford University and Imperial College have contributed to *xDrone*, a programming language that is designed to control drones. Nevertheless, *xDrone* still has some limitations, and can benefit from further development. The previous version of *xDrone* was targeting primary school student, and was used for educational purposes. In spite of the success of *xDrone* in children education, its syntax is not sophisticated enough for professional drone operators. In this project, we will focus on improving the domain-specific language, *xDrone,* targeting professional drone operators.

1.2 Objectives

The main objective of the project is to develop *xDrone* into a more powerful and sophisticated language, targeting drone operators who are not professional programmers but still want to create sophisticated control patterns. The language itself should be simple, intuitive for non-professional programmers, but at the same time be powerful.
Versatility is another objective to consider, as we do not want to limit the language to a certain type of drone. Any programmable drone of any brand can be controlled by drone operators who know *xDrone*, with no extra learning cost. It should be possible to target the same *xDrone* program to different drone hardware.
It should also be able to control multiple drones in parallel to support swarm flying easily.
Corresponding development tools should also be provided, to improve the programming experience and guarantee flight safety. The aim is to provide safety checks before flying the drones, and prevent accidents during flights, especially when multiple drones are flying simultaneously. The safety checks should ideally take the deviation of drones into account.

Chapter 2

Background

2.1 History of *xDrone*

xDrone is a programming language that is used to control drones. It was developed by students at Oxford University and Imperial College London. It has been designed in a way that is very accessible to primary school student with no prior programming experience, and can be used for educational purposes.

2.1.1 The First Version - Basic Movements

The first version of the *xDrone* language was implemented by a group of students [3] at Oxford University using Xtext in 2017. It has a very limited set of commands.

⟨*program*⟩ ::= `'xdrone'` ⟨*program_name*⟩ `'begin'` (⟨*command*⟩ [`';'`])* `'end'`

⟨*command*⟩ ::= `'TAKEOFF'`
| `'LAND'`
| `'WAIT'` `'('` ⟨*milliseconds*⟩ `')'`
| `'UP'` `'('` ⟨*milliseconds*⟩ `')'`
| `'DOWN'` `'('` ⟨*milliseconds*⟩ `')'`
| `'LEFT'` `'('` ⟨*milliseconds*⟩ `')'`
| `'RIGHT'` `'('` ⟨*milliseconds*⟩ `')'`
| `'FORWARD'` `'('` ⟨*milliseconds*⟩ `')'`
| `'BACKWARD'` `'('` ⟨*milliseconds*⟩ `')'`
| `'ROTATELEFT'` `'('` ⟨*milliseconds*⟩ `')'`
| `'ROTATERIGHT'` `'('` ⟨*milliseconds*⟩ `')'`

⟨*program_name*⟩ ::= ⟨*identifier*⟩

⟨*milliseconds*⟩ ::= ⟨*integer*⟩

A simple web editor is also available. Users can use this editor to deploy *xDrone* program to a drone.

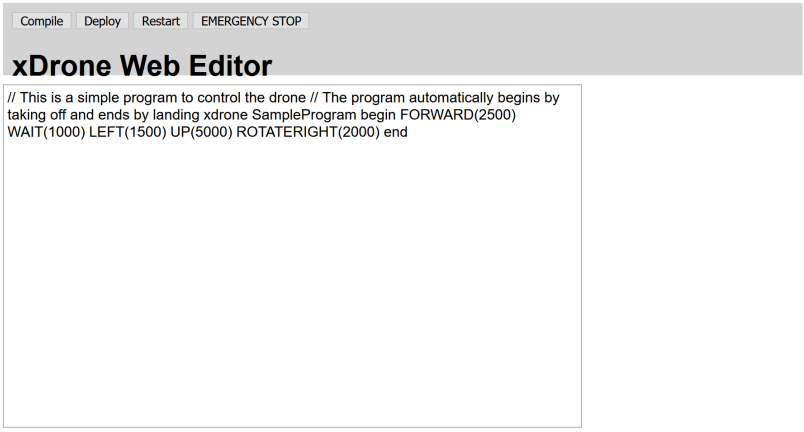


Figure 2.1: The simple web editor for *xDrone*.

2.1.2 The Second Version - Camera and Simple Functions

In 2018, a student [4] at Imperial College London provided support for camera use and user-defined functions without parameters, which has brought the *xDrone* language to its second version.

⟨*program*⟩ ::= ‘xDrone’ ⟨*program_name*⟩
[‘USEDOWNWARDCAMERA’ [‘;’]]
[‘RECORDFLIGHT’ ‘(’ ⟨*video_name*⟩ ‘)’ [‘;’]]
‘main_sequence()’
‘{’
[‘SNAPSHOT’ ‘(’ ⟨*image_name*⟩ ‘)’ [‘;’]]
[⟨*main_commands*⟩]
‘}’
[⟨*on_event*⟩]
⟨*user_function*⟩*

⟨*main_commands*⟩ ::= ‘TAKEOFF’ [‘;’]
(⟨*command*⟩ [‘;’])*
‘LAND’ [‘;’]

⟨*on_event*⟩ ::= ‘on_event’ ‘(’ ⟨*event*⟩ ‘)’ ‘{’
(⟨*command*⟩ [‘;’])*
‘}’

⟨*event*⟩ ::= ‘FACEDETECT’
| ‘FEATUREMATCH’ ‘(’ ⟨*image_name*⟩ ‘)’

⟨*user_function*⟩ ::= ⟨*function_name*⟩ ‘()’ ‘{’ (⟨*command*⟩ [‘;’])* ‘}’

⟨*command*⟩ ::= ⟨*function_name*⟩ ‘()’
| ‘WAIT’ ‘(’ ⟨*milliseconds*⟩ ‘)’
| ‘UP’ ‘(’ ⟨*milliseconds*⟩ ‘)’
| ‘DOWN’ ‘(’ ⟨*milliseconds*⟩ ‘)’
| ‘LEFT’ ‘(’ ⟨*milliseconds*⟩ ‘)’
| ‘RIGHT’ ‘(’ ⟨*milliseconds*⟩ ‘)’
| ‘FORWARD’ ‘(’ ⟨*milliseconds*⟩ ‘)’
| ‘BACKWARD’ ‘(’ ⟨*milliseconds*⟩ ‘)’
| ‘ROTATELEFT’ ‘(’ ⟨*milliseconds*⟩ ‘)’
| ‘ROTATERIGHT’ ‘(’ ⟨*milliseconds*⟩ ‘)’

⟨*program_name*⟩ ::= ⟨*identifier*⟩
⟨*video_name*⟩ ::= ⟨*identifier*⟩
⟨*image_name*⟩ ::= ⟨*identifier*⟩
⟨*function_name*⟩ ::= ⟨*identifier*⟩
⟨*milliseconds*⟩ ::= ⟨*integer*⟩

From the syntax we notice that *xDrone* has been extended with camera support (e.g. `RECORDFLIGHT` and `SNAPSHOT`), as well as on-event and user-defined functions without parameters.
In addition, the web editor has been refurbished, and a gallery has been added to it.

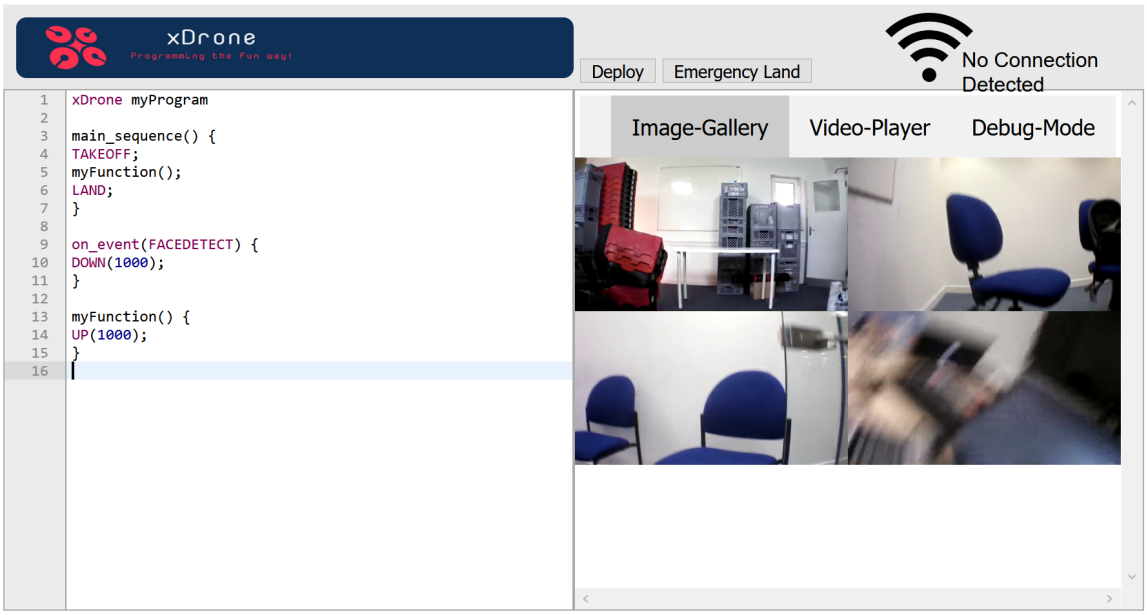


Figure 2.2: The new web editor and the gallery.

2.1.3 The Third Version - Environment, Simulation and Routing

In early 2019, *xDrone* has been further extended with simulation, route planning and environment modelling features [5]. Compared to the first version of *xDrone*, an environment with different objects can be coded within *xDrone*. This environment can then be simulated on the web editor. A new command `GOTO` is added, which enables the drone to automatically plan the route and land on a specified object in the environment.
Also in this version, the units of commands have been changed. Moving and waiting are now in the unit of distance instead of time duration, and the unit of rotating has been changed to degrees. Besides, some small changes are made to the syntax to make the program's main body simpler, e.g. optional semi-colon at the end of each command is removed.

⟨*program*⟩ ::= `'fly' '(' ')' '{'`
`'TAKEOFF'`
⟨*command*⟩*
`'LAND'`
`'}'`
[⟨*environment*⟩]

⟨*command*⟩ ::= `'GOTO' '('` ⟨*object_name_str*⟩ `')'`
| `'WAIT' '('` ⟨*seconds*⟩ `')'`
| `'UP' '('` ⟨*distance*⟩ `')'`
| `'DOWN' '('` ⟨*distance*⟩ `')'`
| `'LEFT' '('` ⟨*distance*⟩ `')'`
| `'RIGHT' '('` ⟨*distance*⟩ `')'`
| `'FORWARD' '('` ⟨*distance*⟩ `')'`
| `'BACKWARD' '('` ⟨*distance*⟩ `')'`
| `'ROTATELEFT' '('` ⟨*angle*⟩ `')'`
| `'ROTATERIGHT' '('` ⟨*angle*⟩ `')'`

⟨*environment*⟩ ::= `'environment' '(' ')' '{'`
(⟨*drone*⟩ | ⟨*object*⟩ | ⟨*walls*⟩)*
`'}'`

⟨*drone*⟩ ::= `'DRONE' '=' '{'`
(`'position' '='` ⟨*vector*⟩ | `'rotation' '='` ⟨*double*⟩)*
`'}'`

⟨*object*⟩ ::= ⟨*object_name*⟩ `'=' '{'`
(`'origin' '='` ⟨*vector*⟩ | `'size' '='` ⟨*vector*⟩ | `'color' '='` ⟨*string*⟩)*
`'}'`

⟨*vector*⟩ ::= `'('` ⟨*double*⟩ `','` ⟨*double*⟩ `','` ⟨*double*⟩ `')'`

⟨*walls*⟩ ::= `'WALLS' '=' '{'`
(`'front' '='` ⟨*double*⟩ | `'right' '='` ⟨*double*⟩ | `'left' '='` ⟨*double*⟩ |
`'back' '='` ⟨*double*⟩ | `'up' '='` ⟨*double*⟩)*
`'}'`

⟨*object_name*⟩ ::= ⟨*identifier*⟩

⟨*object_name_str*⟩ ::= ⟨*string*⟩

⟨*seconds*⟩ ::= ⟨*double*⟩

⟨*distance*⟩ ::= ⟨*double*⟩

⟨*angle*⟩ ::= ⟨*double*⟩

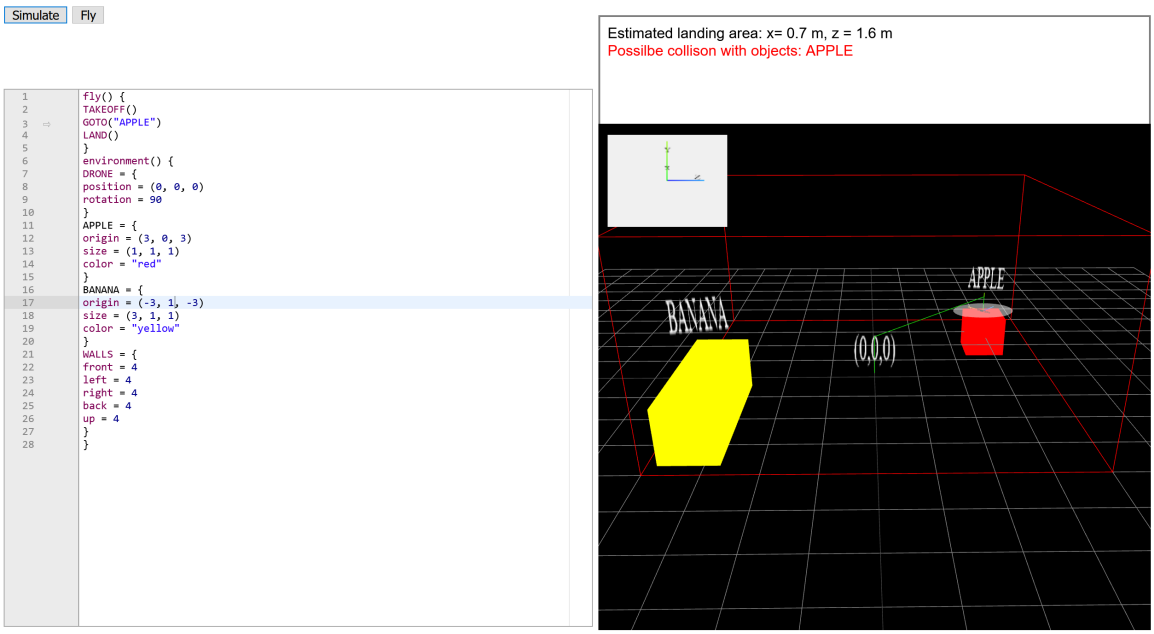


Figure 2.3: The web editor with simulation.

2.1.4 The Fourth Version - the Drone Playground

In late 2019, a team [6] at Imperial College London has created a web interface for the *xDrone* language, which has a simple Scratch-like interface to help students who have no programming experience learn how to program. Also, there is an interface for teachers, where they can easily manage the programs written by students. Notably, the simulation feature has also been refurbished. Students and teachers can view the simulation of the flight to review the program before flying a real drone. There is a restriction on the flying area; each drone can only fly in a 4 by 4 metre space.
However, some features added in the second and third version (e.g. camera using, user-defined functions, moving by distance, route planning) were not inherited.

⟨*program*⟩ ::= `'fly' '(' ')' '{'`
`'TAKEOFF' '(' ')'`
⟨*command*⟩*
`'LAND' '(' ')'`
`'}'`

⟨*command*⟩ ::= `'ACTION' '(' ')'`
| `'WAIT' '(' ⟨seconds⟩ ')'`
| `'UP' '(' ⟨seconds⟩ ')'`
| `'DOWN' '(' ⟨seconds⟩ ')'`
| `'LEFT' '(' ⟨seconds⟩ ')'`
| `'RIGHT' '(' ⟨seconds⟩ ')'`
| `'FORWARD' '(' ⟨seconds⟩ ')'`
| `'BACKWARD' '(' ⟨seconds⟩ ')'`
| `'ROTATELEFT' '(' ⟨degrees⟩ ')'`
| `'ROTATERIGHT' '(' ⟨degrees⟩ ')'`

⟨*seconds*⟩ ::= ⟨*double*⟩

⟨*angle*⟩ ::= ⟨*double*⟩

Figure 2.4: The Drone Playground, with a Scratch-like interface and the simulation feature.

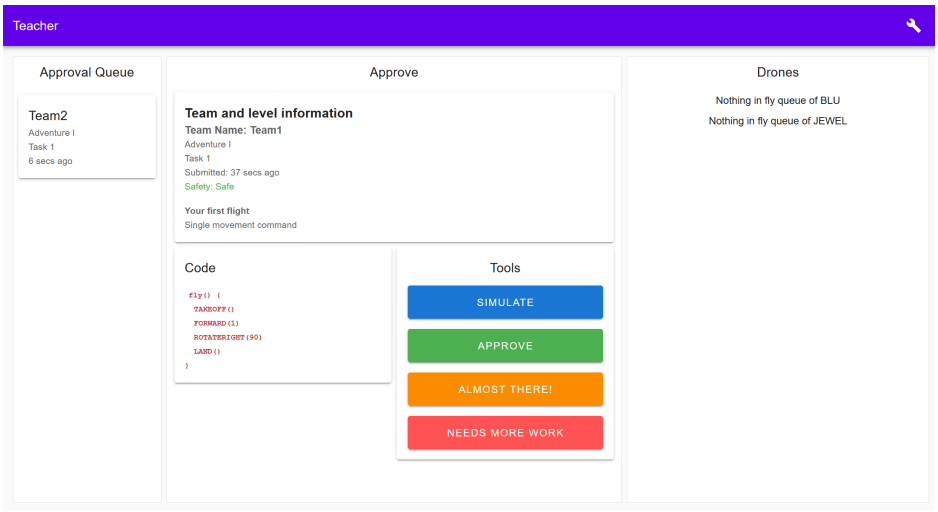


Figure 2.5: The teacher side of the Drone Playground.

2.1.5 Summary

As we can notice, although versions of *xDrone* support basic control of drones and can be used to write simple programs, none of them is sophisticated enough for professional drone operators. There is a restriction on the flying area in the fourth version, which acts like a safety check, but is not configurable. In addition, all previous versions are for controlling one drone only; improvements are needed if users want to control multiple drones simultaneously.

2.2 Other Drone Languages

There are very few programming languages sophisticated in drone programming. Usually, programmable drones provide three approaches for users to program on them - firmware programming, through SDK, and through first-party application.

2.2.1 Firmware Programming

This is a relatively rare approach, as the programmer needs to coordinate the hardware aspects of the drone, like the motors, sensors, battery, etc. This is usually done in low-level programming languages like Assembly, C or C++, on Arduino and Raspberry Pi. [7]
This approach obviously has high barriers to entry, and is not suitable for non-professional drone developers.

2.2.2 Programming through SDK

Many programmable drones have their own SDK or API. These SDK's are usually some texts that are sent to the drone's IP address.
There are open-source packages that encapsulate these SDK calls in various programming languages on GitHub, and most of them are third-party. `TelloEduSwarmSearch` [8] is one of the third-party packages that control Tello EDU drones, and is the package I used in the implementation of the compiler. The following is an example of programming in Python with this package.

```
from fly_tello import FlyTello

my_tellos = list()
my_tellos.append('0TQDFCAABBCCDD')
my_tellos.append('0TQDFCAABBCCEE')

with FlyTello(my_tellos) as fly:
    fly.takeoff()               # All Tellos take-off
    fly.forward(50)             # All Tellos fly forward by 50cm
    with fly.sync_these():      # Keep the following commands in-sync
        fly.left(30, tello=1)   # Tell just Tello1 to fly left
        fly.right(30, tello=2)  # At the same time, Tello2 will fly right
    fly.land()                  # All Tellos land
```

This approach is more accessible than firmware programming, but still requires the developer to know a mainstream programming language, and to search for a suitable library by

themselves. There is no simulation or safety check, which is very helpful in drone programming.

2.2.3 Programming through First-Party Application

Some of the programmable drone companies provide their first-party drone controlling mobile applications.

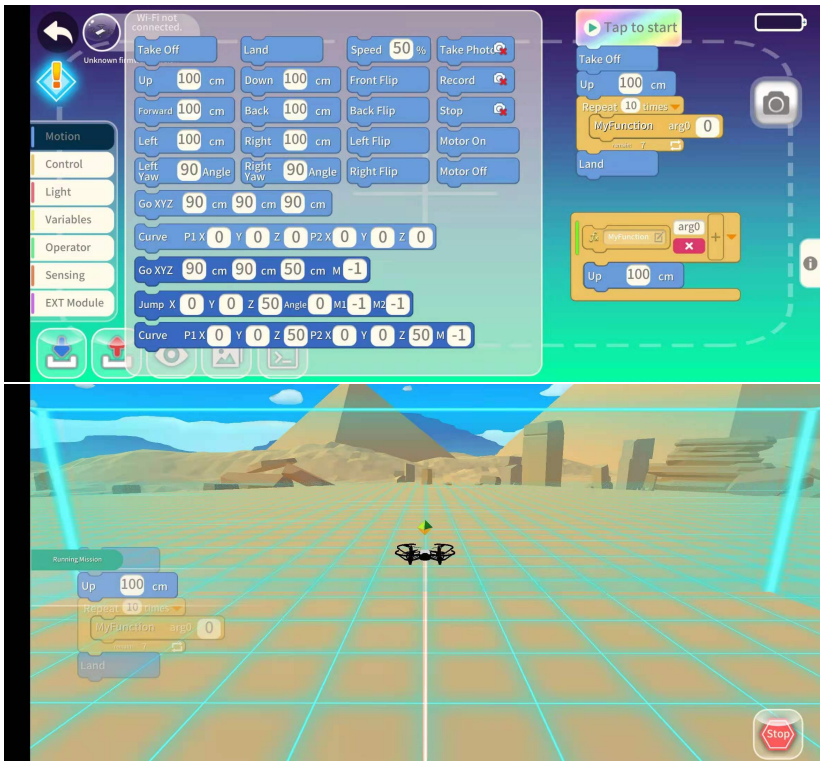


Figure 2.6: The mobile application for Tello EDU.

The above is a typical first-party mobile application for drone control. They are usually in a Scratch-like block language, and target children who are learning programming. Essentially, this mobile application is calling the SDK mentioned above.
This approach provides a simulation ability, but it does not support swarm flying. The GUI looks stimulating for children, but is not very professional.

2.3 Domain-Specific Language Design

There are two categories of domain-specific languages, internal and external.
An internal DSL is typically implemented in another host language, as a library. It usually provides some fluent interfaces, or language-like API. Internal DSL can be constrained by the host language. The `TelloEduSwarmSearch` Python library in section 2.2.2 is a good example of an internal DSL. [9]
An external DSL, on the contrary, is implemented via an independent interpreter or compiler. It can have stand-alone syntax, as its input is pure text, and does not depend on any other languages. [9]

Internal DSLs are easier to implement and maintain, as they are essentially some APIs, while for external DSLs, a new interpreter or compiler needs to be developed. However, internal DSLs are limited by the syntax of the host programming language, and are less flexible. Usually, the learning curve of an internal DSL is less steep given the learner is familiar with the host language. However, *xDrone* targets people who are professional drone operators but do not have much experience in programming, which means they may not know any programming language. This makes the learning curve of internal DSLs steeper than external DSLs in the proposed context of use, as users may need to learn a new and more complex language in order to use *xDrone*. For the reasons above, we have decided to implement *xDrone* as an external DSL.

2.4 Compilation

2.4.1 Phases of a Compiler

Alfred et al. [10] have summarised a typical deposition of a compiler into the following phases:

Lexical Analysis

Lexical analysis or scanning is the first step of a compiler. Its input is the source language in the form of a sequence of characters, and turns it into a sequence of tokens.

Syntax Analysis

After the lexical analyser produced a token stream, the syntax analyser, or the parser, will generate a data structure representing the grammatical structure of the sequence of tokens. A typical data structure used is a syntax tree, explained in 2.4.2.

Semantic Analysis

Then, semantic analysis is performed. It checks the semantic consistency of the source program by interpreting the information in the syntax tree and the symbol table. Type checking also happens in this phase; all the operator and operands will be checked to ensure they are matched.

Intermediate Code Generation

After steps of analysis, the compiler will start to synthesise the target language. Since many compilers generate low-level programs, a similar intermediate representation is generated for optimisation. This intermediate representation must be easy to produce, and to be translated into the target language.

Code Optimisation

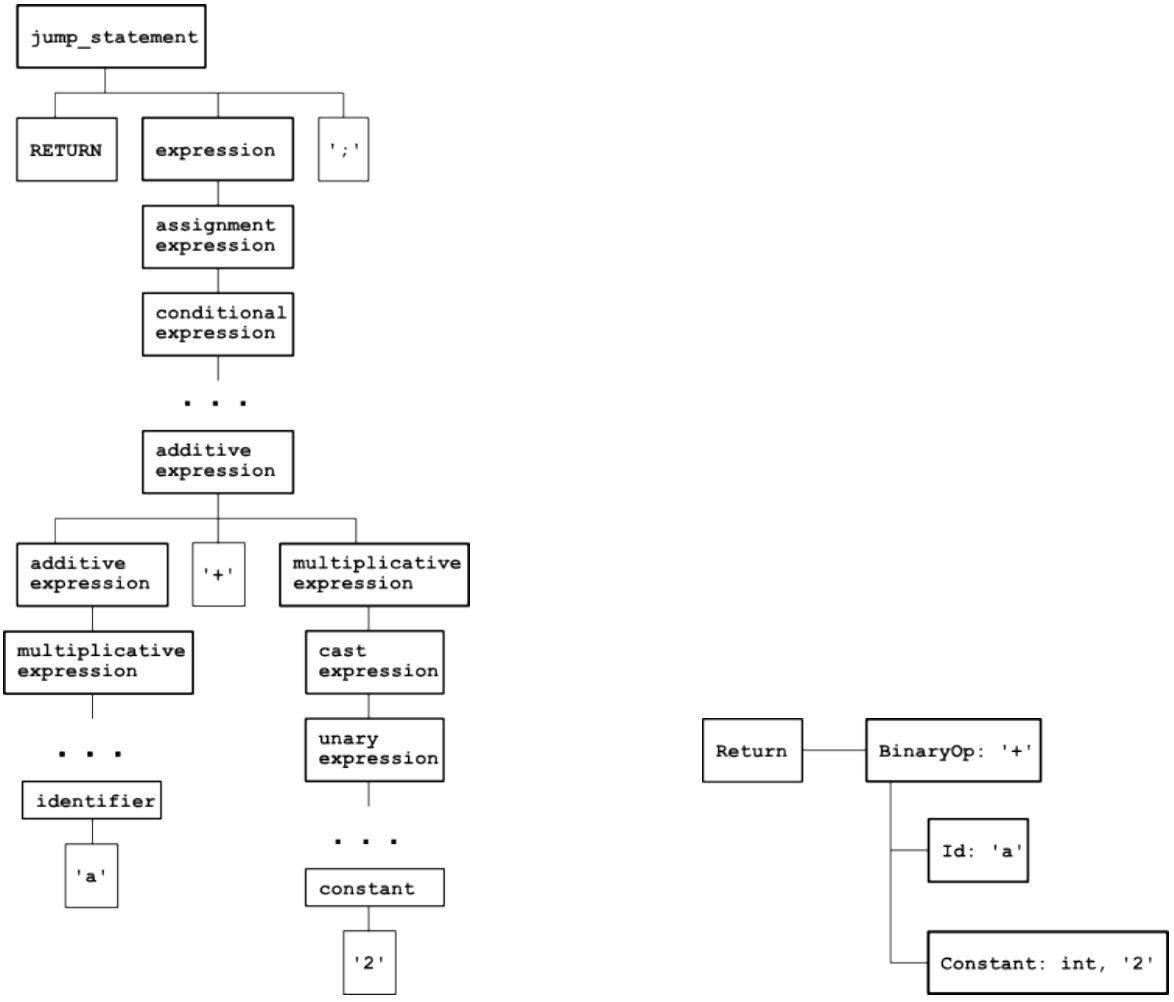
In this phase, the intermediate code is improved so that a better target code can be produced. The code after optimisation is usually shorter in length and more efficient.

Code Generation

As the final step of the whole process, the code generator translates the intermediate representation into the target language.

2.4.2 Abstract Syntax Tree

A parse tree is generated during parsing. However, the parse tree retains all of the information of the source language, including many "helpers" that are not required in the generation of the target language. These helpers usually are tokens that do not add any meaning to the language, e.g. parenthesis and white-spaces, and non-terminal symbols used as interior nodes. This is why the idea of the abstract syntax tree (AST) is introduced. In the AST, all the "helpers" are dropped and only essential tokens are kept. Interior nodes in an AST represent programming constructs (the "meaningful" tokens), but not non-terminal symbols used to represent terms, factors, or other variations of expressions. [10]



(a) The parse tree. **(b)** The AST.

Figure 2.7: The parse tree and the AST of the C statement `return a + 2;`. [11]

In the example above, we can see that the parse tree contains unimportant tokens like the

semi-colon, and there are huge chains of non-terminal symbols that represent expressions. This complexity is not desired during compilation. After extracting useful information, an AST can be constructed and used to represent the syntax better.

2.4.3 Symbol Table

A symbol Table is generated during the analysis phases, and used in the target code generation (the last three phases). It stores information about identifiers, e.g. the symbol name, the type, and the attribute. Usually, there is one symbol table for each scope, in order to support multiple declarations of the same identifier. [10]

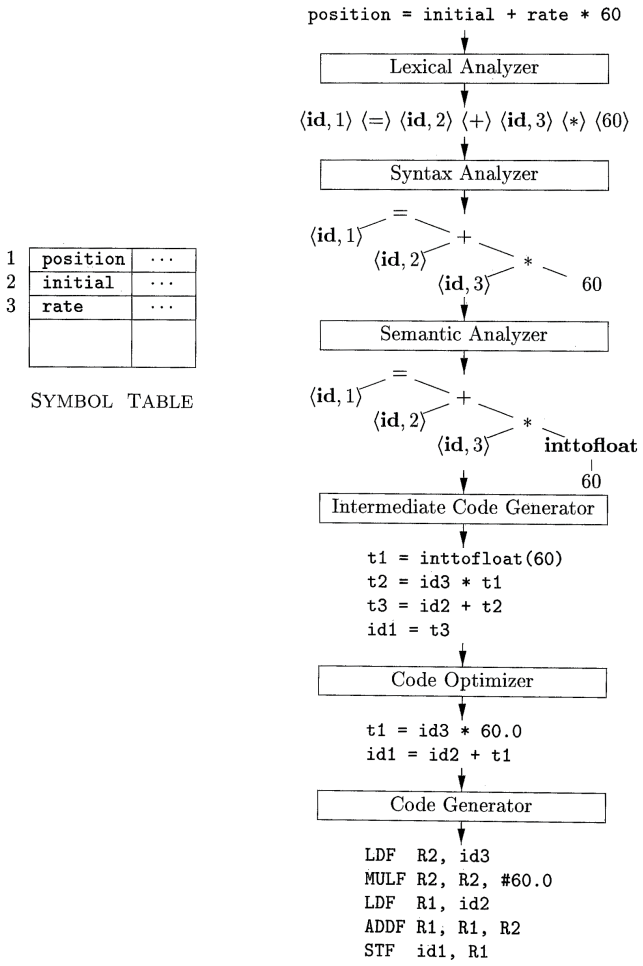


Figure 2.8: Phases of a compiler.[10]

2.4.4 ANTLR

ANTLR is a parser generator that can be used to process and translate structured text. [12] It saves language developers' effort by performing lexical analysis and syntax analysis, and converting the code string into an AST. Then, language developers can directly create a subclass of the generated `Visitor` class, to perform the semantic check and generate intermediate code.

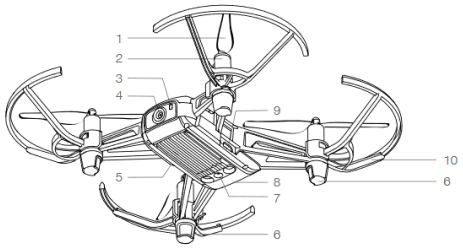
2.5 Drone Selection

2.5.1 Requirements

There are various drones in the market, all differ in functions and prices. To meet the project's requirement, we need to choose our drone carefully.
The very first requirement is that the drone selected must be programmable, and ideally should provide its APIs in Python. Because in the last version (version four), the back-end of translating *xDrone* code and deploying programs onto the drone was written in Python. Although drones of different model and brand can provide different APIs, as long as we can call their APIs in Python, the code can be easily adapted and reused.
In addition, the drone should be able to have a speed setting function or move by a certain distance. In the last version, the drone could only move in a direction for a certain time, which was frustrating because it limited the support of many distance-related functionalities - the team had to do experiments to calculate the average speed.
Swarm flying support is an important feature to have, as one of the language features of *xDrone* is multi-drone control.
Another requirement is that, due to the Covid-19 pandemic, it is more difficult than usual for people to work in the same region. A drone that is accessible and purchase-able globally, having reliable production capacity, and is relatively inexpensive should be chosen.

2.5.2 DJI Tello EDU

DJI Tello EDU [13] meets all the requirements above. It allows programs to send text commands to it via a WiFi UDP port. [14] Apart from moving by distance, it also supports 720p 30 frames per second video streaming.



1. Propellers
2. Motors
3. Aircraft Status Indicator
4. Camera
5. Power Button
6. Antennas
7. Vision Positioning System
8. Flight Battery
9. Micro USB Port
10. Propeller Guards

Command	Description
`ap ssid pass`	set WiFi access point
`takeoff`	auto takeoff
`land`	auto land
`streamon`	set video stream on
`streamoff`	set video stream off
`up x`	fly up with distance x cm
`forward x`	fly forward with distance x cm
`left x`	fly left with distance x cm
`cw x`	rotate x degree clockwise
`go x y z speed`	fly to x y z in speed (cm/s)
`speed x`	set speed to x cm/s
`speed?`	get current speed (cm/s)
`height?`	get height (cm)
...	...

(a) The aircraft diagram of Tello EDU. [15]

(b) Some commands supported by Tello EDU. [14]

Figure 2.9: Information about Tello EDU.

If there is only one drone to control, we can connect to the drone's WiFi, and send commands to and receive responses from Tello at IP 192.168.10.1 via port 8889. When the current state

(e.g. speed, battery, or height) is inquired, Tello will send the data back through port 8890. If the stream function is turned on, the live streaming will be sent via port 11111. [14]
If swarm flying is desired, we firstly need to connect to each drone's WiFi, then send the `ap` command to connect the drones with a third-party WiFi. After connecting all drones to the third-party WiFi, we can connect to this WiFi, perform similar actions as the one-drone case, except that the IP addresses of the drones have been changed.

2.6 Statistics

2.6.1 Cumulative Distribution Function

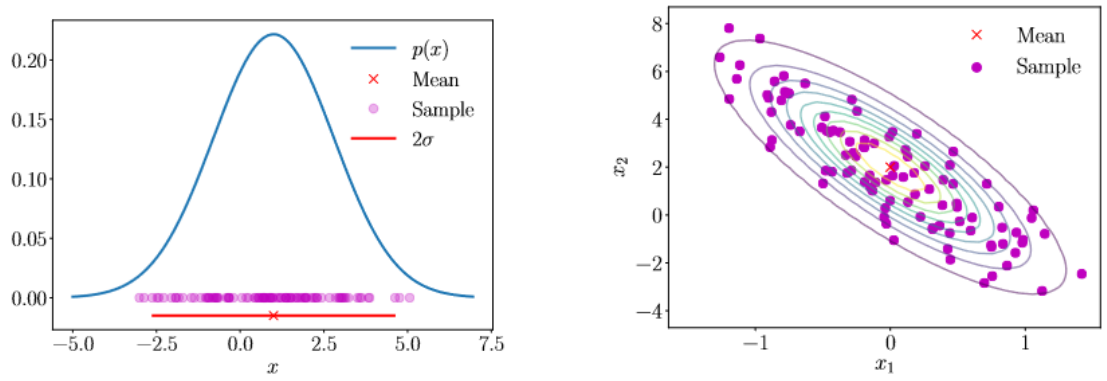
The cumulative distribution function, CDF, of a random variable X is the probability that X will take a value that is less than or equal to the function's input x [16], i.e.

$$F_X(x) = P(X \leqslant x)$$

2.6.2 Multivariate Normal Distribution

The multivariate normal distribution is a generalization of the normal distribution to higher dimensions.
A real random vector $\mathbf{X} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ if and only if there exist $\boldsymbol{\mu} \in \mathbb{R}^k$, $\mathbf{Z} \in \mathbb{R}^\ell$, $\boldsymbol{A} \in \mathbb{R}^{k \times \ell}$ such that $\mathbf{X} = \boldsymbol{A}\mathbf{Z} + \boldsymbol{\mu}$ for independent and identically distributed $Z_n \sim \mathcal{N}(0, 1)$. The covariance matrix $\boldsymbol{\Sigma}$ is defined as $\boldsymbol{A}\boldsymbol{A}^T$.
Or equivalently, the random k-vector $\mathbf{X}$ follows a multivariate normal distribution if and only if for every $\mathbf{a} \in \mathbb{R}^k$, the one-dimensional random variable $\mathbf{a}^T\mathbf{X}$ follows a univariate normal distribution. [17]
A normal random vector $\mathbf{X}$ has the following properties: [17]

- Every component of $\mathbf{X}$ is normal.
- Sum of components of $\mathbf{X}$ is normal.
- Every marginal distribution is normal.



(a) Univariate (one-dimensional) Gaussian; The red cross shows the mean and the red line shows the extent of the variance.

(b) Multivariate (two-dimensional) Gaussian, viewed from top. The red cross shows the mean and the colored lines show the contour lines of the density.

Figure 2.10: Univariate and multivariate normal distribution.[16]

2.6.3 Non-Central Chi-Squared Distribution

The random variable $\sum_{i=1}^{k} X_i^2$ follows the non-central chi-square distribution, if $(X_1, X_2, \ldots, X_k)$ are k independent and normally distributed random variables with means μ_i and unit variances. The non-central chi-squared distribution has two parameters: k, the degrees of freedom, i.e. the number of X_i, and $\lambda = \sum_{i=1}^{k} \mu_i^2$, the non-centrality parameter. [18]
It has the CDF

$$1 - Q_{\frac{k}{2}}\left(\sqrt{\lambda}, \sqrt{x}\right)$$

where Q is the Marcum Q function [19]

$$Q_M(a, b) = \int_b^\infty x \left(\frac{x}{a}\right)^{M-1} \exp\left(-\frac{x^2 + a^2}{2}\right) I_{M-1}(ax)\, dx$$

where I_{M-1} is the modified Bessel function of order $M - 1$.

Chapter 3

Language Features

3.1 Key Principles

When designing the *xDrone* language, some key principles are followed.
First, this language should be easy to write and understand. As the targeted users are drone operators, but are new to or less experienced in programming, this language must be straightforward. Also, it does not make sense to create a more complicated language than most mainstream languages but only for drone control.
Second, *xDrone* should be versatile. The same program in *xDrone* can be run on drones of different brands or series. This means we should only support drone commands that are compatible with most drones, e.g. move and rotate. We also need to decouple the drone API calls from the intermediate code generated by the compiler.
Also, as a basic drone does not have any sensors, we are not going to create programs in *xDrone* that depend on sensor inputs, i.e. we cannot write code to let a drone perform actions depending on some real-world conditions, e.g. see a white light, or reach a certain height.
A typical program is to fly the drone in a relatively simple pattern, perhaps repeatedly, or coordinate multiple drones to fly in a pattern, perhaps with some synchronisation.

3.2 Syntax Rules

To make *xDrone* a more sophisticated language, its syntax rules are redesigned, and language features like expressions, functions and flow controls are added. In this section, the syntax rules of *xDrone* will be discussed briefly. A more detailed language specification can be found in appendix A.

3.2.1 Movement Commands

xDrone supports 11 basic movement commands, namely `takeoff`, `land`, `up`, `down`, `left`, `right`, `forward`, `backward`, `rotate_left`, `rotate_right`, and `wait`.
The drone that the movement command is called on should also be specified in the code, e.g.

```
DRONE1.forward(1);
```

means to fly `DRONE1` forward for 1 metre.

3.2.2 Expressions, Variables, and Types

A very fundamental feature that most languages have is expressions. They are typically useful during some arithmetic calculations, and the results can be stored into variables, so that programmer will no longer need to perform calculations separately and hard code numbers in their programs.
In order to support expressions and variables, types are also introduced, there are 7 types of variables in *xDrone*, namely `int`, `decimal`, `string`, `boolean`, `vector`, `drone`, and `list`.
Many operators are supported, including arithmetic operators (e.g. `+`, `-`), comparison operators (e.g. `==`, `>`), logical operators (e.g. `and`, `not`).
The following is an example of using variables with type `vector`, and subtraction between `vectors` to calculate the distance difference between x-axes of two variables.

```
vector current_position <- (0.0, 0.0, 0.0);
vector destination <- (2.0, 2.0, 2.0);
decimal distance <- (destination - current_position).x;
DRONE1.forward(distance);
```

The type `list` also exists in *xDrone*. It is easy in *xDrone* to insert or remove any value to/from a list. The existence of lists enables list traversal, e.g. taking off all drones in a list,

```
list[drone] drones <- [DRONE1, DRONE2, DRONE3];
int i;
for i from 0 to drones.size - 1 {
  drones[i].takeoff();
}
```

or creating a Fibonacci sequence,

```
list[int] fibonacci <- [1, 1];
while fibonacci.size < 10 {
  int length <- fibonacci.size;
  fibonacci.insert(fibonacci[length - 1] + fibonacci[length - 2]);
}
```

3.2.3 Flow-Controls

When programmers want to traverse a list, or repeat a certain block of code multiple times, some flow control commands, e.g. loops, can come in handy.
xDrone supports traditional `if-else` branches, and `while` loops, as well as a `for` loop that traverses integers in an interval, and a `repeat` loop that simply repeat its content multiple times.
The following is an example of the `for` loop,

```
int i;
for i from 0 to 10 step 2 {
  DRONE1.forward(i);
}
```

The following is an example of the `repeat` loop,

```
repeat 4 times {
  DRONE1.forward(2);
}
```

Though a `while` loop can replace both the `for` loop and the `repeat` loop, we still keep them, as most of the time in drone programming, a simple `repeat` loop is sufficient, and importantly, it also improves readability.

3.2.4 Procedures and Functions

Procedures and functions improve the reusability and readability of *xDrone*. With them, programmers will be able to code a task and apply it to different drones easily, instead of copying and pasting the same code block multiply times.
For example, the following procedure can be reused by different drones, to draw squares with different side length.

```
procedure draw_square(drone my_drone, decimal side_length) {
  repeat 4 times {
    my_drone.forward(side_length);
    my_drone.rotate_right(90);
  }
}
```

If a programmer wants the drones to draw squares with different sizes, instead of copying the `repeat` block multiple times, he can write a `draw_square` procedure first, then only lines of code that call this procedure with different parameters, e.g.

```
draw_square(DRONE1, 1.5);
draw_square(DRONE1, 1.0);
draw_square(DRONE2, 0.5);
```

This greatly improves the readability, and reduces the effort needed to write an *xDrone* program.

Also note that recursions are supported, e.g.

```
function factorial(int n) return int {
  if n <= 1 {
    return n;
  }
  return n * factorial(n - 1);
}
```

3.3 Multiple-Drone Control

Apart from more advanced language syntax, *xDrone* now supports multiple-drone control. When controlling a number of drones, we need to consider the cases of both synchronous and asynchronous commands. Drones should be able to perform actions one by one sequentially, or different drones could act in parallel.
A parallel command is introduced to achieve this,

```
{ DRONE1.takeoff(); } || { DRONE2.takeoff(); };
```

Code in each parallel branch will start simultaneously, and wait until all other branches finish. For example,

```
DRONE1.takeoff();
DRONE2.takeoff();
{ DRONE1.forward(1); } || { DRONE2.forward(10); };
DRONE1.land();
DRONE2.land();
```

means that DRONE1 and DRONE2 takeoff one by one. Then DRONE1 moves forwards for 1 metre, meanwhile DRONE2 move forwards for 10 metres. After both DRONE1 and DRONE2 finish their forward movement, they will land one by one.
Note that this parallel command can even be nested.
With this feature, programmers can fly multiple drones at the same time, and synchronise them when necessary.

3.4 Safety Check

Another key feature that makes *xDrone* different from mainstream programming languages is the safety check. *xDrone* will automatically check whether your drones will fly beyond your space limit, or collide, if there are multiple drones.

3.4.1 Boundary Check

Programmers can specify the size of their environment in a configuration file to limit their drones in a certain space.
In the configuration file, the initial positions of the drones, and the boundaries of x-, y-, z-axis can be set. Also, the maximum flying time is also configurable, which can be used to make sure the drones will not run out of battery. During the compilation, after each drone movement, its estimated position and flying time will be updated and compared with the boundary thresholds. If any of the movements violates the threshold, the compilation will fail, and an error will be reported. The error will be printed like the following,

```
When running command 'DRONE1.up(100);', boundary limits are violated:
Drone 'DRONE1': the z coordinate 102 will go beyond its upper limit 30
```

which contains the movement command that caused the error, and the information about which threshold was violated by what value.
Drone takeoff state checks also happen in this phase, to ensure movement commands, e.g. moving forwards, are run when the drone has been taken off correctly, and the drone does not take off twice or land twice.

3.4.2 Basic (Deterministic) Collision Check

When controlling multiple drones, if there is any chance that two drones fly too close, the collision checker will report this potential danger.
As mentioned in the boundary check, estimated flying time and positions of drones are calculated; but in addition to this, distances between drones are also calculated from their estimated positions. The distances will be compared to a user-specified threshold, which is usually slightly greater than the diameter of the drone. If the distance between two drones is less than the threshold, they are very likely to collide, the compilation will fail, and an error will be reported. The error will be printed like the following,

```
Collisions might happen!
Collision might happen between DRONE1 and DRONE2, at time 2.8s,
near position (x=0.9m, y=0.0m, z=1.0m), distance=0.2m, confidence=100.000%
Collision might happen between DRONE1 and DRONE2, at time 2.9s,
near position (x=0.95m, y=0.0m, z=1.0m), distance=0.1m, confidence=100.000%
```

which contains all sample data points of all possible collisions, with names of the drones involved, and the estimated time and position.

3.4.3 Advanced (Stochastic) Collision Check

The collision check mentioned in section 3.4.2 assumes that drones would not deviate. However, drone deviation is likely to happen during drone flights in the real world.
xDrone provides a stochastic collision check. This requires programmers to measure the variances of their drones, and enter them in the configuration file. Then *xDrone* can calculate the confidence, or probability, of collision for each sample time point. The basic deterministic collision check acts as a default and special case of the stochastic collision check, with all variances set to 0.
With the stochastic collision check enabled, the following report can be generated and saved:

Drone 1	Drone 2	Time	Distance	Confidence
DRONE1	DRONE2	0.1	1	0.000%
DRONE1	DRONE2	0.2	1.00499	0.000%
DRONE1	DRONE2	0.3	1.0198	0.000%
...	...	...	...	...
DRONE1	DRONE2	2.6	0.5	0.002%
DRONE1	DRONE2	2.7	0.4	1.750%
DRONE1	DRONE2	2.8	0.3	43.090%
DRONE1	DRONE2	2.9	0.2	95.291%
DRONE1	DRONE2	3	0.1	99.968%
DRONE1	DRONE2	3.1	0	100.000%
DRONE1	DRONE2	3.2	0.1	99.949%
...	...	...	...	...
DRONE1	DRONE2	3.9	0.8	0.000%
DRONE1	DRONE2	4	0.9	0.000%
...	...	...	...	...

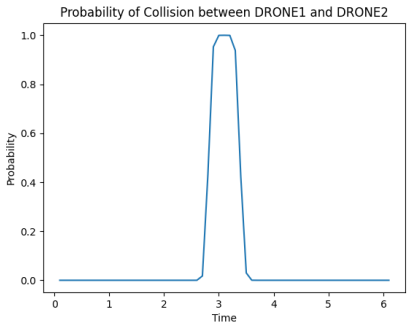


Figure 3.1: The collision probability report.

The programmer can also set a threshold on the confidence. When the collision confidence is higher than this threshold, the compiler will report this possible collision as a safety error, just like the deterministic collision check does. The following is the error printed if the threshold is set to 99.95%.

```
Collisions might happen!
Collision might happen between DRONE1 and DRONE2, at time 2.9s,
near position (x=0.95m, y=0.0m, z=1.0m), distance=0.1m, confidence=99.968%
Collision might happen between DRONE1 and DRONE2, at time 3.0s,
near position (x=1.0m, y=0.0m, z=1.0m), distance=0.0m, confidence=100.000%
```

Chapter 4

Implementation Details

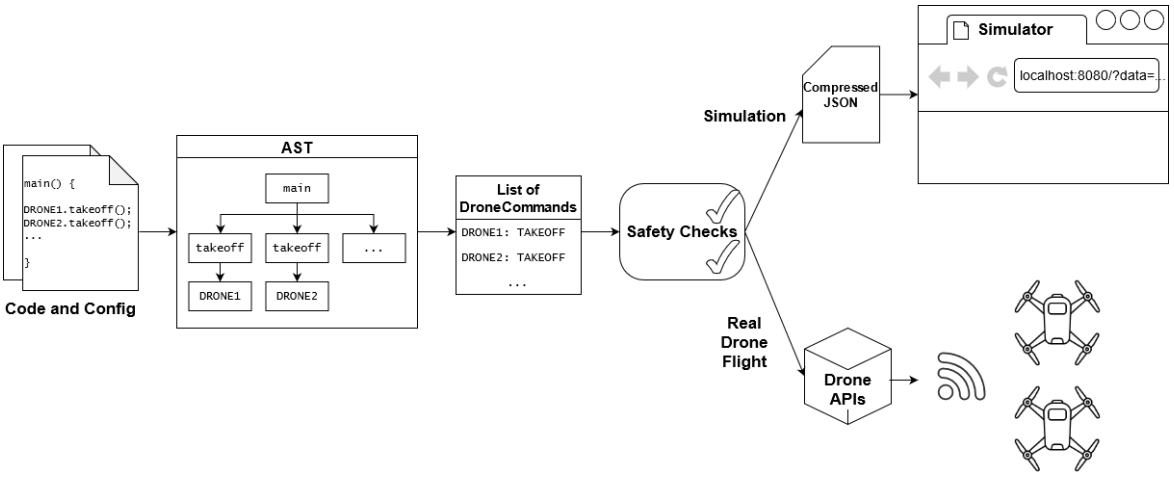


Figure 4.1: Flow chart of the compiler.

4.1 Compiler

The configuration JSON will be parsed and converted into some configuration data classes, e.g. `DroneConfig`, where all the parameters, like the name and the speed of the drone, are stored.
The code will be handled by ANTLR, which will parse the code string into an AST. Then a visitor will traverse all nodes in the AST, handle each node accordingly, unroll all expressions, variables, and functions. At the same time, semantic checks, e.g. the type check, are performed. Finally, the visitor will output a list of `DroneCommands`.
`DroneCommand` is an abstract class, and it has two subclasses, `SingleDroneCommand` and `ParallelDroneCommands`.
A `SingleDroneCommand` object represents a sequential command. It contains a `Command`, i.e. the action to be performed, and the name of the drone so that we know which drone the action will be performed on.
A `ParallelDroneCommands` object represents multiple branches of sequential commands, and all branches will be run in parallel. It stores a list of branches. Each branch is a sequence of `DroneCommands`, so both `SingleDroneCommand` and `ParallelDroneCommands` can appear in a branch.

4.2 Safety Checker

4.2.1 Boundary Check

Having the list of `DroneCommand`s parsed, the safety checks are then performed.
During the boundary check, the estimated flying time and position of drones will be calculated after each `DroneCommand`, according to the configuration data classes parsed from the configuration JSON. There will be one check after each command, where the time and position are compared with the boundary thresholds, to check whether they have violated the limits.
There will also be a boolean value that records the takeoff state of the drone, to ensure the drone performs movement commands in a correct state, e.g. only moves forward when taken off, and cannot take off twice.
The following is an example of boundary check for command `DRONE1.right(0.4);`.

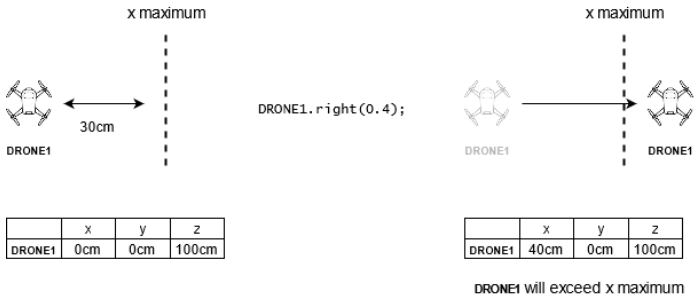


Figure 4.2: The check after each command in the boundary check.

4.2.2 Basic (Deterministic) Collision Check

During the basic collision check, paths of drones are calculated from the `DroneCommand` list and their configurations. For each drone, its path is split into sample data point slices based on the time used. The default time interval is 0.1 second. Then, in every time slice, the distance between sample points of different drones are calculated and compared with a threshold, to check whether the drones are too close.
The following is an example of collision check for command `DRONE3.left(0.3);`.

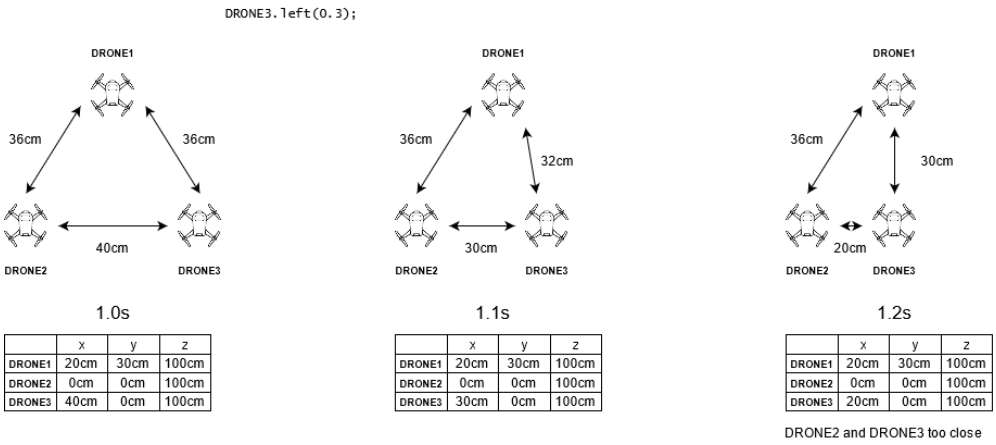
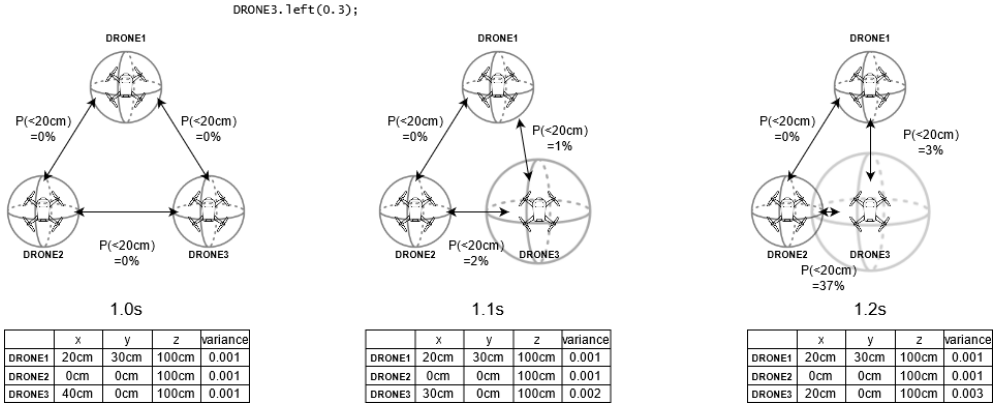


Figure 4.3: Time slices used in the basic deterministic collision check.

4.2.3 Advanced (Stochastic) Collision Check

Similar to the implementation of the deterministic collision check, the stochastic collision check splits the drones' paths into sample data points. However, instead of calculating the distance between drones and comparing it with a threshold, the stochastic collision checker calculates the probability that the distance is less than the threshold based on the variances of drones.
After each time slice, if the drone has moved, its variance will increase by a value that is proportional to the distance it travelled and the `"variance_per_meter"` field in the configuration.



	x	y	z	variance
DRONE1	20cm	30cm	100cm	0.001
DRONE2	0cm	0cm	100cm	0.001
DRONE3	40cm	0cm	100cm	0.001

	x	y	z	variance
DRONE1	20cm	30cm	100cm	0.001
DRONE2	0cm	0cm	100cm	0.001
DRONE3	30cm	0cm	100cm	0.002

	x	y	z	variance
DRONE1	20cm	30cm	100cm	0.001
DRONE2	0cm	0cm	100cm	0.001
DRONE3	20cm	0cm	100cm	0.003

Figure 4.4: Time slices used in the advanced stochastic collision check.

Collision Probability Calculation

We assume the position of drone i follows a multivariate normal distribution $\mathbf{P_i} \sim \mathcal{N}(\boldsymbol{\mu_i}, \boldsymbol{\Sigma_i})$, where

$$\boldsymbol{\mu} = \begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix}, \qquad \boldsymbol{\Sigma} = \begin{pmatrix} \sigma_i^2 & 0 & 0 \\ 0 & \sigma_i^2 & 0 \\ 0 & 0 & \sigma_i^2 \end{pmatrix}$$

If the position of drone 1 follows $\mathbf{P_1} \sim \mathcal{N}(\boldsymbol{\mu_1}, \boldsymbol{\Sigma_1})$, and the position of drone 2 follows $\mathbf{P_2} \sim \mathcal{N}(\boldsymbol{\mu_2}, \boldsymbol{\Sigma_2})$, then the difference between two drones, $\mathbf{P_1} - \mathbf{P_2}$, also follows a multivariate normal distribution $\mathbf{P_1} - \mathbf{P_2} \sim \mathcal{N}(\boldsymbol{\mu_1} - \boldsymbol{\mu_2}, \boldsymbol{\Sigma_1} + \boldsymbol{\Sigma_2})$, where

$$\boldsymbol{\mu_1} - \boldsymbol{\mu_2} = \begin{pmatrix} x_1 - x_2 \\ y_1 - y_2 \\ z_1 - z_2 \end{pmatrix}, \qquad \boldsymbol{\Sigma_1} + \boldsymbol{\Sigma_2} = \begin{pmatrix} \sigma_1^2 + \sigma_2^2 & 0 & 0 \\ 0 & \sigma_1^2 + \sigma_2^2 & 0 \\ 0 & 0 & \sigma_1^2 + \sigma_2^2 \end{pmatrix}$$

We can rewrite the random vector $\mathbf{P_1} - \mathbf{P_2} = (X, Y, Z)^T$, where X, Y, and Z all follow normal distribution:

$$\begin{aligned} X &\sim \mathcal{N}(x_1 - x_2, \sigma_1^2 + \sigma_2^2) \\ Y &\sim \mathcal{N}(y_1 - y_2, \sigma_1^2 + \sigma_2^2) \\ Z &\sim \mathcal{N}(z_1 - z_2, \sigma_1^2 + \sigma_2^2) \end{aligned}$$

We define D as the distance between the two drones. Then, the squared distance D with unit variance is

$$\frac{D^2}{\sigma_1^2 + \sigma_2^2} = \frac{X^2}{\sigma_1^2 + \sigma_2^2} + \frac{Y^2}{\sigma_1^2 + \sigma_2^2} + \frac{Z^2}{\sigma_1^2 + \sigma_2^2}$$

This $\frac{D^2}{\sigma_1^2+\sigma_2^2}$ follows a non-central chi-squared distribution, with degrees of freedom $k = 3$, and the non-centrality parameter

$$\lambda = \frac{(x_1 - x_2)^2}{\sigma_1^2 + \sigma_2^2} + \frac{(y_1 - y_2)^2}{\sigma_1^2 + \sigma_2^2} + \frac{(z_1 - z_2)^2}{\sigma_1^2 + \sigma_2^2}$$

Thus, the probability of the event that the distance is less than or equal to the collision threshold d is

$$P(D \leqslant d) = P\left(\frac{D^2}{\sigma_1^2 + \sigma_2^2} \leqslant \frac{d^2}{\sigma_1^2 + \sigma_2^2}\right) = F\left(\frac{d^2}{\sigma_1^2 + \sigma_2^2}\right)$$

where F is the cumulative distribution function of the non-central chi-squared distribution discussed above, and can be easily obtained by function `scipy.stats.ncx2.cdf`.

4.3 Simulation

After all safety checks have passed, the list of `DroneCommands` will be further processed. If a simulation is desired, the commands and configurations will be passed into a simulation converter. The simulation converter converts the list of `DroneCommands` to a JSON object. The JSON object will then be compressed into a shorter string by the zlib library [20], and used as a parameter in the simulation URL, which will be sent to the simulator web page.

The simulator has been further developed and improved from the last version, to adapt to the changes in *xDrone*.
First of all, the simulator has been extracted from the Drone Playground into a stand-alone application. It supports simulation for multiple drones, as multiple-drone control is a newly added feature. In addition, the play speed of the simulation is adjustable, and it can be paused and replayed. The time used is also shown on the simulator, so that the developer can know the estimated flying time during the simulation.

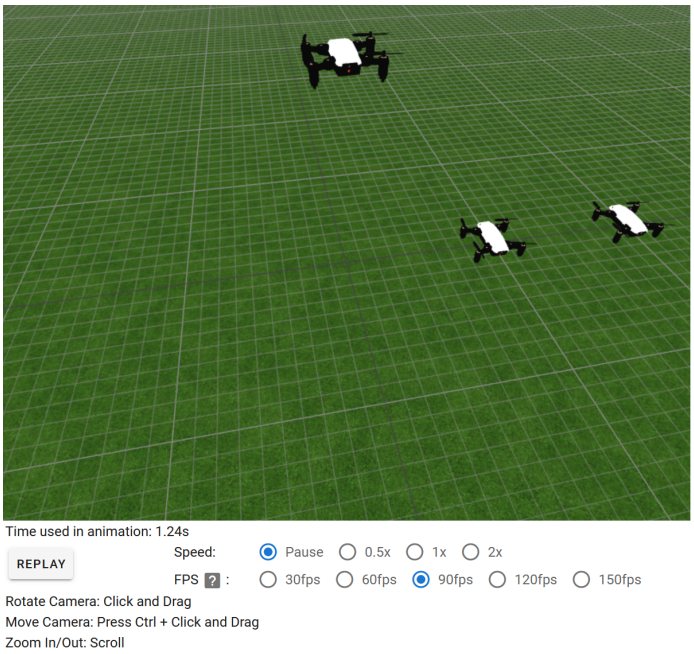


Figure 4.5: The *xDrone* simulator.

It is worth discussing the FPS, frames per second. As the position and orientation of the drones are changed accumulatively in each frame at a fixed rate (the speed/rotate speed of the drone), if the FPS is too low, the step size between each time point sample is large, and drones will be likely to overshoot in the simulation; however, if the FPS is too high, the burden on the browser which runs the simulator will rise, until the limit is reached, at which point the number of frames is too high to be handled in one second, and a delay happens, the simulation will be slowed down. There will be a further discussion on this in section 6.4.1.

4.4 Real Drone Flight

If a real drone flight is wanted, `DroneCommands` will be passed into a drone command executor, which will convert the `DroneCommands` into corresponding drone API calls, or call related library functions, in order to execute these commands on the real drones. Different drone command executors with different APIs can be implemented and used if drones of different brands and series are manipulated.
In my implementation for DJI Tello EDU drones, `TelloEduSwarmSearch` library [8] is used, which is a third-party package of APIs that controls Tello EDU drones. An example of using this package can be found in section 2.2.2.
The following is a picture of *xDrone* controlling two Tello EDU drones to fly at the same time.

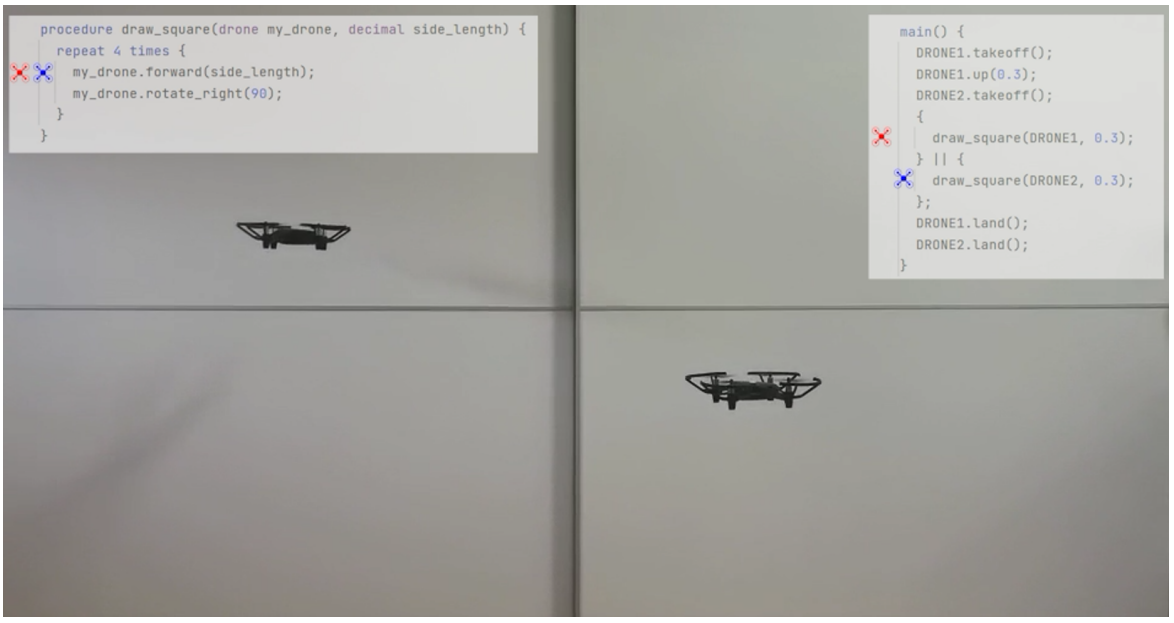


Figure 4.6: A flight of two Tello EDU drones.

4.5 Command-Line Interface

In order to run the compiler, a command-line interface is implemented. The click library [21] is used, which helps to create an easy and clear command-line interface with very little code.
The *xDrone* command-line interface can be used to validate a *xDrone* program, simulate the

program in the simulator, and run the program on real drones. The detailed specification can be found in appendix B.

```
xdrone [--validate | --simulate | --fly] --code <path> --config <path>
```

The terminal command with flag `--validate` can be run to see whether there is a syntax error or semantic error in the code, the safety checks will also be performed, and any safety error will be reported.
With the simulator server installed, the simulation terminal command with the `--simulate` flag can be run to start a simulation. This command will firstly perform the validation, and it will print a link to the simulator in the terminal, and automatically open the web page to display the simulation.
The terminal command with flag `--fly` can be run to start the real drone flight. Before this, the terminal and the Tello EDU drones should be connected to the same WiFi. This command will firstly perform the validation, then send commands via APIs to real drones, to start a flight.

4.6 Pypi Package

The *xDrone* compiler with the terminal command as an entry point has also been packaged in a Pypi package [22], so that developers can simply run

```
pip install xdrone
```

to install the compiler.
After the *xDrone* compiler is installed, the terminal commands mentioned in section 4.5 become available in your terminal. The source code of the *xDrone* compiler is therefore hidden from the *xDrone* programmers.

4.7 Unit Tests

Python unit test package `unittest` was used to test the compiler. The test coverage has also been checked, to ensure the unit tests have 100% coverage.
In the Travis pipeline, the following command is run at the test stage to perform an automated unit test with the coverage check as well.

```
pytest --cov=xdrone --cov-fail-under=100
```

There are mainly two categories of unit tests. One is to check whether the output intermediate code matches the expected ones given the input code string. This category of unit tests is widely used in testing the functionality of the compiler. The other is to check whether desired functions are called with expected arguments. This is done by applying some mocked objects, and is mainly used to test some functional modules, like the drone API caller.

Chapter 5

Programming in *xDrone*

In this chapter, an example of programming in *xDrone* is provided, in order to demonstrate the work flow of how to develop in *xDrone*.

5.1 Install *xDrone*

The first step is to install *xDrone*. Since a Pypi package has been published, *xDrone* installation can be done in one line:

```
pip install xdrone
```

To enable the simulation, the *xDrone* simulator is also required. Clone the simulator repository from GitHub, use `npm install` to install required dependencies, and run

```
npm start serve
```

to start the simulator server.

5.2 Write Program in *xDrone*

Then, write the code in a text editor, and save it as a file. The following is an example, which controls two drones to draw square simultaneously.

```
procedure draw_square(drone my_drone, decimal side_length) {
  repeat 4 times {
    my_drone.forward(side_length);
    my_drone.rotate_right(90);
  }
}

main() {
  list[drone] drones <- [DRONE1, DRONE2];
  int i;
  for i from 0 to drones.size - 1 {
    drones[i].takeoff();
  }
  { draw_square(DRONE1, 1.0); } || { draw_square(DRONE2, 2.0); };
```

```
  for i from 0 to drones.size - 1 {
    drones[i].land();
  }
}
```

It is also important to include a configuration file, as the safety checkers rely on these data. This file stores a JSON object, with configuration information about the drone and some parameters of the safety checkers. It is highly recommended to fill in all the fields except `"advanced"`, but if there are some fields missing, some default values will be used instead, with warnings printed. More information can be found in appendix A.
The following is an example of the configuration file.

```
{
  "drones": [{
    "name": "DRONE1",
    "init_position": {"x": 0, "y": 0, "z": 0},
    "speed_mps": 2,
    "rotate_speed_dps": 180,
    "takeoff_height_meters": 2,
    "advanced": {
      "variance_per_meter": 0.002
    }
  },{
    "name": "DRONE2",
    "init_position": {"x": 1, "y": 0, "z": 0},
    "speed_mps": 1,
    "rotate_speed_dps": 90,
    "takeoff_height_meters": 1,
    "advanced": {
      "variance_per_meter": 0.001
    }
  }],
  "boundary_config": {
    "max_x_meters": 10,
    "max_y_meters": 20,
    "max_z_meters": 30,
    "min_x_meters": -10,
    "min_y_meters": -20,
    "min_z_meters": 0,
    "max_seconds": 100
  },
  "collision_config": {
    "collision_meters": 0.3,
    "time_interval_seconds": 0.1,
    "confidence_threshold": 0.95
  }
}
```

5.3 Validate Program

Then, run the following terminal command to see whether there is a syntax error or safety checker error.

```
xdrone --validate --code mycode.xdrone --config config.json
```

If there is a typo in the program, and the syntax rules are violated, the following line will be shown in the terminal

```
Failed to validate your program, error:
1:0: syntax error, mismatched input 'typo'
expecting {'main', 'function', 'procedure'}
```

After fixing the typo, you can validate the code again, if there is a safety checker error, or a possible collision, the validation will also fail and the error message will be printed, e.g.

```
Failed to validate your program, error:
'forward' command used when drone 'DRONE1' has not been taken off
```

and

```
Failed to validate your program, error:
Collisions might happen!
Collision might happen between DRONE1 and DRONE2, at time 2.9s,
near position (x=0.95m, y=0.0m, z=1.0m), distance=0.1m, confidence=99.968%
Collision might happen between DRONE1 and DRONE2, at time 3.0s,
near position (x=1.0m, y=0.0m, z=1.0m), distance=0.0m, confidence=100.000%
```

You can run the simulation to address why and where the safety error happens, by running

```
xdrone --simulate --code mycode.xdrone --config config.json --no-check
```

Note that the `--no-check` flag disables safety checks, so that you can view the unsafe simulation.

5.4 Generate Collision Probability Report

xDrone can also calculate the probability of collisions based on the variances of drone movements. The advanced stochastic collision check will be activated if the `variance_per_meter` is set in the configuration JSON.
You can run the following command to save the report of collision probabilities to the `./reports/` directory,

```
xdrone --validate --code mycode.xdrone --config config.json --save-report
```

The report contains tables with details about each sample time point and plots that show how the collision probability changes with time. The following is an example,

Drone 1	Drone 2	Time	Distance	Confidence
DRONE1	DRONE2	0.1	1	0.000%
DRONE1	DRONE2	0.2	1.00499	0.000%
DRONE1	DRONE2	0.3	1.0198	0.000%
...	...	...	...	...
DRONE1	DRONE2	2.6	0.5	0.002%
DRONE1	DRONE2	2.7	0.4	1.750%
DRONE1	DRONE2	2.8	0.3	43.090%
DRONE1	DRONE2	2.9	0.2	95.291%
DRONE1	DRONE2	3	0.1	99.968%
DRONE1	DRONE2	3.1	0	100.000%
DRONE1	DRONE2	3.2	0.1	99.949%
...	...	...	...	...
DRONE1	DRONE2	3.9	0.8	0.000%
DRONE1	DRONE2	4	0.9	0.000%
...	...	...	...	...

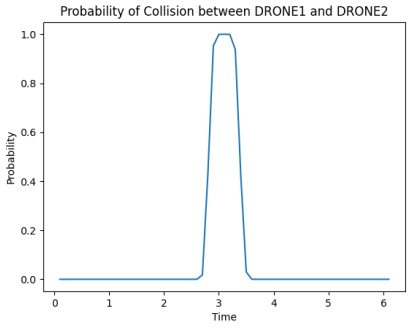


Figure 5.1: The collision probability report.

5.5 Run Simulation

After addressing and resolving the syntax error and the safety issue, you can run the simulation terminal command without the `--no-check` flag

```
xdrone --simulate --code mycode.xdrone --config config.json
```

Your program will be validated first. If it is valid, the simulator web page will automatically be opened in your browser, and you can review the simulation of your drone flight.

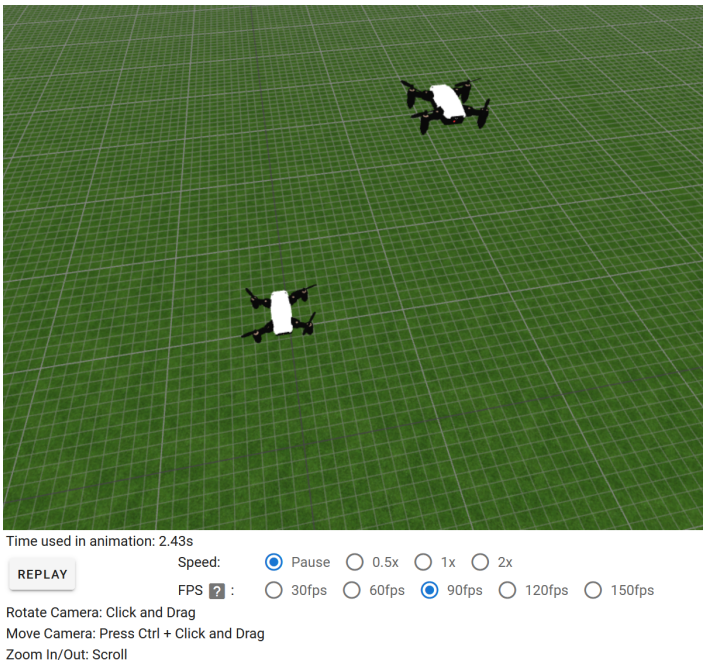


Figure 5.2: The simulation of the program in 5.2.

5.6 Fly Real Drones

If you are satisfied with the simulation of your program, you can connect your drones with the same WiFi as your terminal, the run

```
xdrone --fly --code mycode.xdrone --config config.json
```

to fly your drones.

Chapter 6

Evaluation

6.1 Collision Detection

When operating multiple drones, sometimes it is hard for human beings to distinguish whether they will collide at some point. Therefore, we introduced a collision check in *xDrone*. In this part, the collision check will be evaluated on a straightforward sample program to check whether it will report possible collisions. To make the example easier for humans to follow, drones will have the same parameter configurations, like speed and takeoff height. `DRONE2`'s initial position is 1 metre to the right of `DRONE1`.

```
main() {
  DRONE1.takeoff();
  DRONE2.takeoff();
  DRONE1.right(1);
  DRONE1.left(1);
  DRONE1.land();
  DRONE2.land();
}
```

Drones will collide, and the compiler reports

```
Collisions might happen!
Collision might happen between DRONE1 and DRONE2, at time 2.8s,
near position (x=0.9m, y=0.0m, z=1.0m), distance=0.2m, confidence=95.291%
Collision might happen between DRONE1 and DRONE2, at time 2.9s,
near position (x=0.95m, y=0.0m, z=1.0m), distance=0.1m, confidence=99.968%
Collision might happen between DRONE1 and DRONE2, at time 3.0s,
near position (x=1.0m, y=0.0m, z=1.0m), distance=0.0m, confidence=100.000%
Collision might happen between DRONE1 and DRONE2, at time 3.1s,
near position (x=0.95m, y=0.0m, z=1.0m), distance=0.1m, confidence=99.949%
Collision might happen between DRONE1 and DRONE2, at time 3.2s,
near position (x=0.9m, y=0.0m, z=1.0m), distance=0.2m, confidence=93.780%
```

There are multiple warnings, as the collision checker has a sample time interval of 0.1 second, and it reports all the sample time data points where collisions might happen.
In addition, as the variances of drone movements are set, the confidence of collisions are calculated. A plot that shows how the collision probability changes can also be generated.

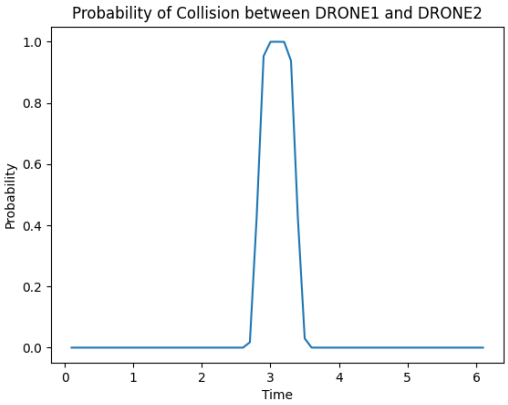


Figure 6.1: The collision probability plot.

A simulation can also be done to view the detail graphically,

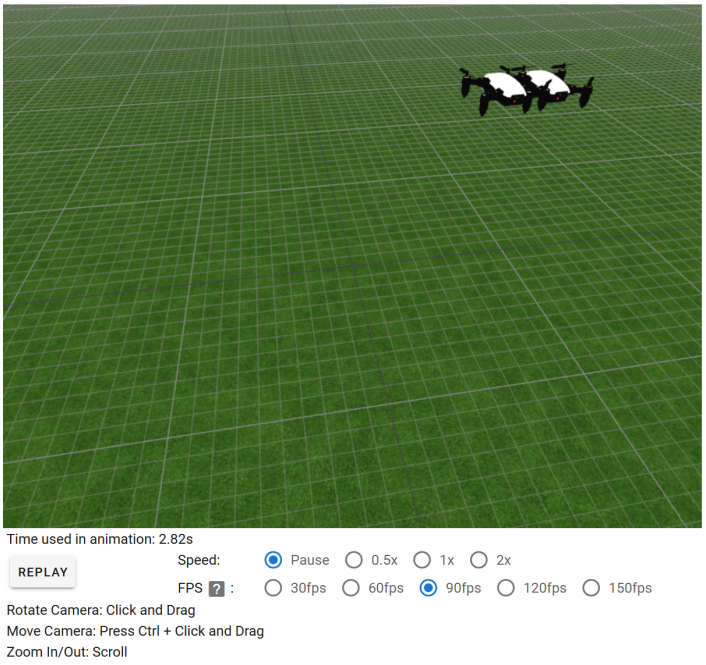


Figure 6.2: The simulation of a possible drone collision.

The "time used in animation" also shows the collision happens at 2.82 seconds after the flight started, which matches the compiler's error report.

All the information can help the programmers address the potential collisions and protect the flight safety.

6.2 *xDrone* v.s. Earlier Version

This section compares *xDrone* with its earlier version, in order to see what improvements have been made to *xDrone*.
In the earlier version, although *xDrone* can write programs that have the same effect, it was

more tedious and less readable.
For example, if I want to draw a rough circle, i.e. a regular polygon with a large number of edges, in the current version, the program is simple and easily understandable:

```
main() {
  int number_of_edges <- 20;
  decimal side_length <- 0.1;
  DRONE1.takeoff();
  repeat number_of_edges times {
    DRONE1.forward(side_length);
    DRONE1.rotate_right(360.0 / number_of_edges);
  }
  DRONE1.land();
}
```

However, in the previous version, the program will be like

```
fly() {
  TAKEOFF();
  FORWARD(0.1);
  ROTATERIGHT(18);
  FORWARD(0.1);
  ROTATERIGHT(18);
  ......              # 18 sets of forward and rotate omitted
  LAND();
}
```

As there were no loops supported, the programmer has to copy and paste the same code multiple times; if the number of iterations is large, this can be tedious and error-prone. Also, the readability is worse because no operators and variables supported in the version before - there are many "magic numbers".
Additionally, the earlier version does not support multiple drone control, and has a very limited and non-adjustable safety check, which is only to check whether the drone is in a 4 by 4 metre space.
The following table provides a summary of the comparison.

	xDrone	*xDrone* previous version
Lines of code	Few, does not depend on number of iterations (10 lines in example above)	Many, depends on number of iterations (44 lines in example above)
Readability	High, as variables and expressions supported	Low, only "magic numbers"
Multi-drone control	Supported	Not supported
Safety check	Both boundary and collision check, highly adjustable	Limited and non-adjustable, only 4 by 4 metre boundary check
Simulator	Supported, for multiple drones, play speed adjustable	Supported, only for one drone, fewer functions

Table 6.1: Comparison between *xDrone* and its earlier version.

6.3 *xDrone* v.s. Python

This section compares *xDrone* with Python. In this comparison, Python with the third-party `TelloEduSwarmSearch` library (mentioned in section 2.2.2), and pure Python without the library will be compared to *xDrone*.
The following is an example that controls two drones to draw squares simultaneously in *xDrone*,

```
procedure draw_square(drone my_drone, decimal side_length) {
  repeat 4 times {
    my_drone.forward(side_length);
    my_drone.rotate_right(90);
  }
}
main() {
  DRONE1.takeoff();
  DRONE2.takeoff();
  {draw_square(DRONE1, 1.0);} || {draw_square(DRONE2, 1.0);};
  DRONE1.land();
  DRONE2.land();
}
```

The following is the same code in Python with the `TelloEduSwarmSearch` library (`fly_tello`),

```
from fly_tello import FlyTello

my_tellos = ["tello-id-1", "tello-id-2"]
with FlyTello(my_tellos) as fly:
    def draw_square(drone_index, side_length):
        for _ in range(4):
            fly.up(side_length * 100, drone_index)
            fly.rotate_cw(90, drone_index)

    with fly.sync_these():
        fly.takeoff(tello=1)
    with fly.sync_these():
        fly.takeoff(tello=2)
    with fly.sync_these():
        draw_square(1, side_length)
        draw_square(2, side_length)
    with fly.sync_these():
        fly.land(tello=1)
    with fly.sync_these():
        fly.land(tello=2)
```

The code in pure Python is not shown here, as it is too complex, and network communication and multi-threading are involved.

The code of multi-drone control in *xDrone* and Python with the library are very similar. In fact, the *xDrone* code will be converted into the code in Python with the library by the

compiler. *xDrone* hides some complexity of importing the library and the `with` blocks. The readability of *xDrone* is slightly better, but the readability of Python really depends on the library's design and implementation.
However, *xDrone* surpasses Python as it provides safety checks and simulation, which are not supported in the Python program and cannot be done very easily.
Also, since *xDrone* is versatile, the same program can be run on different drones with different API's. Programmers in Python have to find a new library and read the documentation if they want to control a new drone with a different set of API; while in *xDrone*, programmers have no extra learning cost to use another drone, if this drone's API is supported by *xDrone*.
The following table provides a summary of the comparison.

	xDrone	Python with library	Pure Python without library
Lines of code	Few (13 lines in example above)	Few (20 lines in example above)	Many (not measured)
Readability	High	Medium, depends on the library	Medium, if encapsulated well
Difficulty	Easy	Easy, depends on the library	Very difficult, as network communication and multi-threading involved
Versatility	Same code for all drones supported	Different code, new library needed	Different code, new implementation needed
Multi-drone control	Supported	Supported, depends on the library	Supported, but multi-threading involved
Safety check	Supported	Not supported	Not supported
Simulator	Supported	Not supported	Not supported

Table 6.2: Comparison between *xDrone* and Python.

6.4 Drone Deviation

6.4.1 Deviation on Simulated Drones

As mentioned in section 4.3, the FPS, frames per second, is adjustable in the simulator. The position and orientation of drones are changed accumulatively in each frame at a fixed rate (the speed/rotate speed); if the FPS is too low, the granularity between two continuous time point samples is large, and drones will be likely to overshoot.
For example, if the rotation speed is 100 degrees per second, and we want the drone to rotate 90 degrees. If the FPS is 15, then at each frame, the orientation is updated by 6.67 degrees; but if the FPS is 30, the orientation is updated by 3.33 degrees. The following table shows how the orientation is updated.

	0s		1/15s		2/15s	...		13/15s		14/15s	
15FPS	0		6.67		13.33	...		86.67		93.33	Stopped
30FPS	0	3.33	6.67	10.00	13.33	...	83.33	86.67	90.00	Stopped	Stopped

Table 6.3: Rotate degree samples under different FPS values.

We can notice when FPS is 15, the orientation finally overshot the target value 90, which causes deviation on simulated drones. However, this is easily avoidable by selecting a higher FPS value.

6.4.2 Deviation on Real Drones

Real drones are not as reliable as simulated ones. There will be some deviations in real flights. More expensive drones have higher accuracy, but for the relatively cheap drones I am using, DJI Tello EDU, it is worth measuring how much deviation it will make during moving and rotation.
Experiments are done on my Tello EDU drone. The results are presented below. Each column is the result of one experiment.
When moving forward for 100 centimetres,

Deviation in distance (cm)	$+20$	$+25$	$+20$	$+11$	$+23$
Deviation in orientation (degree)	-1	$+4$	$+6$	$+7$	$+3$

Table 6.4: Deviation when moving 100 cm.

When rotating clockwise for 360 degrees,

Deviation in distance (cm)	$+10$ LF	$+7$ RF	$+6$ RF	$+5$ RF	$+7$ RF
Deviation in orientation (degree)	-8	-4	-1	-6	-2

Table 6.5: Deviation when rotating 360 degrees. (LF: left-front, RF: right-front)

We can notice that my drone has some systematic error. When moving, it will overshoot around 20%, with some undesired rotation; when rotating, it will undershoot around 0.5% with some undesired moving.
Systematic error is comparatively easier to remove - a larger number of experiments can be done to find a factor that can cancel out the error.
The random error is more difficult to handle. However, with the stochastic collision check, *xDrone* can calculate the probability of collisions based on the variances of deviation. This at least warns drone operators when a collision caused by deviation is highly likely, so that they can decide whether to take the risk.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this project, we have implemented *xDrone*, a powerful and sophisticated language for drone operating. Elegant programs can be written in *xDrone* to operate multiple drones simultaneously.
xDrone has common language features, like variables, branches, loops, functions, etc., but is simpler, more straightforward, and more sophisticated in drone controlling than other programming languages, with or without drone-controlling libraries. It is easy to learn and understand for drone operators who have less programming experience.
xDrone has also been designed to be versatile, which means it will not be bound with a specific drone brand and serious. Supports for new drones can easily be added and extended by the developers of *xDrone*, so that users of *xDrone* can control different drones with no extra learning cost of reading corresponding library documents.
xDrone also provides simulation and safety checks, which visualises the flight and guarantees flight safety. Programmers can specify the size of their flying space, *xDrone* will automatically check whether the flight is within this safe space and check whether there are any possible collisions between drones if multiple drones are operated.
The collision checker also considers the deviation of drones. Reports about collision probabilities can be generated to help the programmer analyse the risk of the flight stochastically in the real world.

7.2 Future Work

xDrone can be further improved to be more powerful in drone controlling, more developer friendly. Some future work is therefore proposed.

7.2.1 Make *xDrone* Interactive

One of the principles of *xDrone* limits the interaction between drones and the environment. This principle makes *xDrone* relatively easier to implement, and able to meet basic drone-use requirements, and protects the drone from potentially running out of battery. However, it is not necessarily a sufficiently good way, as drones can also be applied to fields like filming and searching, which require environmental interaction.
xDrone currently does not support programs with undetermined behaviours, or potential infinite loops, e.g. to land if the camera captures a certain label, or to fly up until the

height is 5 meters, or to make movements depending on the console input. These loops are conditioned on inputs from outside the drone, and cannot be parsed statically into a list of `DroneCommands`.
A rough thought of handling this is to make the immediate code, the list of `DroneCommands`, support loops and conditions. In addition, some input reading functions, like taking a picture and retrieving the console input, should be encapsulated and supported in the immediate code as well. The implementation is not trivial, so it is left as future work.

7.2.2 Support for Other Drones

Though *xDrone* has provided a relatively convenient way for supporting different drones, only DJI Tello EDU drones are supported due to time and equipment limit. However, this can be done very easily in the future, by adding new command executors for new drones. These executors should convert the intermediate code, `DroneCommand`, into corresponding drone API calls.

7.2.3 Built-in Math Library, More Operators

Currently, *xDrone* supports only necessary math operators for drone flight, e.g. basic arithmetic, but more can be implemented to make *xDrone* more programmer-friendly. For example, some trigonometric functions, power, modulo, floor and ceiling, random, etc.
In addition, though types like `string` or `vector` exist in *xDrone*, there are very few operators on them. New operators or built-in functions can be added to make use of these types.

7.2.4 Language Server, Editor Support

In order to make *xDrone* more friendly towards its developers, some editors support can be added in the future. This involves functions like syntax highlight, auto complete, navigate to definition, red underlines on error, documentation on hover, etc. This can be done by creating a language server, using the Language Server Protocol [23].

Chapter 8

Ethical Issues

As this project is related to drone using, it has the potential for military use and affects current standards in military ethics. Similarly, this project also has the potential to be malevolently abused. Drones can be used to deliver dangerous and harmful content, and can cause privacy issues if they have a camera.
According to the drone law [24] in the UK, a drone weighing between 250g and 20kg needs to be registered as an operator. DJI Tello EDU Drone's weight, according to the user manual, is 87g so the registration is not required. Its light weight as well as limitations on flight time, distance and height (max flight time: 13min, max flight distance: 100m, max flight height: 30m [15]) have limited the possibility of abusing and misusing. However, we cannot rule out the possibility of modification on the project and using other larger and more powerful drones for military or malevolent purposes.

Safety is another ethical issue to concern. Flying drones are dangerous to people and can cause physical injuries if they are not handled correctly and safely. This issue has been considered, as a restriction on the flying area is applied and set in the code configuration. The programmer can also review the code and run it in the simulator before starting a drone flight.

Bibliography

[1] Wikipedia contributors. Amazon Prime Air. https://en.wikipedia.org/w/index.php?title=Amazon_Prime_Air&oldid=1023449274, May 2021. Accessed: 2021-6-8. pages 5

[2] Wikipedia contributors. Drone display. https://en.wikipedia.org/w/index.php?title=Drone_display&oldid=1025824432, May 2021. Accessed: 2021-6-8. pages 5

[3] xTextLanguage. https://github.com/ProgrammableDrones/xTextLanguage, May 2017. Accessed: 2021-1-13. pages 5, 6

[4] Muhamad Gafar. xDrone Programming Language: Object Recognition and DSL development. January 2019. pages 5, 7

[5] L Niepolski. Simulation, route planning and environment modelling in xDrone a domain specific programming language for drones. September 2019. pages 5, 8

[6] M Baugh, N Bhat, K Gu, G Ordish, G Soteriou, and M Teguiani. Drone Playground. January 2020. pages 5, 10

[7] Drone programming: Learn to program with drones. https://www.plcgurus.net/drone-programming/, October 2020. Accessed: 2021-6-8. pages 12

[8] Dave Walker. TelloEduSwarmSearch. pages 12, 29

[9] Martin Fowler. *Domain-Specific Languages*. Pearson Education, 2011. pages 13

[10] A V Aho, Monica S Lam, R Sethi, and Jeffrey D Ullman. *Compilers: Pearson new international edition: Principles, techniques, and tools*. Pearson Education, London, England, 2 edition, 2013. pages 14, 15, 16

[11] Abstract vs. Concrete syntax trees. https://eli.thegreenplace.net/2009/02/16/abstract-vs-concrete-syntax-trees, February 2009. Accessed: 2021-1-13. pages 15

[12] About The ANTLR Parser Generator. https://www.antlr.org/about.html. Accessed: 2021-6-1. pages 16

[13] Tello EDU. https://www.ryzerobotics.com/tello-edu, n.d.. Accessed: 2021-6-1. pages 17

[14] Tello SDK 2.0 User Guide. https://dl-cdn.ryzerobotics.com/downloads/Tello/Tello%20SDK%202.0%20User%20Guide.pdf, n.d.. Accessed: 2021-6-1. pages 17, 18

[15] Tello User Manual V1.4 EN.pdf. https://dl-cdn.ryzerobotics.com/downloads/Tello/Tello%20User%20Manual%20v1.4.pdf, September 2018. Accessed: 2021-6-1. pages 17, 44

[16] Marc Peter Deisenroth, A Aldo Faisal, and Cheng Soon Ong. *Mathematics for Machine Learning*. Cambridge University Press, Cambridge, England, 2020. pages 18

[17] Allan Gut. *An Intermediate Course in Probability*. Springer, New York, NY, 2 edition, 2014. pages 18

[18] M A Chaudhry and Asghar Qadir. Incomplete exponential and hypergeometric functions with applications to the non central χ2-distribution. *Commun. Stat. Theory Methods*, 34(3):525–535, 2005. pages 19

[19] A. Nuttall. Some integrals involving the Q_M function (Corresp.). *IEEE Transactions on Information Theory*, 21(1):95–96, 1975. doi: 10.1109/TIT.1975.1055327. pages 19

[20] zlib. https://zlib.net/. Accessed: 2021-6-8. pages 28

[21] Welcome to click — click documentation (8.0.X). https://click.palletsprojects.com/en/8.0.x/. Accessed: 2021-6-8. pages 29

[22] PyPI The Python Package Index. https://pypi.org/. Accessed: 2021-6-8. pages 30

[23] Official page for Language Server Protocol. https://microsoft.github.io/language-server-protocol/. Accessed: 2021-6-8. pages 43

[24] Drone and model aircraft registration. https://www.caa.co.uk/Consumers/Unmanned-aircraft/Our-role/Drone-and-model-aircraft-registration/, November 2019. Accessed: 2021-1-13. pages 44

Appendix A

xDrone Language Specification

A.1 Configurations

Before you start to code in *xDrone*, you need to create a JSON file with some configurations. This JSON file tells the compiler and the simulator some basic information about your drones and your environment. A configuration file is required when *xDrone* code is compiled and/or executed.
The following is a sample configuration JSON file:

```
{
  "drones": [{
    "name": "DRONE1",
    "init_position": {"x": 0, "y": 0, "z": 0},
    "speed_mps": 2,
    "rotate_speed_dps": 180,
    "takeoff_height_meters": 2,
    "advanced": {
      "variance_per_meter": 0.002
    }
  },{
    "name": "DRONE2",
    "init_position": {"x": 1, "y": 0, "z": 0},
    "speed_mps": 1,
    "rotate_speed_dps": 90,
    "takeoff_height_meters": 1,
    "advanced": {
      "variance_per_meter": 0.001
    }
  }],
  "boundary_config": {
    "max_x_meters": 10,
    "max_y_meters": 20,
    "max_z_meters": 30,
    "min_x_meters": -10,
    "min_y_meters": -20,
    "min_z_meters": 0,
    "max_seconds": 100
  },
  "collision_config": {
    "collision_meters": 0.3,
    "time_interval_seconds": 0.1,
    "confidence_threshold": 0.95
```

```
    }
  }
```

All attributes except `"advanced"` are highly recommended to be filled. Although missing attributes will be filled with some default values, but they will affect the accuracy of the safety checks.

A.1.1 Drone Configuration

This is the `drones` JSON list in the configuration JSON. It specifies what drones you have, and some metrics of your drones.
`"name"` is the name of the drone, so when you are controlling multiple drones, you can specify what drone you are sending commands to. These names are highly encouraged to be capital, as they will be used as literals/constants in the program.
`"init_position"` is the initial position of the drone. You should give the coordinates of the drone in 3 axes, x, y, and z. The unit used is in meters. The x-axis has the direction from left to right, so negative means left and positive means right. The y axis has the direction from back to front, so negative mean back and positive means front. Similarly, the z axis has the direction from down to up, so negative means down and positive means up.
`"speed_mps"` is the speed of the drone in meters per second.
`"rotate_speed_dps"` is the speed of the drone during rotating in degrees per second.
`"takeoff_height_meters"` is the height in meters the drone will reach when it takes off. `"advanced"` contains come optional data for some advanced usages.
`"variance_per_meter"` is the variance of the drone deviation, when moving 1 meter in any direction. This is used in the advanced collision check where the probabilities of collision will be reported.

A.1.2 Boundary Configuration

This is the `boundary_config` JSON object in the configuration JSON. It specifies the boundaries of your environment, as well as the upper limit of the flight time.
The 'safe region' is the cuboid limited by attributes `"max_x_meters"`, `"max_y_meters"`, `"max_z_meters"`, `"min_x_meters"`, `"min_y_meters"`, and `"min_z_meters"`. The x, y, z coordinates are the same as what has been discussed before, and the unit used is also in meters. The safety checker will complain if any one drone will fly out of this region.
`"max_seconds"` puts an upper limit on the flight time. This can be used to prevent the drone from running out of battery.

A.1.3 Collision Configuration

This is the `collision_config` JSON object in the configuration JSON. It specifies some parameters used during the collision check.
`"collision_meters"` is the minimal safe distance between two drones in meters, if the distance between two drones is closer than this distance, a collision will happen. Its value should be the slighter larger the diameter of your drones.
`"time_interval_seconds"` is the time interval between two collision detection checkpoints in seconds. This should be set to a reasonable value regarding the speed and flying time of your drones. Otherwise, error may occur in the collision detector.
`"confidence_threshold"` is the threshold of the collision confidence. In the advanced safety check, when the confidence, or probability, of a collision at a time point is greater than this threshold, the compiler will report this collision; otherwise, this collision will be ignored. Its value should be between 0 and 1, and should be set to a reasonable value when `"variance_per_meter"` in drone configuration is set.

A.2 Term Explanations

A ***type*** is an attribute of data which in semantics tells how the data is intended to be used, e.g. as an integer, as a string, as a boolean, etc.
An ***identifier*** is a symbolic name, so that you can distinguish your variables, functions, procedures, etc.
A ***literal*** is an atomic notation, that represents fixed piece of data in source code, e.g. `"Hello world"` is a string literal in *xDrone*, it represents the string 'Hello world'.
A ***variable*** is a storage location with an associated identifier. It stores a piece of data of a given type. The piece of data stored can be accessed and reassigned.
A ***drone literal*** is a literal that represents a drone. They will be automatically injected to the program from the `"name"` of your drone you specified in `"drones"` in the configuration JSON file.
A ***value*** is the data described by a literal, the content of a variable, or the evaluation result of an expression.
An ***operator*** is used to associate one or more input values of specified types, this combination is called an ***expression***, which will finally be evaluated as a value of a specified type.
A ***statement*** is a piece of code that expresses some actions to be done.
A ***function*** is a sequence of statements. It that can take some input values, and use them to perform a specific task, and finally return a value to the caller. A function call is regarded as an expression in *xDrone*, which will be evaluated to the returned value.
A ***procedure*** is a sequence of statements. It that can take some input values, and use them to perform a specific task, but it does not return a value. A procedure is regarded as a statement in *xDrone*.
Arguments are the input values provided by the caller when calling a function or a procedure.
Parameters are variables defined during the definition of a function or a procedure. They are used to receive and store arguments received from a function/procedure call.

A.3 The Program

The program in *xDrone* starts with zero or more function or procedure definitions, then the main program body:

```
<function_or_procedure_definitions>
main() {
  <statements>
}
```

where `<function_or_procedure_definitions>` contains zero or more function or procedure definitions (discussed in A.8), and `main()` is the main entry point of the program, `<statements>` inside `main()` contains zero or more statements, which form the main body of the program.

A.4 Variables

There are 7 types of variables in *xDrone*, namely `int`, `decimal`, `string`, `boolean`, `vector`, `drone`, and `list`.
Variables in *xDrone* can be declared, assigned and deleted.

A.4.1 Variable Declarations

A variable can be declared so that the program will create a new variable with the type and the variable name you specified. This is done by the following command:

```
<type> <variable_name>;
```

where `<type>` is a variable type, and `<variable_name>` is the identifier of the variable name, e.g.

```
int my_variable;
```

where `int` is a variable type, and `my_variable` is the variable name.
You can use any variable name you like, as long as its first character is a letter or underscore (_), and other characters can be either letters, digits, or underscores (_), and it is different from keywords in *xDrone*, but meaningful variable names are highly recommended.
When a variable is declared, it will be assigned by a default value. The default values of different types are further discussed in A.4.4.
Note that you cannot declare variables with the same variable name more than once, even if they have different types; unless you have deleted the variable, see A.4.3 for details about variable deletions.

A.4.2 Variable Assignments

Value of a already declared variable can be updated via variable assignments using the arrow (`<-`) symbol. This is done by the following command:

```
<variable_name> <- <new_value>;
```

where `<variable_name>` is the identifier of the variable name, and `<new_value>` is a value with the same type as the type declared of the variable, e.g.

```
my_variable <- 10;
```

After the assignment, value of `my_variable` will be updated to `10`, and the old value will be overwritten.

The declaration and assignment to a single variable can be combined to one line:

```
<type> <variable_name> <- <new_value>;
```

e.g.

```
int my_variable <- 10;
```

A.4.3 Variable Deletions

A variable can be deleted, so that it will be removed, as it has never been declared. This is done by:

```
del <variable_name>;
```

where `<variable_name>` is the identifier of the variable name to be deleted, e.g.

```
del my_variable;
```

After the deletion, `my_variable` can no longer be used. You can declare the same variable name again if you want.

A.4.4 Variable Types

Int

Variable type `int` represents an integer, a sequence of number digits. When an `int` is declared, its default value is `0`. An `int` variable can be declared and assigned like the following.

```
int my_int;
my_int <- 1;
```

Decimal

Variable type `decimal` represents a decimal, or a float number, which is a sequence of number digits with a decimal point in between. Like many other languages, `decimal` numbers cannot be represented exactly due to binary fractions issues. When a `decimal` is declared, its default value is `0.0`. A `decimal` variable can be declared and assigned like the following.

```
decimal my_decimal;
my_decimal <- 0.1;
```

String

Variable type `string` represents a string. Strings are wrapped in two double quotation marks (`"`), and can have escaped characters in it. When a `string` is declared, its default value is `""` (empty string). A `string` variable can be declared and assigned like the following.

```
string my_string;
my_string <- "Hello world";
```

Boolean

Variable type `boolean` represents a boolean. It can be either `true` or `false`. When a `boolean` is declared, its default value is `false`. A `boolean` variable can be declared and assigned like the following.

```
boolean my_boolean;
my_boolean <- true;
```

Vector

Variable type `vector` represents a vector with 3 dimensions, namely `x`, `y`, and `z`. Dimensions are wrapped in a pair of round brackets (`(` and `)`), and separated by commas (`,`), e.g. `(1.0, 2.0, 3.0)`. Values in each of the 3 dimensions can have the type of `int` or `decimal`. If `int` is used, it will be converted automatically to `decimal`. When a `vector` is declared, its default value is `(0.0, 0.0, 0.0)`. A `vector` variable can be declared and assigned like the following.

```
vector my_vector;
my_vector <- (1, 2.0, -3);
```

Each dimension of the vector can be accessed by using `my_vector.x`, `my_vector.y`, or `my_vector.z`. These will return the value in the corresponding dimension in the type of `decimal`.
They can also be assigned separately, e.g.

```
my_vector.x <- 1;
my_vector.y <- 2.0;
my_vector.z <- -3;
```

Drone

Variable type `drone` represents a drone. It can be used to store a drone literal. When a `drone` is declared, its default value is `null`. A `drone` variable can be declared and assigned like the following (assuming `DRONE1` is a drone literal).

```
drone my_drone;
my_drone <- DRONE1;
```

The `drone` variables are used when calling movement commands, see A.5 for more information. Note that `drone` variables that have not been assigned will have the value `null`, and calling a movement command on `null` will result in a compiler error.

List

Variable type `list` represents a list or variables with the same type. These variables are wrapped in a pair of square brackets (`[` and `]`), and separated by commas (`,`), e.g. `[1, 2, 3]`, or `["a", "b", "c"]`. Variables in each of the entries should have the same type, i.e. the *entry type*. When a `list` is declared, its default value is `[]` (empty list). The entry type of the list can be any type, even another `list` type, and it should be specified inside a pair of square brackets (`[` and `]`) after the `list` keyword. A `list` variable can be declared and assigned like the following.

```
list[int] my_list;
my_list <- [1, 2, -3];

list[list[int]] nested_list;
nested_list <- [[0, 1], [2], [-3]];
```

The size of the list can be retrieved by calling `my_list.size`, this will return an `int` that is equal to the size of the list.

Entries inside a list have indices from 0 to list size - 1 inclusive.
A list entry can be accessed by calling `my_list[i]`, where `i` is an `int` that is equal the index of the entry you want to access, e.g. `my_list[0]`.
Also, a list entry can be updated by assigning the new value to `my_list[i]`, e.g.

```
my_list[0] <- 2;
```

Entries in a nest list can be accessed or updated similarly. Take the `nested_list` declared above as an example, `nested_list[0]` corresponds to the first entry in `nest_list`, which is `[0, 1]`; and `nested_list[0][0]` corresponds to the first entry in `nest_list[0]`, which is `0`.

A new entry can be inserted into the list to any reasonable position. The following command can be used to insert value `value` to index `i`, so all entries from index `i` inclusive will be shifted backwards. The range of `i` is from 0 to list size inclusive.

```
my_list.at(i).insert(value);
```

You can omit `.at(i)` if you want to append the `value` at the end of the list, i.e. `i` is equal to list size.

An entry can be removed from the the list. The following command can be used to remove the entry at index `i`, so all entries after index `i` will be shifted forwards. The range of `i` is from 0 to list size - 1 inclusive.

```
my_list.at(i).remove();
```

You can omit `.at(i)` if you want to remove the last entry, i.e. `i` is equal to list size - 1.

A.5 Movement Command Statements

xDrone supports basic movement commands that are compatible with most common drones. There are 11 of them in total. Their format is

```
<drone>.<command>(<parameters>);
```

The `<drone>` must have type `drone` or is a drone literal. If a `drone` variable is used, make sure it has been assigned, i.e. not `null`. Otherwise, an compiler error will be raised. You need to specify `<drone>` so that the program can know on which drone this action will be performed.
`<command>` and `<parameters>` are different for each movement command, and will be discussed in following subsections.
This is an example movement command:

```
DRONE1.forward(1);
```

If there is only one drone specified in the drone configuration, you can simply omit the drone name, e.g. calling

```
forward(1);
```

directly would be sufficient.

We will go through all 11 commands one by one. `DRONE1` will be used as a sample drone literal.

Takeoff

For a takeoff command, the `<command>` is `takeoff` and `<parameters>` is empty, e.g.

```
DRONE1.takeoff();
```

will take off `DRONE1` to the height specified in the drone configuration.
If the drone has already been taken off, an safety check error will be raised.

Land

For a land command, the `<command>` is `land` and `<parameters>` is empty, e.g.

```
DRONE1.land();
```

will land `DRONE1`, so that the drone will fly downwards until its height becomes 0.
If the drone has not been taken off, an safety check error will be raised.

Up

For a up command, the `<command>` is `up` and `<parameters>` is an `int` or `decimal` value, e.g.

```
DRONE1.up(i);
```

will fly `DRONE1` upwards for `i` meters, where `i` has type `int` or `decimal`.
If the drone has not been taken off, an safety check error will be raised.

Down

For a down command, the `<command>` is `down` and `<parameters>` is an `int` or `decimal` value, e.g.

```
DRONE1.down(i);
```

will fly `DRONE1` downwards for `i` meters, where `i` has type `int` or `decimal`.
If the drone has not been taken off, an safety check error will be raised.

Left

For a left command, the `<command>` is `left` and `<parameters>` is an `int` or `decimal` value, e.g.

```
DRONE1.left(i);
```

will fly `DRONE1` leftwards for `i` meters, where `i` has type `int` or `decimal`. Note that the orientation of the drone will not be changed.
If the drone has not been taken off, an safety check error will be raised.

Right

For a right command, the <command> is right and <parameters> is an int or decimal value, e.g.

```
DRONE1.right(i);
```

will fly DRONE1 rightwards for i meters, where i has type int or decimal. Note that the orientation of the drone will not be changed.
If the drone has not been taken off, an safety check error will be raised.

Forward

For a forward command, the <command> is forward and <parameters> is an int or decimal value, e.g.

```
DRONE1.forward(i);
```

will fly DRONE1 forwards for i meters, where i has type int or decimal.
If the drone has not been taken off, an safety check error will be raised.

Backward

For a backward command, the <command> is backward and <parameters> is an int or decimal value, e.g.

```
DRONE1.backward(i);
```

will fly DRONE1 backwards for i meters, where i has type int or decimal.
If the drone has not been taken off, an safety check error will be raised.

Rotate Left

For a rotate left command, the <command> is rotate_left and <parameters> is an int or decimal value, e.g.

```
DRONE1.rotate_left(i);
```

will rotate DRONE1 to the left (anticlockwise) for i degrees, where i has type int or decimal.
If the drone has not been taken off, an safety check error will be raised.

Rotate Right

For a rotate right command, the <command> is rotate_right and <parameters> is an int or decimal value, e.g.

```
DRONE1.rotate_right(i);
```

will rotate DRONE1 to the right (clockwise) for i degrees, where i has type int or decimal.
If the drone has not been taken off, an safety check error will be raised.

Wait

For a wait command, the <command> is wait and <parameters> is an int or decimal value, e.g.

```
DRONE1.wait(i);
```

will perform no actions on DRONE1 for i seconds, where i has type int or decimal.
This can be called no matter the drone has been taken off or not.

A.6 Operators

Operators are used to evaluate a value from one or more input values of given types.

A.6.1 Arithmetic Operators

Addition

`+` is a binary operation for addition.
Addition can be performed on 2 values with `int` and/or `decimal` type. The addition of 2 `int`s will give the sum of them in `int`, otherwise if `decimal` is involved the result type will be `decimal`. For example, `1 + 1` evaluates to `int` value `2`, while `1 + 1.0` evaluates to `decimal` value `2.0`.
Besides, addition can also be performed on 2 values with `vector` type, so that their corresponding dimensions will be added and result in a new `vector`. For example, `(1.0, 2.0, 3.0) + (4.0, 5.0, 6.0)` evaluates to `vector` value `(5.0, 7.0, 9.0)`.

Subtraction

`-` is a binary operation for subtraction.
Similar to addition, subtraction can be performed on 2 values with `int` and/or `decimal` type. The subtraction of 2 `int`s will give the difference of them in `int`, otherwise if `decimal` is involved the result type will be `decimal`. For example, `2 - 1` evaluates to `int` value `1`, while `2 - 1.0` evaluates to `decimal` value `1.0`.
Also similar to addition, subtraction can also be performed on 2 values with `vector` type, so that their corresponding dimensions will be subtracted and result in a new `vector`. For example, `(5.0, 7.0, 9.0) - (4.0, 5.0, 6.0)` evaluates to `vector` value `(1.0, 2.0, 3.0)`.

Multiplication

`*` is a binary operation for subtraction.
Multiplication can be performed on 2 values with `int` and/or `decimal` type. The Multiplication of 2 `int`s will give the product of them in `int`, otherwise if `decimal` is involved the result type will be `decimal`. For example, `2 * 3` evaluates to `int` value `6`, while `2 * 3.0` evaluates to `decimal` value `6.0`.
Multiplication can also be performed on a `vector` and a `int`, or on a `vector` and a `decimal`, so that each dimension of the `vector` will be multiplied by the `int` or `decimal` and result in a new `vector`. For example, `(1.0, 2.0, 3.0) * 2` evaluates to `vector` value `(2.0, 4.0, 6.0)`.

Division

`/` is a binary operation for division.
Division can be performed on 2 values with `int` and/or `decimal` type. The Division of 2 `int`s will give the quotient of them in `int`, otherwise if `decimal` is involved the result type will be `decimal`. For example, `6 / 3` evaluates to `int` value `2`, while `6 / 3.0` evaluates to `decimal` value `2.0`.
Note that if the result type of division is `int`, the quotient will be floored into an integer, but division the involves `decimal` will not floor the quotient, so is recommended. For example, `3 / 2` evaluates to `1`, but `3 / 2.0` evaluates to `1.5`.
Division by `0` will cause an math error.
Division can also be performed on a `vector` and a `int`, or on a `vector` and a `decimal`, but this time the order of them is important. The `vector` must be the dividend, and the `int` or `decimal` must be the divisor. Each dimension of the `vector` will be divided by the `int` or `decimal` and result in a new `vector`. For example, `(2.0, 4.0, 6.0) / 2` evaluates to `vector` value `(1.0, 2.0, 3.0)`.

Posit

- is also an unary operation. It can be applied to an `int` or a `decimal`. It does not change the sign of the number, but can be used to emphasize a number is positive. For example, `+ 1` evaluates to `1`.

Negate

- is also an unary operation. It can be applied to an `int` or a `decimal` to change the sign of the number. For example, `- 1` evaluates to `-1`.

A.6.2 Comparison Operators

Greater than

> is a binary operation for greater-than comparison.
Greater-than comparison can be performed on 2 `int` values or 2 `decimal` values, and will give the comparison result in `boolean`. If the left high side is greater than the right hand side, it will be evaluated to `true`; otherwise, it will be evaluated to `false`. For example, `2 > 1` evaluates to `true`, `1 > 1` evaluates to `false`, `0 > 1` evaluates to `false`.

Greater than or Equal to

>= is a binary operation for greater-than-or-equal-to comparison.
Greater-than-or-equal-to comparison can be performed on 2 `int` values or 2 `decimal` values, and will give the comparison result in `boolean`. If the left high side is greater than or equal to the right hand side, it will be evaluated to `true`; otherwise, it will be evaluated to `false`. For example, `2 >= 1` evaluates to `true`, `1 >= 1` evaluates to `true`, `0 > 1` evaluates to `false`.

Less than

< is a binary operation for less-than comparison.
Less-than comparison can be performed on 2 `int` values or 2 `decimal` values, and will give the comparison result in `boolean`. If the left high side is less than the right hand side, it will be evaluated to `true`; otherwise, it will be evaluated to `false`. For example, `2 < 1` evaluates to `false`, `1 < 1` evaluates to `false`, `0 < 1` evaluates to `true`.

Less than or Equal to

< is a binary operation for less-than-or-equal-to comparison.
Less-than-or-equal-to comparison can be performed on 2 `int` values or 2 `decimal` values, and will give the comparison result in `boolean`. If the left high side is less than or equal to the right hand side, it will be evaluated to `true`; otherwise, it will be evaluated to `false`. For example, `2 <= 1` evaluates to `false`, `1 <= 1` evaluates to `true`, `0 <= 1` evaluates to `true`.

Equal to

== is a binary operation for equal-to comparison.
Equal-to comparison can be performed on 2 values with the same type, and will give the comparison result in `boolean`. If its 2 sides evaluates to the same value, it will be evaluated to `true`; otherwise, it will be evaluated to `false`. For example, `1 == 1` evaluates to `true`, `2 == 1` evaluates to `false`.

Not Equal to

`=/=` is a binary operation for not-equal-to comparison.
Not-qual-to comparison can be performed on 2 values with the same type, and will give the comparison result in `boolean`. If its 2 sides evaluates to different values, it will be evaluated to `true`; otherwise, it will be evaluated to `false`. For example, `1 =/= 1` evaluates to `false`, `2 =/= 1` evaluates to `true`.

A.6.3 Logical Operators

Not

`not` is an unary operation for logical not.
Logical not can be performed on a `boolean` value, and will give the result after logical not operation in `boolean`. For example, `not true` evaluates to `false`.

And

`and` is a binary operation for logical and.
Logical and can be performed on 2 `boolean` values, and will give the result after logical and operation in `boolean`. For example, `true and false` evaluates to `false`.

Or

`or` is a binary operation for logical or.
Logical or can be performed on 2 `boolean` values, and will give the result after logical or operation in `boolean`. For example, `true or false` evaluates to `true`.

A.6.4 Other Operators

Concatenation

`&` is a binary operation for string concatenation.
Concatenation can be performed on 2 `string` values, and will give the result after string concatenation in `string`. For example, `"a" & "b"` evaluates to `"ab"`.

A.6.5 Parentheses

Expression can be wrapped with parentheses `(` and `)` in order to be evaluated first. For example, `2 * 1 + 1` will evaluate `2 * 1` first; but `2 * (1 + 1)` will evaluate `1 + 1` first.

A.6.6 Precedence

If there are multiple operators in an expression, they will be evaluated in a certain order. The operator with higher precedence will be evaluated first, and if the precedence is the same, the operator on the left will be evaluated first.
The following table lists the operator precedence in *xDrone*, upper groups have higher precedence than the lower ones.

`(...)`	parentheses
`+x, -x`	unary posit and negate
`not`	logical not
`*, /`	multiplication, division
`+, -`	addition, subtraction
`&`	concatenation
`>, >=, <, <=`	greater / less than
`==, =/=`	equality
`and`	logical and
`or`	logical or

Table A.1: Precedence in *xDrone*.

A.7 Flow Control Statements

xDrone has 4 kinds of flow control statements, namely `if-else`, `while`, `for`, and `repeat`.
All flow control statements create a new scope, variables created inside the scope will be deleted after the flow control statement finishes. However, updates on already existed variables will be kept.
Also, statements inside flow control statements are executed lazily, i.e. if there was a problematic statement that would raise a compile error, but was not executed due to flow control, the compiler will not complain.

A.7.1 If Else

The `if-else` statement in *xDrone* is

```
if <condition> {
  <if_statements>
} else {
  <else_statements>
}
```

where `<condition>` is a `boolean` value, `<if_statements>` and `<else_statements>` are zero or more drone commands and/or other statements, e.g.

```
int i <- 1;
if i < 1 {
  DRONE1.forward(i);
} else {
  DRONE1.forward(1);
}
```

If the `<condition>` evaluates to `true`, statements in `<if_statements>` will be executed, otherwise statements in `<else_statements>` will be executed.
The else branch can be omitted if `<else_statements>` is empty, like the following:

```
if <condition> {
  <if_statements>
}
```

A.7.2 While Loops

The `while` statement in *xDrone* is

```
while <condition> {
  <statements>
}
```

where <condition> is a boolean value, <statements> is zero or more drone commands and/or other statements, e.g.

```
int i <- 1;
while i < 5 {
  i <- i + 1;
}
```

If the <condition> evaluates to true, statements in <statements> will be executed, then the <condition> will be evaluated again, and <statements> will be executed again until <condition> evaluates to false.

A.7.3 For Loops

The for statement in *xDrone* is

```
for <variable_name> from <from_value> to <to_value> step <step_value> {
  <statements>
}
```

where <variable_name> should be a already declared int variable, <from_value>, <to_value>, and <step_value> are int values, <statements> is zero or more drone commands and/or other statements, e.g.

```
int i;
for i from 1 to 5 step 2 {
  DRONE1.forward(i);
}
```

The for statement will firstly assign <from_value> to <variable_name>, execute <statements>, then assign <from_value> + <step_value> to <variable_name>, execute <statements>, then assign <from_value> + <step_value> * 2... This will be repeated until the value to be assigned is strictly greater than <to_value>.
step <step_value> can be omitted if <step_value> is 1, like the following:

```
for <variable_name> from <from_value> to <to_value> {
  <statements>
}
```

A.7.4 Repeat Loops

The repeat statement in *xDrone* is

```
repeat <int_value> times {
  <statements>
}
```

where <int_value> is an int value, <statements> is zero or more drone commands and/or other statements, e.g.

```
repeat 4 times {
  DRONE1.forward(1);
}
```

The <statements> will be executed <int_value> times.

A.8 Functions and Procedures

Functions in *xDrone* take zero or more parameters and return a value in a certain type, while procedures in *xDrone* take zero or more parameters but do not return a value.
Functions and procedures should be defined before the main function in *xDrone*. If another function or procedure is called in a function/procedure, make sure the callee has been defined before the caller.
Both functions and procedures will create a new scope, and the parameters are passed by value; both new variables and updates on existing variables will be discarded, which is different from the scope created by flow control statements.
Like flow control statements, statements inside functions and procedures are executed lazily, i.e. if there was a problematic statement that would raise a compile error in a function/procedure, but this function/procedure was not called, the compiler will not complain.

A.8.1 Function Definitions

A function can be defined by

```
function <function_name>(<parameters>) return <return_type> {
  <statements>
}
```

where `<function_name>` is the identifier of the function name. The naming rule of function names is the same as variable names mentioned in A.4.1, and you are encouraged to avoid naming a variable and a function with the same name.
`<parameters>` is zero or more pairs of `type` and parameter name, separated by commas (`,`), e.g. `int i, float j, string s`.
`<return_type>` is a variable type, which is the type this function will be evaluated to.
`<statements>` is one or more drone commands and/or other statements. A `return` statement is compulsory, the `return` statement will be in the format of:

```
return <returned_value>;
```

where `<returned_value>` should have the same type as defined in `<return_type>`.
There can be multiple `return` statement in a function, and the execution of statements in the function will stop once the first `return` statement is executed, and the `<returned_value>` will be returned to the caller of the function.
The following is an example of a function:

```
function integer_addition(int i, int j) return int {
  return i + j;
}
```

A.8.2 Function Calls

The following shows how to call a function,

```
<function_name>(<arguments>)
```

where `<function_name>` is the identifier of the function name, and `<arguments>` is a list of values, separated by commas (`,`), which have types match the types declared in the parameters, e.g. `1, 1.0, "abc"`.
Note that there is no semicolon (`;`) after the function call, as the function call is not a statement, but will be evaluated as a value, and can be used in an expression.

A.8.3 Procedures Definitions

A procedure can be defined by

```
procedure <procedure_name>(<parameters>) {
  <statements>
}
```

where `<procedure_name>` is the identifier of the procedure name. The naming rule of procedure names is the same as function names mentioned before.
Like functions, `<parameters>` is zero or more pairs of `type` and parameter name, separated by commas (`,`), e.g. `int i, float j, string s`.
`<statements>` is one or more drone commands and/or other statements. Different from functions, `return` statements are optional. The `return` statement in procedure does not have a value to be returned with, like the following:

```
return;
```

There can be multiple `return` statement in a procedure, and the execution of statements in the procedure will stop once the first `return` statement is executed.
The following is an example of a procedure:

```
procedure fly_back_and_forth(drone d) {
  d.backward(1);
  d.forward(1);
}
```

A.8.4 Procedures Calls

The following shows how to call a procedure,

```
<procedure_name>(<arguments>);
```

where `<procedure_name>` is the identifier of the procedure name, and `<arguments>` is a list of values, separated by commas (`,`), which have types match the types declared in the parameters, e.g. `1, 1.0, "abc"`.
Note that, different from function calls, there is a semicolon (`;`) after the function call, as a procedure call is a statement.

A.9 Parallel Statements

When controlling multiple drones simultaneously, it is necessary to distinguish which commands are synchronous, which are asynchronous, as we sometimes want drone A to do action X **then** drone B to do action Y, and sometimes we want drone A to do action X **while** drone B to do action Y.
All statements in *xDrone* discussed before are synchronous, i.e. they will be executed one by one in sequence. To support asynchronous statements, parallel statement is introduced:

```
{<statements_1>} || {<statements_2>};
```

where `<statements_1>` and `<statements_2>` are two group of statements that will be executed in parallel. Each of them can contain one or more drone commands and/or other statements.
More group of statements can be added by appending `|| {<new_statements>}`, e.g.

```
{<statements_1>} || {<statements_2>} || {<statements_3>};
```

It is important to note that scopes will not be shared between statement groups, i.e. each group of statement inside curly brackets ({ and }) has its own scope. The scope is the same as the scope in functions and procedures, where both new variables and updates on existing variables will be discarded. The return command without a return value (`return;`) can also be used in the statement groups to exit early. In fact, these state groups can be regarded as anonymous procedures, whose parameters are all variables in the content.
Besides, drones involved in different statement groups should be disjoint, i.e. movements commands cannot be called on the same drone in different statement groups.
Also note that there is a semicolon (`;`) after the parallel statement, as it is a (nested) statement.
The following are some examples of parallel statements:

```
{DRONE1.takeoff();} || {DRONE2.takeoff();};
```

A.10 Sample Programs

The following is a program that takes off 3 drones one by one, and simultaneously let `DRONE1` draw a square with side length 1m then land, `DRONE2` draw a square with side length 2m then land, `DRONE3` land immediately.

```
procedure draw_square(drone my_drone, decimal side_length) {
  repeat 4 times {
    my_drone.forward(side_length);
    my_drone.rotate_left(90);
  }
}
main () {
  list[drone] drones <- [DRONE1, DRONE2, DRONE3];
  int i;
  for i from 0 to drones.size - 1 {
    drones[i].takeoff();
  }
  {
    draw_square(DRONE1, 1.0);
    DRONE1.land();
  } || {
    draw_square(DRONE2, 2.0);
    DRONE2.land();
  } || {
    DRONE3.land();
  };
}
```

Appendix B

Command Line Interface Specification

The *xDrone* terminal command line interface can be used to validate a *xDrone* program, simulate the program in the simulator, and run the program on real drones.

B.1 Validation

The following terminal command can be run to see whether there is a syntax error or semantic error in the code, the safety checks will also be performed, and any safety error will be reported.

```
xdrone --validate --code <code_path> --config <config_path>
```

A `--timeout <seconds>` parameter can be added, so that the compilation will be aborted if timeout. The default value is 10 seconds. This prevents the compiler from taking forever to convert infinite loops in the program to `DroneCommands`.
A `--save-report` flag can be added to save the outputs of the collision check, i.e. the probabilities of collision at each time slice, to the directory `./reports/`.

B.2 Simulation

With the simulator server installed, the following terminal command can be run to start a simulation.

```
xdrone --simulate --code <code_path> --config <config_path>
```

This command will firstly perform the validation, and if the program is valid (no syntax error, semantic error, and passed safety checks), it will print a link to the simulator in the terminal, and automatically open the web page to display the simulation.
A `--port <port_number>` parameter can be added if the simulator server is not on its default port, 8080.
A `--no-check` flag can be added if no safety checks are wanted, in order to see where collisions happen in the simulator.

B.3 Drone Flight

The following terminal command can be run to start the real drone flight. Before this, the terminal and the Tello EDU drones should be connected to the same WIFI.

```
xdrone --fly --code <code_path> --config <config_path>
```

This command will firstly perform the validation, and if the program is valid (no syntax error, semantic error, and passed safety checks), commands will be sent to real drones, and the flight shall start. Similarly, a `--no-check` flag can be added if no safety checks are wanted, but this is dangerous and a confirmation will be required.