Contents

Chapter 1

xDrone Language Specification

1.1 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.

1.1.1 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.

1.1.2 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.

1.1.3 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.

1.2 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.

1.3 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 1.8), 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.

1.4 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.

1.4.1 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 1.4.4.
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 1.4.3 for details about variable deletions.

1.4.2 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;
```

1.4.3 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.

1.4.4 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 1.5 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.

1.5 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.

1.6 Operators

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

1.6.1 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`.

1.6.2 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`.

1.6.3 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`.

1.6.4 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"`.

1.6.5 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.

1.6.6 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 1.1: Precedence in *xDrone*.

1.7 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.

1.7.1 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>
}
```

1.7.2 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.

1.7.3 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>
}
```

1.7.4 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.

1.8 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.

1.8.1 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 1.4.1, 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;
}
```

1.8.2 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.

1.8.3 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);
}
```

1.8.4 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.

1.9 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();};
```

1.10 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();
  };
}
```