Kotlin Interim Test

Imperial College London

Department of Computing

"Kotlin Interpreter"

Friday 14 March 2025
14:00 to 17:00
THREE HOURS
(including 10 minutes of suggested planning time)

- The maximum total is **100 marks**.
- Credit is awarded throughout for conciseness, clarity, *useful* commenting, and the appropriate use of the various language features.
- **Important:** Marks are deducted from solutions that do not compile in the test environment, which will be the same as the lab machines. Comment out any code that does not compile before you submit.
- After you have finished reading the spec, you can start coding by **running the IDEA shortcut on your Desktop**, which will open our provided .iml file. **Please do not attempt to open your project in any other way**, and please do not delete any of our files.
- The extracted files should be in your Home folder, under the "`kotlin-lexis-test`" subdirectory. **Do not move any files**, or the test engine will fail, resulting in a compilation penalty.
- The examples and test cases here are not guaranteed to exercise all aspects of your code. You are therefore advised to define your own tests to complement the ones provided.
- When you are finished, simply **save everything and log out**. **Do not shut down your machine**. We will fetch your files from the local storage.
- If your IDE fails to build your code, you can still compile via the terminal using the provided `./compile-all` perl script. You can then test your code via `./test-all`.

Problem Description

Your task is to write a set of classes to represent programs in a simple programming language, and to build an interpreter for sequential and concurrent programs expressed in this language.

Full details of the features of the language that you will support are provided in the questions below, but here is a brief "big picture" overview.

Programs in the simple language work with integer variables, with names such as `x`, `index`, `cat`, etc. Variables do not need to be declared. When a program executes, it starts in an initial state where certain named variables are already available with given initial values.

Integer expressions can be built from variable names, integer literals, arithmetic operators and parentheses. An example of an integer expression is `x * (y + 2)`.

Basic boolean expressions arise from applying comparison operators to integer expressions, so that e.g. `(x + 2) > 5` is a basic boolean expression. Compound boolean expressions can be built from other boolean expressions using logical operators and parentheses. An example of a compound boolean expression is `!(x * y < 5 && y > x)`.

A program is made from a sequence of statements, where a statement is either an *assignment* statement, a *conditional* (`if`) statement, or a *loop* (`while`) statement.

An assignment statement sets a named variable to the result of an integer expression. For example, `z = x * (y + 2)` is an assignment statement. During execution, if the named variable is already available with a given value, this value is overwritten by the assignment. Otherwise, the named variable becomes available, and has the value provided by the assignment.

A conditional statement comprises a condition, a "then" branch, and optionally an "else" branch. The condition is a boolean expression. The "then" branch is a non-empty sequence of statements, as is the "else" branch if present.

Here is a conditional statement with no "else" branch:

```
if (x > 0) {
    i = i + 1
    j = j + 2
}
```

Here is a conditional statement with an "else" branch that contains a nested conditional:

```
if (x > 0) {
    i = i + 1
    j = j + 2
} else {
    if (x > y) {
        x = 0
    } else {
        y = 0
    }
}
```

A loop statement comprises a condition and a body. The condition is a boolean expression. The body is a possibly empty sequence of statements. Here is an example loop statement:

```
while (i < 100) {
  i = i + 1
  j = j - 1
}
```

A sequential program is a sequence of statements. In a concurrent program, every thread has its own sequence of statements.

Getting Started

The skeleton files are located under the `src` directory.

Under `src` there are `main` and `test` sub-directories. The code you write to implement functionality should go under `main`, while any test code should go under `test`.

Under `main` there are `kotlin` and `java` sub-directories, and code for the respective language should be located under the corresponding sub-directory. Within each of the `kotlin` and `java` sub-directories there is a directory called `proglang`. All functional code you write for this exercise will belong to the `proglang` package, and thus should reside in one of these `proglang` directories.

Under `test` there is only a `kotlin` sub-directory. This is because you are not expected to write tests in Java and all the provided tests are in Kotlin. Under this sub-directory is a `proglang` sub-directory. All provided tests plus any new tests that you create should be located here, in the `proglang` package.

You are free to add additional methods and classes, beyond those specified in the instructions, as you see fit, for example to follow good object-oriented principles, or for testing. Any new files should be placed in the `proglang` package for the appropriate programming language.

Testing

There is a test class, `Question`*i*`Tests`, for each question i. These contain initially commented-out tests to help you gauge your progress. **As you progress through the exercise you should un-comment the test class associated with each question in order to test your work.** In some cases you will be required to add additional tests to these classes as part of your task. Furthermore, you are welcome to add tests to these classes to help you debug your solution.

These tests are not exhaustive and are merely intended to guide you. Your solution should pass all of the given tests. However, passing all of the given tests does not guarantee that your solution is fully correct, and says nothing about your coding style and the appropriateness of your use of Kotlin and Java features. You should thus think carefully about whether your solution is complete, and pay attention to coding style and practices, even if you pass all of the given tests.

What to do

1. **Integer expressions**

   Study the provided file, `IntExpr.kt`. This file defines an interface, `IntExpr`, and one class that implements this interface, `Add`. The `Add` class is nested inside the `IntExpr` interface so that its full name is `IntExpr.Add`.

An `Add` expression comprises a left-hand-side and right-hand-side, both of which are `IntExpr`s.

The file also defines an extension method `eval` on `IntExpr`. This takes a `store` parameter, a map from variable names (strings) to values (integers). Its purpose is to provide the values for any variables used in the expression. When invoked on an `IntExpr`, the `eval` extension method should return the integer obtained by evaluating the expression, and should consult the store to obtain values of any variables that occur in the expression.

The given implementation of `eval` handles `Add` expressions by recursively evaluating their left- and right-hand-sides, and returning the sum of these results.

Add further nested classes to the `IntExpr` interface, and complete the implementation of `eval`, so that the following kinds of integer expression are also supported. Refer to `Question1Tests` for requirements on how you should name the properties of these classes, all of which should be public and read-only:

- `Literal`: Represents a literal expression. Constructed from an integer value. When evaluated, yields this integer value.
- `Var`: Represents an expression that refers to a variable. Constructed from a string name. When evaluated, returns the value associated with this name in the given store. Throws an `UndefinedBehaviourException` (see provided exception class) if the store does not provide a value for the name.
- `Mul`: Similar to `Add`, but represents the product of two expressions.
- `Sub`: Similar to `Add`, but represents the difference between two expressions.
- `Div`: Similar to `Add`, but represents the quotient of one expression by another, using Kotlin's integer division operator. If the right-hand-side evaluates to zero, an `UndefinedBehaviourException` should be thrown.
- `Fact`: Represents a factorial expression. Constructed from a target expression. On evaluation, the target expression is evaluated. If the result is negative, an `UndefinedBehaviourException` should be thrown. Otherwise, the factorial of the result should be returned. You should not use any Kotlin standard library functions when implementing evaluation of this expression. For simplicity, you may assume (and do not need to check) that the result of evaluating a `Fact` expression does not overflow.
- `Paren`: Represents an integer expression in parentheses. Constructed from a target expression. Evaluates to the same result as that to which the target expression evaluates.

Next, override `toString` in each of the classes that implement `IntExpr` to provide a string representation for each kind of expression. Use the examples under "Overview of the simple language" above, and the test cases in `Question1Tests`, to guide you in how expressions should be represented as strings.

Finally, adapt the `IntExpr` interface so that the `else` branch can be removed from the `when` expression in the `eval` extension method.

Test your solution using (at least) the tests in `Question1Tests`.

[10 marks]

2. **Boolean expressions**

   In a new file, `BoolExpr.kt`, write an interface `BoolExpr` to represent boolean expressions.

   Similar to the approach taken in `IntExpr.kt`, declare a series of nested classes inside `BoolExpr`, each of which should implement the `BoolExpr` interface, to support the kinds of boolean expressions described below. In the same file, write an extension method on `BoolExpr`, `eval`, that takes a `store` parameter (similar to the `eval` extension method on `IntExpr`) and returns the boolean value arising from evaluating the expression with respect to the given store.

   The following kinds of boolean expressions should be supported. Refer to `Question2Tests` for requirements on how you should name the properties of these classes, all of which should be public and read-only:

   - `LessThan`: Represents the ordered comparison of two integer expressions. Constructed from a left-hand-side and right-hand-side, both `IntExprs`. When evaluated, returns true if and only if the result of evaluating the left-hand-side integer expression is less than the result of evaluating the right-hand-side integer expression.
   - `GreaterThan`: Similar to `LessThan`, but when evaluated returns true if and only if the evaluated left-hand-side is greater than the evaluated right-hand-side.
   - `Equals`: Similar to `LessThan`, but when evaluated returns true if and only if the evaluated left-hand-side is equal to the evaluated right-hand-side.
   - `And`: Represents a "logical and" expression. Constructed from a left-hand-side and right-hand-side, both `BoolExprs`. Evaluates to true if and only if both the left- and right-hand-sides evaluate to true. Evaluation should use the "short-circuit" semantics that are also used by Kotlin.
   - `Or`: Similar to `And`, but represents a "logical or" expression. Again, evaluation should use the "short-circuit" semantics that are also used by Kotlin.
   - `Not`: Represents a "logical not" expression. Constructed from a target `BoolExpr`. Evaluates to true if and only if the target expression evaluates to false.
   - `Paren`: Represents a boolean expression in parentheses. Constructed from a target expression. Evaluates to the same result as that to which the target expression evaluates.

   Next, override `toString` in each of the classes that implement `BoolExpr` to provide a string representation for each kind of expression. Use the examples under "Overview of the simple language" above, and the test cases in `Question2Tests`, to guide you in how expressions should be represented as strings.

   Test your solution using (at least) the tests in `Question2Tests`.

   [5 marks]

3. **Assignment and conditional statements**

   In a new file, `Stmt.kt`, write an interface, `Stmt`, to represent statements of a program. Every `Stmt` object will act both as a program statement, and as the head of a list of program statements. This will be achieved by equipping each statement with a possibly-null successor.

   Your interface should have the following members:

   - A mutable property, `next`, representing the possibly null successor of the statement.

- Read-access to a property, `lastInSequence`. A default implementation of `get` should be provided for this property, which should return the last statement in the list of statements headed by the receiving `Stmt`.
- An abstract method, `toString`, that takes an integer parameter, `indent`. This will be used to turn nested statements into strings, using indentation to ensure that the resulting strings are readable.

Assignment statements Write a class, `Assign`, that implements the `Stmt` interface and is declared as a nested class inside this interface. This class should represent an assignment statement, and should have the following public properties, provided to its constructor in this order:

- `name`, the name of the variable to be assigned (a read-only string);
- `expr`, representing the right-hand-side of the assignment (a read-only `IntExpr`);
- a possibly-null `Stmt`, overriding the mutable `next` property of the `Stmt` interface. When not null, this represents the statement following the assignment statement. The parameter should default to null if it is omitted.

Implement the version of `toString` required by the `Stmt` interface, which takes an `indent` parameter. This should construct a string representing the sequence of statements headed by the receiving `Assign` statement, as follows:

- The string representation should start with `indent` spaces, followed by the name of the variable being assigned to, an '`=`' symbol with one space on either side, the string representation of the integer expression on the right-hand-side of the assignment, and a new line character.
- If the `Assign` statement has a successor, the string representation should be followed by the string representation of the successor statement, obtained using the same indentation level.

Override `toString` from `Any` so that it provides a string representation of an `Assign` statement with no indentation.

Conditional statements Write a class, `If`, that implements the `Stmt` interface and is declared as a nested class inside this interface. This class should represent a conditional statement, and should have the following public properties, provided to its constructor in this order:

- `condition`, the condition of the statement (a read-only `BoolExpr`);
- `thenStmt`, the "then" branch of the statement (a read-only `Stmt`);
- `elseStmt`, the optional "else" branch of the statement (a read-only possibly-null `Stmt`), which should default to null if the parameter is omitted;
- a possibly-null `Stmt`, overriding the mutable `next` property of the `Stmt` interface. When not null, this represents the statement following the conditional statement. The parameter should default to null if it is omitted.

Implement the version of `toString` required by the `Stmt` interface, which takes an `indent` parameter. This should construct a string representing the sequence of statements headed by the receiving `If` statement. The statements in this sequence should all be indented using `indent` spaces.

The "then" branch of the conditional statement, and the "else" branch if it is present, should be indented by `indent` + 4 spaces. However, you should use a constant so that it would be easy to change the indentation level of 4 to some different value.

Refer to the examples under "Overview of the simple language" above, and the test cases in `Question3Tests`, to guide you in how conditional statement should be represented as suitably-indented strings.

Override `toString` from `Any` so that it provides a string representation of an `If` statement with no indentation.

Test your solution using (at least) the tests in `Question3Tests`.

[15 marks]

4. **Stepping through assignment and conditional statements**

You are now ready to start implementing an interpreter for statements, which will form the basis of an interpreter for sequential programs.

In `Stmt.kt`, write an extension method, `step`, for the `Stmt` interface. This method should take a `store` parameter—a `MutableMap` from strings to integers—and should return a possibly-null `Stmt`.

Applying `step` to an assignment statement When applied to an `Assign` statement, the `step` extension method should:

- Evaluate the right-hand-side of the assignment statement in the context of `store` to obtain an integer value.
- Update `store` so that it maps the name on the left-hand-side of the assignment statement to this value.
- Return the successor of the assignment statement (or null if there is no successor).

If evaluating the right-hand-side expression leads to an `UndefinedBehaviourException`, this should simply be propagated by `step`.

Applying `step` to a conditional statement When applied to an `If` statement, the `step` extension method should:

- Evaluate the condition associated with the conditional statement in the context of `store` to obtain a a boolean value.
- If the boolean value is true, return a `Stmt` comprising the sequence of statements associated with the "then" branch of the conditional, followed by the `Stmt` that follows the conditional statement (if not null).
- If the boolean value is false and an "else" branch is present, return a `Stmt` comprising the sequence of statements associated with the "else" branch of the conditional, followed by the `Stmt` that follows the conditional statement (if not null).
- If the boolean value is false and no "else" branch is present, return the successor of the conditional statement (or null if there is no successor).

If evaluating the condition leads to an `UndefinedBehaviourException`, this should simply be propagated by `step`.

Note that `step` does not modify the given store when applied to an `If` statement.

Test your solution using (at least) the tests in `Question4Tests`.

[20 marks]

5. **Interpreting sequential programs**

   Note: this question must be implemented in Java.

   In the provided Java file, `SequentialProgram.java`, complete the provided skeleton class to meet the following requirements:

   - It should be possible to access `SequentialProgram` from anywhere in the project.
   - It should not be possible to create sub-classes of `SequentialProgram`.
   - A `SequentialProgram` should have a field of type `Stmt`, representing the top-level statement of the program. A value for this field should be provided on construction. The field should be visible only within the `SequentialProgram` class, and it should not be possible to change the value of the field.
   - A `SequentialProgram` should have a public method, `execute`, that should take an `initialStore` parameter: a `java.util.Map` from strings to integers.

     The `execute` method should make a copy of the initial store, to obtain a "working store" that will be mutated as the program executes. It should then repeatedly call the `step` extension method of `Stmt`, starting by calling it on the top level statement of the program. If `step` returns null, this indicates that execution of the program has completed, in which case the working store should be returned by `execute`. Otherwise, if `step` returns a non-null `Stmt`, `execute` should invoke `step` on this statement, and so on.

   To invoke the `step` extension method of `Stmt` from the `SequentialProgram` Java class you need to use the syntax `StmtKt.step`, and then pass the receiving `Stmt` as the first argument to this method, followed by the store argument that `step` requires.

   Importantly, the `execute` method should not modify the initial store with which it is provided.

   Override `toString` from Java's `Object` class so that the string representation of a `SequentialProgram` is the string representation of its top level statement.

   Test your solution using (at least) the tests in `Question5Tests`.

   [10 marks]

6. **Loop statements**

   So far you have dealt with loop-free programs. Your task now is to introduce loop statements.

   Representing loop statements and turning them into strings In the `Stmt.kt` file, add a nested class, `While`, to the `Stmt` interface, where `While` implements `Stmt`. This class should represent a loop statement, and should have the following public properties, provided to its constructor in this order:

   - `condition`, the condition of the loop (a `BoolExpr`);
   - `body`, the body of the loop (a possibly-null `Stmt`);

- a possibly-null `Stmt`, overriding the mutable `next` property of the `Stmt` interface. When not null, this represents the statement following the loop statement. The parameter should default to null if it is omitted.

Implement the version of `toString` required by the `Stmt` interface, which takes an `indent` parameter. This should construct a string representing the sequence of statements headed by the receiving `While` statement. The statements in this sequence should all be indented using `indent` spaces.

The body of the loop, if it is present, should be indented similarly to the "then" and "else" branches of conditional statements. Refer to the examples under "Overview of the simple language" above, and the test cases in `Question6Tests`, to guide you in how loop statement should be represented as suitably-indented strings.

Override `toString` from `Any` so that it provides a string representation of an `While` statement with no indentation.

Supporting loop statements in `step` Supporting loop statements in `step` is more complex than was the case for assignment and conditional statements. This is because when the condition of a loop with a body evaluates to true, the remainder of the program to be executed comprises:

- The loop body;
- The entire loop statement (because its condition may remain true after the body has been executed);
- The statements that follow the loop.

The complication here is that the loop body is part of the loop and so needs to be duplicated.

To support this, add an abstract method `clone` to the `Stmt` interface, that takes no arguments and returns a `Stmt`.

Implement `clone` in the `Assign`, `If` and `While` classes to meet the following requirements:

- When invoked on an `Assign` statement, `clone` should return a new `Assign` statement with the same variable name and expression as the original statement (i.e., these should not be cloned), and a clone of the `next` statement (if it is not null).
- When invoked on an `If` statement, `clone` should return a new `If` statement with the same condition expression as the original statement (i.e., this should not be cloned), a clone of the "then" branch, a clone of the "else" branch (if it is not null), and a clone of the `next` statement (if it is not null).
- When invoked on a `While` statement, `clone` should return a new `While` statement with the same condition expression as the original statement (i.e., this should not be cloned), a clone of the body (if it is not null), and a clone of the `next` statement (if it is not null).

With `clone` in place, it is time to add support for loop statements in the `step` extension method. When applied to a `While` statement, the `step` extension method should:

- Evaluate the condition associated with the loop statement in the context of `store` to obtain a a boolean value.

- If the boolean value is true and the loop has a body, return a `Stmt` comprising: a clone of the sequence of statements comprising loop body, followed by the original loop statement.
- If the boolean value is true and the loop does not have a body, return the original statement.
- If the boolean value is false, return the successor of the loop statement (or null if there is no successor).

Note that `step` does not modify the given store when applied to an `While` statement.

Test your solution using (at least) the tests in `Question6Tests`.

[20 marks]

7. **Concurrent programs**

Your final task is to write some concurrent code to simulate the execution of *concurrent* programs expressed in the simple programming language.

A concurrent program will be executed by multiple threads, and each thread will be provided with a `Stmt`—the top-level statement that it should execute. This will be similar to how a `SequentialProgram` works, but for simplicity you should not try to make use of the code that you wrote in `SequentialProgram`.

Each thread will also have a *pause value*, indicating a period of time for which it should pause before taking a step. All threads should share a lock, and a thread should acquire this lock before taking a step and release it after taking a step.

In a new file, `ProgramExecutor.kt`, write a class, `ProgramExecutor`, that is suitable for being executed by a Java `Thread`. A `ProgramExecutor` instance should have access to:

- A *thread body*: the `Stmt` in our simple programming language that should be executed.
- A lock, that will be shared between all threads, to ensure mutual exclusion.
- A pause value of type `Long`, indicating the number of milliseconds for which a thread should sleep before taking a step.
- A *store*—a `MutableMap` from strings to integers—that will be shared between all threads and which records values of the variables used by the program.

When a `ProgramExecutor` runs, it should use `step` to repeatedly step through program statements, similar to the case for a `SequentialProgram`. Before each step, the `ProgramExecutor` should sleep for a number of milliseconds, according to the pause value. The `ProgramExecutor` should acquire the lock directly before taking a step and release it directly after taking a step. The store that was passed to the `ProgramExecutor` on construction should be directly updated during this process, i.e. a copy of the store should not be made.

Finally, in a new file, `ConcurrentProgram.kt`, write a class, `ConcurrentProgram`.

A `ConcurrentProgram` should be constructed with a list of `Stmt`s, representing *thread bodies*, and a list of `Long`s, representing the pause values for threads. If the sizes of these lists are not the same, an `IllegalArgumentException` should be thrown. A `ConcurrentProgram` should also have a lock as a property.

Add an `execute` method to `ConcurrentProgram`, which takes a `Map` from strings to integers as a parameter (the *initial store*), and returns a `Map` from strings to integers (the *final store*). The `execute` should method should work as follows:

- Make a copy of the initial store (the *working store*).
- Launch N threads, where N is the size of the thread body list. Each thread should run a `ProgramExecutor` to execute one of the thread bodies with the corresponding pause value, operating on the working store and using the `ConcurrentProgram`'s lock for mutual exclusion.
- Once all threads have terminated, return the working store.

If any thread fails by throwing an `UndefinedBehaviourException`, the `execute` method should propagate this exception.

Test your solution using (at least) the tests in `Question7Tests`.

[20 marks]

Total: 100 marks