Table 1. Mapping grammar terminals to methods in BDK's API that users can use to compose block tests.

Terminal	API Methods	Description
blocktest, **lambdatest**	`blocktest(), blocktest(String name), lambdatest(), lambdatest(String name)`	Declares a block test with an optional name.
start	`start(EndAt value), start(EndAt value, boolean endAfter), start(EndAt value, int at), start(EndAt value, int at, boolean endAfter)`	Specifies the start of a target fragment by traversing the code backwards based on the `EndAt` construct, with optional arguments to more precisely specify the starting point.
end	`end(EndAt value), end(EndAt value, boolean endAfter), end(EndAt value, int at), end(EndAt value, int at, boolean endAfter)`	Specifies the end of a target fragment by traversing the code forwards based on the `EndAt` construct, with optional arguments to more precisely specify the ending point.
given	`given(Object variable, Object value), given(Object variable, Object value, Object type)`	Specifies inputs for a block test as a variable-value pair, with an optional type argument.
setup	`setup(Runnable setupFunction), setup(String setupFunction)`	Specifies additional initialization for an input variable in a block test using a `Runnable` or its `String` representation. That `Runnable` will be executed before the block test is run.
mock	`mock(Object call, Object... values)`	Replaces method `call` with `values` as its return values, instead of invoking `call`.
args	`args(Object... value)`	Specifies arguments for a lambda application.
checkDataFlow	`checkEq(Object actual, Object expected), checkEq(Object actual, Object expected, Object delta), checkFalse(Object value), checkReturnEq(Object expected), checkReturnEq(Object expected, Object delta), checkReturnFalse(), checkReturnTrue(), checkTrue(Object value)`	Specifies a data flow oracle using the value of a block test's output variable, or return values.
checkControlFlow	`checkFlow(Flow... value)`	Specifies a control flow oracle: the target fragment should exercise the sequence of control flow steps captured in a `Flow` object.
expect	`expect(Exception value)`	Specifies that the target fragment should throw an exception.

1 BDK's API

Table 1 shows BDK's API that users can use to compose block tests. Many of these methods have been intuitively described via examples, or in the description of our grammar. So, we focus here on methods that require further explanation or on method parameters that we did not describe before.

It can be seen from the first row in Table 1 that the `blocktest()` and `lambdatest()` methods allow users to name each block test. Doing so can be helpful for developers during debugging after a block test fails. If users do not name their block tests, then BDK automatically assigns a default name based on the location where the block test is declared.

The `start()` and `end()` methods can be used to specify the beginning and end, respectively, of a target fragment, depending on the location of the block test relative to the fragment. If the block test is placed immediately after location e in the target fragment, then e marks the end of the fragment. In this case, a user can use `start()` to specify how far back before e the target fragment extends in the enclosing method. If no `start()` call is provided, then BDK treats the start of the basic block containing *e* as the beginning of the fragment. On the other hand, if the block test is placed immediately before location *b* in the target fragment, then *b* marks the beginning of the fragment. In this case, user can use `end()` to specify how much further after *b* the target fragment extends in the enclosing method. If no `end()` call is provided, BDK finds the end of the target fragment as the last location in the enclosing method where an output variable in a data flow oracle is assigned, or it defaults to the end of the basic block containing the block test.

The `EndAt` `enum` in BDK's API specifies how far forwards (when using `end()`) or backwards (when using `start()`) a target fragment extends. The current `EndAt` values that BDK supports are as follows. (i) `FIRST_ASSIGN_BLOCK` - first or last block with assignments to all input variables for `start()` or `end()`, respectively. (ii) `FIRST_BLOCK` - first or last marker (*e.g.*, opening or closing brace) of a Java block for `start()` or `end()`, respectively. (iii) `FIRST_RETURN` - the preceding or next `return` statement before or after the block test for `start()` or `end()`, respectively. (iv) `FIRST_STATEMENT` - the preceding or next Java statement before or after the block test for `start()` or `end()`, respectively. (v) `FIRST_THROW` - the preceding or next `throw` statement before or after the block test for `start()` or `end()`, respectively. (vi) `LAMBDA` - the beginning or end of a lambda for `start()` or `end()`, respectively. (vii) `LAST_REFERENCE` - the first or last reference to (*i.e.*, a read of) an input variable before or after the block test for `start()` or `end()`, respectively. Clearly, this set of supported `EndAt` kinds will grow as block testing matures and becomes more popular. But, the current set was sufficient to write all the 1,012 block tests in this paper. So, we think that they are sufficient to prove our concept.

Using the versions of `start()` and `end()` that only take an `EndAt` parameter restricts the scope of target fragments to one boundary (assignment, `throw` statement, etc.) before or after the block test. To relax this restriction, BDK allows using two other parameters separately or in combination: an integer `at` and a Boolean `endAfter`. Setting `at` to *n* means that the target fragment should extend to the *n*th `EndAt` boundary. For example, `end(FIRST_RETURN, 3)` means that the end of the target fragment is the third `return` statement after the block test. When `at` is not specified, it has a default value of 1. Setting `endAfter` to `true` means that the code at the `EndAt` boundary. The default value is `true`. For example, `end(FIRST_RETURN, 3, false)` means that the expression in the third `return` statement is *not* included in the target fragment. But, `end(FIRST_RETURN, 3, true)` will include the expression in that third `return` statement in the target fragment.

Using the `given(Object variable, Object value)` variant to assign values to input variables works only when BDK can infer the type of `variable`. In cases where BDK cannot infer the type of `variable`, *e.g.*, because the variable is not declared in the same class as the target fragment, then the user can also use the variant of `given` that takes an additional type parameter. Our current implementation allows specifying that type as a `String`.

Several methods that implement data flow oracles in BDK allow specifying that the `actual` value should be equal to the `expected` value within a error tolerance value of `delta`. Doing so allows inexact matches, *e.g.*, when comparing floating point numbers. Since unit tests are well suited for validating the outputs of whole methods, the reader may wonder why BDK's API supports data flow oracles like `checkReturnEq` that validate values returned from target fragments. The reason for this design choice is that, if unit test struggle to reach a fragment containing a `return` statement, then such oracles allow validating what the method would return if reached. Such data flow oracles are particularly useful when fragments that unit tests do not cover contain multiple `return` statements.

The parameter type for BDK's sole method for specifying control flow oracles is an `enum`, `Flow` that provides a way to more systematically express program paths that a block test should exercise. In theory, the values of `Flow` should allow specifying the expected branch of each expression containing a control flow decision along a path (*e.g.*, ternary expressions, `goto`, conditions in statements like `while`, `do-while`, `if`, etc.). But, as a first step, our current BDK implementation currently supports only four intuitive `Flow` values that are related to `if` statements: (i) `IfStmt` - control reaches the condition in an `if` statement; (ii) `ElseIf` - control reaches the condition in an `else if` statement; (iii) `Then` - control reaches the `then` branch of an `if` statement; and (iv) `Else` - control reaches the `else` branch of an `if` statement. So, far we did not encounter any target fragment that required more than these four `Flow` values. When we do, we will add values for supporting them to `Flow`.