*A Compiler for the Verification of a C-Like Programming Language: User Manual*

EECS 4302 – Compilers and Interpreters

Koko Nanah Ji Ege Cakmak

215168057 215173131

[koko96@my.yorku.ca](mailto:koko96@my.yorku.ca) [cakmake@my.yorku.ca](mailto:cakmake@my.yorku.ca)

EECS: koko96 EECS: cakmake

Lassonde School of Engineering

York University

Table of Contents

[1 Input Programming Language 3](#_Toc36488669)

[1.1 Overall Structure of a Program 3](#_Toc36488670)

[1.1.1 Boolean and Integer Primitive Types 5](#_Toc36488673)

[1.1.2 Variable Declaration and Assignment 5](#_Toc36488674)

1.1.3 Basic Type Checking………………………………………………………………..…………………6

[1.1.4 Conditional Statements (if/else) with Contracts (require/ensure) 5](#_Toc36488673)

[1.1.5 Functions and Functions Calls (with Contracts) 5](#_Toc36488674)

1.1.6 Feature 3: Lists (w/o Indexing) and Quantification Operations over Lists (all/some)…………………6

1.1.7 Feature 3: Lists (w/o Indexing) and Quantification Operations over Lists (all/some)…………………6

[1.2 How a Specification is Written 4](#_Toc36488671)

[1.3 List of Advanced Programming Features 5](#_Toc36488672)

[1.3.1 Feature 1: Functions with Local Scopes and Function Calls (with Contracts) 5](#_Toc36488673)

[1.3.2 Feature 2: Non-Nested Loops (with Contracts) 5](#_Toc36488674)

1.3.3 Feature 3: Lists (w/o Indexing) and Quantification Operations over Lists (all/some)…………………6

[2 Output Specification Language 6](#_Toc36488675)

[2.1 General Structure of Output 3](#_Toc36488670)

[2.2 Output Feature Translations 3](#_Toc36488670)

[2.2.1 Conditional Statements (if/else) with Contracts (require/ensure) 5](#_Toc36488673)

[3 Examples 7](#_Toc36488676)

[3.1 If statement 7](#_Toc36488677)

[3.2 Double Nested If statement 7](#_Toc36488678)

[3.3 Triple Nested If statement 7](#_Toc36488679)

[3.4 multiple Functions 7](#_Toc36488680)

[3.5 Function with If statement inside 7](#_Toc36488681)

[3.6 Loop 7](#_Toc36488682)

[3.7 Loop 7](#_Toc36488683)

[3.8 List and quantification 7](#_Toc36488684)

[3.9 Function called from loop 7](#_Toc36488685)

[3.10 Chef’s Special 7](#_Toc36488686)

[3.11 Chef’s Special 7](#_Toc36488686)

[3.12 Chef’s Special 7](#_Toc36488686)

[3.13 Chef’s Special 7](#_Toc36488686)

[4 Miscellaneous Features 8](#_Toc36488687)

[5 Limitations 9](#_Toc36488688)

# Input Programming Language

## Overall Structure of a Program

The input programming language that will be translated into Alloy supports the below features.

* Boolean and Integer Primitive Types (with Operations)
* Variable Declaration and Assignment
* Basic Type Checking
* Conditional Statements (if/else) with Contracts (require/ensure)
* Functions with Local Scopes and Function Calls (with Contracts)
* Non-nested Loops with Contracts
* Lists (w/o Indexing) and Quantification Operations over Lists (all/some)

The syntax of the input programming language is like modern programming languages such as (C, Python etc.) except for couple of exceptions. The rest of this subsection describes and briefly explains each bullet point above.

### Boolean and Integer Primitive Types

The input programming language supports only Integer and Boolean primitive data types.

Keywords “int” and “bool” are reserved to enable users to declare their own variables.

Integer operations such as add, subtract, divide, multiply, modulo are supported as well as Boolean operations such as and, or, implies, equality, inequality and not.

|  |
| --- |
| int var1;  int var2 = 1;  int var3 = 0;  var1 = var1 + var 2; *// Addition Example*  bool var5; *// Uninitialized Boolean Variable Declaration*  bool var6 = true; *// Initialized Boolean Variable Declaration*  bool var 7 = false;  var 5 = var6 && true;*// Conjunction Example* |
|  |

Figure 1: Demonstration: Primitive Types

### Variable Declaration and Assignment

The input programming language supports declaration of variables and initializing their values using either a constant or value of another variable. It is also worth noting here that sequential assignments are supported as will be shown in the comments.

|  |
| --- |
| int var1; *// Uninitialized Integer Variable Declaration*  int var2 = 1; *// Initialized Integer Variable Declaration*  int var3 = 0;  int var4 = -1;  var1 = var3; *// Copying Variables*  bool var5; *// Uninitialized Boolean Variable Declaration*  bool var6 = true; *// Initialized Boolean Variable Declaration*  bool var 7 = false;  var5 = var6; *// Copying Values of Variables* |
|  |

Figure 2: Demonstration: Variable Declaration and Assignment

### Basic Type Checking

The input programming language also supports type checking of variables during translation. Uninitialized variables will also be detected for type checking therefore our compiler can warn the user in case they attempt to access uninitialized variables.

|  |
| --- |
| int var1;  int var2 = 1;  bool var5;  bool var6 = true;  var1 = var5; *// WILL FAIL*  var2 = var1; *// WILL FAIL*  var6 = var5; *// WILL FAIL* |
|  |

Figure 3: Demonstration: Type Checking

### Conditional Statements (if/else) with Contracts (require/ensure)

The input programming language also supports if/else statements with or without contracts. Nested if statements are allowed as well. (will be demonstrated in later sections)

|  |
| --- |
| int var1 = 1;  if (var1 == 1) {  var1 = 5;  } else {  var1 = 10;  }  bool var2 = false;  if\_require (var1 == 10) *// Ensures that var1 has value 10 right before the if statement*  if (var2 <=> false) {  var1 = 20;  }  if\_ensure (var1 == 20) *// Ensures that var1 has value 20 right after the if statement* |
|  |

Figure 4: Demonstration: Conditional Statements

### Functions and Function Calls (with Contracts)

Functions can be declared and called in the input programming language. Variables in the input language are passed by values to the functions. And, every function must have a return type but can have any number of parameters.

Currently, it is enforced by design that every function has a require and ensure section. Yet, these contracts are not mandatory to implement as they can be skipped by inserting just “true” in them.

A limitation that is worthwhile to mention here is that variable declaration inside functions are not supported at the moment. Thus, these variables need to be declared at the parameters section of the function, and also, while invoking the function a placeholder argument should be passed for those arguments.

Lastly, values of variables before the translation of the function can be retrieved by appending “\_old” to the names of the variables.

|  |
| --- |
| fun int fun1 (int var1) { *// Function Declaration with 1 Parameter*  fun\_require(var1 > 0) *// Ensures that var1 has a positive value right before the function call.*  var1 = var1;  return var1;  fun\_ensure(var1 == var1\_old) *// Ensures that var1 is unchanged right after the function call.*  }  int x = 10;  x = fun1(x); *// Calls function fun1.*  fun int fun2 (int var1, int tmp) { *// Function Declaration with 1 Actual and 1 Extra Parameter*  fun\_require(var1 > 0) *// Ensures that var1 has a positive value right before the function call.*  tmp = var1 + 1;  return tmp;  fun\_ensure(var1 == var1\_old) *// Ensures that var1 is unchanged right after the function call.*  }  int y = 10;  int z;  x = fun2(y,z); *// Calls function fun2. Note that this will fail since z is uninitialized yet.* |
|  |

Figure 5: Demonstration: Functions

### Non-nested Loops with Contracts

The input programming language supports loops for statements that repeat. The syntax of loops in the input programming language is a bit different than the ordinary programming languages. Contracts are supported in loops and users are required to implement the invariant, variant, and initialization sections separately along with the implementation body.

|  |
| --- |
| int sum;  int current;  loop\_require(true) // Precondition need to be implemented.  loop\_init{ // Initialization need to be implemented.  sum = 0;  current = 3;  }  loop(current != 0) { // Exit condition goes here  loop\_invariant((sum == (3-current)) && (current >= 0) && (current < 4))  loop\_variant(current) // Loop variant and invariant need to be implemented as well.  sum = sum + 1; // Implementation body starts here.  current = current - 1;  }  loop\_ensure(sum == 3) Postcondition need to be implemented. |
|  |

Figure 6: Demonstration: Loops

### Lists (w/o Indexing) and Quantification Operations over Lists (all/some)

The input programming language also supports lists of primitive types.

Currently, each list must have a primitive type and type of each element in lists must comply with its list’s primitive type. The current state of lists only supports “add” and “remove” operations. However, quantification operations over lists are supported and these operators can be used as demonstrated below. The “each” keyword is reserved for quantification operations and can only be used in inside “all” / “some” like below.

|  |
| --- |
| int[] list1; *// Creates an array of integers.*  list1.add(1); *// Adds integers to the list.*  list1.add(2);  list1.add(3);  list1.add(true); *// WILL FAIL.*  if\_require(list1.all(each > 0)) *// Ensures list only has positive numbers right before the if statement.*  if (list1.some(each == 1)) { *// Adds integer 4 to the list if integer 1 exists in the list.*  list1.add(4);  }  if\_ensure(list1.some(each == 1)) *// Ensures list has integer 1 right after the if statement.* |
|  |

Figure 7: Demonstration: Lists and Quantification Operators

## How a Specification is Written

The input programming language supports verification of all features mentioned in the previous section. Due to the declarative nature of Alloy though, our compiler only translates the statements with contracts as there is nothing to verify otherwise. For instance, there is nothing to verify when the only operation is a variable declaration.

Below keywords are reserved for the input programming language and cannot be used as a variable name.

* if, else, if\_require, if\_ensure
* loop, loop\_require, loop\_ensure, loop\_init, loop\_invariant, loop\_variant
* fun, fun\_require, fun\_ensure, return
* true, false, int, bool, int[], bool[]
* each, all, some
* int, bool, true, false

Below keywords are specifically used for specification. These keywords can only be used in their respective places as suggested by their names. (eg. if\_require can only be used for enforcing preconditions for if statements etc.)

* if\_require, if\_ensure (Inside if statements)
* loop\_require, loop\_ensure (Inside loops)
* invariant, variant (Inside loops)
* fun\_require, fun\_ensure (Inside functions)

All the above keywords for specification can be used to verify the correctness of the input programming language.

The specification of each specification construct can be either propositional or in predicate logic depending on user’s wish as our compiler makes a singleton data structure for all available variables within the scope and their respective values.

As demonstrated in section 1.1.5, the keyword “old” can be used to retrieve the pre-state of variables, if the implementation block introduces any changes to the value of the variables.

Since the only way variables’ values can change is through assignments, our compiler keeps record of changes every time it encounters an assignment. The compiler will also create new variables for the new values of the variables so that the original value of the variable can be preserved. Although this process is not explicitly visible to the user during the translation, these new variables will be available to the user in the output code. These new variables will store the post-states’ of variables, and their name will be the name of the pre-state followed by a single quote (‘). There are no restrictions on the number of post-states. (i.e. arg1’’’ is possible)

e.g. arg1 is the original name, arg1’ is the post-state

It is currently possible to write fairly complex specification constructs thanks to all the operators and features of the input programming language. We will not be giving any specific complex examples here as there are already many examples in the following sections and chapters.

## List of Advanced Programming Features

### Feature 1: Functions with Local Scopes and Function Calls (with Contracts)

The syntax of functions is much similar to modern programming languages as demonstrated in 1.1.5. We will not be reviewing the syntax as it has already been presented to reader’s pleasure earlier.

The current state of the compiler seems to be able to generate sound outputs. The Alloy Analyzer seems to find counterexamples when a logically correct program is tampered to be incorrect on purpose. Here are two code snippets with translations where one is correct and the other is incorrect.

|  |
| --- |
| fun int add5(int x){  fun\_require(true)  x = x + 5; // Adds 5 to x as the function name suggests  return x;  fun\_ensure(x == (x\_old + 5)) // Contract is valid  }  int x = 10;  int xplus5 = add5(x);  bool result = (x + 5) == xplus5; |
|  |

Figure 8: Function Example Input Programming Language (Logically Correct)

|  |
| --- |
| open logicFuncs  pred predFunction0 [arg1,arg1':Int] {  ((True) in True) and arg1'= arg1.add[5] // corresponds to the assignment at line 3  ((((arg1' = arg1.add[5]) => True else False)) in True) // post condition  }  fun funFunction0 [arg1,arg1':Int] : Int {  { return : Int | ((True) in True) and arg1' = arg1.add[5] and return = arg1' }  }  check assertFunction0 {  { all arg1:Int | some arg1':Int | ((True) in True) => predFunction0[arg1,arg1'] }  } |
|  |

Figure 9: Function Example Output Specification Language (Logically Correct)

|  |
| --- |
| fun int add5(int x){  fun\_require(true)  x = x - 5; // Mistakenly subtracts 5 instead of adding  return x;  fun\_ensure(x == (x\_old + 5)) // Contract is invalid  }  int x = 10;  int xplus5 = add5(x);  bool result = (x + 5) == xplus5; |
|  |

Figure 10: Function Example Input Programming Language (Logically Incorrect)

|  |
| --- |
| open logicFuncs  pred predFunction0 [arg1,arg1':Int] {  ((True) in True) and arg1'= arg1.sub[5]  ((((arg1' = arg1.add[5]) => True else False)) in True) // post condition  }  fun funFunction0 [arg1,arg1':Int] : Int {  { return : Int | ((True) in True) and arg1'=arg1.sub[5] and return = arg1' }  }  check assertFunction0 {  { all arg1:Int | some arg1':Int | ((True) in True) => predFunction0[arg1,arg1'] }  } |
|  |

Figure 11: Function Example Output Specification Language (Logically Incorrect)

We would like to briefly explain here how the input programming language is translated into the output specification language at a high-level, before we debate whether our transformation is semantics-preserving and correct.

We are going to be explaining examples in Figure 10 and 12 at once since the only difference in between is the operator at line 3.

Both snippets in Figure 10 and 12 aim to do the same thing, adding 5 to a given integer and returning it. However, the code snippet in Figure 12 mistakenly subtracts 5 instead of adding.

We will be examining the body of the check statement first and then examine the bodies of the predicate and the function.

In the output specification language (Figure 11 and 13), we can see 2 variables getting declared in the check body. “arg1” refers to the pre-state of variable “x” whereas “arg1’” refers to the post-state. This check body basically states that for all possible pre-states of “x”, there exists a post-state of “x” where the function predicate holds if the precondition is true. The left-hand side of the implication statement is our precondition (it was “true” in the input programming language) which is translated to ((True) in True) which basically means true in Alloy. In this case the statement in fact makes sense, it is saying that if the precondition holds then the predicate for the function should hold as well!

The predicate ***pred predFunction0*** is the translation of the function with contracts. It states that if the assignments are logically correct then the post condition must hold. We can see the assignment (x = x +5) being done in Alloy (line 4 of Figure 11 and 13) and we can also see the post condition (line 5 of Figure 11 and 13). Note that this predicate looks structurally intact.

When we try verifying both outputs (both correct and incorrect) we can clearly observe the correct one passing and the incorrect one failing with a counterexample found by the Alloy Analyzer.

The function ***fun funFunction0*** is generated to be used by Alloy for subsequent function calls to this function from other Alloy predicates.

We believe that our transformation is semantics-preserving since we can clearly explain the transformation in a way that makes sense. Also, the fact that Alloy Analyzer is able to find a counterexample when we deliberately break our model allows us to convince ourselves that the translation is valid thus it is semantics-preserving.

Finally, we should note here that the input language does not support variable declaration inside functions. It is a requirement that all required variable declarations are done before the function declaration.

### Feature 2: Non-Nested Loops with Contracts

The syntax of loops in our input language is a bit different than standard loop syntax, as can be observed at Figure 5. We will not be explicitly explaining the syntax of loops here as it has been already explained in section 1.1.6, but we can note here to remind the reader that our input language syntax allows user to implement the variant, invariant, initial step and body of the loop separately.

Currently we believe that our compiler can translate a given loop in our input language to valid and accurate Alloy Analyzer code. Conversely, when we tamper to make the input logically incorrect, our code still compiles but Alloy Analyzer is unable to find any counter examples suggesting issues in the output specification code. However, we will not be fixing it since fixing/learning semantics of Alloy is beyond the scope of this project.

We will be attempting to strengthen this hypothesis in the rest of this subsection. Here are two not so simple looking code snippets that aim to add all numbers from 4 to 1.

|  |
| --- |
| int sum;  int current;  loop\_require(true)  loop\_init{  sum = 0;  current = 4;  }  loop(current < 5) {  loop\_invariant((current >= 0) && (current < 5) )  loop\_variant(current)  sum = sum + current;  current = current - 1;  }  loop\_ensure(sum == 10) |
|  |

Figure 12: Loop Example Input Programming Language (Logically Correct)

|  |
| --- |
| open logicFuncs  pred predForStatement0 [arg2, arg2’, arg2'', arg2''':Int, arg1, arg1', arg1'':Int] {  (((True) in True) and arg1'= 0 and arg2'' = 4 and arg2' = 4  =>  (andGate[((arg2'' >= 0) => True else False), ((arg2'' < 5) => True else False)] in True))  ((andGate[((arg2 >= 0) => True else False), ((arg2 < 5) => True else False)] in True)  and(((arg2 < 5) => True else False) in True ))  =>  ((True) in True) and arg1'' = arg1.add[arg2] and arg2''' = arg2.sub[1]  =>  (andGate[((arg2''' >= 0) => True else False), ((arg2''' < 5) => True else False)] in True and (arg2''' >= 0) and (arg2 > arg2'''))  ((andGate[((arg2 >= 0) => True else False), ((arg2 < 5) => True else False)] in True) and not((((arg2 < 5) => True else False) in True))  =>  ((((arg1 = 10) => True else False)) in True)) // post condition  }  check assertForStatement0 {  { all arg2:Int,arg1:Int | some arg2':Int,arg2'':Int,arg2''':Int,arg1':Int,arg1'':Int | ((True) in True) => predForStatement0[arg2,arg2',arg2'',arg2''',arg1,arg1',arg1''] }  } |
|  |

Figure 13: Loop Example Output Specification Language (Logically Correct)

|  |
| --- |
| int sum;  int current;  loop\_require(true)  loop\_init{  sum = 0;  current = 4;  }  loop(current < 5) {  loop\_invariant((current >= 0) && (current < 5))  loop\_variant(current)  sum = sum - current;  current = current - 1;  }  loop\_ensure(sum == 10) |
|  |

Figure 14: Loop Example Input Programming Language (Logically Incorrect)

|  |
| --- |
| open logicFuncs  pred predForStatement0 [arg2,arg2',arg2'':Int,arg1,arg1',arg1'':Int] {  (((True) in True) and arg1'= 0 and arg2'= 4  =>  (andGate[((arg2' >= 0) => True else False), ((arg2' < 5) => True else False)] in True))  ((andGate[((arg2 >= 0) => True else False), ((arg2 < 5) => True else False)] in True)  and(((arg2 < 5) => True else False) in True))  =>  ((True) in True) and arg2'' = arg2.sub[1] and arg1'' = arg1.sub[arg2]  =>  (andGate[((arg2'' >= 0) => True else False), ((arg2'' < 5) => True else False)] in True and (arg2'' >= 0) and (arg2 > arg2''))  ((andGate[((arg2 >= 0) => True else False), ((arg2 < 5) => True else False)] in True) and not((((arg2 < 5) => True else False) in True))  =>  ((((arg1 = 10) => True else False)) in True)) // post condition  }  check assertForStatement0 {  { all arg2:Int,arg1:Int | some arg2':Int,arg2'':Int,arg1':Int,arg1'':Int | ((True) in True) => predForStatement0[arg2,arg2',arg2'',arg1,arg1',arg1''] }  } |
|  |

Figure 15: Loop Example Output Specification Language (Logically Incorrect)

We note here that the only difference between the two snippets is the operator at line 10.

Luckily, with both samples (Figure 14 and 16) our compiler can generate code that can be compiled by the Alloy Analyzer. Unfortunately, as mentioned earlier that Alloy Analyzer fails to find a counterexample for logically the incorrect code (Figure 16). This might mean there is a semantic issue in the generated output, however we believe that the generated code is syntactically and semantically correct based on the Alloy specification. This might even be a bug, considering the number of inconsistencies we have come across in Alloy, we will talk about some of these in the “Limitations” chapter.

In the check body of the generated code we can see two variables and their post states getting declared for the loop predicate with no precondition. At high-level, this loop predicate checks that each of the below are TRUE. (This information was taken from Jackie’s 3311 Slides.)

* Establishment of Loop Invariant
* Maintenance of Loop Invariant
* Establishment of Postcondition upon Termination
* Loop Variant Stays Non-Negative Before Exit
* Loop Variant Keeps Decrementing Before Exit

We leave examining the details of the output code to the reader. Upon our examinations we believe that the predicate should be valid thus the output should be valid as well. Ultimately, the conversion should be semantics-preserving, ignoring the issue that we believe that exists in Alloy as we just talked about.

### Feature 3: Lists (w/o Indexing) and Quantification Operations over Lists (all/some)

The primitive syntax of lists in our input language is straight forward as has been demonstrated already in section 1.1.7.

Currently, we believe that our compiler supports declaration of lists and addition / removal operations over them as well as quantification operations such as “all” / ”some” for batch checking the correctness of a Boolean expression for each item over a list.

Since it is extremely easy to cause a lot of possible states with lists (state explosion), it is on the very contrary, extremely difficult to verify programs that cause state explosions.

As a result, we do not support adding to lists inside any block with contracts. Although we can still generate code that compiles, we just do not know whether it is correct or not simply because we cannot verify.

We are also aware of an issue that using existential operator (some) might cause a counterexample to be found. For instance, output generated from below code causes Alloy to find a counter example, which does not make sense since we can easily see from Figure 18 that the postcondition is correct. This might suggest that there is a semantic error in the way we generate output. However, we will not be fixing it due to timing constraints. Yet, this could also be an issue with Alloy Analyzer itself!

|  |
| --- |
| int[] x;  x.add(1);  x.add(2);  x.add(3);  int y;  if\_require(x.all(each > 0))  if(true) {  y = 1;  }  if\_ensure(x.some(each > 0)) |
|  |

Figure 16: Lists Example of an Issue Causing a Counterexample

Because of the state explosion issue and the issue regarding the existential operator we believe that we cannot do any extensive tests for this feature now. Therefore, we will not be doing any tests with correct and incorrect code snippets for this feature.

Here’s a very simple working code snippet that we will use to summarize the translation of quantification operations at a high level.

|  |
| --- |
| int[] x;  x.add(1);  x.add(2);  x.add(3);  int y = 15;  if\_require(x.all(each > 0))  if(true) {  y = 4;  }  if\_ensure(x.all(each > 0)) |
|  |

Figure 17: Working Very Simple Input Programming Language

|  |
| --- |
| open logicFuncs  pred predIfStatement0 [arg1:seq Int,arg2,arg2':Int] {  ((True) in True) =>  ((True) in True) and arg2'=4  ((((all arrayElems: arg1.elems | arrayElems in {each: Int | (((each > 0) => True else False) in True)})=> True else False)) in True) // post condition  }  check assertIfStatement0 {  { all arg1:seq Int,arg2:Int | some arg2':Int | ((((all arrayElems: arg1.elems | arrayElems in {each: Int | (((each > 0) => True else False) in True)})=> True else False)) in True) => predIfStatement0[arg1,arg2,arg2'] }  } |
|  |

Figure 18: Working Very Simple Output Specification Language

|  |
| --- |
| x.all(each > 0) |
|  |

Figure 19: Quantification Operation Input Programming Language

|  |
| --- |
| all arrayElems: arg1.elems | arrayElems in {each: Int | (((each > 0) => True else False) in True)})=> True else False)) in True) |
|  |

Figure 20: Quantification Operation Specification Language

As you can see from Figure 17 and 18, translation is rather simple. Our code creates a satisfying set by looking at the statement inside the quantification operation and then in the translation it checks whether each item in the list is in the satisfying set or not.

Despite the issue with the existential operator and the state explosion, we believe that our reasoning makes sense and translation of quantification operators should be semantics-preserving as demonstrated in Figure 19 and 20.

# Output Specification Language

In this chapter we will first briefly talk about the general structure of the output and then we will demonstrate how some of the features from 1.1 are translated.

## General Structure of Output

Let’s refer to the very simple example we have in Figure 20.

In every translation the first line is always “open logicFuncs”. This is a statement that tells Alloy Analyzer to load our logical operation utilities library. One might wonder why we had to implement Booleans ourselves. The answer is that Alloy by default does not allow Boolean constants in the code. By implementing Booleans ourselves we are able to bypass this issue. This file (logicFuncs.als) can be found in our submission and should be placed in the same path as the Alloy code that is about to be verified.

In every translation, there is a check block and a predicate block. In the predicate block, we usually verify functions, loops, if statements or quantification operations followed by their postconditions. Variable value assignments take place in the predicate block as well.

In the check block we declare each variable in the input and the post states for the variables if necessary (i.e. if body of the statement has assignments). Also using these variables, we specify the preconditions and refer to the predicate we just mentioned to verify the model.

## Output Feature Translations

Due to the declarative nature of Alloy, our compiler only translates the statements with contracts as there is nothing to verify in trivial statements such as declaring a variable. In this section we will be demonstrating the output our compiler is able to generate.

Since we have already demonstrated sample output for Functions, Loops and Lists we will not be demonstrating those ones again and kindly ask the reader to refer to section 1.3. Therefore, we will now be demonstrating output for if statements, which is the only thing left we must demonstrate.

### Conditional Statements (if/else) with Contracts (require/ensure)

Figure 23 is the output generated by our compiler when the code snippet in Figure 3 is inputted.

|  |
| --- |
| open logicFuncs  pred predIfStatement0 [arg1:Bool,arg2,arg2':Int] {  ((((arg1 in False) => True else False)) in True) =>  ((True) in True) and arg2'=20  ((((arg2' = 20) => True else False)) in True) // post condition  }  check assertIfStatement0 {  { all arg1:Bool,arg2:Int | some arg2':Int | ((((arg2 = 10) => True else False)) in True) => predIfStatement0[arg1,arg2,arg2'] }  } |
|  |

Figure 21: Demonstration: If Statements Output

We can clearly see a predicate and a check body in the output just like we discussed in 2.1.

In the check body we have the variables and their required post states getting declared as well as the precondition and the reference to the predicate generated for the if statement.

In the predicate body, our compiler generates a logically equivalent Alloy code along with the postcondition(s).

Please notice that our compiler did not generate any output for the if statement without the contract since there is nothing to verify!

# Examples

In this chapter we will be sharing code examples involving real-world scenarios rather than specifically demonstrating the features of the compiler.

## Example Input – If Statement

|  |
| --- |
| // Bank Transaction Application  int cust1Balance = 100;  int cust2Balance = 600;  int cust3Balance = 300;  // customer 3 transfers funds to customer 1  cust3Balance = cust3Balance - 200;  cust1Balance = cust1Balance + 200;  // cust3Balance deposits 300  cust3Balance = cust3Balance + 300;  // cust1Balance withdraws 200  cust1Balance = cust1Balance - 200;  // cust3 asks cust1 for a loan of 200  // this time since customer 1 does not know how much money he has, he says he will loan the money if he has enough.  int borrowAmount = 200;  if\_require(cust1Balance > borrowAmount)  if (cust1Balance > borrowAmount) {  cust1Balance = cust1Balance - borrowAmount;  cust3Balance = cust3Balance + borrowAmount;  }  if\_ensure((cust1Balance == cust1Balance\_old - borrowAmount) && (cust3Balance == cust3Balance\_old + borrowAmount)) |
|  |

Figure 22: Example Input – If Statement

|  |
| --- |
| open logicFuncs  pred predIfStatement0 [arg2:Int,arg1,arg1':Int,arg3,arg3':Int] {  ((((arg3 > arg2) => True else False)) in True) =>  ((True) in True) and arg1'=arg1.add[arg2] and arg3'=arg3.sub[arg2]  ((andGate[((arg3' = arg3.sub[arg2]) => True else False), ((arg1' = arg1.add[arg2]) => True else False)]) in True) // post condition  }  check assertIfStatement0 {  { all arg2:Int,arg1:Int,arg3:Int | some arg1':Int,arg3':Int | ((((arg3 > arg2) => True else False)) in True) => predIfStatement0[arg2,arg1,arg1',arg3,arg3'] }  } |
|  |

Figure 23: Example Output – If Statement

* Variable Declarations and Assignments
* If Statements with Contracts
* Arithmetic Operations
* Relational Operations
* Logical Operations
* Old syntax
  + Notice in the postcondition of the if statement that we are accessing the pre-state of some of the variables.
* This program is logically correct.
* The output program compiles and verifies with no counterexamples in Alloy.

## Example Input - Double Nested If Statement

|  |
| --- |
| // Yet Another Bank Transaction Application  int cust1Balance = 100;  int cust2Balance = 600;  int cust3Balance = 300;  // customer 3 transfers funds to customer 1  cust3Balance = cust3Balance - 200;  cust1Balance = cust1Balance + 200;  // cust3Balance deposits 300  cust3Balance = cust3Balance + 300;  // cust1Balance withdraws 200  cust1Balance = cust1Balance - 200;  cust2Balance = cust2Balance + 9999;  cust1Balance = cust1Balance + 9999;  // cust3 asks cust1 for a loan of 200  // this time since customer 1 knows that he received a hefty amount of money recently, he does not care about  // the amount. though, he says that he will only pay half and on the condition that cust2 will pay the other half as well.  // cust3 also makes contingency plans to get a loan from the bank in case he cannot borrow any money from the other customers  int borrowAmount = 200;  int borrowAmountHalf = borrowAmount / 2;  if\_require(cust1Balance > borrowAmountHalf && cust2Balance > borrowAmountHalf)  if (cust1Balance > borrowAmountHalf) {  if (cust2Balance > borrowAmountHalf) {  cust1Balance = cust1Balance - borrowAmountHalf;  cust2Balance = cust2Balance - borrowAmountHalf;  cust3Balance = cust3Balance + borrowAmount;  } else {  // cust3 gets a loan from the bank  cust3Balance = cust3Balance + borrowAmount;  }  } else {  // cust3 gets a loan from the bank  cust3Balance = cust3Balance + borrowAmount;  }  if\_ensure(cust3Balance == cust3Balance\_old + borrowAmount) |
|  |

Figure 24: Example Input – Double Nested If Statemen

|  |
| --- |
| open logicFuncs  pred predIfStatement0 [arg4:Int,arg1,arg1':Int,arg2,arg2',arg2'',arg2''':Int,arg3:Int,arg5,arg5':Int] {  ((((arg5 > arg3) => True else False)) in True) =>  ((True) in True) and (((((arg1 > arg3) => True else False)) in True) =>  ((True) in True) and arg1'=arg1.sub[arg3] and arg5'=arg5.sub[arg3] and arg2'=arg2.add[arg4] and arg2''=arg2'  else  ((True) in True) and arg2''=arg2.add[arg4] and arg1'=arg1 and arg5'=arg5) and arg2'''=arg2''  else  ((True) in True) and arg2'''=arg2.add[arg4] and arg1' = arg1 and arg5'=arg5  ((((arg2''' = arg2.add[arg4]) => True else False)) in True) // post condition  }  check assertIfStatement0 {  { all arg4:Int,arg1:Int,arg2:Int,arg3:Int,arg5:Int | some arg1':Int,arg2':Int,arg2'':Int,arg2''':Int,arg5':Int | ((andGate[((arg5 > arg3) => True else False), ((arg1 > arg3) => True else False)]) in True) => predIfStatement0[arg4,arg1,arg1',arg2,arg2',arg2'',arg2''',arg3,arg5,arg5'] }  } |
|  |

Figure 25: Example Output – Double Nested If Statement

* Variable Declarations and Assignments
* If Statements with Contracts (DOUBLE NESTED)
* Arithmetic Operations
* Relational Operations
* Logical Operations
* Old syntax
* This program is logically correct.
* The output program compiles and verifies with no counterexamples in Alloy.

## Example Input - Triple Nested If Statement

|  |
| --- |
| // Bank Transaction Application  // Takes forever to verify.  int cust1Balance = 100;  int cust2Balance = 600;  int cust3Balance = 300;  // customer 3 transfers funds to customer 1  cust3Balance = cust3Balance - 200;  cust1Balance = cust1Balance + 200;  // cust3Balance deposits 300  cust3Balance = cust3Balance + 300;  // cust1Balance withdraws 200  cust1Balance = cust1Balance - 200;  cust2Balance = cust2Balance + 9999;  cust1Balance = cust1Balance + 9999;  int bank = 99999;  // cust3 asks cust1 for a loan of 200  // this time since cust1 knows that he received a hefty amount of money recently, he does not care about  // the amount. though, he says that he will only pay half and on the condition that cust2 will pay the other half as well.  // cust3 also makes contingency plans to get a loan from the bank in case he cannot borrow any money from the other customers.  // bank also decides to make a gesture and says that they will give cust3 the full amount for free on the condition that  // cust1 and cust2 pay their halves.  int borrowAmount = 200;  int borrowAmountHalf = borrowAmount / 2;  if\_require(cust1Balance > borrowAmountHalf && cust2Balance > borrowAmountHalf)  if (cust1Balance > borrowAmountHalf) {  if (cust2Balance > borrowAmountHalf) {  cust1Balance = cust1Balance - borrowAmountHalf;  cust2Balance = cust2Balance - borrowAmountHalf;  cust3Balance = cust3Balance + borrowAmount;  if (bank > borrowAmount){  bank = bank - borrowAmount;  cust3Balance = cust3Balance + borrowAmount;  }  } else {  // cust3 gets a loan from the bank  cust3Balance = cust3Balance + borrowAmount;  }  } else {  // cust3 gets a loan from the bank  cust3Balance = cust3Balance + borrowAmount;  }  if\_ensure((cust3Balance >= (cust3Balance\_old + borrowAmount))) |
|  |

Figure 26: Example Input – Triple Nested If Statement

|  |
| --- |
| open logicFuncs  pred predIfStatement0 [arg1:Int,arg2,arg2':Int,arg3,arg3':Int,arg4,arg4',arg4'',arg4''',arg4'''':Int,arg5:Int,arg6,arg6':Int] {  ((((arg6 > arg5) => True else False)) in True) =>  ((True) in True) and ( ((((arg3 > arg5) => True else False)) in True) =>  ((True) in True) and ( ((((arg2 > arg1) => True else False)) in True) =>  ((True) in True) and arg4''=arg4'.add[arg1] and arg2'=arg2.sub[arg1] ) and arg4'=arg4.add[arg1] and arg3'=arg3.sub[arg5] and arg6'=arg6.sub[arg5] and arg4'''=arg4''  else  ((True) in True) and arg4'''=arg4.add[arg1] and arg3'=arg3 and arg2'=arg2 and arg6'=arg6 ) and arg4''''=arg4'''  else  ((True) in True) and arg4''''=arg4.add[arg1] and arg3'=arg3 and arg2'=arg2 and arg6'=arg6  ((((arg4'''' >= arg4.add[arg1]) => True else False)) in True) // post condition  }  check assertIfStatement0 {  { all arg1:Int,arg2:Int,arg3:Int,arg4:Int,arg5:Int,arg6:Int | some arg2':Int,arg3':Int,arg4':Int,arg4'':Int,arg4''':Int,arg4'''':Int,arg6':Int | ((andGate[((arg6 > arg5) => True else False), ((arg3 > arg5) => True else False)]) in True) => predIfStatement0[arg1,arg2,arg2',arg3,arg3',arg4,arg4',arg4'',arg4''',arg4'''',arg5,arg6,arg6'] }  } |
|  |

Figure 27: Example Output – Triple Nested If Statement

* Variable Declarations and Assignments
* If Statements with Contracts (TRIPLE NESTED)
* Arithmetic Operations
* Relational Operations
* Logical Operations
* Old syntax
* This program should be correct.
* The output program compiles and but the verification takes too long to finish. Therefore we cannot verify it.

## Example Input – Functions

|  |
| --- |
| // Bank Transaction Application  // One day, miracally, the bankers at the bank finally realize that  // they can use functions instead of doing everything manually.  // However government started taxing functional transaction for deposits. that's why now each deposit operation cause 1 dollar fee  // Customer Accounts  int cust1Balance = 100;  int cust2Balance = 600;  int cust3Balance = 300;  fun int withdraw (int amount, int balance, int result) {  fun\_require((balance >= amount) && (amount >= 0))  result = balance - amount;  return result;  fun\_ensure(result == balance\_old-amount\_old && (result>=0))  }  fun int deposit (int amount, int balance, int result) {  fun\_require(amount > 1)  balance = withdraw (1, balance, result); // Tax  result = balance + amount; // new balance  return result;  fun\_ensure(result == balance\_old + amount\_old-1)  }  // customer 3 transfers funds to customer 1  cust3Balance = withdraw(200, cust3Balance, 0);  cust1Balance = deposit(200, cust1Balance, 0);  // cust3Balance deposits 300  cust3Balance = deposit(300, cust3Balance, 0);  // cust1Balance withdraws 200  cust1Balance = withdraw(200, cust1Balance, 0); |
|  |

Figure 28: Example Input – Functions

|  |
| --- |
| open logicFuncs  pred predFunction0 [arg2,arg2':Int,arg3:Int,arg1:Int] {  ((True) in True) and arg2'=arg1.sub[arg3]  ((andGate[((arg2' = arg1.sub[arg3]) => True else False), ((arg2' >= 0) => True else False)]) in True) // post condition  }  fun funFunction0 [arg2,arg2':Int,arg3:Int,arg1:Int] : Int {  { return : Int | ((True) in True) and arg2'=arg1.sub[arg3] and return = arg2'}  }  check assertFunction0 {  { all arg2:Int,arg3:Int,arg1:Int | some arg2':Int | ((andGate[((arg1 >= arg3) => True else False), ((arg3 >= 0) => True else False)]) in True) => predFunction0[arg2,arg2',arg3,arg1] }  }  pred predFunction1 [arg2,arg2',arg2'':Int,arg3:Int,arg1,arg1':Int] {  ((True) in True) and arg1'=funFunction0[arg2,arg2',1,arg1] and arg2''=arg1'.add[arg3]  ((((arg2'' = arg1.add[arg3].sub[1]) => True else False)) in True) // post condition  }  fun funFunction1 [arg2,arg2',arg2'':Int,arg3:Int,arg1,arg1':Int] : Int {  { return : Int | ((True) in True) and arg1'=funFunction0[arg2,arg2',1,arg1] and arg2''=arg1'.add[arg3] and return = arg2''}  }  check assertFunction1 {  { all arg2:Int,arg3:Int,arg1:Int | some arg2':Int,arg2'':Int,arg1':Int | ((((arg3 > 1) => True else False)) in True) => predFunction1[arg2,arg2',arg2'',arg3,arg1,arg1'] }  } |
|  |

Figure 29: Example Output – Functions

* Variable Declarations
* Arithmetic Operations
* Relational Operations
* Logical Operations
* Old syntax
* Functions, Function Declarations, Function Calls, Function Contracts
* Please notice that one of the functions is calling another function in this code snippet.
* This program is logically correct.
* The output program compiles and verifies with no counterexamples in Alloy.

## Example Input – Function with an If Statement

|  |
| --- |
| // Alyx got 7 on her English test.  // Marking Scheme 6-7 A 3-5 B 0-2 C  // A -> 1  // B -> 2  // C -> 3  fun int mark (int score, int result) {  fun\_require((score >= 0) && (score <= 7))  if ((score >= 6) && (score <= 7)) {  result = 1;  } else if ((score >= 3) && (score <= 5)) {  result = 2;  } else {  result = 3;  }  return result;  fun\_ensure((result >= 1) && (result <= 3))  }  int alyxScore = 7;  int alyxMark = mark(alyxScore, 0); |
|  |

Figure 30: Example Input – Function with an If Statement

|  |
| --- |
| open logicFuncs  pred predFunction0 [arg1,arg1',arg1'',arg1''':Int,arg2:Int] {  ((True) in True) and ( ((andGate[((arg2 >= 6) => True else False), ((arg2 <= 7) => True else False)]) in True) =>  ((True) in True) and arg1'=1 and arg1'''=arg1'  else (( andGate[((arg2 >= 3) => True else False), ((arg2 <= 5) => True else False)] ) in True) =>  ((True) in True) and arg1''=2 and arg1'''=arg1''  else  ((True) in True) and arg1'''=3 )  ((andGate[((arg1''' >= 1) => True else False), ((arg1''' <= 3) => True else False)]) in True) // post condition  }  fun funFunction0 [arg1,arg1',arg1'',arg1''':Int,arg2:Int] : Int {  { return : Int | ((True) in True) and ( ((andGate[((arg2 >= 6) => True else False), ((arg2 <= 7) => True else False)]) in True) =>  ((True) in True) and arg1'=1 and arg1'''=arg1'  else (( andGate[((arg2 >= 3) => True else False), ((arg2 <= 5) => True else False)] ) in True) =>  ((True) in True) and arg1''=2 and arg1'''=arg1''  else  ((True) in True) and arg1'''=3 ) and return = arg1'''}  }  check assertFunction0 {  { all arg1:Int,arg2:Int | some arg1':Int,arg1'':Int,arg1''':Int | ((andGate[((arg2 >= 0) => True else False), ((arg2 <= 7) => True else False)]) in True) => predFunction0[arg1,arg1',arg1'',arg1''',arg2] }  } |
|  |

Figure 30: Example Output – Function with an If Statement

* Variable Declarations
* Arithmetic Operations
* Relational Operations
* Logical Operations
* Functions, Function Declarations, Function Calls, Function Contracts
* If Statements (Notice that this time the if statement is in the function)
* This program is logically correct.
* The output program compiles and verifies with no counterexamples in Alloy.

## Example Input - Loop

|  |
| --- |
| // Joan started learning adding numbers today!  // Will you help her on her adventure adding all numbers between 1-3?  int sum; // Joan will add each number to sum.  int current; // This is the current number Joan is looking at.  loop\_require(true)  loop\_init{  sum = 0; // Joan has nothing added to sum yet. So she sets it to 0.  current = 3; // Joan sets the current number before starting.  }  loop(current > 0 ) {  loop\_invariant((sum == (((3-current) \* ((3-current) + 1)) / 2)) && (current >= 0) && (current < 4) )  loop\_variant(current)  sum = sum + (4-current);  current = current - 1;  }  loop\_ensure(sum == 6)  // current sum in loop invariant is calculated using the formula n \* (n + 1) / 2 |
|  |

Figure 31: Example Input – Loops

|  |
| --- |
| open logicFuncs  pred predForStatement0 [arg2,arg2',arg2'':Int,arg1,arg1',arg1'':Int] {  (((True) in True) and arg1'=0 and arg2'=3  =>  ( andGate[andGate[((arg1' = 3.sub[arg2'].mul[3.sub[arg2'].add[1]].div[2]) => True else False), ((arg2' >= 0) => True else False)], ((arg2' < 4) => True else False)] in True ))  (( andGate[andGate[((arg1 = 3.sub[arg2].mul[3.sub[arg2].add[1]].div[2]) => True else False), ((arg2 >= 0) => True else False)], ((arg2 < 4) => True else False)] in True )  and( ((arg2 > 0) => True else False) in True ))  =>  ((True) in True) and arg2''=arg2.sub[1] and arg1''=arg1.add[4.sub[arg2]]  =>  (andGate[andGate[((arg1'' = 3.sub[arg2''].mul[3.sub[arg2''].add[1]].div[2]) => True else False), ((arg2'' >= 0) => True else False)], ((arg2'' < 4) => True else False)] in True and ( arg2'' >= 0 ) and ( arg2 > arg2''))  (( andGate[andGate[((arg1 = 3.sub[arg2].mul[3.sub[arg2].add[1]].div[2]) => True else False), ((arg2 >= 0) => True else False)], ((arg2 < 4) => True else False)] in True ) and not(( ((arg2 > 0) => True else False) in True ))  =>  ((((arg1 = 6) => True else False)) in True) ) // post condition  }  check assertForStatement0 {  { all arg2:Int,arg1:Int | some arg2':Int,arg2'':Int,arg1':Int,arg1'':Int | ((True) in True) => predForStatement0[arg2,arg2',arg2'',arg1,arg1',arg1''] }  } |
|  |

Figure 32: Example Output – Loops

* Variable Declarations
* Arithmetic Operations
* Relational Operations
* Logical Operations
* Loops
* This program is logically correct.
* The output program compiles and verifies with no counterexamples in Alloy.

## Example Input - Loop

|  |
| --- |
| // Lucienne is trying to demonstrate her friend Alienor that and'ing any number of falses will still yield false.  // Since Lucienne is a coder she decides to use functions to make it look fancy. (We have no idea why Lucienne would ever want to do this using this compiler!)  fun bool andFalse (bool input, bool result) {  fun\_require(true)  result = input && false;  return result;  fun\_ensure(result <=> false)  }  int current;  bool result = true;  loop\_require(result <=> true)  loop\_init {  current = 3;  }  loop(current > 0) {  loop\_invariant((current >= 0) && ( (current<3) => (result <=> false) ))  loop\_variant(current)  result = andFalse(result, false);  current = current - 1;  }  loop\_ensure(result <=> false) |
|  |

Figure 33: Example Input – Loops

|  |
| --- |
| open logicFuncs  pred predFunction0 [arg1,arg1':Bool,arg2:Bool] {  ((True) in True) and arg1'=andGate[arg2, False]  ((((arg1' in False) => True else False)) in True) // post condition  }  fun funFunction0 [arg1,arg1':Bool,arg2:Bool] : Bool {  { return : Bool | ((True) in True) and arg1'=andGate[arg2, False] and return = arg1'}  }  check assertFunction0 {  { all arg1:Bool,arg2:Bool | some arg1':Bool | ((True) in True) => predFunction0[arg1,arg1',arg2] }  }  pred predForStatement0 [arg1,arg1':Bool,arg2,arg2',arg2'':Int,arg3,arg3':Bool] {  (((True) in True) and arg3=False and arg2'=3  =>  ( andGate[((arg2' >= 0) => True else False), ((((((arg2' < 3) => True else False)) in True) => ((((arg1 in False) => True else False)) in True)) => True else False)] in True ))  (( andGate[((arg2 >= 0) => True else False), ((((((arg2 < 3) => True else False)) in True) => ((((arg1 in False) => True else False)) in True)) => True else False)] in True )  and( ((arg2 > 0) => True else False) in True ))  =>  ((True) in True) and arg1'=funFunction0[arg3,arg3',arg1] and arg2''=arg2.sub[1] and arg3=False  =>  (andGate[((arg2'' >= 0) => True else False), ((((((arg2'' < 3) => True else False)) in True) => ((((arg1' in False) => True else False)) in True)) => True else False)] in True and ( arg2'' >= 0 ) and ( arg2 > arg2''))  (( andGate[((arg2 >= 0) => True else False), ((((((arg2 < 3) => True else False)) in True) => ((((arg1 in False) => True else False)) in True)) => True else False)] in True ) and not(( ((arg2 > 0) => True else False) in True ))  =>  ((((arg1 in False) => True else False)) in True) ) // post condition  }  check assertForStatement0 {  { all arg1:Bool,arg2:Int,arg3:Bool | some arg1':Bool,arg2':Int,arg2'':Int,arg3':Bool | ((((arg1 in True) => True else False)) in True) => predForStatement0[arg1,arg1',arg2,arg2',arg2'',arg3,arg3'] }  } |
|  |

Figure 34: Example Output – Loops

* Variable Declarations
* Arithmetic Operations
* Relational Operations
* Logical Operations
* Loops
* This program is logically correct.
* The output program compiles and verifies with no counterexamples in Alloy.

## Example Input – Lists and Quantification Operations

|  |
| --- |
| // Bank Account Balance Monitor  // A Multinational bank wants to figure out if there are any empty accounts, but they would like to know  // if there are multiple ways of doing it.  int[] accountBalances;  accountBalances.add(400);  accountBalances.add(30000);  accountBalances.add(5000);  accountBalances.add(10000000);  accountBalances.add(0);  // First way with the existential quantifier.  bool check1 = false;  if\_require(true)  if(accountBalances.some(each == 0)) {  check1 = true;  }  if\_ensure(true)  bool check2 = false;  if\_require(true)  if(accountBalances.some(each == 0)) {  check2 = true;  }  if\_ensure(true) |
|  |

Figure 35: Example Input – Lists and Quantification Operations

|  |
| --- |
| open logicFuncs  pred predIfStatement0 [arg2,arg2':Bool,arg1:seq Int] {  ((((some arrayElems: arg1.elems | arrayElems in {each: Int | (((each = 0) => True else False) in True)})=> True else False)) in True) =>  ((True) in True) and arg2'=True  ((True) in True) // post condition  }  check assertIfStatement0 {  { all arg2:Bool,arg1:seq Int | some arg2':Bool | ((True) in True) => predIfStatement0[arg2,arg2',arg1] }  }  pred predIfStatement1 [arg2,arg2':Bool,arg1:seq Int] {  ((((some arrayElems: arg1.elems | arrayElems in {each: Int | (((each = 0) => True else False) in True)})=> True else False)) in True) =>  ((True) in True) and arg2'=True  ((True) in True) // post condition  }  check assertIfStatement1 {  { all arg2:Bool,arg1:seq Int | some arg2':Bool | ((True) in True) => predIfStatement1[arg2,arg2',arg1] }  } |
|  |

Figure 36: Example Output – Lists and Quantification Operations

* Variable Declarations
* Arithmetic Operations
* Relational Operations
* Logical Operations
* If Statements with Contracts
* Lists, List Operations (Add), Quantification Operation
* This program is logically ???????????
* The output program ???????????

## Example Input – Function Calling Loop

|  |
| --- |
| // Gregory writes an algorithm that counts the number of integers that are a multiple of 3 and lower than 100;  // Since the numbers are greater than 7 therefore Alloy won't behave normally (as explained in the User Manual)  int counter = 0;  fun bool shouldIncrementCounter (int number, bool result) {  fun\_require((number >= 0) && (number < 100))  result = false;  if ((number % 3) == 0) {  result = true;  }  return result;  fun\_ensure(true)  }  int current;  bool shouldIncrement;  loop\_require(true)  loop\_init {  current = 99;  }  loop(current !=0 ) {  loop\_invariant((current >= 0) && (current < 100) && (counter == ((99-current)/3)) )  loop\_variant(current)  shouldIncrement = shouldIncrementCounter(current, false);  if (shouldIncrement <=> true) {  counter = counter + 1;  }  current = current - 1;  }  loop\_ensure(counter == 33) |
|  |

Figure 37: Example Input – Function Calling Loop

|  |
| --- |
| open logicFuncs  pred predFunction0 [arg1,arg1',arg1'':Bool,arg2:Int] {  ((True) in True) and arg1'=False and ( ((((arg2.rem[3] = 0) => True else False)) in True) =>  ((True) in True) and arg1''=True )  ((True) in True) // post condition  }  fun funFunction0 [arg1,arg1',arg1'':Bool,arg2:Int] : Bool {  { return : Bool | ((True) in True) and arg1'=False and ( ((((arg2.rem[3] = 0) => True else False)) in True) =>  ((True) in True) and arg1''=True ) and return = arg1''}  }  check assertFunction0 {  { all arg1:Bool,arg2:Int | some arg1':Bool,arg1'':Bool | ((andGate[((arg2 >= 0) => True else False), ((arg2 < 100) => True else False)]) in True) => predFunction0[arg1,arg1',arg1'',arg2] }  }  pred predForStatement0 [arg1,arg1',arg1'':Int,arg3,arg3':Bool,arg2,arg2':Int,arg4,arg4',arg4'':Bool] {  (((True) in True) and arg1'=99 and arg4=False  =>  ( andGate[andGate[((arg1' >= 0) => True else False), ((arg1' < 100) => True else False)], ((arg2 = 99.sub[arg1'].div[3]) => True else False)] in True ))  (( andGate[andGate[((arg1 >= 0) => True else False), ((arg1 < 100) => True else False)], ((arg2 = 99.sub[arg1].div[3]) => True else False)] in True )  and( ((arg1 != 0) => True else False) in True ))  =>  ((True) in True) and ( ((((arg3' in True) => True else False)) in True) =>  ((True) in True) and arg4=False and arg2'=arg2.add[1] ) and arg4=False and arg3'=funFunction0[arg4,arg4',arg4'',arg1] and arg1''=arg1.sub[1]  =>  (andGate[andGate[((arg1'' >= 0) => True else False), ((arg1'' < 100) => True else False)], ((arg2' = 99.sub[arg1''].div[3]) => True else False)] in True and ( arg1'' >= 0 ) and ( arg1 > arg1''))  (( andGate[andGate[((arg1 >= 0) => True else False), ((arg1 < 100) => True else False)], ((arg2 = 99.sub[arg1].div[3]) => True else False)] in True ) and not(( ((arg1 != 0) => True else False) in True ))  =>  ((((arg2 = 33) => True else False)) in True) ) // post condition  }  check assertForStatement0 {  { all arg1:Int,arg3:Bool,arg2:Int,arg4:Bool | some arg1':Int,arg1'':Int,arg3':Bool,arg2':Int,arg4':Bool,arg4'':Bool | ((True) in True) => predForStatement0[arg1,arg1',arg1'',arg3,arg3',arg2,arg2',arg4,arg4',arg4''] }  } |
|  |

Figure 38: Example Output – Function Calling Loop

* Variable Declarations
* Arithmetic Operations
* Relational Operations
* Logical Operations
* If Statements with Contracts
* Functions, Function Declarations, Function Calls, Function Contracts
* Loops
* Please notice that loop is making a call to another function.
* This program is logically ???????????
* The output program ???????????

## Example Input – Chef’s Special

|  |
| --- |
| // Ege decided to write a simple list application to keep track of his marks at school.  // Causes state explosion (Skolemize error as explained in the User Manual)  int[] myMarks;  // Ege coincidentally got 70 on all his midterm tests, so he adds those first.  int current;  loop\_require(true)  loop\_init {  current = 3;  }  loop(current > 0) {  loop\_invariant((current >= 0) && (current < 4) )  loop\_variant(current)  myMarks.add(70);  current = current - 1;  }  loop\_ensure(current == 0)  // Then he gets 35 and 90 on two tests.  myMarks.add(35);  myMarks.add(90);  // Then his prof says one of his quizzes can count for marks if he has any marks lower than 50 or greater than 90.  // His prof also drops his lowest mark.  // On this quiz Ege got 80.  bool hasLowMark = (myMarks.some(each < 50) || myMarks.some(each > 90));  if (hasLowMark <=> true) {  myMarks.add(80);  myMarks.remove(35);  } |
|  |

Figure 39: Example Input – Chef’s Special

|  |
| --- |
| open logicFuncs  pred predForStatement0 [arg1,arg1',arg1'':Int,arg2,arg2':seq Int] {  (((True) in True) and arg1'=3  =>  ( andGate[((arg1' >= 0) => True else False), ((arg1' < 4) => True else False)] in True ))  (( andGate[((arg1 >= 0) => True else False), ((arg1 < 4) => True else False)] in True )  and( ((arg1 > 0) => True else False) in True ))  =>  ((True) in True) and arg2'=arg2.add[70] and arg1''=arg1.sub[1]  =>  (andGate[((arg1'' >= 0) => True else False), ((arg1'' < 4) => True else False)] in True and ( arg1'' >= 0 ) and ( arg1 > arg1''))  (( andGate[((arg1 >= 0) => True else False), ((arg1 < 4) => True else False)] in True ) and not(( ((arg1 > 0) => True else False) in True ))  =>  ((((arg1 = 0) => True else False)) in True) ) // post condition  }  check assertForStatement0 {  { all arg1:Int,arg2:seq Int | some arg1':Int,arg1'':Int,arg2':seq Int | ((True) in True) => predForStatement0[arg1,arg1',arg1'',arg2,arg2'] }  } |
|  |

Figure 40: Example Output – Chef’s Special

* Variable Declarations
* Arithmetic Operations
* Relational Operations
* Logical Operations
* If Statements with Contracts
* Lists, List Operations (Add), Quantification Operation
* Loops
* This is just an ordinary example attempting to put together everything we did.
* This program is logically ???????????
* The output program ???????????

## Example Input – Incorrect 3.1

|  |
| --- |
| // THIS IS ALMOST THE SAME FILE AS input-1.txt, BUT WITH THE OPERATORS FLIPPED IN THE  // IF BODY CAUSING POSTCONDITION TO FAIL ON PURPOSE  // ALLOY FINDS A COUNTER EXAMPLE!  // Bank Transaction Application  int cust1Balance = 100;  int cust2Balance = 600;  int cust3Balance = 300;  // customer 3 transfers funds to customer 1  cust3Balance = cust3Balance - 200;  cust1Balance = cust1Balance + 200;  // cust3Balance deposits 300  cust3Balance = cust3Balance + 300;  // cust1Balance withdraws 200  cust1Balance = cust1Balance - 200;  // cust3 asks cust1 for a loan of 200  // this time since customer 1 does not know how much money he has, he says he will loan the money if he has enough.  int borrowAmount = 200;  if\_require(cust1Balance > borrowAmount)  if (cust1Balance > borrowAmount) {  cust1Balance = cust1Balance + borrowAmount;  cust3Balance = cust3Balance - borrowAmount;  }  if\_ensure((cust1Balance == cust1Balance\_old - borrowAmount) && (cust3Balance == cust3Balance\_old + borrowAmount)) |
|  |

Figure 41: Example Input – Incorrect 3.1

|  |
| --- |
| open logicFuncs  pred predIfStatement0 [arg2:Int,arg1,arg1':Int,arg3,arg3':Int] {  ((((arg3 > arg2) => True else False)) in True) =>  ((True) in True) and arg1'=arg1.sub[arg2] and arg3'=arg3.add[arg2]  ((andGate[((arg3' = arg3.sub[arg2]) => True else False), ((arg1' = arg1.add[arg2]) => True else False)]) in True) // post condition  }  check assertIfStatement0 {  { all arg2:Int,arg1:Int,arg3:Int | some arg1':Int,arg3':Int | ((((arg3 > arg2) => True else False)) in True) => predIfStatement0[arg2,arg1,arg1',arg3,arg3'] }  } |
|  |

Figure 42: Example Output – Incorrect 3.1

* This is just the Incorrect version of 3.1
* This program is logically ???????????
* The output program ???????????

## Example Input – Incorrect 3.7

|  |
| --- |
| // THIS IS ALMOST THE SAME FILE AS input-7.txt, BUT WITH THE ANDFALSE FUNCTION AND'ING TRUE INSTEAD BY MISTAKE!  // ALLOY FINDS A COUNTER EXAMPLE!  // Lucienne is trying to demonstrate her friend Alienor that and'ing any number of falses will still yield false.  // Since Lucienne is a coder she decides to use functions to make it look fancy. (We have no idea why Lucienne would ever want to do this using this compiler!)  fun bool andFalse (bool input, bool result) {  fun\_require(true)  result = input && true;  return result;  fun\_ensure(result <=> false)  }  int current;  bool result = true;  loop\_require(result <=> true)  loop\_init {  current = 3;  }  loop(current > 0) {  loop\_invariant((current >= 0) && ( (current<3) => (result <=> false) ))  loop\_variant(current)  result = andFalse(result, false);  current = current - 1;  }  loop\_ensure(result <=> false) |
|  |

Figure 43: Example Input – Incorrect 3.7

|  |
| --- |
| open logicFuncs  pred predFunction0 [arg1,arg1':Bool,arg2:Bool] {  ((True) in True) and arg1'=andGate[arg2, True]  ((((arg1' in False) => True else False)) in True) // post condition  }  fun funFunction0 [arg1,arg1':Bool,arg2:Bool] : Bool {  { return : Bool | ((True) in True) and arg1'=andGate[arg2, True] and return = arg1'}  }  check assertFunction0 {  { all arg1:Bool,arg2:Bool | some arg1':Bool | ((True) in True) => predFunction0[arg1,arg1',arg2] }  }  pred predForStatement0 [arg1,arg1':Bool,arg2,arg2',arg2'':Int,arg3,arg3':Bool] {  (((True) in True) and arg3=False and arg2'=3  =>  ( andGate[((arg2' >= 0) => True else False), ((((((arg2' < 3) => True else False)) in True) => ((((arg1 in False) => True else False)) in True)) => True else False)] in True ))  (( andGate[((arg2 >= 0) => True else False), ((((((arg2 < 3) => True else False)) in True) => ((((arg1 in False) => True else False)) in True)) => True else False)] in True )  and( ((arg2 > 0) => True else False) in True ))  =>  ((True) in True) and arg1'=funFunction0[arg3,arg3',arg1] and arg2''=arg2.sub[1] and arg3=False  =>  (andGate[((arg2'' >= 0) => True else False), ((((((arg2'' < 3) => True else False)) in True) => ((((arg1' in False) => True else False)) in True)) => True else False)] in True and ( arg2'' >= 0 ) and ( arg2 > arg2''))  (( andGate[((arg2 >= 0) => True else False), ((((((arg2 < 3) => True else False)) in True) => ((((arg1 in False) => True else False)) in True)) => True else False)] in True ) and not(( ((arg2 > 0) => True else False) in True ))  =>  ((((arg1 in False) => True else False)) in True) ) // post condition  }  check assertForStatement0 {  { all arg1:Bool,arg2:Int,arg3:Bool | some arg1':Bool,arg2':Int,arg2'':Int,arg3':Bool | ((((arg1 in True) => True else False)) in True) => predForStatement0[arg1,arg1',arg2,arg2',arg2'',arg3,arg3'] }  } |
|  |

Figure 44: Example Output – Incorrect 3.7

* This is just the Incorrect version of 3.7
* This program is logically ???????????
* The output program ???????????

## Example Input – Incorrect 3.10

|  |
| --- |
| // THIS IS ALMOST THE SAME FILE AS input-10.txt, BUT NOW loop\_ensure CHECKS FOR current IS EQUAL TO 2 WHICH IS INCORRECT!  // Line 16 is commented so that there won't be any skolemization issues with Alloy (as it is explained in the user manual)  // ALLOY FINDS A COUNTER EXAMPLE!  // Ege decided to write a simple list application to keep track of his marks at school.  // Causes state explosion  int[] myMarks;  // Ege coincidentally got 70 on all his midterm tests, so he adds those first.  int current;  loop\_require(true)  loop\_init {  current = 3;  }  loop(current > 0) {  loop\_invariant((current >= 0) && (current < 4) )  loop\_variant(current)  // myMarks.add(70);  current = current - 1;  }  loop\_ensure(current == 2)  // Then he gets 35 and 90 on two tests.  myMarks.add(35);  myMarks.add(90);  // Then his prof says one of his quizzes can count for marks if he has any marks lower than 50 or greater than 90.  // His prof also drops his lowest mark.  // On this quiz Ege got 80.  bool hasLowMark = (myMarks.some(each < 50) || myMarks.some(each > 90));  if (hasLowMark <=> true) {  myMarks.add(80);  myMarks.remove(35); |
|  |

Figure 45: Example Input – Incorrect 3.10

|  |
| --- |
| open logicFuncs  pred predForStatement0 [arg1,arg1',arg1'':Int] {  (((True) in True) and arg1'=3  =>  ( andGate[((arg1' >= 0) => True else False), ((arg1' < 4) => True else False)] in True ))  (( andGate[((arg1 >= 0) => True else False), ((arg1 < 4) => True else False)] in True )  and( ((arg1 > 0) => True else False) in True ))  =>  ((True) in True) and arg1''=arg1.sub[1]  =>  (andGate[((arg1'' >= 0) => True else False), ((arg1'' < 4) => True else False)] in True and ( arg1'' >= 0 ) and ( arg1 > arg1''))  (( andGate[((arg1 >= 0) => True else False), ((arg1 < 4) => True else False)] in True ) and not(( ((arg1 > 0) => True else False) in True ))  =>  ((((arg1 = 2) => True else False)) in True) ) // post condition  }  check assertForStatement0 {  { all arg1:Int | some arg1':Int,arg1'':Int | ((True) in True) => predForStatement0[arg1,arg1',arg1''] }  } |
|  |

Figure 46: Example Input – Incorrect 3.10

* This is just the Incorrect version of 3.10
* This program is logically ???????????
* The output program ???????????

# Miscellaneous Features

The main purpose of this project is program verification. Any additional features supported by your compiler such as type checking, error reporting, handling of multiple input files, *etc.* should be listed here. For each feature:

1. Describe how it works
2. Give examples (or refer to some of the examples you submit), screenshots, *etc.*

**ADD section to talk about Nested if statements**

**Operator precedence**

**How are we handling Sequential assignments**

**Explain Boolean class**

**documentationda bahset return x - y olmuyor**

**variablelar disarida declare edilip fonksyiona verilecek**

**never talked about delete in arrays**

**explain array methods more in detail**

**explain how we match with each**

**cannot declare variables inside functions**

**function scope does not see globals**

**quantifications inside functions cause null pointer**

**improve explanations in the examples**

**edit table of contents**

**fix table of contents page numbers**

**check git issues**

**fix question marks in examples in docs**

# Limitations

* Make sure to Allow warnings in Alloy (Maybe mention it in the first section):
  + Because of the way we handle Boolean statements. Alloy generates warnings, saying some of the statements are redundant. By default, Alloy analyzer will not execute with warnings. The user needs to disable this functionality by clicking on Allow warnings in the Options tab.
* Output inconsistency in Alloy

|  |
| --- |
| * open logicFuncs * pred predIfStatement0 [arg1: Int, arg1': Int] { * arg1'=3 * } * pred predIfStatement1 [arg1: Int, arg1': Int] { * arg1'=3 => arg1'=2 * } * check assertIfStatement0 { * { all arg1:Int | some arg1':Int | predIfStatement0[arg1,arg1'] } * } * check assertIfStatement1 { * { all arg1:Int | some arg1':Int | predIfStatement1[arg1,arg1'] } * } |
|  |

Figure : Lists Example of an Issue Causing a Counterexample

* + In the given example, check assertIfStatement0 succeeds because assigning arg1’ to a constant does not generate any counter examples. Based on our understanding, since arg1’ is declared as a free variable (i.e. it is declared as some arg1’:Int) therefore there exist a case where arg1’=3 is true.
  + On the other hand, predIfStatement1 does not generate counter examples either. However, this is incorrect because the implication can not possibly be correct arg1'=3 => arg1'=2. Meaning, the case where arg1’=3 is true arg1’=2 can not be correct and this should generate counter examples.
  + Luckily this only causes issues when translating the loops and does not effect the other statement translations.
* Using large values for inputs causes inconsistencies with Alloy. Since the default Bit-Width is 4 bits. Unless the user specifically changes the Bit-Width in Alloy. The Alloy analyzer will have issues when working with values out of the range [-8,7] (both inclusive).
* For each programming feature, do you support it fully? Or there is certain scenario that’s not supported, e.g., simple loops rather than nested loops?
* List any other known limitations (e.g., certain input programs, although can be compiled to generate outputs, cannot be verified by the target tool).