*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)

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

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

[1.3.1 Feature 1: ?? 5](#_Toc36488673)

[1.3.2 Feature 2: ?? 5](#_Toc36488674)

[2 Output Specification Language 6](#_Toc36488675)

[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 Multiple functions, if statements and a loop 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)

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 and equivalence.

|  |
| --- |
| **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 : Demonstration: Primitive Types

### Variable Declaration and Assignment

The input programming language supports declaration of variables and assigning them values including assigning values from other variables. It is also worth noting here that sequential assignments are also supported.

|  |
| --- |
| **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 : Demonstration: Primitive Types

### Basic Type Checking

The input programming language supports type checking of variables during translation. Uninitialized variables will also be detected for type checking.

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

Figure 2: Demonstration: Type Checking

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

The input programming language supports if/else statements with or without contracts. Nested if statements are allowed as well.

|  |
| --- |
| **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 3: Demonstration: If Statements

### Functions and Function Calls (with Contracts)

Functions can be declared and called in the input programming language. The variables are passed by values to the function. Therefore, every function must have a return type and can have a number of parameters. It is enforced by design that every function has a require and ensure section for contracts. These contracts are not mandatory to implement and can be skipped by inserting “true”. The functions do not support deceleration of new variables in the function bodies. In order to use extra variables, the user needs to pass extra arguments to the function to use them in the function bodies. In addition, values of variables before the execution of function body 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. \*/* |
|  |

Figure 4: 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 actual loop body.

|  |
| --- |
|  |
|  |

Figure 5: Demonstration: Loops

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

The input programming language also supports lists of primitive types. 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 support add and remove operations. However, quantification operations over lists are supported 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 integer of arrays.\*/*  **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 6: 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, our compiler only translates the statements with contracts as there is nothing to verify otherwise, such as declaring a variable.

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\_invariant, loop\_variant
* fun, fun\_require, fun\_ensure, return
* true, false, int, bool, int[], bool[]
* each

Below keywords are 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 ()
* loop\_require, loop\_ensure ()
* invariant, variant ()
* fun\_require, fun\_ensure ()

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 predicate depending on user’s wish as our compiler makes a singleton data structure for all available variables within the scope and their respective values. However, users are

As demonstrated in 1.1.5 the keyword “old” can be used to retrieve the pre-state of variables if the code block introduces changes to the value of the variables. Since the only way variables’ values can change is through assignments, our compiler keeps record of changes to variables whenever it encounters an assignment and creates new variables for the new values of the original variables so that the original value of the variable can be preserved. Although, this is not explicitly visible to the user before or during the translation, these new variables will be available to the user in the translated code with the same original name of the variable followed by the prime symbol (‘).

It is currently possible to write fairly complex specification constructs thanks to all the operators and features of the input programming language. Below are code snippets demonstrating capabilities of our compiler.

|  |
| --- |
|  |
|  |

Figure 7: Demonstration: Complex Example 1

|  |
| --- |
|  |
|  |

Figure 8: Demonstration: Complex Example 2

|  |
| --- |
|  |
|  |

Figure 9: Demonstration: Complex Example 3

## List of Advanced Programming Features

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

The syntax of functions are much similar to modern programming languages as demonstrated in 1.1.5. We will not be explaining 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 examples 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 10: 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 11: 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 12: 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 13: Function Example Output Specification Language (Logically Incorrect)

We would like to briefly explain here how the input programming language is translated 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 want to explain the body of the check statement First and then explain the bodies of the predicate and 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 the 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 which was “true” in the input programming language and 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 or 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.

As a side note here

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 as to convince ourselves that the translation is valid thus it is semantics-preserving.

Finally, we should note one more time 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 at 1.1.6, however 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 given loop in our input language to valid and accurate Alloy Analyzer code. Conversely though 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 examples 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 14: 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 15: 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 16: 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 17: Loop Example Output Specification Language (Logically Incorrect)

We note here that the only difference between the two samples 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 incorrect code (Figure 16). This might mean there is a semantic issue in generated output, however we believe that the generated code is syntactically and semantically correct based on the Alloy specification. The reason for this is because we noted a logical inconsistency with Alloy which is explained in the Limitations section of this documentation.

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 a high-level this loop predicate checks that each of the below are TRUE.

* 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 error that we believe exists in Alloy.

### 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 explained in 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 in lists.

Due to state explosion issues and limitations in Alloy we currently do not support adding to lists inside any block with contracts. However, we can still generate code that compiles, we just cannot verify whether it is sound or not due to limitations in Alloy.

We are also aware of an issue that using existential operator (some) might cause a counterexample to be found. For instance, output generated from the below code would cause Alloy to find a counter example, which might mean that there is a semantic error in the way we generate output. However, we will not be fixing it now due to time constraints and Alloy documentation limitation regarding lists.

|  |
| --- |
| 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 18: 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 examples for this feature.

Here’s a working very simple code snippet that we will use to summarize the translation of quantification operations at 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 19: 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 20: Working Very Simple Output Specification Language

In this example we will not be explaining the whole translation since most of the translation is pretty similar to the previous one we have already explained, rather we will only explain how the translation for quantification operators work.

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

Figure 21: 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 22: Quantification Operation Specification Language

As you can see from Figure 21 and 22, translation is rather simple. Our code creates a satisfying set looking at the statement inside the quantification operation and then in the translation it checks whether each item in the list is the satisfying set or not.

Despite the issue with existential operator and the state explosion issue, we believe that our reasoning makes sense and translation of quantification operators are semantics-preserving as demonstrated in Figure 21 and 22.

# Output Specification Language

In this chapter we will first briefly talk about the general structure of 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 can be found in ???????????????????????????????????????????????????????????? folder. Storing this file at the same level as the file that is going to be executed on Alloy, will allow the Alloy analyzer to read from logicFuncs.als file and use it.

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 going to 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 23: 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 statement our compiler encodes the conditions that causes variable assignments as well as the variables of course! In addition, the post conditions are encoded into the predicate as well just like we mentioned in 2.1

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

|  |
| --- |
| // A program allowing liquor stores to check customers’ age.  int[] customerID;  int[] customerAge;  int counter = 1;  fun int addCustomer (int age) {  fun\_require(true)  customerID.add(counter);  customerAge.add(age);  counter = counter + 1;  fun\_ensure(counter == counter\_old + 1)  }  addCustomer(20);  addCustomer(25);  addCustomer(15);  bool allAdult = false;  if (customerAge.all(each > 19)) {  allAdult = true;  } |
|  |

Figure 18: Lists Example of an Issue Causing a Counterexample

## Example input

## Example input

## Example input

## Example input

## Example input

## Example input

## Example input

## Example input

## Example input

# 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.*

**Change /\* comment \*/ to / comment / in the documentation example**

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

**How are we handling Sequential assignments**

# 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).