BKMR Compiler User Manual

Team members:

|  |  |  |  |
| --- | --- | --- | --- |
| Last Name | First Name | EECS login | Student Number |
| Perit | Brynn | dperit | 216181281 |
| Manoharan | Krishaanth | krish100 | 216463150 |
| Rif | Yuval | uval45 | 216986960 |
| Rooney | Nicholas | rooneyn | 216259392 |

* This first page is the title page.
  + Feel free to design a title page of this user manual of your compiler

(e.g., with an informative title such as “*A Compiler for the Verification of a C-Like Programming Language: User Manual*”)

* + At least include **each** team member’s:
    - Name (Last name, First name)
    - Student Number
    - EECS Prism login name
* This template lists sections which your final report **must at least** cover. Feel free to add more sections if you think they **help readers learn about using your compiler**.
* There is **no** need for you to explain the design of Java classes in the model package.
* The **minimum** number of working examples you must submit is **10**.
  + These examples should **best** illustrate the ability of your compiler: **readers will assume that they represent the most sophisticated programs and/or tests that your compiler is capable of measuring**. For example, if there is no nested loop included in these examples, it will then be assumed that your compiler cannot handle nested loops.
  + However, optionally, you can supply as many more working examples as you see fit, if they help readers learn about the capability of your compiler.
* Keep the **Table of Contents** up to date when you submit this report.

Table of Content

s

[1 Input Languages3](#_Toc119991373)

[1.1 Structure of a Program3](#_Toc119991374)

[1.2 List of Advanced Programming Features5](#_Toc119991375)

[1.2.1 Feature 1: ??5](#_Toc119991376)

[1.2.2 Feature 2: ??5](#_Toc119991377)

[1.2.3 Feature 3: ??6](#_Toc119991378)

[1.3 Structure of a Test8](#_Toc119991379)

[1.4 How a Program and a Test Should be Distinguished8](#_Toc119991380)

[2 Output Structure9](#_Toc119991381)

[3 Justification of Output10](#_Toc119991382)

[3.1 Justification for the ?? Coverage Criterion10](#_Toc119991383)

[3.2 Justification for the ?? Coverage Criterion10](#_Toc119991384)

[3.3 Justification for the ?? Coverage Criterion10](#_Toc119991385)

[3.4 Justification for the ?? Coverage Criterion10](#_Toc119991386)

[4 Summary of Submitted Examples11](#_Toc119991387)

[4.1 Highlights of Example Input 111](#_Toc119991388)

[4.2 Highlights of Example Input 211](#_Toc119991389)

[4.3 Highlights of Example Input 311](#_Toc119991390)

[4.4 Highlights of Example Input 411](#_Toc119991391)

[4.5 Highlights of Example Input 511](#_Toc119991392)

[4.6 Highlights of Example Input 611](#_Toc119991393)

[4.7 Highlights of Example Input 711](#_Toc119991394)

[4.8 Highlights of Example Input 811](#_Toc119991395)

[4.9 Highlights of Example Input 911](#_Toc119991396)

[4.10 Highlights of Example Input 1011](#_Toc119991397)

[5 Miscellaneous Features13](#_Toc119991398)

[6 Limitations14](#_Toc119991399)

# Input Languages

## Structure of a Program

--Comments are indicated by a -- at the beginning of the line

--Each program must have one “void main()” method

void main(){

--Variables can be declared as int32, double, or bool types

int32 first = 7

--Variables can be initialized on declaration or not

double second

--Function calls are performed like this

second = testFunction(first, 5.0)

--Arithmetic operations include +,-,/,\*

int32 third = first + second

--Program output is performed through the print statement

print(third)

}

--Functions must declare a return type and can have no parameters or many

double testFunction(int32 param1, double param2){

int32 second = 5 + 8

double first = param1 \* param2

--The return statement is optional and terminates the function,

--returning the specified value

return first

}

This section is closely related to **Section 2** and **Section 3**.

In this section, outline, explain, and exemplify how a typical input program should look like.

For example, there might be sections of an input program (e.g., variable declarations, functions):

|  |
| --- |
| **module** M1  */\* Section: Variable Declarations \*/*  x: **INTEGER**  */\* Section: Functions \*/*  increment (v: **INTEGER**)  **do**  x := x + v  **do**  **end** |

Figure 1: Structure of Input Program

**Note**. Each of the ten examples you submit may focus on a different perspective of your programming language. On the other hand, the above example program, should demonstrate as many **programming** features as possible that are supported by your compiler; **it can just be symbolic** (e.g., using x, y, f1, a1), but it must be syntactically and type correct.

## List of Advanced Programming Features

You already implemented the basic programming features from the two milestones. In this section, there is no need to address them again. Instead, focus on those advanced features which you have (partially or completely) implemented.

### Feature 1: While loops

Syntax:

while (logical or relational expression){

expression

expression

...

}

Example:

while (param1 < param2){

param1 = param1 + 1

result = result + 1

}

Seen in input-4.txt

For each advanced feature you support (for verification), **at least**:

1. Explain its syntax
2. Give examples (by **presenting code snippets**)
3. Cross-reference the feature to the example(s) you submit.

### Feature 2: Nested if statements

Syntax:

if (logical or relational expression){

if (logical or relational expression){

expression

expression

...

}

}

Example:

if (param2 <= 10){

second = param1 \* param2

if (1 < 2){

double third = 5.0

}

first = param1 + second

}

Seen in input-3.txt

### Feature 3: if, else if, else statement

Syntax:

if (logical or relational expression){

expressions

}else if (logical or relational expression){

expressions

}else{

expressions

}

Example:

if (a < 5) {

b = 1 --skip

} else if (a < 6 && c == 7) {

b = 10 --execute

} else if (a < 3) {

b = 5 --skip

} else {

b = 6 --skip

}

Seen in input-5.txt

### Feature 4: bitwise operations

Syntax:

integer & integer --bitwise and

integer | integer --bitwise or

integer ^ integer --bitwise xor

Note that an integer can be any expression that evaluates to an integer: an integer literal, an int32 variable reference, an arithmetic operation that evaluates to an integer, etc.

Example:

int32 first = 9 --1001

int32 second = 7 --111

int32 bitAnd = first & second --0001=1

int32 bitOr = first | second --1111=15

int32 bitXor = first ^ second --1110=14

Seen in input-6.txt

### Feature 5: integer-real number conversion

Syntax:

double-variable = integer

integer-variable = real

Note that an integer can be any expression that evaluates to an integer: an integer literal, an int32 variable reference, an arithmetic operation that evaluates to an integer, etc.

And similarly for doubles/real numbers

Example (with results indicated at the end of the line after the --):

int32 firstInt = 5

double firstDouble = firstInt --5.0

int32 multipliedInt = firstDouble \* firstDouble --25

double fraction = firstDouble/2.0 --2.5

int32 truncatedFraction = fraction --2

Seen in input-8.txt

Add more subsections accordingly if there are more advanced features implemented.

## Structure of a Test

Tests are a normal function except that they have @Test instead of the return type and they do not have any parameters. They can and should call non-test methods. Code inside of a test does not contribute towards any code coverage metrics but does contribute to data-flow metrics.

Tests **should** be placed in a separate file from the main program and the test file, if there is one, **must** be the second argument given to the compiler. Otherwise the code coverage will give unexpected results.

Example (multiply is a non-test method):

@Test t1(){

int32 first = 7

double second = 20.0

double result = multiply(first, second)

bool testPassed = result == 140.0

print(testPassed)

}

This section is closely related to **Section 2** and **Section 3**.

In this section, outline, explain, and exemplify how a typical input test should look like.

**Note**. Each of the ten examples you submit may focus on a different perspective of your test language. On the other hand, the example test(s) you give here should demonstrate as many **testing** features (i.e., syntax specific for testing) as possible that are supported by your compiler; **it can just be symbolic** (e.g., using x, y, f1, a1), but it must be syntactically and type correct.

## How a Program and a Test Should be Distinguished

See section 1.3- Structure of a Test

The previous sections outlined, explained, and exemplified how an input **program** and an input **test** should look like. In this section, discuss how a new user of your compiler should understand, both conceptually and syntactically, between writing a **program** and writing a **test**. For example, in the case of Java, a JUnit test method is just a regular Java method, with the @Test tag and using assertions, whose execution result is included in a testing report.

# Output Structure

Coverage criteria in the output HTML file includes:

Statement coverage: A line-by-line indication of which lines were executed or not across the tests and main program execution. Cyan lines indicate a function definition, green lines indicate a line was executed, and red lines indicate a line was not executed

Decision coverage: Yellow lines indicate a decision where only one of two paths was followed

TODO: Indicate what complete decision coverage looks like

P-Uses: Predicate-use coverage:

|  |  |
| --- | --- |
|  | **3.** int32 a → **6.** a<5 |

Indicates that the variable a of type int32 which was declared on line 3 was used on line 6 in the a<5 statement.

C-Uses: Computation-usage coverage

**4.** int32 b → **7.** b=1

Indicates that the variable b of type int32 which was declared on line 4 was assigned the value of 1 on line 7

Explain how the user is supposed to read/interpret the results displayed in the output HTML file(s). For examples:

* What coverage criteria are expected
* Meaning of highlights (e.g., an input program line being highlighted green vs. red)
* Any details of results that should be attended to by the user

# Justification of Output

In this section, for **each** coverage criterion supported by your compiler:

* Summarize in your own words on **what the coverage criterion means** (based on your own understanding). This should **not** be longer than a few sentences. Try to be concise and precise here.
* **Cross-reference** to the relevant output HTML file(s) and **explain** why the reported statistics is **accurate**. For example, you may write:
  + Output HTML file(s) generated from input-1.txt and input-3.txt show the statement coverage.
    - Explain why the statement coverage is measured accurately from input-1.txt. There is not a fixed format on how you structure your explanations, but it should be straightforward and convincing to the readers that your reported results are correct and accurate with respect to the coverage criterion in question. There is **no** need to make references to the Java code; focus on relating the input and output files.
    - Explain why the statement coverage is measured accurately from input-3.txt.

## Justification for the Statement Coverage Criterion

TODO

## Justification for the Decision Coverage Criterion

TODO

## Justification for the Predicate Usage Coverage Criterion

TODO

“A use of a variable is a predicate use (p-use) if the variable is in a predicate and its value is used to decide an execution path” from <https://www.cs.ccu.edu.tw/~naiwei/cs5812/st5.pdf>

## Justification for the Computation Usage Coverage Criterion

TODO

“A use of a variable is a computation use (c-use) if the value of the variable is used to compute a value for defining another variable or as an output value.” from <https://www.cs.ccu.edu.tw/~naiwei/cs5812/st5.pdf>

Add more subsections if you support more coverage criteria.

It is **optional** to give extra example in these sections for your explanations (if you judge that cross-referencing to the submitted input files and their outputs should suffices).

# Summary of Submitted Examples

* You are required to submit (at least) **10 working examples** representative of your compiler's capability.
* **This section is not meant to be long**: there is no need to repeat contents of the submitted working examples. Instead:
  + Summarize (in bullet points) what **each** of the submitted example demonstrates. Clearly indicate the **name of the input file** that should be looked at.
  + In these input files, there should also be **comments** surrounding critical lines to guide the readers.

## Highlights of Example Input 1

input-1.txt

Shows variable declaration, reference, and assignment

## Highlights of Example Input 2

input-2.txt

Shows declaration and calling of functions

## Highlights of Example Input 3

input-3.txt

Demonstrates nested if statements

## Highlights of Example Input 4

input-4.txt

Demonstrates while loops

## Highlights of Example Input 5

input-5.txt

Demonstrates if/else-if/else statements

## Highlights of Example Input 6

input-6.txt

Demonstrates bitwise operations

## Highlights of Example Input 7

input-7.txt

Demonstrates all supported relational and logical expressions

## Highlights of Example Input 8

input-8.txt

Demonstrates all supported arithmetic operations as well as integer-double type coercion

## Highlights of Example Input 9

input-9.txt

Demonstrates array assignment and access

## Highlights of Example Input 10

input-10.txt

Demonstrates uninitialized array declaration and setting array values in a while loop

## Highlights of Example Input 11

input-11.txt

Demonstrates print statements

If you decide to submit more than 10 examples, just create more sub-sections.

# Miscellaneous Features

Multiple input files:

The first file should contain the main function and any additional non-test functions. The second file should contains all test functions.

The main purpose of this project is the measurement of code coverage. 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.*

# Limitations

* 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 reported for some coverage criteria).