The University of Melbourne

SWEN90010: High Integrity Systems Engineering

Assignment 3

Due Date: 11:59pm, **Monday 26th May** 2025 (Melbourne time)

1 Introduction

The assignment is worth 20% of your total mark and is done in pairs (the same pairs as assignments 1 and 2, unless you have been re-allocated to a new pair).

The aim of this assignment is to use the SPARK Ada toolset to implement and verify the correctness and security of a small command-line Desktop Calculator utility.

That is, your pair will implement the functionality of the calculator program (as specified below) in SPARK Ada, and use the SPARK Prover to prove that it is free of runtime errors and, additionally, that it is also secure.

As usual, get started early and use your pair to maximum advantage.

Download, install and check you can run the GNAT tools (see Section 3.1) ASAP!

2 The Calculator

The program you have to implement is a command-line utility for performing numerical calculations. It takes input from the terminal (i.e. standard input, aka stdin).

2.1 Commands

Each line of input is a *command*. Commands conform to the following grammar:

<COMMAND> ::= "+"
| "-"
| "*"
| "/"
| "push1" <NAME>
| "push2" <NAME><NAME>
| "pop"
| "loadFrom" <NAME>
| "storeTo" <NAME>
| "remove" <NAME>
| "list"
| "unlock" <NUMBER>
| "lock" <NUMBER>

Tokens in the grammar above are separated by one or more *whitespace* characters, namely space, tab, and line-feed. Each <NAME> is a string of *non-whitespace* characters. <NUMBER> is a 4-digit string of non-whitespace characters that represents a non-negative number (i.e. a natural number) in the range 0000 . . . 9999.

- The calculator can be in one of two states, either *locked* or *unlocked*.
- When the user starts the calculator, they supply (via a command-line argument) a 4-digit string *masterpin* that represents a 4-digit PIN (i.e. a number in the range 0000 . . . 9999), which is the *master PIN* needed to unlock the calculator. If no master PIN is supplied, the calculator should exit immediately.
- The calculator begins in the locked state.
- For a string *pin* that represents a 4-digit PIN, the command "unlock *pin*" does nothing when the calculator is in the unlocked state. Otherwise, it checks whether *pin* is equal to the master PIN and, if so, changes the state of the calculator to unlocked. If *pin* is not equal to the master PIN, then the state of the calculator is not changed.
- For a string *newpin* that represents a 4-digit PIN, the command "lock *newpin*" does nothing when the calculator is in the locked state. Otherwise, it updates the master PIN to become *newpin* and changes the state of the calculator to locked.
- For a string *num* representing a decimal integer, e.g. "5", the command "push1 *num*" pushes the value represented by *num* onto operand stack.
- For two strings *num1* and *num2* representing decimal integers, e.g. "-9" "20", the command "push2 *num1 num2*" pushes the value represented by *num1*, followed by the value represented by *num2* onto operand stack.
- The command "pop" pops the value from the top of the operand stack, discarding it.
- The commands "+", "-", "*" and "/" each pop the top two values from the operand stack and compute the corresponding arithmetic operation on them (addition, subtraction, multiplication and division, respectively), and push the result onto the stack.
- For a string *loc* representing a memory location, the command "storeTo *loc*" pops the value from the top of the operand stack and stores it into memory location *loc*. A memory location is a positive integer starting from 1.
- For a string *loc* representing a memory location, the command "loadFrom *loc*" loads the value stored at memory location *loc* and pushes it onto the operand stack. A memory location is a positive integer starting from 1.
- The command "list" prints out all currently defined memory locations and their corresponding values.
- For a string *loc* representing a memory location, the command "remove *loc*" marks memory location *loc* as undefined (i.e. it will not be printed by subsequent "list" commands). A memory location is a positive integer starting from 1.

2.2 Notes

- The calculator only takes input from the terminal (i.e. from stdin).
- When the calculator is started, the user supplies on the command line the master PIN, as a command line argument.

- PINs are 4-digit strings in the range $0000 \ldots 9999$.
- The calculator can be in one of two states: either *locked* or *unlocked*.
- The "unlock" and "lock" commands change the state of the calculator between locked and unlocked. The "lock" command allows updating the master PIN (see above).
- PINs that are not 4-digit strings in the range $0000 \ldots 9999$ are invalid.
- Input lines (i.e. commands) longer than 2048 characters are invalid.
- When receiving invalid input, the calculator should exit immediately (possibly after printing an appropriate error message).
- When performing a calculation that would produce a result outside the range $-2^{31} \ldots 2^{31}-1$ inclusive (e.g. when computing -1×-2^{31} the calculator is allowed to do whatever it wants *except it is not allowed to raise any Ada errors* e.g. it cannot raise a `CONSTRAINT_ERROR` etc. For instance it could choose not to perform the operation, or to exit immediately, etc.
- Similarly, when performing an operation that would cause division by zero, the calculator should not raise an Ada error, but otherwise it is allowed to do whatever it wants e.g. do nothing, exit immediately etc.
- Likewise, when performing an operation that would exceed the capacity of the operand stack, the calculator is free to do whatever it wants except it cannot raise an Ada error. Similarly for an operation for which there are insufficient operands on the stack.
- The capacity of the calculator's operand stack is 512.
- Decimal integers in commands (e.g. "5" in "push 5") that are outside the range $-2^{31} \ldots 2^{31}-1$ inclusive are treated as representing 0.
- The calculator supports up to 256 memory locations. Memory locations are positive integers in the range 1 to 256 inclusive. If a command attempts to access a memory location outside this range, the calculator should not raise an Ada error, but otherwise it is allowed to do whatever it wants e.g. do nothing, exit immediately etc.
- If a command attempts to read from a memory location that is not currently defined (i.e. no value has been stored in it yet, or the value has been previously removed by using the command remover), the calculator should not raise an Ada error, but otherwise it is allowed to do whatever it wants, e.g. do nothing, exit immediately, etc.
- Any 32-bit signed integer (i.e. in the range $-2^{31} \ldots 2^{31}-1$ inclusive) can be stored in a memory location.

2.3 A Demo Session with the Calculator

The user supplies the initial master PIN when starting the application, which is called `main`. For example, to start the application setting the master PIN to "1234" the user would run (assuming you are in obj directory):

```
$ ./main 1234
```

The calculator accepts commands from the user and its prompt indicates whether it is in the locked or the unlocked state. Initially, it is always locked. Here is an example session showing its expected output for valid commands.

```
$ ./main 1234
locked>   unlock 1234
unlocked> push2 7 3
unlocked> +
unlocked> push1 5
unlocked> *
unlocked> storeTo 10
unlocked> loadFrom 10
unlocked> push1 20
unlocked> *
unlocked> storeTo 2
unlocked> list
   2 =>        1000
   10 =>         50
unlocked> pop
unlocked> remove 10
unlocked> list
   2 =>        1000
unlocked> lock 2345
locked>
```

3 Your tasks

Get started early. This assignment is worth 20 marks in total.

3.1 Task -1: Download and Install GNAT Community Edition 2019

Download and install GNAT Community Edition from: `https://www.adacore.com/download`.

Ensure that the `bin/` directory is in your `PATH` so that you can run the Ada tools directly. If your setup is correct, you should be able to run commands like `gnatmake` and `gnatprove` and see output like the following (noting that in this case the commands were run on MacOS):

```
$ gnatmake --version
GNATMAKE Community 2019 (20190517-83)
Copyright (C) 1995-2019, Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

$ gnatprove --version
2019 (20190517)
Why3 for gnatprove version 1.2.0+git
```

```
/Users/hira/opt/GNAT/2019/libexec/spark/bin/alt-ergo: Alt-Ergo version 2.3.0
/Users/hira/opt/GNAT/2019/libexec/spark/bin/cvc4: This is CVC4 version 1.7.1-prerelease
/Users/hira/opt/GNAT/2019/libexec/spark/bin/z3: Z3 version 4.8.0 - 64 bit
```

3.2 Task 0: Downloading and Building the Helper Code

You need to implement the calculator in SPARK Ada. However you are provided with some helper code to get you started.

Download the ZIP file containing the helper code from the LMS. It contains a small number of Ada packages:

- `main.adb`: a top level main program that shows basic usage of the other packages. You will replace this with the top level of your calculator implementation.
- `MyCommandLine`: A small wrapper around Ada's built-in `Ada.Command_Line` package. This wrapper provides a simplified interface for accessing command-line arguments in a SPARK-compatible way. For more information, see the files `mycommandline.ads` and `mycommandline.adb`.
- `MyString`: Provides a simple SPARK-compatible abstract data type for strings. It is used for representing lines of input as well as variable names, command names, etc. MyString avoids dynamic-length strings and supports safe, bounded operations that are verifiable in SPARK. For more details, see the files `MyString.ads` and `MyString.adb`.
- `MyStringTokeniser`: Provides a simple SPARK interface for tokenising strings. By "tokenising" we mean to break a string up into its various whitespace-separated tokens (where each token contains no whitespace).
- `MemoryStore`: Provides a simple SPARK-compatible API for the calculator's memory. It manages up to 256 numbered locations (1 – 256), letting you `put`, `get`, `remove` and `list` 32-bit signed integers stored at those locations.
- `PIN`: Provides a simple SPARK abstract data type to represent 4-digit PINs in the range 0000 . . . 9999.
- `StringToInteger`: Provides a simple SPARK interface for parsing tokens that represent decimal integers and converting them to `Integer`s. Note that if supplied an integer outside the range of $-2^{31} \ldots 2^{31} - 1$ inclusive (e.g. if supplied with the string "2147483648" (2^31), the `Integer` 0 is returned.

Besides `main.adb` don't modify the other supplied code, excpet for adding comments.

After unpacking the ZIP file, it will create the directory `assignment3` in which the Ada code is placed. You can build the code by running '`gnatmake` -P default.gpr' in that directory.

```
$ gnatmake -P default.gpr
Compile
   [Ada]          main.adb
   [Ada]          mycommandline.adb
   [Ada]          mystring.adb
```

```
   [Ada]          mystringtokeniser.adb
   [Ada]          pin.adb
   [Ada]          stringtointeger.adb
   [Ada]          variablestore.adb
Bind
   [gprbind]      main.bexch
   [Ada]          main.ali
Link
   [link]         main.adb
```

Note: if you are building the code on MacOS, you might get the warning message:

```
ld: warning: URGENT: building for OSX, but linking against dylib
(/usr/lib/libS ystem.dylib) built for (unknown). Note: This will be
an error in the future
```

This warning can be safely ignored.

Building the code should produce the placeholder `main` in obj folder that you can then run. As mentioned, the supplied main code simply shows some examples of how to use the other supplied packages.

3.3 Task 1: Understanding `MyStringTokeniser` (3 marks)

Your first task is to understand the `MyStringTokeniser` package. This package has no comments but it does have SPARK annotations which describe aspects of its central procedure `Tokenise`.

You should read and modify the provided `main.adb` program: in particular the parts of it that use the `MyStringTokeniser` package. This will help you to get an idea of what this package is doing.

You should then carefully read the SPARK annotations on the `Tokenise` procedure.

You need to:

1. Add comments to the file `mystringtokeniser.ads` describing each part of the postcondition of the `Tokenise` procedure. For each part of its postcondition, you need to describe what that part is saying and why it is necessary to have it as part of the postcondition.

   Hint: think about code that uses this package. Try removing parts of the postcondition and run the SPARK prover over code that uses this package to see what happens.

2. Add comments to `mystringtokeniser.adb` that explain the loop invariant for the `Tokenise` procedure and, in particular, why the following part of the loop invariant is necessary:

   ```
   (OutIndex = Tokens'First + Processed);
   ```

3.4 Task 2: Implementing the Calculator (6 marks)

Using the provided code, implement the calculator as specified in this document.

Your implementation should follow good software engineering practices regarding modularity, information hiding / abstraction, loose coupling, and so on.

Therefore, you are strongly encouraged to decompose the core operations of the calculator into a separate Ada package that your `main.adb` can make use of.

3.5 Task 3: Proving it Free of Faults (5 marks)

Your next task is to use the SPARK Prover to prove that your calculator is free of runtime errors, and that all pre-conditions on the provided code are always satisfied.

To do this, you will likely need to write loop invariants, and add defensive checks to your code.

Your goal is to have a working calculator implemented that, when somebody selects *SPARK* → *Prove All* in GPS, does not produce any warning or error messages from the SPARK Prover.

To get full marks here, your code must of course avoid things like integer overflow. However it should not avoid performing legitimate operations that do not cause overflow.

3.6 Task 4: Proving it Secure (6 marks)

Your final task is to use the SPARK Prover to prove that your implementation is secure. By "secure" we mean it should *at least* satisfy the following security properties (however this list is intentionally not complete—to get full marks here you will also need to think of and prove additional security properties):

- The arithmetic operations ("+", "-", "*", ""), load, store, remove, and lock operations can only ever be performed when the calculator is in the unlocked state.
- The Unlock operation can only ever be performed when the calculator is in the locked state.
- The Lock operation, when it is performed, should update the master PIN with the new PIN that is supplied.

By "performed" we mean that the operation has executed. This is different to an operation *attempting* to be executed. For instance, in the following example session the user attempts to perform the Lock operation; however the operation is *not* executed, because the caculator was in the locked state.

```
$ ./main 1234
locked>   lock 2345
Already locked
locked>
```

For this task you need to **write a short description (no more than one page)** describing:

- The security properties that you proved of your implementation and, for each,

- How you specified the security property using SPARK annotations and how the annotations encode the security property.

If you prove additional properties of your implementation, besides those mentioned above, you should document those in your report.

Your report should be written as Ada comments at the top of your `main.adb` file.

4 Submission

You should submit a ZIP file containing your SPARK code. Your code should live inside the `assignment3/` directory in your ZIP file. We expect to be able to build and run your code by doing the following from the command line, assuming your submission is called `submission.zip`:

```
$ unzip submission.zip
$ cd assignment3/
$ gnatmake -P default.gpr
$ cd obj
$ ./main 1234
```

Your `main.adb` should inclue comments that clearly identify *all* authors in your pair.

Your code should build and run against the original packages you were supplied with. However your submission should contain updated versions with (only) additional comments added, as required (see above).

Late submissions Late submissions will attract a penalty of 2 marks for every day that they are late. If you have a reason that you require an extension, email Hira *well before the due date* to discuss this.

Please note that having assignments due around the same date for other subjects is not sufficient grounds to grant an extension. It is the responsibility of individual students to ensure that, if they have a cluster of assignments due at the same time, they start some of them early to avoid a bottleneck around the due date. The content required for this assignment was presented before the assignment was released, so an early start is possible (and encouraged) .

5 Academic Misconduct

The University misconduct policy applies to this assignment. Students are encouraged to discuss the assignment topic, but all submitted work must represent the pair's understanding of the topic.

The subject staff take plagiarism very seriously. In the past, we have successfully prosecuted several students that have breached the university policy. Often this results in receiving 0 marks for the assessment, and in some cases, has resulted in failure of the subject.