The University of Melbourne

SWEN90010: High Integrity Systems Engineering

Assignment 3

Due Date: 11:59pm, **Friday May 24th** 2024 (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 a small decompression library.

That is, your pair will implement the functionality of the library (as specified below) in SPARK Ada, and use the SPARK Prover to prove that it is free of runtime errors. You will also learn how design by contract can help produce safe, efficient code.

Imagine somebody malicious sends you a compressed file (e.g. an image or a ZIP file) which you subsequently decompress (e.g. by viewing the image or unzipping the ZIP file). How do you know that the file won't trigger a bug in the decompression program (e.g. a stack buffer overflow) that allows the sender to run arbitrary code on your machine, thereby installing malware? Indeed, such bugs have been common in compression programs.

We will use SPARK Ada in this assignment to implement decompression routines and prove the absence of runtime errors, including array index out of bounds that lead to stack buffer overflows, as well as integer overflows and other runtime errors.

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 Library

The library you have to implement is for decompressing data that has been compressed using the LZ77 algorithm. You are provided with a package specification (.ads file) and a partial package implementation (.adb file). You will have to extend and complete the implementation (.adb file).

2.1 LZ77 Decompression

The LZ77 compression algorithm is an example of a non-trivial early compression algorithm that has influenced the design of many modern compression formats. There are many different ways to compress data in the LZ77 format. However, LZ77 *decompression* is quite straightforward and is therefore the focus of this assignment.

There are many useful online resources that explain LZ77. I recommend this one: `https://towardsdatascience.com/how-data-compression-works-exploring-lz77-3a2c2e06c097`

Data compressed in the (original) LZ77 format is a sequence of *tokens*, where each token contains three pieces of information:

- A non-negative *offset*;
- A non-negative *length*;
- A "next" byte

For normally compressed data, the offset and length in the first token are always the value 0, and the "next" byte in the first token is the first byte of the data that was compressed.

Subsequent tokens can refer to data that has already been decompressed and instruct the decompression algorithm that the next bytes in the decompressed data should be obtained by copying from earlier data that has already been decompressed: the offset indicates where in the decompressed data stream is being referred to, relative to the current position, and the length indicates how many bytes in the decompressed stream are being referred to.

Suppose we are decompressing some data whose length, once decompressed, is n bytes. At some point during decompression we have already decompressed k bytes, for some $k \geq 0$ and $k < n$. Let us label the bytes that were originally compressed (and that we will obtain after decompression is complete) as $b_1, \ldots, b_n$. Suppose the next token is (o, l, c), i.e. its offset is the value o, its length is l, and its next byte is c. Then this means that the bytes $b_{k+1}, \ldots, b_{k+1+(l-1)}$ are identical to the bytes $b_{k+1-o}, \ldots, b_{k+1-o+(l-1)}$, and that byte b_{k+1+l} is c. Therefore decompression works by doing a byte-by-byte copy (i.e. one-byte-at-a-time) copying l bytes starting at b_{k+1-o} to form the next l bytes in the decompressed data; byte c is then the byte that comes after these l bytes.

Note that l is permitted to be larger than o.

As an example suppose the compressed data was the two tokens $(0, 0, `a')$, $(1, 2, `b')$.

Initially no data has been decompressed, i.e. $k = 0$. The first token says copy 0 bytes from earlier in the decompressed data, and then the next byte is 'a', so 'a' is the first byte in the decompressed data, i.e. $b_1 = `a'$.

Now $k = 1$ since we have decompressed 1 byte. The second token says copy two bytes from one byte earlier in the decompressed data, i.e. byte b_1 is copied to make byte b_2, and byte b_2 is copied to make byte b_3. Therefore, $b_2 = `a'$ and $b_3 = `a'$. The next byte is 'b' so $b_4 = `b'$. Therefore the decompressed data is the string "aaab".

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

3.2 Task 0: Downloading and Building the Helper Code

Download the ZIP file containing the helper code from the LMS. You are provided with a package specification (.ads) file for the small library that you will implement: `lz77.ads`. You are also provided a partial implementation (.adb) file: `lz77.adb`. You are also provided with a project file: `default.gpr`. Finally you are provided a simple "main" program that shows some example usage of the (mostly unimplemented) library routines: `main.adb`.

The helper code also includes a file `big_integers.ads` and the directory `proof/_theories/` which contains a file `big_integers.mlw` that allow the use of mathematical integers in specifications and proofs (and in ghost code) but not in actual, runnable SPARK Ada code. The type `Big_Integer` refers to these mathematical integers.

Do not modify any of the provided files except the package body file: `lz77.adb` and the main driver: `main.adb` (the latter you may wish to modify to test the code you write).

After unpacking the ZIP file, it will create the directory `assignment3` in which the Ada code is placed. You can build the code by using GPS as tutorials shown or running '`gnatmake` main' in that directory.

```
$ gnatmake main
gcc -c main.adb
gcc -c lz77.adb
gnatbind -x main.ali
gnatlink main.ali
```

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` that you can then run. As mentioned, the supplied main code simply shows some examples of how to use the library; but since the library hasn't yet been implemented (you will do this), it doesn't do anything useful.

You should read through the package specification file carefully to make sure you understand each part of it.

Hint: check the discussion board and ask if you are not sure about something.

When running the SPARK Prover over the provided code, you will need to set the Proof Level to at least 1 (the default is 0). See subsubsection 3.2.1.

3.2.1 Setting the Proof Level

This can be done in GNAT Programming Studio: after choosing one of the "Prove" options from the *SPARK* menu (e.g. *Prove File* or *Prove All* etc.) a dialog box is displayed with a drop-down to allow setting the Proof Level, before pressing the *Execute* button to run the prover.

3.3 Task 1: Implementing `Decode` (4 marks)

Your first task is to implement the `Decode` procedure, for decompressing a sequence of tokens `Input` to produce uncompressed bytes (characters) in the output array `Output`.

Note that it does not have a precondition. Therefore, `Decode` should run without raising any runtime errors (e.g. without indexing an array out of bounds, or causing integer overflow, etc.) no matter what input it is given.

Consider the invalid sequence of tokens: $(0, 0, \text{`}a\text{'})$, $(2, 1, \text{`}b\text{'})$. Attempting to process the second token would cause the output array `Output` to be indexed out of bounds: after producing a single output byte 'a' (so `Output(Output'First) = 'a'` or using the notation above, $b_1 = \text{`}a\text{'}$), the second token says that the subsequent byte $b_2$ is equal to the non-existent byte $b_0$. In particular this would cause `Decode` to attempt to read `Output(Output'First-1)` (to copy this non-existent byte to `Output(Output'First+1)`) thereby indexing the `Output` array out of bounds.

Therefore, your implementation of `Decode` will have to perform certain checks on the compressed data tokens as it is decoding each one to ensure that it does not cause runtime errors. If your implementation detects a token which if decoded would cause a runtime error, then it should set the `Error` output parameter to `True`, and the `Output_Length` output parameter to 0 and return, without attempting to decode any further.

When decoding all tokens completes successfully, `Output_Length` should be set to the number of bytes in the decompressed output and `Error` should be set to `False`.

You should add comments to your implementation, especially to explain the checks you have added to ensure the absence of runtime errors.

To obtain full marks here your implementation should be as clean as possible without unnecessary checks. In particular, it should only contain checks needed to prevent runtime errors. Reading uninitialised data from the `Output` array does not cause a runtime error, since the `Output` parameter has the `in out` mode.

Hint: Note that some strange inputs will not cause runtime errors. For example, $(0, 2, \text{`}A\text{'})$, $(0, 1, \text{`}B\text{'})$ *should produce '*$X_1X_1AX_3B$*' where each '*X_i*' stands for the ith byte of data in the Output array that is passed to Decode, i.e. this is an example input in which uninitialized data in the Output array ends up in the final decompressed output. While this behaviour is strange, it doesn't cause any runtime errors and so it is allowed for the purpose of this assignment.*

3.4 Task 2: Proving `Decode` (4 marks)

Your next task is to use the SPARK Prover to prove that `Decode` is indeed free of runtime errors and meets its specification from `lz77.ads`.

Hint: you will almost certainly have to write a suitable loop invariant.

Remember, when running the SPARK Prover over the provided code, you will need to set the Proof Level to at least 1 (the default is 0). See subsubsection 3.2.1. If your proofs require the Proof Level to be set to something greater than 1, you should document this at the top of your package body file that you submit.

To obtain full marks here the SPARK Prover needs to report that `Decode` does indeed meet its specification (without warnings or errors) *non-trivially*: i.e. a trivial implementation that always sets `Error` to `True` and `Output_Length` to 0 without doing anything will not score well.

3.5 Task 3: Guarantees (2 marks)

Add comments above your implementation of `Decode` that explain:

1. Because your implementation of `Decode` is written in Ada, *even if it was not proved safe* (i.e. even if you didn't do Task 2, or if your implementation has a bug), what guarantees does implementing `Decode` in Ada provide, in comparison to an implementation in an unsafe language, like C? For example, what situations could your Ada implementation be used in that a (potentially buggy) C implementation should not be used in?

2. Because it is proved safe, i.e. *assuming you successfully completed Task 2, even if you didn't complete that task*, what additional guarantees does that provide? For example, what situations could an implementation that is proved safe according to Task 2 be used in that a (potentially buggy) Ada implementation should not be used in?

3.6 Task 4: Implement and Verify `Is_Valid` (4 marks)

The `Is_Valid` function is meant to check all the tokens in the compressed data to ensure that there is no possibility of a runtime error occurring while decoding them, plus no uninitialised data will be copied, assuming that the output buffer is large enough to hold all the decompressed data.

Your next task is to implement this function and verify that it meets its specification. In particular, this function should return `True` if and only if the predicate `Valid` holds, which formally specifies when input data can be decompressed without runtime errors and no uninitialised data will be copied(assuming the output buffer is large enough).

The `Valid` predicate is implemented as a *ghost* function that returns a `Boolean`. Recall that ghost functions are functions that can only be used in specifications—they are not real code and instead are used only in specifications (e.g. in pre/postconditions and loop invariants and `Assert` pragmas). The `Valid` predicate makes use of the `Length_Acc` ghost function that formally specifies what the decompressed length is for a compressed token sequence as an array of partial sums. See Lecture 16 for more explanation of these concepts.

Hint: you will almost certainly have to write a suitable loop invariant.

As with `Decode` your implementation of `Is_Valid` should be as clean and simple as possible, without redundant or unnecessary checks, and the SPARK Prover has to report that it meets its specification without warnings or errors.

Remember, when running the SPARK Prover over the provided code, you will need to set the Proof Level to at least 1 (the default is 0). See subsubsection 3.2.1. If your proofs require the Proof Level to be set to something greater than 1, you should document this at the top of your package body file that you submit.

3.7 Task 5: Implement and Verify `Decode_Fast` (4 marks)

Your next task is to implement and verify the `Decode_Fast` procedure. This procedure assumes in its precondition that the compressed data `Input` can be decoded without runtime errors (including that the output buffer `Output` is large enough and that the `Input` tokens have been checked by the `Is_Valid` function). Therefore it has no need for the `Error` output parameter, because when its precondition is satisfied there is no way for decoding to cause a runtime error.

For the same reason, there is no need for this procedure to perform checks on the compressed tokens as it is decoding them. Its job is to *blindly* decompress the data without doing any validity checks: it is allowed to do that because its precondition ensures that (in a program that the SPARK Prover says is valid) it can only ever be used in situations in which decoding cannot cause runtime errors anyway.

In this sense, this procedure shows how we can take advantage of formal verification and design-by-contract to help us write code that is not only free of errors but also *more efficient* than defensive implementations like the original `Decode`.

To obtain full marks here your implementation of `Decode_Fast` should be as clean and simple as possible, without unnecessary checks on the input tokens, and the SPARK Prover has to report that it meets its specification without warnings or errors.

Hint: you will almost certainly have to write a suitable loop invariant.

Remember, when running the SPARK Prover over the provided code, you will need to set the Proof Level to at least 1 (the default is 0). See subsubsection 3.2.1. If your proofs require the Proof Level to be set to something greater than 1, you should document this at the top of your package body file that you submit.

3.8 Task 6: More Guarantees (2 marks)

Add comments above your implementation of `Decode_Fast` that explain:

The postcondition of Decode_Fast only talks about a part of the requirements of this function. Compared to Section 2.1, what conditions are missing? If you were asked to add those conditions and then verify them, what could be the main challenges?

4 Submission

You should submit just your `lz77.adb` file. Use comments at the top to indicate the names of *all* authors in your pair.

We will run your code and proofs in GNAT Community Edition (GNAT Programming Studio) 2019.

Your code and proofs should build and run against the original packages you were supplied with.

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 Toby *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.