CTEC3904 FUNCTIONAL SOFTWARE DEVELOPMENT ASSIGNMENT

Lampros Karseras

P2424629

Contents

Task 1: Explanation of insertSubsection method

```
def insertSubsection(after: List[Int], newSection: Section): Section =
  after match {
    case List(m: Int) if m >= 0 && m <= subsections.length =>
      subsections.splitAt(m) match {
        case (left, right) => new Section(heading, content, left ++
          (newSection :: right))
      }
    case m :: n :: ps if m > 0 && m <= subsections.length =>
      subsections.splitAt(m - 1) match {
        case (left, s :: right) =>
          new Section(heading, content, left ++
            (s.insertSubsection(n :: ps, newSection) :: right))
      }
    case _ => bad(s"trying to insert after invalid position ${fromSectionNumber(after)}")
  }
```

Definition of method

The "insertSubsection" method takes two variables as arguments: the "after" and the "newSection" variables. The "after" variable is a List of integers while the "newSection" variable is of type Section. The return value of the method will also be of type Section.

The Section type is a data structure defined by the class Section, and it is essentially a tree. The root node is what we going to call "Chapter" and the leaf nodes are the "SubSections". Each Section consists of a heading (which is a String giving the Section a name), the content (which is a List of Strings, the data of this Section) and the subsections (which is a List of Sections, the leaf nodes of this Section).

Under the parent object Document, the implicit method "toSectionNumber" method is defined, which does an implicit conversion of String => List[Int]. This brings the "toSectionNumber" method into scope of main function, Section class and Book class, and therefore passing a String in the form of "1.4" to "after" variable will implicitly convert it into a List of integers [1,4], since the implicit methods are called automatically by the compiler when it assumes is a good idea to do so to make the code compile. In this case, passing a String to "after" variable while expects a List[Int] and having the implicit method "toSectionNumber" in scope with the definition of String => List[Int], it is called automatically by the compiler.

Explanation of Recursion and Pattern Matching

The "insertSubsection" method is a recursive function using pattern matching. As an example, the method will be invoked with the following call from "main" using the debugger:

```
println(level4.insertSubsection("1.1.0", testSection))
```

Figure 1 shows the initial stack, which consists of:

1. "this": The instance of "level4" which has all the first-year modules. It consists of two subsections ("First term", "Second term"), which in turn are consists of four subsections (the modules of the term) each.

2. “after”: The variable which was converted due to the implicit conversion explained above now as a List[Int] = (1,1,0).
3. “newSection”: The test Section to be inserted.

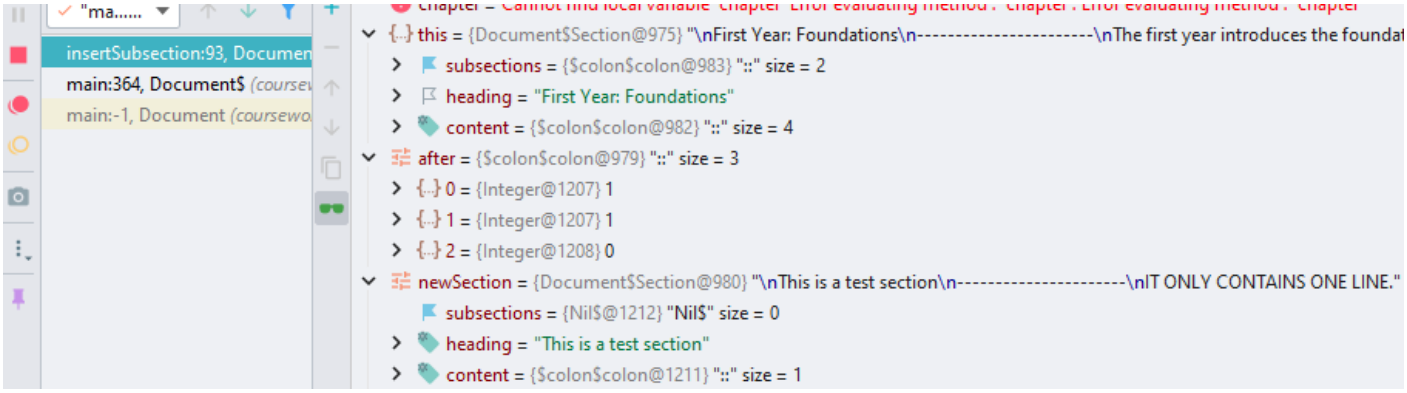

Figure 1: Initial Stack

“insertSubsection” method is using pattern matching on the “after” variable. Three cases are used:

- The first case is the “Base” case which terminates the recursion and unwinds the stack.
- The second case is the recursive call.
- The general case is the “anything else” case which catches undefined cases.

As the “after” variable has more than one element, the “base” case is skipped. This is because the first case has this definition:

```
case List(m: Int) if m >= 0 && m <= subsections.length =>
```

will only return true and execute the body if the “after” variable, which is getting pattern matching, has a single element and the guard, the “if” statement”, also returns true.

The second case:

```
case m :: n :: ps if m > 0 && m <= subsections.length =>
```

on the other hand, is satisfied and will execute its body. This is because, if the “base” case, which is more restrictive and requires just a single element in the List of “after” variable, was skipped, that means that the List contains more than one element. The second case deconstructs the “after” List, taking the head (first element) as “m”, then the next element in the List as “n”, then anything else that remains, the tail of the List (in this case a List(0)). The guard then is checked, m is indeed greater than zero and less than or equal to length of subsections, which is two as shown in Figure 1.

Next the body of second case is executed:

```
subsections.splitAt(m - 1) match {
  case (left, s :: right) =>
    new Section(heading, content, left ++
      (s.insertSubsection(n :: ps, newSection) :: right))
}
```

which is another pattern matching. “splitAt” is method belongs to the value member of the class List. It returns a pair of Lists consisting of the first elements until the split, and then the rest.

As shown in Figure 2, the split created two lists which are pattern matched as “left” and “right”. The case is deconstructing the List of Lists and taking the head of the “right” List. The subsections are

split at “m - 1” (in this case 1-1). This is because the insertion of the new Section should happen after the given Section number (“1.1.0”) and to be able to get the correct Section, it is moved as the head of the “right” List after the split. The “left” List is empty since the split happens at position 0. The “s” (head of “right” List) is now the desired Section (the “First Term”).

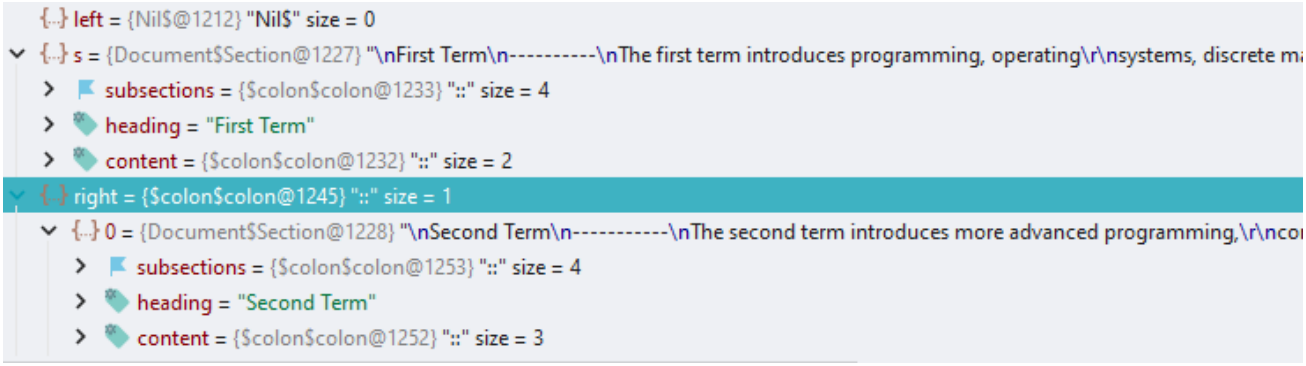

Figure 2: Stack of 2nd case

After the pattern matching, a new Section is created since the data structure is immutable. This Section has the same heading and content (in this first recursion, the “First term” heading and content), but the subsections is a recursive call to the same “insertSubsection” method.
The “left” split List is prepend using the “++” method (which is alias for concat, the same as “:::”).
The “s” (the head of “right” List which is the “First term”) is then used as the input structure to recursively call the “insertSubsection” method.

```
(s.insertSubsection(n :: ps, newSection) :: right))
```

The “after” variable now consists of “n” and “ps” (n=1, p=0) which is List(1,0). The “right” split List (“Second Term”) is then appended (cons) to the result of the recursion.

After that, the “insertSubsection” method is called again, but this time with different input and “after” variable (the newSection is the same) as shown in Figure 3.

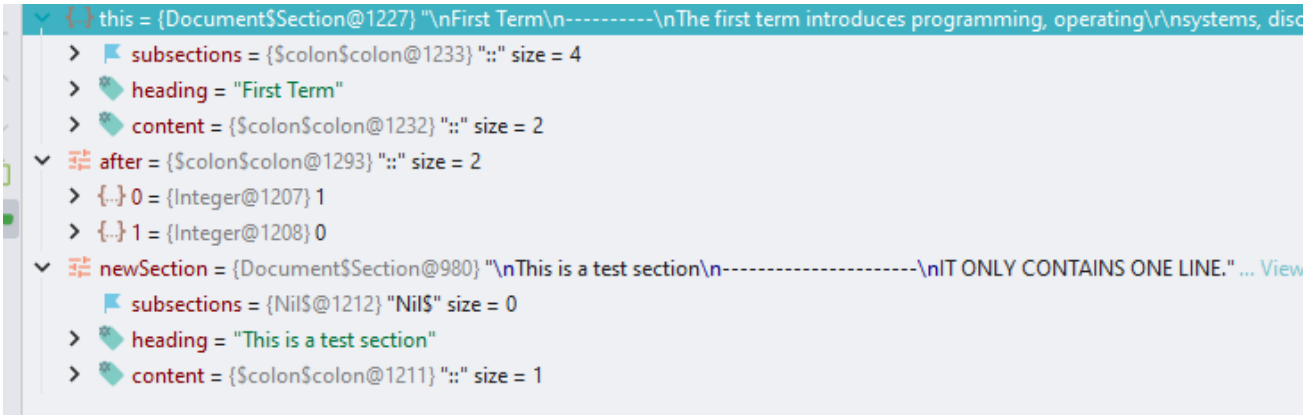

Figure 3: 1st recursive call

“this” now is the “First term” which has four subsections, the 4 modules.
“after” consists only of two elements, List(1,0)
The first case again it is not satisfied, and therefore it is skipped.
The second case again is executed. The exact same process as before happens and we have another recursive call to “insertSubsection” method, this time with the following:

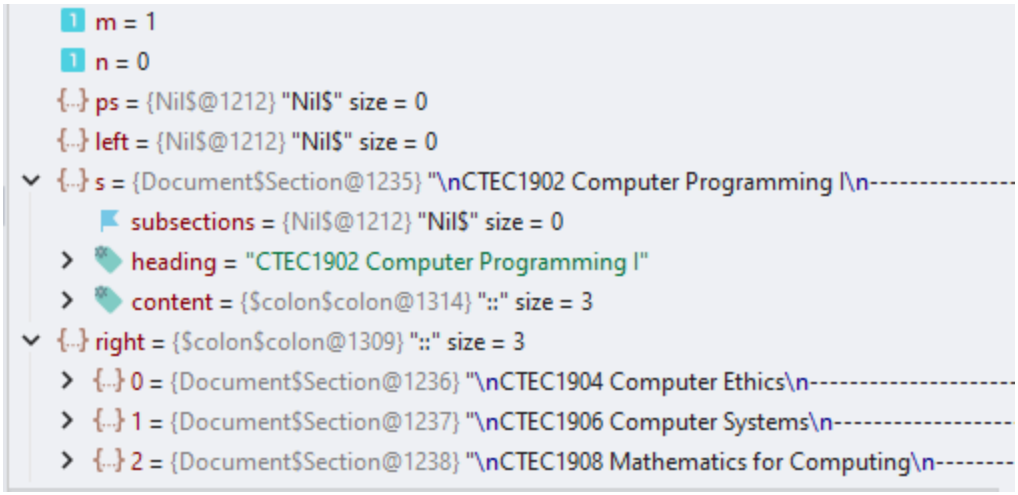

Figure 4: 1st recursion 2nd case

- "m" is again 1 and "n" is now 0. "ps" is Nil since there are not anymore elements in the "after" variable.
- "s", the head of the "right" List, is now the subsection of "first term", the module "CTEC1902 Computer Programming I".
- The "right" split List is all the other subsections of "first term" (the three remaining modules). This will be appended to the result of the final recursion.
- The "insertSubsection" method is recursively called with the "after" variable as "n" + "ps", which is a List(0).

Finally, the recursive call now satisfies the first case, as shown in Figure 5. This is because now "after" variable is just a List with a single element, 0, in it. The guard is also satisfied since the element 0 is indeed greater than or equal to 0 and the subsections of "CTEC1902 Computer Programming I" subsection is indeed less than or equal to 0, since "CTEC1902 Computer Programming I" is the end leaf of the tree (does not have any subsections). This is shown in Figure 6

```
def insertSubsection(after: List[Int], newSection: Section): Section =   newSection: "\nThis is a
  after match {   after: "::" size = 1
    case List(m: Int) if m ≥ 0 && m ≤ subsections.length ⇒   m: 0    m: 0
      subsections.splitAt(m) match {
        case (left, right) ⇒ new Section(heading, content, left ++
          (newSection :: right))
      }
    case m :: n :: ps if m > 0 && m ≤ subsections.length ⇒
      subsections.splitAt(m - 1) match {
        case (left, s :: right) ⇒
          new Section(heading, content, left ++
            (s.insertSubsection(n :: ps, newSection) :: right))
      }
    case _ ⇒ bad( reason = s"trying to insert after invalid position ${fromSectionNumber(after)}")
  }
```

Figure 5: Final Recursion

The "base" case now splits the subsections at "m", since now we are at the desired subsection and the newSection should be inserted after the given Section number. The "left" split is the given Section number and that is why the case creates a new Section which prepends the "left" split and concat it with the result of newSection appended to "right" split.

This also means that the "CTEC1902 Computer Programming I" Section, which did not had any subsections, now has the newSection as a subsection.

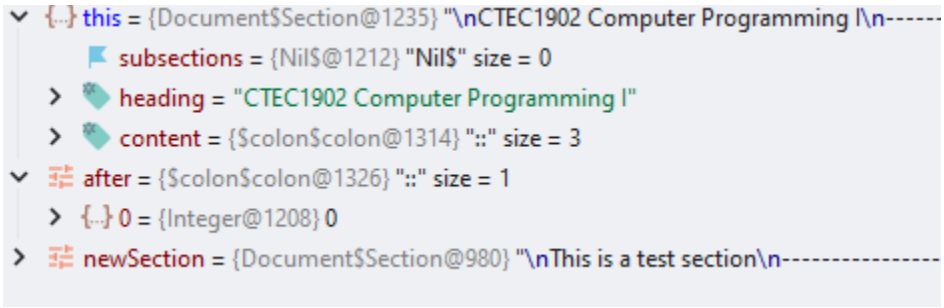

Figure 6: Final Recursion Stack

Now that the "base" case is done, the stack is unwind making all the changes to the data structure, which is to create an identical data structure with the given newSection inserted at the correct position.

As show in Figure 7, after the "base" case finishes, the stack starts to unwind by jumping back to the previous recursive call. This is repeated until the stack is empty and then the println() is called to display the results, Figure 9.

```
def insertSubsection(after: List[Int], newSection: Section): Section =   after: "::" size = 2    new
  after match {   after: "::" size = 2
    case List(m: Int) if m ⩾ 0 && m ⩽ subsections.length ⇒
      subsections.splitAt(m) match {
        case (left, right) ⇒ new Section(heading, content, left ++
          (newSection :: right))
      }
    case m :: n :: ps if m > 0 && m ⩽ subsections.length ⇒   m: 1    m: 1    ps: "Nil$" size = 0
      subsections.splitAt(m - 1) match {
        case (left, s :: right) ⇒   right: "::" size = 3    right: "::" size = 3    s: "\nCTEC1902
          new Section(heading, content, left ++
            (s.insertSubsection(n :: ps, newSection) :: right))   newSection: "\nThis is a test sect
      }
    case _ ⇒ bad( reason = s"trying to insert after invalid position ${fromSectionNumber(after)}")
  }
```

Figure 7: Unwind the Stack

As we can see in the Figure 9, the newSection is inserted at 1.1.0, or as a subsection of

"First Term"->"CTEC1902 Computer Programming I".

Since "CTEC1902 Computer Programming I" did not had any other subsections, newSection is the only one at 0.

A diagram showcasing the trace of the recursion is shown in Figure 10.

Critical Evaluation

One critical point that can be mentioned is, the way the method deals with errors. The "general" case catches all the errors, but for example in Figure 8, if we call the method with

```
println(level4.insertSubsection("1.5", testSection))
```

that causes the whole "First Term" to be replaces with a single "bad Section". That is because after the first call and going to first recursion, "this" is the "First Term" and "m"=5 which does not satisfy any of the two cases and therefore drops to "general" case. This in turn replaces "this" with the "bad" Section. If it is called like:

```
println(level4.insertSubsection("5", testSection))
```

"bad" Section will replace the level4. If it is call with "1.1.1" it will replace the "CTEC1902 Computer Programming I" Section.

It might be more sensible to explore other Scala error handling, like "Option", "Try" or "Either".

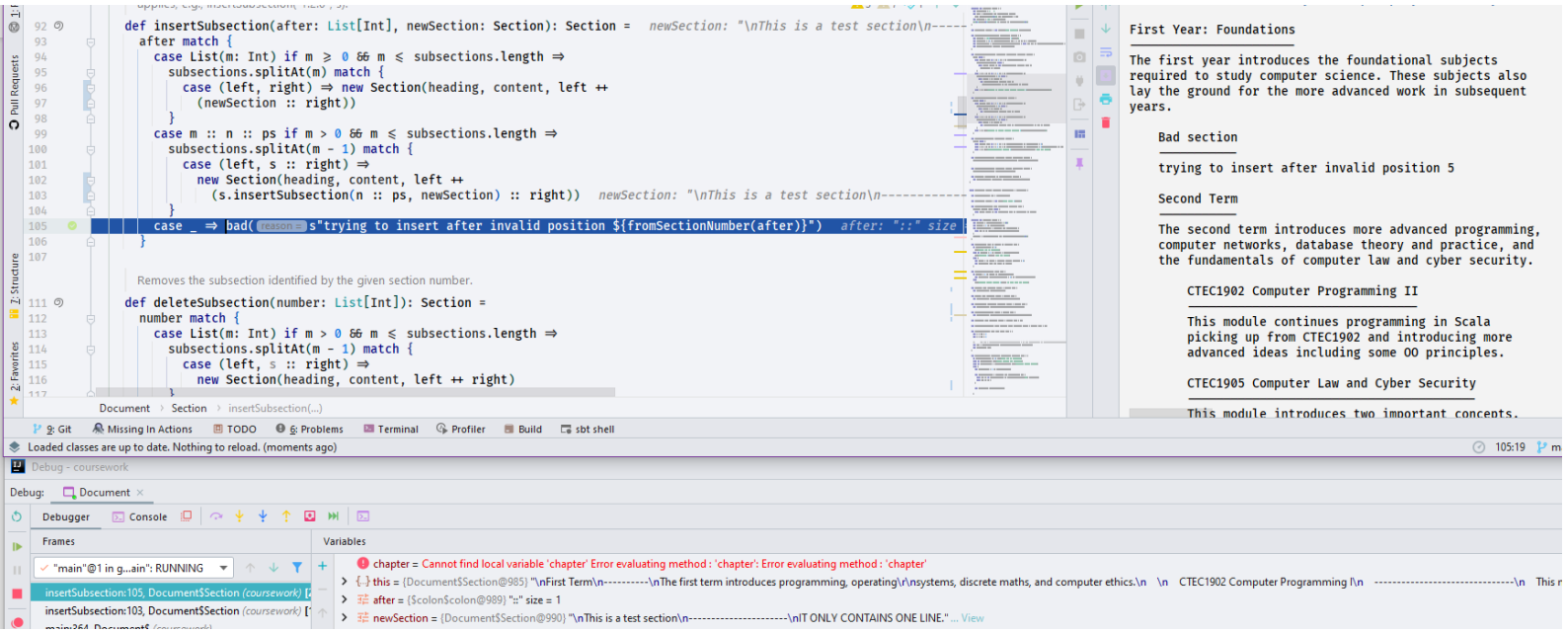

Figure 8: Error handling

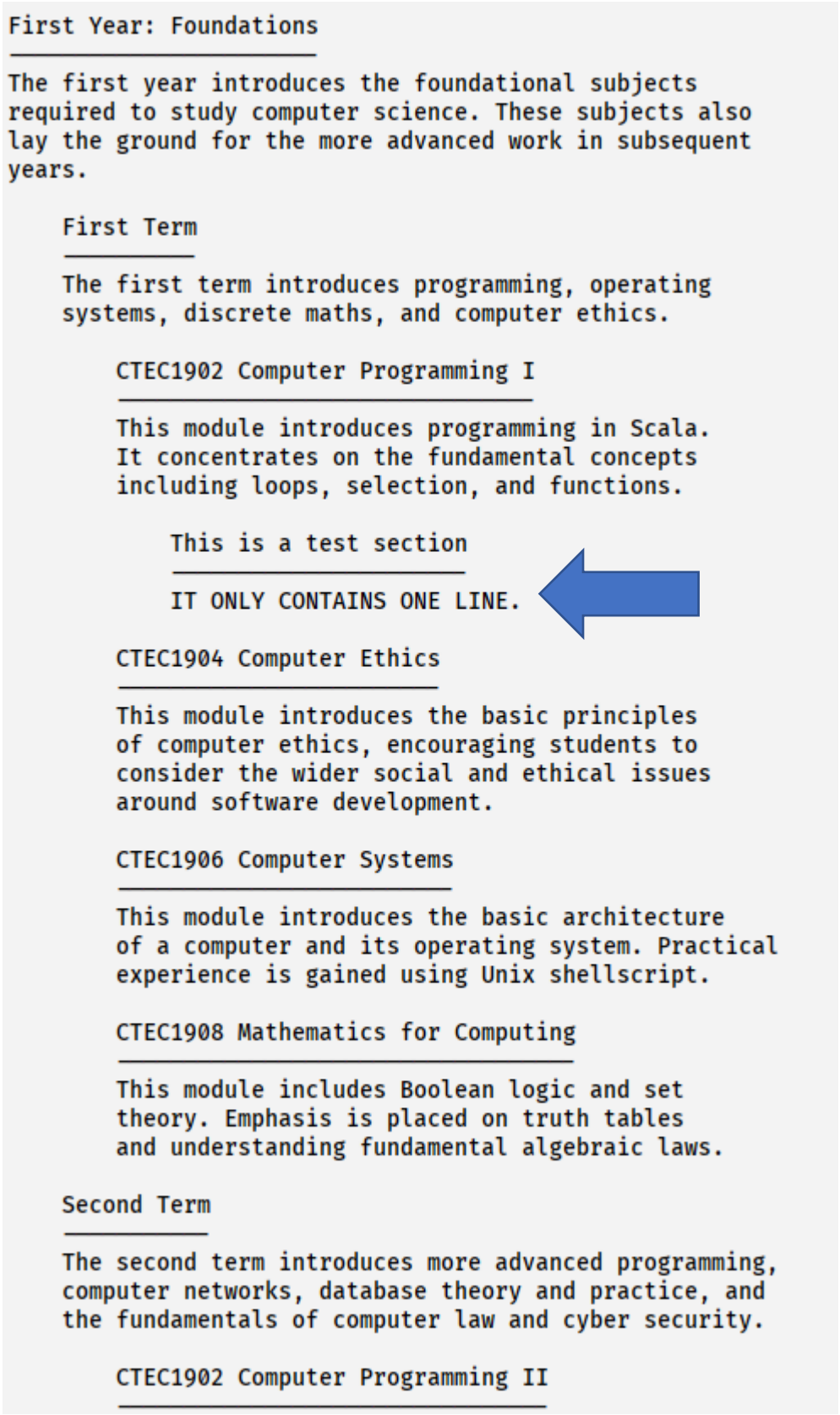

Figure 9: Results.

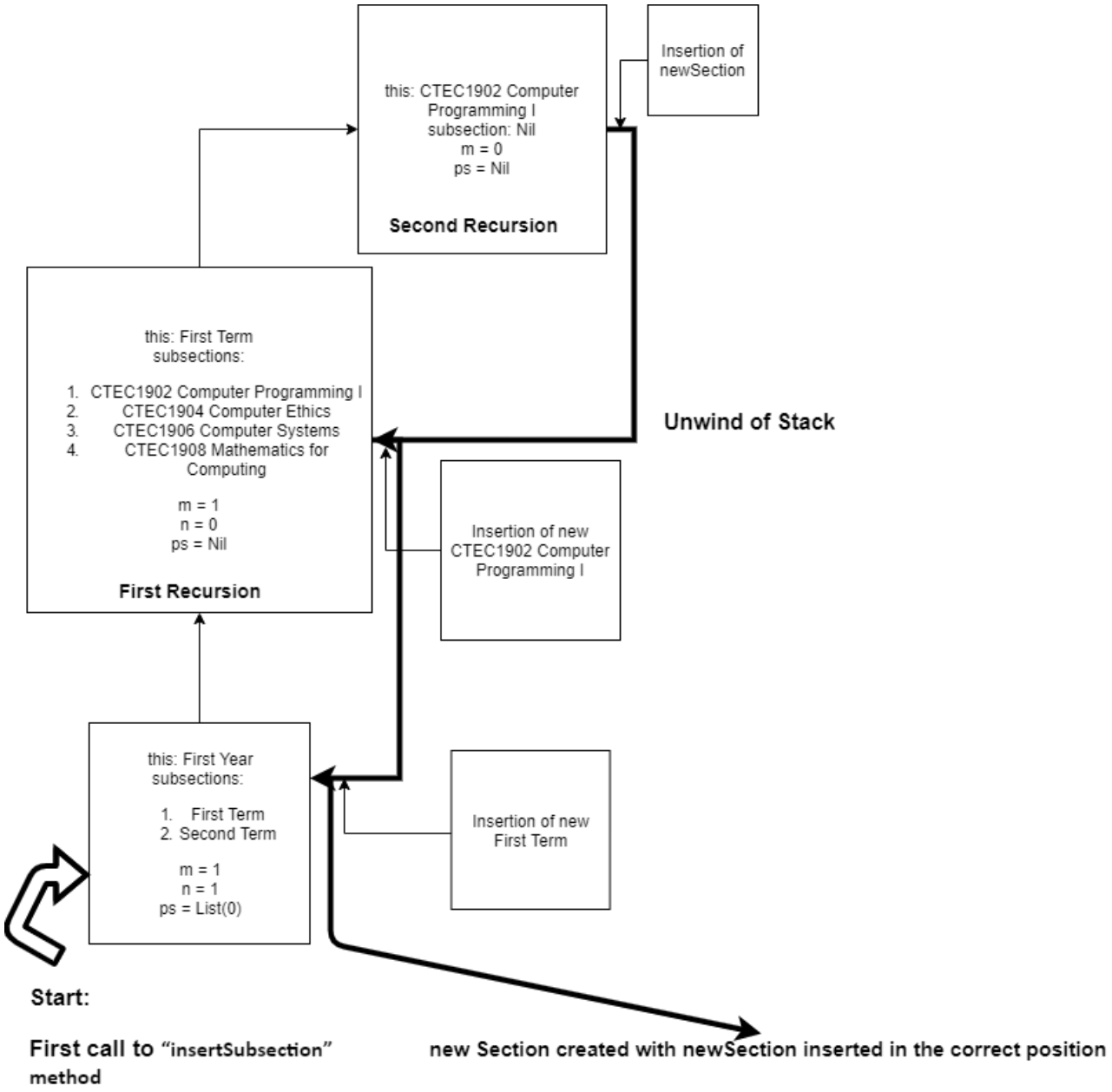

Figure 10: Diagram of Stack

Task 2: insertAtZero Implementation

```
def insertAtZero(chs: List[Int], newSection: Section): Book = {
  if (numberOfChapters == 0) bad("book has no chapters")
  else if (chs.isEmpty) bad("Didnt provide any Chapters")
  else if (chs.exists(_ > numberOfChapters)) bad(s"Chapter doesnt exists")
  else {
    val chapterRange = 1 to numberOfChapters

    val modifiedChapters = (chapters zip chapterRange) map {
      case (c, n) if chs.contains(n) => c.insertSubsection("0", newSection)
      case (c, n) => c
    }

    new Book(title, modifiedChapters)

  }

}
```

First thing to tackle when implementing "insertAtZero" method is, to add/modify the implicit method "toSectionNumber" so it can handle comma separated strings of integers.
My first thought was to create another implicit method String => List[Int], but this was obvious that it is not going to work, since under the same object Document, there are two implicit methods with the same definition and the compiler will complain about ambiguity.
For this reason the existing method is modified like below:

```
implicit def toSectionNumber(s: String): List[Int] = {
  s match {
    case i if i.contains(".") || i.contains(",") => s.split("[.,]").toList.map(_.toInt)
    case j if j.length == 1 => List(j).map(_.toInt)
    case _ => List.empty
  }
}
```

This will use pattern matching to check if the provided String "s" contains either dot or comma and proceed to split the String using regex matching any of them and create a List of integers.
Next case will happen if the provided String is a single integer, it will convert it to List and to Int.
Otherwise, it will return an empty List, for the other methods to handle.

Using this call to "insertAtZero" method:

```
println(compsci.insertAtZero("3,2", testSection))
```

it is expected to insert the Section "testSection" to the Book "compsci" at the top of positions two and three.

The use of the debugger makes it easy to see the individual calls. For example the Figure 11 shows the initial stack when the method is first called.

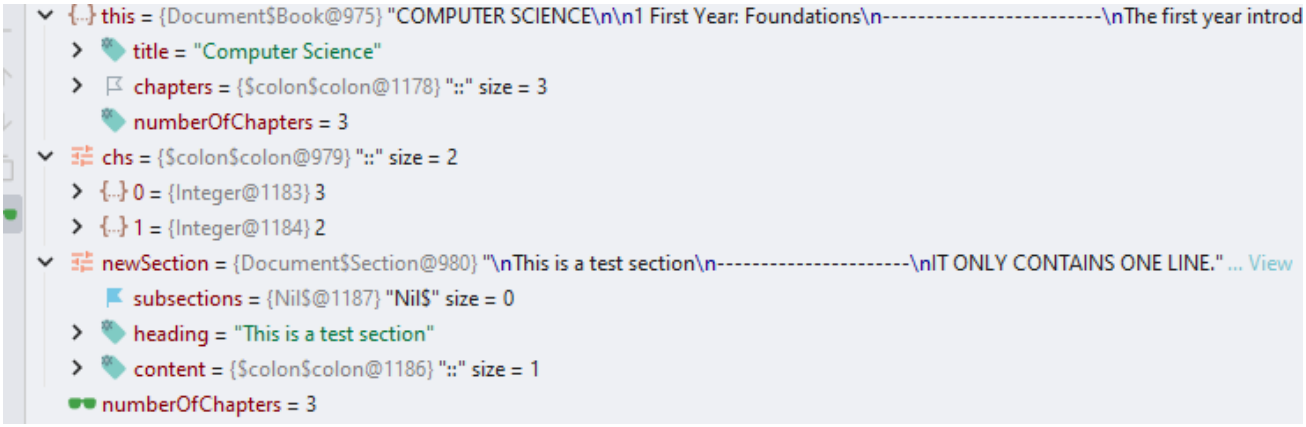

Figure 11: Initial Stack

The Book "compsci" is the "this" instantiation of the class Book. It has three chapters.

The "chs" variable is a List[Int] containing the positions that the newSection should be inserted at index zero.

The first three if statements are the error handling of the method. It will:

- Check if the class variable "numberOfChapters" is zero, which will indicate that the Book does not have any chapters and therefore the method cannot proceed.
- Check if the "chs" variable is an empty List. Without any positions to insert the testSection, the method will fail
- Finally, check if there is any element in the "chs" List that has a value greater than the "numberOfChapters" variable. That would mean that the provided position is out of bounds of the current Book and therefore cannot proceed.

If the error handling does not terminate the method, it will execute the "else" statement which is the main body of the method.

First, it will create a "Range" of numbers, from 1 to "numberOfChapters". These numbers represent the current chapters of the Book (Figure 12).

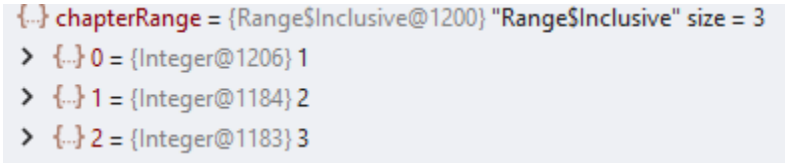

Figure 12: Range creation

Then the "modifiedChapters" variable is created. First is the "zip" of chapters and chapterRange variables. As shown in Figure 13, the "zip" higher order function takes two Lists and returns a new List[Tuple2]. Each element of both Lists is associated with the opposite element of the other List at that index, creating a Tuple of both elements. For example, the First Year chapter is in a tuple with the number 1 of the chapterRange. The Second Year with number 2 and the Final Year with number 3.

This is needed since we need to know the chapter's versioning. These numbers are the input of "chs" variable.

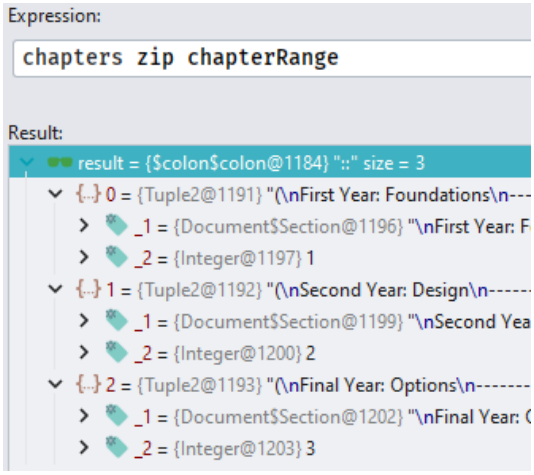

Figure 13: Zip Method

The "map" higher order function is then use in combination with pattern matching. "Map" is a function that takes a function and applies it to the elements of a data structure. By using pattern matching to deconstruct the tuples in the List, we can access the individual Chapters and versioning. This allows the selection of the desired chapter to apply a function on its elements, the subsections, which they are a List of Sections.

```
case (c, n) if chs.contains(n) => c.insertSubsection("0", newSection)
```

This case selects the correct chapter based on the versioning. If the "n" variable (which is the chapterRange numbers zipped with the chapters) is contained in the "chs" List, that means the user wants that specific Chapter "c" to be modified and insert the "newSection" at position zero.

If "chs" List contains the "n" integer of the tuple, that means it is the correct Chapter and then the right hand site of case is executed.
If it does not contain the "n", then the second case is executed. The second case just deconstructs the tuple from the List[Tuple2] after the "zip" function was applied and return just the Chapter. This Chapter does not need modifying, since the user did not select it.

If the first case is satisfied, then `c.insertSubsection("0", newSection)` will be applied to the chapter "c". The "insertSubsection" method is detailed explained above. It takes an "after" section position and a Section as its two arguments. The "after" determines the position that the newSection will be inserted after. So by definition, if "after" variable pointing at 0, the newSection will be inserted at

the top of the chapter. This is shown in Figure 14, that the resulted new Book has two new Sections at the top of chapters two and three.

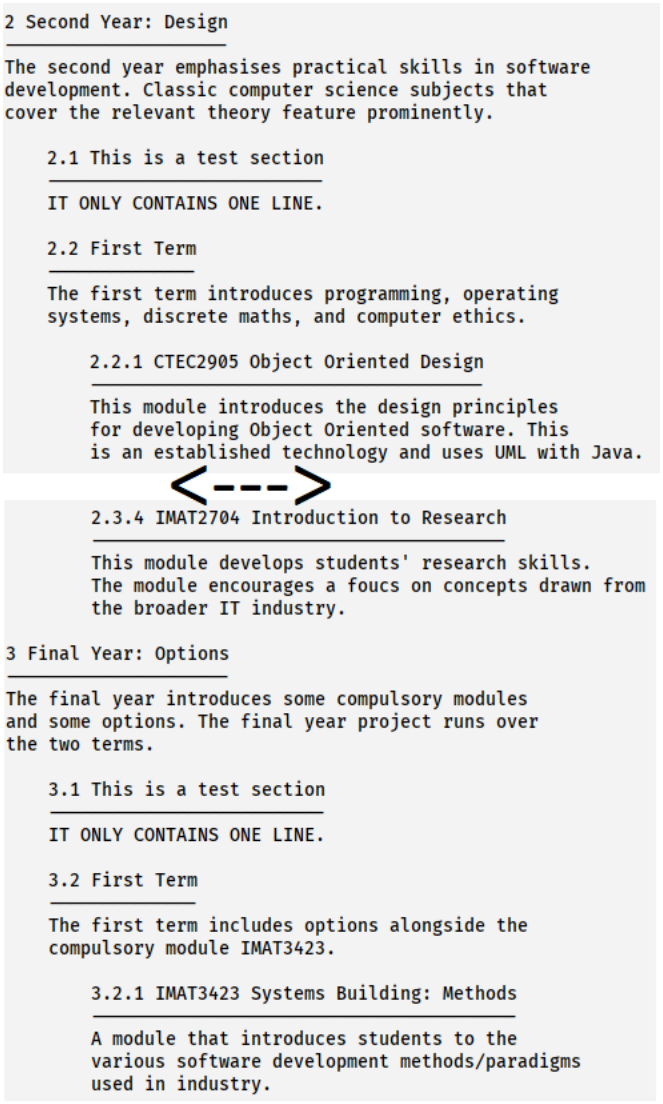

Figure 14: Final Result

After the "insertSubsection" method ends, the "map" function continues to run on every element left in the List[Tuple2]. At the end, the new Sections, which they are the product of return new Section from "insertSubsection" method or the second case in "map" function, are stored in the variable "modifiedChapters".

The "insertAtZero" method then executes the final step, which is to create a new Book, with the same "title" as the Book which the "insertAtZero" method was applied on, but with the new modified Sections stored in the "modifiedChapters" variable. This will call the class Book and return a new instance, a new Book, with the new modified Sections, as shown above in Figure 14. Also in Figure 15, it is shown the final stack before creating the new Book. The stack shows the new modified Chapters two and three with their new Section in place, but we can see that the "this" is not changed, and can not be changed, since it is immutable. This avoids concurrency problems when multiple threats might try to modify the compsci Book, adding new Sections, deleting etc, at the same time.

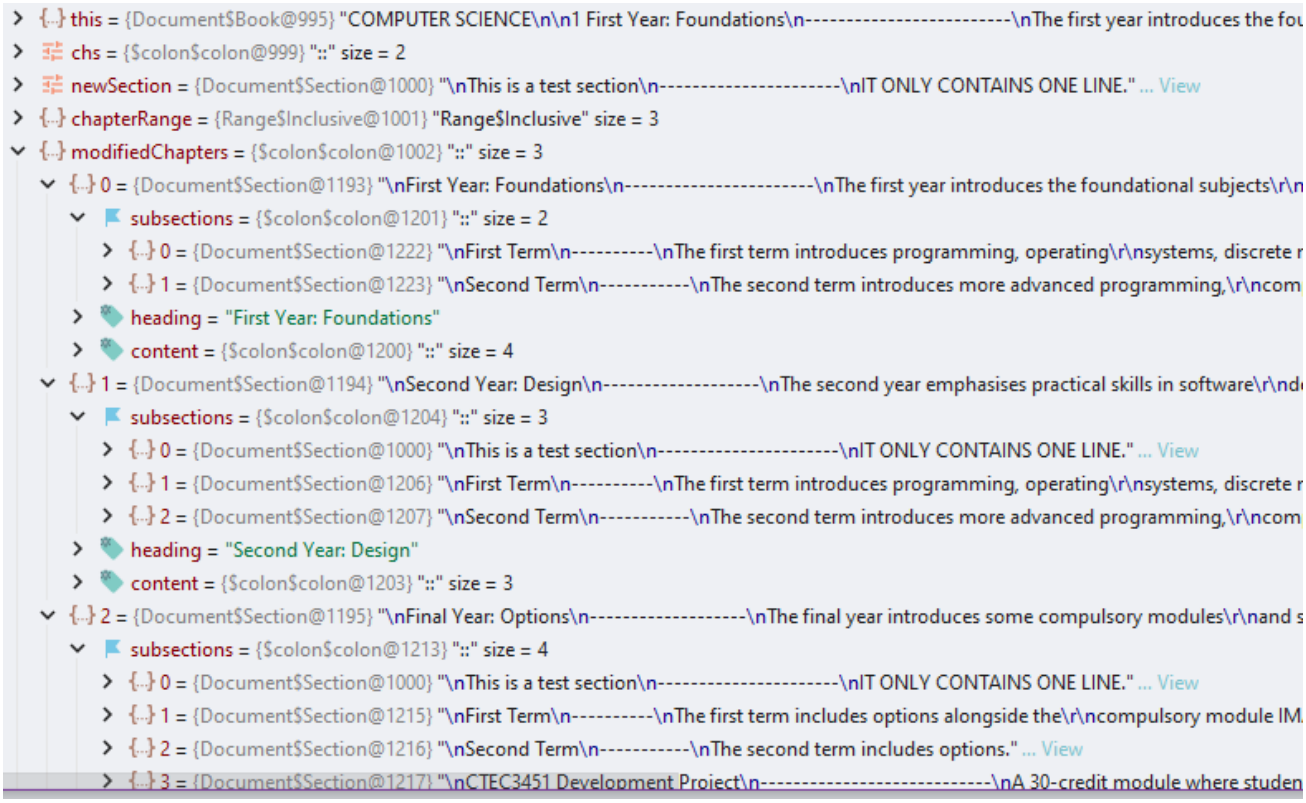

Figure 15: Final Stack

Critical Evaluation

The “insertAtZero” method works but has again a naïve approach to error handling. It does not use any of the Scala error handling techniques, like Option, Try, Either, to avoid using multiple if statements and catch and handle unexpected behaviours.