The project spans Chapters 9 and 10 of the book, which are about the high level Jack language and about the first part of the compiler (syntax analyzer) from Jack language to VM language. Please read these two chapters of the book thoroughly.

This project consists of two parts, one for each chapter.

Part 1

Assignment description

Objective The purpose of this assignment is to get acquainted with the Jack language for two purposes: completing the Jack compiler in the next two assignments, and completing the Jack operating system in the last assignment.

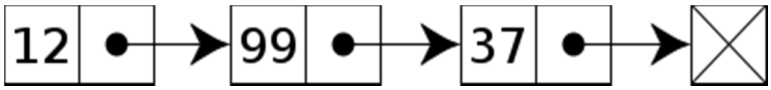
Contract Implement the *insert* and the *search* functions in a linked list.

Resources You will need three tools: the Jack compiler used to translate your program into a set of *.vm* files, the VM emulator used to run and test your translated program, and the Jack Operating System.

For details about the Jack OS and how to compile and run a Jack program read page 197 of the text book.

Background

A linked list is a data structure consisting of an ordered set of data elements, each containing a link to its successor (and its predecessor if the list is doubly linked). The list in the figure below is a linked list whose nodes contain two fields: an integer value and a link to the next node. The last node is linked to a terminator used to signify the end of the list. Note that the ordering in a linked list is represented by the links, and it does not imply that the contents of the data elements are ordered, as in the following example



Nonetheless, one can of course create and maintain a linked list, e.g., a linked list whose elements store integer numbers, in a way that the contents of the list elements are ordered according to their values (e.g., in the above example the ordering would be 12, 37, 99).

Details

Section 9.1.4 in the text book describes a Jack implementation of a linked list. You will start from this implementation, which is supplied in the file *List.jack*, and you will have to implement the following methods:

- *insertInOrder(int element)*: it assumes the list to be ordered in ascending order. It inserts *element* in the appropriate position in the list.

Example:

```
1. do list.print();
   -> 5 -> 7 -> 9
```

2. *do list.insertInOrder(8);*

3. *do list.print();*

```
-> 5 -> 7 -> 8 -> 9
```

- *find(int element)*: it makes no assumption on the ordering of the list (it can be unordered). It searches *element* in the list. If *element* is in the list, it returns the list starting from *element*. If not, it returns *null*.

You can run *Main.jack* to test your implementation. If you implemented the two methods correctly, you should get the following output:

```
-> 3 -> 5 -> 6
-> 2 -> 3 -> 5 -> 6
-> 2 -> 3 -> 4 -> 5 -> 6
-> 5 -> 6
Not Found!
```

Part 2

Assignment description

The Jack to VM compiler is the subject of two assignments. In this assignment (project 8) we focus on the syntax analyzer for the Jack programming language. In the next assignment (project 9) we will extend this analyzer into a full scale Jack to VM compiler.

Objective The purpose of this part is to complete the implementation of the syntax analyzer that parses Jack programs according to the Jack grammar.

Contract Implement the functions in charge of parsing the grammar rules defining the non-terminals: *subroutineBody*, *subroutineCall* and *ifStatement*.

Resources The main tool in this assignment is the partially implemented syntax analyzer written in Python. You will also need the supplied *TextComparer* utility, which allows to compare the output files generated by your analyzer to the compare files supplied by us.

Background

The syntax analyzer supplied with this assignment is documented in sections 10.2 and 10.3 of the text book. It is strongly recommended that you read and understand these sections before trying to read the code supplied with the assignment.

Details

Section 10.3 of the text book describes an implementation of a syntax analyzer. You will start from this implementation and complete the analyzer by implementing the following methods in the *CompliationEngine* module:

1. *_CompileSubroutineBody()*: implements the grammar rule `subroutineBody: ⋯`

2. *_CompileCall()*: implements the grammar rule `subroutineCall: ⋯`

3. *_CompileIf()*: implements the grammar rule `ifStatement:⋯` .Observe that the parsing of `'if' '(' expression ')' '{' statements '}'` is already implemented. You just need to implement the parsing of the optional `else` part.

You should complete the implementations of these routines following the order indicated above. In order to test your implementation of each one of the routines, use the *.jack* file in the corresponding folder:

1. *SubroutineBodyDecTest*
2. *SubroutineCallTest*
3. *ElseTest*

Each one of these folders cointains a *CorrectOutput* folder with the *.xml* output file that is expected from your syntax analyzer. You can use the *TextComparer* utility to check if your implementation behaves as expected.

In order to get acquainted with your tools for this assignment, make sure to understand the code in Figure 1 (at the end of this document).

Hints:

1. Read the `README.txt` file. What is the python module you need to run to parse a *.jack* file?
2. Make sure you fulfill the contractual agreement while implementing each routine (page 215 of the textbook)
3. Before starting, make sure you understand the code of *_CompileSubroutine()* in Figure 1
4. By looking at the code in Figure 1 try to answer these questions:
    - What are the two functions used to write the output Xml file? What is the difference between them?
    - What do the *_Expect*()* functions do? Why is it important to call them? (check their implementation to answer this question). How do they work?
    - How is the tokenizer used? How do we advance to the next token? How do we retrieve the symbol or keyword in the present token?
5. When implementing *_CompileCall()*, make sure to check if the optional argument *subroutineName* is set (read the comment below the function declaration).

Submission

You should submit two files: List.jack (part 1) and CompilationEngine.py (part 2), which should include your implementation of the above named functions.

Include the two files and the declaration in one zip-archive.

Make sure that the subject field and the name of the zip file states EP1200-Seminar8-group*N*-Firstname-Lastname, where *N* is your group number.

```
def _CompileSubroutine(self):
    """
    Compiles <subroutine-dec> :=
        ('constructor' | 'function' | 'method') ('void' | <type>)
        <subroutine-name> '(' <parameter-list> ')' <subroutine-body

    ENTRY: JackTokenizer positioned on the initial Keyword.
    EXIT:  JackTokenizer positioned after <subroutine-body>.
    """
    self._WriteXmlTag('<subroutineDec>\n')
    self._ExpectKeyword((KW_CONSTRUCTOR, KW_FUNCTION, KW_METHOD))
    self._WriteXml('keyword', self.tokenizer.KeywordStr())
    self._NextToken()
    if self.tokenizer.TokenType() == TK_KEYWORD:
        self._ExpectKeyword((KW_INT, KW_CHAR, KW_BOOLEAN, KW_VOID))
        self._WriteXml('keyword', self.tokenizer.KeywordStr())
    else:
        self._ExpectIdentifier()
        self._WriteXml('identifier', self.tokenizer.Identifier())

    self._NextToken()

    self._ExpectIdentifier()
    self._WriteXml('identifier', self.tokenizer.Identifier())
    self._NextToken()


    self._ExpectSymbol('(')
    self._WriteXml('symbol', self.tokenizer.Symbol())
    self._NextToken()

    self._CompileParameterList()

    self._ExpectSymbol(')')
    self._WriteXml('symbol', self.tokenizer.Symbol())
    self._NextToken()

    self._CompileSubroutineBody()
    self._WriteXmlTag('</subroutineDec>\n')
```

Figure 1