Name: TODO
Course: CSCI 220 Life Beyond Python
Exam Deadline: December 19, 2022 7:00 p.m.

Part I - Potpourri

POSIX Definitions

1. Definitions:

   1.1 **Application Program Interface (API)** - The definition of syntax and semantics for providing computer system services.

   1.2 **Built-In Utility** - A utility implemented within a shell. The utilities referred to as special built-ins have special qualities. Unless qualified, the term ”built-in” includes the special built-in utilities. Regular built-ins are not required to be actually built into the shell on the implementation, but they do have special command-search qualities.

   1.3 **Executable File** - A regular file acceptable as a new process image file by the equivalent of the exec family of functions, and thus usable as one form of a utility. The standard utilities described as compilers can produce executable files, but other unspecified methods of producing executable files may also be provided. The internal format of an executable file is unspecified, but a conforming application cannot assume an executable file is a text file.

   1.4 **Protocol** - A set of semantic and syntactic rules for exchanging information.

   1.5 **Shell** - A program that interprets sequences of text input as commands. It may operate on an input stream or it may interactively prompt and read commands from a terminal.

   1.6 **Standard Error** - An output stream usually intended to be used for diagnostic messages.

   1.7 **Standard Input** - An input stream usually intended to be used for primary data input.

   1.8 **Standard Output** - An output stream usually intended to be used for primary data output.

   1.9 **Terminal** - A character special file that obeys the specifications of the general terminal interface.

   1.10 **Utility** - A program, excluding special built-in utilities provided as part of the Shell Command Language, that can be called by name from a shell to perform a specific task, or related set of tasks.

Git

2. UPDATE ME

Tarball

3. -

Part II - Bash

Learning the Shell

4. DONE

5. Types of commands:

Type	Example
shell builtin	type
executable	which
alias	ll
shell function	bash

6. Redirection operators:

Redirection	Operator
Redirecting input	<
Redirecting output	>
Redirecting error	**2**>
Appending redirected output	> >
Redirecting standard output and standard error	**&**>
Appending standard output and standard error	cmd > >file **2**>**&1**

7. Types of expansion:

    1.1 Pathname Expansion

    1.2 Tilde Expansion

    1.3 Arithmetic Expansion

    1.4 Brace Expansion

    1.5 Parameter Expansion

    1.6 Command Substitution

8. Types of quoting:

Type	What does it suppress?
Double Quotes	word splitting, pathname expansion, tilde expansion, and brace expansion
Single Quotes	all expansions
Escaping Characters	quote only a single character and selectively prevent an expansion

9. Types of files:

Name	Attribute	Type
file1	-	regular file
file2	l	symbolic link
file3	l	symbolic link
file4	d	directory
file5	l	symbolic link
file6	p	named pipe
file7	l	symbolic link

10. Consists of entry type, owner permissions, and group permissions as well as other permissions.
    Entry type

    b Block special file.
    c Character special file.
    d Directory.
    l Symbolic link.
    s Socket link.
    p FIFO.
    - Regular file.

    Each of following fields has three character positions:

    r the file is readable. -, it is not readable.
    w the file is writable. -, it is not writable.

11. Reading, writing, and execute permissions are allowed for the owner, and read permissions only for the group and "world" users.

12. You can do anything, others can only execute it

13. Owner can read/write/execute, group/others can read/execute.

14. Only the owner of the file has full read and write access to it.

15. All can read/write/execute (full access).

16. Because we do not have reading permissions to do so.

17. We have permission to read file.txt, but do not have reading permission for the directory it is located in and therefore can not list it.

18. If group permissions for wall and write are not set so that that one can be "written" to and that message "read" from, then the message will not be accessible by one of the two parties.

Configuration and the Environment

19. Types of data:

    1.1 Environment variables

    1.2 Shell (local) variables

20. A subset of environment variables, such as PATH, affects the behavior of the shell itself. It also specifies, in order, the directories that the shell searches to find the program to run when the user types a command. If the directory is not in the search path, users must type the complete path name of a command.

21. LD_LIBRARY_PATH

22. Types of sessions:

    1.1 Bourne Shell

    1.2 C Shell

23. Startup files:

    1.1 BOURNE: /.bashrc, /.bash$_{profile}$, $Filenamedby$ENV (typically .shrc or .shinit), /etc/profile, /.profile

    1.2 C: .cshrc and .login

Common Tasks and Essential Tools

24. One of the main differences between BREs and EREs is the way they handle metacharacters, which are special characters that have a special meaning in regular expressions.

    In BREs, most metacharacters are treated as literal characters, meaning that they are interpreted as themselves rather than as special characters. For example, the "." character in a BRE represents a literal period, rather than any single character.

    In contrast, EREs allow the use of more metacharacters, which gives them more flexibility and power. In an ERE, the "." character represents any single character, and there are many other metacharacters that can be used to specify more complex patterns.

Another difference between BREs and EREs is the way they handle backslashes. In a BRE, a backslash is treated as a literal character unless it is followed by a metacharacter that is special to BREs. In an ERE, a backslash is treated as a special character that can be used to escape other metacharacters.

Overall, EREs are generally more powerful and flexible than BREs, but they can also be more complex to work with. The choice of which type of regular expression to use depends on the specific needs of the task at hand.

Part III - C

Maintain your perceptron ADT

25. Valgrind is a tool that helps programmers detect and debug memory-related issues in their programs. It works by analyzing a program as it runs and looking for memory errors, such as memory leaks, buffer overflows, and invalid memory accesses.

    Memory errors can be difficult to detect and debug because they often don't manifest themselves until long after the error has occurred. Valgrind helps programmers find these errors by providing detailed information about the program's memory usage and pointing out any issues it finds.

    Some common types of memory errors that Valgrind can help with include:

    Memory leaks: These occur when a program allocates memory but fails to release it when it is no longer needed. This can result in the program using more and more memory over time, eventually leading to poor performance or even crashing.

    Buffer overflows: These occur when a program writes more data to a buffer than it was designed to hold. This can overwrite other parts of memory and cause the program to behave unpredictably or crash.

    Invalid memory accesses: These occur when a program tries to access memory that it doesn't have permission to access. This can also cause the program to behave unpredictably or crash.

    Overall, Valgrind is a useful tool for finding and fixing memory-related issues in programs, and can help programmers write more stable and reliable code.

26. Memcheck is a feature of Valgrind that is specifically designed to detect and diagnose memory-related issues in programs. It works by analyzing a program's memory usage as it runs and looking for errors such as memory leaks, buffer overflows, and invalid memory accesses.

    When Memcheck detects an issue, it will provide detailed information about the error, including the location of the error in the source code and a description of the issue. For example, if Memcheck detects a memory leak, it will provide information about which memory blocks were not properly freed and where in the source code the leaks occurred.

It seems that in this code the model and data structures are not properly freed at the end of the program. Therefore in order to fix this functions would need to be called so that unnecesary memory is not stil consumed.

27. -

28. -

Evolve your perceptron ADT

29. DONE%

Part IV - Python

Evolve a simple interpreter

30. DONE

Part V - Java

Describe the functionality of a class

31. Selected 2 (line by line constructed into human readable format ignoring trivial lines of code, does not map one to one with Java code, simple captures all ideas):

    The first line defines the package that the class is a part of. This is used to organize Java classes into namespaces and prevent naming conflicts.

    The second line imports the CONTINUE constant from the FileVisitResult enum in the java.nio.file package. This constant is used to indicate that the file tree traversal should continue.

    The third line imports the File class from the java.io package, which represents a file on the file system.

    The fourth line imports the FileInputStream class from the java.io package, which allows you to read bytes from a file.

    The fifth line imports the IOException class from the java.io package, which represents an exception that is thrown when an input or output operation fails.

    The sixth line imports the InputStream class from the java.io package, which represents an input stream of bytes.

    The seventh line imports the StandardCharsets class from the java.nio.charset package, which provides constants for the character sets supported by the Java platform.

    The eighth line imports the FileVisitResult enum from the java.nio.file package, which defines constants that are used to indicate the result of visiting a file.

The ninth line imports the Files class from the java.nio.file package, which provides utility methods for working with files.

The tenth line imports the Path class from the java.nio.file package, which represents a path to a file or directory.

The eleventh line imports the Paths class from the java.nio.file package, which provides utility methods for creating Path objects.

The twelfth line imports the SimpleFileVisitor class from the java.nio.file package, which is a utility class for visiting files in a file tree.

The thirteenth line imports the BasicFileAttributes class from the java.nio.file.attribute package, which represents the basic attributes of a file.

The fourteenth line imports the ArrayList class from the java.util package, which is an implementation of the List interface that uses an array to store its elements.

The fifteenth line imports the HashMap class from the java.util package, which is an implementation of the Map interface that uses a hash table to store its elements.

The sixteenth line imports the List interface from the java.util package, which is an interface for a list of elements.

The seventeenth line imports the Map interface from the java.util package, which is an interface for a mapping of keys to values.

The eighteenth line imports the StringJoiner class from the java.util package, which is a utility class for constructing a string from a sequence of elements.

The nineteenth line imports the Collectors class from the java.util.stream package, which provides utility methods for collecting elements from streams.

The twentieth line imports the ANTLRInputStream class from the org.antlr.v4.runtime package, which is an input stream for use with ANTLR (Another Tool for Language Recognition).

The twenty-first line imports the Token class from the ‘org

The twenty-second line imports the ParseException class from the org.apache.commons.cli package, which represents an exception that is thrown when a command line argument cannot be parsed.

The twenty-third line declares the JLexer class, which extends the SimpleFileVisitor class.

The twenty-fourth line declares a private boolean field named normalize, which is set to false by default.

The twenty-fifth line declares a private field named filesAnalyzed, which is an instance of ArrayList¡String¿ and is used to keep track of the names of the files that have been analyzed.

The twenty-sixth line declares a private field named vocabulary, which is an instance of HashMap¡String, Integer¿ and is used to keep track of the frequency of each word in the analyzed files.

The twenty-seventh line declares the visitFile method, which is overridden from the SimpleFileVisitor class and is called for each file visited during the file tree traversal. It takes

two arguments: a Path object representing the file, and a BasicFileAttributes object representing the attributes of the file. The method throws an IOException if an I/O error occurs.

The twenty-eighth line is an if statement that checks if the file is a regular file, has a .java extension, and is not empty. If any of these conditions are not met, the method returns CONTINUE to indicate that the file tree traversal should continue.

The twenty-ninth line creates an InputStream from the file using its absolute path.

The thirtieth line creates a JavaLexer from the InputStream using the ANTLRInputStream constructor.

The thirty-first line declares a StringJoiner object named line, which is used to construct a string from a sequence of elements.

The thirty-second line declares a List object named lines, which will be used to store the lines of the resulting .lex file.

The thirty-third line declares an integer variable named previousLine and initializes it to 1. This variable is used to keep track of the previous line number when iterating over the tokens produced by the JavaLexer.

The thirty-fourth line begins a for loop that iterates over the tokens produced by the JavaLexer. The loop continues until the EOF (end of file) token is reached.

The thirty-fifth line is an if statement that checks if the current token is on the same line as the previous token. If it is, the token's text is added to the line StringJoiner object. If it is not, the current line string is added to the lines list, a new line StringJoiner object is created, and the current token's text is added to it. The previousLine variable is also updated to the current line number.

The thirty-sixth line adds the final line string to the lines list.

The thirty-seventh line removes any empty elements from the beginning of the lines list using the removeIf method and a lambda function.

The thirtyeighth line creates a Path object named lexPath representing the file that the resulting .lex file will be written to. The file's path is created by replacing the .java extension of the original file's path with .lex.

The thirty-ninth line writes the lines list to the lexPath file using the write method of the Files class.

The fortieth line adds the name of the file to the filesAnalyzed list.

The forty-first line returns CONTINUE to indicate that the file tree traversal should continue.

The forty-second line declares the getText method, which takes a Token object as an argument and returns a string representation of the token.

The forty-third line declares a String variable named text and assigns it the text of the Token object.

The forty-fourth line is an if statement that checks if the normalize field is true. If it is, a switch statement is executed that replaces certain token types with predefined strings.

These predefined strings are used to normalize the tokens so that they can be compared and counted more easily.

The forty-sixth line updates the vocabulary map by incrementing the count for the current text value. If the text value does not exist in the map, it is added with a count of 1.

The forty-seventh line returns the text string.

The forty-eighth line declares the main method, which is the entry point for the program. It takes an array of strings as an argument and throws IOException and ParseException if an I/O error occurs or if the command line arguments cannot be parsed, respectively.

The forty-ninth line creates an Option object named input representing the -input command line option.

The fiftieth line creates an Option object named normalize representing the -normalize command line option.

The fifty-first line creates an Options object named options and adds the input and normalize options to it.

The fifty-second line creates a CommandLineParser object named parser from the DefaultParser class.

The fifty-third line creates a CommandLine object named cmd by parsing the command line arguments using the parse method of the parser object and the options object.

The fifty-fourth line declares a String variable named inputValue and assigns it the value of the -input command line option. If the option is not present, the value is set to the current working directory.

The fifty-fifth line creates a Path object named start from the inputValue string.

The fifty-sixth line creates a JLexer object.

The fifty-seventh line sets the normalize field of the visitor object to the value of the -normalize command line option.

The fifty-eighth line traverses the file tree starting at the start path using the walkFileTree method of the Files class and the visitor object as the file visitor.

The fifty-ninth line prints the names of the files that were analyzed to the console.

The sixtieth line prints the vocabulary of the analyzed files to the console, sorted by frequency in descending order. The sixty-first line is the end of the main method. The sixty-second line is the end of the JLexer class.