**Final Project: Readme**

**Overview:**

The intended program uses the programming language Python to create a unique Banking language that will handle the creation and removal of accounts. The program will also store unique identifiers for the account, which consist of the person's initials and a six-digit number, to ensure each account can be tracked without duplicates.

The accounts are stored in a JSON file containing a key, the account ID, and the value, which is the account balance. Using the accounts, the Banking language we created will handle three\* types of transactions: deposit, withdraw, and accrual, which will\* be reflected in the accounts balance and saved back to the JSON. The Banking language consists of classes Shell, Token, Lexer, Parser, Token, Interpreter, and an EBNF Grammar File.

The Banking language consists of classes Shell, Token, Lexer, Parser, Token, Interpreter, and an EBNF Grammar File. The language also has a simple banking syntax that will allow a user who is not familiar with Python to operate the banking program and language.

**EBNF Grammar:**

The EBNF Grammar consists of the grammar of the banking language looking at it you can see the various ways to handle a transaction. Tokens such as “fullname”, “account”, “sign”, “operation\_word”, and “value” are defined in the standard\_command grammar. Options for account management such as “create”, “view” and “drop” are also provided.

Each of these keywords can be found by looking above and finding their definitions. Some of the keywords such as "account", "digit", "float", "int", and “namepart” can be found when following the grammar tree from the “standard\_command” once again you can find these definitions by looking above.

**Shell:**

The Shell is used to take in a user input to decide what operation is taking place. The Shell has no logic in the banking language, but it does help us decide how we will use it. The Shell consists of a while loop inside another nested while loop to determine if the entry is a blank string. The user is given two options. 1. Enter in a command manually to have the Lexer look over the input. 2. Read commands from a text file and have the Lexer review that text input. Reading from a text file will handle multiple lines if each command is on a separate line. The Shell also calls the the Lexer to run first, followed by the Parser with a list from the the Lexer, and then lastly the Interpreter which handles the Nodes and AST from the Parser.

**Token:**

The Token class is a simple class that creates a Token object with a type and an optional value. This can be used within multiple classes to check a token's type or value and use logical operations to perform specific tasks.

**Lexer:**

The Lexer is the part of the Banking Language that will take the raw input of the command or text file and parse through it character by character to form tokens that will be used later on by the Parser. Each token, as described above, must have a type and will have an optional value. Inside the Lexer, there is an Error handling class which is used for any errors that occur during the Lexical Analysis. The IllegalCharError is a subclass of the Error class that will be used if a character the Lexer cannot tokenize properly is found.

The Lexer also has a Position class which will act as an object within the Lexer class itself at the current character position of where the lexical analysis is occurring. It can do this by tracking the index number, line number, and column number of the text that is being analyzed. This will allow us to track the position over even multiple line entries from the text file.

The Lexer class itself is where the text from the Shell is used. It has attributes such as Position and currentChar. It contains multiple functions that assist with the makeTokens function that returns the list of Tokens created. It also contains and advance method to advance the position of the currentChar in order to prevent the program from getting stuck on the same character if it were unreadable for some reason.

The makeTokens function goes through each character and uses if and else logic to determine if the character is a digit, matches a regular expression, appears in the dictionary defined in the same function, or if none of these match it will return an error of Illegal Character. If one of the chars matches with our if/else logic, it will then run either a subfunction to create a token from a word or number or use a token from the dictionary that is already created. These subfunctions are called makeNumber, makeID, and makeWord and all will use different logic to determine patterns in the current string and run the appropriate tokenization methods.

The makeWords function works by using regex to match if a character has a-z or A-Z in it and appends each of those to a string where when returned becomes a WORD token with a value of the string. Sperate cases for if the tokens value matches a word will cause a different token to be created in leu of the WORD token.

The makeDigit function works similarly to the makeWord, where it will check if the current character is a digit or period and append it to a string if it returns true. The function checks to see if more then one period has been added on and will return and error of Invalid Number if a non-existent number is entered. By tracking the periods, we are able to determine if the number is an INT or FLOAT for the Token Type with the number as the value.

The makeID will match a regular expression of a-z, A-Z, or a digit 0-9. Similar to the other subfunctions, it will append these to a string until a whitespace character or blank character is reached, and then it will output the value as a Token Type ID.

The run method calls the creation of the Lexer object and the makeTokens function so all the text should be tokenized after. The result of the Lexer will be an output of tokens in a list that will be then passed into the Parser.

**Parser:**

The Parser is responsible for taking the list of tokens provided by the Lexer or Lexical analysis and then using them to create an AST (Abstract Syntax Tree), which provides the order for which our program will and its token interpretation. The Parser class is initialized with a position which is the current token, a list of Tokens passed through by the Lexer, a list of operator types, and multiple functions for creating Nodes in the AST.

The Parser class has multiple functions that will use the values of the current token(s) to create Nodes once they have been called. Once the parser is complete, these nodes will be used when interpreting the AST. More information on how nodes function can be found below in the nodes section.

The operator and transaction functions both take into account the grammar of the banking language and use logic in handling specific tokens. For the operator, we know from the grammar that an operator must be preceded by an account and proceeded by a number. Using this we are able to say that an operator is the value between these two values. The same goes for creating a nameNode if two WORD tokens are next to each other based off of the Grammar it, they must be a firstName and lastName, so we can use those to create that nameNode.

The parse function inside the Parser class runs the transaction function which is used to create a transaction Node this then is the base/root of the AST. From running the transaction function, we can see if the token type matches token types in the Lexer and if they do run functions to create a Node for that token. Once the Nodes have been created, they will be appended to the list. We can use these as our AST.

**Nodes:**

The Nodes are a type of object similar to the tokens that make up our AST and hold either values or other Nodes. Using this logic we can use the TransactionNode as the base/root Node and break it up into a nameNode and an operatorNode. The nameNode just holds a value of a firstName and lastName and can not be broken down further. The operatorNode consists of a token, an operation, an IDNode, and a NumberNode. This can be broken down even further as the NumberNode holds a number value, and the IDNode will hold an account ID. Using this logic of nodes breaking down into other nodes, we can run through our AST properly, from a whole transaction into broken down components for the interpreter to run through.

**Interpreter:**

The Interpreter is responsible for taking the AST that was generated from the Parser and then preforming the correct order of visitation of the Nodes in the AST and handling the logic that each of those Nodes should have which is seen by functions such as “get\_balance”, “change\_balance”, “save\_data”, “find\_highest\_account\_id”, and “generate\_account\_id”.

The visit function will handle each Node type that is read from the AST by using some python functionality of grabbing the name attribute of the currentNode. Since the Banking language follows our EBNF grammar, we can start with the transaction Node as the base. That is the first Node we will visit, which will call its respective visit function and perform operations defined by it. Using this method we can break down Nodes into other Nodes and values which again can be broken down into more values. The operations performed inside these visit functions provide all the logic required to change balances, manage accounts, and run our specification tests to ensure data is being handled correctly and as expected.

**Specification Testing:**

For DSL specification testing we have included a test mode that acts that same way a developer would add a debug mode. If the keyword "test" is entered into the program, it will pull values from the Shell that have been coded into the Shell for this particular instance. The test mode will trigger a Boolean that will run in the Interpreter to allow the logic of the banking language to run on the test data instead of real data from the "bankinginput.txt". The test will run all of the operations that our language supports and will compare the values that are returned from the Interpreter to the expected values that were passed through in the Shell. The console will give us the actual value and expected value as well as a "TEST PASSED" or "TEST FAILED" result.

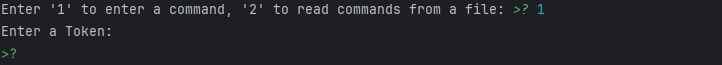
**How To Run:**

The user will run the shell.py file, and they will be prompted with the following:

A black background with white text

Description automatically generated

Option 1 is a manual entry, while option 2 reads from the file “bankinginput.txt”. This readme demonstrates option 1 but will show a picture of how option 2 can be inputted. After selecting option 1, you will see the following.



**Syntax:**

*‘\*’ If an account already exists*

* **\*Deposit:** CHASE DALLMANN CD000001 + | deposited | deposit | deposits 5000
  + AGENT ID OPERATOR VALUE
* **\*Withdraw:** JOHN PETRIE JP000011 -50 - | withdrew | withdraw | withdraws
  + AGENT ID OPERATOR VALUE
* **\*Interest:** JOHN PETRIE JP000011 \* | accrue | accrued | accrues
  + AGENT ID OPERATOR VALUE
* **Account Management:**
  + **Creation:** CHASE DALLMANN create | open | creates | opens | opened
    - AGENT OPERATOR
  + **\*Deletion:** JOHN PETRIE JP000011 close | drop | drops | closes | closed
    - AGENT ID OPERATOR
  + **\*View:** CHASE DALLMANN CD000001 view | views | viewed
    - AGENT ID OPERATOR

**Sources:**

<https://stackoverflow.com/questions/1855558/call-method-from-string>  
<https://ruslanspivak.com/lsbasi-part1/>  
[**https://www.youtube.com/watch?v=Eythq9848Fg&list=PLZQftyCk7\_SdoVexSmwy\_tBgs7P0b97yD**](https://www.youtube.com/watch?v=Eythq9848Fg&list=PLZQftyCk7_SdoVexSmwy_tBgs7P0b97yD)[**https://www.youtube.com/watch?v=RriZ4q4z9gU&list=PLZQftyCk7\_SdoVexSmwy\_tBgs7P0b97yD&index=2**](https://www.youtube.com/watch?v=RriZ4q4z9gU&list=PLZQftyCk7_SdoVexSmwy_tBgs7P0b97yD&index=2)[**https://www.youtube.com/watch?v=YYvBy0vqcSw&list=PLZQftyCk7\_SdoVexSmwy\_tBgs7P0b97yD&index=3**](https://www.youtube.com/watch?v=YYvBy0vqcSw&list=PLZQftyCk7_SdoVexSmwy_tBgs7P0b97yD&index=3)

[**https://stackoverflow.com/questions/71764921/how-to-delete-an-element-in-a-json-file-python**](https://stackoverflow.com/questions/71764921/how-to-delete-an-element-in-a-json-file-python)

[**https://www.geeksforgeeks.org/how-to-fix-list-object-is-not-callable-in-python/**](https://www.geeksforgeeks.org/how-to-fix-list-object-is-not-callable-in-python/)

[**https://stackoverflow.com/questions/11174024/why-do-i-get-str-object-has-no-attribute-read-when-trying-to-use-json-loa**](https://stackoverflow.com/questions/11174024/why-do-i-get-str-object-has-no-attribute-read-when-trying-to-use-json-loa)

[**https://www.geeksforgeeks.org/append-to-json-file-using-python/**](https://www.geeksforgeeks.org/append-to-json-file-using-python/)

**<https://dev.to/fractalis/creating-a-dsl-in-python-dj6>**

**Screenshots**

**EBNF GRAMMAR:**

A computer screen shot of a code

Description automatically generated

**SHELL:**

A screenshot of a computer program

Description automatically generated

**TOKEN:**

A screen shot of a computer

Description automatically generated

**LEXER:**

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screen shot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screen shot of a computer

Description automatically generated

A screen shot of a computer program

Description automatically generated

**Parser:**

A screenshot of a computer program

Description automatically generated

**A screenshot of a computer program

Description automatically generated**

A screenshot of a computer program

Description automatically generated

**Nodes:**

**A screenshot of a computer program

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

**Interpreter:**

**A computer screen shot of a computer code

Description automatically generated**

**A screen shot of a computer code

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

**A screen shot of a computer program

Description automatically generated**

**A screen shot of a computer code

Description automatically generated**

**A screen shot of a computer program

Description automatically generated**