DESIGN DOCUMENT ---- COMP 40 ---- 9/10/18 Vladimir Hugec & Ethan Tu

DATA STRUCTURES:

1. Linked – list
    a. Struct Node {
        Node *nextMatch; //if address of both char *str are the same
        Node *nextCollision; //if address of char *str are different
        char *str; //the parsed line
        int lineNum; // the line number
        int argNum; // the filename
    }
2. Hash table
    a. Array of pointers to linked list made from Node above
    b. Hashes the address of each char *str, node saves relevant info
       Such as filename and line number of the original line
3. Hanson Atom
    a. Each time readaline(..) is called, the associated char array (after it is formatted to be free of extra whitespace or delimiters) is passed into Atom_string(..) which saves a copy of the line from the file and returns a pointer to the save location of the copied line within the Atom structure
    b. This pointer is then hashed by our hash function and saved with the corresponding line number and filename

Example code:

```
int main(int argc, const char *argv[]) {
        Node htable[TABLE_SIZE];
        InitializeHTable(&htable);

        For (int i = 0; i < argc; i++) {
                Int count = 0;
                Char *datapp = NULL;
                FILE *fp = the file;
                While(&datapp != NULL) {
                        Readaline(fp, &datapp);
                        Count++;
                        Char *line = formatLine(&datapp);
                        Char *hline = Atom_string(line);
                        insertHash(&hline, i, count);
                }
        fclose(fp);

        for (int j = 0; j < TABLE_SIZE; j++) {
                printMatches(&htable);
        }
        return 1; }
```

TO DETERMINE MATCH GROUPS:

Once all the input has been read:

1. Traverse the hash array
2. For each spot in array check if pointer == NULL
   If it is NULL -> move to next spot
3. IF it is NULL -> traverse linked list and check each node if pointer to match is NULL,
   If the pointer to match exists -> print the matches (in correct format)
   Keep traversing linked list until the end checking for matches at each node
4. Once array has been traversed fully, then all matches will have been printed

INVARIANTS:

Int i is invariant with respect to the file currently being parsed, I.e.
File 2 -> i=2 ->argv[2] = filename
Int count is the line number of the parsed line from the current file

If stopped partway throught 'i' will tell you the current file being parsed, how many have already been parsed and count will tell you the line you are currently parsing.
The Linked-list would show all found matches up until the stop

TEST CASES:

PART B:

We will need to make a main(..) and a few test files (test2.txt)
We will for loop for an arbitrarily large number (1 loop = 1 line parsed)
Tests files will be:

1. 1 line 10000-1mil chars long
2. A lot of '\n' at the start of the file, the 1 line at bottom
3. File with invalid chars such as æ or other non Unicode chars
4. File with 1 char on one line, nothing for 300, then 1 on the 301st
5. Empty file

To test out error handleing:

- Run readaline(NULL, &datapp); -> checks file pointer is null error
- Run readaline() on file 3, -> check unable to parse input error
- Run with file 1 to check malloc failure error

PART C:

Similarly to PART B, most testing will involve varying our input files and trying to break our simlines program, since simlines is much more complex than readaline, with many more moving pieces, we will need to test out our structures first and if/how well they work with the Atom structure
Since Atom_string(..) always returns a pointer to the char array, we will use that returned address to store information about the line in question
We will need to make sure our hash function is as efficient as possible (evenly distributed pointers to linked lists), since a bad hash function will drastically impact our program's performance, it needs to be a top priority, we will go about this by printing out the entire

hashed array and whether each spot had an associated linked list. The most spots that are associated -> the better the performance.
The linked list structure will also need to be tested, since our linked list is ‘2D’ we need to make sure a traversal hits every node, likely this will involve faking inputs into the 2D Linked list and traversing and printing to see if all inputs are found again by the traversal
Once these tests are done we can being file testing:

1. A file with all the same lines
2. Line with invalid chars
3. Empty files
4. A bunch of newlines followed by 1 line at the bottom
5. Weird file format used for input file
6. One crazy long line (1mil chars?)
7. Uppercase vs lowercase handling
8. Sames lines with varying delimiters and whitespace
9. File using invalid chars as delimiters