Restoration Architecture (1 page)

Kalen Lauring (klauri04) and Alex Vu (avu07)

At a high level, restoration will take in the lines from readaline and store the dirty (non-digit & digit bytes on the same line) lines in a table. The lines are then stripped to only their digit-bytes, and printed in the order they appear in the file.

- We will use **Hanson's Table_new** to initialize our table. We will fill the table with atoms as keys and cstrings as values.
    - Each atom will be created as Atom_new(const char *datapp,
      int **whatever readaline returns**)
    - The table will hold all non-digit bytes as the key, added line by line from readaline. We do this because once we parse an already-existing non-digit byte c-string, we know that is an original line. Therefore, we can simply continue to look for lines that have the same non-digit byte c-string and ignore the rest of the lines.
- We will use ***Table_put** to enter our key value pairs inside the table
    - We will use **Hanson's table_get** method to check if this key already exists in the table.
    - **If the key does NOT exist**, we add the key value pair to the table, and continue reading in the next line
    - **If the key DOES exist AND found_original is FALSE**
        - We initialize a new sequence with **Seq_new**. The sequence will hold all of the original lines stripped of non-digits as cstrings
        - We take the key already existing in the table, make it the first element of our new sequence holding original lines with **Seq_put**
        - We add the line that has just been found to be the second element of the sequence using Seq_put again. This sequence at this point will have two elements. We will update found_original to be true.
    - **If the key DOES exist AND found_original is TRUE,** we ignore the table, and use **Seq_addhi** to add the c-string into our sequence and move on.

Outputting to stdout

- We know the original maxval is 255, so when printing the header, we just print that directly. Same goes for the type, which is P5.
- We will get our dimensions from the sequence of original lines
    - Width will be determined by the length of a the first cstring stored in our sequence ADT (due to the fact that every original line will be the same length)
    - Length will be determined using **Seq_length**
- After everything has been printed we will use **Table_free** to free our table memory
- We will use **Seq_free** to free our sequence memory.

Implementation Plan + Testing (1 page)

1. Create the .c file for your restoration program. Write a main function that spits out the ubiquitous “Hello World!” greeting. Make sure the Makefile rules are up to date, compile and run. Time: 5 minutes
    a. *Testing*:
        i. We will use various “Hello world” statements in our files to ensure that the files can be linked, compiled, and run without errors or warnings, printing to terminal. We will test a basic call which has no arguments and makes sure Hello World prints correctly.

2. Create the .c file that will hold your readaline implementation. Move your “Hello World!” greeting from the main function in restoration to your readALine function and call readaline from main. Make sure the Makefile rules are up to date, compile and run. Time: 5 minutes.
    a. *Testing:*
        i. We will use various “Hello world” statements in our files to ensure that the files can be linked, compiled, and run without errors or warnings, printing to terminal. We will test a basic call which has no arguments and makes sure Hello World prints correctly.

3. Extend restoration to open and close the intended file and ensure the program correctly handles command line inputs. Time: 15 minutes.
    a. *Testing:*
        i. We will create files with one line in it (short and simple) to test that files are correctly being opened and closed with print lines to see results.
        ii. We will test using valgrind to make sure that we are not losing any memory or causing other errors.
        iii. Should raise appropriate errors if too many arguments are given, file is unable to open, if input is expected but not supplied, if errors are encountered reading the file, or if memory cannot be allocated using malloc. This will be tested by using certain invalid commands and files.
        iv. Ensure that the contents of a file can be correctly printed to stdout.
        v. CRE when given file is problematic (doesn’t exist, bad read, too many arguments)

4. Update readaline to take the actual parameters to be used later, and update code in readaline and restoration to utilize it. This means to update readline’s functionality so that it is able to read through the file until it reaches an endline character, return the length that it took for the datapp pointer to travel from the first character to the endline character, and update datapp to point to the first character in the saved line. Time: 45 minutes
    a. *Testing:*

i. Provide readline with our own files with short lines to text that it is correctly seeing endline characters.
ii. Include in our testing files strange/weird characters to make sure that they are being correctly read in and stored in our cstring array.
    1. Print cstring array to check this storage
iii. Include test cases with different formats of file endings (eg. a newline before EOF) in order to ensure end of file is correctly being detected in restoration and that the appropriate errors are being thrown.
iv. Will throw a check runtime error when: Either or both of the supplied arguments is NULL, an error is encountered reading from the file, or memory allocation fails
v. Ensure that our basic implementation maxing out at 1000 lines throws the correct error and exit status if the input line is too long.
vi. Assert that datapp != NULL && readaline returns correct information

5. Declare all ADTs and relevant variables to be used during restoration's runtime. Time: 10 minutes.
    a. *Testing*:
        i. Comment out any in-progress code, and try compiling to make sure libraries have been included correctly and ADTs have been correctly declared without syntax issues. Run valgrind to make sure that ADTs are properly being cleared.
        ii. Assert that creation correctly allocates memory

6. Update restoration implementation to call readaline, take its output, and use it to create the Atoms and cstrings that will be stored in our table. This will involve parsing the line to separate non-digit characters from digit characters. Add that the original lines can be correctly found, and that our interaction with the ADTs is correct. Time: 60 minutes.

    a. Update the line-reading loop to check if the non-digit Atom key already exists, and handle operations described above. (see restoration architecture). Time: 30 minutes.
        i. In the creation of our sequence ADT, we will test different "hints" for the sequence's initial size in order to test timely expansion.
    b. *Testing:*
        i. First, verify readaline correctly reads lines correctly, printing the correct string to stdout as well as the string length. Then, ensure that multiple lines can be read.
        ii. Next, ensure that digits and non-digits can be separated correctly in restoration, and that the Atoms can be created correctly.
        iii. Raise allocation failed errors if table_new, seq_new, or atom_new fails to allocate memory

7. Using the sequence, access its size to determine length, and use our previously-saved variable that holds the width. The MaxVal is 255, and the magic number is P5. Time: 20 minutes.
    a. *Testing:*
        i. First, width calculation from digit sequences. Then, confirm the height and width calculation from sequence length. We will use custom files so we can control the height and width with basic files.