Kevin Croker
Etebom Samuel

Design Document

Part A : Two Dimensional, Polymorphic, Unboxed arrays

1. We are trying to adapt the unboxed-array abstraction to support two dimensional arrays.
2. Some of the functions that we will implement will be defining two analogs of the bitmap function.
    a. **UArray2_map_row_major** is a function that will allow us to call an apply function for each element in the array.
    b. **UArray2_map_col_major** calls an apply function for each element in the array. Row indices vary more rapidly than column indices.
    c. **Uarray_new -** allocates and initializes, and returns a new array of length elements with bounds zero through length - 1. Unless length is zero, in that case the array has no elements.
    d. **Uarray_free -** deallocates and clears *uarray, it is a checked runtime error for uarray or *uarray to be null
    e. **Uarray size -** would return the size of the element of the array
    f. **Uarray_at -** returns a pointer to element number i
    g. **Uarray_height -** returns the height of the array
    h. **Uarray_width -** returns the width of the array
3. In the book Hanson has described each of these functions and gives us a brief explanation of what they do and return
4. We will be using an array as the representation and the invariants will be every time that we get an input for the row and column (i,j). This way we will have inserted that position into the array and are now able to move onto the next position.
5. Once all of the invariants have been passed we will have our entire uarray2 completely full. First, we get the size of the array to know how long is should be, and then we will put in each position as we go along. Once all of the invariants have been passed into the array, it will then be full.
6. Test cases are left as an exercise for the reader.
7. Some idioms that we are going to need include:
    a. The idiom for allocating and initializing pointers to structures
    b. The idiom for storing values into an array

Part B:

1. We are trying to represent an image in the form of an array of bits that will be in row-major and column-major mapping. The bits will either be 1 if it is black or 0 if white.
2. Some functions that we will include in this part of the project include
    a. **bit_value(image)** - This function will allow us to get a bit and then determine the value that is will get depending on the color
    b. **get_next_bit(image)** - this will allow us to move from bit to bit after we have already determined whether the previous bit is getting a value of 0 or 1
3. For an example, I believe that we could use brightness.c from the last assignment to heavily give us a base on what happens here. We are going to determine if the bit is bright (giving it a value of 0) or not bright (assigning it a value of 1).
4. We will use an array and the invariants that will be satisfied are all of the bits that we have already processed. Once we determine the value of the bit and move on, that will now become and invariant
5. Our array will be full once all bits become invariants. In the beginning we will determine the length of the array by getting how many bits are in the image
6. Test cases are left as an exercise for the reader
7. Idioms
    a. Array_new
    b. Array_free
    c. Array_length
    d. Array_size
    e. Void *Array_put