```c
// Each entry in a symbol table consists of an integer memory address and an identifier name
// (for e.g. a variable or label).
struct TableEntry {
    char *name;
    int address;
}; typedef struct TableEntry TableEntry;

// A table is a collection of TableEntries stored in table_array, with the number of entries
// stored in table_length. Every entry is required to have a unique name. You can add an unlimited
// number of entries to the table, and you can search for an entry by its name. The table_space
// variable is only used internally and tracks the memory allocated to table_array.
struct SymbolTable {
    TableEntry **table_array;
    int table_length;
    int table_space;
}; typedef struct SymbolTable SymbolTable;

// Creates and returns a new empty symbol table.
SymbolTable *malloc_table();
// Frees every entry in table, then table itself.
void free_table(SymbolTable *table);
// Adds a new entry to table with the given name and address.
void add_to_table(SymbolTable *table, const char *name, int address);
// If table contains an entry with name search_name, returns i such that table->table_array[i] is
// that entry. If table doesn't contain such an entry, returns -1.
int get_table_entry(const SymbolTable *table, const char *search_name);
```

Figure 1: Interface for a symbol table in C (`symboltable.h`).

Week 8 assignment: A Hack assembler

1 Tasks

1. Consider how to implement a symbol table in C.
2. Write a Hack lexer in C and test it on the scripts provided.
3. Extend your lexer into a full assembler and test that on the scripts provided.

2 Required software

For this lab, you will need some way of comparing two text files for differences. One way is to use the "fc" terminal command on Windows or the "diff" terminal command on Linux and Mac OS. On non-lab non-Mac machines, you can also use e.g. Meld, a piece of free open-source software with a nice graphical user interface. You will also need a C compiler of your choice — we hope you have one set up already after 6 weeks of Programming in C!

3 Symbol tables

In lectures this week, we covered the idea of a *symbol table*. In Figure 1, you can see one possible header file for a symbol table along with documentation in comments. **Spend a few minutes considering how you would implement the accompanying source file yourself.**

While implementing a symbol table is definitely within your abilities, and it's an interesting exercise in programming C, it's not a good use of time in an architecture assignment. Most languages offer built-in types with this

```c
/* A union is a special type that uses one spot in memory to store one of several
 * different variables of different types. Here, TokenData can store either a Keyword,
 * an int, a character, or a string. Assigning to e.g. key_val will overwrite what's
 * stored in int_val, and there's no built-in way to tell whether what's currently stored
 * there is e.g. an int or a string --- we'll keep track of that with the TokenType enum.
 *
 * Usage examples: Suppose data is a TokenData variable.
 *      data.int_val = 42; printf("%d", data.int_val); // Prints 42.
 *      data.char_val = '@'; printf("%c", data.char_val); // Prints '@'.
 *      data.char_val = '@'; printf("%d", data.int_val); // Prints 64 (the ASCII value of @).
 */
union TokenData {
    Keyword key_val;
    int int_val;
    char char_val;
    char *str_val;
}; typedef union TokenData TokenData;
```

Figure 2: The `TokenData` type with usage examples.

functionality — in Java (which you'll learn next teaching block) they're called HashMaps, and in other languages they might be called "hash tables" or "dictionaries". Since the point of the assignment is to write an assembler, we've provided a symbol table for you in `symboltable.h` and `symboltable.c` (downloadable from the unit page). **Spend a few minutes reading through the source code and making sure you understand it.** You'll also be using this code for the rest of the unit.

4 Tokens

Another bad use of your time in an architecture assignment is grappling with C's string handling and file I/O, which are again more cumbersome than most other languages. To some extent this is unavoidable, but we've tried to mitigate it by providing code for a `Token` struct in `token.h` and `token.c` matching the list given in lectures. Each token contains a type (an `enum` which can be `SYMBOL`, `KEYWORD`, `INTEGER_LITERAL`, `IDENTIFIER` or `NEWLINE`) and a value. The value can be a `char` (for symbols), an `enum` (for keywords), an `int` (for integer literals), a string (for identifiers), or nothing (for newlines). It's stored in a "union" type, a special sort of variable which can store one of several types in the same memory location — see Figure 2.

`Token.c` provides functions to create new tokens, free old tokens, write tokens to a file (in somewhat human-readable format) and read tokens back from that file. **Spend a few minutes reading through the code and making sure you understand it, starting with `token.h`.** You'll also be using simple variants of this code for the rest of the unit.

5 Lexing

A good first stage in writing any compiler or assembler is to implement a lexer which reads in code, outputs the corresponding list of tokens to an intermediate temporary file using the `write_token` function, and (in this case) creates a symbol table of labels. You **don't** need to worry about error handling — while this would be important for any assembler intended to be used, it's not the focus of the unit and there's plenty to do already!

Your first step should be to **read and understand the given `lex_file` function**, then **fill in the remaining code for the `lex_line` function**. Given a string `line` and a pointer to its corresponding ROM address `rom_address`, the `lex_line` function should write all tokens in the line to the given file `output` and store any labels in the given symbol table `labels`. We've given you most of the code for this — it will scan through `line`, calling `lex_token` if the next thing on the line is a token and `lex_label` if the next thing on the line is a label. **Add code to handle whitespace and comments and to keep the ROM address updated.** (Recall that lines which only contain labels, comments and/or whitespace don't correspond to lines of machine code, while other lines correspond to exactly one line of machine code.)

Your next step should be the `lex_token` function. Given a string `line`, this function should identify the first token in the line, write it to the provided Token pointer `dest`, and return the length of the token (e.g. `1` for "`@`"). You don't need to deal with labels or whitespace or comments, since they're handled by `lex_line` and `lex_label`. We've also provided code to handle newlines, symbols, and integer literals. **Add code to handle keywords and identifiers.** Before proceeding further, you should test your implementation. Here are some good test cases, all of which are valid Hack assembly:

- `@431`
- `@THAT` (`THAT` should be lexed as a keyword)
- `@R13` (`R13` should be lexed as a keyword)
- `@BANANA`
- `@APPLE` (should not lex `A` as a keyword)
- `@AD` (should not lex `A` and `D` as keywords)
- `0`
- `0;JMP`
- `D=0;JLT`
- `D=0`
- `AD=D-M;JNE`

Finally, you should **fill in the `lex_label` function**. This code should add the label to the given symbol table with the appropriate ROM address and return.

Now **test your lexer** on the four test scripts from the corresponding Nand2Tetris project, which we have provided on the unit page. We've provided the lexing outputs for each test script, so you can check your output against the correct output using either fc/diff or Meld. We recommend testing add.asm first, then max.asm, then rect.asm, then pong.asm (which is over 25k lines and is autogenerated from a higher-level language). To help with debugging, max-L.asm, rect-L.asm and pong-L.asm are alternative versions with no labels or identifiers.

6 Parsing

Once your lexer is up and running, you should try to implement a parser that reads the resulting tokens, populates the variable symbol table and actually generates the Hack machine code. As with the lexer, we've provided skeleton code for this in `main.c`, and we recommend you don't worry about error handling to start with.

The provided `parse_file` function repeatedly reads one instruction's worth of tokens from the .lex lexer output, using the `get_next_instruction` function together with the fact that instructions always end with a newline token. For each instruction, it then calls `parse_instruction` with a symbol table of labels (generated during lexing), a symbol table of variables (currently being generated), the array of Tokens that make up the instruction, the instruction's length, and the output file handle. `parse_instruction` then checks to see whether the provided instruction is an A-instruction or a C-instruction, then calls `parse_a_instruction` or `parse_c_instruction` accordingly. Each of those functions will deposit a string into a provided buffer, which `parse_instruction` will write to the output file.

First you should read and understand `parse_file`, `get_next_instruction` and `parse_instruction`, then **fill in the logic in `parse_instruction` for detecting A-instructions**.

Next, you should **fill in the remaining code in the `parse_a_instruction` function**. Your objective here is to load the A instruction's operand into the integer `value_to_load` variable provided — after doing this, the `int_to_bin_string` call at the end will copy the appropriate A-instruction into `dest`.

Next, look at the `parse_c_instruction` function provided, which splits the job into parsing the `comp`, `dest` and `jump` operands, then concatenates the resulting binary strings together into a single machine code instruction.

Fill in the `parse_c_jump` and `parse_c_dest` functions, as well as the remaining code in the `parse_c_comp` function. (You should refer back to the instruction set from lectures as you do this!) Note that in our test data, we assume that in C-instructions with computations that don't involve A or M, the `a` bit of the `comp` operand will be set to `0`. We also assume that the second and third bits of the instruction will be set to `1`. For example, we assume the instruction `D;JMP` will become

`1110001100000111`, not
`1001001100000111`,

even though both would be valid Hack machine code.

Lastly, **uncomment the code in `main.c` that calls the parsing functions** and **test your assembler**! We've provided valid .hack files for the same four test scripts as the lexer.