```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 in weeks 8–10.

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 in weeks 8–10.

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, and (in this case) creates a symbol table of labels. We've provided skeleton code for functions `lex_file`, `lex_line`, `lex_token` and `lex_label` in `main.c` — you don't have to follow this outline, but you might find it helpful. We recommend that you **don't** worry too much about error handling to begin with — focus on getting a lexer that works on valid Hack code.

We recommend testing your lexer on the four test scripts from the corresponding Nand2Tetris project, which we have provided on the unit page. If you use the `Token` type and `write_token` function we've provided, 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. Here are some further tips:

- With one exception (see below), you can always classify a token based entirely on its own text and the text immediately following it. As such, we recommend a general approach of lexing a line in token-by-token as in the `lex_line` and `lex_token` functions in the skeleton code.
- Your lexer should safely remove spaces at the start or end of lines, as well as comments either on their own line or immediately following a line. For example, it should be able to handle these two lines of code:

```
AD=M
  AD=A+D        // Doubles A and D
```

- In order to fill out the labels symbol table, you'll need to keep track of which ROM address the current line of assembly corresponds to. Recall that lines which only contain labels, comments and/or white space don't correspond to any lines of assembly. We suggest doing this by updating `line_no` in the call to `lex_line`.
- Since `AD`, `AM` and so on aren't keywords, the following code is syntactically valid Hack assembly that your lexer should be able to handle:

```
AD=M
@AD
```

  This means that in the `lex_token` function, if the first character of the token you're analysing is "A", "D" or "M", you'll want to respond differently depending on whether you're parsing a C-instruction (in which case they will always be a keyword) or an A-instruction or label (in which case they will always be an identifier). This is a good place to use a static variable.

6 Parsing

Once your lexer is up and running, you should try to implement a parser that 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 too much about error handling to start with. We've also provided valid .hack files for the four test scripts. Some further tips:

- Don't forget to uncomment the code in `main.c` that calls the parsing functions!
- Remember that an instruction doesn't need to have either a destination or a jump to be valid. For example, `A+D` or `0` are both valid lines of Hack assembly by themselves.
- Remember that computations with binary operations can't be between A and M — only between A and D or D and M. For example, `M=A+D` is not valid Hack assembly. It's worth referring back to the instruction set from lectures when writing the parser.
- While it's valid Hack code either way, in our test data (see below), 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`.

- The `strtol` function in `<stdlib.h>` provides a useful way of converting a string "472" to the corresponding integer 472. The function header is

$$\texttt{longintstrtol}(\texttt{constchar} * \texttt{str}, \texttt{char} * * \texttt{endptr}, \texttt{intbase});$$

in this example, you should set the first argument to "472", the second argument to NULL, and the third argument to 10 (the base of the integer you're converting). Less elegantly, you can convert a character digit `c` to an integer `x` with the code `x=c-'0'`. Can you see why this works?

- Don't worry about error handling for dubious code like `A=D+A;JMP` (which both updates A and jumps to the value stored in A on the same clock cycle).