61Ccc: Lexing

Hello and welcome to your first 61C project! Please make sure you read this spec in its entirety. There is a lot of information here, and we wouldn't want you to miss any of it.

At the beginning of each section you can find the objectives or learning goals for each project part, and at the end of the section are the items you'll be expected to turn in. If you have any questions, please post on the dedicated piazza threads; anything outside of the threads will be ignored.

Good luck! We know you can do it :)

Background

For the purposes of this project, we've invented a language which is a variant of the more common C language. Like the C language, this language requires compilation and so we must build a compiler for it! Project 1 and Project 2 will guide you through building several 61Ccc compiler components: a lexer, parser, and code-generator. We'll talk more about how a program goes from code you've written to a program on your machine, but for now all you'll need to know is that the goal of compilation is to generate a machine-code file that (after assembly, linking, and loading, which you'll learn about later in the semester!) your system can run.

Compilation:

- Input: Your code (`sample.61c`)
    - Preprocessing
    - Lexer
    - Parser
    - …
    - Code Generator
    - Assembler, Linker, Loader
- Output: An executable (`run.me`)

Let's talk about how each of these components fit together. Time for a crash-course in CS164!

Note: If you like this project or are interested in learning how compilers work in more depth, check out CS164! It typically has a higher workload so you'll want to take it in a lighter semester, but it's a great course and applies many of the topics we cover in CS61C.

Preprocessing

- Input: Your code (`sample.61c`)
- Output: Your code, modified

Before we begin feeding your code into the lexer, we might want to make some changes to it. These changes are made by the preprocessor. If you've worked with C code before, you may have seen statements like `#DEFINE` or `#include <lib.h>` or `#include "my_file.h"`. These are known as directives and they perform textual substitution in files. Consider the `#DEFINE <name> <value>` directive, it scans through the file and, every time it comes to a place where <name> is used, it replaces it with <value>. All of the directives are processed before the rest of the compiler runs.

In our compiler, the only directive we support is the `#include` directive. When the pre-processor sees this string, it (recursively) looks up the mentioned file. You won't need to handle this. We keep a running list of all included files and their contents; new 'includes' add the file to the beginning of this list. We pass this entire list of files to the lexer independently. Then, we concatenate the lexer output together and parse all files at once. This ensures all functions referenced from included files are local by the time we get to the parsing phase.

Lexing

- Input: Your code (`.temp_sample.61c`)
- Output: `[Token: <Type> (optional: data), ...]`

When you want to compile and run your code, the first thing the compiler does is read in your file as one long string. This file may be arbitrarily long and contain any number of lines. Your lexer should be able to handle any input size. After reading the file, the compiler passes this string to a component called a 'lexer'.

The goal of the lexer is to classify each substring in your code as a particular token. The lexer does not do any 'thinking' apart from trying to match your code to the list of approved tokens it has been given. What that means is the lexer won't error if your code isn't type-checked, formatted correctly, or doesn't make semantic sense (i.e. trying to assign a string to an integer variable, forgetting a closing brace or semicolon, passing an incorrect amount of arguments to a function). All it tries to do is determine if you're using valid tokens for the language you're working in.

Check out the appendix for information on the tokens our lexer supports as well as a sample lexer input/output.

Parsing

- Input: `[("substring", token), ("substring", token), … ]`
- Output: AST (Abstract Syntax Tree)

After the lexer has ensured your code contains only valid tokens, the list of string-token pairs are passed to the next component in the compiler: the parser. **The goal of the parser is to ensure the tokens from the previous phase are arranged in valid ways.** The parser won't do any type checking (i.e. trying to assign a string to an integer variable) or semantic checking (ensuring you pass no more/less than three arguments to a function which expects as much), but it will verify, for example, that all your conditionals have open and closing braces and that when you use a '<=' operator, you have two items to compare.

The output of the parser is what we call an AST or abstract syntax tree. It shows us how the pieces of our program are related to each other in parent-child relationships. If the AST is complete, that is, every token from our lexer output is connected by some path to the root and every leaf of the tree is a token, the parser considers our code to be valid. If there are errors present, our tree may contain error nodes instead of tokens. **See the appendix for more details on error handling and various node types.**

Type Checking, Semantic Analysis, and Code Generation

We will cover these items in your project two spec! Stay tuned. The above information should be all the background you need for project one.

Project 1-1: Lexing

Deadline: Friday February 8th (But look below for partial deadlines!)

Objectives:

- The student will be able to read, write, and understand C-unit tests and integration tests.
- TSWBAT explain the relationship between characters and integers and how to compare characters in C.
- TSW understand memory leaks and how to fix/avoid them using Valgrind

In this part of the project, you'll be finishing up the lexer for the 61Ccc language. We have already completed several parts for you.

Your first job will be to write a few tests for the functions described in `string-helpers.h`. These functions correspond to the regular expressions we used to describe the lexer tokens in the first stage of the background section. Please read the function descriptions before posting questions on piazza.

Note: this file contains descriptions and function headers only! You'll be writing the body of the functions later, but we'd like to expose you to test writing first. Please complete this section before working on the rest of the project.

Lab 1 - Due: Tuesday, February 5th

What is a unit test?

Unit testing is a level of software testing where individual components of a software are tested by themselves. The purpose is to validate that each unit of the software performs as designed. A unit is the smallest testable part of any software (a function, class, instance, etc.). It usually has one or a few inputs and usually a single output. In `string-helpers`, for example, each function is its own unit.

For this course, we'll use CUnit as our unit testing framework. If you've worked with JUnit in CS61B, this is the C equivalent! We've given you a sample test to base your tests on. You can find it in `tests/cunit`. If you'd like to look through additional documentation, we recommend looking at the CUnit docs to start.

ACTION ITEM: Create a new CUnit suite for each function you complete. In each suite create 3 tests (NOTE: not three asserts, but three individual test functions; you can have as many

asserts per test as you'd like) with unique contents. Any correctness tests that you write should fail before you implement your `string-helpers` functions and pass after.
You can run the CUnit tests with the following commands (Note: you should not need to create any new files, only modify test.c). The make file is set up to work with the CUnit version installed on the hive machines. You should do your testing there :)

```
> cd tests/cunit  # move to the cunit directory
> make build      # this compiles the cunit executable
> make run        # this will run the cunit tests if you use
```

You should also run your tests under valgrind to ensure there are no memory leaks or undefined behaviour. In generally this is really helpful and you should do it with all unit testing. We have set up Valgrind for you:

```
> cd tests/cunit      # move to the cunit directory
> make build          # this compiles the cunit executable
> make run-valgrind
```

To run Valgrind on other setups in the future you should refer to the valgrind description in lab 1 or open and read the Makefile.

Now you'll need to complete the body of the functions described in `string-helpers.h`. You must fill in all of the functions in `string-helpers.c`. You may add helper functions if you wish, though they are not required. The file itself contains documentation for each function.

ACTION ITEM: Complete the function bodies for `string-helpers.c`. Some have been done for you. These are the functions you'll need to do (we suggest you approach them in this order):

- `is_alpha`
- `is_space`
- `is_digit`
- `is_identifier_component`
- `is_valid_identifier`

You'll also need to implement `str_concat` by the project deadline (Feb 8th), but this will not be required for this lab.

Once you're done with this section, see your TA for checkoff.

Homework 1 - Due: Wednesday, February 6th

In addition to the project components below, remember to complete the number-rep assignment on gradescope!

What is an integration test?

Integration testing is a level of software testing where individual units are combined and tested as a group. The purpose of this level of testing is to expose faults in the interaction between integrated units.

View the appendix for more information on how to write and run lexer tests.

ACTION ITEM: Write the following tests and place them in `/tests/lexer/homework1`. Note, these should be separate files; one per requirement, named as follows:

- `correct.61c` - Write a test that lexes correctly
- `error.61c` - Write a test that errors for *any* reason
- `negative.61c` - Write a test that produces a negative number with a single token (i.e. not -12, which is two tokens)
- `string.61c` - Write a test with a string that contains a double quote
- `bad-char.61c` - Write a test for a character that results in an error

What are memory leaks and how do we test for them?

When coding in C, you (the programmer) are responsible for explicitly managing the program's memory. If you want something to persist through function calls, you have to explicitly say so (using malloc/calloc/realloc…) otherwise it will disappear from your owned memory once the stack frame closes. Just like you must explicitly say when a piece of memory should stick around, you must also explicitly say when that piece of memory should disappear (free). Memory leaks occur when a programmer allocates memory using one of our allocation functions, but forgets to free it by the time the program exits.

Morgan and Nick made a mistake in the starter code! The compiler they gave you leaks memory that grows with input size. They'd really appreciate it if you could help them find the bug and fix it! In general, we are concerned with memory leaks that grow with input size and will grade your final project accordingly. It is ok if your fix involves leaking a constant amount of memory.

ACTION ITEM: Find the memory leak in our compiler code and patch it. HINT: You may find it helpful to review the Valgrind instructions we gave you in lab.

See piazza for submission instructions.

Writing the Lexer - Due: Friday, February 8th

The last thing you'll need to do for the lexer is complete the rest of the `SelectToken` function in `tokenizer.c`. We have given you several examples to work with, and we suggest reading through the function to understand how it works before attempting to write code.
When you finish, you should be able to produce the correct set of token outputs for any file containing 61Ccc code. This means you'll need to add support for the following tokens:

- Comments
    - We should remove comments from our program as they do not change how our program runs. Comments start with two slashes '//' and are followed by any number of characters (not including newlines '\n'). You don't need to worry about supporting multi-line comments (/* this is one of those */).
- Characters
    - Our language supports all printable characters (we suggest checking the manual pages for `isprint`) as well as a few whitespace characters ('\n' '\t' '\r' and single space), the null terminator ('\0'), and the single backslash ('\\').
- Identifiers
    - Identifiers must start with a letter (lowercase or uppercase English alphabet) and may contain any number of letters, digits, or underscores following this first letter.
- Integers
    - Integers are defined as 0, or a single digit 1-9 followed by any number (possibly none!) of digits 0-9.

ACTION ITEM: Add lexing support for comments, characters, identifiers, and integers in `SelectToken`. Your code should handle relevant error cases as well.

We have provided `generate_character_error` and `generate_string_error` which you'll need to pass in the relevant items (they're located in `tokenizer-errors.h`). You should not be writing your own error messages. Check out the appendix for more information on error handling.

NOTE: There are /*YOUR CODE HERE*/ markers where you need to implement things. You can use the functions you created in `string-helpers.c`

Congrats! You've finished part one!

Submission instructions will follow on piazza.

Appendix - Project 1-1

What is the preprocessor?

The preprocessor for `gcc` is a program which performs textual replacement based upon a series of directives each with a specific functionality. It is not necessary to support a preprocessor to have a functional compiler, but we do need a way of specifying functions that will be found in other files. To do so we implement a modified version of the include directive, where any #include followed by a “string” (after possibly some white space) will specify another file to be tokenized. We will then combine these tokens for later parts of the project. If an error occurs in opening the specified included file, then the executable will exit with an error in the preprocessor. The path of the included file is relative to the path of the file being compiled.

As with all portions of this project errors between stages will not carry through, but we will attempt to present the maximum number of errors at each stage. This means we should try to display all files that won’t open at the preprocess (and similarly all errors you encounter for the lexer and parser).

We are not asking you to implement the functionality behind the preprocessor but you should be aware of it in case you run into problems.

Lexer Tokens

Below is a sample input/output from the lexer. Note that the input is a regular “61Ccc” program while the output is a list of tokens. The tokens contain the type of token along with any additional information (such as associate value in the case of Identifiers and Integers). Tokens may contain no additional information, as you see with the “`int`” token below.

`Input (Sample.61c)`	`Output`

`int my_age = 6;`	`[Token: int ]`
	`[Token: Identifier: my_age ]`
	`[Token: = ]`
	`[Token: Integer: 6 ]`
	`[Token: ; ]`

To view all tokens supported by the lexer, check out this file.

Lexer Testing

During lab, we tested our string-helpers.c functions using CUnit. We used CUnit because each of the functions in our file functioned independently from one another, and we could call them on their own inputs easily. When testing the lexer, however, this becomes more difficult. If we want to lex a token, we have to first succeed at reading in the file, or we have to test SelectToken with a provided buffer, token list, etc..

In this case, it makes more sense to do integration testing (i.e. make sure both the file reading and token processing components of our lexer work well together).

To write integration tests, let's start by taking a look at the sample file we gave you in `tests/lexer/sample-tests`. The sample file matches the example given in the "Lexer Tokens" section above. Note that the file is written as if in plain C and uses no additional formatting. We have given you two samples; one that errors (bad-sample) and one that should lex correctly. In the sample-tests folder we have also given you their outputs.

You should write additional tests in the same way and keep them in a new directory under `tests/lexer/`. If you'd like to run a particular test, you can use the following command after `make`:

```
./61ccc -t tests/lexer/sample-tests/good-sample.61c
```

Lexer Error Handling

In the process of searching for tokens it is possible to encounter tokens which do not meet the requirements of the language. When this occurs you should produce an error token, which should store the illicit token which does not meet the requirements. We do not want to simply exit on the first error as it is more useful to get a list of many illicit tokens in a file to fix all at once. Some of the errors are already handled but you will need to create an error token at the appropriate time for characters, strings, identifiers, and integers. For characters and strings the error token should consist of all characters from the starting quote (double for strings, single for characters) until either a closing quote or a newline character (whichever comes first). For other errors you should consume tokens until any whitespace character is reached. This functionality is already handled for you in `generate_generic_error`, `generate_string_error`, and `generate_character_error`, you will just need to call the functions at the appropriate time.

Discarded Input

In addition to tokenizing there are three situations in which the lexer will simply discard input without producing tokens for them (erroneous or correct).

The first is in the presence of whitespace (not inside a string or character). These inputs are meaningless and thus discarded. The starter code already implements this for you.

The second is in the presence of a sequence with an #include, possibly some white space, and then a “string”. This input sequence matches the sequence used by the preprocessor to include other files (which we will use for project 2). The lexer should discard both the string and the #include. The string does not have the printable character requirement (and if it is not already a valid filename should error before reaching this stage). This is already implemented in the starter code.

The third is in the presence of a comment. A comment is defined by the “//” sequence (without any whitespace). If a comment is encountered the rest of the line should be discarded without producing any tokens. **<u>You will need to implement this.</u>**