ASSIGNMENT 1

CS 473: Compiler Design / Fall 2025

Description

In this assignment, you will implement a lexical parser for the language Cardinal. The language is described below. Your lexical parser should reject programs with invalid tokens and print a list of tokens for programs with only valid tokens.

The Cardinal Language

The Cardinal language is defined in a separate document.

Format of the output

Your lexical parser should output each token recognized in a separate line, with the following format:

```
<line> <col> <token> <info>
```

Where:

- **line** is a number denoting the line in the input file where the token starts
- **col** is a number denoting the column in the input file where the token starts
- **token** is a string denoting which type of token was recognized
- **info** is further information about some tokens

For instance, consider the following valid program in Cardinal:

```
<?cardinal?>return 0;</?cardinal?>
```

The expected output for this program is:

```
1 13 RETURN
1 20 INT 0
1 21 SEMICOLON
```

Tokens

Your assignment should recognize the tokens as per the table below.

- Parenthesis, brackets, commas, and curly brackets; with the token itself as the name.
- All operators for expressions, the name of the token is as follows:

Operator	Name
+	PLUS
-	MINUS
*****	MUL
/	DIV
%	REM
==	EQ
<>	NE
<	LT
<=	LE
>	GT
>=	GE
!	NOT
&	BAND
|	BOR
^	XOR
&&	AND
||	OR

- All operators for statement keywords, the name of the token is the capitalized keyword (e.g., RETURN, IF, FUN, etc.)
- Types “int”, “string”, and “void”. The name of the token is TYPE and the info is a null-terminated string representing the type (e.g., TYPE “int”)
- Variable names: alphanumeric sequences of characters that start with a letter. The name of the token is NAME and the info is the name of the variable. Example:
    - “variable”
        - 1 1 NAME variable
- Integer constants: sequences of numbers. The name of the token is INT and the info is the content of the constant (e.g., INT 473)
- String constants: sequences of characters between double-quotes. The name of the token is STRING and the info is the length of the string, followed by the contents of the string
    - (e.g., STRING 6 CS 473)
- Octal constants: sequences of numbers in base 8 (i.e., between 0 and 7) that start with one leading zero. The name of the token is INT and the info is the content of the constant in base 10 (e.g., the octal 010 should result in the token INT 9)

- Hexadecimal constants: sequences of numbers in base 16 (i.e., numbers between 0 and 9 and capital characters between A and F) starting with 0x. The name of the token is INT and the info is the content of the constant in base 10 (e.g., the hex 0xA should result in the token INT 10)

Name analysis

For programs that contain variable names, your program should end the output by printing the names found, in alphabetical order, followed by the number of times each name was used in the program. For instance, the following program:

```
<?cardinal?>
int name
string variable
void name
int variables
</?cardinal?>
```

Should result in the following output:

```
2 1 TYPE int
2 5 NAME name
3 1 TYPE string
3 8 NAME variable
4 1 TYPE void
4 6 NAME name
5 1 TYPE int
5 5 NAME variables
name 2
variable 1
variables 1
```

You can use existing library functions to sort data-structures.

Escape characters

Strings may have escape characters, which should be replaced by the character they represent. Cardinal has the following espace sequences:

- `n -> new line character
- `t -> tab character
- `” -> double-quote character
- `` -> the escape character itself `

Return value and lexing errors

When lexing a file successfully, your submission should print the output described above and return with code zero. When there is a lexing error, the output is left unspecified but your submission should exit with code 73.

Useful library functions

You can use any functions which are part of the standard C library. In particular, functions in `string.h` are useful for dealing with strings in C as arrays of characters, and string conversion functions in `stdlib.h` are useful to turn strings that represent numbers into native integers. See the links below:

https://pubs.opengroup.org/onlinepubs/7908799/xsh/string.h.html

https://cplusplus.com/reference/cstdlib/

Entry Point

You will modify file `lexer.lex` with all the rules for the lexer to implement the behavior described above. As we will see in class, you will have to modify several sections of the .lex file.

You will also modify file `driver.c`, adding tokens to the large switch-case statement and printing the relevant information for each token as described above.

You will also modify the file `tokens.h` that will define all the tokens used by the lexer, and their format.

Due Date and Resubmission Policy

This assignment is due on **September 13th 2025** (Saturday) at **5pm CST**. There is no late policy.

The code and date used for your submission is defined by the last commit to your Git repository.

You can resubmit your assignment until **September 20th 2025** (following Saturday) at **5pm CST**. Together with your resubmission, you will have to submit a written description of what you changed from the original submission (on Gradescope).

Bonus Points

This assignment has a total of **20% bonus points**.

You can earn 10% extra points by using Piazza as described in the syllabus. Your posts should be public, tagged with the `assignment1` label, and non-anonymous to the instructors to count towards the bonus. You can claim bonus points through **a Gradescope quiz**.

You can earn 10% extra points by fixing all memory leaks in your program. Each passing test without memory leaks is worth 1% bonus points.

Submission and Grading

This assignment is submitted through Github, and has an automatic grade component of 100%. You can check your current grade at any point by submitting your code and checking the autograder. The automatic grade is determined by 10 tests, that will check if your submission outputs the expected result. Each test is worth 10%.

Graduate students

On top of the behavior described above, graduate students have to implement multi-line comments. Graduate students should pick a lexical construction to denote one of these and implement them.

Each regular test is only worth 8% for graduate students. The remaining 20% will be graded as described below.

This language extension will be graded via one video screen-cast (**through Gradescope**) that answers the questions below by explaining how your code:

1. What lexical form do multi-line comments take in your language?
2. Which lex rules did you use?
3. Show one example of a successful program that uses multi-line comments
4. Show one example of a program that contains lexing errors on the multi-line comment

The maximum length for the video is 5 minutes, instructors will stop watching at the 5 minute mark (nothing past that point in the video will be graded). This video should be a screencast of your IDE open on the code submitted, and you should highlight the code. Note that longer videos are not better videos, and you should record a video as short as needed to show all the expressions and answer the questions above.

You can record such a video using Zoom, which you may already have installed to attend lectures remotely. Simply start a meeting (without any other participants), share your screen, and start recording. Note that Zoom requires some time to process your video after you record it, so plan accordingly. Extension requests to upload videos after the due time and date because Zoom is still processing them will be denied.

The final grade for the assignment will be the grade of the original submission, for assignments without a resubmission; or the average between the original grade and the resubmission grade, for assignments with a resubmission. The grade of the original submission includes any bonus points.

Errors and Omissions

If you find an error or an omission, please post it on Piazza as soon as you find it.

Hardcoding and Academic Integrity

Any hardcoding will result in a 0% grade. Hardcoding is when you submit code that detects which test is being run, and simply outputs the expected result. For instance, detecting that test 22 is running, and replacing the usual execution of your submission with `printf("expected result")`.

The academic integrity policy described in the syllabus applies to this assignment. You are responsible for writing all the code that you submit. We will use an automatic tool that detects plagiarism on all submitted code, and we will investigate all instances where plagiarism is more than likely.

Please refer to the syllabus for the full academic integrity policy.