CS305 – Programming Languages

Fall 2025-2026

HOMEWORK 1

Implementing a Lexical Analyzer (Scanner) for Calculator Scientifique (CS)

Due date: October 20th, 2025 @ 23:55

NOTE

Only SUCourse submissions are allowed. Submissions by e-mail are not allowed. Please see the note at the end of this document for the late submission policy.

1 Introduction

In this homework, you will use flex to implement a scanner for the (**CS**) language. Your scanner will be used to detect the tokens in a CS program. CS is a scripting language that will be used to calculate advanced mathematical expressions.

The CS program comprises certain types of tokens. Your scanner should catch them. The details will be explained in the upcoming sections.

An example CS program is given below:

```
    f(x) = 3x^3 + 6x^2 - 84;
    g(x,y) = 8x^2y + xy^2 + 3;

    calculate f(x) + g(x,y);
    calculate f(3) * g(2,y);
    calculate g(6,2);
```

Your program should read and detect the tokens in such CS programs.

Note: In this homework, we are only implementing a scanner. Since the purpose of the scanner is to detect tokens (it does not have anything to do with the grammatical structure of the input), even grammatically incorrect inputs can be successfully processed by the scanner. For example, when the following (grammatically wrong) CS program is processed by the scanner, there will be no complaints, and the tokens appearing in this program will be detected.

```
calculate = 5;
```

Note: The indentation is unimportant; any number of tabs or new lines can be used between the lexemes of tokens. It does not change anything in the end, as long as the lexemes are separated from each other with at least one blank space, tab, or new line. Also note that, in some cases, the lexemes of different tokens may not be separated by a space. For example, in

```
f(x)=75;
```

we have 7 tokens with lexemes '`f`', '`(`', '`x`', '`)`', '`=`', '`75`', and '`;`', without any space between them.

In a CS program, there may be keywords (e.g., calculate), integers (e.g., 12057), the assign operator (=), the semicolon (;), and many others. Your scanner will catch these language constructs (introduced in Section 2 and Section 3) and print out the token names together with their positions (explained in Section 5) in the input file. Please see Section 6 for an example of the required output. The following sections provide extensive information on the homework. Please read the entire document carefully before starting your work on the homework.

2 Keywords & Punctuations

Below is the list of keywords to be implemented, together with the corresponding token names.

Lexeme	**Token**	**Lexeme**	**Token**
+	tPLUS	=	tASSIGN
-	tMINUS	,	tCOMMA
*	tMUL	;	tSEMICOLON
^	tEXP	calculate	tCALCULATE
(	tLPR	D	tDERIVATION
)	tRPR	I	tINTEGRATION

CS is case–sensitive. Only the lexemes given above are used for the keywords. The keyword ‘calculate’ will be caught as the token tCALCULATE. ‘D’ is the lexeme for the derivation operation, and ‘I’ is the lexeme for the integration operation.

3 Identifiers & Integers

Token Type	**Token**
identifier	tIDENTIFIER
integer	tINTEGER

You need to implement identifiers and integer tokens. Here are some rules you need to pay attention to.

- An identifier is a single lower case letter.
- The token name for an identifier is tIDENTIFIER.

  e.g. `f`, `g`, `x`, `z`.
- The token name for an integer is tINTEGER. The valid lexeme for tINTEGER token is any sequence of digits which do not start with 0. In fact, in CS we only have positive integer values in the input.
- Any character that is not a whitespace character and which cannot be detected as part of the lexeme of a token must be considered an illegal character and must be printed out together with an error message (see below, for example, execution part about the error message details). The whitespace characters that should not be considered illegal characters are space or blank (␣), tab (`\t`), carriage return (`\r`), and newline (`\n`).

4 Comments

Comments in a CS program are used to provide explanations, notes, or descriptions for the CS program. There are two ways to indicate comments:

- Single line comments: A comment that starts with // and extends up to the end of the line. Anything following // on the same line is considered a comment. Example:

```
// Defining functions
    f(x) = 9x^6 + x^3 + 8;
    g(x,z) = 2z^3x + x^5 + 67;
// Calculating the expressions
    calculate f(x) + g(a,b);
    calculate f(0) * g(2,y);
    calculate g(1,2);
```

- Multiline comments: These comments begin with (/*) and end with (*/). Anything enclosed within this comment block is considered a multiline comment. This type of comment can span multiple lines. If you encounter another opening comment tag (/*) inside an existing multiline comment block, it should be considered as the start of a separate comment. The number of possible nested comments can vary; that is, it cannot be known in advance, and it is essential to treat each nested comment as a new comment block, each with its own content and its corresponding closing tag (*/). A multiline comment does not have to have a closing tag. In such a case, your scanner does not complain about it and should treat the remainder of the program as if it's still within the comment block. A closing tag (*/) without a corresponding opening tag is not related to comments and should be considered as illegal characters.

Example:

```
/*
    This is a multiline comment
    /*
        a(x) = x^3 + 6x^2 + 8;
        b(x,y) = x^2y + x^3y^2 + 3;
        c() = 89;
    */
    calculate a(x) + b();
    calculate c(3) * d(2,y);
    calculate e(6,2);
*/
```

A comment may appear on any line of a program. Anything that is commented out will be eaten up by your scanner silently. That is, no information will be produced for the comments or the comments' contents. However, the lines used after the comments will be taken into consideration. For example, for the following CS program, an error will be produced for the illegal character #, and the line for the error must be 5:

```
1   /*
2       This is a multiline comment
3   */
4   // Another comment  /* this does not start a block comment
5   #
```

5 Positions

You must keep track of the position information for the tokens. For each token that will be reported, the line number at which the lexeme of the token appears is considered to be the position of the token. Please look at Section 6 to see how the position information is reported together with the token names.

6 Input, Output, and Example Execution

Assume that your executable scanner (which is generated by passing your flex program through flex and bypassing the generated lex.yy.c through the C compiler gcc) is named CSscanner. Then, we will test your scanner on a number of input files using the following command line:

```
CSscanner < example1.cs
```

As a response, your scanner should print out a separate line for each token it catches in the input file. The output format for a token is given below for each token separately:

Detected	Output
tokens from Section 2	⟨*row*⟩_⟨*token*⟩
tokens from Section 3	⟨*row*⟩_⟨*token*⟩_(⟨*lexeme*⟩)
illegal character	⟨*row*⟩_ILLEGAL_CHARACTER_(⟨*lexeme*⟩)

There are placeholders like ⟨*row*⟩ , ⟨*lexeme*⟩, ⟨*token*⟩ in the table above.

- ⟨*row*⟩ should be replaced by the location of the lexeme of the token.
- ⟨*token*⟩ is the token name for the detected token. The names of the tokens to be used here are given in Section 2 and Section 3 above.
- ⟨*lexeme*⟩ should be replaced by the lexeme of the token.

Please note the use of underscore (_) characters as spaces in the output to be generated. **Please use no spaces, and use only a single underscore character as indicated in the output explanations and as given in the examples.** Also, do not insert unnecessary newline characters. Each token must be on a separate line; there should not be a line without any information.

The reason for this is the following. We automatically compare your output with the correct output. Although inserting a space or an additional underscore in the output may seem harmless, an automatic comparison would indicate a difference between your output and the correct output, which can cost you some points in the grading. As an example, let us assume that the following is our CS program:

```
1     f(x) = 3x;
2     g()=5  ;
3     h(x) = x + 2
4
5     ;
6     j(x,y) = (3x^2 - 2) (y^3 + 5);
7     k(x) = x^9 - x^3;
8
9     calculate f(x) + f(3);
10    calculate g() * f(y);
11    calculate h(x) j(a,b);
12    calculate k(x) + D(k,x,3);
```

Then the output of your scanner must be:

```
1_tIDENTIFIER_(f)
1_tLPR
1_tIDENTIFIER_(x)
1_tRPR
1_tASSIGN
1_tINTEGER_(3)
1_tIDENTIFIER_(x)
1_tSEMICOLON
2_tIDENTIFIER_(g)
2_tLPR
2_tRPR
2_tASSIGN
2_tINTEGER_(5)
2_tSEMICOLON
3_tIDENTIFIER_(h)
3_tLPR
3_tIDENTIFIER_(x)
3_tRPR
3_tASSIGN
3_tIDENTIFIER_(x)
3_tPLUS
3_tINTEGER_(2)
5_tSEMICOLON
```

```
6_tIDENTIFIER_(j)
6_tLPR
6_tIDENTIFIER_(x)
6_tCOMMA
6_tIDENTIFIER_(y)
6_tRPR
6_tASSIGN
6_tLPR
6_tINTEGER_(3)
6_tIDENTIFIER_(x)
6_tEXP
6_tINTEGER_(2)
6_tMINUS
6_tINTEGER_(2)
6_tRPR
6_tLPR
6_tIDENTIFIER_(y)
6_tEXP
6_tINTEGER_(3)
6_tPLUS
6_tINTEGER_(5)
6_tRPR
6_tSEMICOLON
7_tIDENTIFIER_(k)
7_tLPR
7_tIDENTIFIER_(x)
7_tRPR
7_tASSIGN
7_tIDENTIFIER_(x)
7_tEXP
7_tINTEGER_(9)
7_tMINUS
7_tIDENTIFIER_(x)
7_tEXP
7_tINTEGER_(3)
7_tSEMICOLON
9_tCALCULATE
9_tIDENTIFIER_(f)
```

```
9_tLPR
9_tIDENTIFIER_(x)
9_tRPR
9_tPLUS
9_tIDENTIFIER_(f)
9_tLPR
9_tINTEGER_(3)
9_tRPR
9_tSEMICOLON
10_tCALCULATE
10_tIDENTIFIER_(g)
10_tLPR
10_tRPR
10_tMUL
10_tIDENTIFIER_(f)
10_tLPR
10_tIDENTIFIER_(y)
10_tRPR
10_tSEMICOLON
11_tCALCULATE
11_tIDENTIFIER_(h)
11_tLPR
11_tIDENTIFIER_(x)
11_tRPR
11_tIDENTIFIER_(j)
11_tLPR
11_tIDENTIFIER_(a)
11_tCOMMA
11_tIDENTIFIER_(b)
11_tRPR
11_tSEMICOLON
12_tCALCULATE
12_tIDENTIFIER_(k)
12_tLPR
12_tIDENTIFIER_(x)
12_tRPR
12_tPLUS
12_tDERIVATION
```

```
12_tLPR
12_tIDENTIFIER_(k)
12_tCOMMA
12_tIDENTIFIER_(x)
12_tCOMMA
12_tINTEGER_(3)
12_tRPR
12_tSEMICOLON
```

Note that the test file content need not be a complete or correct CS program. If the content of a test file is the following:

```
    f(x)=25;\
    g();
```

Then your scanner should not complain about anything and output the following information:

```
1_tIDENTIFIER_(f)
1_tLPR
1_tIDENTIFIER_(x)
1_tRPR
1_tASSIGN
1_tINTEGER_(25)
1_tSEMICOLON
1_ILLEGAL_CHARACTER_(\)
2_tIDENTIFIER_(g)
2_tLPR
2_tRPR
2_tSEMICOLON
```

Also note that some of the test cases may have nested comment lines. In that case, you should handle them. If the content of a test file is the following:

```
1      /*
2  /*
3          This is a nested comment
4          f(x) = x; */
5
6          This is a comment string as well
7      */
8          calculate f(x); // This part should be detected!
```

Then your scanner output should be the following:

```
8_tCALCULATE
8_tIDENTIFIER_(f)
8_tLPR
8_tIDENTIFIER_(x)
8_tRPR
8_tSEMICOLON
```

7 How to Submit

Submit only your flex file (**without zipping it**) on SUCourse. The name of your flex file must be:

username-hw1.flx

where username is your SUCourse username.

If your username is ***husnu.yenigun***, then the name of your flex file must be ***husnu.yenigun-hw1.flx***. It must not be husnuyenigun-hw1.flx or something else.

If you fail to submit your file with this correct name, you will get -20 points.

8 Notes

- **Important**: Name your file as you are told and **don't zip it.** [**-20 points otherwise**]

- Do not copy-paste CS program fragments from this document as your test cases. Copying/pasting from PDFs can create some unrecognizable characters. Instead, all CS code fragments that appear in this document are provided as a text file for you to use.
- Make sure you print the token names precisely as it is supposed to be. Also, make sure you use underscore characters (not spaces), as explained above. You will lose points otherwise.
- No homework will be accepted if it is not submitted using SUCourse.
- You may get help from our TAs, LA, or friends. However, **you must write your flex file by yourself**.
- Start working on the homework immediately.
- If you develop your code or create your test files on your own computer (not on cs305.sabanciuniv.edu), there can be incompatibilities once you transfer them to the cs305 machine. Since the grading will be done automatically on the cs305 machine, we strongly encourage you to develop it on the cs305 machine or at least test your code on the cs305 machine before submitting it. If you prefer not to test your implementation on the cs305 machine, this means you accept to take the risks of incompatibilities. Even if you have spent hours on homework, you can easily get 0 due to such incompatibilities.

LATE SUBMISSION POLICY

Late submission is allowed subject to the following conditions:

- Your homework grade will be decided by multiplying what you get from the test cases by a “submission time factor (STF)”.
- If you submit on time (i.e., before the deadline), your STF is 1. So, you don’t lose anything.
- If you submit late, you will lose 0.01 of your STF for every 5 minutes of delay.
- We will not accept any homework later than 500 minutes after the deadline.
- SUCourse+’s timestamp will be used for STF computation.
- If you submit multiple times, the last submission time will be used.