CS 335 Semester 2022–2023-II: Assignment 1

11th January 2023

Due Your assignment is due by Jan 24 2023 11:59 PM IST.

General Policies

- You should do this assignment ALONE.
- Do not plagiarize or turn in solutions from other sources. You will be PENALIZED if caught.
- We MAY check your submission(s) with plagiarism checkers.

Submission

- Your solution SHOULD use Flex. Do not use alternate software like JFlex and PLY for this assignment.
- Submission will be through Canvas.
- Upload a zip file named "⟨`roll`⟩`-assign1.zip`". The zipped file should contain a directory ⟨`roll`⟩`-assign1`, two sub-directories `problem1` and `problem2`, and the following files:
  - The Flex specification files.
  - A `PDF` file describing any tools that you used, and INCLUDE compilation and execution instructions.
- We encourage you to use the LaTeX typesetting system for generating the PDF file.
- You will get up to TWO LATE days to submit your assignment, with a 25% penalty for each day.

Evaluation

- Write your code such that the EXACT output format (if any) is respected.
- We will evaluate your implementations on a Unix-like system, for example, a recent Debian-based distribution.
- We will evaluate the implementations with our OWN inputs and test cases (where applicable), so remember to test thoroughly.
- We may deduct marks if you disregard the listed rules.

Problem 1 **[30 marks]**

In the following, we describe the lexical specification of a toy programming language called CSEI-ITK, with a file extension ".335". Write a lexical analyzer (i.e., scanner) for CSEIITK using Flex.

Keywords: `array begin case const do downto else end file for function goto if label nil of packed procedure program record repeat set then to type until var while with`

Comments: Comments are any text between { and }. The first occurrence of the terminating character } ends the comment; comments can be multiline but cannot be nested.

White Space: White space is defined as the ASCII space character, horizontal tab character, form feed character, and line terminator characters.

Blanks, tabs, ends of lines, and comments are considered to be separators. The Lexical Analyzer consumes these, but does not return anything.

A structured block starts with a `begin` and ends with `end`.

Operators: + - * / . := = <> < <= >= > ^

The following reserved words are treated as operators: `and or not div mod in`

Delimiters are: , ; : ( ) [ ] ..

Identifiers: Identifiers are unlimited-length sequences beginning a letter, followed by any number of letters or digits.

Strings: A string literal consists of zero or more characters enclosed in double quotes. Characters such as newlines may be represented by escape sequences.

Numeric literals: Numeric literals are expressed only in decimal (base 10). Unsigned numbers must begin with a digit; signed numbers will have a leading "-". There must be at least one digit following a decimal point for floating-point numbers. A number may be followed by a signed decimal exponent, in which case it is floating-point irrespective of whether there is a decimal point or not.

The numeric literals do not need any format suffixes like "l", "f", "d", or "L". The toy language does not support boolean, char, hex, and oct literals.

Check the example files to learn about the usage. The meaning of the programs are not important for this assignment.

- The output should list and classify all unique lexemes into proper syntactic categories.
- Redirect the statistics to a `CSV` file, with a header column of the form **Lexeme** | **Token** | **Count**.
- The output should ordered by the occurrence of lexemes/words in the input. For example, the first non-header row in the output should correspond to "program" if the first word in the input file is "program". See the file `public2.335`.
- The tool should report any tokenization error due to "illegal characters" in the input program. The error report should highlight the line number and the illegal character.
- Your scanner can terminate after reporting the first error. You may optionally continue beyond the first error.
- Assume that the input characters will only be ASCII.

Example

Input. Consider the following input CSEIITK program.

```
program example(output);
var  s: real; i: integer;
begin
   s := 3.0;
   for i:= 1 to 5 do
       s := 0.5 * i;
    writelnf (s)
end
```

Lexeme	Token	Count
`program`	Keyword	1
`example`	Identifier	1
`(`	Separator	2
`output`	Identifier	1
`)`	Separator	2
`;`	Separator	5
`var`	Keyword	1
`s`	Identifier	4
`:`	Separator	2
`real`	Identifier	1
`i`	Identifier	3
`integer`	Identifier	1
`begin`	Keyword	1
`:=`	Operator	3
`3.0`	Literal	1
`for`	Keyword	1
`1`	Literal	1
`to`	Keyword	1
`5`	Literal	1
`do`	Keyword	1
`0.5`	Literal	1
`*`	Operator	1
`writelnf`	Identifier	1
`end`	Keyword	1

Problem 2 **[70 marks]**

Create a lexer for the Java language. The complete lexical structure for Java 17 is publicly available.

- The output should list and classify all unique lexemes into proper syntactic categories.
- Redirect the statistics to a `CSV` file, with a header column of the form **Lexeme** | **Token** | **Count**.

- The output should ordered by the occurrence of lexemes/words in the input. For example, the first non-header row in the output should correspond to "program" if the first word in the input file is "program". See the file `public2.335`.
- The tool should report any tokenization error due to "illegal characters" in the input program. The error report should highlight the line number and the illegal character.
- Your scanner can terminate after reporting the first error. You may optionally continue beyond the first error.
- Assume that the input characters will only be ASCII. Ignore the complexities from handling Unicode characters and escape sequences.

Example

Input. Consider the following input Java program.

```java
public class Program {
    /*
     * This is my first java program.
     */

    public static void main(String[] args) {
        int data = 50; // declaration
        boolean flag = false;
    }
}
```

Expected output.

Lexeme	**Token**	**Count**
`public`	Keyword	2
`class`	Keyword	1
`Program`	Identifier	1
{	Separator	2
`static`	Keyword	1
`void`	Keyword	1
`main`	Identifier	1
`(`	Separator	1
`String`	Identifier	1
[	Separator	1
]	Separator	1
`args`	Identifier	1
`)`	Separator	1
`int`	Keyword	1
`data`	Identifier	1
`=`	Operator	2
`50`	Literal	1
;	Separator	2
}	Separator	2
`boolean`	Keyword	1
`flag`	Identifier	1
`false`	Literal	1