

CECS 342

Lab assignment 1

Due date: Wednesday, February 14

20 points

This lab assignment is a group assignment of two students.

Problem:

Consider the following example of an assignment statement:

```
result = oldsum - value / 100;
```

Following are the tokens and lexemes of this statement:

Token	Lexeme
IDENT	result
ASSIGN_OP	=
IDENT	oldsum
SUB_OP	-
IDENT	value
DIV_OP	/
INT_LIT	100
SEMICOLON	;

Lexical analyzers extract lexemes from a given input string and produce the corresponding tokens. In the early days of compilers, lexical analyzers often processed an entire source program file and produced a file of tokens and lexemes. Now, however, most lexical analyzers are subprograms that locate the next lexeme in the input, determine its associated token code, and return them to the caller, which is the syntax analyzer. So, each call to the lexical analyzer returns a single lexeme and its token. The only view of the input program seen by the syntax analyzer is the output of the lexical analyzer, one token at a time.

The lexical-analysis process includes skipping comments and white space outside lexemes, as they are not relevant to the meaning of the program. Also, the lexical analyzer inserts lexemes for user-defined names into the symbol table, which is used by later phases of the compiler. Finally, lexical analyzers detect syntactic errors in tokens, such as ill-formed floating-point literals, and

report such errors to the user.

Suppose we need a lexical analyzer that recognizes only arithmetic expressions, including variable names and integer literals as operands. Assume that the variable names consist of strings of uppercase letters, lowercase letters, and digits but must begin with a letter. Names have no length limitation. The first thing to observe is that there are 52 different characters (any uppercase or lowercase letter) that can begin a name, which would require 52 transitions from the transition diagram's initial state. However, a lexical analyzer is interested only in determining that it is a name and is not concerned with which specific

name it happens to be. Therefore, we define a character class named LETTER for all 52 letters and use a single transition on the first letter of any name.

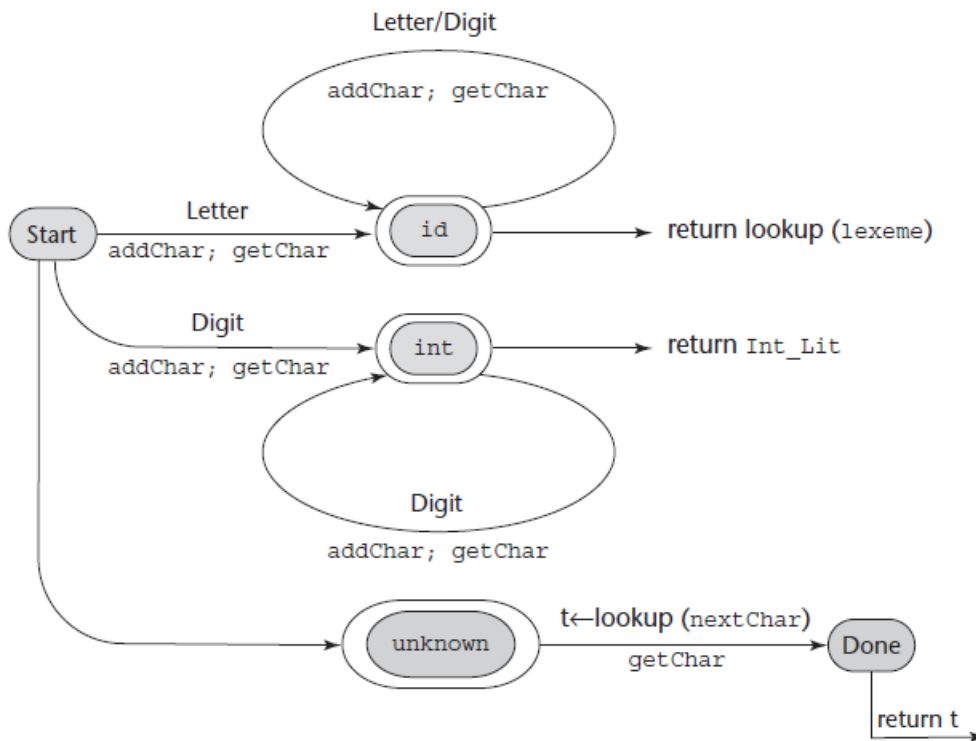
Another opportunity for simplifying the transition diagram is with the integer literal tokens. There are 10 different characters that could begin an integer literal lexeme. This would require 10 transitions from the start state of the state diagram. Because specific digits are not a concern of the lexical analyzer, we can build a much more compact state diagram if we define a character class named DIGIT for digits and use a single transition on any character in this character class to a state that collects integer literals. Because our names can include digits, the transition from the node following the first character of a name can use a single transition on LETTER or DIGIT to continue collecting the characters of a name.

Next, we define some utility subprograms for the common tasks inside the lexical analyzer. First, we need a subprogram, which we can name `getChar`, that has several duties. When called, `getChar` gets the next character of input from the input program and puts it in the global variable `nextChar`. `getChar` must also determine the character class of the input character and put it in the global variable `charClass`. The lexeme being built by the lexical analyzer, which could be implemented as a character string or an array, will be named `lexeme`.

We implement the process of putting the character in `nextChar` into the string array `lexeme` in a subprogram named `addChar`. This subprogram must be explicitly called because programs include some characters that need not be put in `lexeme`, for example the white-space characters between lexemes. In a more realistic lexical analyzer, comments also would not be placed in `lexeme`.

When the lexical analyzer is called, it is convenient if the next character of input is the first character of the next lexeme. Because of this, a function named `getNonBlank` is used to skip white space every time the analyzer is called. Finally, a subprogram named `lookup` is needed to compute the token code for the single-character tokens. In our example, these are parentheses and the arithmetic operators. Token codes are numbers arbitrarily assigned to tokens by the compiler writer.

The state diagram below describes the patterns for our tokens. It includes the actions required on each transition of the state diagram.



Attached is a partial implementation of a lexical analyzer specified in the state diagram above, including a main driver function for testing purposes:

You need to complete the program written in C. Name the program lab2.

Create a text file with the following strings to test your program:

```
(sum + 47) / total
oldsum - value /100
```

Output:

```
Next token is: 25, Next lexeme is (
Next token is: 11, Next lexeme is sum
Next token is: 21, Next lexeme is +
Next token is: 10, Next lexeme is 47
Next token is: 26, Next lexeme is )
Next token is: 24, Next lexeme is /
Next token is: 11, Next lexeme is total
```

Next token is: 11, Next lexeme is oldsum
Next token is: 22, Next lexeme is -
Next token is: 11, Next lexeme is value
Next token is: 24, Next lexeme is /
Next token is: 10, Next lexeme is 100
Next token is: -1, Next lexeme is EOF

Grading;

1. All group members submit the following:
 - a. Lab assignment file lab1.c file or lab1.py (20 points deduction for missing the file)
 - b. PDF file with code and runtime output (-10 points deduction for missing the file)
 - c. Write on the comment the completion of the lab assignment for each team member. All the team members must agree with the completion of each team member.
2. Grading criteria:
 - The program is efficient and runs successfully with correct output (100% completion)
 - The program run successfully with incorrect output (60% completion)
 - The program is incomplete and has syntax errors (30% completion)