Project Mini-Pascal (Lexical-analyzer)
Version I
Csci465
Fall 2020
Due October 1, 2020

Write a lexical-analyzer and source-handler module for a Pascal compiler. Your lexical analysis module should transform the input (i.e., the program input in Pascal) into a stream of tokens. Documents all design decisions!!! At this point, you may create two modules:

1. **I/O module** that does input buffering and generates the program listing,
2. **Lexical analyzer** called **getsym module** that returns one symbol at a time.

The **I/O module** communicate as follows:

- **Receives** entire input lines from the input file;
- **Receives** error messages from **getsym, parser**, and other **modules**;
- **Sends** character stream to the **getsym** module using getch (var ch: char);
- **Sends** the (edited) input lines to the **lister file** (i.e., output file)
- **Sends** the error messages to the lister file (i.e., output file), marking the error position.

At this point, you should be able to handle arbitrary Pascal programs. Input could be handled **by records** i.e., you will read in entire 80-char records (e.g., **read (line)**, where line can be **an array [1..80] of char**). Input lines should be available to be printed out as part of a **lister file.**

The **getsym module** communicates as follows:

- **Receives** character stream from lister module (input);
- **Sends** error messages to lister module; (output)
- **Sends** token stream to the Parser module (Syntax analyzer).

These interactions (sending/receiving) all occur through procedure calls/function calls; receiving occurs directly via procedure calls and sending occurs when one module "allows" its procedures to be used by another. Ultimately, the main section of your compiler will be what drives everything.

For example, the following enumerated type tries to include an entry for Pascal **keywords** (reserved words) and every **punctuation** symbol (new symbols can be added

provided that all productions rules effected by the new symbols are rewritten). The important task of your lexical analyzer is to decide which of these symbols appears next in the input stream.

```
Type
   tokentype =
                (andsym,        arraysym,       beginsym,       charsym,        chrsym,
                divsym,         dosym,          elsesym,        endsym,         ifsym,
                integersym,     modsym,         notsym,         ofsym,          orsym,
                ordsym,         proceduresym,   programsym,     readsym,        readlnsym,
                thensym,        varsym,         whilesym,       writesym,       writelnsym,

                (* end of reserved words *)

                plus,           minus,          times,
                (* +            -               *          *)
                less,           lessequal,      notequal,       greater,  greaterequal,   equal,
                (*  <           <=              <>              >         >=              = *)
                assign,         colon,          semicolon,      comma,
                (* :=           :               ;               ,         *)
                lparen,         lbrack,         rparen,         rbrack,   period,
                (* (            [.              )               .]        .         *)

                identifier,     (*letter (letter|digit)*                           *)
                number,         (* digit (digit)*                                  *)
                quotestring,    (* such as 'here ' 's a string'                    *)
                litchar,        (* quoted strings of length=1: e.g.,    'a', 'b' *)
                eofsym,         (* returned by getsym at end of file               *)
                illegal);       (* for lexical errors: e.g., #id                   *)

Type
        Symboltype = record
                        Lexeme: tokentype;
                        Spelling: stringtype;       (* array or table entry *)
```

You may begin with method/procedure named getsysm, which takes as an input a variable of type Symboltype. The procedure getsym(s: symboltype) should return one symbol at a time. Comments should **not** appear in this symbol stream; getsym should skip them. Identifiers should appear in the spelling field, padded with blanks. Numbers should appear in the spelling field as **strings of digits**; only unsigned integer strings need appear. If you encounter a quote mark, ', beginning a character string, store the actual string in (in a single call to getsym) in a global variable **gstring**, and its length in **gslength**. The symbol.lexeme field will then be set to string. Set symbol.lexeme to eofsym only if you reach the end of file.

Some changes will be made in this lexical analyzer between now and your final compilers. However, this will be a reasonable start.

You need to write a "*driver program*" that calls your getsym(symbol) repeatedly until eofsym (i.e., end of file) is reached. Your driver may look like this:

```
Repeat
        getsym(symbol);
        If symbol.Lexeme = identifier then begin
           writeln('Identifier':20; symbol.spelling:20)  //print statement is Pascal
        end else if symbol. Lexeme = number then begin
          < similar>
        end else if symbol. Lexeme = string then begin
           writeln('string');
           writeln(gstring);
        end else
           <print appropriate message corresponding to symbol.Lexeme >
    Until symbol.Lexeme = eofsym.
```

You MUST use the following Pascal program as an **input file** to your lexical analyzer.

```
program example(input,output);
var x,y:integer;
function gcd(a,b:integer):integer;
begin{gcd}
if b=0then gcd:=a else gcd:=(b,a mod b)
end;{gcd}
begin{example}
read(x,y);
write(gcd(x,y));
end.
```

Your lexical analyzer must generate **output file** having two columns as follows:

LEXEME	SPELLING
PROGRAM	program
ID	example
…	…