Project Mini-Pascal (Lexical-analyzer)
Version I
Csci465
Fall 2017
Due September 28, 2017

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 will create two modules: 1) an **I/O module** that does input buffering and generates the program listing, and 2) the lexical analyzer, say, **getsysm module**.

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 **getsysm** module using getch (var ch: char);
Sends the (edited) input lines to the **lister file** (i.e., output file), and the assembler 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 should be handled **by records**; i.e., you will read in entire 80-char records (with **readln(line)**, where line is **a packed array [1..80] of char**). Input lines should be available to be printed out as part of a **listing** 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 complier will be what drives everything.

The following enumerated type includes an entry for every Pascal keyword and every punctuation symbol (new symbols can be added provided that all productions rules effected by the new symbols are rewritten). The heart of your lexical analyzer is to decide which of these symbols appears next in the input stream.

Csci 465 (Principle of Translation)   08/23/2017   Dr. Reza

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
             Kind: tokentype;
             Spelling: stringtype;   (* array or table entry *)

The procedure getsym(var 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.kind field will then be set to string. Set symbol.kind 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 is reached. Your driver may look like this:

```
Repeat
        Getsym(symbol);
        If symbol.kind = identifier then begin
            writeln('Identifier':20; symbol.spelling:20)
        end else if symbol.kind = number then begin
            < similar>
        end else if symbol.kind = string then begin
```

writeln('string');
writeln(gstring);
end else
　　　　<print appropriate message corresponding to symbol.kind>
Until symbol.kind = 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: the first columns as follows:

LEXEME	SPELLING
PROGRAM	program
ID	example
...	...