CSE 4714 / 6714 — Programming Languages
Project Part 1

Our first programming assignment for the project is to write a lexical analyzer for a subset of the language TIPS, which itself is a subset of the Pascal programming language. TIPS stands for *Ten Instruction Pascal Subset*; see the attached book (which dates from the Pleistocene epoch of computing).

The job of a lexical analyzer is to return the lexemes (i.e., fundamental syntactical elements) in the input program to a parser for further analysis. You will use *C++* and *flex* for this assignment.

Assignment

Use *flex* to generate the lexical analyzer using the language specification shown in the following table. A set of files that you will use as a starting point for your code is in the attached `Part_1_Starting_Point.zip` file.

Tables of the TIPS lexemes:

Keyword Lexemes	Token Identifier Value	Token Constant
BEGIN	1000	TOK_BEGIN
BREAK	1001	TOK_BREAK
CONTINUE	1002	TOK_CONTINUE
DOWNTO	1003	TOK_DOWNTO
ELSE	1004	TOK_ELSE
END	1005	TOK_END
FOR	1006	TOK_FOR
IF	1007	TOK_IF
LET	1008	TOK_LET
PROGRAM	1009	TOK_PROGRAM
READ	1010	TOK_READ
THEN	1012	TOK_THEN
TO	1013	TOK_TO
VAR	1014	TOK_VAR
WHILE	1015	TOK_WHILE
WRITE	1016	TOK_WRITE

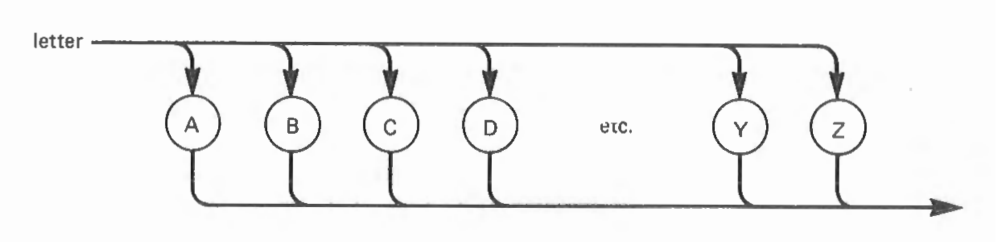
Datatype Specifier Lexemes	Token Identifier Value	Token Constant
INTEGER	1100	TOK_INTEGER
REAL	1101	TOK_REAL

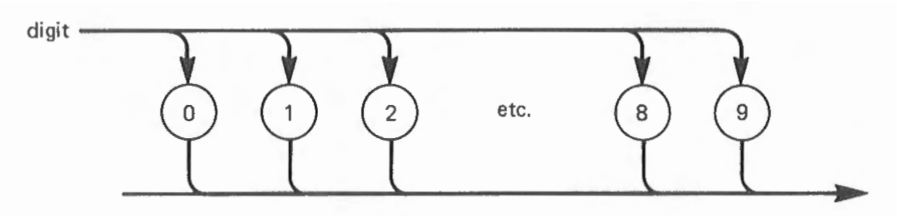
Punctuation Lexemes	Token Identifier Value	Token Constant
;	2000	TOK_SEMICOLON
:	2001	TOK_COLON
(	2002	TOK_OPENPAREN
)	2003	TOK_CLOSEPAREN
{	2004	TOK_OPENBRACE
}	2005	TOK_CLOSEBRACE

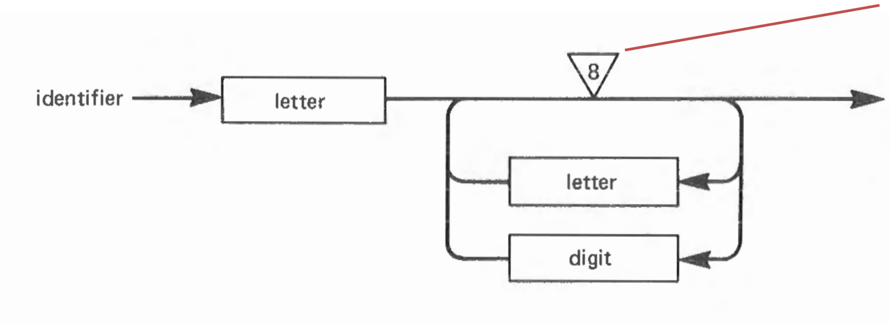
Operator Lexemes	Token Identifier Value	Token Constant
+	3000	TOK_PLUS
-	3001	TOK_MINUS
*	3002	TOK_MULTIPLY
/	3003	TOK_DIVIDE
:=	3004	TOK_ASSIGN
=	3005	TOK_EQUALTO
<	3006	TOK_LESSTHAN
>	3007	TOK_GREATERTHAN
<>	3008	TOK_NOTEQUALTO
MOD	3009	TOK_MOD
NOT	3010	TOK_NOT
OR	3011	TOK_OR
AND	3012	TOK_AND

Useful Abstraction Lexemes	Token Identifier Value	Token Constant
identifier	4000	TOK_IDENT
integer literal	4001	TOK_INTLIT
floating-point literal	4002	TOK_FLOATLIT
string literal	4003	TOK_STRINGLIT
end of file	5000	TOK_EOF
	6000	TOK_UNKNOWN

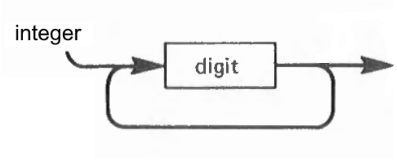
Syntax diagrams for some of the *Useful Abstractions* lexemes:

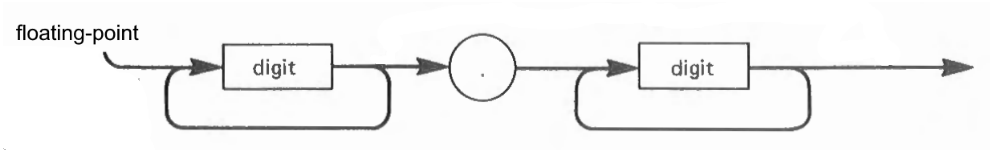






identifiers are at most 8 characters long





Notes

(1) Identifiers and keywords only consist of uppercase letters.

(2) Identifiers can be at most 8 characters long. This length limit can be determined during lexical analysis by measuring the length of the `yytext` variable. In the flex file, the code block can contain C code in addition to `return TOK_IDENT`.

(3) Integer literals only consist of digits. A negative integer literal should lexically scan as a minus sign followed by an integer literal. The issue of the maximum and minimum allowed integer literal is not handled during lexical analysis. Similar comments apply to floating point literals.

(4) Sequences of letters called strings (`TOK_STRINGLIT`) are enclosed between single quote marks. The maximum length of a string is 80 characters. Examples are:

```
'THE BANK BALANCE IS:'
'HAPPY BIRTHDAY TO YOU'
'WHAT CHARACTERS ARE PERMISSIBLE IN character STRINGS?'
```

`''` (the empty string)

Any character may appear in a string, including whitespace. *Except:* a single quote mark may not appear in a string, and a newline may not appear in a string.

(5) Other than separating lexemes, whitespace (space, tab, newline) should be ignored by your lexical analyzer (except for spaces and tabs in a string).

(6) Ambiguity is resolved in favor of longer lexemes; therefore, the word `IFFINESS` in the input stream should create an identifier token, and not an `IF` keyword token followed by an identifier token.

(7) The driver program implements the following behavior: If started with no arguments, the program runs interactively, which allows easy testing. If started with a file name, the program processes that file. If a lexical error is found, the lexical analyzer prints an error message and keeps running.

(8) Place the following header in your `rules.l` file. Modify the header to use your name, etc.

```
/*********************************************************************
  Name: Any Student             NetID: as3
  Course: CSE 4714              Assignment: Part 1
  Programming Environment:
  Purpose of File: Contains the ....
*********************************************************************/
```

(9) See the provided sample inputs and outputs for examples of the completed program’s execution.

Helpful Tips

(1) The following command runs the lexical analyzer on the file `input1.in,` and then uses a Unix *pipe* to compare the output to the file `input1.correct`. This is easier than saving the output to a temporary file.

```
$ ./tips_lex input1.in | diff - input1.correct
```

(2) The following command runs the lexical analyzer on the file `input1.in,` and then uses a Unix *pipe* to print the output up to the first error, and then stops. This allows that error to be investigated. *How it works:* The *grep* command searches for the word *ERROR* (which itself is a regular expression!); when found, it prints up to 1000 preceding lines of context (`-B 1000`), and stops after one match (`-m 1`). Check it out!

```
$ ./tips_lex input1.in | grep -B 1000 -m 1 ERROR
```

Deliverables

Place all of the source files (including files that you did not modify) needed to build your program using `make` in a zip file named *`yournetid`*`_part_1.zip` . For example, my submission would be named `pmb137_part_1.zip` .

Do NOT include object files (`*.o`), generated source code files (`lex.yy.c`), or executable files (`*.exe`) .

Upload your zip file to the assignment.