CS F363 Compiler Construction
Assignment-2

1 Language Specification

Consider a programming language, basic PASCAL, which supports integer constants, keywords, variables, expressions, assignments, conditional statements, loops, arrays, single line comments (using //) and read and write statements:

1. **Keywords:** program, integer, real, boolean, char, var, to, downto, if, else, while, for, do, array, and, or, not, begin, end, read, and write. The keywords are not case-sensitive. (Include any keywords used in the below, but missing from the list.)

2. **Variables or Identifiers:** The name of a variable can be composed of letters, digits, and the underscore character. It must begin with a letter. The variable names are not case-sensitive, so uppercase and lowercase letters mean same here. However, the keywords are not allowed to use as a variable names.

   All variables must be declared before we use them in the program. All variable declarations are followed by the *var* keyword. A declaration specifies a list of variables, followed by a colon (:) and the type. Syntax of variable declaration is

   ```
   var
        variable_list : type;
   ```

   Here, *variable_list* is a list of variables separated by a comma (,) and *type* from the list $\{char, integer, real, boolean\}$.

   Example:

   ```
   var
           age , weekdays : integer;
           taxrate , net_income: real;
           choice , isready: boolean;
           initials , grade: char;
   ```

   Note that at the variable declaration, assigning a value to one or more variables is not allowed.

3. **Operators:** An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. We allow the following types of operators:

   - Arithmetic operators:
     + (*addition*) ,
     − (*subtraction*),
     ∗ (*multiplication*),
     / (*division*, return real value),
     % (*reminder*, returns integer type)
   - Relational operators:
     = (equals, comparison operator)
     <> (not equal to)
     <, >, <=, >= (these operators have usual meaning)
   - Boolean operators:
     and : boolean AND operator, if both the operands are true, then condition becomes true.
     or : boolean OR Operator. If any of the two operands is true, then condition becomes true.
     not: boolean NOT Operator. Used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.

4. **Statements:** In the program, a statement can be of any of the following:

   (a) **Read and Write statements:**

   - **write**: it prints the text or values of variables on the screen. The syntax is given below:

```
write("text");    // prints the text on the screen.
write(variable_list);  //prints the values of the variables on the screen.
```

     Note that in *write*("*text*"), text is just a sequence of characters, it will not have any meaning. Hence, do not tokenize the text.
     example:
     write("Welcome to CS F363");
     write(day, age, cost); //day, age, cost are variables and the type of these variables need not be the same.

   - **read**: takes the value from the user as an input and the syntax is given below:

```
read(id);  //where id is a varaible in the program
```

   (b) **Assignment statement:** To assign a value to a variable, follow this syntax:

```
variable_name := expression;
```

   here the *expression* is a single value or a variable, or an arithmetic expression over constants/variables with arithmetic operators mentioned above.

   (c) **Block of statements:** A set of one or more statements is consider as a block and each block starts with **begin** and ends with **end**.

```
begin
statement_1;
....
......
statement_k;

end
```

   (d) **Conditional statements:** We allow if-then and if-then-else statements as in PASCAL language.

   - **Simple if**: *if condition then S*;
     where *condition* is a Boolean or relational expression and S is a compound statement (block of statements). See an example below:

```
i:= 10;

if i > 10 then
        begin
          i :=10;
          i := i - 1;
          write(i);
        end;
```

   - **if-then-else**: *if condition then* S_1 *else* S_2;
     where *condition* is a Boolean or relational expression, and S_1 and S_2 are compound statements (block of statements). Note that there is no ; (semicolon) after S_1.
     example:

```
i:= 10;

if i > 10 then
        begin
          i :=10;
          i := i - 1;
          write(i);
        end
else
        begin
          i:=20;
          write(i);
        end;
```

For the sake of simplicity, we do not consider nested if statements.

(e) **Looping statements:** We allow while-do and for-do loops as in PASCAL. For the sake of simplicity, we do not consider nested loops.

- **while-do**: *while condition do S*;
  where *condition* is a Boolean or relational expression and S is a compound statement (block of statements). See an example below:

```
while number>0 do // it was mentioned in the initial version, (number > 0)
     begin
        sum := sum + number;
        number := number - 1;
      end;
```

- **for-do**: *for variable−name := initial_value to* [*downto*] *final_value do S*;
  Where, the $variable - name$ specifies a variable of ordinal type, called control variable or index variable; *initial_value* and *final_value* values are values (or arithmetic expressions) that the control variable can take; and S is the body of the for-do loop that is a group of statements / block statements. See examples below:

```
b:=20;

 for a := 10 to b+10  do

begin
       write("value of a: ");
       write(a);
end;
```

```
b:=20;
a:=0;
//example to illustrate downto
for a := a+5 downto b-15  do

begin
        write("value of a: ");
        write(a);
end;
```

5. **Program structure:** the program starts with keyword "program" followed by the name of the program and terminates with ; (semicolon). Next is the variable declaration section, then followed by main program block. The main program block starts with keyword "begin" and ends with keyword "end" followed by a period (.).

```
program name_of_the_program;

var
        variabl_list:type;    //declaration of varaibles

begin // main program block starts




end. // the end of main program block
```

Here, *name_of_the_program* follows the rules of variables.

Example:

```
program AddTwoNumbers;


var
        num1, num2, sum: Integer;

begin
        Write("Enter the first number: ");
        read(num1);

        Write("Enter the second number: ");
        read(num2);

        // Perform addition
        sum := num1 + num2;

        // Display the result
        Write("The sum is ");
        write(sum)
end.
```

6. **Arrays**: we consider only one-dimensional arrays as in PASCAL and we consider a static declaration of array. The syntax is given below:

```
aray_name: array[c1..c2] of type;
```

where *array_name* is an identifier (variable), *array* and *of* are keywords. Further, $c1$ and $c2$ are integer constants such that $c1 \leq c2$, and $type \in \{integer, char, real, boolean\}$. See an example in the below:

```
program ArraySum;

var
        numbers: array[1..10] of Integer;
        i, sum: Integer;

begin
        // Read 10 values into the array
        write("Enter 10 integer values: ");
        for i := 1 to 10 do
          begin
                read(numbers[i]);
          end;

        // Calculate the sum of the values in the array
        sum := 0;
        for i := 1 to 10 do
          begin
                sum := sum + numbers[i];
          end;

        // Display the sum
        write("The sum is : ");
        write(sum);


end.
```

2 Tasks

1. **Lexical Analysis** ***[6 marks]*** - For this phase, write a LEX program that brakes the input program into tokens (identifiers, operators, numbers, keywords, punctuators, etc.).
2. **Syntax Analysis** ***[6 marks]***- For this phase, write a YACC program that takes tokens generated after lexical analysis and checks the given input program is syntactically correct or not. A symbol table is created with the list of tokens obtained from the input. No need to print symbol take at this phase.
3. **Semantic Analysis** ***[8 marks]***- Extend your YACC program written in the last phase to examine semantic errors and the abstract syntax tree is printed.

   Semantic errors like type checking, undeclared variables, multiple declarations of variables, and using the variable before a value is set to it are verified.
4. **Code Generation** ***[8 marks]*** - Extend your YACC program in the last phase to generate the intermediate code in three address code format.
5. **Final stage:** ***[8 marks]*** Finally, extend your YACC program to print the output of the given input program. Also, print the symbol table.

3 Instructions

1. You work on the assignment with your groups members and strongly discouraged to discuss with other group members.
2. You can refer internet/web resources only to understand the syntax of PASCAL language.
3. Taking code from internet/web is strictly prohibited. If you do so, it will lead to severe penalty.
4. Submit the assignment in two parts:

   (a) **Part-1**: Submit the tasks 1, and 2 (Dead line: 20 April 2024 11:59 PM)

   (b) **Part-2**: Submit the tasks 3, 4 and 5 (Dead line: 30 April 2024 11:59 PM)
5. Late submissions will be allowed up to maximum 24 hrs after the deadline with penalty of 2% per each hour.

4 Submission guidelines

1. Only one submission per group.
2. Place the LEX and YACC programs (name the programs with your group submission code) for each task in a separate folder and name the folder with the task name.
3. Create a *readme.txt* file that contains the information of the group members and mention how to compile the programs for each task.
4. Finally, place all the folders and readme.txt inside a new folder folder (name it with your group submission code) and zip the folder and submit it.
5. Your programs must execute on Ubuntu 23.04.

5 Input and Output formats

1. Input will be given in a file and the name of the file will be given at the runtime.
2. You print the output on the terminal and the format for each task is given below.
3. **Lexical Analysis**: print the list of valid tokens in the given input program in the following format

line number	lexeme	token type

4. **Syntax Analysis**: if the input program has any syntax error print "syntax error" message, otherwise print "valid input". (No need to print the type of the syntax error.)
5. **Semantic Analysis**: Print the abstract syntax tree (if the input program has no syntax errors). For the structure and further details refer the web link `https://www.epaperpress.com/lexandyacc/calcg.html`

   If a variable is used before declaration, print an error message "undeclared variable" along with the name of the variable.

   If a variable is declared more than once, print an error message "multiple declarations of a variable" along with the name of the variable.

   Further, print if there are any type mismatches in the given input program like, a real value is assigned to an integer variable (similar cases), a character type is added to integer type, etc.
6. **Code Generation**: print the three-address code generated in the form of **Quadruple** (see lab sheet 10 for the exact structure (will upload in the coming weeks).)
7. **Final stage**: print the output of the given input and print the symbol table in the following format:

Variable	Type	Value

————————————————All the best————————————————-