Design an original programming language using YACC.

1. Your language should should include

    a. type declarations :

        - predefined types (int , float, char, string, bool),
        - array types
        - user defined data types (similar to classes in object orientated languages, but with your own syntax):

            - provide specific syntax to allow initialization and use of variables of user defined types
            - provide specific syntax for accessing fields and methods

    b. variable declarations/definition, constant definitions, function definitions

    c. control statements (if, for, while, etc.), assignment statements;

    d. assignment statements should be of the form: *left_value = expression* (where left_value can be an identifier, an element of an array, or anything else specific to your language)

    e. arithmetic and boolean expressions

    f. function calls which can have as parameters: expressions, other function calls, identifiers, constants,etc.

        ➔ Your language should include a predefined function *Eval(arg)* (arg can be an arithmetic expression, variable or number) and a predefined function *TypeOf*(*arg*)

        ➔ Your programs should be structured in 4 sections: a section for global variables, a section for functions, a section for user defined data types and a special function representing the entry point of the program

1. create a symbol table for every input source program in your language, which should include:

    a. information regarding variable or constant identifiers (type, value);

    b. information regarding function identifiers (the returned type, the type and and name of each formal parameter);

    The symbol table should be printable in two files: symbol_table.txt and symbol_table_functions.txt (for functions)

3) provide semantic analysis and check that:

    a) any variable that appears in a program has been previously defined and any function that is
    called has been defined
    b) a variable should not be declared more than once;
    c) all the operands in the right side of an expression must have the same type (the language should not support casting)
    d) the left side of an assignment has the same type as the right side (the left side can be an element of an array, an identifier etc)
    e) the parameters of a function call have the types from the function definition
    Detailed error messages should be provided if these conditions do not hold (e.g. which variable is not defined or it is defined twice and the program line);
    <u>A program in your language should not execute if there exist semantic or syntactic errors!</u>

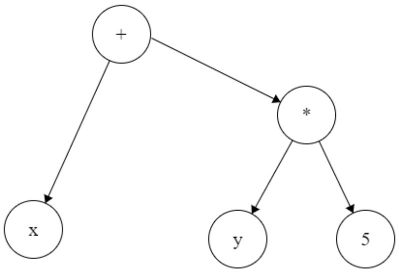
4) (3pt) Implement Eval and TypeOf

- Implement TypeOf
  Remark: TypeOf(x + f(y)) should cause a semantic error if TypeOf(x) != TypeOf(f(y)) (see 3(c) above)
- In order to implement Eval, build abstract syntax trees (AST) for the arithmetic expressions in a program;

if type of *expr* is int , for every call of the form *Eval(expr)* , the AST for the expression will be evaluated and the actual value of *expr* will be printed.
Also, for every assignment instruction *left_value* = *exp*r (left_value is an identifier or element of an array with int type), the AST will be evaluated and its value will be assigned to the left_value
AST: abstract syntax tree - built during parsing of expressions
abstract syntax tree for an arithmetic expression:

- inner nodes: operators
- leafs: operands of expressions (numbers, identifiers, vector elements, function calls etc)

Ex: x+y * 5



- write a data structure representing an AST
- write a function which builds an AST:
  - buildAST(root, left_tree, right_tree, type)
    - root: a number, an identifier, an operator a vector element, other operands
    - type: an integer/ element of enum representing the root type
    - if expr is expr1 op expr2 (op is an operator)
    - expr.AST = buildAST(op, expr1.AST, expr2.AST, OP) //OP denotes the root type
    - if expr is an identifier X
    - expr.AST = buildAST(X, null, null, IDENTIFIER)
    - if expr is a number n
    - expr.AST = buildAST(n, null, null, NUMBER)
    - if expr is other operand
    - expr.AST = buildAST(“operand”, null, null, OTHER)
- (expr.AST denotes the AST corresponding to expression *expr*)
- write a function evalAST(*ast*) which evaluates an AST and returns an int:
  - if *ast* is a leaf labeled with:
    - a number: return the number
    - an id: return the value of the identifier
    - anything else: return 0
  - else (*ast* is a tree with the root labeled with an operator):
    - evalAST for left and right tree
    - combine the results according to the operation