Design an original programming language using YACC. Your language should should include

- Predefined types (int , float, char, string, bool), array types and 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 **0.5pt;**
- variable declarations/definition, constant definitions, function definitions **0.5pt;**
- control statements (if, for, while, etc.), assignment statements **0.25;**
  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)
- arithmetic and boolean expressions **1.5pt**
- function calls which can have as parameters: expressions, other function calls, identifiers, constants,etc. **0.75pt**

➔ Your language should include a predefined function *Print* (with two parameters, a string and an int or bool or float (eg: Print("value of expression x+2 is:",x+2))

➔ Your programs should be structured in 3 sections: one section for global variables, another section for functions and user defined data types and a special function representing the entry point of the programm

Your yacc analyzer should:

1) do the syntactic analysis of the program

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

- information regarding variable or constant identifiers (type, value – for identifiers having a predefined type, scope: global definition,defined inside a function , defined inside a user defined type, or any other situation specific to your language) **1.pt**
- information regarding function identifiers (the returned type, the type and and name of each formal parameter, whether the function is a method in a class etc) - **1 pt**

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:

- any variable that appears in a program has been previously defined and any function that is called has been defined **0.25**
- a variable should not be declared more than once; **0.25**
- a function is not defined more than once with the same signature **0.5pt** (function overloading should be permitted)
- 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) **0.5**
- the parameters of a function call have the types from the function definition **0.5pt**

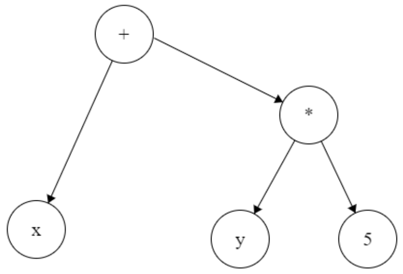
Detailed error messages should be provided if these conditions do not hold (e.g. which variable is not defined or it is defined twice and the programm line);

4) build abstract syntax trees (AST) for the integer expressions in a program;
if a program is syntactically and semantically correct and the *expr* type is int , for every call of the form *Print(str,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 = expr* (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
(**2.5p**t)

AST: abstract syntax tree - built during parsing
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
  - 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 an element of a vector (ID[NR])
    expr.AST = buildAST(element, null, null, ARRAY_ELEM)
  - if expr is other operand (function calls or other syntactic structures from your language)
    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 labelled with:
    - a number: return the number
    - an id: return the value of the identifier
    - vector element: return the corresponding value of the vector element
    - anything else: return 0
  - else (*ast* is a tree with the root labelled with an operator):
    - evalAST for left and right tree
    - combine the results according to the operation

Besides the homework presentation, students should be able to answer specific questions regarding grammars and parsing algorithms (related to the first part of your homework) or yacc details related to the second part (the answers will also be graded).