Design an original programming language using YACC/BISON. **The code must be written in C++**

1) (3.5pt) Syntax requirements

Your language should should include

- type declarations **(0.75pt**):
  - predefined types (int , float, char, string, bool),
  - array types
  - classes:
    - provide specific syntax to allow initialization and use of objects
    - provide specific syntax for accessing fields and methods
    - allow classes to be defined only in the global scope
- variable declarations/definition, function definitions **0.25pt;**
- control statements (if, for, while), 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**
  The values of boolean expressions are *true* and *false*.
  <u>if and while statements can have as conditions only boolean expressions</u>
- function calls which can have as parameters: expressions, other function calls, identifiers, etc. **0.75pt**

➔ Your language should include two predefined functions *Print(expr), Type(expr)* (*expr* can be an arithmetic or boolean expression)

➔ Your programs should be structured in 4 sections as follows: 1) a section for classes; 2) a section for global variables; 3) a section for function definitions; 4) a special function representing the entry point of the program

2) (1.5pt) Symbol tables

Create a symbol tables for every scope in the program.

A scope is the portion of the program where an identifer is visible and can be used. In your language, you will consider the following scopes:

- global scope: each program has a *global scope*, which contains the entire program: an identifier defined outside all functions/classes/blocks are in the global scope and are visible anywhere
- block scope: each *if*, *for*, *while* instruction introduces a block scope: every identifier defined/declared inside a if/ for/ while block has *block scope* and is visible only inside the block.
- function scope: each function definition introduces a function scope (the block of code corresponding to the function definition).
- class scope: each class definition introduces a class scope.

Use a class SymTable in order to implement a symbol table for a scope. Since the scopes can be nested, every class SymTable corresponding to a scope should keep a pointer to the SymTable of the surrounding scope.

```
//some program in some language
int x; // x is in the global scope

class A: //A is in the global scope
     ////y and f are in the class scope of A, they belong to the SymTable of A
     int y;
     function f():int {
        x = 5;
     }
```

```
endclass
//f is in the global scope
function f (): int { //the function introduces a symbol table with variables x and y
        int x; int y; //
        if(x > 2) { //this block should have its own SymTable, with a pointer to the SymTable of
function f
            int x; //x is in the block scoped introduced by if
            y  = 5;
        }
}
```

A SymTable class should should have a name field (“global”, “block”, the name of the function, the name of the class) and should also contain:

a) information regarding variables (type, name, value)

b) information regarding function identifiers (name, the returned type, the type of each formal parameter, the class in which the identifier is defined)

c) information regarding classes (name)

After parsing the program, print in a separate file the symbol tables for the global scopes, the classes scopes, and the function scopes;

3) (2pt) Semantic analysis

Verify that:

a) any variable or function used in a program has been previously defined **0.25**

b) a variable should not be declared more than once in the same scope; **0.25**

c) all the operands in the right side of an expression must have the same type (the language should not support casting) **0.5**

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) **0.5**

e) 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 program line);

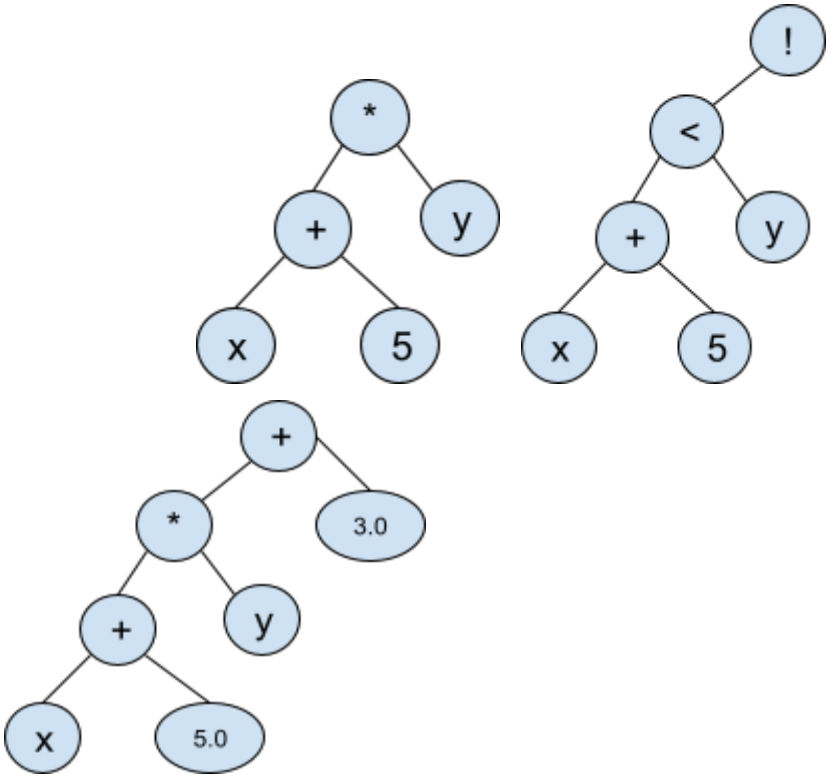
4) (3pt) Evaluation of arithmetic expressions and boolean expressions

The evaluation of arithmetic expressions and boolean expressions using **Abstract Syntax Trees (AST)**:

Abstract Syntax Trees for expressions are intermediary representations built during the parsing of an expression. The AST will be evaluated to produce the value of the expression. The AST should store, as well, the type of the expression.

An AST for an expression is a tree such that:

- if the expression is a number (float, int), a boolean value, an identifier, an element of a vector, a function call, a class member etc (anything that can appear as an operand in an expression), the tree has only one node.
- if the expression has the form *expr1 op expr2* , the AST has the root labeled with the operator *op*, the left subtree is the tree corresponding to *expr1* and the right subtree is the tree for *expr2*.
- if the the expression has the form *op expr1* (*op* is a unary operand) the root is labeled with *op* and has only a left subtree corresponding to *expr1*



For ASTs:

- Write **a class representing an AST** (ASTNode)
- write a member function that evluates the AST.
    - if the root has no children and is labeled with:
        - a float, an int, a boolean value: return the value
        - an id: return the value of the identifier (from the corresponding SymbolTable)
        - anything else: return 0
    - else (*ast* is a tree with the root labeled with an operator):
        - evaluate the subtrees
        - combine the results according to the operation in the root

For every call of the form *Print(expr)* in a program in your language, the AST for the expression will be evaluated and the actual value of *expr* along with the type of expr will be printed. For every call of Type(expr), the type of the expression should be printed

```
int x = 4; bool v ;
v = true && x < 3;
Print(x+3) should print "7". TypeOf(x+3)  should print "int"
Print (v) should print true. Typeof(v|| x == 4) should print "bool"
Print(x+v); //error  (see requirement (3)c)
```

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 in order to compute the *left_value*

Remarks:

- **DO NOT use global variables of type AST**
- Use as few global variables as possible and avoid defining functions in the .y file
- **Define classes for ASTs and symbol tables**
- Besides the homework presentation, students should be able to:

- answer specific questions regarding grammars and parsing algorithms or yacc/bison details related to the second part (the answers will also be graded).
- modify the grammar/the code during the presentation

Deadline:
Part (1) (Syntax) : week 12 (the week before the winter holliday)
The entire project: week 14