Develop the following modules of the compiler for implementing ERPLAG language.

**Symbol Table (ST):** This module takes as input the parse tree generated during the previous stage. Symbol table is a special data structure that maintains the information about the identifiers (variables that participate in computation as the source code is executed). The information gathered during semantic analysis phase is extremely valuable for generating the assembly language code for the input source code. Variables declared in different static scopes can be maintained in separate symbol tables. Since the scope of the variables is known only when the syntactic structure is established, i.e. after parsing, the symbol table links can be established to the function definitions after syntax analysis.

Implement Symbol Table to incorporate following information for all identifiers.

* Type
* Scope
* Offset etc.

Identifiers corresponding to the function names must be maintained separately. The information such as whether the token ID corresponds to a variable or it corresponds to the function name is obtained by the syntactic structure of the sub‐tree that contains this identifier.

**Abstract Syntax Tree (AST):**This module takes as input the parse tree generated in stage 1. The abstract syntax tree is generated by eliminating unnecessary details such as semicolon, colon, comma, parenthesis, square brackets, range operator, assignment operator etc.

Any node in the abstract Syntax Tree retains the information about the non terminal symbol that would have derived the corresponding subtree (which is essential to keep the syntactic structure intact with you throughout, during front end ).

The AST retains only those children that are essential later for semantic analysis, while those appearing as a linear chain, are collapsed. Any child node corresponding to the operator (+, ‐, <= , AND etc) can be collapsed and the information regarding the terminal token such as PLUS, MINUS, LE, AND etc. is fetched up to the parent node.

Implement an AST with the following general rule

 (parent)(children list)

where parent is the name of the construct representing the sub-tree and can be either any new name or the same non‐terminal symbol that exists in the parse tree. The children list contains the nodes which are meaningful. The leaf nodes of the AST still continue to contain the tokens and other relevant information extracted during stage 1. The AST later helps in traversing the tree faster than the parse tree traversing the meaningful nodes only. You must

* Prepare the semantic rules to derive the abstract syntax tree structure
* Modify the structure of the parse tree node (at least a link to the ST can be added) to use for AST.
* Ensure a single pass of the parse tree.
* Produce as output the Abstract Syntax Tree, if the source code is syntactically correct.

**Type Extractor and Checker:** Type of an identifier is extracted from the declaration statement that declares the identifier. The data types supported in the language you are implementing are: integer, real, boolean and array. The type of an element of an array is the type of the data the array refers to. For instance, consider the statement declare a: array[1..15] of integer; the type of an element a[5], say is integer. The type checker verifies the type of an expression appearing at the right hand side of the assignment statement such as value := (a+b‐c ) and checks if it matches with that of the identifier on the left hand side. An arithmetic operator can have two operands of the similar type, where types can be integer and real data types. Example : let a,b, c and d be declared as integer then

An expression in a statement a := (b\*2‐4\*c)+5\*d; its RHS has a type integer and since the identifier a is also of the same type, the type checker approves the expression assignment

An expression in a statement a:= (b\*2 +c)/d <= c\*10; has a boolean valued expression which is assigned to the real valued identifier, hence the expression value assignment is wrong.

The **Static type checking rules** are:

1. The type of an identifier is the type appearing while declaring the variable.

2. The type of NUM is integer.

3. The type of RNUM is real.

4. The type of true or false value is boolean.

5. The type of an array variable A of type *array [12..20] of real* (say) is defined as an expression **<real, 12, 20>**.  The type of an array element A[13] (say) is **real** if 13 is within the bounds **[12, 20]**

6. The type of a simple expression (say E) of the form expression(say E1) <operator>

Expression(say E2)

* is integer, if both expressions are of type integer and the operator is arithmetic operator.
* is real, if both the expressions are of type real and the operator is arithmetic operator.
* is boolean, if both expressions are of type integer and the operator is a relational operator.
* is boolean, if both expressions are of type real and the operator is relational.
* is boolean, if both expressions are of type boolean and the operator is logical.

The type of the expression is ERROR, if the above rules do not derive the type of E appropriately.

Type checking rules for array construct are as follows:

* The operations +, -, \*, / and all relational and logic operators cannot be applied on **array variables** of type <type, 12, 20> (say)
* The assignment operator applied to two **array variables** of the same type is valid. For example, if A and B are the array variables of type array[12..20] of real, then A:= B; is a valid statement. This applies to dynamic arrays of type array[a..b] of real as well.
* Consider **array elements** with **index** represented by integer identifier say A[k]. Here **type checking** of variable *k* is done at compile time. If the type of k is integer, then it is valid else it is reported as an error. Also, the type checking of A[13], for type of index (NUM), is done at compile time.
* The **bound checking** of A[13] where A is a static array is done at compile time. If A is a dynamic array, then bound checking of A[13] is done at run time [see below]

The type of an identifier or an expression is computed by traversing the abstract syntax tree.

Following **type checks are dynamic** and your code generator takes care of dynamic type checking module.

1. ***Address computation and type checking*** of variables of **dynamic arrays** such as in declare A: array[a..b] of integer; The offset computation is dependent on values of a and b and is done at run time.

2. ***Bound checking* of elements** A[10] and A[n] for **dynamic arrays**, and of A[n] of **static arrays** is done at run time.

**Semantic Analyzer:** This module verifies the semantics of the code. Following are the rules that ERPLAG supports.

* An identifier cannot be declared multiple times in the same scope.
* An identifier must be declared before its use.
* The types and the number of parameters returned by a function must be the same as that of the parameters used in invoking the function.
* The parameters being returned by a function must be assigned a value. If a parameter does not get a value assigned within the function definition, it should be reported as an error.
* The function that does not return any value, must be invoked appropriately.
* Function input parameters passed while invoking it should be of the same type as those used in the function definition.
* A switch statement with an integer typed identifier associated with it, can have case statement with case keyword followed by an integer only and the case statements must be followed by a default statement.
* A switch statement with an identifier of type real is not valid and an error should be reported.
* A switch statement with a boolean type identifier can have the case statements with labels true and false only. The switch statement then should not have a default statement.
* Function overloading is not allowed.
* A function declaration for a function being used (say F1) by another (say F2) must precede the definition of the function using it(i.e. F2), only if the function definition of F1 does not precede the definition of F2.
* If the function definition of F1 precedes function definition of F2(the one which uses F1), then the function declaration of F1 is redundant and is not valid.
* A for statement must not redefine the variable that participates in the iterating over the range.
* The function cannot be invoked recursively.
* An identifier used beyond its scope must be viewed as undefined
* etc. (More semantics will be made available in the test cases)

**Code Generator :** This module takes as input the abstract syntax tree (AST) as intermediate representation. The function generates 8086 **assembly code**. Only trivial optimization such as avoiding redundant code, appropriate register usage etc is needed while the detailed code optimization techniques are not expected to be implemented. The code generator generates the code for **dynamic type checking** as well.

1. Read the language specifications document carefully and understand semantic specifications.
2. Formulate rules for creation of abstract syntax tree. Work out the AST creation for expression grammar as discussed in the class.
3. Focus on individual constructs and understand the need of meaningful information for code generation.
4. Many issues needed for your stage 2 implementation will be discussed during regular lectures.
5. Students are advised to attend classes regularly. As we progress with semantic analysis and symbol table creation concepts in the class, students must keep on preparing themselves to do the following groundwork.
6. Read more to understand semantics of individual constructs such as expression, boolean expression, assignment statements, control statements, return statement, parameter list, activation record structure, name binding and scopes, type checking etc. [You can refer book on Concepts of Programming Languages - by Robert W. Sebesta, Tenth edition, Pearson Publication, to refresh concepts of Principles of Programming Languages]
7. Students are advised to read the text book and solve problems regularly. Reading and coming prepared to the class makes us more motivated and interested in the subject.
8. Work out rules for abstract syntax tree generation and prepare the document.
9. If there were issues with your lexer and parser, keep modifying the code. Discuss with me if you need help in identifying the flaws in your code.
10. Regularly discuss your understanding with me and clarify your doubts. [after the class or by fixing over email a mutually convenient time for discussion]

**Function Prototypes and File Names**

Function prototypes are flexible. Students are advised to use names of data structures such as ast, parseTree, symbolTable etc. appropriately. You can select names of implementation files appropriately from file names ast.c, symbolTable.c, typeExtractor.c, semantics.c, codegen.c etc. You can have additional files, if you need as support, but the name of the file must be indicative of the contents within it.

The function prototype declarations should be in file \*.h corresponding to the implementation file name [ For example , if you are naming the interface file for the functions in symbolTable.c, name that interface file as symbolTable.h]. The data definitions should also be split in the files appropriately as symbolTableDef.h. etc.

**Intermediate Code Generation**

Teams can generate assembly language code by first constructing the intermediate code using instructions given in text book or can generate the code directly. [The process of code generation through intermediate code generation is more systematic and produces correct code while direct code generation may be erroneous.] However, there is no extra credit for IR (Intermediate Representation) creation as it is left to the decision of teams as to whether to generate code through IR or by skipping IR . The correctness of the generated code will be of significance.

**Instruction Set**

Your compiler must generate equivalent assembly code (in file code.asm) with instructions taken from instruction set of the NASM simulator (linux based). Download NASM (Version 2.14.02) from <http://www.nasm.us/> to verify correctness of the output obtained by executing assembly code generated by your compiler for the user source code in given toy language.

**Efficiency**

Efficiency (Time and Space ) is an expected feature of your compiler code. Design efficient data structure for symbol table etc. Abstract Syntax Tree (AST) is a copy of the user source code in concrete form and semantic analysis is expected to be done by traversing the AST instead of traversing the parse tree. While constructing AST, the unused nodes of the parse tree can be freed.

The semantic analysis, type checking and code generation rules should be based on the constructs (sub trees) of the AST. Compatibility with the GCC version specified during stage 1 must be ensured.