1. Consider the following grammar:

\[
\begin{align*}
P & \rightarrow \text{declare } D \text{ begin } B \text{ end} \\
D & \rightarrow D : \text{int } \text{id} | \text{int } \text{id} \\
B & \rightarrow B ; S | S \\
S & \rightarrow \text{id } = E \\
E & \rightarrow E + T | E - T | T \\
T & \rightarrow ( E ) | \text{id } | \text{num}
\end{align*}
\]

(a) Convert the above grammar to an LL(1) grammar.

(b) Write a translation scheme for the resulting grammar for interpreting the programs written in the above language. The translation scheme you write should evaluate the values of the expressions and store the values of the variables using the procedures provided below.

// These procedures are given. You do not have to write them.
void enterId(String varName) { ... } // creates a storage for the identifier with name varName
void setValue(String varName; int value) { ... } // sets the value of the identifier with name varName to value
int getValue(String varName) { ... } // returns the value of the identifier with name varName

(c) Write a top-down translator based on the translation scheme above using the token definitions and the procedures provided below.

int NUM=..., ID=..., INT=..., LPAREN=..., RPAREN=..., PLUS=..., MINUS=..., EQUALS=..., SEMICOLUMN=..., DECLARE=..., BEGIN=..., END=...;
token lookahead;
class Token { int type; int value; String name; ... }
void match(int tokenType) {
  if (lookahead.type == tokenType) lookahead = getNextToken();
  else error();
}
// These procedures are given. You do not have to write them.
void main() {
  lookahead = getNextToken();
P();
  match(EOF);
  printValues();
}
void printValues() { ... } // prints the values of all declared variables
2. The grammar given below is used for specifying the declarations in a simple programming language.

\[
\begin{align*}
\text{Program} & \rightarrow \text{DeclList} \\
\text{DeclList} & \rightarrow \text{DeclList Decl} \\
& \mid \text{Decl} \\
\text{Decl} & \rightarrow \text{RecordDecl} \\
& \mid \text{BasicDecl} \\
\text{RecordDecl} & \rightarrow \text{struct id begin FieldList end;} \\
\text{FieldList} & \rightarrow \text{FieldList BasicDecl} \\
& \mid \text{BasicDecl} \\
\text{BasicDecl} & \rightarrow \text{int id;} \\
& \mid \text{real id;} \\
\end{align*}
\]

You are asked to convert this grammar to a translation scheme for storage allocation. Use a global variable called \( \text{offset} \) to keep the address of the next available memory location (initialize it to 0). Use the procedure \( \text{enter.loc(id.name, mem.loc)} \) to store \( \text{mem.loc} \) as the address of the memory location for identifier \( \text{id.name} \) in the symbol table (we assume that the symbol table entry is created in a previous pass). The memory location of a record should be same as the memory location of its first field. Assume that the size of \( \text{int} \) is 4 and the size of \( \text{real} \) is 8.

3. Consider the following grammar for binary numbers:

\[
\begin{align*}
\text{N} & \rightarrow \text{L . L} \\
& \mid \text{L} \\
\text{L} & \rightarrow \text{L B} \\
& \mid \text{B} \\
\text{B} & \rightarrow \text{0} \\
& \mid \text{1} \\
\end{align*}
\]

(a) Using only synthesized attributes, write the semantic rules to evaluate the value of the binary numbers generated by this grammar. For example, the value of the input string:

\[110.011\]

should be evaluated as: 6.375

(b) Below we give the LR parse table for the grammar given in part (a)
Using the semantic definitions you derived in part (a), show the evaluation of the synthesized attributes using the stack-based shift-reduce parsing algorithm (LR parsing algorithm) for the input string:

1 0 . 1 0 1

Assume that synthesized attributes of each nonterminal is stored next to it in the parser stack. Your solution should show the contents of the stack, values of the attributes of each nonterminal symbol in the stack, and the production and the semantic rule used in computing each attribute. You should also show the final value computed for the `val` attribute of the start symbol `N` and the production and the semantic rule that computes it.

4. Write the semantic rules for type checking the following expression grammar:

\[
E \rightarrow E \text{ aop } E \\
| \quad E \text{ rop } E \\
| \quad E = E \\
| \quad E \text{ lop } E \\
| \quad \text{id} \\
| \quad \text{bool} \\
| \quad \text{num}
\]

The token `bool` is a boolean literal and the token `num` is an integer literal. Tokens `aop`, `rop`, and `lop` denote arithmetic, relational, and logical operators, respectively. Assume that the token `id` has an attribute called `type` which could be `integer` or `boolean`. Nonterminal `E` also has an attribute called `type` which could be `integer`, `boolean` or `type-error`. You should write the semantic rules to compute the `type` attribute for nonterminal `E`. The type checking rules are:

- Both operands of an arithmetic operator should be of integer type.
- Both operands of a logical operator should be of boolean type.
- Both operands of a relational operator should be of integer type.
- The two operands of the equality operator `=` should be of the same type.

5. Write the type-expressions for `foo` and `bar` in the following C code fragments.

(a)

```c
int foo(int bar[10]; char *x);
```

(b)

```c
typedef struct {
    int a;
    char b;
} data, *pdata;
data foo[100];
pdata bar(int w, data y) { ... }
```
6. Consider the following program (with nested procedures):

```plaintext
procedure main
  float x;
  int a, b;
  procedure p1(int a)
    int b;
    float y;
    procedure p3()
      int c;
      float x;
      begin
        ...
      end;
    begin
      ...
    end;
  procedure p2(float x, float z)
    int a;
    begin
      ...
    end
    begin
      ...
    end;
end;
```

(a) Show the contents of a lexically-scoped symbol table for this program. Also show the memory offset of each variable in the symbol table. Assume the following: Memory offsets for variables are computed in the order they appear in the procedure. \texttt{int} is 4 bytes and \texttt{float} is 8 bytes. Parameters are passed using call-by-value. Parameters of a procedure are stored in the local data area of that procedure.

(b) Draw the activation tree for the following execution sequence:
main calls p1, p1 returns, main calls p2, p2 calls p1, p1 calls p3, p3 calls p3, ...
show the contents of the control stack at this point ..., p3 returns, p3 returns, p1 calls p2, p2 returns, p1 returns, p2 returns.
Draw the contents of the control stack during the second activation of procedure \texttt{p3}. Show the access and control links in the activation records.