1. Answer the following questions:
   a. While the popularities of various languages have increased and decreased during the past forty years, it has rarely (if ever) been the case that any one language totally dominated the programming scene. For example, in the late 1960s, FORTRAN, COBOL, Algol and LISP all enjoyed popularity. What factors make it unlikely that a single language could achieve domination in all programming domains? What arguments can you make for and against the idea of a single language for all programming domains?

   One language has dominated a scene until next better equipped one arrived to suite the evolving machine and changing necessities.

   – for Runtime Efficiency (60’s)
     • Fortran, Assembly
   – for System Development (70’s)
     • Modularity, Regularity
     • Structured Languages
   – for Parallelism Exploitation (70’s)
     • Concurrent Fortran, CSP (Communicating Sequential Process)
   – Natural Language Understanding (80’s)
   – Special-purpose (80’s ~ )
     • for Hardware Design
     • for Expert Systems
     • for Internet
     • for distributed computing

   However no one language will be sufficient to include all the features required by different scenarios. For example C is a language which is near to assembly language and is extensively used to write efficient kernels like UNIX, Linux. FORTRAN being a formula specific language is still used for the scientific calculations in NASA.

   b. What is binding time? What is the advantage of binding things as early as possible? What is the advantage of delaying bindings?

   The resolving of the symbols in the program to addresses in the memory is called binding. The advantage of binding as early as possible is the speed of execution and also it is easy to debug. The delay bindings are useful when the same symbol is expected to behavior in n number of ways depending on the arguments that it receives at runtime. The delay binding could eliminate lot of redundant code.
c. Explain the difference between static and dynamic scope. Why does the use of dynamic scoping imply the need for run-time type checking?

If the type of the symbol is known at compile time then it is said to have static scope. If the type of the symbol could be known at runtime then it is said to have dynamic scope. The compiler could easily warn the user about the illegal type conversion at the compile time for static binding, the same is not possible in the case of the symbols having dynamic scope. It is important to find illegal type casts at runtime also.

d. What is meant by deep access vs. shallow access in dynamic scoping?

Shallow binding refers to when the (non-local) referencing environment of a procedure instance is the referencing environment in force at the time the procedure is invoked (i.e., the one in which the procedure is invoked).

Deep binding refers to when the (non-local) referencing environment of a procedure instance is the referencing environment in force at the time the procedure's declaration is elaborated (i.e., the one in which the procedure was passed as an argument).

e. What is a referencing environment? Describe the difference between deep and shallow binding of referencing environments. What is a closure? What is it used for?

The referencing environment of a statement or expression is the set of active bindings during execution. When a procedure is passed as an argument, the referencing environment and the procedure itself are packaged together and called a closure.

The closure has access to the lexical environment in which it was created, so we need not bother about which function called it with what arguments, and having to work out in that function what you've been called back with.

2. Given the following grammar in BNF:

\[
\text{<assign> -> <id> = <expr>} \\
\text{<id> -> A | B | C} \\
\text{<expr> -> <id> + <expr> | <id> * <expr> | (<expr>) | <id>} 
\]

Using the above grammar show a parse tree and a leftmost derivation for each of the following statements:
a) \[ A = A \times (B + (C \times A)) \]

Derivation:

- Derivation:
  \[
  \begin{align*}
  \text{<assign>} & \Rightarrow \text{<id>} = \text{<expr>}
  \\
  \Rightarrow A & = \text{<expr>}
  \\
  \Rightarrow A & = \text{<id>} \times \text{<expr>}
  \\
  \Rightarrow A & = A \times \text{<expr>}
  \\
  \Rightarrow A & = A \times (\text{<expr>})
  \\
  \Rightarrow A & = A \times (\text{id} + \text{<expr>})
  \\
  \Rightarrow A & = A \times (B + \text{<expr>})
  \\
  \Rightarrow A & = A \times (B + (\text{<id>} \times \text{<expr>}))
  \\
  \Rightarrow A & = A \times (B + (C \times \text{<expr>}))
  \\
  \Rightarrow A & = A \times (B + (C \times <id>))
  \\
  \Rightarrow A & = A \times (B + (C \times A))
  \end{align*}
  \]

Parse tree:

```
<assign>
  <id>
  A
  =
  <expr>
    <id>
    A
    *
    <expr>
      ( <expr>
        <id>
        B
        +
        <expr>
          ( <expr>
            <id>
            C
            *
            <expr>
              <id>
              A
            )
          )
      )
  )
```
b) \( B = C \times (A \times C + B) \)

Derivation:
\[
\begin{align*}
\text{<assign> => <id> = <expr>} \\
\Rightarrow B = <expr> \\
\Rightarrow B = <id> \times <expr> \\
\Rightarrow B = C \times <expr> \\
\Rightarrow B = C \times ( <expr> ) \\
\Rightarrow B = C \times ( <id> \times <expr> ) \\
\Rightarrow B = C \times ( A \times <expr> ) \\
\Rightarrow B = C \times ( A \times <id> + <expr> ) \\
\Rightarrow B = C \times ( A \times C + <expr> ) \\
\Rightarrow B = C \times ( A \times C + <id> ) \\
\Rightarrow B = C \times (A \times C + B )
\end{align*}
\]

Parse tree:
```
<assign>
  <id>
    B
    =
    <expr>
      <id>
        C
        *
        <expr>
          ( <expr>
            <id>
              A
              *
              <expr>
                <id>
                  C
                  +
                  <expr>
                    <id>
                      B
                      )
          )
    )
```
3. Given the following grammar in BNF:
<assign> -> <id> = <expr>
/id> -> A | B | C
<expr> -> <expr> + <term> | <term>
<term> -> <term> * <factor> | <factor>
<factor> -> ( <expr> ) | <id>

a) Rewrite the given grammar to give + precedence over * and force + to be right associative.

<assign> -> <id> = <expr>
/id> -> A | B | C
<expr> -> <expr> * <term> | <term>
\[
\begin{align*}
\text{<term>} & \rightarrow \text{<factor>} + \text{<term>} | \text{<factor>} \\
\text{<factor>} & \rightarrow ( \text{<expr>} ) | \text{id}
\end{align*}
\]

b) Rewrite the given grammar to add ++ and - - unary operators of Java

\[
\begin{align*}
\text{<assign>} & \rightarrow \text{id} = \text{<expr>} \\
\text{id} & \rightarrow \text{A} | \text{B} | \text{C} \\
\text{<expr>} & \rightarrow \text{<expr>} + \text{<term>} | \text{<term>} \\
\text{<term>} & \rightarrow \text{<term>} * \text{<factor>} | \text{<factor>} \\
\text{<factor>} & \rightarrow ( \text{<expr>} ) | \text{id} | \text{id} + + | \text{id} - -
\end{align*}
\]

4. Convert the following EBNF grammar to BNF:

\[
\begin{align*}
\text{S} & \rightarrow \text{A} \{ \text{b A} \} \\
\text{A} & \rightarrow \text{a [ b ] A} \\
\text{S} & \rightarrow \text{A} | \text{A B} \\
\text{B} & \rightarrow \text{b A} | \text{b A B} \\
\text{A} & \rightarrow \text{a A} | \text{a b A}
\end{align*}
\]

5. Write EBNF descriptions for the following

a) A Java class definition header statement

The following is an example class header statement:

public class A extends B implements C, D

where “public” is a modifier and “A”, “B”, “C”, and “D” are identifiers. Assume non-terminal <id> is given.

\[
\begin{align*}
\text{<method_head>} & \rightarrow [\text{public}] [(\text{abstract} | \text{final})] \text{class <id>} [\text{extends <id>}] \\
& \quad [\text{implements <id>} \{, <\text{id}>\}]
\end{align*}
\]

b) A C/C++/Java switch statement

The following is an example switch statement:

switch (a+b)
{
    case 1 : x = 7; break;
    case 2 : x = 8; break;
    default : x = 9;
}
where “a+b” is an expression, “1” and “2” are literals, and “x=7;break;”,
“x=8;break;” and “x=9;” are statement lists. Assume non-terminals <expr>, <literal>, and <stmt_list> are given.

<switch> -> switch ‘(‘ <expr> ‘)’ ‘{‘ {case <literal> : <stmt_list>} [default : <stmt_list>] ‘}’

(3) A C/C++/Java for-loop statement

The following is an example for statement:

```c
for (int k = 0, m = 100;  k < n;  k++, m++)
{
    x = x + 1;
    y = y – 1;
}
```

where “int k = 0, m = 100” is an variable declaration, in which “int” is a type
name, “k” and “m” are identifiers, and “0” and “100” are literals. If there is no
appearance of “int”, “k = 0, m = 100” are a sequence of assignments. Also, “k <
n” is an expression, “k++; m++” are also expressions, and “x=x+1;y=y+1;” is a
statement list.

Assume the following non-terminals are given: <type>, <id>, <literal>, <assign>,
<expr>, and <stmt_list>.

<for> -> for ‘(‘ [[<type>] <id> = <expr> {, [<type>] <id> = <expr>}]; [<expr>];
[<expr> {, <expr>}] ‘)’ ‘{‘ <stmt_list> ‘}’

6. Compute the weakest precondition for each of the following assignment:

a) a = 2 * (b – 1) – 1 {a > 0}
b) b = (c + 10) / 3 {b > 6}
c) a = a + 2 * b – 1 {a > 1}
d) x = 2 * y + x – 1 {x > 11}

a) {2*(b-1)-1 > 0} => {b > 3/2} a = 2 * (b – 1) – 1 {a > 0}
b) {(c+10)/3 > 6} => {c > 8} b = (c + 10) / 3 {b > 6}
c) {a+2*b-1 > 1} => {a + 2*b > 2} a = a + 2 * b – 1 {a > 1}
d) {2*y+x-1 > 11} => {2*y+x >12} x = 2 * y + x – 1 {x > 11}