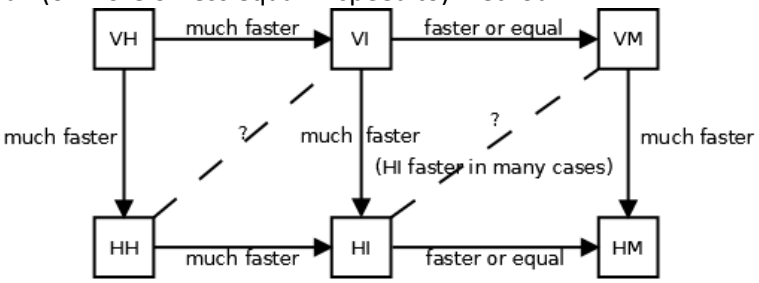
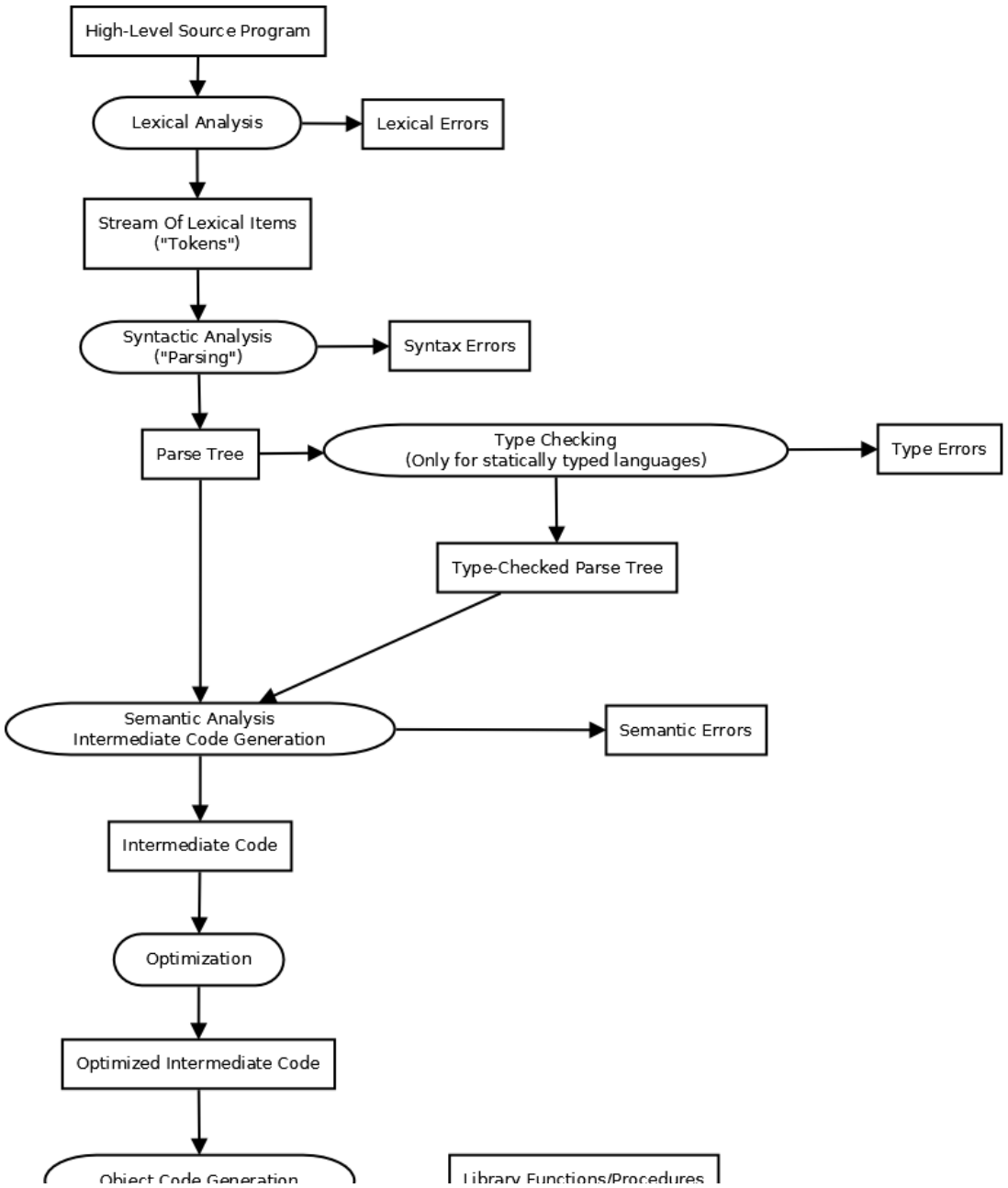
Exercise Set #1----------

1) Compare virtual machines and hardware machines for a programming language *L* and point out one advantage of each.
    a. Virtual Machines: a software program that simulates the execution of any programs in *L*. A virtual machine for *L* is also referred to as a simulator or a software interpreter for *L*.
        i. Advantage: Can be constructed for any programming language whatsoever.
    b. Hardware Machines: a hardware processor that directly executes any programs in *L*.
        i. Advantage: Can be built to execute any high-level languages directly.
2) Suppose that a high-level language *H* is compiled into an intermediate language *I*, which in turn is compiled into a native machine language *M*. Each of these can be executed by a virtual machine or a hardware machine. Then there are 6 possible execution methods for the high-level language *H*:
    a. VH: virtual machine for H
    b. HH: hardware machine for H
    c. VI: virtual machine for I
    d. HI: hardware machine for I
    e. VM: virtual machine for M
    f. HM: hardware machine for M
        i. Which execution method is the slowest of the 6?
            1. VH
        ii. Which execution method is the fastest of the 6?
            1. HM
        iii. Draw a lattice of these 6 methods, with links "X→Y" meaning that method Y is faster than (or more or less equal in speed to) method X.
            1. 

3) Concisely explain the difference(s) between intermediate-code compilers and native-code compilers with regard to how high-level source code is compiled and executed.
    a. Intermediate-Code Compilers:
    b. Native-Code Compilers:
4) Concisely explain how the use of a well-designed intermediate language would facilitate construction of compilers that translate *m* high-level languages into *n* native machine languages.
    a.
5) Give 3 main components of the Java Virtual Machine and succinctly describe the function of each.
    a. Function code area – JVM code for method instructions
    b. Runtime stack – controls function calls
        i. At function call – a stack frame is pushed
        ii. At function return – it's stack frame is popped
    c. Heap area
        i. Dynamically created arrays and class objects are collected here

ii. Garbage collector will collect unused array/class objects here

6) Give a flow diagram specifying the input to and output from each of the following stages of compilers. If error messages are possible output, indicate them appropriately as well.
   a. Lexical Analysis
   b. Syntactic Analysis
   c. Type Checking
   d. Semantic Analysis and Intermediate Code Generation
   e. Optimization
   f. Object Code Generation

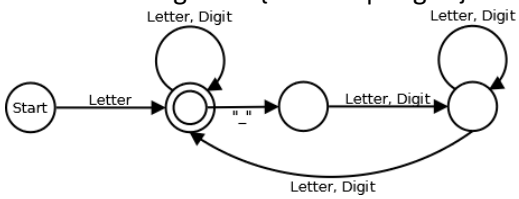


i.

7) Give a complete BNF grammar for each of the following. You may use extended BNF (EBNF).
   a. Identifiers: Any letter followed by 0 or more letters or digits.
   b. Signed Integers: “+” or “-“ followed by one or more digits.

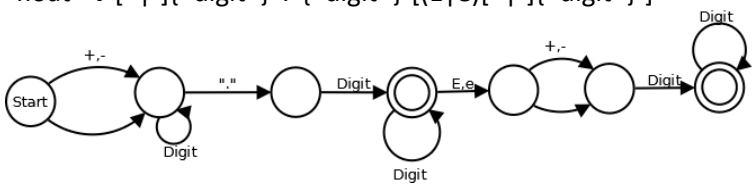
i. <identifier>→<letter>{<letter>|<digit>}
ii. <signed integer>→(+|-){<digit>}$^+$

8) Consider the following EBNF grammar:
   a. <id>→<letter>{<letter>|<digit>}
   b. <extended id>→<id>{"_"<letters and digits>}
   c. <letters and digits>→{<letter>|<digit>}$^+$



   i. Determine if each of the following strings belong to the category <extended id>:
      1. 8ABC No
      2. CS316 Yes
      3. CS316_ No
      4. CS316_ABCX Yes
      5. _CS316 No
      6. CS316_ _ABC No
      7. CS316_987 Yes
      8. CS316_ABC_32A Yes
      9. CS316_543_7B5 Yes
      10. CS316_A_ No
   ii. Rewrite the above grammar to an equivalent one in the original BNF without using any notation in EBNF.
      1. <letter>→a|b|…|z|A|B|…|Z
      2. <digit>→0|1|2|…|9
      3. <id>→<letter><rest of id>
      4. <rest of id>→ε|<letter><rest>|<digit><rest>
      5. <extended id>→<id><rest of extended id>
      6. <rest of extended id>→ ε|"_"<letters and digits><rest of extended id>
      7. <letters and digits>→<letter>|<digit>|<letter><letters and digits>|<digit><letters and digits>

9) Given the following EBNF grammar:
   a. <float>→[+|-]{<digit>}"."{<digit>}$^+$[(E|e)[+|-]{<digit>}$^+$]

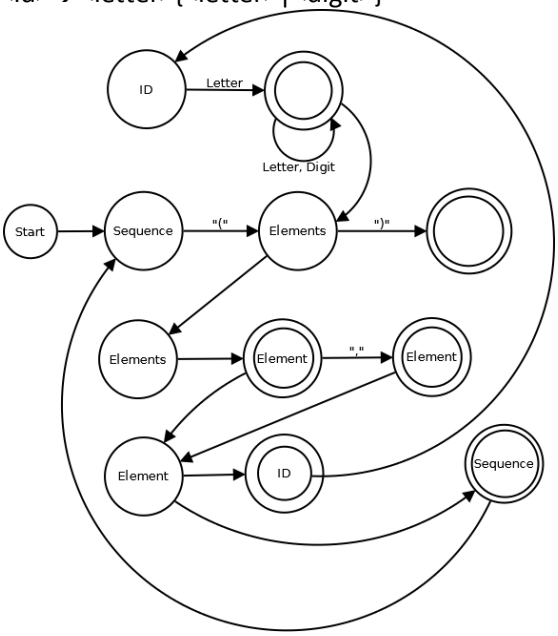


   i. Determine if each of the following strings belong to the category <float>:
      1. .e1 No
      2. .2e No
      3. .2e3 Yes
      4. .45E-32 Yes
      5. 3.2145e10 Yes
      6. 768.43 Yes
      7. 2709 No
      8. 2709. No

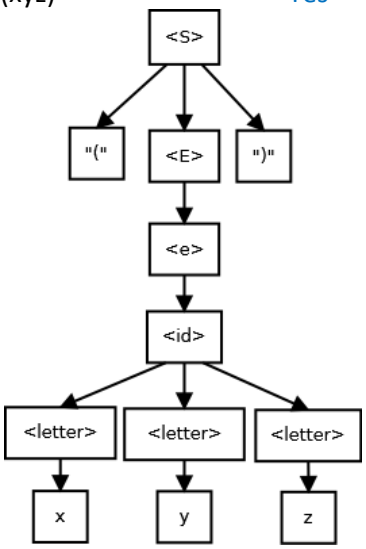
9. -.562e2 Yes
10. +34E+5 No
11. -65.67 Yes
12. 75647.74653e- No
13. 756.65e-7564 Yes
14. +.64-8 No

10) Consider the following EBNF grammar:

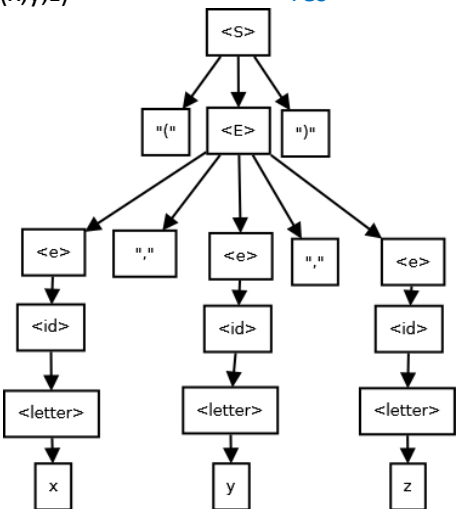
a. <sequence>→"("<elements>")"
b. <elements>→<element>{","<element>}
c. <element>→<id>|<sequence>
d. <id>→<letter>{<letter>|<digit>}



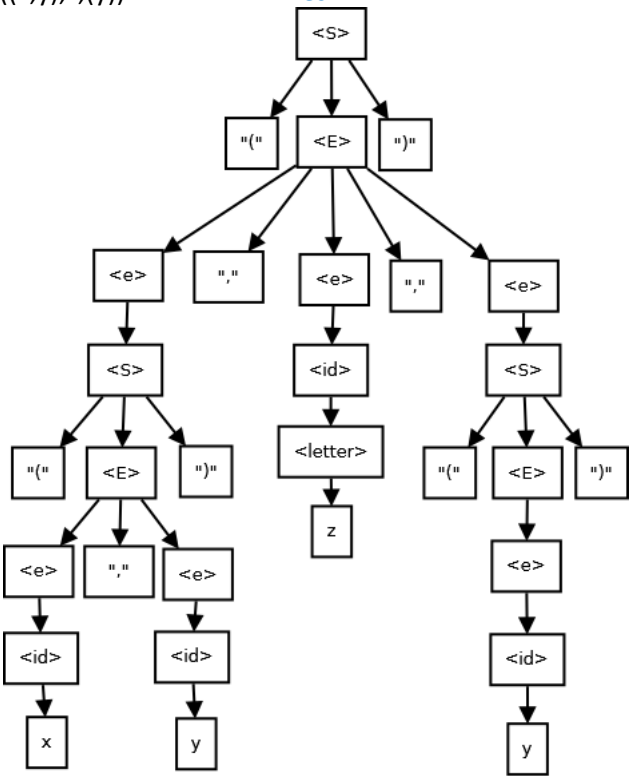
i. Determine if each of the following strings belong to the category <sequence>; if it does, give a parse tree for it:
   1. () No
   2. (xyz) Yes



3. (x,y,z) Yes

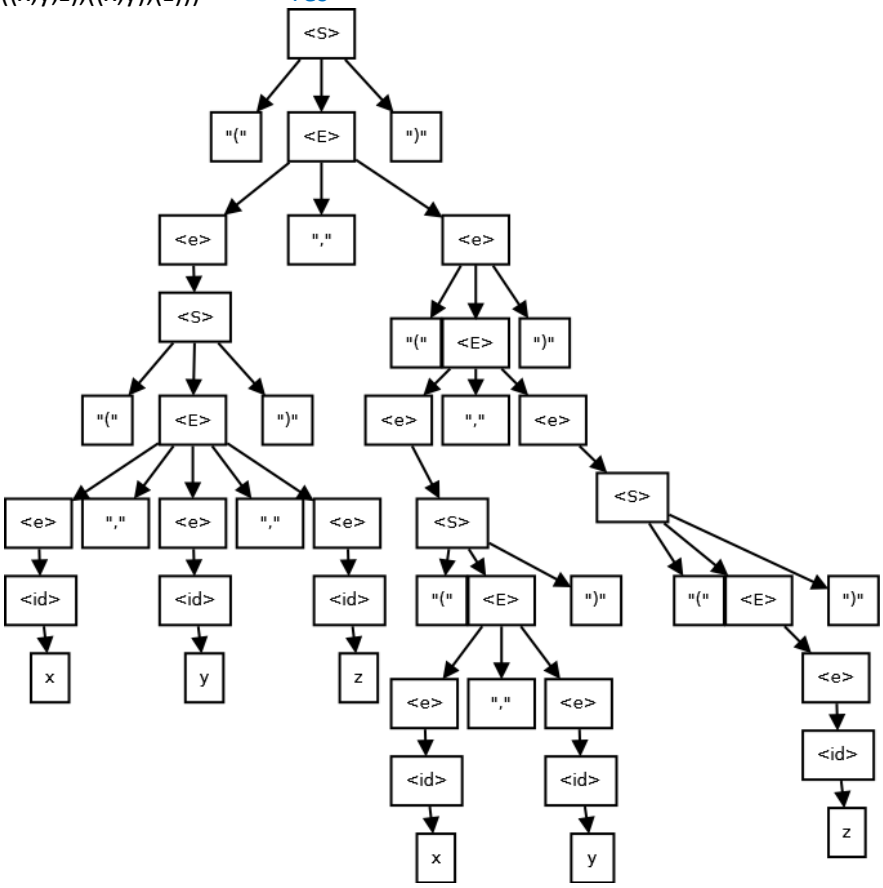


4. ((x,y),z,(y)) Yes



5. (x)(y) No

6. ((x,y,z),((x,y),(z))) Yes



7. x)y)z No

ii. Rewrite the production rule for <elements> to an equivalent one in the original BNF without using any notation in EBNF.

1. <elements>→<element>|<element>“,”<elements>

11) Consider the following ENBF grammar:

a. <X>→”{“<inside>”}”|<id>”;”
b. <inside>→{<X>}$^{+}$
c. <id>→<letter>{<letter>|<digit>}

i. Determine if each of the following strings belong to the category <X>; if it does, give a parse tree for it:

1. {a;b;c} x
2. {ab{ x
3. {a;{b;c;}} x
4. }a;b;{ x
5. {{a;}b;{c;d;}} x

12) Consider the following EBNF grammar:

a. <E>→(+|-)<E><E>|<id>
b. <id>→<letter>{<letter>|<digit>}

i. This is known as expressions in *prefix form*, since the binary operators + and – come before 2 arguments. This grammar is known to be unambiguous (and hence doesn’t need disambiguation parentheses). Determine if each of the following strings belong to the category <E>; if it does, give a parse tree for it.

1. abc x
2. + abc xyz x
3. + + abc xyz ABC x
4. – abc + xyz x
5. – abc + abc xyz x

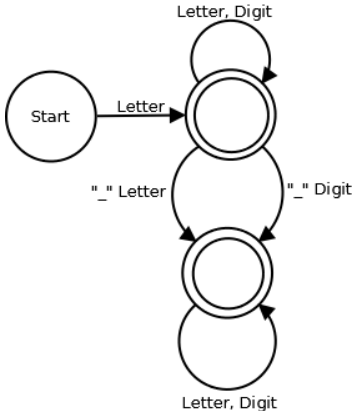
13) Consider the following EBNF grammar:
    a. <E>→<E><E>(+|-)|<id>
    b. <id>→<letter>{<letter>|<digit>}
        i. This is known as expressions in *postfix form*, since the binary operators + and – come after 2 arguments. This grammar isknown to be unambiguous (and hence doesn't need disambiguation parentheses). Determine if each of the following strings belongs to the category <E>; if it does, give a parse tree for it:
            1. abc x
            2. abc xyz + x
            3. abc xyz + ABC + x
            4. abc – xyz + x
            5. abc abc xyz + - x
14) Consider the following BNF grammar for arithmetic expressions, which incorporates the *unary* "-" operator:
    a. <E>→<term>|<term>+<E>
    b. <term>→<primary>|<primary>*<term>
    c. <primary>→<id>|"("<E>")"|-<primary>
    d. <id>→<letter>{<letter>|<digit>}
        i. Give the parse tree for each of the following:
            1. x+y*z
            2. (x+y)*z
            3. x+y+z
            4. x*y*z
            5. x+-y*-z
            6. –(x+y)*z
        ii. According to this grammar, are the + and * operators left-associative or right-associative?
        iii. Expand this grammar to incorporate the binary subtraction and digision operators "-" and "/" with their proper precedence.
15) Consider the following BNF grammar for Boolean expressions:
    a. <BE>→<id>|<BE>"||"<BE>|<BE>"&&"<BE>|"!"<BE>|"("<BE>")"
    b. <id>→<letter>{<letter>|<digit>}
        i. Give all parse trees for:
            1. !x||y
            2. X||y&&x||z
        ii. This grammar is ambiguous – explain why.
        iii. Give an unambiguous BNF grammar which equivalently defines <BE> by incorporating the operator precedence rules:
            1. "!" has higher precedence than "&&",
            2. which has higher precedence than "||".
            3. You may introduce auxiliary syntactic categories and use EBNF.
16) Consider the following BNF grammar for conditionals:
    a. <cond>→if"("<B>")"<S>|if"("<B>")"<S>else<S>

b. <S>→<cond>|...other kinds of statements…
    i. Give all parse trees for:
        1. If(B_1) if(B_2) S_1 else S_2
            a. Assume that $B_1$and $B_2$are some Boolean expressions and S_1 and $S_2$are some statements – derive these directly from <B> and <S>.
        2. This grammar is ambiguous – explain why.

17) Consider the unambiguous BNF grammar for the conditional statements using <matched S> and <unmatched S> discussed in class. Using this grammar, give the parse tree for:
    a. If(B_1) if(B_2) if(B_3) S_1 else S_2 else S_3
    b. If(B_1) S_1else if(B_2) S_2 else S_3
        i. B_1, B_2, and B_3 are Boolean expressions and S1, S2, and S3 are matched statements – derive these directly from <B> and <matched S>
18) Concisely explain what problem will arise in compiler construction if an ambiguous BNF grammar is used to describe programming language syntax.
    a. Ff

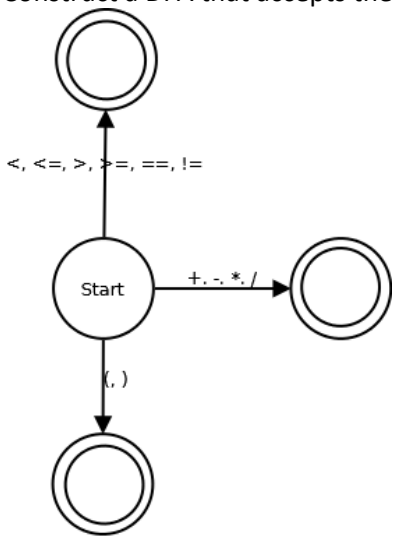
Exercise Set #2----------

1) Consider the following EBNF grammar:
    a. <id>→<letter>{<letter>|<digit>}
    b. <extended id>→<id>{"_"<letters and digits>}
    c. <letters and digits>→{<letter>|<digit>}$^+$
        i. Construct a DFA that accepts the strings in <extended id>



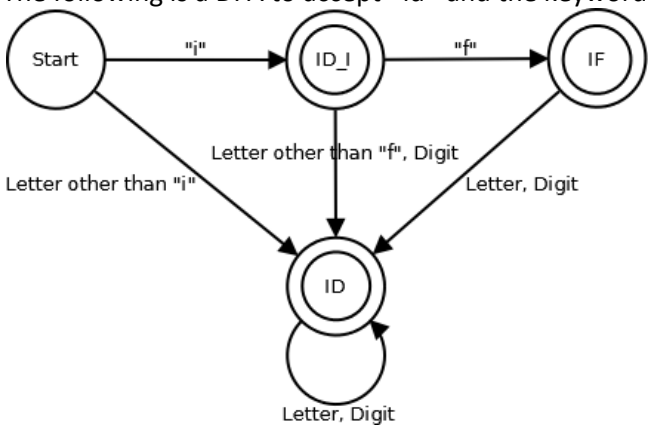
2) Given the following BNF grammar:
    a. <comp op>→"<"|"<="|">"|">="|"=="|"!="
    b. <arith op>→+|-|*|/
    c. <paren>→"("|")"
    d. <token>→<comp op>|<airth op>|<paren>

i. Construct a DFA that accepts the strings in <token>



3) Given the following BNF grammar:
   a. <float>→[+|-]{<digit>}“.”{digit}$^{+}$[(E|e)[+|-]{<digit>}$^{+}$]
      i. Construct a DFA that accepts the strings in <float>. Note that according to this grammar, the integer part may be empty.

4) Building keyword recognition into DFAs is not conceptually difficult but considerably increases the number of states and transitions. The key fact about usual keywords is that an identifier may contain an initial segment or the whole of a keyword; e.g. “whil”, “whil5”, whilea”, etc. Based on this observation, the following method can be used:
   a. Create a linear sequence of transitions for each keyword; any non-empty, proper initial segment of each keyword is accepted by a final state as an identifier.
   b. Whenever the input letter or digit breaks a keyword, the transition branches off to the <id> state.
   c. The final state for each keyword has the transition to the <id> state on the letters and digits.
      i. The following is a DFA to accept <id> and the keyword “if”:



      ii. Modify this DFA to accept additional keywords “else” and “while”

5) Consider the following EBNF grammar for <int> that defines a subset of the Java integer numerals:
   a. <digit>→0|1|…|9

b. <nonZeroDigit>→1|…|9
c. <hexDigit>→<digit>|a|b|…|f|A|B|…|F
d. <octalDigit>→0|1|…|7
e. <int>→<decimal>|<hexadecimal>|<octal>
f. <decimal>→0|<nonZeroDigit>{<digit>}
g. <hexadecimal>→("0x"|"0X"){<hexDigit>}$^{+}$
h. <octal>→0{<octalDigit>}$^{+}$
    i. Construct a DFA that accepts the strings in <int>.

6) Lexical analyzers are an example of pattern matching based on BNF grammars and finite automata, which has many other applications. Quite a number of modern script and programming languages provide built-in features or library functions to recognize patterns specified by EBNF grammars.
    a. Suppose we want to find all file names containing an occurrence of "CS316" and ending with the extension ".txt". The "C" or "S" may be in lowercase. Give an EBNF grammar that defines this pattern.
    b. Spam filters. Suppose we want to catch all email subject headers containing an occurrence of "pharmacy" where any of its characters may be in uppercase and any single ASCII character may be inserted between any of its characters. For example, "pHar%Ma?cy" and "p+hArMac!y". Give an EBNF grammar that defines this pattern.

7) Consider the following grammar:
    a. <sequence>→"("<elements>")"
    b. <elements>→<element>{","<element>}
    c. <element>→<id>|<sequence>
    d. <id>→<letter>{<letter>|<digit>}
        i. Give a top-down parser in pseudo code for <sequence>, <elements>, and <element>. Presume that the function getToken() is given to extract the next token and assign it to the string variable *t*. The tokens are: <id>, "(", ")", and ",".

1.
```
void sequence()
{
    if(t is "(")
    {
        getToken();
        elements();
        if(t is ")") getToken();
        else print("Error: ) expected");
    }
    else print("Error: ( expected");
}
```
2.
```
void elements()
{
    element();
    while(t is ",")
    {
        getToken();
        element();
    }
}
```

3. void element()
   {
      if(t is <id>) getToken();
         else sequence();
   }

8) Consider the following grammar for arithmetic expressions, with the unary – operator in <primary>.
   a. <E>→<term>{(+|-)<term>}
   b. <term>→<primary>{(*|/)<primary>}
   c. <primary>→<id>|"("<E>")"|-<primary>
      i. Presume that the function getToken() is given to extract the next token and assign it to the string variable *t*. The tokens are: <id>, "(", ")", +, 0, *, and /
         1. Give pseudo code for a top-down parser for <E>, <term>, and <primary>; include pseudo code to emit suitable stack-based instructions *push*, *add*, *sub*, *mul*, *div*, and *neg*.
            a. void E()
               {
                  getToken();
                  term();
                  while(t is "+" || t is "-")
                  {
                     getToken();
                     term();
                  }
               }
            b. void term()
               {
                  primary();
                  while(t is "*" || t is "/")
                  {
                     getToken();
                     primary();
                  }
               }
            c. void primary()
               {
                  if(t is <id>) getToken();
                  else if(t is "(")
                  {
                     getToken();
                     E();
                     if (t is ")") getToken();
                     else print("Error:) expected");
                  }
                  else if(t is "-")
                  {
                     getToken();
                     primary();

```
        }
        else print("Error: <id>, (, or – expected");
    }
```

2. Give the sequence of *push*, *add*, *sub*, *mul*, *div*, and *neg* instructions that would be generated from each of the following expressions:
    a. X+Y+Z
    b. X-Y+Z-W
    c. X+Y*Z
    d. (X+Y)/(Z-W)
    e. X*-Y+Z/-W

9) Consider the following grammar:
    a. <A>→(+|-)<B><B>
    b. <B>→<id>|<A>
        i. Give pseudo code for a top-down parser for <A> and <B>. Presume that the function getToken() is given to extract the next token and assign it to the string variable *t*. The tokens are: <id>, +, and -.

```
1. void A()
   {
      if(t is "+" || t is "-")
      {
         getToken();
         B();
         B();
      }
      else print("Error: + or – expected");
   }
2. void B()
   {
      if(t is <id>) getToken();
      else A();
   }
```

10) In class, we studied a top-down parser for the following grammar:
    a. <S>→<assignment>|"{"<S List>"}"
    b. <assignment>→<id>=<E>;
    c. <S List>→{<S>}$^{+}$
        i. Let us extend the production rule for <S> with while loops and conditionals: <S>→<assignment>|"{"<S List>"}"|while"("<B>")"<S>|if"("<B>")"<S>[else <S>] Give pseudo code for a top-down parser for the extended <S>, <assignment>, and <S List>. Presume that you are given the function B() to parse Boolean expressions <B>, E() to parse arithmetic expressions <E>, and getToken() to extract the next token and assign it to the string variable *t*. The tokens are: <id>, if, else, while, "{", "}", "(", ")", "=", ";". The top-down parser, if correctly constructed according to this grammar, should automatically build implicit parse trees in which each else matches the closest preceding unmatched if.