# EEL702: Major Test

January 7, 2015

Maximum Marks: 30

1. One way of showing the equivalence of two regular expressions is to show that their corresponding minimal DFAs are the same, upto renaming of states. Use this approach to prove the equivalence of the regular expressions $(a|b)^*$ and $(a^*|b^*)^*$. You may use the sequence of conversion algorithms discussed in class: RegEx $\rightarrow$ NFA with $\epsilon$-trans. $\rightarrow$ NFA $\rightarrow$ DFA $\rightarrow$ minimal DFA. **[6]**

2. Design context-free grammars for the following languages. Use your grammar to give a leftmost derivation for the given example string in each case.

   a) All binary strings which are palindromes, i.e., they read the same backwards as they do forwards. (Example: 110010011) **[1.5]**

   b) All binary strings with an unequal number of 0s and 1s. (Example: 0100111011) **[2.5]**

3. Consider the following syntax-directed definition for declarations of the type `int`:

| Production | Semantic Rules |
|---|---|
| $D \rightarrow TL$ | $L.inh = T.type$ |
| $T \rightarrow \textbf{int}$ | $T.type =$ integer |
| $L \rightarrow L_1, \textbf{id}$ | $L_1.inh = L.inh$; $addType(\textbf{id}.entry, L.inh)$ |
| $L \rightarrow \textbf{id}$ | $addType(\textbf{id}.entry, L.inh)$ |

   Use this to give an annotated parse tree for the statement `int x,y,z`. **[2]**

4. Give the output of the following piece of code assuming (i) static scoping and (ii) dynamic scoping. You should explain your answers, by drawing some kind of symbol table to show how the scoping information will be maintained in each case. **[3]**

```
int a = 16;
int b = 4;
void div(){
        printf("%d\n",a/b);
}
void rec1(){
        int a = 64;
        div();
}
void rec2(){
        int a = 8;
        int b = 2;
        rec1();
}
void main(){
        div();
        rec1();
        rec2();
}
```

5. Draw an activation tree for function calls involved in the execution of the following code:

```
int Ackermann(int m, int n){
        if(m==0)
                return n+1;
        else if(n==0)
                return Ackermann(m-1,1);
        else
```

```
                return Ackermann(m-1,Ackermann(m,n-1));
}
void main(){
        printf("%d\n", Ackermann(1,2));
}
```

What is the output of the code? **[2]**

6. Consider the following code in an object-oriented language:

```
class Animal{
        String sound(){
        return "!";
        }
}
class Duck extends Animal{
        String sound(){
        return "Quack!";
        }
}
class Rooster extends Animal{
        String sound(){
        return "Cackle!";
        }
}
void main(){
        Animal a = new Animal();
        print(a.sound());
        Duck b = new Duck();
        a = b;
        print(a.sound());
        print(b.sound());
        a = new Rooster();
        print(a.sound());
}
```

a) What is polymorphism? How does this code demonstrate polymorphism? **[1]**

b) What is dynamic dispatch? Assuming the language implements dynamic dispatch, what will be the output of this program? **[2]**

7. Consider the following three-address code:

```
        a=1;
        b=2;
Label1: c=a+b;
        d=c-a;
Label2: d=b+d;
        if(d>6) goto Label3
        d=a+b;
        e=e+1;
        goto Label2
Label3: b=a+b;
        e=c-a;
        if(c<5) goto Label1
        a=b*d;
        b=a-d;
```

a) Identify the basic blocks in this code, and draw the control-flow graph. **[2]**

b) Do a global available expressions analysis. Number the basic blocks ($B_1$, $B_2$, etc.), and clearly show your working in obtaining the $IN$ and $OUT$ sets for each block, along with the set of available expressions at each point in the program. **[3]**

c) Use the available expressions obtained above to eliminate common subexpressions in the code. **[1]**

d) Do a global live variables analysis on the code resulting from part c), following the same instructions as part b). **[3]**

e) Eliminate dead code using the results from part d), to obtain the final optimised code. (You need not do a second round of live variable analysis.) **[1]**