Programming Languages - Lab 3: Type Equivalence and Parsing

Name: YourName

1. [*20pts*] Here are some type and variable declarations in Pascal syntax.

```
type
  range = -5..5;
  table1 = array [range] of char;
  table2 = table1;
var
  x, y : array [-5..5] of char;
  z : table1;
  w : table2;
  i : range;
  j : -5..5;
```

State which **variables** are type equivalent under:

(a) structural equivalence: x,y,z,w and i,j
(b) name equivalence: x,y (possibly if you allow that they are defined for the same anonymous type)
(c) declaration equivalence: x,y and z,w

2. [*15pts*] Here is another example of type and variable declarations in Pascal syntax.

```
type
  rec1 = record
    x : integer;
    case boolean of
      true : (y : char);
      false : (z : boolean);
    end;
  rec2 = rec1;
  rec3 = record
    x : integer;
    case b : boolean of
      true : (y : char);
      false : (z : boolean);
    end;
var
  a, b : rec1;
  c : rec2;
  d : rec3;
```

State which **variables** are type equivalent under:

(a) structural equivalence: a,b,c (note d is NOT structural equivalent because its discriminated union has a variable in the desciminator)
(b) name equivalence: a,b
(c) declaration equivalence: a,b,c

3. [*10pts*] Can a union in C, Modula-2, or Pascal be used to convert integers to reals and vice versa? Why or why not? (Hint: try this in C to see what happens).

No, it cannot. All data object within a union share the *same* memory space. The size of that space is determined by the size of the largest object defined in the union. Reals and integers have very different storage representations. Interpretation of a block of memory as a real number means knowing that certain groups of bits represent different parts of the real (e.g., sign, exponent, whole part, fractional part) and similarly for integers (although the internal layout is very different). Access to a union member returns an interpretation of the bits of the shared memory location according to the member's data type regardless of the original meaning of those bits when they were assigned to the storage. So, if an integer is stored in the union and then we attempt to read it as a real, the machine will interpret those bits as a real number. It cannot know that they were stored there initially as an integer, so no conversion is done.

4. [*15pts*] Left factor the following grammar.

```
S → Aa | B | D
A → Ct | Cu
B → Db | Dc  | Cx
C → t | m
D → z | q | w
```

Only two rules change (A) and (B):

```
A  → CA'
A' → t | u
B  → DB'| Cx
B' → b | c
```

5. [*15pts*] Eliminate the left recursion from the following grammar.

```
S → Aa | Bb | c
A → Ab | d
B → ab | BcA
```

Only two rules change (A) and (B):

```
A  → dA'
A' → bA' | ϵ
B  → abB'
B' → cAB' | ϵ
```

6. [*15pts*] Compute the FIRST and FOLLOW sets for the following grammar. The bold items are terminal symbols.

P → D Q
Q → D Q | ε
D → **id** **=** E **;**
E → G F | **if** E **then** E **else** E | **func** **id** **.** E | **[**E E**]**
F → **op** E | ε
G → **id** | **con** | **(**E**)**

```
FIRST (P) = {id}
FIRST (Q) = {id, ϵ }
FIRST (D) = {id}
FIRST (E) = {id, con, (, if, func, [}
FIRST (F) = {op, ϵ}
FIRST (G) = {id, con, (}
```

```
FOLLOW (P) = {$}
FOLLOW (Q) = {$}
FOLLOW (D) = {id, $}
FOLLOW (E) = {;, then, else, [, ], (, ), id, con, if, func}
FOLLOW (F) = {;, then, else, [, ], (, ), id, con, if, func}
FOLLOW (G) = {op, ;, then, else, [, ], (, ), id, con, if, func}
```

7. [*10pts*] Is the grammar from the previous problem LL(1)? Explain.

Yes, the grammar from the previous problem is LL(1). There are four production rules to consider:
Q -> D Q | ε
E -> G F | if E then E else E | func id . E | [E E]
F -> op E | ε
G -> id | con | (E)

First(DQ) ∩ Follow(Q) is disjoint.
First(GF) ∩ First(if E then E else E) ∩ First(func id . E) ∩ First([E E]) is disjoint.
First(op E) ∩ Follow(F) is disjoint.
First(id) ∩ First(con) ∩ First((E)) is disjoint.