Spring 2018 Programming Languages Qualifying Exam

CODE_____

This is a closed book test.

Correct, clear and precise answers receive full marks

Please start a new page for each question.

There are five (5) questions, each 20 points

1) Consider the enumerated type language construct. Define enumerated type including how it is used. Compare and contrast enumerated types of two different languages (for example C and Java). Define the term "type-safe" as it applies to enumerated type and state/demonstrate if the language implements type-safe enumerated types.

Enumerated type is a data type consisting of a set of named values called elements, members, enumeral, or enumerators of the type.

Enumerated types are used to aid in language readability and reliability. In readability, use of ENUM clarifies the specific element being used (via its name). Similar argument for reliability. ENUM usage can allow the language designer to enforce usage rules which will aid in identifying static type checking that is needed to produce reliable executable code

type-safe enumerated types means that any operations done to an ENUM result in an ENUM that is in the same group.

In C, ENUM is implemented by assigning an integer to each element of an enumerated type. This aids in readability. However, C enforces arithmetic rules to ENUM, which allows the programmer to add, multiply , divide, and subtract ENUMs with other ENUMS, or even integers. There is no type checking that enforces the rules. This means that C is not type-safe. Additionally, ENUMs can have overlapping values, which means that a programer can mix the ENUM elements between different ENUM definitions.

In Java, ENUMs are also implemented as numbers (as most language do). However, Java enforces both static and dynamic semantic checking to ENUM usage. There is no default arithmetic operations allowed on ENUMS. Since there is no arithmetic allowed, ENUM operations maintain type-safe behavior. Additionally, Java allows ENUMS to be static objects to include methods that are part of the object. This allows the ENUM to have some dynamic responses based on dependencies it may have on other static values in other classes.

*2) **L**anguage Implementation*

*Consider the following Context Free Grammar. Perform any necessary transformation to the grammar and then provide a recursive decent parser. You may assume you have token **methods** :*

- ***CurToken()** : returns the current token without consuming*
- ***MatchToken(token) :** consumes designated token, or fails if there is no match*

E → E + E
E → F
F → (E)
F → id
F → num

This GRAMMAR is not LL(1) as it has left recursion on the first production rule. We apply left factoring to these two rules producing the following grammar:

E → F B
B → + E B | epsilon
F → (E)
F → id
F → num

calculate First() of each non-terminal
First(F) = {num, id, (}
First(B)= {+, epsilon}
First(E) = First (F) = {num, id, (}

Calculate the Follow of each non-terminal
Follow(F)= First(B) = {+, epsilon}
Follow(B) ={empty}
Follow (E) = {‘)’} + First(B) = { ‘)’ , + }

Write Functions for each non-terminal

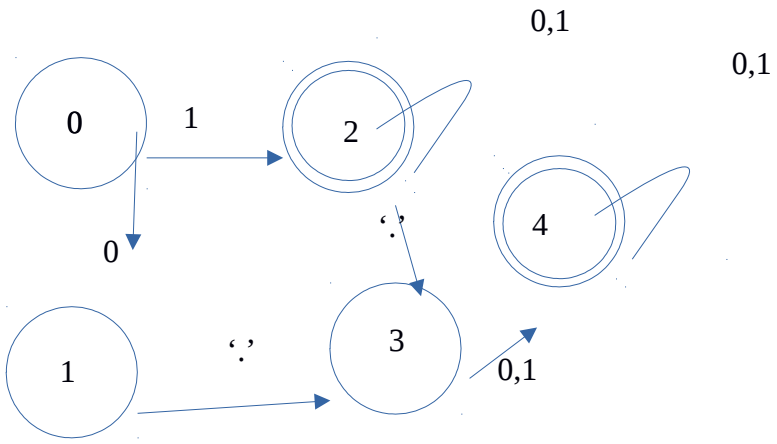
```
void F()
{   if (curToken == ‘num)  { match(‘num); return;}
    if (curToken == id)  { match (id); return;}
   if (curtToken == ‘(‘
       { match ( ‘(‘); E(); match ( ‘)’); return}
   error;
}

void B() {  if( curtoken == ‘ + ) { match(‘+’); E(); B(); return}
               else return;
 }

void E() { F(); B(); }
```

3) Draw a deterministic finite automaton (DFA) that accepts a binary fraction (BF) like "101.01" or "101" or "0.1". A BF can be a binary integer with no fractional part or a true fraction with an integer part (which can be zero), a decimal point and a fractional part (which can be zero). No leading zeros are allowed, i.e., if the BF starts with a zero, it must be immediately followed by a decimal point and a fractional part or followed by nothing. Any number of trailing zeros are OK. Empty integer (".01") or fractional ("10.") parts are not allowed.

- These are legal binary fractions: 0, 1, 101, 0.0, 0.00, 1.0, 1.00, 0.101, 10.01.
- These are illegal: 010, 01.01, .0101 , 101.



4) Write a Java class **Person** with instance variables

String firstName and **String lastName**

- Provide a constructor that takes two string arguments providing values for the first and last names.
- Override the methods **toString** to return a person's full name as a string.
- Override the method **equals** to return true if two Person objects have the same first and last names.

```
public class Person
{
        public String firstName;
        public String lastName;
    public Person(String fn, String ln)
        {
            firstName = fn;
            lastName = ln;
        }
    public String toString()
        {
            return firstName + " " + lastName;
        }
    public boolean equals(Object o)
    // the method should matchfor any object!
    {
        if (o==null) {    // your method should work for null!
            return false;
        }
        Person p = (Person) o;
        return (p.firstName.equals(firstName) && p.lastName.equals(lastName));
        }
}
```

5) Describe lexical and dynamic scoping. Provide details on how each is implemented in a language.

Lexical scoping is also know as static scoping. This is the usual way of dealing with non-local variable references. This sort of variable binding is done at compile time. The main idea is to come up with a process to identify the memory location in the runtime stack for each variable reference. For flat languages like C, the main idea is to use a "level" variable which helps you find the correct version of the variable by doing successive lookups in the symbol table (or maintain a stack for each variable). The routine can determine the offset of the stack pointer from the symbol table directly.

If you are using an Algol like language which allows functions to be defined in functions, you need to maintain an association of the variable with the function distance. In this case,the runtime stack needs to be enhanced to maintain a "static link" to the next correct upstream activation record. When a non-local reference is made during runtime, the system needs to de-reference the static link to reach the enclosing function activation record, and from that point the offset is added to that activation record to get the proper memory location for the variable.

In dynamic scoping, a non-local reference is found by walking the dynamic link up the runtime stack. At each activation record, during run-time, the system needs to lookup the variable name to see if it is defined for that activation method/function. If found, then the offset is used to determine the memory reference. If not, found then look at next link on stack.

Non-local references for dynamic scoping can only be resolved at runtime and the runtime system must maintain a symbol table mapping (during runtime) of every function so that non-local reference can be mode. For static scoping, only a static link has to be maintained for languages that allow embedded function/method definitions.