CptS 580 Advanced Programming Languages

Final Exam

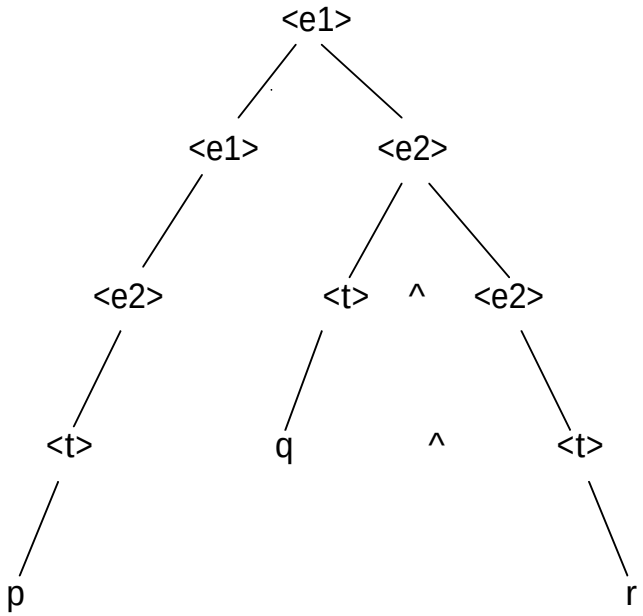
Time Spent-5 Hours

**1. Grammars**

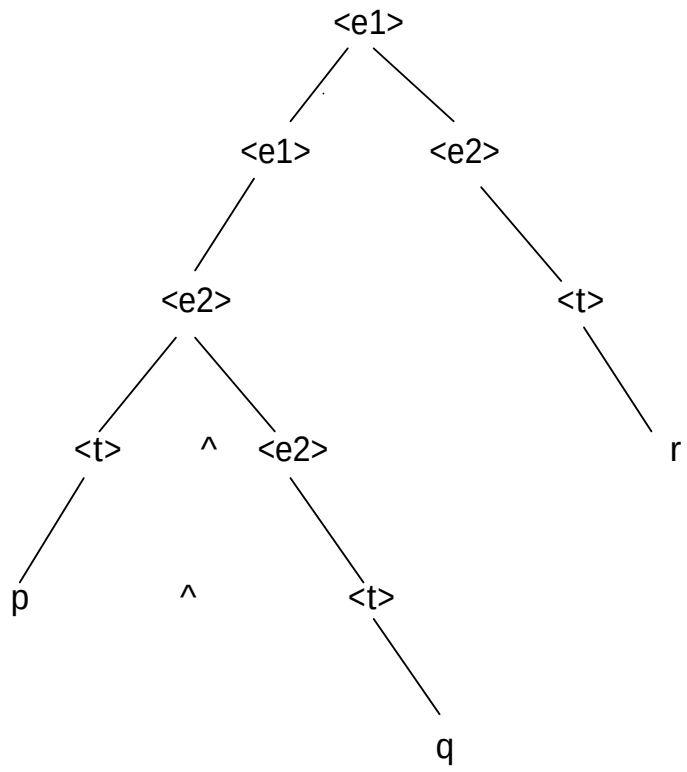
Solution:

<e1> -> <e2> | <e1> <e2>
<e2> -> <t> | <t> ^ <e2>

Example 1: pq^r



Example 2: p^qr



**2. Function closures and storage management**

a) First class functions are functions that can be:

-Created anonymously. A Programming language has first class functions if it can create new functions from the existing ones at runtime

-These functions can be stored in data structures and assigned to variables

-These functions can be passed as arguments to other functions or can be returned from other functions

b) Why first-class function values are much more likely to be found in languages with garbage collection than in languages without garbage collection?

-In Programming languages, where first class functions are implemented, the lifetime of various structures cannot be determined statically. Therefore some kind of automatic memory management like garbage collection is required

-Closures are implementation techniques for first class functions. Closures can be classified as downward closures and upward closures. Upward closures are returned from functions and carry values back up the call stack. This results in statically unbounded value lifetimes and therefore requires garbage collection. Downward closures on the other hand are only available to deeper function calls. Therefore they may be stack allocated and may not require garbage collector

-It is observed that in languages which support first class functions, without garbage collector, the non garbage collected nature of the language makes it problematic for functions to be returned as results, thereby limiting the functionalities of first class function

-Example 1

Consider the below code snippet where we have a function that takes in a value and returns a new function which adds that value to the parameter:

```
func myAdder(n:int) = (fn x:int => x + n);
val myTwoAdder = makeAdder(2);
val myThreeAdder = makeAdder(3);
myTwoAdder(1)---- 3
myThreeAdder(1)-- 4
```

Each time this function is called, it returns a closure that consists of a function pointer that always points to the same function and an integer value for n that varies based on what was passed in myAdder. The returned function can then be treated as any normal function and can be called with any integer parameter of your choice, which will return each parameter with the value of n added to it. Now lets assume that main calls myAdder function twice-once with a value of 1 and again with a value of 2. Quite likely, the second call will use the exact storage layout as the first call, which means the first result will be overlaid. The above example could be fixed using malloc. But this would introduce a leak, so calls to free would also have to be added. Just to avoid the burden of releasing memory manually, the first call functions are more common in languages with garbage collection facility

Example 2

```
 func derive(f , dx) {

   return func(x) {

     return ( f(x+dx)  - f(x)) / dx;

  };

}
```

In the above example, the closure outlives the scope of the function that creates it. The variables x and fx live on even after the function-derive returns. In languages with closures, variables must continue to exist as long as any existing closures references them. This can be implemented using garbage collection. In languages without closures, the lifetime of a local variable coincides with the execution of the scope which has the variable declaration.

-Using first class functions necessitates the creation of values whose lifetime escape the scope of the variable that defined them. In languages without garbage collection, the programmers will be forced to place the values whose storage escapes the scope of their variables in a separately managed part of memory called the heap. Memory allocated on the heap is managed explicitly

by the programmers. The programmer must explicitly place values whose lifetime escapes the scope of their variables on the heap, and explicitly reclaim memory from the heap when it is no longer in use. This becomes very tedious. This results in more usage of memory and also dangling references needs to be handled. But with a garbage collector, the memory on the heap is managed automatically. Garbage collector automatically reclaims memory in a way that there are no dangling references. This explains why **first-class function values are much more likely to be found in languages with garbage collection than in languages without garbage collection**

c) Implications of first-class function values lifetimes for management of the stack, both manually and when garbage collection is used

-In programming languages, where functions are first class entities which can be stored and passed around like any other kind of data, stacks can be problematic because a function can out-live the stack frame they were created in

-As explained in the previous section, without garbage collection, the programmer must explicitly place values whose lifetime escapes the scope of their variables on the heap, and explicitly reclaim memory from the heap when it is no longer in use and also handle dangling references, which is tedious. With a garbage collection, memory is reclaimed automatically when it is no longer in use and dangling references are also handled

Reference- ftp://ftp.cs.princeton.edu/reports/2001/640.pdf

3. Argument evaluation order

To implement a program that demonstrates the order of argument evaluation for a function of three parameters.

1. Imperative language used to prove the above- Java

2. The function call evaluates arguments from left to right. In Java, the order of evaluation of arguments for a function with parameters is defined precisely and is always from left to right. Suppose there is a function call in a program written in Java like f(exp1,expr2,exp3), then exp1 will be evaluated first, followed by exp2 and exp3. Suppose the evaluation of one argument expression completes abruptly, then no part of argument expression to its right will be evaluated

3. Below is the sample code that I wrote to make sure that the order of evaluation is from left to right:

```
package iterator_package;

public class Test_EvaluationOrder{
public static void f1(int l, int m,int n)
{
  System.out.println("Output of Function f1:");
  System.out.print("l="+l+" ");
  System.out.printf("m="+m+" ");
  System.out.println("n="+n+" ");

}
public static void f2(int a, int b,int c)
{
  System.out.println("Output of Function f2:");
  System.out.printf("a="+a+" ");
  System.out.printf("b="+b+" ");
  System.out.println("c="+c+" ");
}
public static void f3(int x, int y,int z)
{
  System.out.println("Output of Function f3:");
  System.out.printf("x="+x+" ");
  System.out.printf("y="+y+" ");
  System.out.println("z="+z+" ");
}

 public static void main (String[] args)
 {
       int i=5;
       int j=3;
       int k=2;
       f1(k,k=(k*k),--k);
       f2(i++,i++,i++);
```

```
      f3(j,++i,++j);


 }
}
```

4. Output of the program:

```
Output of Function f1:
l=2 m=4 n=3
Output of Function f2:
a=5 b=6 c=7
Output of Function f3:
x=3 y=9 z=4
```

The program has three functions. The first function f1 has three arguments. Function f1 looks like this:

*f1*(k,k=(k*k),--k)

The function call f1 evaluates the arguments from left to right. So since k=2, the first argument evaluates to 2, in the second argument the value of k is 4 since we have k=(k*k). The call performs a pre decrement before evaluating the third argument. Thus, the third argument evaluates to 3.

Consider the second function f2:

*f2*(i++,i++,i++)

Since i=5, f2 evaluates the first argument to 5.It then performs the post increment on i. So the value of i is 6. Therefore, f2 evaluates the second argument to 6. It again performs a post increment on i. The value of i is now 7. Function f2 evaluates the third argument to 7. Function f2 then performs a post increment. The value of i is now 8.

The third function f3 is as below:

*f3*(j,++i,++j)

Since j is set to 3, function f3 evaluates the first argument to 3. Note-after evaluating the third argument to 7, function f2 performs a post increment on i and the value of i at the end of this operation is 8. Function f3 now performs a pre increment of i before evaluating the second argument. The second

argument is evaluated to 9. Function f3 now performs a pre increment on j and then evaluates the third argument to 4.

The entire concept can be summarized as follows:

For a function with three arguments, the arguments are pushed into the stack from right to left. For a function call of the form f(exp1,exp2,exp3), exp3 will be pushed to the stack first, followed by exp2 and exp1. So exp3 will be at the bottom of the stack,exp2 in the middle and exp1 will be at the top of the stack. Now, exp1 which is at the top of the stack will be popped first and evaluated. Once this is done, exp2 will be popped out and evaluated. Finally, exp3 which is at the bottom of the stack will be pooped out and evaluated.

I did a research on the language's documentation. As per The Java™ Language Specification document(*Java SE 7 Edition),* the function call evaluates the arguments from left to right and the same can be inferred from my code.

Below is the link for the document- The Java™ Language Specification document(*Java SE 7 Edition)*

http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf

I have included a zip file called Test_EvaluationOrder that has the necessary source code

4. True Iterators using Threads

Methods Used:

a) Yield()

thread.yield();

This is a java static method. This method is used to switch the state of currently running thread to runnable state so as to make sure that a thread of the same priority has a chance to run. In other words, it makes the thread re join the ready queue.

b)Sleep()

thread.sleep();

This method forcibly puts the thread to sleep

C) wait()

This method tells the calling thread to free the monitor and go to sleep until some other thread enters the same monitor and calls the notify() method

d)notify()

This method wakes up the first thread that called wait() method on the same object

e)notifyall()

This method wakes up all the threads that called wait() method on the same object. Here the highest priority thread will run first

I have created a IteratorHelper class. This class has yield() and get() methods. The run() method will execute once the thread starts. I then created one more class called the Range class that extends the IteratorHelper class and implements the Thread interface. Range class has the code for the run method. Here I perform simple operation of finding range of numbers, given the upper and lower limits. This is followed by the calls to next(), hasNext() and ih.get() methods and ih.get() returns some value. Now once the value is sent, IteratorHelper class waits until Range class calls get() method again and this

process continues. Methods wait(), sleep(), synchronize(), notify are all implemented. Their function is as described above.

Tricky part:

-Implementation of yield

-From my research and from what I read from the textbook, I guess the concept could have been implemented better using C# or python. The notion of true iterators varies across different programming languages like Clu, Lua, C#, Python etc. I wanted to try it in C#, but couldn't do it because of time constraint.

I have attached a zip file called CptS580_Final which has the necessary code.