Meenakshi Rameshkumar
11198814

Advanced Programming Languages Final Exam

Hours spent: 3.5 (first 3 questions) + 4 (4^{th} question)

1. **Grammars:**

   Taking juxtaposition as f x

   <expression> → <expression> <function>
   | <function>

   <function> → <t> ^ <function>
   |<t>

   <t> → id | String | number |…

Meenakshi Rameshkumar
11198814

2. Function closures and storage management

References: http://en.wikipedia.org/wiki/First-class_function
http://en.wikipedia.org/wiki/Closure_%28computer_science%29
http://en.wikipedia.org/wiki/Garbage_collection_%28computer_science%29
http://www.codethinked.com/c-closures-explained

Any language that supports first class function values, allows the programmer to pass functions as arguments to other functions, return functions from other functions and assign functions as values to variables. This is also the basis of functional programming languages.

The implementation of the function call stack is to push the state (local variable, etc) of the new function call on top of the calling function's state after saving the function pointer of the calling function. When the function returns, the last entry is popped off the call stack and the calling function's state is restored.

First class functions are implemented as closures in programming languages. First class functions can define variables that need to be active beyond the referencing environment they are defined in. Such variables are called free variables. These variables need to be bound to the lexical environment. The scope of the first class function values might have to extend even after the function returns (or the function state is popped), i.e, the calling function can refer to the called function's values. Because the function's variables are popped off the stack when the function returns, e an option to extend the scope/lifetime of the first class function values is to store the state of the first class function values in the heap. Now, either automatic garbage collection can to take care of these stored values or we can manually clean up the values once they are no longer needed.

For example, in Java and C#, ***first class functions*** are implemented using ***anonymous functions/ delegate***, where the variables are allocated in heap and are garbage collected once they are no longer needed.

Hence, it is difficult to implement first class functions without garbage collection.

Meenakshi Rameshkumar
11198814

3. **Argument evaluation orders:**

References:
http://stringoftheseus.com/blog/2011/07/23/argument-evaluation-order/
http://www.eetimes.com/discussion/programming-pointers/4023961/Evaluating-Function-Arguments
Java Documentation: http://docs.oracle.com/javase/specs/jls/se5.0/jls3.pdf : Page 462

1) ***Language Used:*** Java
2) ***Implementation:*** Java SE 5.0 / SE 6
3) ***Source Code***

```
public class Test {

	public static void main(String args[])
	{
		int i=5,j=6,k=7;
		System.out.println("Simple Arguments");
		function(i,j,k);
		System.out.println("Arguments with inline functions");
		function(i++,i--,i);
		System.out.println("Function & Assignment Arguments");
		function(max(i,j,k),j=min(i,j,k),print(j));

	}

	//Print the function parameters
	static void function(int i, int j, int k)
	{
		System.out.println("i="+i+" j="+j+" k="+k);
	}

	//get the maximum of three values

	static int max(int i, int j, int k)
	{
		if(i>=j && i>=k)
			return i;
		else if(j>=k)
			return j;
		else
			return k;
	}

	//get the minimum of three values
```

```
        static int min(int i, int j, int k)   {
              if(i<=j && i<=k)
                    return i;
              else if(j<=k)
                    return j;
              else
                    return k;
        }

        //Print the parameter passed
        static int print(int j)
        {
              return j;
        }
}
```

4) Output:

```
Simple Arguments
i=5 j=6 k=7
Arguments with inline functions
i=5 j=6 k=5
Function & Assignment Arguments
i=7 j=5 k=5
```

The first call to function

***function*(i,j,k);**

passes the integer values of i,j,k with no evaluation required. Thus these 3 values are printed as expected.

The second call to function

***function*(i++,i--,i);**

Since Java evaluates arguments in ***left-to-right manner.*** This function call passes the variable i as parameter i, then performs increment operation, passes the new value of i as parameter j, performs decrement operation on i and then passes the newly decremented value of i as parameter k.

Since we started with i=5, the expected output is

Meenakshi Rameshkumar
11198814

i=5,
j=6 (i++)
k=5 (i--)

The third call to function

***function*(*max*(i,j,k),j=*min*(i,j,k),*print*(j));**

Again, since Java evaluates arguments in ***left-to-right manner.*** This function executes the function ***max(i,j,k)*** passes the output of this execution as parameter i, then performs ***min(i,j,k)*** passes the output of this execution as parameter j, assigns j to the output of ***min(i,j,k)*** and then executes the ***print(j)*** with the new value of j and passes the output of this execution as parameter k.

The Java language specification specifies the argument evaluation order, there is no ambiguity when the Java programs are executed across different platforms. But languages like, C/C++ do not specify the argument evaluation order and hence it is left to the compiler designer to decide on the argument evaluation order. This leads to ambiguity when the C/C++ code is executed across different platforms.

Meenakshi Rameshkumar
11198814

4) True Iterators using Threads

Range class – Iterator class which uses the IteratorHelper class. It implements the runnable class so that we can create threads using the Range object.

Data Members:

1. **low, high** – specify the range of data to be generated by the iterator
2. **ih** – IteratorHelper object to call the methods of IteratorHelper class

Methods:

1. **Range(int low, int high**) – constructor initializes the variables, low and high. Starts a ***new thread (Thread 0)*** using the current Range object
2. **hasNext()** – Returns true if the iterator has not finished parsing through all the values from low to high
3. **next()** – Returns the current value of the iterator
4. **run()** - It is called when the thread created on the Range object is started. It uses a for loop to iterate through all the values between low and high. It calls the yield() method at every iteration.

IteratorHelper class – true iterator class that uses threads to iterate through the data

Data Members:

1. **data** – holds the current value of the iterator
2. **r** – is the Runnable object used the create the thread for the Range object
3. **flag** – is used to synchronize behavior between the get() and yield() methods. It signals when the get()/yield() has to wait
4. **doneflag** – is used to signal the hasNext() method in the Range class that the iterator has exhausted all the values for this object

Methods:

1. **start()** - starts the thread **r** defined on the Range object

2. **yield()** - acts like a producer, wherein it produces a new value that can be consumed by the get() method. It waits till the get() method has used/returned the value in the data variable and then assigns a new value to the data variable. It then notifies the thread waiting in the get() method to consume the value. This method is synchronized so that two threads cannot access the yield() method at the same time.

Meenakshi Rameshkumar
11198814

3. **get() -** acts like a consumer, wherein it uses the new value in the data variable (which was set by the yield() method). It waits till the yield() method has assigned new value to the data variable and then returns this value to the next() method. It then notifies the thread waiting in the yield() method to produce/assign a new value. This method is synchronized so that two threads cannot access the get() method at the same time.

Caller class – driver program that creates a Range object and iterates through the data in the Range object using hasNext() and next() methods

Execution Flow:

In order to simulate a true iterator using these classes, ***a main thread*** creates the Range class object, runs through the values of the Range iterator using the next() and hasNext() methods of the Range class.

The constructor of the Range class creates a ***new thread (Thread 0)*** using the same Range object as the one created in the main thread. The constructor of the Range class also **starts this thread**, which executes the **run()** method. The run() method executes **a for loop which calls the yield() method** at every iteration.

Now **yield()** method uses the **flag variable** to determine whether to wait or assign the value passed to it to the data variable and notify the thread waiting on get() method.

The **get()** method uses the **flag variable** to determine whether to wait or to return the value in the data variable to the next() method and notify the thread waiting on yield() method.

Once the **for loop in the run()** method finishes execution, it means that **yield()** method has produced all the values in the iterator and hence after the for loop, the run() method **sets the doneflag** to signal the hasNext() method that the iterator has run through all the values.

References: **http://www.yoda.arachsys.com/csharp/csharp2/iterators.html**
http://www.java-samples.com/showtutorial.php?tutorialid=306

Meenakshi Rameshkumar
11198814

Source Code

Range class:

```
class Range implements Runnable {

	int low, high;
	IteratorHelper<Integer> ih = new IteratorHelper<Integer>(this);

	public Range (int low, int high) {
		this.low = low;
		this.high = high;
		//Start the thread to yield values
		ih.start();
	}
	public boolean hasNext() {
		//Check if doneflag has not been set in the IteratorHelper
		return !(ih.done());
	}
	public int next() {
		//Get the next value from IteratorHelper
		return ih.get();
	}
	public void run () {
		for (int i = low; i<high; i++) {
			//Yield one value at a time
				ih.yield(i);
		}
		//Set the doneflag to true to signal hasNext() method
		ih.setDone();
	}
    }
```

IteratorHelper class:

```
class IteratorHelper<T> {

	T data;
	Runnable r;
	boolean flag=false;
	public boolean doneflag=false;

	public IteratorHelper (Runnable r) {
		this.r = r;
	}

	public void start() {
		//Start the thread which runs the run() method which calls yield to
produce the next value of the iterator
		new Thread(r).start();
	}

	public synchronized void yield(T v) {
```

```
		//Check if the value has been consumed, if not then wait
		if(flag)
		{
			try
			{
				wait();
			}
			catch(Exception ie)
			{
				ie.getMessage();
			}
		}
		//If the value has been consumed by get(), set the next value
		data = v;
		flag = true;
		//Notify the main thread running the get() method
		System.out.println("Yield "+data);
		notify();
	}

	public synchronized T get() {

		//Check if the producer has produced a value, if not wait
		try
		{
			if(!flag)
			{
				wait();
			}
		}
		catch(InterruptedException ie)
		{
			ie.getMessage();
		}
		flag=false;
		//Notify the thread yielding values to yield the next value
		notify();
		//Return this value to the next()
		return data;
	}

	public void setDone()
	{
		doneflag = true;
	}
	public boolean done(){
		//Yield has completed
		return doneflag;
	}
}
```

Meenakshi Rameshkumar
11198814

Caller class (Driver class):

```
public class Caller {

	//Driver class creates & uses the Range iterator

	public static void main(String args[])
	{
			Range ran = new Range(5, 15);
			while(ran.hasNext())
				System.out.println("Value  "+ran.next());
	}

}
```