Q1: Grammars

Grammar for algebraic expressions involving multiplication and exponentiation:

$$
\begin{aligned}
&1)\ expr \rightarrow term \mid expr\ mult_op\ term \\
&2)\ term \rightarrow factor \mid factor\ exp_op\ term \\
&3)\ factor \rightarrow id \mid number \mid -\ factor \mid (expr) \\
&4)\ mult_op \rightarrow \times \\
&5)\ exp_op \rightarrow \wedge
\end{aligned}
\tag{1}
$$

In Grammar 1: placing 1) before 2) ensured the exponentiation has higher precedence than multiplication. In 1), writing *expr* at the left of *mult_op* while *term* to the right is ensuring the left associativity. Similarly, in 2), put *term* at the right of *exp_op* and *factor* to the left is to ensure the right associativity of exponentiation.

Q2: Function closures and storage management

To see the difference of first class function implementation between manual memory management language and auto memory management, we should start from the definition of *first class function*: function could be passed as arguments to other functions, be returned as the values from other functions, and be assigned to variables or be stored in data structures.

Consider a situation that with a nested function return as a result of another function, according to the call stack design, if the language use manual memory management, problems come out which is known as *funarg* problems. The state of function is stored on the call stack in a data structure called an activation record or stack frame. This stack frame is pushed, or allocated, when a function is called, and popped, or deallocated, when the function returns.In languages without closures, the lifetime of a local variable coincides with the execution of the scope where that variable is declared (Stack allocation). In languages with closures, variables must continue to exist(must has Heap allocation) as long as any existing closures have references to them. This is most commonly implemented using some form of garbage collection. First, when the nested function return, when use manual memory management, they do not know how to deal with the non-local values. Second, problem arises when the calling function refers to the called/exited (nested) function's state after that function has returned. In manual memory management language, if you delete this information at the time of function return, this will cause the long-lived closure become full of dangling references, if you do not delete, you never know when is a right time to deal with these references and cause memory leak. However, in auto memory management language, they will keep the local objects on a heap and their lifetime continue indefinitely, which known as *unlimited extent*. When the reference count of these objects go to zero, the garbage collector will delete the reference, and reclaim the memory.

Thus, first class function is more likely to be found in language with garbage collection than manual storage management. Though with certain restrictions, manual storage management language could have first-class function.

Q3: Argument evaluation order

1) Language of my choice: C,
Compiler: GCC 4.6.3,
OS Type: Ubuntu 12.04 LTS 64bit;

2) I have tested several cases in a single C program, and I found the order of evaluation of function arguments is unspecified in C. See detail in part 4);
Reference: ISO/IEC 9899: `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf` Page 72, §6.5.2.3

3) Source Code:

```
#include <stdio.h>

int f(int exp1, int exp2, int exp3) {
  printf("exp1 = %d, exp2 = %d, exp3 = %d\n", exp1, exp2, exp3);
  return 0;
}

int main() {
  int i = 1, j = 1, k = 1;
  // Evaluation Order: 3->2->1
  printf("Case 1: f(i, i++, i++):\n");
  f(i, i++, i++);
  printf("\n");

  // Evaluation Order 3->1->2
  printf("Case 2: f(j++, j, j++):\n");
  f(j++, j, j++);
  printf("\n");

  // Evaluation Order 2->1->3
  printf("Case 3: f(k++, k++, k):\n");
  f(k++, k++, k);

  return 0;
}
```

4) Output:

```
Case 1: f(i, i++, i++):
exp1 = 3, exp2 = 2, exp3 = 1

Case 2: f(j++, j, j++):
exp1 = 2, exp2 = 3, exp3 = 1

Case 3: f(k++, k++, k):
exp1 = 2, exp2 = 1, exp3 = 3
```

In Case 1: the evaluation order is: $exp3 \rightarrow exp2 \rightarrow exp1$. Since the i has an initial value of 1. Thus the argument which is the first one to evaluate must has a value of 1, here, only $exp3$ is 1 and we know $exp3$ is the first one to evaluate. Now the value of i turns 2, thus the argument which is the second one to evaluate must has a value of 2, here only $exp2$ is 2, thus $exp2$ is the second one to evaluate and $exp1$ is the last one.

Similarly, we could find the evaluation order in Case 2: $exp3 \rightarrow exp1 \rightarrow exp2$. In Case 3: $exp2 \rightarrow exp1 \rightarrow exp3$.

Some thoughts:
1. When we want to identify the order of evaluation of function arguments, you must know the order of evaluation of any pair (exp_i, exp_j) of the arguments list. Thus one argument of any pair (exp_i, exp_j) must has side effect to the other one. Postfix "++" operator is quite suitable for this job. And at most only allow one argument which does not have postfix ++ operator, otherwise, we can not tell the order of the pair (exp_i, exp_j) which both do not have postfix ++ operator.
2. Is $f(i, i++, i++)$ enough for our test?
Yes, if there are no two arguments being evaluated at the same time. As you could examine yourself, six different orders(3!) give six distinct outputs. However, if allow two arguments being evaluated at the same time, for example, we can not distinguish the orders: $exp1 \rightarrow exp2 \rightarrow exp3$ and $(exp1, exp2) \rightarrow exp3$, because both of them produce output $(1, 1, 2)$. In this case, we could use $f(i++, i++, i++)$ instead.

Q4: True iterators using threads

When the Runnable object r is passed to its iterator, we will creat a new thread for its iterator. That is what *t = new Thread(r)* is doing. When the start() method is called, we start the thread by *t.start()*. This is the beginning part of the Iterator. Let's see the detail of how a Range object runs.

When a Range object's run() method is running, it goes into a *for* loop and call a yield method, at this moment, the iterator will yield a value *i*, and tell the thread that it has picked a value, then we must design a lock for this yield to hold this value until the program get the value, otherwise, it will yield the next value. So we set the *lock* value to *true*. When the *for* loop continue to run and yield the next value *i+1* the *while* loop will hold the *lock* and *wait()* for somewhere in the program call the next() method to fetch the value *i*. When the next() method is called, the iterator's *get()* method is used. At this moment, this *get()* must do two jobs, fetch the value *i* and release the *lock* by set *lock = false* and notify the *yield()* method.

At last, when we have yielded all values in the range, we need inform the thread there are no elements left and the *finish()* method is called and set the *NoElement* value to be *true*, and call *t.join()* to terminate the thread *t*.

I think I get the basic idea right, though the code may be problematic since I do not have enough time to learn JAVA from the beginning and debug the code...

Here is the code:

```
class Range implements Runnable {
    int low, high;
    IteratorHelper<int> ih = new IteratorHelper<int>(this);

    public Range (int low, int high) {
      this.low = low;
      this.high = high;
    }

    public boolean hasNext() {
      return ih.done();
    }
```

```
    public int next() {
      if (this.hasNext())
        return ih.get();
    }

    run () {
      for (int i = low; i< high; i++) {
        ih.yield(i);
      }
      ih.finish();
    }
}


class IteratorHelper<T> {
    private Boolean lock = false;
    private Boolean NoElement = false;
    private Thread t;
    private <T> v, v1;

    public IteratorHelper (Runnable r) {
      t = new Thread(r); //creat a new thread for runnable object r
      this.start();
    }

    public void start() {
      t.start(); //start the iterator in the new thread
    }

    public synchronized void yield(<T> v) {
      while (lock) {
        wait();      //wait until the element is retrieved by get()
      }
      this.v = v;
      lock = true;  // lock the yield until the value is fetched by get()
      notify(); //tell the get() the element is available to retrieve
    }
```

```
    public synchronized <T> get() {
      while (!lock) {
        wait();     //wait until the element is sent by yield();
      }
      this.v1 = this.v; //Maybe no needed, just to be safe
      lock = false;   // release the lock for yield()
      notify();//tell the yield the element is sucessfully retrieved
      return this.v1;
    }

    public synchronized Boolean done() {
      if (!NoElement)
        return true;
      else {
        t.join(); // Job done! The end of the thread
        return false;
      }
    }

    public synchronized void finish() {
      NoElement = true; //No more value to yield
    }
}
```