CptS 580 Final Exam

Yujue Wang

May 4, 2012

1 Grammars

In ordinary algebraic notation, multiplication may be denoted by juxtaposition of constants and variables, while exponentiation may be denoted by ^. Assuming that non-terminal $< t >$ will generate (through some unspecified grammar whose details don't matter) any variable or number, give a grammar for algebraic expressions involving multiplication (denoted by juxtaposition) and exponentiation (denoted by ^). Exponentiation has higher precedence than multiplication and associates to the *right*. Multiplication associates to the *left*.

Ans:

1. $expr \rightarrow term \mid term \ \ expr$
2. $term \rightarrow factor \mid term\ expo\ factor$
3. $factor \rightarrow id \mid number \mid (expr)$
4. $expo \rightarrow$ ^

Explanation: Grammar 1 guarantees the exponentiation has higher precedence than multiplication and the multiplication is left associative. Grammar 2 makes sure that the exponentiation operation is right associative. Grammar 3 indicates that () has the highest precedence.

2 Function closures and storage management

Based on what you know about manual and automatic storage management (garbage collection), call stack design, and what it means to be a first-class value, argue convincingly that first-class function values are much more likely to be found in languages with garbage collection (e.g. C#, Haskell) than in languages without garbage collection. Be sure to consider the implications of first-class function values' lifetimes for management of the stack, both manually and when garbage collection is used.

Ans:
Manual storage management refers to that programmer would write instructions to deallocate the unused objects (garbage) such as in C and C++. Garbage collection is a form of automatic memory management. The garbage collector attempts to reclaim garbage, or memory occupied by objects that are no longer in use by the program. First class function is the 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. Call stack, which is normally hidden in automatic memory management mechanisms, is a stack data structure that stores information about the active subroutines of a computer program.

After some research on this question, I think the difficulties of first class function implementation can attribute to the funarg problem. When caller function calls the callee function, the local state of the caller (including parameters and local variables) must be preserved in order for execution to proceed after the callee returns. These contexts will be store in the stack frame. This stack frame is pushed, or allocated, when a function is called, and popped, or deallocated, when the function returns. But some times, the calling function may still refer to the state of the callee function after the callee function has returned (function closures). So without automatic memory management, the state of the callee cannot be deallocated immediately after the caller has returned. It breaks the rules of the call stack design. So it needs the heap-based memory allocation.

The heap-based memory allocation can make it very difficult for human beings to do the memory management. It is no longer as easy as stacks. It is very hard for a programmer to make the right decisions on when to deallocate. It would lead to memory leak or dangling pointer very frequently.

In the automatic management case, some form of techniques such as reference counting and tracing collection will be applied to deallocate the objects when they are no longer needed. It can make it very feasible to maintain the heap-based memory allocation and make the first class function

easier to be implemented.

3 Argument evaluation order

In the imperative language of your choice, implement a program that demonstrates the order in which arguments are evaluated for a function of 3 parameters. 1) Tell me the name of the language, 2) identify the particular implementation that you tested, 3) provide the source code that you wrote, and 4) the output of the program along with notes describing why the output justifies your conclusion about argument evaluation order. In addition, research the language's documentation to see if you can find a specification of what the order should be. Cite any sources, of course.

Example: suppose there is a call in your chosen language that looks like $f(exp1, exp2, exp3)$

In what order are exp1, exp2, and exp3 evaluated? Note: you *must* write a program to demonstrate the order.

Ans: For this question, I tried both C language and Java language. I just used `printf ("%d, %d, %d", i++, i++, i++);` function to do this test. The function will print the value from left to right. So the order of the evaluation will be very obvious as the increment of `i`.

3.1 C language

I wrote a very brief program *test.c* and ran it on both my desktop and my laptop.

The program is very simple, just as:

```
# include < stdio.h >
 int main (){
 int i = 1;
 printf ("%d, %d, %d", i++, i++, i++);
 return 1;
 }
```

Firstly, I ran it on my destktop.

OS: Kubuntu 12.04

Compiler: gcc (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3

The *result* is: `3, 2, 1` which means that the function evaluate the arguments *from right to left*.

But when I try to find some references to support my experiment, I found that actually C programming language has no specified argument evaluation order(ISO/IEC 9899 6.5.2.2 http://www.open-std.org/jtc1/sc22/wg14/www

/docs/n1256.pdf). So I decided to try it on other operating system and compiling environment, so I tried it on my laptop

OS: Mac OS X version 10.7.3

Compiler: i686-apple-darwin11-llvm-gcc-4.2 (GCC) 4.2.1

The *output* is still `3, 2, 1`, which means the evaluation order is still *from right to left (epx3 →exp2 →exp1).*

3.2 Java language

I also tried the Java program:

```
public class test {
  public static void main (String args[]) {
  int i = 0;
  System.out.print(i++ + " "+ i+++ " "+ i++); }
}
```

The output is `1,2,3`. It indicates that the evaluation order is *from left to right(exp1 →exp2 →exp3).* I looked it up from the *The Java Language Specification.* From 15.12.4.2. Evaluate Arguments, we can know that the order is from left to right which coincides with our output.

4 True Iterators using Threads

This is a simple Java Implemention of an IteratorHelper. When an iteratorHelper is first created, it starts a new thread (runs it) which runs the `run()` method of the master class (`Range` in this example).

The `run` method should be implemented like it is done in Python. To generate a value, the `yield` method of the `iteratorHelper` is called.

```
for(Integer i = low; i<high; i++){
  ih.yield(i); //Generate value by calling yield
}
```

After the values are generated, the `setFinished()` method of iteratorHelper must be called. If this is done, then the threads will not reach their completion, as we will see.

There is a queue in `iteratorHelper` to store the generated values. The size of the queue at any point depends upon how the threads are scheduled by the scheduler. A "finished" boolean variable keeps track whether all the values have been generated.

`get():`

If the queue is empty and finished is false, then the thread waits until a value is put into the queue by the yield method or the "finished" is set to false by setFinished() method. When either happens, it returns accordingly.

`done():`

Very similar to get method, except it returns boolean value instead.

`yield():`

This method adds the passed object to the queue and notifies everyone waiting on the queue.

`setFinished():`

This method sets the "finished" variable to true and notifies everyone waiting on the queue.