Banner number: Name:

Final Exam

CSCI 3136: Principles of Programming Languages

April 19, 2013

Group 1		Group 2		Group 3		
Question 1.1		Question 2.1		Question 3.1		$\sum$
Question 1.2		Question 2.2		Question 3.2		
Question 1.3		Question 2.3		▨	▨	
Question 1.4		Question 2.4		▨	▨	
$\sum$		$\sum$		$\sum$		

Information and instructions:

- The questions are divided into three groups. The questions in the first group ask you to explain some basic concepts in programming language design. The questions in the second group ask you to demonstrate your understanding of the basic principles underlying certain programming language features. The questions in the third group are problem solving questions. Make sure you allocate sufficient time to the questions in this latter group.
- Provide your answer in the box after each question. If you absolutely need extra space, use the backs of the pages, but try to avoid it. The size of each box is an indication of the length of the answer I expect.
- **You are not allowed to use a cheat sheet.**
- **Read every question carefully before answering.**
- **Do not forget to write your banner number and name on the top of this page.**
- **This exam has 12 pages, including this title page. Notify me immediately if your copy has fewer than 12 pages.**
- The total number of marks in this exam is 100.

1 Basic Concepts

Question 1.1 (Variables) 10 marks

(a) Define the terms “L-value” and “R-value”.

An L-value refers to a memory location and thus can be used as the left-hand side of an assignment. An R-value refers to a value and thus can only be used on the right-hand side of an assignment.

A variable can be used as an R-value or as an L-value. Explain the difference between the following two uses of variable `x` in the following two assignments. What does “x” refer to in each assignments?

```
x = 1;          y = 2 * x;
```

In the first assignment `x` *is used as an L-value, in the second as an R-value. Thus, the first use of* `x` *refers to its memory location, whereas the second use refers to the value stored in this memory location.*

(b) What does it mean for a language to use a “value model” or “reference model” for its variables? Define the two terms.

A language that uses a value model of variables stores values directly in variables. If a reference model is used, variables store references to the objects they hold.

Does C use a value model or reference model?

C uses a value model.

Does Java use a value model or reference model?

Java uses a value model for primitive types and a reference model for objects.

Question 1.2 (Function parameters) **10 marks**

(a) Explain the following parameter passing modes.

By value

No matter whether the argument could be interpreted as an L-value, its value, not its memory location, is passed to the function.

By reference

This works only for L-values. The memory address of the argument, not its value, is passed to the function.

By sharing

This applies only to languages that use a reference model of variables. The reference to the object is passed by value, but since it is a reference, the caller and the callee both manipulate the same object.

(b) What are "optional function parameters" and "named function parameters"?

Optional function parameters are ones that have a default value provided in the function definition and thus can be omitted in a function call. Named parameters can be specified using their name given in the function definition, as opposed to their position in the argument list.

Is there a run-time cost to using optional or named function parameters as opposed to using "normal" positional parameters? Explain.

No.

For named parameters, the compiler matches each named parameter to its correct position in the argument list at compile time. At runtime, the parameters are passed by position as in a "normal" function call using positional function arguments.

For omitted optional parameters, the compiler provides their default values as arguments to the function call. Thus, once again, this looks like a "normal" function call using all mandatory positional arguments at runtime.

Question 1.3 (Object lifetime and memory management) **10 marks**

(a) We identified three different regions of a program's address space where objects can be stored. Where an object is stored depends mostly on its lifetime. Name the three different regions and for each explain what the lifetime of an object needs to be for the object to be stored in this region.

Objects can be stored in the static address space, on the stack or on the heap. The static address space holds static objects, that is, objects whose lifetime spans the whole execution of the program. The stack is used to store local variables of functions. These variables start their existence when the function is invoked and are destroyed when the function returns. The heap stores dynamic objects, that is, objects that can be allocated and deallocated at arbitrary points during the execution of the program.

(b) Explain under what circumstances a static object may be stored on the heap.

If the object's size is not known at compile time, it needs to be allocated on the heap. An example is an array stored in a static local variable of a function and whose size is determined only when the function is invoked for the first time.

(c) Explain the terms "internal fragmentation" and "external fragmentation".

Heap management algorithms often allocate only blocks of certain sizes to a program. Thus, for a given request, the allocated block is of the smallest permissible size no smaller than the size requested. This may allocate more space than requested. Internal fragmentation refers to this wasted space that is neither used by the program nor available for allocation in response to a subsequent request. External fragmentation refers to the fact that the total size of available free space may far exceed the size of the largest available contiguous block. A consequence is that it may be impossible to satisfy a given allocation request because no block of this size is available even though the total amount of free space would be sufficient to satisfy the request.

Question 1.4 (Names, binding, and scope) 10 marks

(a) What is a "name"?

A name is a mnemonic for another object. For example, a variable gives a name to a memory location. A function names a chunk of code.

(b) What is a "binding"?

A binding is an assocation between a name and the object it names.

(c) Explain the terms "static binding" and "dynamic binding".

With static binding, the object a name refers to can be determined at compile time by identifying the smallest enclosing lexical unit that defines an object with this name. Using dynamic binding, a name refers to the object most recently bound to it during the execution of the program and in a scope the program has not exited yet.

(d) What is a "(stack) frame"?

A frame holds all local variables of a function.

(e) What is a "referencing environment"?

A referencing environment holds all bindings visible at a particular location in the program.

(f) What is a "closure"?

A closure is a pair of pointers, to a function and to the referencing environment that was in effect when the function was defined.

2 How Is It Done?

Question 2.1 (Heap management) — 7 marks

Since objects stored on the heap can be allocated and deallocated at arbitrary times in a program, the run-time environment needs to track the free heap space. When the program makes a request for a heap block of a certain size, it has to be decided which part of the free heap space to allocate to the program. When a program releases an allocated heap block, the heap block needs to be added back to the free heap space. Describe an efficient method for managing the free heap space. "Efficient" means that the time per heap allocation/deallocation should be small and that the method can service allocation requests of arbitrary sizes while keeping internal fragmentation low.

The buddy system divides the memory into blocks whose sizes are powers of 2. Each block of size 2^k is composed of two blocks of size 2^{k-1}. These two "child blocks" are "buddies". We maintain a free-list per block size, $\log m$ lists in total, where m is the memory size.

When an allocation requset of size s is made, we increase k starting from $k_0 = \lceil \log s \rceil$ until we find the first k such that the kth free-list is non-empty. We remove the first block from this free list, split it into its two child blocks, insert one into the $(k-1)$st free-list, and hold on to the other one. We repeat this splitting process until the block we hold is of size 2^{k_0}. This is the block we return to the program.

When an allocated block of size 2^k is released, we add it to the kth free-list. Now, while the current block and its buddy are in the kth free-list, we remove both blocks from the list, add their parent to the $(k+1)$st free-list, increase k by one, and make the parent the current block.

Internal fragmentation is bounded because each request is satisfied using a block less than twice the requested size. The cost per allocation/deallocation is $O(\log m)$ because the work to be done in each free-list is constant.

Question 2.2 (Garbage collection or the next best thing) **13 marks**

(a) What problem do tomb stones and locks and keys address?

Tomb stones and locks and keys are techniques to detect the dereferencing of a dangling reference and raise an error.

(b) Explain how tomb stones work.

A tomb stone is a constant-size record that points to an object. A variable that refers to the object points to the tomb stone. Accessing the object through this variable now requires two dereferencing operations: the first accesses the tomb stone, the second the object referred to by the tomb stone. When assigning a reference from one variable to another, both variables now point to the same tomb stone instead of both referring directly to the object. When freeing an object, we assign a null pointer into its tomb stone. Every subsequent attempt to access the object through the tomb stone now raises a null pointer exception during the second dereferencing operation. This null pointer exception can be caught, in order to provide a meaningful error message. Alternatively, if the exception is left unhandled, the program terminates. In both cases, the incorrect memory access is detected instead of quietly allowing it.

(c) How do reference counts work?

When using reference counts for garbage collection, every object includes a counter that stores the number of pointers to the object that exist. When an object is created, this count is 0*. When assigning a pointer to an object o into a variable p, the reference count of the object previously referenced by p is decreased by* 1 *(p no longer points to it) and o's reference count is increased by* 1 *(p now points to it). When the reference count of an object becomes* 0 *at any time after its initial creation, there are no more pointers to it. Thus, the object is inaccessible and can be deallocated.*

(d) Can reference counts be considered a general garbage collection method? If so, argue that a garbage collector using reference counts always succeeds in deallocating objects that are no longer used. If not, give an example of a type of heap-allocated structure that a garbage collector using reference counts fails to deallocate even though it is no longer used.

No.

The problem is with circular structures.

Consider two objects A and B that point to each other, assume that p is a pointer in the static address space or on the stack that points to A, and assume that these are the only references to A and B. Thus, initially, A's reference count is 2 *and B's is* 1*. Once we make p point to a different object, A's reference count decreases to* 1*, while B's reference count remains* 1*. Since p was the only pointer in the static address space or on the stack that referred to A or B, neither A nor B is accessible any more, yet they have non-zero reference counts, that is, the garbage collector fails to deallocate these objects.*

Question 2.3 (Inheritance and dynamic method binding) — 8 marks

(a) Explain the difference between static and dynamic method binding in object-oriented languages.

Using static method binding when calling an overloaded method, the implementation of the method that is invoked depends on the type of the variable through which the object is accessed. Using dynamic method binding, the implementation is determined by the type of the object, which may be a subclass of the type of the variable used to access the object and thus may provide a different implementation of the method.

(b) Explain how to implement single inheritance and dynamic method binding.

Consider a class A and a derived class B. Every object of type B inherits A's data members and may add its own additional members. In B, A's members are stored before B's members, in the same order as in an object of type A. Objects of both types A and B store a pointer to a v-table that contains pointers to A and B's method implementations. Again, the methods B inherits from A are listed before B's new methods, in the same order as in A's v-table. If B does not override a method f of A, both A and B's v-tables point to the same implementation of f. Otherwise, A's v-table points to A's implementation of f, and B's v-table points to B's implementation. Now, when accessing an object o of type B through a pointer of type A, every method of A can access o as if it was of type A because every data member and method shared between A and B is stored at the same offset from the beginning of the object or v-table, respectively. If B overrides a method f inherited from A, it is still B's implementation of f that is called because o's v-table stores a pointer to B's implementation of f in the v-table slot used by A to identify the implementation of f to be invoked.

Question 2.4 (Iteration and recursion) 7 marks

We discussed that every iterative procedure can be turned into a recursive procedure but that doing the reverse, while possible, is not easy. Consider the following iterative procedure. (I intentionally kept this question language-agnostic and provided pseudo-code. You should do the same in your answers.)

```
**procedure** Sum($A$) **begin**
  $s := 0$
  **for** $i := 1$ **to** $|A|$ **do**
    $s := s + A[i]$
  **end**
  **return** $s$
**end**
```

(a) Translate this procedure into a recursive procedure.

```
procedure Sum($A$) begin
  if $|A| = 0$ then
    return 0
  else
    return $A[1] + \text{Sum}(A[2 \,..\, |A|])$
end
```

(b) Is your procedure tail-recursive?

No. It does work after the recursive call returns.

(c) If not, provide a tail-recursive version.

```
procedure Sum($A$) begin
  return $\text{Sum}'(A, 0)$
end

procedure $\text{Sum}'(A, s)$ begin
  if $|A| = 0$ then
    return $s$
  else
    return $\text{Sum}'(A[2 \,..\, |A|], s + A[1])$
  end
end
```

(d) What is the advantage of tail-recursive procedures over ones that are not tail-recursive in many languages?

Since no work is to be done after a recursive call returns, the sequence of recursive calls can be translated into a loop. Thus, the amount of stack space used is constant and independent of the recursion depth.

3 Problems, Problems, Problems

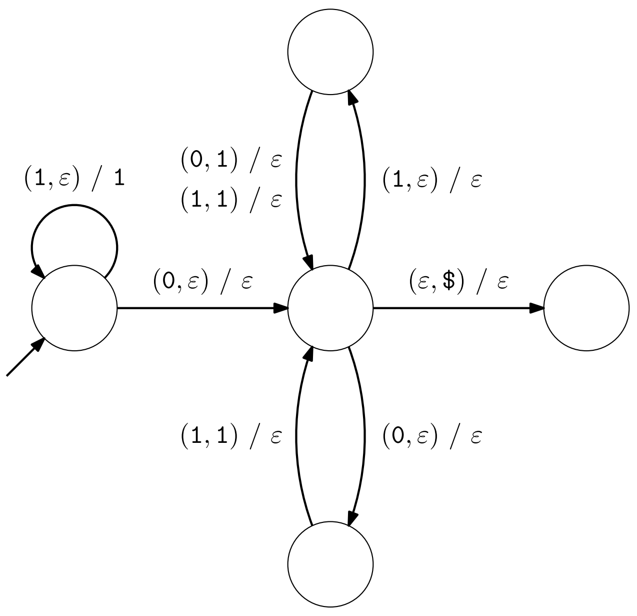
Question 3.1 (Push-down automata) 10 marks

Provide a graphical representation of a push-down automaton that recognizes the language

$$\{1^n 0(01|10|11)^n \mid n \geq 0\}.$$

Do not forget to specify which mode of acceptance your PDA uses and which is the start symbol initially on the stack.

The following PDA recognizes this language by accepting a string using empty stack. The start symbol initially on the stack is \$.



$(1, \varepsilon)$ / 1
$(0, 1)$ / ε
$(1, 1)$ / ε
$(1, \varepsilon)$ / ε
$(0, \varepsilon)$ / ε
$(\varepsilon, \$)$ / ε
$(1, 1)$ / ε
$(0, \varepsilon)$ / ε

Question 3.2 (Parsing) **15 marks**

Provide a context-free grammar for the language in Question 3.1, annotate each production with its PREDICT set, and provide a recursive-descent parser for the language based on these PREDICT sets.

$S \to A\$$	$\{0, 1\}$
$A \to 1AB$	$\{1\}$
$A \to 0$	$\{0\}$
$B \to 01$	$\{0\}$
$B \to 1C$	$\{1\}$
$C \to 0$	$\{0\}$
$C \to 1$	$\{1\}$

```
procedure ParseS begin
   switch next input symbol of
      case 0, 1:  ParseA
                  Match($)
      otherwise: Error
   end
end

procedure ParseA begin
   switch next input symbol of
      case 0:     Match(0)
      case 1:     Match(1)
                  ParseA
                  ParseB
      otherwise: Error
   end
end

procedure ParseB begin
   switch next input symbol of
      case 0:     Match(0)
                  Match(1)
      case 1:     Match(1)
                  ParseC
      otherwise: Error
   end
end

procedure ParseC begin
   switch next input symbol of
      case 0:     Match(0)
      case 1:     Match(1)
      otherwise: Error
   end
end

procedure Match($x$) begin
   if next input symbol $= x$ then
      consume next input symbol
   else
      Error
   end
end
```