Banner number: Name:

Final Exam

CSCI 3136: Principles of Programming Languages

April 14, 2018

Binding		Control flow		Control abstractions		Data abstractions		
Q1.1		Q2.1		Q3.1		Q4.1		$\sum$
Q1.2		Q2.2		Q3.2		Q4.2		
Q1.3		Q2.3		Q3.3		Q4.3		
$\sum$		$\sum$		$\sum$		$\sum$		

Instructions:

- 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. Keep your answers short and to the point.
- You are not allowed to use a cheat sheet.
- Make sure your answers are clear and legible. If I can't decipher an answer or follow your train of thought with reasonable effort, you'll receive 0 marks for your answer.
- 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 11 pages, including this title page. Notify me immediately if your copy has fewer than 11 pages.

1 Binding

Question 1.1: Scopes and referencing environments — 6 marks

Define the following terms:

Scope of a binding

The scope of a binding is the collection of locations in a program where the binding is visible (lexical scoping) or the time interval during the program's execution when the binding is visible (dynamic scoping).

Scope

A scope is a maximal collection of locations in a program where no bindings are destroyed or a largest time interval during the program's execution during which no bindings are destroyed.

Referencing environment

A referencing environment is the collection of bindings visible at a specific location in the program or at a particular time during the program's execution.

Question 1.2: Static scoping vs lexical scoping — 4 marks

Consider the following Scheme function:

```
(define (f)
  (let ((z 1))
    (define (g) z)
    (define (h) (let ((z 2)) (g)))
    (h)))
```

What value does the function invocation `(f)` return?

Using static scoping

1

Using dynamic scoping

2

Question 1.3: Aliasing **5 marks**

Consider the following C function for computing the length of a vector:

```
float veclen(float *x, float *y) {
  *x *= *x;
  *y *= *y;
  *x += *y;
  return sqrt(*x);
}
```

If, before calling this function, x is the value stored in the memory location referenced by x and y is the value stored in the memory location referenced by y, does this function always return $\sqrt{x^2 + y^2}$ as intended? Explain.

No. If x *and* y *are pointers to the same memory location, then the line* `*x *= *x` *squares* x *and the line* `*y *= *y` *squares it again, that is, both* x *and* y *now reference the value* $x^4 = y^4$*. Thus, the function returns* $\sqrt{x^4 + y^4}$ *in this case.*

2 Control Flow

Question 2.1: Control flow primitives **9 marks**

List and briefly describe three control flow primitives. Try to separate the control flow principle from its syntax in a particular language.

1. *Alternation (if-then-else): Based on the truth of a Boolean expression, one of two possible blocks of code is executed.*

2. *Iteration (loops): A block of code is executed a number of times. The number of executions can be specified explicitly or can be determined at runtime via a condition for exiting the loop (while-loop).*

3.

Sequencing: A sequence of statements is executed one after another.

Question 2.2: Switch statements — 5 marks

Programming languages that have switch-statements usually have them because they are more efficient than multi-way if-statements. Describe an implementation of switch-statements that is more efficient than a multi-way if-statement.

A "multi-way" if-statement is nothing but a chain of conditions that need to be checked in sequence until either the first true condition is found—in which case, the corresponding then-block is executed—or the list of tests is exhausted—in which case, either the else-block is executed or the code does nothing if there is no else-block.

The switch-statement is more restrictive in that each condition is a comparison of the same variable against a constant. This allows us to find the condition that is true without checking all of them in sequence. For example, we can store a table indexed by the range of all possible values the variable can assume. The table entry corresponding to each possible value stores the location of the code block to be executed if the variable has this value. This jump table is constructed at compile time. At runtime, the program uses the variable value as an index into the table and jumps to the address found in the table. This takes constant time no matter how many branches the switch-statement has and is in constrast to the linear cost incurred by checking each condition in turn using a multi-way if-statement.

The problem with jump tables is that they may be large. Alternative implementations based on hashing or binary search offer different trade-offs between efficient branching and space-usage of the implementation.

Question 2.3: Applicative-order and normal-order evaluation — 4 marks

Consider the following piece of code:

```
bool print_or_stop(int x) {
  if (x < 10) {
    printf("%d ", x);
    return true;
  }
  else {
    return false;
  }
}

int f() {
  static int x = 0;
  return x++;
}

void main() {
  while (print_or_stop(f()));
}
```

What does this program output?

Using applicative-order evaluation of function arguments

0 1 2 3 4 5 6 7 8 9

Using normal-order evaluation of function arguments

1 3 5 7 9

3 Control Abstractions

Question 3.1: Parameter passing modes — 9 marks

Explain the difference between call by value, call by reference, and call by sharing, both in terms of the mechanics of these parameter passing modes and in terms of the degree to which the called function can alter the contents of the caller's variables.

When a variable is passed as an argument to a function, call by value means that the value of the variable is passed to the function. In contrast, call by reference passes the address of the variable to the function. The consequence is that, using call by value, any change the called function makes to its formal parameter does not affect the content of the variable passed as argument; using call by reference, the formal parameter of the called function is simply an alias for the caller's variable passed as an argument, so any change to the formal parameter also changes the variable passed as an argument.

Call by sharing is a parameter passing mode that is used in languages with a reference model of variables. It technically behaves like call by value: since the value of the variable is a reference to an object, the caller receives a copy of this value, that is, a reference to the same object. As a consequence, since both the caller and the callee have references to the same object, any change the callee makes to the referenced object will also be visible to the caller. However, the callee's formal argument is not an alias for the caller's variable. Thus, the callee cannot change which object the caller's variable refers to, something that is *possible with call by reference.*

Question 3.2: Exception handling — 5 marks

There are numerous ways in which the runtime system of a language supporting exceptions can implement the handling of exceptions. Describe one method that ensures that exception handling incurs no runtime overhead at all as long as no exceptions are thrown while still handling exceptions reasonably efficiently when they are thrown.

The compiler generates a table containing one entry for each program scope. Each entry contains the starting address of this program scope and a pointer to the exception handler associated with this scope. The entries are sorted by the addresses of their program scopes.

As long as no exception is thrown, code just executes normally, without any need to interact with this exception handler table. Thus, the cost is zero.

When an exception is thrown, the program needs to locate the exception handler associated with the block of code where the exception was thrown. This can be done by searching the table of exception handlers to find the block whose starting address is the largest address not greater than the address where the exception was thrown. Since the table is sorted by increasing addresses, this search can be carried out using binary search and is thus reasonably efficient.

Question 3.3: Closures — 3 marks

Consider the following piece of Scheme code:

```
(define count-all 0)

(define (new-counter)
  (let ((counter 0))
    (lambda () (set! counter (+ counter 1))
               (set! count-all (+ count-all 1))
               (list counter count-all))))

(define (run)
  (let ((a (new-counter))
        (b (new-counter)))
    (list (a) (a) (b) (a) (b))))
```

Note: `(list ...)` creates a list containing its arguments.
`(set! var val)` stores the value `val` in the variable `var`.

What is the return value of the function call `(run)`?

```
'((1 1) (2 2) (1 3) (3 4) (2 5))
```

4 Data Abstractions

Question 4.1: Type systems — 5 marks

Explain the difference between a statically typed and a dynamically typed language. Assume that both are strongly typed.

Both languages ensure that operations are applied only to objects of types that support these operations. That's strong typing. The difference is when this check is performed.

Using static typing, either the programmer explicitly states the types of variables, function arguments, etc. or the compiler infers this information. In either case, the type information is determined at compile time and any applications of operations or functions to arguments of the wrong type are caught at compile time, before the code is ever run.

Using dynamic typing, type checks are performed at runtime.

Question 4.2: Dynamic local arrays — 5 marks

Efficient access to local variables at runtime is based on knowing, at compile time (!), the address of each variable relative to the frame pointer. Explain how you can ensure this while also supporting local arrays, allocated on the stack, whose size is determined only at runtime. How do you support testing for out-of-bounds accesses on such arrays?

A language with variable-size local variables usually splits its stack frames into two parts, a static part and a dynamic part. Each variable occupies a fixed slot in the static part of the stack frame. Each variable-size array is allocated in the dynamic part with a pointer to its start stored in the static part. Thus, every variable, including dynamic arrays, still has a fixed offset in the static part. The starting positions of dynamic arrays in the dynamic part differ depending on the sizes of preceding dynamic arrays. The pointer to the starting position of each array stored in the static part ensures that we can still locate the array.

To implement bounds checks, we augment the representation of each array in the static part of the stack frame so it stores not only a pointer to the start of the array but also the bounds of the array in each dimension. This is known as a dope vector.

Question 4.3: Tombstones **5 marks**

Tombstones are used to detect a common programming error in languages with manual memory management (no garbage collection). Which type of error do they detect?

Memory access errors due to dangling references

Explain how tombstones work.

Every pointer to a dynamically allocated object is replaced with a pointer to a tombstone, which in turn points to the object. Dereferencing a pointer now involves dereferencing the pointer itself to access the tombstone followed by dereferencing the tombstone to access the referenced memory block.

When assigning one pointer variable to another, the effect is that both point to the same tombstone now.

When reclaiming a memory block, its corresponding tombstone is labelled as dead (usually represented using a null pointer).

If another pointer still points to this tombstone and we try to use it to access the referenced memory block, the dereferencing of the pointer to access the tombstone succeeds, but the runtime system detects an error as it tries to dereference the dead tombstone.