Programming Paradigms

—Final Exam—

Department of Computer Science
University of Stuttgart

Summer semester 2020 – September 3, 2020

Note: The solutions provided here may not be the only valid solutions.

Part 1 [4 points]

1. Which of the following statements is true? (Only one statement is true.)

   - ☐ A recursive descent parser alternates between "shift" and "reduce" operations.
   - ☐ A recursive descent parser constructs the parse in a bottom-up manner.
   - ☐ A recursive descent parser splits a sequence of characters into tokens.
   - ☐ A recursive descent parser is also called an LR(k) parser.
   - [✓] A recursive descent parser constructs the parse in a top-down manner.

2. Which of the following statements is true? (Only one statement is true.)

   - ☐ Precedence rules specify which character sequences are part of a programming language.
   - ☐ Precedence rules determine how much space values of complex types occupy in memory.
   - [✓] Precedence rules decide which operators group more tightly than others.
   - ☐ Precedence rules specify how to compute the scope of a variable.
   - ☐ Precedence rules determine whether two variable accesses are synchronized with each other.

3. Which of the following statements is true? (Only one statement is true.)

   - [✓] A "true" iterator is a subroutine that "yields" one element after another.
   - ☐ A "true" iterator is a logically controlled loop.
   - ☐ A "true" iterator exist exclusively in functional programming languages.
   - ☐ A "true" iterator uses tail recursion to call itself in an efficient way.
   - ☐ A "true" iterator accepts a function argument and implicitly apply the function to all elements of a list.

4. Which of the following statements is true? (Only one statement is true.)

   - ☐ Type checking reduces the memory footprint of a program.
   - ☐ Type checking relies on programmer-provided type annotations.
   - ☐ Type checking eliminates recursion and replaces it with loops.
   - [✓] Type checking ensures that a program obeys the type compatibility rules of the language.
   - ☐ Type checking is performed only on strongly typed languages.

Part 2 [12 points]

Consider the following Python code:

```
 1  x = [1, 2, 3]
 2  y = x
 3
 4  def f(a):
 5      def g(a):
 6          a = a
 7          x = 5
 8
 9      x = a
10      b = x
11      g(b)
12
13  if __name__ == "__main__":
14      f(y)
15      print(x)
```

Some reminders about Python: The language uses static scoping. Local variables are created upon the first assignment in a function. Lists are objects, and object references are passed using call by sharing.

1. Use the following table to describe all scopes that exist in the above code. Each row corresponds to a line number. Add one column for each scope and give the scopes meaningful names (i.e., *not* 1, 2, etc.). Mark which lines belong to which scope(s) by adding a cross (✗) at the corresponding cells in the table.

	Scopes		
Line	Global	Function `f`	Function `g`
1	✗		
2	✗		
6			✗
7			✗
9		✗	
10		✗	
11		✗	
14	✗		
15	✗		

2. Use the following table to indicate the lifetime of some values and bindings during the execution of the code by adding a cross (✗) at all lines where a value or binding is active.

Executed	Lifetime of values		Lifetime of bindings		
line	List created at line 1	Primitive created at line 7	`y` written at line 2	`b` written at line 10	`x` written at line 7
1	✗		✗		
2	✗		✗		
14	✗		✗		
9	✗		✗		
10	✗		✗	✗	
11	✗		✗	✗	
6	✗		✗	✗	
7	✗	✗	✗	✗	✗
15	✗		✗		

3. What is the value printed at line 15?

Solution: [1, 2, 3]

Part 3 [7 points]

The following code is in a toy language with JavaScript-inspired syntax. Variables and functions declared in the main script are global. Variables and functions declared within a function are local.

```
1  function f(a, b, c) {
2    // ...
3  }
4
5  function g(a, c) {
6    var b = 4;
7    a(a, b, c);
8  }
9
10 var a = 1;
11 var b = 2;
12 var c = 3;
13
14 var x = f;
15
16 g(x, b);
```

1. At what line(s) does the code create a reference to a function?

   Solution:
   At line 14 (where `x` is becoming a reference to function `f`).

2. Assume the language is using dynamic scoping and shallow binding.

   When reaching line 2, what values are the names `a`, `b`, and `c` bound to?

   - The value of `a` is function `f`.
   - The value of `b` is 4.
   - The value of `c` is 2.

3. Now, assume the language is using static scoping and deep binding.
   When reaching line 2, what values are the names `a`, `b`, and `c` bound to?

   - The value of `a` is function `f`.
   - The value of `b` is 4.
   - The value of `c` is 2.

Part 4 [9 points]

Consider the following definitions of structs in C:

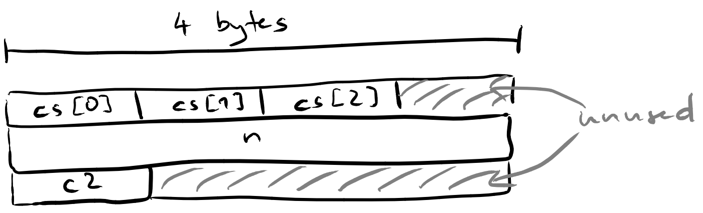
```
struct y {
  char cs[3];
  int n;
  char c2;
};

struct x {
  float coords[2];
  char c1;
  struct y *details;
};
```

Assumptions about sizes and alignment constraints:

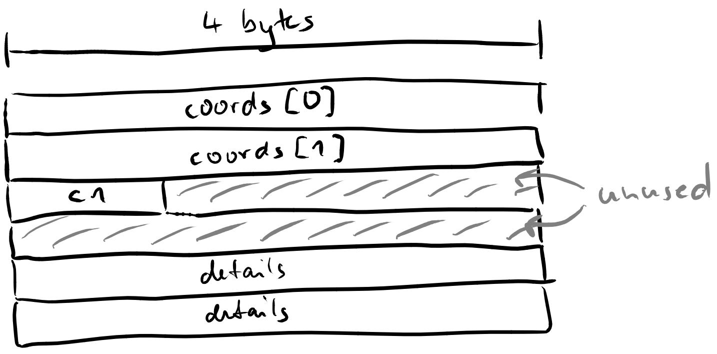
- For the sizes of primitive types, assume 1 byte for chars, 4 bytes for ints and floats, and 8 bytes for pointers.
- Assume no alignment requirement for chars, 4 byte alignment for ints, and 8 byte alignment for floats and pointers.
- There is no padding between consecutive elements of arrays.
- The alignment constraints must always be respected, i.e., no "packed" representation.
- The first offset in each struct is 0.
- The size of a struct always is a multiple of 4 bytes.

1. Assuming that the fields of structs may not be re-ordered, draw the most compact representation of the two structs using the following template. Clearly mark what each memory region is used for, or whether it is unused.

   Solution: struct y:



   Solution: struct x:



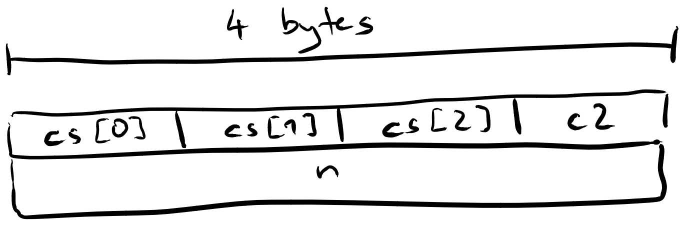
2. What is the size of the structs?

   - Struct `y` has size 12 bytes.

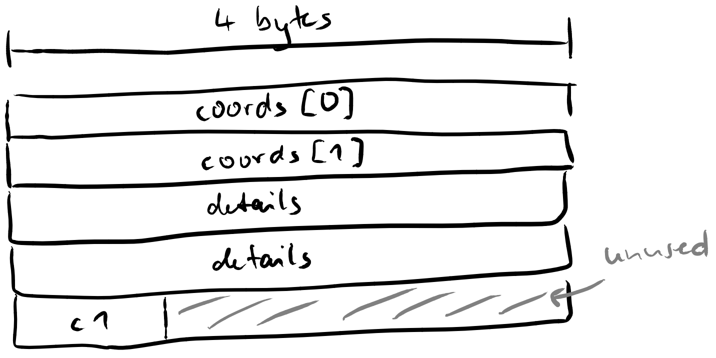
   - Struct `x` has size 24 bytes.

3. Now, assume that the fields of structs may be re-ordered. Again, draw the most compact representation of the two structs using the following template. Clearly mark what each memory region is used for, or whether it is unused.

   Solution: struct y:



   Solution: struct x:



4. What is the size of the structs?

   - Struct `y` has size 8 bytes.

   - Struct `x` has size 20 bytes.

Part 5 [10 points]

Consider the following code written in a toy language with Java-like syntax:

```
 1  class A {
 2      int m = 4;
 3      int n = 5;
 4
 5      public static void main(String[] args) {
 6          A a = new A();
 7          a.doIt(m, n);
 8          // Location 2
 9      }
10
11      void doIt(int i, int j) {
12          i = j;
13          // Location 1
14      }
15  }
```

1. The meaning of this code may depend on the parameter passing mode used by the language. Give the values that `m`, `n`, `i`, and `j` refer to at Location 1 (line 13) and Location 2 (line 8). Use the following table to indicate your answers.

Mode	Location 1				Location 2	
	`m`	`n`	`i`	`j`	`m`	`n`
Call by value	4	5	5	5	4	5
Call by reference	5	5	5	5	5	5
Call by value/result	4	5	5	5	5	5
Call by sharing	4	5	5	5	4	5

2. Which of the above behaviors do you get in Java for this code example?

   Solution:
   The program will have the "call by value" behavior, which is the parameter passing mode that Java uses for primitives.

Part 6 [12 points]

The following is about concurrency in Java. We consider a program where the main thread initializes two fields of a class:

```
1 public int a = 3;
2 public int b = 5;
```

After the initialization, the main threads spawns two additional threads T1 and T2, which execute the following code:

```
1 // Code in thread T1
2 a = a + 1;
3 if (b < 4)
4   a = a + b;
5
6 // location x
```

```
1 // Code in thread T2
2 b = 2;
```

1. Does the program contain a data race? If yes, indicate the memory access operations involved in the race.

   Solution:
   Yes, there are even two races: One between T1's read of `b` at line 3 and T2's write to `b`, and another one between T1's read of `b` at line 4 and T2's write to `b`

2. At "location x" (at the end of thread T1), could `a` have the value 4? If yes, explain what happens in the execution that leads to this value.

   Solution:
   Yes, this behavior is possible. For example, it happens when line 2 of T2 executes after the `if (b < 4)` check in T1 is reached, i.e., line 4 of T1 is not executed. In this execution, `a` gets assigned 4 at line 2 and keeps this value until "location x".

3. At "location x" (at the end of thread T1), could `a` have the value 6? If yes, explain what happens in the execution that leads to this value.

   Solution:
   Yes, this behavior is also possible. For example, it happens when line 2 of T2 executes before any of the statements in T1. In this execution, `b` is assigned value 2 in T2, then `a` is assigned 4 at line 2 of T1, then the `if (b < 4)` check succeeds because the new value of `b` is read by T1, and finally `a` gets increased by 2 (i.e., the new value of `b`) and hence is 6.

4. At "location x" (at the end of thread T1), could `a` have the value 7? If yes, explain what happens in the execution that leads to this value.

   Solution:
   No, this behavior is impossible.

5. At "location x" (at the end of thread T1), could `a` have the value 9? If yes, explain what happens in the execution that leads to this value.

   Solution:
   Yes, this behavior is possible. It may happen because the Java memory model allows a read of a shared variable to "see" any value the variable has had since the last synchronization point. Because of the races from question 1, the reads of `b` at lines 3 and 4 of T1 may either see `b` initial value 5 or the newly written value 2. As a result, an execution may read `b==2` at line 3 of T1 and hence take the branch, but then read `b==5` at line 4 and hence increase `a` to 9.

6. Now, suppose that field `b` is declared to be `volatile`. Would any of your above answers change? If yes, explain why and how.

   Solution:
   If `b` is volatile, then all accesses to `b` are synchronized, i.e., there is no data race anymore. As a result, reading an "old" value, as in question 5, is impossible, and the answer to question 5 becomes "no".

Part 7 [6 points]

This task is about code written in the function programming language Scheme. Your task is to understand the following code pieces and to answer some questions about them.

1. Scheme code:

```
(let
  ((f (lambda (x) (- x 1))))
  (f 3)
)
```

   (a) What does this code evaluate to?

   Solution:
   The code evaluates to 2.

   (b) What kind of operation does the `(f 3)` perform?

   Solution:
   The operation performs a function call (where argument 3 is given to function `f`).

2. Scheme code:

```
(cdr (cons 1 '(4 2 3)))
```

   (a) What does this code evaluate to?

   Solution:
   The code evaluates to (4 2 3), i.e., the list of three values.

   (b) What is the behavior of the program if we remove the `'`?

   Solution:
   In that case, the program results in an error, because it tries to call function 4 with arguments 2 and 3, but the number 4 is not a function.

3. Scheme code:

```
1  (if
2    (
3      <
4      ((lambda (f g) (* f g)) (+ 1 3) (- 5 2))
5      15
6    ) "hey" 23
7  )
```

   (a) What does this code evaluate to?

   Solution:
   The code evaluates to "hey".

   (b) What single-character change will cause the code to evaluate to a value of a different type?

   Solution:
   For example, changing `15` to `5` at line 5 will result in a program that evaluates to 23, i.e., a number instead of a string.