UCLA CCLE

Winter 2021 - COM SCI131-1 - EGGERT

Started on	Tuesday, 9 February 2021, 2:15 PM PST
State	Finished
Completed on	Tuesday, 9 February 2021, 4:04 PM PST
Time taken	1 hour 49 mins
Points	84.00/100.00
Grade	**92.40** out of 110.00 (**84**%)

Question **1**

Complete

3.00 points out of 3.00

Write an example call to the OCaml function defined by:

let rec a b c = b (a b) c

```
let temp f x = f x;;
let res = a temp 2;; (* Actual call *)
(* This results in infinite recursion, but it's still a valid function call *)
```

UCLA CCLE :)

Complete

1.50 points out of 5.00

Suppose tokenizers were generous instead of greedy. That is, suppose that, instead of taking the longest sequence of characters that would be a valid token, they take the shortest sequence. Explain what would go wrong. Use OCaml as an example.

```
This would be problematic specifically in OCaml, as it would compromise the ability to curry.
A structure such as f a b c where c is passed to b, which is then passed to a is very common.
With greedy tokenization this works fine as the entire expression (if valid) will be tokenized.
However, since f a is also valid, a “generous” tokenizer might simply apply f to a and then separately
c to b which would produce a different, incorrect result.
```

Complete

10.00 points out of 10.00

In the Awk programming language, the expression ‘E1 E2’, where E1 and E2 are expressions, stands for string concatenation. For example, the Awk expression ‘"abc" "de"’ evaluates to the string "abcde". Suppose we extend Homework 2’s awkish_grammar to support string concatenation, by replacing ‘Binop -> [["+"]; ["-"]]’ with ‘Binop -> [[]; ["+"]; ["-"]]’, so that the resulting grammar looks like this:

```
let ambigrammar =
  (Expr,
   function
     | Expr ->
         [[N Term; N Binop; N Expr];
          [N Term]]
     | Term ->
         [[N Num];
          [N Lvalue];
          [N Incrop; N Lvalue];
          [N Lvalue; N Incrop];
          [T"("; N Expr; T")"]]
     | Lvalue ->
         [[T"$"; N Expr]]
     | Incrop ->
         [[T"++"]; [T"--"]]
     | Binop ->
         [[]; [T"+"]; [T"-"]]
     | Num ->
         [[T"0"]; [T"1"]; [T"2"]; [T"3"]; [T"4"];
          [T"5"]; [T"6"]; [T"7"]; [T"8"]; [T"9"]])
```

Convert ambigrammar to ISO EBNF.

```
Term = Num | Lvalue | Incrop, Lvalue | Lvalue, Incrop | "(", Expr, ")";
Lvalue = "$" Expr;
Incrop = "++" | "--";
Binop = [ "+" | "-" ];
Num = "1"| "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
```

Complete

10.00 points out of 10.00

Prove that ambigrammar is ambiguous.

```
The string $2++2 can be parsed 2 different ways, according to ambigrammar.

Parse tree 1 (Increment 2):
Expr -> Term Binop Expr ($2++2)
    Term -> LValue ($2)
        Lvalue -> "$" Expr
            Expr -> Term
                Term -> Num
                    Num -> "2"

    Binop -> []

    Expr -> Term (++2)
        Term -> Incrop LValue
            Incrop -> "++"
            Lvalue -> Expr
                Expr -> Term
                    Term -> Num
                        Num -> "2"

Parse tree 2 (Increment $2)
Expr -> Term Binop Expr ($2++2)
    Term -> LValue Incrop ($2++)
        Lvalue -> "$" Expr
            Expr -> Term
                Term -> Num
                    Num -> "2"
        Incrop -> "++"

    Binop -> []

    Expr -> Term (2)
        Term -> Num
             Num -> "2"
```

Complete

15.00 points out of 15.00

Suppose you have a working solution to Homework 2, and apply it to ambigrammar. What will your resulting parser do when given ambiguous input? Explain with an example.

```
A working solution to Homework 2 requires that the rules of the grammar be checked in order.
This means that my solution will read down the list until a successful rule match is found.
Due to this constraint, the resulting parser generates the leftmost derivation of a given string.
Take the ambiguous string presented in question 4: $2++2. When a parser resulting from this solution
is used on this string, our parser will see it as the following:

$2++2
Check Expr -> Term, Binop, Expr (Success)
   Check Term -> Num (Fail)
   Check Term -> Lvalue (Success)
   Check Lvalue -> "$" Expr (Success)
        Check Expr -> Term, Binop, Expr (Fail)
        Check Expr -> Term (Success)
                Check Term -> Num (Success)
                        Check Num -> “0” (Fail)
                        Check Num -> “1” (Fail)
                        Check Num -> “2” (Success)
   Check Binop -> [] (Success)
   Check Expr -> Term, Binop, Expr (Fail)
   Check Expr -> Term (Success)
        Check Term -> Num (Success)
        Check Num -> “0” (Fail)
        Check Num -> “1” (Fail)
        Check Num -> “2” (Success)
```

UCLA CCLE :)

Complete

2.50 points out of 15.00

Consider the DNA fragment analyzer in the hint code at the end of the old version of Homework 2. This code defines the type 'matcher' via 'type matcher = fragment -> acceptor -> fragment option'. Suppose we change the calling convention for matchers by having their argument being the acceptor, not the fragment, so that we define the type via 'type matcher = acceptor -> fragment -> fragment option'. Modify the hint code to use this new calling convention. Your modified version should be simple and elegant (as opposed to being as close to the original as possible).

```
  match accept frag with
    | None ->
        matcher frag
                (fun frag1 ->
                   if frag == frag1
                   then None
                   else match_star matcher frag1 accept)
    | ok -> ok

let match_nucleotide nt accept frag =
  match frag with
    | [] -> None
    | n::tail -> if n == nt then accept tail else None

let append_matchers matcher1 matcher2 frag accept =
  matcher1 frag (fun frag1 -> matcher2 frag1 accept)

let make_appended_matchers make_a_matcher accept ls =
  let rec mams = function
    | [] -> match_empty
    | head::tail -> append_matchers (make_a_matcher accept head) (mams tail)
  in mams ls

let rec make_or_matcher make_a_matcher accept = function
  | [] -> match_nothing
  | head::tail ->
      let head_matcher = make_a_matcher accept head
      and tail_matcher = make_or_matcher make_a_matcher accept tail
      in fun frag accept ->
           let ormatch = head_matcher accept frag
           in match ormatch with
                | None -> tail_matcher accept frag
                | _ -> ormatch

let rec make_matcher accept frag = function
  | Frag frag -> make_appended_matchers match_nucleotide accept frag
  | List pats -> make_appended_matchers make_matcher accept pats
  | Or pats -> make_or_matcher make_matcher accept pats
  | Junk k -> match_junk k
  | Closure pat -> match_star (make_matcher accept pat)
```

Complete

10.00 points out of 10.00

Suppose we want to extend C++ to support Java-style generics, using a slightly different syntax when using generics (e.g., List<*String*> instead of List<String>). Conversely, suppose we also want to extend Java to support C++-style templates, using a slightly different syntax when using templates (e.g., List</String/> instead of List<String>).

Which of these two language extensions would be harder to implement and document, and why?

```
It would be harder to allow C++ to support Java-style generics. C++ implements templates by compiling t
when the types are actually defined by the caller/user of the template. This results in a need for mult
copies of the machine code for each instantiation. On the other hand, generics work with only 1 copy of
machine code that works on any type of argument. This works in Java, because all Java types are represe
a pointer under the hood. This means that all types "smell the same" in the language. In C++, this is n
types have many various representations. As a result, it would be very possible to create duplicate cop
the machine code to implement templates in Java, but implementing generics in C++ would require a compl
overhaul of how types are represented.
```

Complete

10.00 points out of 10.00

Does C++ support duck typing (as described in class)? Briefly explain why or why not.

C++ does not support duck typing because C++ is statically typed. Runtime duck typing is expensive, so C++ doesn't support it. During compilation, all placeholder types have to be substituted with concrete types specified in a particular instantiation. When a C++ program executes object.foo(x), and object is a reference to a base class with a virtual function foo, C++'s inheritance requirements ensure that object.foo actually refers to an object that has a member function named foo, and that function's return type has the same internal representation, regardless of which derived class foo is actually called.

Complete

10.00 points out of 10.00

If a program contains an exit monitor operation A followed by a normal store B, the Java Memory model allows the thread's implementation to reorder operations so that it does B before A. However, the reverse is not true: if the program contains a normal store B followed by an exit monitor A, the JMM does not allow the thread's implementation to do A followed by B. Briefly explain why the former is correct but the latter is not.

```
This is because it is always fine to require more operations to be synchronized, even though this
might lead to inefficient execution. It's fine to reorder an (exit monitor, normal store) into
(normal store, exit monitor), because this is simply moving the normal store into the synchronized
portion, which will behave as if it followed the exit monitor.

However, if we have an (normal store, exit monitor) in the code, then that code may require that the
normal store be synchronized to produce the correct results. Therefore, if we were to change the order
to (exit monitor, normal store), we no longer have a guarantee that it will exhibit the correct behavio
Exiting the monitor and then performing the normal store might create a race condition that the
programmer put the synchronization in place to avoid.
```

UCLA CCLE :)

Complete

12.00 points out of 12.00

The Java skeleton code in Homework 3's jmm.jar has a bug: its output line "Average swap time S ns real" overestimates the actual real average swap time, because the code assumes that each thread consumes real time equal to the difference between the last ending time of any thread and the first starting time of any thread. In theory the overestimate error could be large, as some threads could consume considerably more real time than others. Fix the bug by modifying the code to measure each thread's real time independently. Be careful to avoid race conditions in your fix.

```
AtomicLongArray realTimeArray = new AtomicLongArray(nThreads);
        for (var i = 0; i < nThreads; i++) {
                realTimeArray.set(i, System.nanoTime());
            t[i].start ();
        }
        for (var i = 0; i < nThreads; i++) {
            t[i].join ();
                realTimeArray.getAndAdd(i, -System.nanoTime());
        }

        long realtime = 0, cputime = 0;
        for (int i = 0; i < nThreads; i++){
                // negative so we get positive delta time
                realtime += -realTimeArray.get(i);
        }

        for (var i = 0; i < nThreads; i++)
            cputime += test[i].cpuTime();
        double dTransitions = nTransitions;
        System.out.format("Total time %g s real, %g s CPU\n",
                          realtime / 1e9, cputime / 1e9);
        System.out.format("Average swap time %g ns real, %g ns CPU\n",
        realtime / dTransitions * nThreads,
                          cputime / dTransitions);
```

◀ Sample midterm (2008) with ...

Jump to...