Question 1:
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 *)

Question 2:
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 apply c to b which would produce a different, incorrect result.

Question 3:
Expr = Term, Binop, Expr | Term;
Term = Num | Lvalue | Incrop, Lvalue | Lvalue, Incrop | “(”, Expr, “)”;
Lvalue = “$” Expr;
Incrop = “++” | “–”;
Binop = [“+” | “-”];
Num = “1”| “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”;

Question 4:
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"
```

Question 5:

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)
```

Question 6:

type nucleotide = A | C | G | T
type fragment = nucleotide list
type acceptor = fragment -> fragment option

```
type matcher = fragment -> acceptor -> fragment option

type pattern =
  | Frag of fragment
  | List of pattern list
  | Or of pattern list
  | Junk of int
  | Closure of pattern

let match_empty frag accept = accept frag

let match_nothing frag accept = None

let rec match_junk k frag accept =
  match accept frag with
    | None ->
        (if k = 0
          then None
          else match frag with
                 | [] -> None
                 | _::tail -> match_junk (k - 1) tail accept)
    | ok -> ok

let rec match_star matcher frag accept =
  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 = match frag with
  | 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)
```

Question 7:
It would be harder to allow C++ to support Java-style generics. C++ implements templates by compiling the code when the types are actually defined by the caller/user of the template. This results in a need for multiple copies of the machine code for each instantiation. On the other hand, generics work with only 1 copy of the machine code that works on any type of argument. This works in Java, because all Java types are represented by a pointer under the hood. This means that all types "smell the same" in the language. In C++, this is not true; types have many various representations. As a result, it would be very possible to create duplicate copies of the machine code to implement templates in Java, but implementing generics in C++ would require a complete overhaul of how types are represented.

Question 8:
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.

Question 9:

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 behavior. Exiting the monitor and then performing the normal store might create a race condition that the programmer put the synchronization in place to avoid.

Question 10:

```
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);
```