NATIONAL UNIVERSITY OF SINGAPORE

CS1101S — PROGRAMMING METHODOLOGY

CURATED VERSION OF 16/11/2020 (CORRECTED ON 16/11/2020, 15:00)

(Semester 1 AY2019/2020)

Time Allowed: **2 Hours**

SOLUTION

INSTRUCTIONS TO STUDENTS

1. This assessment paper contains **NINE (9)** questions and comprises **TWENTY (20)** printed pages, including this page.
2. The full score of this paper is **75 marks**.
3. This is a **CLOSED BOOK** assessment, but you are allowed to use **ONE** A4 sheet of notes.
4. Answer **ALL** questions **within the space provided** in this booklet.
5. Where programs are required, write them in the **Source §4** language.
6. In any answer, you can assume a correct solution from any preceding question or sub-question as given, even if *your* solution from that preceding (sub-)question was not correct.
7. Write legibly with a **pen or pencil**. **Untidiness will be penalized**.
8. Do not tear off any pages from this booklet.
9. Write your **Student Number** below **using a pen**. Do not write your name.

(write legibly with a pen)

Student No.:

This portion is for examiner's use only

Q#	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**Σ**
MAX	4	4	4	10	8	18	6	15	6	75
SC										

Question 1: Evaluation [4 marks]

In the lectures, we have encountered the assignment statement. Consider a Source program *P* that contains assignment statements, which are executed during evaluation of the program. Indicate for each of the following statements, whether it is **true** (**T**) or **false** (**F**). There is no need for any explanation:

1A. [1 mark]

Program *P* can be evaluated using a meta-circular evaluator.

T

1B. [1 mark]

Program *P* can be evaluated using the substitution model.

F

1C. [1 mark]

Program *P* can be evaluated using the environment model.

T

1D. [1 mark]

Program *P* can be evaluated using the Source Academy (with Source §4 selected).

T

Question 2: Scope [4 marks]

In the lectures, we have introduced the notion of name scope. Indicate for each of the following statements, whether it is **true** (**T**) or **false** (**F**). There is no need for any explanation:

2A. [1 mark]

The scope of names determines the evaluation order: applicative-order reduction vs. normal-order reduction.

F

2B. [1 mark]

The scope of names can change during evaluation of a program.

F

2C. [1 mark]

The scope of names determines the order in which arguments of functions are evaluated: from left to right or from right to left.

F

2D. [1 mark]

The scope of names determines which declarations particular occurrences of names refer to.

T

Question 3: Pairs [4 marks]

In the lectures, we have introduced the functions `pair`, `head` and `tail` as primitive functions that are built into the language processing system Source Academy used in the module. Indicate for each of the following statements, whether it is **true** (**T**) or **false** (**F**). There is no need for any explanation:

3A. [1 mark]

The functions `pair`, `head` and `tail` could be implemented just by using function definitions, function applications and constant declarations.

T

3B. [1 mark]

The functions `pair`, `head` and `tail` could be implemented using array operations.

T

3C. [1 mark]

The functions `pair`, `head` and `tail` can be added to a meta-circular evaluator by adding them to the list of primitive functions.

T

3D. [1 mark]

The function `stream_tail` for stream processing can be implemented using the functions `pair`, `head` and/or `tail` and function application.

T

Question 4: It's a Mystery! [10 marks]

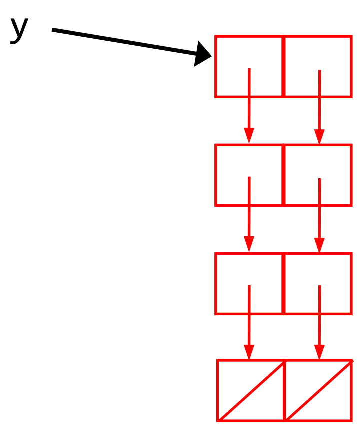
Consider the following Source program:

```
function mystery(x) {
    if (x === 0) {
        return null;
    } else {
        const ys = mystery(x - 1);
        return pair(ys, ys);
    }
}
```

4A. [5 marks]

Draw the box-and-pointer diagram for the value of the constant `y`, after evaluating the following program.

```
const y = mystery(4);
```



4B. [1 mark]

What is the result of evaluating the following program?

```
mystery(4) === mystery(4);
```

false

4C. [1 mark]

What is the result of evaluating the following program?

```
equal(mystery(5), mystery(5));
```

true

4D. [1 mark]

What is the result of evaluating the following program?

```
const z = mystery(3);
equal(head(z), head(z));
```

true

4E. [1 mark]

What is the result of evaluating the following program?

```
const w = mystery(7);
head(w) === head(w);
```

true

4F. [1 mark]

What is the result of evaluating the following program?

```
const v = mystery(6);
head(v) === tail(v);
```

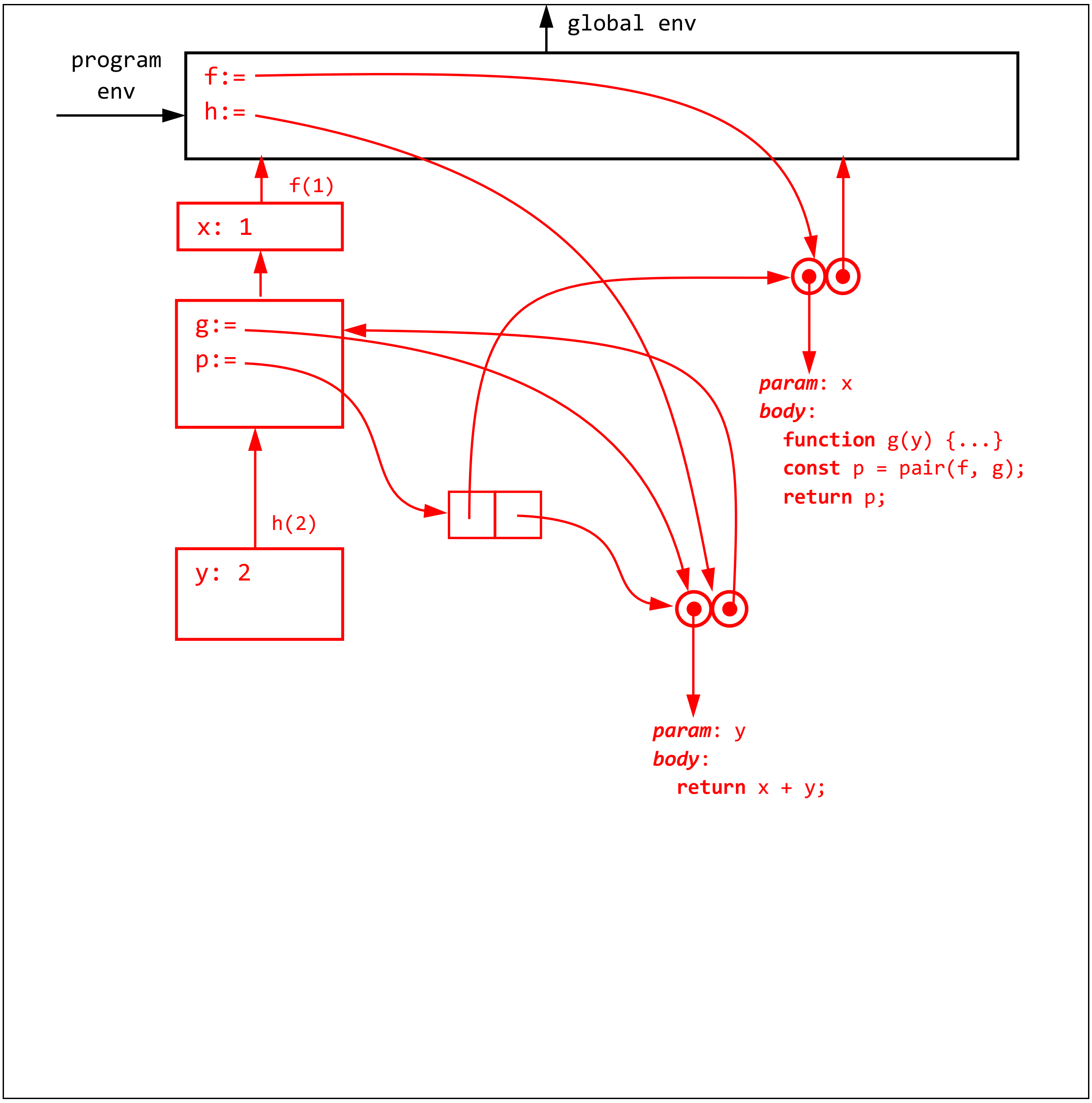
true

Question 5: Environment Model [8 marks]

Consider the following Source program:

```
function f(x) {
    function g(y) {
        return x + y;
    }
    const p = pair(f, g);
    return p;
}
const h = tail(f(1));
h(2);
```

Draw the environment model diagram for this program. Make sure to include all non-empty frames that are created during the evaluation of the program, except the global environment frame. Show the final value of each binding.



Question 6: Arraytrees [18 marks]

A tree of numbers is a list whose elements are numbers or trees of numbers. Analogously, we define an *arraytree of numbers* as an array whose elements are numbers or arraytrees of numbers.

For example, the array `[[10, 20, 30], [30, 20, 10]]` is an arraytree of numbers, because both its elements are arraytrees of numbers.

6A. [6 marks]

Write a function `tree_to_arraytree` that converts a tree of numbers to an arraytree of numbers. The following examples clarify the required behavior of the function.

Examples:

```
tree_to_arraytree(list());
  // returns []

tree_to_arraytree(list(10, 20, 30));
  // returns [10, 20, 30]

tree_to_arraytree(list(list(10, 20, 30), list(30, 20, 10)));
  // returns [[10, 20, 30], [30, 20, 10]]
```

```
function tree_to_arraytree(xs) {

    if (is_number(xs)) {
        return xs;
    } else {
        const a = [];
        let i = 0;
        while (!is_null(xs)) {
            a[i] = tree_to_arraytree(head(xs));
            i = i + 1;
            xs = tail(xs);
        }
        return a;
    }

}
```

6B. [6 marks]

Write a function `arraytree_to_tree` that converts an arraytree of numbers to a tree of numbers.

The following examples clarify the required behavior of the function.

Examples:

```
arraytree_to_tree([]);
  // returns null

arraytree_to_tree([10, 20, 30]);
  // returns a value equal to the result of list(10, 20, 30)

arraytree_to_tree([[10, 20, 30], [30, 20, 10]]);
  // returns a value equal to the result of
  // list(list(10, 20, 30), list(30, 20, 10)));
```

```
function arraytree_to_tree(a) {

    if (is_number(a)) {
        return a;
    } else {
        let xs = null;
        const len = array_length(a);
        for (let i = len - 1; i >= 0; i = i - 1) {
            xs = pair(arraytree_to_tree(a[i]), xs);
        }
        return xs;
    }

}
```

6C. [6 marks]

In this question, you can use the function `permutations` presented in the lectures, which computes the set of all permutations of a given set of numbers.

```
function permutations(s) {
    return is_null(s)
        ? list(null)
        : accumulate(append, null,
            map(x => map(p => pair(x, p),
                         permutations(remove(x, s))),
                s));
}
```

Example:

```
const my_s = list(10, 20, 30);
const my_permutations = permutations(my_s);
```

The value of `my_permutations` is now equal to

```
list(list(10, 20, 30), list(10, 30, 20),
     list(20, 10, 30), list(20, 30, 10),
     list(30, 10, 20), list(30, 20, 10))
```

Note that both the given set and the result set of the function `permutations` are represented by lists.

In this question, you need to provide a function `array_permutations` that computes an array of permutations from a given set. The given set and each permutation of the result need to be represented by arrays of numbers.

Example:

```
const my_a = [10, 20, 30];
const my_array_permutations = array_permutations(my_a);
```

The value of `my_array_permutations` is now printed as

```
[[10, 20, 30], [10, 30, 20],
 [20, 10, 30], [20, 30, 10],
 [30, 10, 20], [30, 20, 10]]
```

The permutations are allowed to appear in different order.

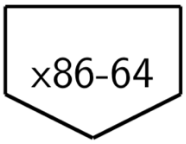
```
function array_permutations(a) {

    return tree_to_arraytree(permutations(arraytree_to_tree(a)));

}
```

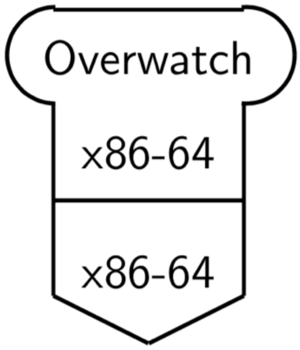
Question 7: Language Processors [6 marks]

In the lectures, we have introduced components, such as **processors** and **programs**, and saw that we can run programs on a processor if they are written in the **machine code** that the processor is designed to execute.



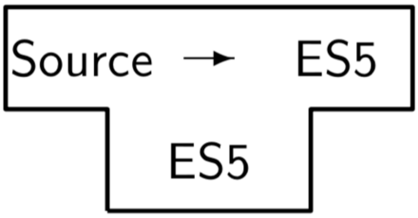
x86-64 processor

program "Overwatch" written in x86-64 machine code



"Overwatch" running on x86-64

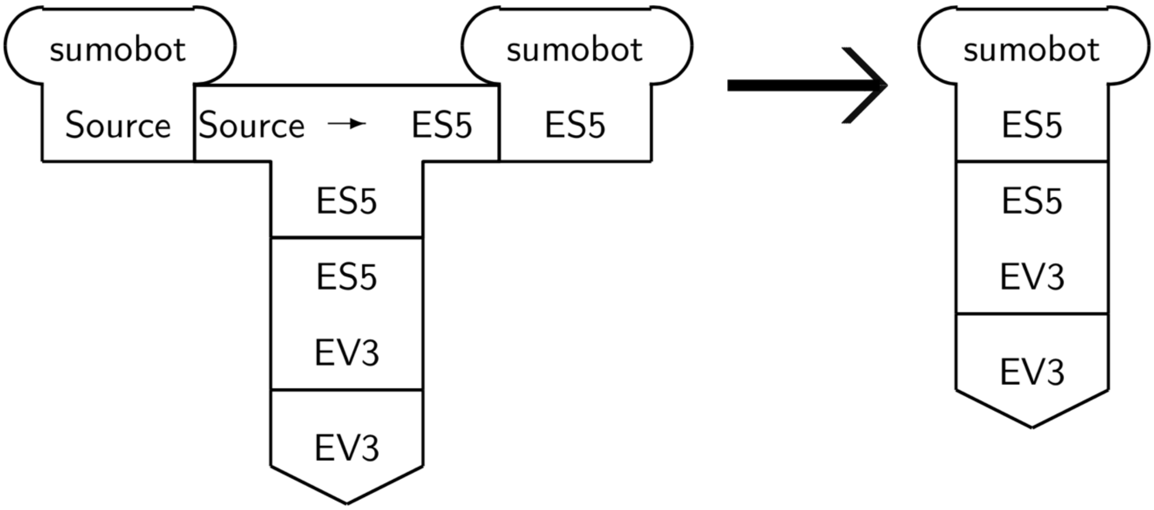
We have also seen two special kinds of programs: **compilers** that can translate programs from one language to another, and **interpreters** that can run programs written in a particular language.



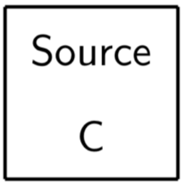
compiler from Source to ES5, written in ES5

interpreter for ES5, written in EV3

For example, our team installed a compiler from Source to ES5 in order to run "sumobot" Source programs on the EV3 processors built into the Lego Mindstorms bricks.



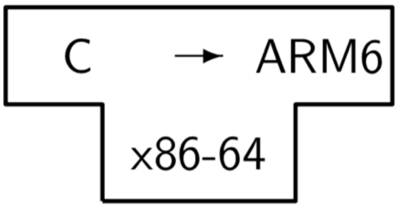
Rumor has it that next year, Lego will come out with bricks that have ARM6 processors inside. Then we could directly use an interpreter for Source on the ARM6, instead of compiling the Source programs on the brick. We do have an interpreter for Source, but it is written in the language C, instead of the ARM6 machine code.



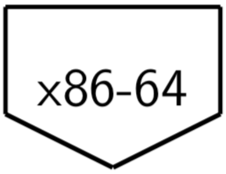
interpreter for Source, written in C

ARM6 processor

Fortunately, we have a compiler that can translate programs written in C to programs written in ARM6 machine code. The compiler is written in x86-64, and we have an x86-64 processor.

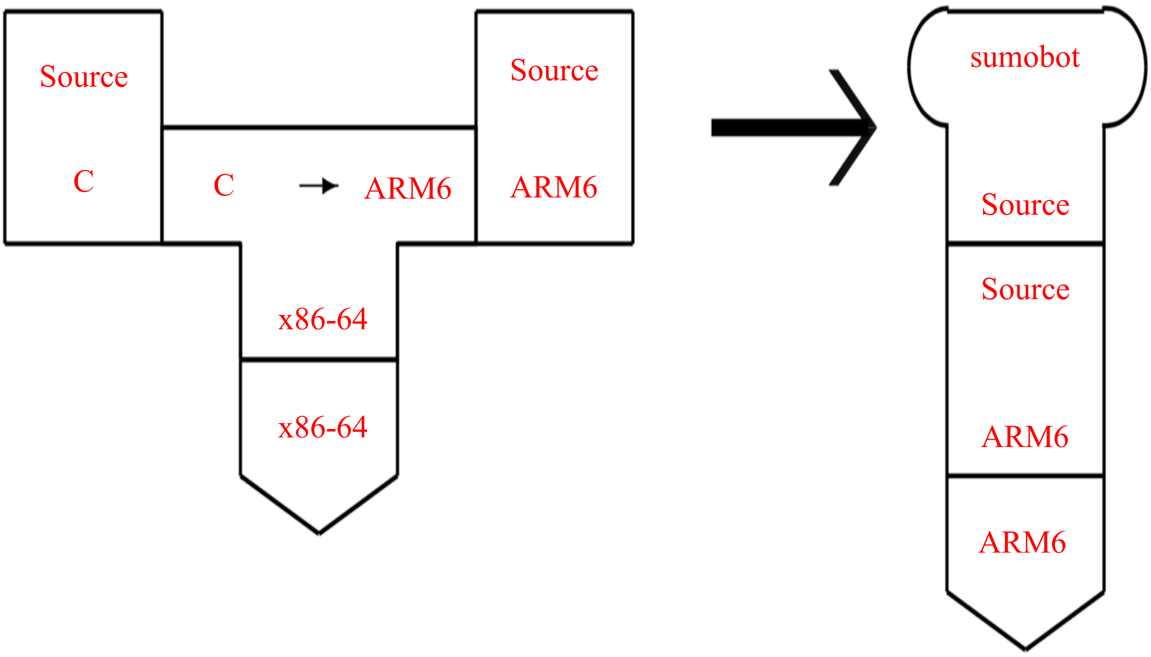


C-to-ARM6 compiler, written in x86-64



x86-64 processor

Complete the following T-diagram of the language processing steps that will allow us to run our Source program “sumobot” on the ARM6 processor.



Question 8: Binary Permutations [15 marks]

Consider the `permutations` function from the lectures:

```
function permutations(s) {
    return is_null(s)
        ? list(null)
        : accumulate(append, null,
            map(x => map(p => pair(x, p),
                         permutations(remove(x, s))),
                s));
}
```

This function also works when the given list contains duplicates: Each permutation in the result list has as many duplicates as the given list, and all permutations are included in the result.

In this question, we will look at a special case where only the numbers `0` and `1` occur in the input. For example,

```
permutations(list(0, 1, 1, 0, 0, 0));
```

Unfortunately, the function `permutations` computes repeated permutations. Part A avoids the computation of repeated permutations, and Part B optimizes the solution using memoization.

8A. [7 marks]

Write a function `perms01` that takes two numbers `n` and `m` as arguments and returns a list of all permutations with `n` occurrences of `0` and `m` occurrences of `1`. Your implementation should be more efficient than the function `permutations`: **During the evaluation of `perms01(n, m)`, any permutation of length `n + m` should be computed at most once**, but permutations of length less than `n + m` might be computed repeatedly.

Example:

```
perms01(2, 3);
  // returns
  // list(list(1, 1, 1, 0, 0), list(1, 1, 0, 1, 0), list(1, 1, 0, 0, 1),
  //      list(1, 0, 1, 1, 0), list(1, 0, 1, 0, 1), list(1, 0, 0, 1, 1),
  //      list(0, 1, 1, 1, 0), list(0, 1, 1, 0, 1), list(0, 1, 0, 1, 1),
  //      list(0, 0, 1, 1, 1))
  // These 10 permutations may appear in a different order in the result,
  // but each permutation should be computed only once
```

```
function perms01(n, m) {
```

(*more writing space next page*)

```
    if (n === 0 && m === 0) {
        return list(null);
    } else {
        const p0 = (n > 0)
            ? map(p => pair(0, p), perms01(n - 1, m))
            : null;
        const p1 = (m > 0)
            ? map(p => pair(1, p), perms01(n, m - 1))
            : null;
        return append(p0, p1);
    }
}
```

8B. [8 marks]

In the lectures, we have encountered the idea of memoization to optimize a given algorithm. Write a function `perms01memo` that solves the problem described in 8A, but meets the following additional requirement: **During the evaluation of `perms01memo(n, m)`, any permutation of length *less than or equal to* `n + m` should be computed at most once.**

(more writing space next page)

```
function two_d_memoize(f) {
    const mem = [];
    function read(x, y) {
        return (mem[x] === undefined) ?
            undefined : mem[x][y];
    }
    function write(x, y, value) {
        if (mem[x] === undefined) {
            mem[x] = []; } else {}
        mem[x][y] = value;
    }
    function mf(x, y) {
        const mem_xy = read(x, y);
        if (mem_xy !== undefined) {
            return mem_xy;
        } else {
            const result = f(x, y);
            write(x, y, result);
            return result;
        }
    }
    return mf;
}

const perms01memo =
two_d_memoize((n, m) => {
    if (n === 0 && m === 0) {
        return list(null);
    } else {
        const p0 = (n > 0)
            ? map(p => pair(0, p), perms01memo(n - 1, m))
            : null;
        const p1 = (m > 0)
            ? map(p => pair(1, p), perms01memo(n, m - 1))
            : null;
        return append(p0, p1);
    }
});
```

Question 9: Meta-circular Evaluator [6 marks]

Consider a meta-circular evaluator for a sublanguage of Source §2. Modify the evaluator such that the name `recurse` occurring inside a function body always refers to the closest surrounding function definition or declaration. You can assume that the name `recurse` is never explicitly declared by programs in the interpreted sublanguage of Source §2.

You can also assume that any name is only declared once in the interpreted sublanguage of Source §2.

Examples:

```
parse_and_eval("(n => n <= 1 ? 1 : n * recurse(n - 1))(4);");
  // evaluates to 24

parse_and_eval("                         \
const d = x => recurse;                  \
d === d(1);                               ");
  // evaluates to true

parse_and_eval("                         \
function f(x) {                          \
    return pair(recurse, x);             \
}                                        \
const g = head(f(6));                    \
tail(g(7));                               ");
  // evaluates to 7
```

You can achieve the desired result by just changing the function `apply` of the evaluator. On the following page, we have included a simplified version of the `apply` function. You can assume that the given `apply` function behaves correctly for any program in the interpreted sublanguage of Source §2, except that the name `recurse` is not treated as required.

We leave extra space before and after each line. You may strike out any line and write your modifications in the provided space.

```
// feel free to strike out any part and replace it with your program
function apply(fun, args) {
   if (is_primitive_function(fun)) {
      return apply_primitive_function(fun, args);
   } else if (is_compound_function(fun)) {

      const body = function_body(fun);

      const locals = local_names(body);

      const names = append(function_parameters(fun), locals);

      const temp_values = map(x => no_value_yet, locals);

      const values = append(args, temp_values);

      const result =

         evaluate(body,

                  extend_environment(
                      replace following line with  pair("recurse", names),
                      names,
                      replace following line with  pair(fun, values),
                      values,

                      function_environment(fun)));

      if (is_return_value(result)) {
         return return_value_content(result);
      } else {
          return undefined;
      }

   } else {
      error(fun, "Unknown function type in apply");
   }
}
```

—— END OF QUESTIONS ——

Appendix

Some Primitive Functions

The following are some of the primitive functions in Source §4:

- `display(x)`
- `is_number(x)`
- `is_boolean(x)`
- `is_string(x)`
- `pair(x, y)`
- `head(x)`
- `tail(x)`
- `list(x1, x2,..., xn)`
- `is_null(x)`
- `is_pair(x)`
- `set_head(p, x)`
- `set_tail(p, x)`
- `array_length(x)`
- `is_array(x)`
- `stream_tail(x)`

Some Pre-declared Functions

Some of the pre-declared functions in Source §4 are declared as follows:

```
function map(f, xs) {
    return is_null(xs)
        ? null
        : pair(f(head(xs)), map(f, tail(xs)));
}

function filter(pred, xs) {
    return is_null(xs)
        ? xs
        : pred(head(xs))
          ? pair(head(xs), filter(pred, tail(xs)))
          : filter(pred, tail(xs));
}

function accumulate(op, initial, xs) {
    return is_null(xs)
        ? initial
        : op(head(xs), accumulate(op, initial, tail(xs)));
}

function append(xs, ys) {
    return is_null(xs)
        ? ys
        : pair(head(xs), append(tail(xs), ys));
}
```

(Blank page. Do not tear off.)

END OF PAPER