NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING

MIDTERM QUIZ

ADAPTED TO SOURCE 2021 IN 9/2020

Semester 1 AY2014/2015

CS1101S — PROGRAMMING METHODOLOGY

1 October 2014 Time Allowed: **1 Hour 40 Minutes**

Matriculation No.: | | | | | | | | | |

Instructions (please read carefully):

1. Write down your **matriculation number** on the **question paper**. DO NOT WRITE YOUR NAME ON THE QUESTION SET!
2. This is an **open-sheet quiz**. You are allowed to bring one A4 sheet of notes (written or printed on both sides).
3. This paper comprises **6 questions** and **TWENTY (20)** printed pages. The time allowed for solving this quiz is **1 hour 40 minutes**.
4. The maximum score of this quiz is **60 marks**. The weight of each question is given in square brackets beside the question number.
5. All questions must be answered correctly for the maximum score to be attained.
6. All questions must be answered in the space provided in the question paper; no extra sheets will be accepted as answers.
7. The pages marked "scratch paper" in the question paper may be used as scratch paper.
8. You are allowed to use pencils or pens, as you like (no red color, please).
9. Write legibly; **UNTIDINESS will be penalized**.

GOOD LUCK!

Q#	1	2	3	4	5	6	Σ
MAX	5	5	7	15	11	17	60
SC							

Question 1: Making Logical Sense [5 marks]

Rewrite the following Source expressions without using any of the logical operators `&&`, `||`, and `!`. The result of evaluating your expression must be exactly the same as that of the original, regardless of the values of `cond_a`, `cond_b`, `do_x` and `do_y`.

A. [1 mark]

```
!cond_a ? do_x() : do_y()
```

```
if (cond_a) {
    return do_y();
} else {
    return do_x();
}
```

B. [2 marks]

```
(cond_a || cond_b) ? do_x() : do_y()
```

```
if (cond_a) {
    return do_x();
} else if (cond_b) {
    return do_x();
} else {
    return do_y();
}.
```

C. [2 marks]

```
(cond_a && cond_b) ? do_x() : do_y()
```

```
if (cond_a) {
    if(cond_b) {
        return do_x();
    } else {
        return do_y();
    }
} else {
    return do_y();
}
```

Question 2: Almost Malfunction [5 marks]

A. [4 marks]

Consider a restricted form of Source in which functions are allowed to have at most one parameter. Rewrite the following function definition under this restriction. You must not use lists or any structure created using `pair`.

```
function myfunc(a, b, c) {
    return a * b + c;
}
```

```
function myfunc(a) {
    function inner(b) {
        function inner_2(c) {
            return a * b + c;
        }
        return inner_2;
    }
    return inner;
}
```

return b => c => a * b * c

B. [1 mark]

With your new function definition in Part A, how would you rewrite the function call `myfunc(3, 2, 1);`?

```
my_func(3)(2)(1);
```

Question 3: Function Composition Reincarnated [7 marks]

Considered the following functions:

```
function plus_one(x) {
    return x + 1;
}

function trans(f) {
    return x => 2 * f(2 * x);
}

function twice(f) {
    return x => f(f(x));
}
```

2 * plus-one (2 * x)

plus-one (plus-one (x))

⇒ 2*x + 1 + 1

What are the results of the following Source programs (no need for explanation)?

A. [1 mark]

```
trans(plus_one)(1);
```

6

B. [2 marks]

```
trans(twice(plus_one))(1);
```

8

C. [2 marks]

x ⇒ trans(plus-one) (trans (plus-one) (x))

6

```
twice(trans(plus_one))(1);
```

26.

D. [2 marks]

x ⇒ trans(trans(x))

x ⇒ 2 * trans ((2 * x))

```
twice(trans)(plus_one)(1);
```

12 20 x ⇒ 2 · 2 · f (2 · x).

plus-one ?

f ⇒ x ⇒ trans (trans(f)(x)) (+1)) (1))

trans (2(po (2x))))

a

2 (a (2~~x~~)) (1)

2(2(po(2(2)))))

10

⇒ 20

Question 4: Binary Search Tree [15 marks]

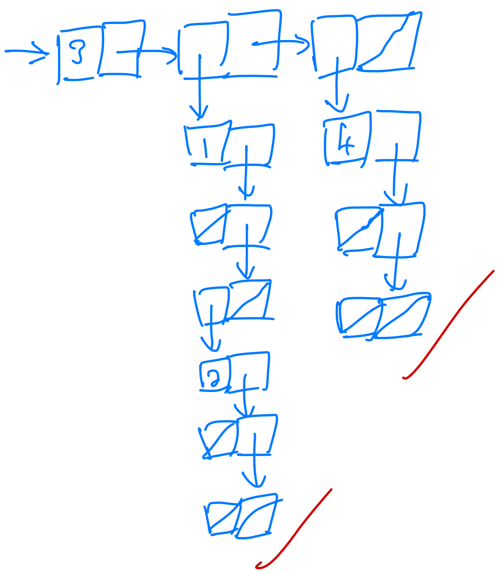
[This question was posed to students with NO prior exposure to the abstraction of binary search trees. It is not the purpose of the question to properly introduce the abstraction.]

A *binary search tree* (BST) is either null or a list with three elements: a number x, a left BST, and a right BST, where every number in the left BST is smaller than the number x, and every number in the right BST is larger than the number x.

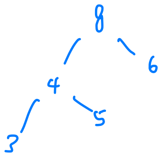
A. [3 mark]

Draw the box-and-pointer diagram for the following list, which represents a BST containing the numbers 1, 2, 3 and 4.

```
list(3,list(1, null, list(2,null,null)), list(4,null,null));
```



Test ([|] , , ,)



B. [3 marks]

Write a Source function `BST_min` that takes a BST as its only parameter and returns the smallest number in the BST. If the BST is an empty tree, the function returns `Infinity`.

```
function BST_min(bst) {
    const left_tree = bst => head(head(bst));
    const value = bst => head(tail(head(bst)));
    const right_tree = bst => tail(tail(head(bst)));
    if (is_null(bst)) {
        return Infinity;
    } else if (is_null(left_tree(bst))) {
        return value;
    } else {
        return BST_min(left_tree(bst));
    }
}
```

not sure if correct?

C. [4 marks]

[3 marks] Write a Source function `BST_find` that takes a number as its first parameter and a BST as the second parameter, and returns `true` if the number is in the BST, otherwise it returns `false`.

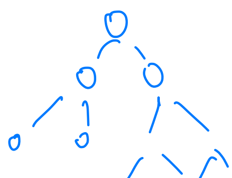
```
function BST_find(x, bst) {
    const left_tree = bst => head(head(bst));
    const value = bst => head(tail(head(bst)));
    const right_tree = bst => tail(tail(head(bst)));

    if (is_null(bst)) {
        return false;
    } else if (x === value(bst)) {
        return true;
    } else {
        return BST_find(left_tree(bst)) || BST_find(right_tree(bst));
    }
}
```

not sure if correct.

$T(n) = O(1) + 2(\frac{n}{3})$

logn



$2^{n-1} = k$

$\log_2 n = k+1$

$\frac{1(1-x^n)}{1-x}$ 2

1 2 4 8

[1 mark] Let the number of numbers in the BST be *n*. Assuming that the numbers are evenly distributed to every pair of left and right BSTs, in the worst case, what would be the order of growth of runtime of `BST_find`, in Θ notation, with respect to *n*? (no need for explanation)

Θ(log n)

D. [5 marks]

[4 marks] Write a Source function `BST_to_list` that takes a BST as its only parameter and returns a list that contains only the numbers in the BST in increasing order (i.e. the smallest number appears in the front of the list). For example, applying `BST_to_list` to the BST in Part A produces the list `list(1, 2, 3, 4)`.

```
function BST_to_list(bst) {
    const left_tree = bst => head(head(bst));
    const value = bst => head(tail(head(bst)));
    const right_tree = bst => tail(tail(head(bst)));

    return is_null(bst)
        ? null
        : append(BST_to_list(left_tree(BST)),
            append(list(value(BST)),
                BST_to_list(right_tree(BST))));
}
```

not sure if correct.

on we parse

$O(n) + 2T(\frac{n}{2})$

$n \log n$

[1 mark] Let the number of numbers in the BST be *n*. Assuming that the numbers are evenly distributed to every pair of left and right BSTs, what would be the order of growth of runtime of `BST_to_list`, in Θ notation, with respect to *n*? (no need for explanation)

Θ(logn) Θ(n logn)

Question 5: Putting Them In Order [11 marks]

In this exercise, we want to sort a list of *unique* numbers in increasing order using the ranks of the numbers in the list. The rank of a number in a list is the number of numbers in the list that are equal or smaller than it. Therefore, the smallest number in the list has the rank 1, the second-smallest number rank 2, and so on.

A. [7 marks]

[6 marks] Write a Source function `find_ranks` that takes a list of *unique* numbers as its only parameter and returns a list that contains the ranks of the corresponding numbers in the input list. For example, `find_ranks(list(9, 8, 5, 6))` should return the list `list(4, 3, 1, 2)`. You get 6 marks if you use at least one of the functions `filter`, `map` and `accumulate` in a correct and meaningful way, and 4 marks for any other correct solution.

```
function find_ranks(lst) {
    function rank(x){
        return length(filter( y => y >= x, lst));
    }

    return map( x => rank(x), lst);
}
```

[1 mark] Let n be the length of the input list. What is the order of growth of runtime of `find_ranks`, in Θ notation, with respect to n? (no need for explanation)

$\Theta(n^2)$

B. [4 marks]

Consider the following function:

```
function get_num(lst, ranks) {
    return rank =>
        head(ranks) === rank
        ? head(lst)
        : get_num(tail(lst), tail(ranks))(rank);
}
```

The function call `get_num(lst, ranks)(rank)` takes in `lst` (a list of unique numbers), and `ranks` (the list of ranks of the corresponding numbers in `lst`) and returns the number in `lst` that has the rank equal to `rank`.

[3 marks] Complete the following Source function `rank_sort`, that takes in a list of unique numbers as its only parameter and returns a list with the numbers sorted in increasing order. You must make use of the function `map`, the function `enum_list` (see Appendix) and the above `get_num` function in a correct and meaningful way.

```
function rank_sort(lst) {

    const ranks = find_ranks(lst);

    return map( x => get_num( lst, ranks (x)), enum_list( 0, length( lst) - 1));

}
```

?

[1 mark] Let n be the length of the input list. What is the order of growth of runtime of `rank_sort`, in Θ notation , with respect to n? (no need for explanation)

$\Theta(n^2)$

Question 6: Life Skill: Writing a Cheque [17 marks]

When you use a paper-based bank cheque, you need to write out the amount in English. This exercise is about writing this number properly, without any mistakes.

For example, the number 1234 is written as "one thousand two hundred thirty four" or "one thousand two hundred and thirty four." In this exercise, we will use the version without the word "and."

You are to complete a Source program that takes a *positive* whole number and produces its English form.

The followings are some of the functions of the program:

```
function singles_to_english(d) {
    return list_ref(list("", "one", "two", "three",
                         "four", "five", "six",
                         "seven", "eight", "nine"), d);
}

function tens_to_english(t) {
    return list_ref(list("", "ten", "twenty", "thirty",
                         "forty", "fifty", "sixty",
                         "seventy", "eighty", "ninety"), t);
}

function ten_to_nineteen_to_english(s) {
    return list_ref(list("ten", "eleven", "twelve", "thirteen",
                         "fourteen", "fifteen", "sixteen",
                         "seventeen", "eighteen",
                         "nineteen"), s);
}

function power_of_thousand(n) {
    return list_ref(list("", "thousand", "million", "billion",
                         "trillion", "quadrillion",
                         "quintillion"), n);
}

// for each triplet of hundred/ten/single
function triplet_to_english(h, t, s) {
    const he = h > 0 ? singles_to_english(h) + " hundred" : "";
    const te = t > 0 ? tens_to_english(t) : "";
    const se = s > 0 ? singles_to_english(s) : "";
    const tese = te === "" ? se
                 : t === 1 ? ten_to_nineteen_to_english(s)
                   : se === "" ? te : te + " " + se;
    return he === "" ? tese
           : tese === "" ? he
             : he + " " + tese;
}
```

12340 / 10
1234 / 10
123 / 10
12 / 10
1

A. [3 marks]

Write a Source function `number_to_digits` that takes a positive whole number as its only parameter and returns a list that contains the digits of the number, where the least significant digit is at the front of the result list. For example, `number_to_digits(12340)` should return the list `list(0, 4, 3, 2, 1)`.

```
function number_to_digits(n) {

    return reverse( map( x => parse_int( char_at( stringify(n), x), 10),
                         enum_list(0, math_ceil(math_log10(n)) - 1)));

    return n === 0
           ? null
           : pair(n % 10, number_to_digits( math_floor(n/10)));
}
```

B. [5 marks]

Write a Source function `triplets` that takes a list of digits as its only parameter and returns a list of triplets, where each triplet is a list of three consecutive digits in the input list. If the input list is not enough to fill the last triplet, the triplet will be padded with 0. For example, `triplets(list(3, 2, 4, 1))` should return the list `list(list(3, 2, 4),list(1, 0, 0))`. Note that the order of the triplets and the order of the digits within each triplet must be maintained.

```
function triplets(digits) {

        function create_triplets(digits) {
            return length(digits) < 2
                   ? list(head(digits), 0, 0)
                   : length(digits) < 3
                   ? list(head(digits), head(tail(digits)), 0)
                   : map(x => list_ref(digits, x), enum_list(0, 2));
        }

        return is_null(digits)
               ? null
               : pair(create_triplets(digits),
                      triplets(tail(tail(tail(digits)))));
}
```

C. [7 marks]

Write a Source function `triplets_to_english` that takes a list of triplets as its only parameter, where the list has at most seven triplets, and returns a string that contains the English form of the number represented by the input list of triplets. For example, for the number `12340`, the list of triplets is `list(list(0, 4, 3), list(2, 1, 0))`, and `triplets_to_english` should return the string `"twelve thousand three hundred forty"`. Your function can call the functions `triplet_to_english` and `power_of_thousand`.

hundreds

tens

```
function triplets_to_english(triplets) {
    const power_number = length(triplets)

    function helper(triplets, count) {
        const triplet = list_ref(triplets, count);
        return count === 0
            ? triplet_to_english(list_ref(triplet, 2),
                                 list_ref(triplet, 1),
                                 list_ref(triplet, 0))
            : triplet_to_english(list_ref(triplet, 2),
                                 list_ref(triplet, 1),
                                 list_ref(triplet, 0))
              + power_of_thousand(count)
              + helper(triplets, count - 1);
    }
    return helper(triplets, power_number);
}
```

answer correct

D. [2 marks]

Write a Source function `number_to_english` that takes a positive whole number less than 10^{21} as its only parameter and returns a string that contains the English form of the number.

```
function number_to_english(n) {
    return triplets_to_english(triplets(number_to_digits(n)));
}
```

—— END OF QUESTIONS ——

Appendix

List Support

Source supports the following list processing functions:

- `pair(x, y)`: Makes a pair from `x` and `y`.
- `is_pair(x)`: Returns `true` if `x` is a pair and `false` otherwise.
- `head(x)`: Returns the head (first component) of the pair `x`.
- `tail(x)`: Returns the tail (second component) of the pair `x`.
- `is_null(xs)`: Returns `true` if `xs` is null and `false` if `xs` is a pair.
- `is_list(x)`: Returns `true` if `x` is a list as defined in the lectures, and `false` otherwise. Iterative process; time: $O(n)$, space: $O(1)$, where n is the length of the chain of `tail` operations that can be applied to `x`.
- `list(x1, x2,..., xn)`: Returns a list with `n` elements. The first element is `x1`, the second `x2`, etc.
- `length(xs)`: Returns the length of the list `xs`. Iterative process; time: $O(n)$, space: $O(1)$, where n is the length of `xs`.
- `map(f, xs)`: Returns a list that results from list `xs` by element-wise application of `f`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`.
- `build_list(n, f)`: Makes a list with `n` elements by applying the unary function `f` to the numbers 0 to `n - 1`. Recursive process; time: $O(n)$, space: $O(n)$.
- `for_each(f, xs)`: Applies `f` to every element of the list `xs`, and then returns `true`. Iterative process; time: $O(n)$, space: $O(1)$, where n is the length of `xs`.
- `list_to_string(xs)`: Returns a string that represents list `xs` using the box-and-pointer notation `[...]`.
- `reverse(xs)`: Returns list `xs` in reverse order. Iterative process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`. The process is iterative, but consumes space $O(n)$ because of the result list.
- `append(xs, ys)`: Returns a list that results from appending the list `ys` to the list `xs`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`.
- `member(x, xs)`: Returns first postfix sublist whose head is identical to `x` (===); returns `null` if the element does not occur in the list. Iterative process; time: $O(n)$, space: $O(1)$, where n is the length of `xs`.

- `remove(x, xs)`: Returns a list that results from `xs` by removing the first item from `xs` that is identical (===) to `x`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`.

- `remove_all(x, xs)`: Returns a list that results from `xs` by removing all items from `xs` that are identical (===) to `x`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`.

- `filter(pred, xs)`: Returns a list that contains only those elements for which the one argument function `pred` returns `true`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`.

- `enum_list(start, end)`: Returns a list that enumerates numbers starting from `start` using a step size of 1, until the number exceeds (>) `end`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`. For example, `enum_list(2, 5)` returns the list `list(2, 3, 4, 5)`.

- `list_ref(xs, n)`: Returns the element of list `xs` at position `n`, where the first element has index 0. Iterative process; time: $O(n)$, space: $O(1)$, where n is the length of `xs`.

- `accumulate(op, initial, xs)`: Applies binary function `op` to the elements of `xs` from right-to-left order, first applying `op` to the last element and the value `initial`, resulting in r_1, then to the second-last element and r_1, resulting in r_2, etc., and finally to the first element and r_{n-1}, where n is the length of the list. Thus, `accumulate(op, zero, list(1,2,3))` results in `op(1, op(2, op(3, zero)))`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`, assuming `op` takes constant time.

Miscellaneous Functions

- `is_number(x)`: Returns `true` if `x` is a number, and `false` otherwise.

- `equal(x, y)`: Returns `true` if `x` and `y` have the same structure (using pairs and `null`), and corresponding leaves are ===, and `false` otherwise.

(Scratch Paper: Tear off, if needed.)

(Scratch Paper: Tear off, if needed.)

(Scratch Paper: Tear off, if needed.)

(Scratch Paper: Tear off, if needed.)

NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING

MIDTERM QUIZ

ADAPTED TO SOURCE 2021 IN 9/2020

Semester 1 AY2014/2015

CS1101S — PROGRAMMING METHODOLOGY

1 October 2014 Time Allowed: **1 Hour 40 Minutes**

ANSWER

Matriculation No.: | | | | | | | | | |

Instructions (please read carefully):

1. Write down your **matriculation number** on the **question paper**. DO NOT WRITE YOUR NAME ON THE QUESTION SET!
2. This is an **open-sheet quiz**. You are allowed to bring one A4 sheet of notes (written or printed on both sides).
3. This paper comprises **6 questions** and **TWENTY (20)** printed pages. The time allowed for solving this quiz is **1 hour 40 minutes**.
4. The maximum score of this quiz is **60 marks**. The weight of each question is given in square brackets beside the question number.
5. All questions must be answered correctly for the maximum score to be attained.
6. All questions must be answered in the space provided in the question paper; no extra sheets will be accepted as answers.
7. The pages marked "scratch paper" in the question paper may be used as scratch paper.
8. You are allowed to use pencils or pens, as you like (no red color, please).
9. Write legibly; **UNTIDINESS will be penalized**.

GOOD LUCK!

Q#	**1**	**2**	**3**	**4**	**5**	**6**	**Σ**
MAX	5	5	7	15	11	17	60
SC							

Question 1: Making Logical Sense [5 marks]

Rewrite the following Source expressions without using any of the logical operators `&&`, `||`, and `!`. The result of evaluating your expression must be exactly the same as that of the original, regardless of the values of `cond_a`, `cond_b`, `do_x` and `do_y`.

A. [1 mark]

```
!cond_a ? do_x() : do_y()
```

```
cond_a ? do_y() : do_x()
```

B. [2 marks]

```
(cond_a || cond_b) ? do_x() : do_y()
```

```
cond_a ? do_x()
       : cond_b
         ? do_x()
         : do_y()
```

C. [2 marks]

```
(cond_a && cond_b) ? do_x() : do_y()
```

```
cond_a ? (cond_b ? do_x() : do_y())
       : do_y()
```

Question 2: Almost Malfunction [5 marks]

A. [4 marks]

Consider a restricted form of Source in which functions are allowed to have at most one parameter. Rewrite the following function definition under this restriction. You must not use lists or any structure created using `pair`.

```
function myfunc(a, b, c) {
    return a * b + c;
}
```

```
function myfunc(a) {
    return b => c => a * b + c;
}
```

B. [1 mark]

With your new function definition in Part A, how would you rewrite the function call `myfunc(3, 2, 1);`?

```
myfunc(3)(2)(1);
```

Question 3: Function Composition Reincarnated [7 marks]

Considered the following functions:

```
function plus_one(x) {
    return x + 1;
}

function trans(f) {
    return x => 2 * f(2 * x);
}

function twice(f) {
    return x => f(f(x));
}
```

What are the results of the following Source programs (no need for explanation)?

A. [1 mark]

```
trans(plus_one)(1);
```

6

B. [2 marks]

```
trans(twice(plus_one))(1);
```

8

C. [2 marks]

```
twice(trans(plus_one))(1);
```

26

D. [2 marks]

```
twice(trans)(plus_one)(1);
```

20

Question 4: Binary Search Tree [15 marks]

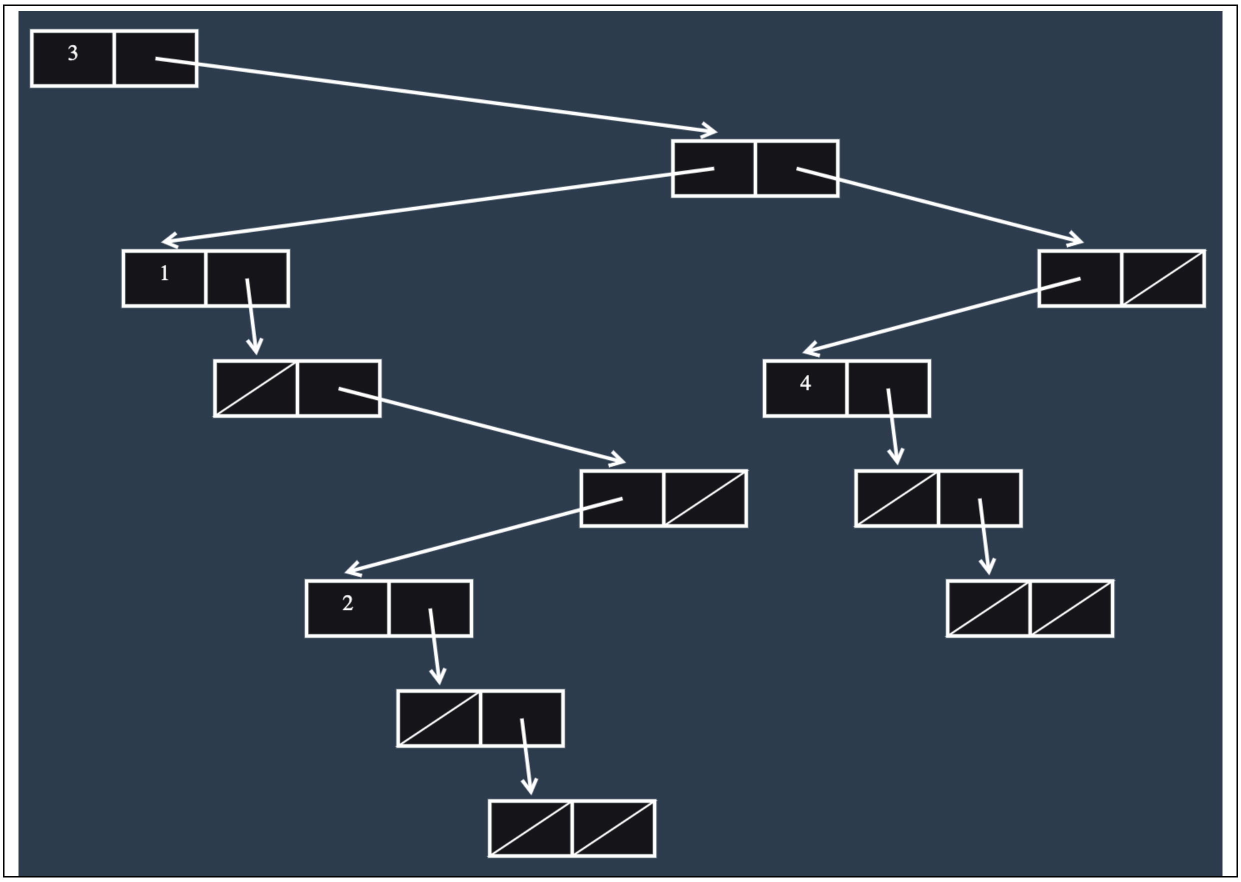
[This question was posed to students with NO prior exposure to the abstraction of binary search trees. It is not the purpose of the question to properly introduce the abstraction.]

A *binary search tree* (BST) is either null or a list with three elements: a number *x*, a left BST, and a right BST, where every number in the left BST is smaller than the number *x*, and every number in the right BST is larger than the number *x*.

A. [3 mark]

Draw the box-and-pointer diagram for the following list, which represents a BST containing the numbers 1, 2, 3 and 4.

```
list(3,list(1, null, list(2,null,null)), list(4,null,null));
```



B. [3 marks]

Write a Source function `BST_min` that takes a BST as its only parameter and returns the smallest number in the BST. If the BST is an empty tree, the function returns `Infinity`.

```
function BST_min(bst) {

    return is_null(bst)
        ? Infinity
        : is_null(head(tail(bst)))
          ? head(bst)
          : BST_min(head(tail(bst)));

}
```

C. [4 marks]

[3 marks] Write a Source function `BST_find` that takes a number as its first parameter and a BST as the second parameter, and returns `true` if the number is in the BST, otherwise it returns `false`.

```
function BST_find(x, bst) {

    return is_null(bst)
        ? false
        : x === head(bst)
          ? true
          : x < head(bst)
            ? BST_find(x, head(tail(bst)))
            : BST_find(x, head(tail(tail(bst))));

}
```

[1 mark] Let the number of numbers in the BST be n. Assuming that the numbers are evenly distributed to every pair of left and right BSTs, in the worst case, what would be the order of growth of runtime of `BST_find`, in Θ notation, with respect to n? (no need for explanation)

$\Theta(\log n)$. Such a "balanced" BST has approximately $\log_2 n$ levels. `BST_find` visits one number per level, from the top level to the bottom.

D. [5 marks]

[4 marks] Write a Source function `BST_to_list` that takes a BST as its only parameter and returns a list that contains only the numbers in the BST in increasing order (i.e. the smallest number appears in the front of the list). For example, applying `BST_to_list` to the BST in Part A produces the list `list(1, 2, 3, 4)`.

```
function BST_to_list(bst) {

    if (is_null(bst)) {
        return null;
    } else {
        const ltree = head(tail(bst));
        const num = head(bst);
        const rtree = head(tail(tail(bst)));
        return append(BST_to_list(ltree),
                      pair(num, BST_to_list(rtree)));
    }

}
```

[1 mark] Let the number of numbers in the BST be n. Assuming that the numbers are evenly distributed to every pair of left and right BSTs, what would be the order of growth of runtime of `BST_to_list`, in Θ notation, with respect to n? (no need for explanation)

$\Theta(n \log n)$. For a "balanced" BST, there are approximately $\log_2 n$ levels. At each level the *total* runtime of `append` has order of growth $\Theta(n)$.

Question 5: Putting Them In Order [11 marks]

In this exercise, we want to sort a list of *unique* numbers in increasing order using the ranks of the numbers in the list. The rank of a number in a list is the number of numbers in the list that are equal or smaller than it. Therefore, the smallest number in the list has the rank 1, the second-smallest number rank 2, and so on.

A. [7 marks]

[6 marks] Write a Source function `find_ranks` that takes a list of *unique* numbers as its only parameter and returns a list that contains the ranks of the corresponding numbers in the input list. For example, `find_ranks(list(9, 8, 5, 6))` should return the list `list(4, 3, 1, 2)`. You get 6 marks if you use at least one of the functions `filter`, `map` and `accumulate` in a correct and meaningful way, and 4 marks for any other correct solution.

```
function find_ranks(lst) {

    return map(y => length(filter(x => x <= y, lst)),
               lst);

}
```

[1 mark] Let n be the length of the input list. What is the order of growth of runtime of `find_ranks`, in Θ notation, with respect to n? (no need for explanation)

$\Theta(n^2)$

B. [4 marks]

Consider the following function:

```
function get_num(lst, ranks) {
    return rank =>
        head(ranks) === rank
        ? head(lst)
        : get_num(tail(lst), tail(ranks))(rank);
}
```

The function call `get_num(lst, ranks)(rank)` takes in `lst` (a list of unique numbers), and `ranks` (the list of ranks of the corresponding numbers in `lst`) and returns the number in `lst` that has the rank equal to `rank`.

[3 marks] Complete the following Source function `rank_sort`, that takes in a list of unique numbers as its only parameter and returns a list with the numbers sorted in increasing order. You must make use of the function `map`, the function `enum_list` (see Appendix) and the above `get_num` function in a correct and meaningful way.

```
function rank_sort(lst) {

    const ranks = find_ranks(lst);

    const one2n = enum_list(1, length(lst));

    return map(get_num(lst, ranks), one2n);

}
```

[1 mark] Let n be the length of the input list. What is the order of growth of runtime of `rank_sort`, in Θ notation , with respect to n? (no need for explanation)

$\Theta(n^2)$

Question 6: Life Skill: Writing a Cheque [17 marks]

When you use a paper-based bank cheque, you need to write out the amount in English. This exercise is about writing this number properly, without any mistakes.

For example, the number 1234 is written as “one thousand two hundred thirty four” or “one thousand two hundred and thirty four”. In this exercise, we will use the version without the word “and”.

You are to complete a Source program that takes a *positive* whole number and produces its English form.

The followings are some of the functions of the program:

```
function singles_to_english(d) {
    return list_ref(list("", "one", "two", "three",
                         "four", "five", "six",
                         "seven", "eight", "nine"), d);
}

function tens_to_english(t) {
    return list_ref(list("", "ten", "twenty", "thirty",
                         "forty", "fifty", "sixty",
                         "seventy", "eighty", "ninety"), t);
}

function ten_to_nineteen_to_english(s) {
    return list_ref(list("ten", "eleven", "twelve", "thirteen",
                         "fourteen", "fifteen", "sixteen",
                         "seventeen", "eighteen",
                         "nineteen"), s);
}

function power_of_thousand(n) {
    return list_ref(list("", "thousand", "million", "billion",
                         "trillion", "quadrillion",
                         "quintillion"), n);
}

// for each triplet of hundred/ten/single
function triplet_to_english(h, t, s) {
    const he = h > 0 ? singles_to_english(h) + " hundred" : "";
    const te = t > 0 ? tens_to_english(t) : "";
    const se = s > 0 ? singles_to_english(s) : "";
    const tese = te === "" ? se
                 : t === 1 ? ten_to_nineteen_to_english(s)
                   : se === "" ? te : te + " " + se;
    return he === "" ? tese
           : tese === "" ? he
             : he + " " + tese;
}
```

A. [3 marks]

Write a Source function `number_to_digits` that takes a positive whole number as its only parameter and returns a list that contains the digits of the number, where the least significant digit is at the front of the result list. For example, `number_to_digits(12340)` should return the list `list(0, 4, 3, 2, 1)`.

```
function number_to_digits(n) {

    return n === 0
        ? null
        : pair(n % 10,
               number_to_digits(math_floor(n / 10)));


}
```

B. [5 marks]

Write a Source function `triplets` that takes a list of digits as its only parameter and returns a list of triplets, where each triplet is a list of three consecutive digits in the input list. If the input list is not enough to fill the last triplet, the triplet will be padded with 0. For example, `triplets(list(3, 2, 4, 1))` should return the list `list(list(3, 2, 4),list(1, 0, 0))`. Note that the order of the triplets and the order of the digits within each triplet must be maintained.

```
function triplets(digits) {

    return is_null(digits)
        ? null
        : is_null(tail(digits))
          ? list(list(head(digits), 0, 0))
          : is_null(tail(tail(digits)))
            ? list(list(head(digits),
                             head(tail(digits)), 0))
            : pair(list(head(digits),
                            head(tail(digits)),
                            head(tail(tail(digits)))),
                       triplets(tail(tail(tail(digits)))));

}
```

C. [7 marks]

Write a Source function `triplets_to_english` that takes a list of triplets as its only parameter, where the list has at most seven triplets, and returns a string that contains the English form of the number represented by the input list of triplets. For example, for the number `12340`, the list of triplets is `list(list(0, 4, 3), list(2, 1, 0))`, and `triplets_to_english` should return the string `"twelve thousand three hundred forty"`. Your function can call the functions `triplet_to_english` and `power_of_thousand`.

```
function triplets_to_english(triplets) {

    function ts2english(ts, p) {
        if (is_null(ts)) {
            return "";
        } else {
            const t2e = triplet_to_english(
                            head(tail(tail(head(ts)))),
                            head(tail(head(ts))),
                            head(head(ts)));
            const ts2e = ts2english(tail(ts), p + 1);
            return ts2e + (ts2e === "" || t2e === ""
                          ? "" : " ") +
                   t2e + (t2e === "" || p === 0
                         ? "" : " " + power_of_thousand(p));
        }
    }

    return ts2english(triplets, 0);
}
```

D. [2 marks]

Write a Source function `number_to_english` that takes a positive whole number less than 10^{21} as its only parameter and returns a string that contains the English form of the number.

```
function number_to_english(n) {

    return triplets_to_english(triplets(number_to_digits(n)));



}
```

For the full solution, use this URL:

https://sourceacademy.nus.edu.sg/playground#chap=2&exec=1000&ext=CURVES&prgrm=GYV
wdgxgLglg9mABAZxmA5gGwKblPpRx7YaYzIAWAFACYCUiA3gFCJuIBO2UIHSZyKHi7AqAqF
QBEkgDSJJCbLPIQA7nGWSoFLkpmt2ho8eOTgcXpuAwAbnvmoAHrIMm3bycmx2wm7DHQKK
E0wNCU6OXoAbmYAX2ZmUEhYBEQoEnxCYlJyaigGFkMuHj5EcWFsUXEpTXTfOS1VEigAT1qK
GA5Wl3de03MutobrYG6GpzHXPr7Pb2ah%2BX9AsflQsG42iLS6GPjE8Gh4JDqCIjXubBJT7Kxcq
mQCqeLefnIhETE3qTq-
HB9apqYOztTrpEg9abTMwWLqXeryEZg%2BGeGCOJEQyG9WY%2BdENJZBOEYzEec7orYPX
YJJKHVIABzgTQ4eDgwAIFAsyAAhmAaFQwl8itwXmU3hUql9pA1tBzuTRNABbGCYMgITQAIyV
KvqU2JpigHE1R00AEcQJyaAblUb9Lr3JJTWhYFbVVt%2BZT9skjmkDbScEIsiRbpQqBQ5FA5A8
mFMIAhBIgKNhEABeeOIAB8iAADIgAPwoNBYXDXQMCagUBgAank8fAFuwcsQAC55JIYoYY2A
4%2BIk2l01nc2kMsWcsH8k2W232B2414e8g%2B9m86hSEWAyPqJHm9JJ2xp1BB7OU92kye
WwPZzrps39yeUwBGAcnLKkuHDoMbhiXyHNw%2Bn6SPxNr0TKtJGrKsvB3TghVKBNkz-
UC83SC8%2BiA38U3-PNYK-
Yxm1gkCwlPbB3WpFIkDAEB5TVbAmSyGgAhgKBkD5AV2GeUokFvfs83I5VsPYZtaU5To%2BU
QABSRA70zG1bXIyjqOuOj0AYpj5U5bQ8GATA4DgDgRIAegkzM6GM4iDlI70YF9bgmMU5SWLY
NikHIPAeMwWh6MYhhuJAXi3GbZzXKoKAhLc2zPLoPiB2qaoE3NdyIM8uQpKzYzEEi-
z8EC4KlSCkL4rs4zIrYPNoq%2BWLeTCh45HK3Kcsq4yksKq9EEE4SYuwOL6uk20jBq7LQo8h4Ii
KyE%2Bry-
r8vChqRpMfVLL9JiJqWvL6uMky4ipMyvTmqzGLfUsgp9Bb7MQEjtuQAAmEs7kYuRaQKNK3Bg
YBECoAKfLc8Ko2mRyJ2w2JEGwTBZ0YR6%2Bj3NIrp7Ha-
X2u4xpyia%2BqG6aeuJBG3ORtburRvoaqx9bpghxioaPS7rtHcaqpaxAqzvHYRt%2BkngNe5m4
PQ0CAB9OchxNOP-
GbMTzf8t3kSswdtKAoarILSfgxBuZpzjs0F6ZhdA0XQKrelGWZVlpRALkeSoe6GbceJDAt1joOOcn
10O%2BbrMa0zPVSWSqJoogKeoflvutkpjiO6y4dHIPGL5CiPYUwbmPWvZ3fktd3yoO8LskzNMw
AZjTjOjJiBIEiAA

— END OF QUESTIONS —

Appendix

List Support

Source supports the following list processing functions:

- `pair(x, y)`: Makes a pair from `x` and `y`.
- `is_pair(x)`: Returns `true` if `x` is a pair and `false` otherwise.
- `head(x)`: Returns the head (first component) of the pair `x`.
- `tail(x)`: Returns the tail (second component) of the pair `x`.
- `is_null(xs)`: Returns `true` if `xs` is null, and `false` if `xs` is a pair.
- `is_list(x)`: Returns `true` if `x` is a list as defined in the lectures, and `false` otherwise. Iterative process; time: $O(n)$, space: $O(1)$, where n is the length of the chain of `tail` operations that can be applied to `x`.
- `list(x1, x2,..., xn)`: Returns a list with `n` elements. The first element is `x1`, the second `x2`, etc.
- `length(xs)`: Returns the length of the list `xs`. Iterative process; time: $O(n)$, space: $O(1)$, where n is the length of `xs`.
- `map(f, xs)`: Returns a list that results from list `xs` by element-wise application of `f`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`.
- `build_list(n, f)`: Makes a list with `n` elements by applying the unary function `f` to the numbers 0 to `n - 1`. Recursive process; time: $O(n)$, space: $O(n)$.
- `for_each(f, xs)`: Applies `f` to every element of the list `xs`, and then returns `true`. Iterative process; time: $O(n)$, space: $O(1)$, where n is the length of `xs`.
- `list_to_string(xs)`: Returns a string that represents list `xs` using the box-and-pointer notation `[...]`.
- `reverse(xs)`: Returns list `xs` in reverse order. Iterative process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`. The process is iterative, but consumes space $O(n)$ because of the result list.
- `append(xs, ys)`: Returns a list that results from appending the list `ys` to the list `xs`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`.
- `member(x, xs)`: Returns first postfix sublist whose head is identical to `x` (===); returns `null` if the element does not occur in the list. Iterative process; time: $O(n)$, space: $O(1)$, where n is the length of `xs`.

- `remove(x, xs)`: Returns a list that results from `xs` by removing the first item from `xs` that is identical (===) to `x`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`.

- `remove_all(x, xs)`: Returns a list that results from `xs` by removing all items from `xs` that are identical (===) to `x`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`.

- `filter(pred, xs)`: Returns a list that contains only those elements for which the one argument function `pred` returns `true`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`.

- `enum_list(start, end)`: Returns a list that enumerates numbers starting from `start` using a step size of 1, until the number exceeds (>) `end`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`. For example, `enum_list(2, 5)` returns the list `list(2, 3, 4, 5)`.

- `list_ref(xs, n)`: Returns the element of list `xs` at position `n`, where the first element has index 0. Iterative process; time: $O(n)$, space: $O(1)$, where n is the length of `xs`.

- `accumulate(op, initial, xs)`: Applies binary function `op` to the elements of `xs` from right-to-left order, first applying `op` to the last element and the value `initial`, resulting in r_1, then to the second-last element and r_1, resulting in r_2, etc., and finally to the first element and r_{n-1}, where n is the length of the list. Thus, `accumulate(op, zero, list(1,2,3))` results in `op(1, op(2, op(3, zero)))`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`, assuming `op` takes constant time.

Miscellaneous Functions

- `is_number(x)`: Returns `true` if `x` is a number, and `false` otherwise.

- `equal(x, y)`: Returns `true` if `x` and `y` have the same structure (using pairs and `null`), and corresponding leaves are ===, and `false` otherwise.

(Scratch Paper: Tear off, if needed.)

(Scratch Paper: Tear off, if needed.)

(Scratch Paper: Tear off, if needed.)

(Scratch Paper: Tear off, if needed.)