National University of Singapore
School of Computing
CS1101S: Programming Methodology
Semester I, 2021/2022

R10
Streams

Stream Processing Functions

In addition to the Stream Processing functions of Source §3, we further define the following functions and values:

```
function add_streams(s1, s2) {
    if (is_null(s1)) {
        return s2;
    } else if (is_null(s2)) {
        return s1;
    } else {
        return pair(head(s1) + head(s2),
                    () => add_streams(stream_tail(s1),
                                      stream_tail(s2)));
    }
}

function mul_streams(s1, s2) {
    if (is_null(s1)) {
        return s2;
    } else if (is_null(s2)) {
        return s1;
    } else {
        return pair(head(s1) * head(s2),
                    () => mul_streams(stream_tail(s1),
                                      stream_tail(s2)));
    }
}

function scale_stream(s, f) {
    return stream_map(x => x * f, s);
}

const integers = integers_from(1);
const ones = pair(1, () => ones);
```

Problems:

debug – remove stuff

1. In order to take a closer look at delayed evaluation, we will use the function `display` which displays its argument and returns it. What is displayed in response to evaluating the following sequence?

```
const x = stream_map(display, enum_stream(0, 10));
stream_ref(x, 3);
stream_ref(x, 5);
```

Consider the optimized version of `stream_map` covered in SICP JS, using the function `memo_fun` used in Lecture L10:

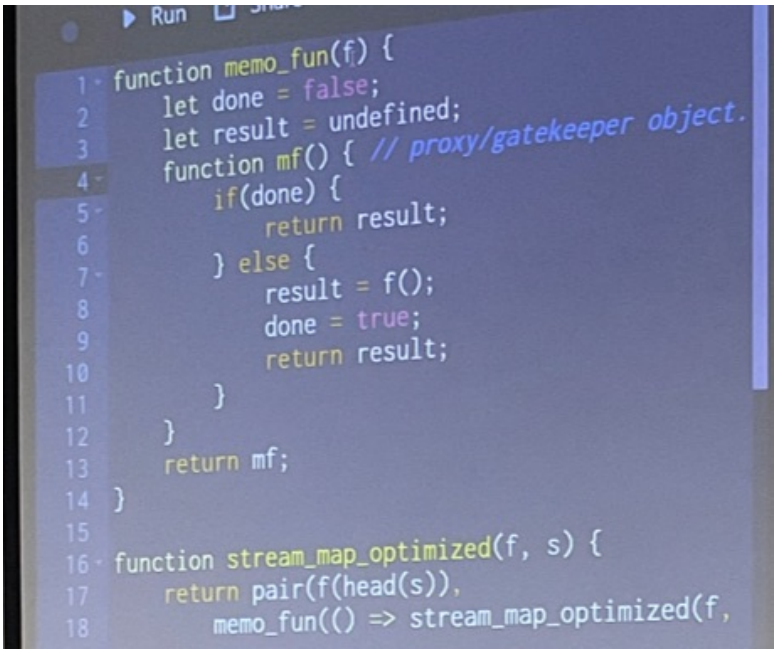
```
function stream_map_optimized(f, s) {
    return is_null(s)
           ? null
           : pair(f(head(s)),
                  memo_fun( () => stream_map_optimized(
                                      f, stream_tail(s)) ));
}
```

What will be displayed when evaluating the following sequence?

```
const x = stream_map_optimized(display, enum_stream(0, 10));
stream_ref(x, 3);
stream_ref(x, 5);
```

2. Consider the function `zip_stream` from Path P10 that takes as parameters two infinite streams and returns an infinite stream in which the elements of the given streams are interleaved. For example if s_1 is the infinite stream 1, 11, 111,... and s_2 is the infinite stream 2, 22, 222,... then `zip_stream`(s_1, s_2) returns the stream 1, 2, 11, 22, 111, 222,.... Declare the function `zip_list_of_streams` that takes a non-empty list of infinite streams as argument and zips up them all, in the given order. For example, if s_1 and s_2 are as above and s_3 is the infinite stream 3, 33, 333,..., then `zip(list(`s_1, s_2, s_3`))` should return the stream 1, 2, 3, 11, 22, 33,....

3. Consider the function `partial_sums` from Path P10 that takes as parameter an infinite stream of numbers `s` and returns the infinite stream whose elements are s_0, s_0+s_1, $s_0+s_1+s_2$, ... For example, `partial_sums(integers)` should be the stream 1, 3, 6, 10, 15, ... Declare the function `partial_sums` using the function `add_streams` above. Does your function also work on finite streams as argument? How about the empty stream?



Problems:

1. In order to take a closer look at delayed evaluation, we will use the function `display` which displays its argument and returns it. What is displayed in response to evaluating the following sequence?

```
const x = stream_map(display, enum_stream(0, 10));
stream_ref(x, 3);
stream_ref(x, 5);
```

Consider the optimized version of `stream_map` covered in SICP JS, using the function `memo_fun` used in Lecture L10:

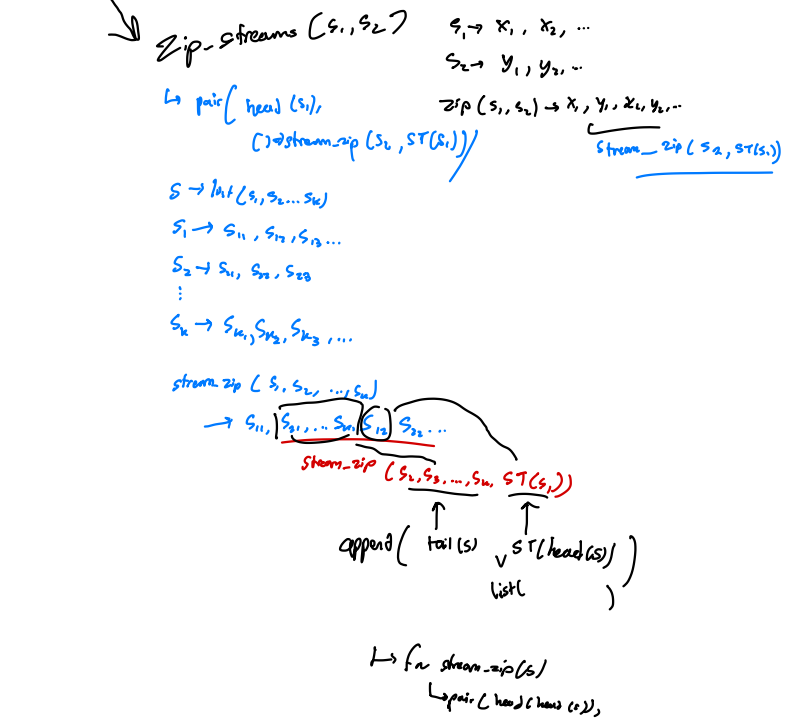
```
function stream_map_optimized(f, s) {
    return is_null(s)
           ? null
           : pair(f(head(s)),
                  memo_fun( () => stream_map_optimized(
                                      f, stream_tail(s)) ));
}
```

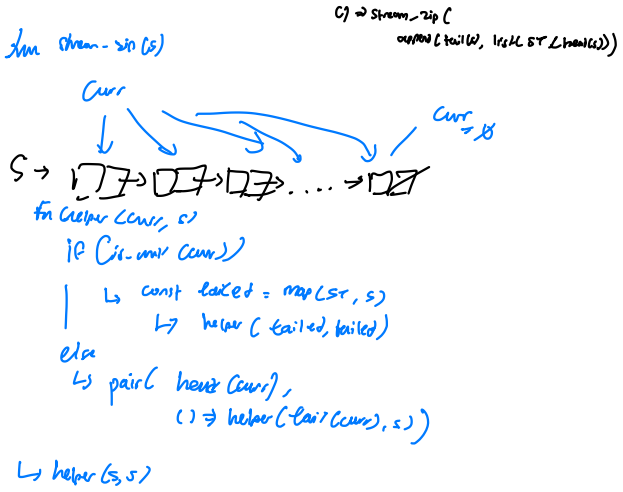
What will be displayed when evaluating the following sequence?

```
const x = stream_map_optimized(display, enum_stream(0, 10));
stream_ref(x, 3);
stream_ref(x, 5);
```

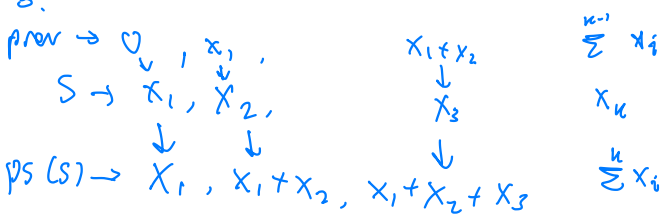
2. Consider the function `zip_stream` from Path P10 that takes as parameters two infinite streams and returns an infinite stream in which the elements of the given streams are interleaved. For example if s_1 is the infinite stream 1, 11, 111,... and s_2 is the infinite stream 2, 22, 222,... then `zip_stream`(s_1, s_2) returns the stream 1, 2, 11, 22, 111, 222,.... Declare the function `zip_list_of_streams` that takes a non-empty list of infinite streams as argument and zips up them all, in the given order. For example, if s_1 and s_2 are as above and s_3 is the infinite stream 3, 33, 333,..., then `zip(list(`s_1, s_2, s_3`))` should return the stream 1, 2, 3, 11, 22, 33,....

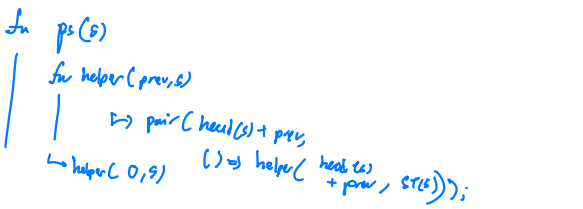
3. Consider the function `partial_sums` from Path P10 that takes as parameter an infinite stream of numbers `s` and returns the infinite stream whose elements are s_0, s_0+s_1, $s_0+s_1+s_2$, ... For example, `partial_sums(integers)` should be the stream 1, 3, 6, 10, 15, ... Declare the function `partial_sums` using the function `add_streams` above. Does your function also work on finite streams as argument? How about the empty stream?

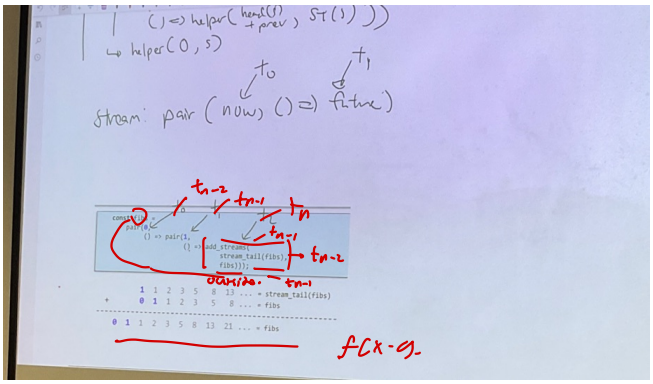



fn stream_zip(s)
curr
curr = 18
fn helper(curr, s)
if (is_empty curr))
const tailed = map(ST, s)
helper(tailed, tailed)
else
pair(head(curr),
() => helper(tail(curr), s))
helper(s, s)

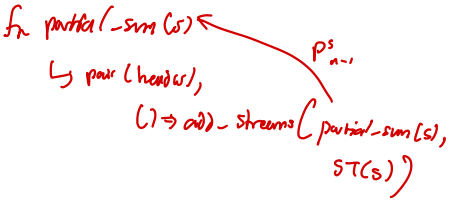
2.

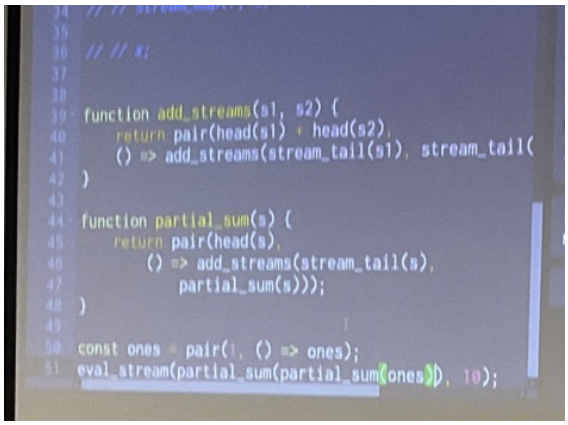

prev → 0 , x1 ,
S → x1 , x2 ,
PS(S) → x1 , x1+x2 , x1+x2+x3
x1+x2
x3
Σ xi (n-1)
xn
Σ xi (n)


fn ps(s)
fn helper(prev, s)
pair(head(s)+prev,
() => helper(head(s)+prev, ST(s)));
helper(0, s)


() => helper(head(s)+prev, ST(s)))
helper(0, s)
stream: pair(now, () => future)
t0
t1
tn-2
tn-1
tn
outside.
fCx-eg.

$P^s_n = P^s_{n-1} + S_n$


fn partial_sum(s)
pair(head(s),
() => add_streams(partial_sum(s),
ST(s)))
P^s_{n-1}


function add_streams(s1, s2) {
return pair(head(s1) + head(s2),
() => add_streams(stream_tail(s1), stream_tail(
}
function partial_sum(s) {
return pair(head(s),
() => add_streams(stream_tail(s),
partial_sum(s)));
}
const ones = pair(1, () => ones);
eval_stream(partial_sum(partial_sum(ones)), 10);