# 高阶函数

## Q1:Product

> 题目陈述：
> 写一个函数 `product(n, term)` ，输入一个整数 `n` 和一个函数 `term(x)` ，返回值为
>
> $$\prod_{i=1}^{n} \text{term}(i)$$
>
> 数据范围：
> $n \in \mathbb{N}_+$
>
> 输入输出：
> `#如以下代码块`

### 答案：

```
def product(n, term):
    """Return the product of the first n terms in a sequence.

    n: a positive integer
    term: a function that takes an index as input and produces a term

    >>> product(3, identity) # 1 * 2 * 3
    6
    >>> product(5, identity) # 1 * 2 * 3 * 4 * 5
    120
    >>> product(3, square) # 1^2 * 2^2 * 3^2
    36
    >>> product(5, square) # 1^2 * 2^2 * 3^2 * 4^2 * 5^2
    14400
    >>> product(3, increment) # (1+1) * (2+1) * (3+1)
    24
    >>> product(3, triple) # 1*3 * 2*3 * 3*3
    162
    """
    result = 1
    for i in range(1, n + 1):
        result *= term(i)
    return result
```

## Q2:Accumulate

### [💭 ]EXPANSION:

Q1中的 `product(n, term)` 函数可以被推广成一个更广泛的函数 `accumulate(fuse, start, n, term)`

> 题目陈述：
> 写一个函数 `accumulate(fuse, start, n, term)` ，
> 其中输入两个变量 `start` `n` ，两个函数 `fuse(a, b)` `term(x)`
>
> 数据范围：
>
> 输入输出：
> `#见代码块`

## 答案：

```
def accumulate(fuse, start, n, term):

    """Return the result of fusing together the first n terms in a
sequence
    and start. The terms to be fused are term(1), term(2), ..., term(n).
    The function fuse is a two-argument commutative & associative
function.

    >>> accumulate(add, 0, 5, identity) # 0 + 1 + 2 + 3 + 4 + 5
    15
    >>> accumulate(add, 11, 5, identity) # 11 + 1 + 2 + 3 + 4 + 5
    26
    >>> accumulate(add, 11, 0, identity) # 11 (fuse is never used)
    11
    >>> accumulate(add, 11, 3, square) # 11 + 1^2 + 2^2 + 3^2
    25
    >>> accumulate(mul, 2, 3, square) # 2 * 1^2 * 2^2 * 3^2
    72
    >>> # 2 + (1^2 + 1) + (2^2 + 1) + (3^2 + 1)
    >>> accumulate(lambda x, y: x + y + 1, 2, 3, square)
    19
    """

    result = start
    if n == 0:
        return start
    else:
        for i in range(1, n + 1):
            result = fuse(result, term(i))
    return result
```

### [🔨]IMPLEMENTATION

现在可以应用 [💭]**EXPANSION:** 里面写的函数 `accumulate(fuse, start, n, term)` 来重写两个函数：求和 `summation_using_accumulate` 和连乘 `product_using_accumulate`

而且仅需一行 `return`

## 答案：

```
def summation_using_accumulate(n, term):
    """Returns the sum: term(1) + ... + term(n), using accumulate.

    >>> summation_using_accumulate(5, square) # square(1) + square(2) +
... + square(4) + square(5)
    55
    >>> summation_using_accumulate(5, triple) # triple(1) + triple(2) +
... + triple(4) + triple(5)
    45
    >>> # This test checks that the body of the function is just a return
statement.
    >>> import inspect, ast
    >>> [type(x).__name__ for x in
ast.parse(inspect.getsource(summation_using_accumulate)).body[0].body]
    ['Expr', 'Return']
    """
    return accumulate(add, 0, n, term)



def product_using_accumulate(n, term):
    """Returns the product: term(1) * ... * term(n), using accumulate.

    >>> product_using_accumulate(4, square) # square(1) * square(2) *
square(3) * square(4)
    576
    >>> product_using_accumulate(6, triple) # triple(1) * triple(2) * ...
* triple(5) * triple(6)
    524880
    >>> # This test checks that the body of the function is just a return
statement.
    >>> import inspect, ast
    >>> [type(x).__name__ for x in
ast.parse(inspect.getsource(product_using_accumulate)).body[0].body]
    ['Expr', 'Return']
    """
    return accumulate(mul, 1, n, term)
```

## Q3:Make Repeater

> 问题陈述：
> 写一个函数 `implement(f, n)` 返回值为一个函数使得：
> `implement(f, n)(x)=f(f(f(...f(x)...))) #共n个f`
>
> 数据范围：
>
> 输入输出：
> `#见代码块`

**错误做法：**

```
def make_repeater(f, n):
    """Returns the function that computes the nth application of f.

    >>> add_three = make_repeater(increment, 3)
    >>> add_three(5)
    8
    >>> make_repeater(triple, 5)(1) # 3 * (3 * (3 * (3 * (3 * 1))))
    243
    >>> make_repeater(square, 2)(5) # square(square(5))
    625
    >>> make_repeater(square, 3)(5) # square(square(square(5)))
    390625
    """
    result = lambda x : f(x)
    i = 1
    while i < n:
        result = lambda x : f(result(x))
        i += 1
    return result
```

**[❗]报错信息如下：**

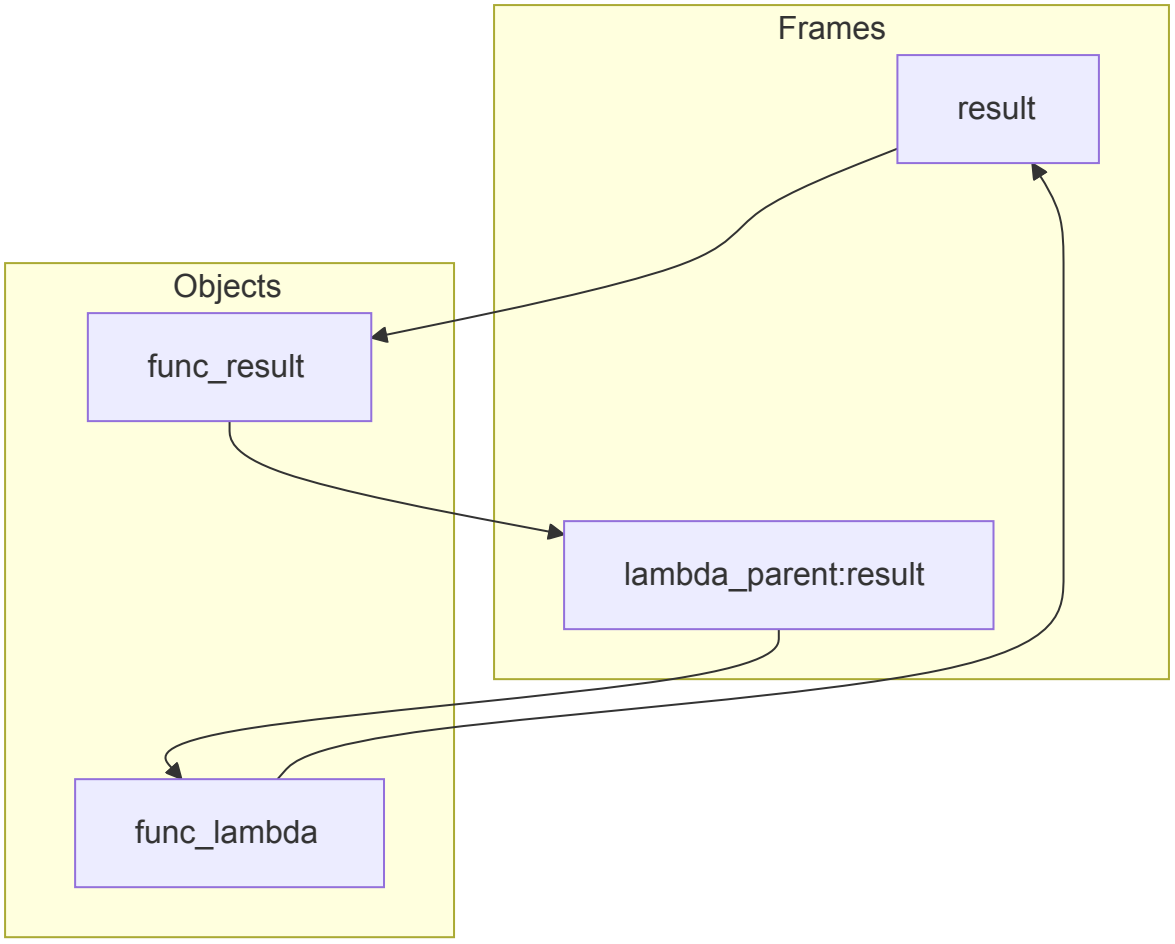
```
>>> add_three = make_repeater(increment, 3)
>>> add_three(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/caoteling/compScience/hw02/hw02.py", line 118, in <lambda>
    result = lambda x : f(result(x))
  File "/Users/caoteling/compScience/hw02/hw02.py", line 118, in <lambda>
    result = lambda x : f(result(x))
  File "/Users/caoteling/compScience/hw02/hw02.py", line 118, in <lambda>
    result = lambda x : f(result(x))
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
```

**[🤔] 原因:**

在 while 循环中，`lambda x : f(result(x))` 这个表达式并不会立刻“固化”当时 `result` 指向的函数。相反，它只记住了 `result` 这个**名字**。

当循环结束后，`result` 这个名字最终指向的是**最后一个**被定义的 `lambda`，即 `lambda x : f(result(x))`。

因此，当最终的函数被调用时，它执行 `f(result(x))`，此时它查找 `result` 这个名字，发现指向的正是它自己，从而导致了没有出口的无限递归。




**[➡️]用指针类比：**

> **[➡️]用指针类比：**
> 这相当于在 C 语言中创建了一个特殊的结构体，该结构体包含一个函数指针。
> 当定义这个结构体时，它内部的函数指针指向一个“调用自己”的函数。
> 当执行这个函数对象时，它会调用 `f`，而 `f` 的参数又是“调用自己”的结果，从而形成了一个没有出口的递归调用，最终导致栈溢出。

具体类比代码如下：

```
/*--Generated by Gemini 2.5 Pro--*/
#include <stdio.h>

// --- 定义我们的“函数对象”结构体 ---
// 它需要知道如何调用自己 (self_ptr)，以及要调用哪个外部函数 (f)
struct SelfCallingFunc {
```

```c
    // 函数指针：指向一个接收"自己"和"输入值x"的函数
    int (*execute)(struct SelfCallingFunc* self, int x);

    // 外部函数 f (比如 square)
    int (*f)(int);
};

// --- 这就是 lambda x: f(result(x)) 的具体实现 ---
// 它接收"自己"(self)和"输入值"(x)
int recursive_call(struct SelfCallingFunc* self, int x) {
    printf("进入递归...\n");
    // 调用 f，参数是“调用自己”的结果
    // self->execute(self, x) 就是在调用自己！
    return self->f(self->execute(self, x));
}

int square(int x) {
    return x * x;
}

int main() {
    // --- 构建这个死循环的 lambda 对象 ---
    struct SelfCallingFunc my_lambda;

    // 1. 把它要调用的外部函数 f 设置为 square
    my_lambda.f = square;

    // 2. 把它自己的执行逻辑设置为 recursive_call
    // 这就完成了自引用：这个对象知道如何调用自己
    my_lambda.execute = recursive_call;

    // --- 尝试调用 ---
    // 相当于 Python 里的 my_lambda(5)
    // 我们手动把 "自己" 作为第一个参数传进去
    printf("开始调用...\n");
    int result = my_lambda.execute(&my_lambda, 5);
    // 这行代码会无限打印"进入递归..."，直到栈溢出

    printf("Result: %d\n", result); // 永远走不到这里

    return 0;
}
```

## 答案：

```python
def make_repeater(f, n):
    """Returns the function that computes the nth application of f.

    >>> add_three = make_repeater(increment, 3)
```

```
    >>> add_three(5)
    8
    >>> make_repeater(triple, 5)(1) # 3 * (3 * (3 * (3 * (3 * 1))))
    243
    >>> make_repeater(square, 2)(5) # square(square(5))
    625
    >>> make_repeater(square, 3)(5) # square(square(square(5)))
    390625
    """
    assert n >= 0, 'n must be positive.'
    if n == 1:
        return lambda x : f(x)
    else:
        return lambda x : f(make_repeater(f, n - 1)(x))
```