

Quiz 5

Name: Gelik Köseoglu ID: 23400196 Section:

Consider the following two alternative ways of computing the factorial function:

```
int fact(int n) {
    if (n == 0)
        return 1;
    int r = n * fact(n - 1);
    return r;
}

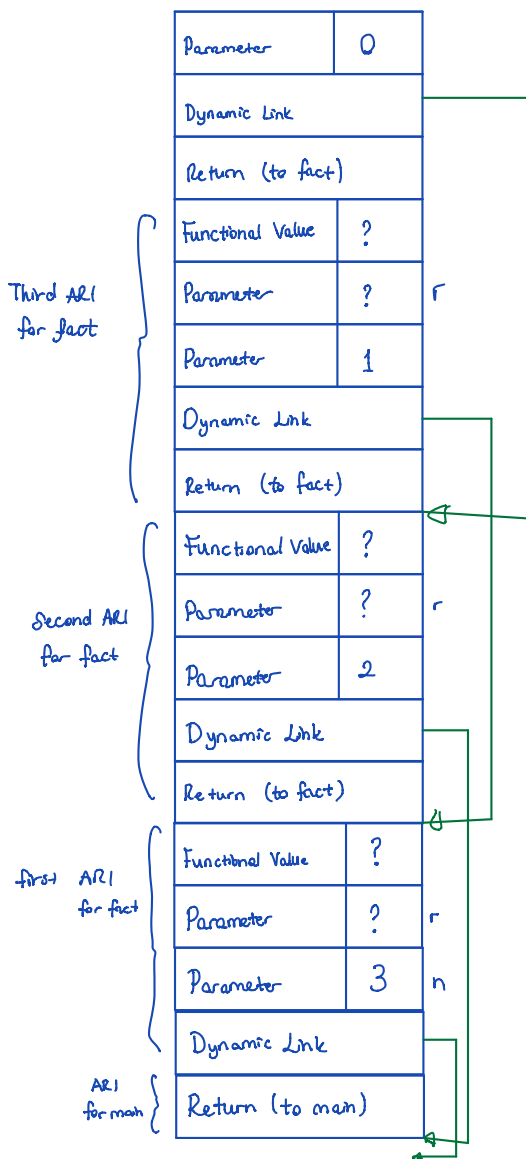
void main() {
    return fact(3);
}
```

```
int fact_i(int n, int s) {
    if (n == 1)
        return s;
    int ns = n * s;
    return fact_i(n - 1, ns);
}

int fact_t(int n) {
    if (n == 0)
        return 1;
    return fact_i(n, 1);
}

void main() {
    return fact_t(3);
}
```

a) (8pts) For `fact(3)` and for `fact_t(3)`, draw the contents of the stack, showing the *activation record instances*. Draw the stack at its largest extent.



b) (2pts) Think about how the function call stack will unwind in the two cases you draw in part a, that is for the `fact` and the `fact_i` functions. Write down the most fundamental difference you see. *Hint*: The second implementation is known as *tail recursive*.

On the non tail-recursive `fact`; we need to store the Activation RI. Because we rely on the calculations of another recursive call to return our result.

On the other hand, the return value of `fact_i`; we don't need to store the older ARs because we don't need the previous values to return the correct result.

c) (3pts) Put yourself into the shoes of a compiler implementer. Could you generate code for `fact_i` somewhat differently so that the result is more efficient? If so, how? *Hint*: Think about the overhead of creating activation record instances and a way to avoid it.

Tail-call optimization is where you're able to avoid allocating a new stack frame for a function because the calling function will simply return the value that it gets from the called function.

I would turn the `fact_i` function into

```
int fact_i (int n, int s)
while(true) {
    int num = s
    if (num ≤ 1)
        return n;
    n *= s
    s--;
}
```

d) (7pts) Implement summation as a recursive function that uses *tail recursion*.

Version that is not tail-recursive:

```
double sum(double* vals, int sz) {
    if (sz == 0)
        return 0;
    return *vals + sum(vals + 1, sz - 1);
}
```

→ sums elements of a `double[]`.

Tail Recursion: recursive call is the very last thing in a function.

```
double sum(double* vals, int sz, s=0) {
    if (sz == 0)
        return s;
    return sum(vals + 1, sz - 1, *vals + s);
}
```

→ optional c++ param

// `sum(vals, 5, 0);`