Recursividade na cauda

Observe a sequência aritmética a seguir e crie um programa para encontrar o valor do n-ésimo elemento: $\{2, 7, 12, 17, 22, ...\}$

Formulação recursiva
$f(1) = 2$
$f(n) = 5 + f(n-1)$

Observe com atenção o desdobramento da aplicação da formulação recursiva para o cálculo do quinto elemento da sequência.

Desdobramento
$f(5)$
$= 5 + f(4)$
$= 5 + (5 + f(3))$
$= 5 + (5 + (5 + f(2)))$
$= 5 + (5 + (5 + (5 + f(1))))$
$= 5 + (5 + (5 + (5 + 2)))$
$= 22$

Veja que a cada chamada recursiva, forma-se uma sequência acumulada de operações cujo resultado final permanece em suspenso até a última aplicação da função. Esse **acúmulo de valores exige espaço em memória** da máquina e pode eventualmente causar problemas de estouro de memória.

Uma **forma alternativa** de lidar com esse comportamento e **antecipar toda a computação parcial possível**, reduzindo a necessidade de armazenamento da valores em memória, **é acrescentar um parâmetro à função para agir como *acumulador***.

Formulação recursiva com acumulador
$f(n) = f(n, 2)$
$f(1, acc) = acc$
$f(n, acc) = f(n-1, 5 + acc)$

Desdobramento
$f(5) = f(5, 2)$
$= f(4, 5 + 2) \rightarrow f(4, 7)$
$= f(3, 5 + 7) \rightarrow f(3, 12)$
$= f(2, 5 + 12) \rightarrow f(2, 17)$
$= f(1, 5 + 17) \rightarrow f(1, 22)$
$= 22$

Esse tipo de abordagem é conhecida como **Recursividade na Cauda** porque a **chamada recursiva é a ÚLTIMA operação** realizada! Atente para a diferença de comportamento da 1^a abordagem que não usa "cauda": a última operação realizada não é a chamada recursiva mas, sim, o somatório pendente $5 + (5 + (5 + (5 + 2))) = 22$.

Implementação em Javascript

```
const f = (n) => {
    return fAux(n,2)
}

const fAux = (n,acc) => {
    if (n == 1) {return acc}
    else return fAux(n-1,5+acc)
}
```

Perceba que a implementação passa a considerar o nosso já conhecido uso de *funcão de interface*.

⚠️ **CURIOSIDADE**

Compiladores de muitas linguagens de programação (inclusive as tipicamente funcionais) conseguem otimizar bastante as operações e, assim, reduzir o tempo de execução dos programas recursivos quando detectam o uso de recursividade na cauda.

Em Javascript, porém, essa otimização não é tão garantida. Depende de versão do compilador, suporte do browser, etc. Não é objetivo da disciplina tratar de questões específicas a linguagens ou compiladores, mas apenas passar o conceito a ser aprendido. Quem tiver interesse em conhecer mais sobre essa otimização sugiro a busca na Internet pelo termo TAIL CALL OPTIMIZATION (TCO).