Recursividade

Recursão é o processo de definir algo em termos de si mesmo e é, algumas vezes, chamado de definição circular. Assim, pode-se dizer que o conceito de algo recursivo está dentro de si, que por sua vez está dentro de si e assim sucessivamente, infinitamente.

O exemplo a seguir define o ancestral de uma pessoa:
Os pais de uma pessoa são seus ancestrais (caso base);
Os pais de qualquer ancestral são também ancestrais da pessoa inicialmente considerada (passo recursivo).
Definições como estas são normalmente encontradas na matemática. O grande apelo que o conceito da recursão traz é a possibilidade de dar uma definição finita para um conjunto que pode ser infinito.
Um exemplo aritmético:

O primeiro número natural é zero.
O sucessor de um número natural é um número natural.

Na computação o conceito de recursividade é amplamente utilizado, mas difere da recursividade típica por apresentar uma condição que provoca o fim do ciclo recursivo. Essa condição deve existir, pois, devido às limitações técnicas que o computador apresenta, a recursividade é impedida de continuar eternamente.

Função para cálculo de Fatorial

Na linguagem C, as funções podem chamar a si mesmas. A função é recursiva se um comando no corpo da função chama a si própria. Para uma linguagem de computador ser recursiva, uma função deve poder chamar a si mesma. Um exemplo simples é a função fatorial, que calcula o fatorial de um inteiro. O fatorial de um número N é o produto de todos os números inteiros entre 1 e N.
Por exemplo, 3 fatorial (ou 3!) é 1 * 2 * 3 = 6.

A função abaixo apresenta uma versão iterativa para cálculo do fatorial de um número.

```
18 int fatorial_iterativo(int n){
19     int t, f;
20     f = 1;
21     for(t = 1; t <= n; t++){
22           f = f * t;
23     }
24     return f;
25 }
26
```

Programa Fatorial (versão iterativa)

Mas multiplicar n pelo produto de todos os inteiros a partir de n-1 até 1 resulta no produto de todos os inteiros de n a 1. Portanto, é possível dizer que fatorial:

- 0! = 1
- 1! = 1 * 0!
- 2! = 2 * 1!
- 3! = 3 * 2!
- 4! = 4 * 3!

Logo o fatorial de um número também pode ser definido recursivamente (ou por recorrência) através das seguintes regras (representação matemática) [1, 1]:

- n! = 1, se n = 0
- n! = n * (n-1)!, se n > 0

A função abaixo mostra a versão recursiva do programa fatorial.

```
18 int fatorial_recursivo(int n){
19     int t, f;
20     if (n == 0 || n == 1){ //condição de parada
21           return 1;
22     }
23     f = fatorial_recursivo(n-1) * n; //rechamada da função
24     return f;
25 }
26
```

Programa Fatorial (versão recursiva)

A versão não-recursiva de fatorial deve ser clara. Ela usa um laço que é executado de 1 a n e multiplica progressivamente cada número pelo produto móvel.

A operação de fatorial recursiva é um pouco mais complexa. Quando a função `fatorial_recursivo(n)` é chamada com um argumento de 1, a função devolve 1. Caso contrário, ela devolve o produto `fatorial_recursivo(n-1) * n`.

Para avaliar essa expressão, `fatorial_recursivo` é chamada com n-1. Isso acontece até que n se iguale a 1 e as chamadas à função comecem a retornar.

Calculando o fatorial de 2, a primeira chamada a `fatorial_recursivo(n)` provoca uma segunda chamada com o argumento 1. Essa chamada retorna 1, que é, então, multiplicado por 2 (o valor original e n). A resposta então é 2.

Para melhor entendimento, é interessante ver como o programa é executado internamente no computador. No caso do programa iterativo é necessário o uso de duas variáveis, f e t para armazenar os diversos passos do processamento. Por exemplo, ao calcular fatorial de 6, o computador vai passar sucessivamente pelos seguintes passos da tabela 1.

t	f
1	1
2	2
3	6
4	24
5	120
6	720

Tabela 1 - Cálculo de Fatorial de 6

No programa recursivo nada disso acontece. Para calcular o fatorial de 6, o computador tem primeiro que calcular o fatorial de 5 e só depois é que faz a multiplicação por 6 pelo resultado (120). Por sua vez para calcular p fatorial de 5, vai ter que calcular o fatorial de 4. Resumindo, aquilo que acontece internamente é uma expressão seguida de uma contração:

fatorial_recursivo(6)
6 * *fatorial_recursivo*(5)
6 * 5 * *fatorial_recursivo*(4)
6 * 5 * 4 * *fatorial_recursivo*(3)
6 * 5 * 4 * 3 * *fatorial_recursivo*(2)
6 * 5 * 4 * 3 * 2 * *fatorial_recursivo*(1)
6 * 5 * 4 * 3 * 2 * 1
6 * 5 * 4 * 3 * 2
6 * 5 * 4 * 6
6 * 5 * 24
6 * 120
720

Quando uma função chama a si mesma, novos parâmetros e variáveis locais são alocados na pilha, e o código da função é executado com essas novas variáveis. Uma chamada recursiva não faz uma nova cópia da função; apenas os argumentos são novos. Quando cada função recursiva retorna, as variáveis locais e os parâmetros são removidos da pilha e a execução recomeça do ponto da chamada à função dentro da função.

Números de Fibonacci

Fibonacci (matemático da Renascença Italiana) estabeleceu uma série curiosa de números para modelar o número de casais de coelhos em sucessivas gerações. Assumindo que nas primeiras gerações só existe um casal de coelhos, a sequência de Fibonacci é a sequência de inteiros: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

No trecho de programa abaixo na função `Fibonacci_iterativo()` é codificada com a versão iterativa para calcular o n-ésimo termo da sequência Fibonacci:

```
18 int fibonacci_iterativo(int n){
19     int l, h, x, i;
20     if (n <= 2){
21           return 1;
22     }
23     l = 0;
24     h = 1;
25     //Cálculo para o próximo número da sequência.
26     for (i = 2; i <= n; i++){
27         x = l;
28         l = h;
29         h = x + l;
30     }
31     return h;
32 }
```

Programa Fibonacci versão iterativa

O n-ésimo número da sequência é definido como sendo a soma dos dois números anteriores. Logo, fazendo a definição recursiva:

- `fibonacci(n) = n` se `n <= 2`;
- `fibonacci(n) = fibonacci(n-2) + fibonacci(n-1)` se `n > 2`.

A sua determinação recursiva impõe o cálculo direto do valor para dois elementos de base (a primeira e a segunda geração). No programa `fibonacci_recursivo` é mostrada a versão recursiva para calcular o n-ésimo termo da sequência Fibonacci.

```
18 int fibonacci_recursivo(int n){
19     if (n <= 2){
20        return 1;
21     }
22     //chama a si mesma 2 vezes!!
23     return fibonacci_recursivo(n-2) + fibonacci_recursivo(n-1);
24 }
```

Esta solução é muito mais simples de programar do que a versão iterativa. Contudo, esta versão é ineficiente, pois cada vez que a função `fibonacci_recursivo()` é chamada, a dimensão do problema reduz-se a apenas uma unidade (de n para n-1), mas são feitas duas chamadas recursivas. Isto dá origem a uma explosão combinatorial e o computador acaba por ter de calcular o mesmo termo várias vezes.
Para calcular `fibonacci_recursivo(5)` é necessário calcular `fibonacci_recursivo(4)` e `fibonacci_recursivo(3)`. Consequentemente, para calcular `fibonacci_recursivo(4)` é preciso calcular `fibonacci_recursivo(3)` e `fibonacci_recursivo(2)`. E assim sucessivamente.
No exemplo anterior para calcularmos o 5º termo da sequência Fibonacci seria necessário calcular `fibonacci_recursivo(4)` uma vez, `fibonacci_recursivo(3)` duas vezes, `fibonacci_recursivo(2)` três vezes e `fibonacci_recursivo(1)` duas vezes.

Cuidados com a recursividade

Ao escrever funções recursivas, deve-se ter um comando `if` em algum lugar para forçar a função a retornar sem que a chamada recursiva seja executada. Se não existir, a função nunca retornará quando chamada (equivalente a um *loop* infinito). Omitir o comando `if` é um erro comum quando se escrevem funções recursivas.
A condição de parada, neste caso um "`if`" que captura um valor chave, garante que o programa recursivo não gere uma sequência infinita de chamadas a si mesmo. Portanto, todo programa deve ter uma condição de parada que não seja recursiva.
Nos exemplos vistos, as condições de parada não recursivas capturadas pelo comando "`if`", eram:

- Fatorial: 0! = 1;
- Sequência Fibonacci: `fibonacci_recursivo(1) = 1` e `fibonacci_recursivo(2) = 1`

Sem essa saída não recursiva, nenhuma função recursiva poderá ser computada. Dessa forma, todo programa recursivo deve ter uma condição de parada não recursiva.

Vantagens

A maioria das funções recursivas não minimiza significativamente o tamanho do código ou melhora a utilização da memória. Além disso, as versões recursivas da maioria das rotinas podem ser executadas um pouco mais lentamente que as suas equivalentes iterativas devido às repetidas chamadas à função. De fato, muitas chamadas recursivas a uma função podem provocar um estouro de pilha. Como o armazenamento para os parâmetros da função e variáveis está na pilha e cada nova chamada cria-se uma nova cópia dessas variáveis, assim, a pilha pode provavelmente escrever sobre outros dados na memória ou até na área de memória de outro programa. Contudo não é necessário se preocupar com isso, a menos que a função recursiva seja executada de forma desenfreada.
A principal vantagem das funções recursivas é a possibilidade de utilizá-las para criar versões mais claras e simples de vários algoritmos, embora uma solução não recursiva envolvendo outras estruturas (pilhas, filas, etc.) seja mais difícil de desenvolver e mais propensa a erros. Dessa forma, ocorre um conflito entre a eficiência da máquina e a do programador. O custo da

programação está aumentando e o custo da computação está diminuindo. Isto leva a um cenário que não vale a pena para um programador demandar muito tempo para elaborar programas iterativos quando soluções recursivas podem ser escritas mais rapidamente. Somente deve-se evitar o uso de soluções recursivas que utilizam recursão múltipla, como por exemplo Fibonacci.