A [resposta do Guilherme Oderdenge](https://pt.stackoverflow.com/a/22409/215) demonstra bem o conceito de metaprogramação: "programas que criam programas". Na minha resposta, vou dar uma abordagem mais prática (raramente alguém escreve um programa cuja saída é outro, exceto por diversão ou por se estar escrevendo um compilador...).

Muitas tarefas de programação são repetitivas. Embora a finalidade do computador seja nos ajudar com a repetição, na prática nem sempre isso é possível. Devido a limitações nas linguagens de programação (entre outras ferramentas computacionais), muitas vezes nos vemos escrevendo as mesmas coisas de novo e de novo e de novo. Se não dá para escrever diretamente um programa que evite essas repetições, o que resta é escrever um programa diferente e então alterá-lo através de uma rotina automatizada (um "meta-programa").

De um modo geral, metaprogramação é toda programação que atua sobre outro programa, seja em formato fonte, binário, ou numa representação abstrata em memória.

### Transformando o código-fonte

O exemplo mais conhecido é o #define do C. Ele permite substituir uma expressão complexa por outra mais simples, sendo que a substituição é feita pelo próprio compilador:

#define CURSOR(top, bottom) (((top) << 8) | (bottom))

Outro exemplo é o eval, suportado por diversas linguagens. Ele permite que um trecho de código seja formado usando-se não somente um texto estático mas também componentes dinâmicos:

eval("function(x) { alert(x + " + y + "); }"); // Ex.: function(x) { alert(x + 10); }

### Transformando o código binário

O [AspectJ](http://pt.wikipedia.org/wiki/AspectJ) é uma ferramenta que traz a [Programação Orientada por Aspectos](http://pt.wikipedia.org/wiki/Programa%C3%A7%C3%A3o_orientada_a_aspecto) à linguagem Java. Um dos seus modos de uso é inserir código novo a classes existentes diretamente no código binário (*bytecode-weaving*). Escolhe-se um ou mais pontos bem definidos na execução de um programa, e escreve-se código para ser executado nesses pontos:

pointcut set() : execution(\* set\*(..) ) && this(Point); after () : set() { Display.update(); }

O código acima executa Display.update(); após a execução de todo método da classe Point cujo nome comece com set.

### Transformando uma representação abstrata

Toda linguagem que possui funções de primeira classe abre espaço para que as mesmas sejam modificadas programaticamente. A forma mais simples é através de decoradores:

function checarNulos(f) { return function() { for ( var i = 0 ; i < arguments.length ; i++ ) if ( arguments[i] == null ) throw 'Parâmetro nulo'; return f.apply(this, arguments); } } var func = checarNulos(function(a, b) { return a + b; });

Nesse caso se está transformando uma função simples - sem nenhuma checagem de erro - em outra mais robusta (*fail-fast*). Isso é feito de forma genérica, podendo ser aplicada a diversos tipos de função, com diferentes quantidade e formatos dos parâmetros.

Outra possibilidade, se a linguagem possui "classes de primeira classe", é aplicar transformações em uma classe como um todo (ex.: [metaclasses](https://pt.stackoverflow.com/q/22418/7261)).

Além desses casos, que eu particularmente chamo de "caixa preta" (pois somente a "fronteira" do código é alterado, sem o conhecimento do seu conteúdo), linguagens [homoicônicas](http://en.wikipedia.org/wiki/Homoiconicity) (como Lisp, Prolog ou XSLT) também permitem a transformação do código como um todo - incluindo instruções específicas, expressões, chegando até os [símbolos terminais](http://pt.wikipedia.org/wiki/S%C3%ADmbolos_terminais_e_n%C3%A3o-terminais). Essas transformações são genericamente chamadas de "macros" (embora sejam bem mais flexíveis que as "macros" da família do C).

term\_expansion((Cabeca :- Cauda), (Cabeca :- NovaCauda)) :- transforma(Cauda, NovaCauda). transforma((A, B), (NA, NB)) :- transforma(A, NA), transforma(B, NB). transforma(A is B + C, A is C + B). transforma(X, X).

O código acima pega toda instrução "somar duas variáveis" e inverte a ordem dos operandos. Se há duas ou mais instruções separadas por **E**, ele transforma tanto a primeira quanto a segunda parte. Embora esse código execute "em tempo de compilação" (i.e. antes da definição da função ser incorporada ao programa), ele já atua sobre uma representação abstrata do código fonte, não sobre o texto em si (seria como a [AST](http://pt.wikipedia.org/wiki/%C3%81rvore_sint%C3%A1tica_abstrata), mas em Prolog não há diferença entre "código" e "dados", então a AST do programa é representada por estruturas de dados da própria linguagem).

Por fim, mesmo linguagens sem "X de primeira classe" ainda podem dar suporte a alguma forma de metaprogramação, na forma de reflexão (*reflection*). Ela consiste em se inspecionar a estrutura de seus dados e atuar nos mesmos de forma genérica:

void funcaoDemorada(int argumento, float argumento2, Object callback, String nomeMetodo) { ... Method m = callback.getClass().getMethod(nomeMetodo); m.invoke(callback); }

Esse exemplo define uma função que pode chamar qualquer outra função [pública, zero parâmetros] em um objeto, bastando conhecer o nome dele. Sem isso, seria necessário que o objeto callback implementasse uma interface específica, e definisse um único método em conformidade com essa interface. Dessa forma, objetos de classes totalmente distintas e com métodos também distintos podem ser usados igualmente.

**Nota:** vários dos exemplos acima não podem ser considerados boa prática, e provavelmente existem melhores maneiras de se atingir o mesmo objetivo. Estão aí somente para exemplificar as técnicas mencionadas.

Concluindo, metaprogramação é muito mais amplo que "um programa gerar outro programa como saída", se tratando de diversos casos em que usa-se um programa (ou trecho de um programa) para se manipular outro programa - transformá-lo em algo diferente daquilo que ele seria se fosse interpretado literalmente, tal como o programador o escreveu. É provável que alguém discorde em relação ao limite do que é "programação normal" (i.e. o simples uso das funcionalidades da própria linguagem) e o que é "meta-programação", mas espero ter transmitido a ideia geral.