Recursão VS Iteração

Quanto à recursão e iteração, pode-se dizer que são formas estruturalmente diferentes, mas que realizam o mesmo objetivo em comum: Retornar o resultado desejado. Um exemplo muito prático de aplicação da recursão e iteração pode ser visto na inserção de um elemento no fim de uma lista ligada. Temos, então:

```
LL* Iterativo(LL *início, int valor){
        LL *temp = início;
        while(temp->next != NULL)t emp = temp->next;
        temp->next = new LL(valor);
        return início;
}
```

```
(define (recursivo lista valor)
        (if (null? lista)
                (cons valor ‘())
                (cons (car lista) (recursivo (cdr lista) valor))))
```

Vemos, pelo exemplo acima que ambos realizam a inserção, mesmo que ajam de forma diferente. Em suma, a recursão pode ser apenas uma solução elegante em alguns casos, mas podem haver outras situações que são naturalmente recursivas e, portanto, imprimem o resultado desejado com poucas linhas de código.

Recursão de Cauda

Antes de comentar sobre a transformação de uma recursão de cauda em um loop, vou, primeiramente, deixar claro o que é uma recursão de cauda: Assim como foi visto em aula, é uma forma sutil de evitar que ocorram estouros na pilha de recursão e esse comportamento é obtido quando usamos uma chamada recursiva dentro da função usando, por exemplo, um argumento extra que pode agilizar os cálculos, mas o que, de fato, configura uma recursão de cauda é que após o último cálculo ou chamada feito pela função não há nenhum tratamento antes de retornar seu valor. Veja abaixo alguns exemplos que deixam claro o método usado:

```
(define (fatorial_sem_cauda n)
        (if ( = n 1)
                1
                (* n (fatorial_sem_cauda (- n 1)))))

(define (fatorial_com_cauda n)
        (fatorial_auxiliar n 1))

(define (fatorial_auxiliar n aux)
        (if (= n 1)
                aux
                (fatorial_auxiliar (- n 1)  (* n aux))))
```

Observe que a diferença entre fat1 e fat2 consiste no uso de uma função auxiliar com um acumulador que permite imprimir a resposta desejada rapidamente, tal que essa velocidade poderia ser observada ao fazermos uma simulação do que ocorre na execução das duas chamadas: a função fat() precisa decrescer N até que ele seja igual à 1 para, depois, “subir” fazendo as multiplicações e subtrações, enquanto que a função fat2() só precisa decrescer N até 1 para retornar o resultado sem que haja tratamento algum antes de retornar. Agora, podemos notar que o fatAux() se parece com um while loop assim como está escrito abaixo:

```
Int acc = 1;
while(n>1){
        acc = acc * n;
        n = n -1;
}
return acc;
```

Portanto, é notável a correlação entre uma recursão de cauda e um loop iterativo, de tal modo que compiladores inteligentes convertem recursões de cauda automaticamente em loops.

Fechamentos

A noção de fechamento é fundamental para a compreensão do curry, let, let* e letrec. Um fechamento é a união de um ambiente e uma função, formando um escopo local, isto é, um bloco de código que possui variáveis, de tal modo que quando ele é finalizado, suas variáveis locais podem ser coletadas pelo garbage collector. Em LISP, particularmente, vimos os fechamentos serem criados em situações em que temos um ambiente, um conjunto de variáveis que recebem o que está no ambiente e uma expressão que transforma esses dados. Ou seja, estamos aplicando a expressão ao ambiente. Portanto, o fechamento é essencial para que o programador tenha a autonomia de, por exemplo, aplicar uma função em cada elemento de uma lista. Note que neste caso, a função é a expressão e a lista, o ambiente.

Funcionais comuns

Vamos, agora, explicitar alguns dos principais recursos de linguagens funcionais que são, também, ótimos exemplos do tópico anterior: Fechamentos. Isso fica claro quando prestamos atenção aos argumentos das funções e como é o comportamento de cada uma delas. Temos:

mapcar: Recebe uma lista e uma função e retorna uma lista em que cada um de seus elementos é o resultado da aplicação da função dada nos elementos da lista inicial. Portanto se a função for 'X-> X+1' e a lista for '(1 2 3 4 5). Então o resultado é '(2 3 4 5 6).

curry: Recebe uma série de argumentos que podem ser tratados de diversas formas, porque o quê ocorre de fato é a fixação parcial de parâmetros de outras funções que podem ser aplicados da forma que o programador desejar. Portanto se fizermos um curry que tem uma cadeia de lambdas, de tal modo que o primeiro lambda recebe uma função e os últimos recebam números, tal que a expressão resultante dessa sequência de lambdas é a aplicação da função nos dois valores, por exemplo: (curry (lambda (f)(lambda (x)(lambda(y)(f x y))))). Então (((curry *) 3) 4) irá retornar (* 3 4) que é igual à 12.

combine: recebe uma função binária, uma função unária e um valor inicial. A partir disso, quando aplicado à uma lista, o combine aplica a função unária em cada elemento e em seguida aplica a função binária, de tal modo que no fim retorna um único valor que representa a aplicação em cadeia da função em cada elemento da lista. Portanto, se a função binária fosse '*', a função unária fosse 'x -> x' e o valor inicial fosse 1 aplicado à seguinte lista '(1 2 3 4 5). Então o resultado seria 1*2*3*4*5 que é um produtório dos elementos da lista.

find: Recebe uma lista e um predicado, de tal modo que a lista é percorrida, verificando-se cada elemento com o predicado e retorna true se algum satisfazer ou false caso contrário. Portanto, supondo que o predicado seja '(number? valor)' e a lista seja '('alan 'eduardo 'mac0316). Então o resultado do final será falso porque todos os elementos da lista são símbolos.

Polimorfismo

O polimorfismo é um recurso muito útil que é usado em linguagens orientadas a objetos e consiste no uso de um mesmo método(com assinatura igual), mas com comportamento distinto quando usado em classes diferentes. Durante as aulas vimos 3 abordagens diferentes de implementação do polimorfismo, que podemos enumerar:

1)Listas de Associação: consiste na adição de um argumento extra na chamada das operações, sendo esse argumento extra a função de igualdade do tipo usado.

2)Listas de Associação com a função de igualdade: visando a diminuição no número de argumentos nas chamadas das operações, esse método inclui a função de igualdade na representação, de tal modo que podemos usar o curry para parametrizar o uso da função de igualdade. Assim, carregamos um elemento a mais por conjunto.

3)Listas de Associação com funções de manipulação criadas pelo Scheme e, em seguida, atribuídas nomes: Essa abordagem tenta ser intermediária, não forçando o usuário a usar argumentos extras ou carregar um elemento adicional por conjunto. Para isso, fazemos as funções de manipulação automaticamente pelo scheme e atribuímos nomes para cada uma delas.

Um exemplo que facilita a compreensão desse conceito pode ser o seguinte: Dado um tipo 'Point' que possui um inteiro x0 e v0, podemos definir que a igualdade entre dois objetos do tipo Point ocorre quando os inteiros da classe são iguais. Assim, no caso 1), teríamos que carregar a função que determina essa igualdade em cada chamada função de manipulação, enquanto que no método 2) poderíamos carregar no início da lista.

Escopos: como criar variáveis locais com fechamentos.

O principal comando que vimos em LISP e teve um grande papel na compreensão dos conceitos seguintes foi o ‘(lambda (x) (exp)) valor’ e ele representa justamente a criação de uma variavél local em um fechamento porque assim como foi visto no tópico sobre fechamentos, basta que haja um ambiente, uma expressão e a definição de uma variável que recebe um valor desse ambiente.
No entanto, também vimos como criar variáveis locais usando, por exemplo uma chamada de uma função auxiliar com um argumento extra que agiliza ou facilita o cálculo do que será retornado. Assim, o mesmo exemplo que foi usado no tópico de Recursão de Cauda pode ser reutilizado: Em fatorial_com_cauda chamamos a função fatorial_auxiliar que possui o argumento extra ‘aux’, tal que ele carrega o resultado do fatorial a ser retornado imediatamente quando N for igual à 1.

Açúcar sintático

Um açúcar sintático é criado quando existe algum comando que é usado com muita frequência pelos programadores e poderia ser simplificado. Vemos muitos exemplos disso, seja nos for loops do C/C++ ou em LISP com o let,let* e letrec. No nosso interpretador do EP havia uma função desugar, que tinha como objetivo ser uma espécie de “filtro”, porque para cada chamada da função fazia uma série de verificações para entender a entrada e redirecionar para o comando primitivo (que foram definidos em ExprC). Assim sendo, podemos ver uma clara correlação entre o que foi dito sobre açúcares sintáticos e o uso de desugar no EP, pois o interpretador meramente redirecionava para o comando primitivo algo que foi inserido de outra forma, mas que realiza aquilo que é desejado.

- Escopos: let, let* e letrec: funcionamento. Porque são açúcar sintático.

Primeiramente, explicitando o funcionamento do let, let* e letrec: O let permite a criação de variavéis e executa expressões em cada uma deles de forma paralela, enquanto que o let* seque o mesmo conceito, no entanto, realizando as expressões de forma sequencial, permitindo, por exemplo, que novas variáveis sejam definidas a partir de variáveis anteriores. O letrec é o mais complexo dos três porque permite a criação de uma função e a sua chamada em seguida. Veja um exemplo a seguir:

```
(letrec ([is-even? (lambda (n) (or (zero? n) (is-odd?  (sub1 n))))]
         [is-odd?  (lambda (n) (or (zero? n) (is-even? (sub1 n))))])
 (is-odd? 11))
#t
```

O let e let* são açúcares sintáticos porque poderiam ser implementados usando uma sequência de comandos diferentes, mas foram simplificados para que sejam mais facilmente utilizados. Em aula, nos foi mostrado como poderiamos fazer a implementação, por exemplo, do let :

```
((lambda (x1 … xn) exp) e1 … em)
```

Observe que está completamente de acordo com a definição dada acima, pois cada comando é executado paralelamente e cada variável recebe um valor dado. Portanto, o mesmo se aplica ao let*.

-Variáveis próprias (exemplo do rand): criação de "escopos globais" limitados.

Assim como foi visto no exemplo do rand, a ideia é usar a expressão lambda para criar variáveis locais que mantém um estado da chamada da função para a seguinte. Assim , orand se aproveitava dessa propriedade para assegurar sua aleatoriedade, pois sempre tinha em mãos o valor anterior para gera o próximo. Veja como isso foi feito:

```
(define init-rand (lambda (valor-inicial)
        (lambda () (begin
                (set! valor-inicial (remainder (+ (* valor-inicial 9) 5) 1024))
                valor-inicial)))
```

Observe que em (lambda () (exp) obtemos um escopo com a variável local valor-inicial. Assim, usamos o begin e set! Para modificá-lo e conter novo valor a partir da expressão utilizada com remainder, etc.

- Continuações: o que são, para que são usadas. Exemplo. Uso em recursão dupla

As continuações são formas de postergar comandos para criar efeitos alternativos, pois, assim como foi visto nas funções que exprimiam o mdc de uma lista, nos era possível esperar até que se chegasse até o final da lista para que os cálculos fossem feitos. Portanto, é uma espécie de encapsulação do “futuro” porque indica o que deve ser feito em seguida, fazendo um fluxo de controle. Um exemplo muito elegante visto em aula usando fechamentos é dado por:

```
(define mdc-otimo (lambda (lista)
        (mdc-aux (lista id))))
(define mdc-aux (lambda (lista resto-da-conta)
        (if (= (car lista) 1)
                1
                (if (null? (cdr lista))
                        (resto-da-conta (car lista))
                        (mdc-aux (cdr lista) (lambda (n) (resto-da-conta (mdc (car lista) n))))
)))))
```

Vemos que, somente após a lista acabar são feitas as chamadas da função resto-da-conta e há um nítido fluxo de controle para o caso em que o número 1 é encontrado na lista, porque ele imediatamente é tido como solução do máximo divisor comum.

Call/cc: qual a diferença com continuações usando fechamentos. Como funciona, qual seu uso.

Dado que se há continuação, há recursão de cauda, o Call/cc (call with current continuation) permite que seja usada a continuação atual do interpretador. Usando um exemplo prático, temos que em uma dada função que manipula um numero qualquer de variáveis é composta por várias continuações, pois em (+ 3 4):

+ é (lambda (f) (f 3 4));
3 é (lambda (x) (+ x 4));
4 é (lambda (x) (+x 3));

Portanto o call/cc é uma função de um argumento, tal que esse argumento é uma função e aplica essa função dada como argumento à continuação atual, seja:

```
(define f (lambda (k) (k 5)))
(+ (call/cc f) 4)
9
```

Observe que f é uma chamada de lambda que aplica o valor 5 para a variavel ‘k’, assim a continuação de call/cc passou de (lambda (x) (+ x 4)) para (f (lambda (x) (+ x 4))), então (lambda (x) (+ x 4)) 5) = (+ 5 4) = 9.

Interpretador do EP: como é a estrutura básica de processamento de uma s-exp no interpretador do ep? Quais passos e o que cada um faz?

Uma s-exp é um argumento de entrada do interpretador e, ela é recebida pelo interpS, que imediatamente faz a chamada da função interp com o desugar do parse da s-exp. No parse, é verificado seela se trata de um número, um símbolo ou uma listas, tal que nesse últimos caso é percorrido uma série de condicionais para verificar qual o comando escolhido e quais os argumentos dados. Após a verificação das condições, o desugar entra em ação, no qual seu objetivo é fazer uma espécie de “redirecionamento” para o que foi definido de forma primitiva no nosso interpretador para que a chamada do interp funcione com exatidão.

Interpretador do EP: quais ExprC voce criou, e ExprS, e subtipos de Value? Porque?

Os ExprC criados foram quoteC(), letrecC e read-loopC enquanto que os ExprS foram letS, let*S, QuoteS, letrecS e read-loopS. O sub-tipo de Value criado foi SymV como obrigatoriedade para o funcionamento do quote. O let e let* foram feitos apenas no ExprS porque foram feitos apenas como açúcares sintáticos, enquanto que o letrec foi feito no ExprC porque foi projetado como algo primitivo, assim como o quoteC e read-loopC. A existência do quote, letrec e read-loop em ExprS foi apenas uma forma de redirecionar para a chamada primitiva.