Universidade Federal de São Carlos - Departamento de Computação
Paradigmas de Linguagens de Programação - Prof. Zorzo

- **Trabalho - Paradigma de Programação Funcional**

Uma das áreas em que a programação funcional é utilizada é em aplicações que envolvem derivação e processamento simbólico.

No processamento simbólico de expressões matemáticas visa-se, não à avaliação numérica, mas sim à transformação dessas expressões em outras. A derivação de uma expressão y em relação a uma variável x é um bom exemplo de processamento simbólico.

As expressões matemáticas serão representadas sob a forma de listas. Por conveniência, as expressões estarão completamente parentizadas, o que significa que cada expressão consiste de um operando A, um operador OP e um operando B, onde tanto A como B poderão ser elementares (uma variável ou uma constante) ou, uma expressão matemática. Assim uma expressão como **a + x + 3** deverá ser representada por **(a + (x + 3)) ou ((a + x) + 3)** com a associatividade a direita e esquerda, respectivamente.

É indiferente escolher a primeira ou a segunda forma, nessa nossa abordagem. O essencial em expressões completamente parentizadas é não indicar mais de uma operação em cada sub-expressão (ou lista).

Por motivos de brevidade, restringiremos no exemplo a seguir o operador OP à soma e produto. O exemplo poderá ser facilmente estendido para incluir outras operações, incluindo as suas regras de derivação. Esses operadores podem ser simples, como a divisão ou complexos, como o seno, cosseno, etc.

O aspecto mais interessante ao elaborar o programa é que este é uma cópia imediata das próprias regras de derivação. Para derivar uma expressão y em relação a x, sendo y restrita a soma de produtos, temos as regras:

- se y == x , der(y,x) = 1
- se y == variável diferente de x ou constante, der(y,x) = 0
- se y == u+v, der(y,x) = der(u,x) + der(v,x)
- se y==u*v, der(y,x) = u* der(v,x) + v* der(u,x)

- note que essas duas regras são recursivas.
- note que $2x^2 + x + 2$ deve ser dado como (((2*(x*x))+ x) + 2)

Em cada sub-expressão, y tem a forma (A OP B), poderemos selecionar cada um dos componentes como :

Operando A - por (car y)
Operador OP - por (car (cdr y)) ou (cadr y)
Operando B - por (car (cdr (cdr y))) ou (caddr y)

```
(defun der ( y x)
 (cond
  ((atom y) (cond
             (  (equal y x)    1)
             ( T               0)
           )
  )

  (  (equal (cadr y)    '+)
           (list  ( der (car y) x) '+ (der (caddr y) x ) )
  )
```

```
    ((equal (cadr y) '* )
              (list  ( list (car y) '* (der (caddr y) x ))
                         '+
                        ( list (der (car y) x )) '* (caddr y)
              )
    )

  )
)
```

e se aplicarmos em lisp teremos o seguinte resultado:

```
> (der '(a * (x + b)) 'x)
((A * (1 + 0)) + (0) * (X + B))
> _
```

que é a resposta correta, embora expressa de forma inconveniente.

Gostaríamos de efetuar simplificações óbvias, dadas pelas regras a seguir, que poderiam ser reduzidas a metade se incorporássemos a regra da comutatividade da soma e do produto.

1) 0 + e = e
2) e + 0 = e
3) 0 * e = 0
4) e * 0 = 0
5) 1 * e = e
6) e * 1 = e

A tendência seria a implementação imediata dessas regras, como foi feita para a derivação. Entretanto, aqui precisamos de mais atenção. Antes de tudo, notaremos a necessidade de aplicações recursivas, como no exemplo (0 + (b + 0)) reduziria a (b + 0) pela regra 1, mas poderíamos continuar a aplicar as regras até encontrar a expressão mais simples. Por outro lado, nenhuma regra se aplica a ((a + 0) + (b + 0)) de uma forma direta, embora a regra 2 possa ser aplicada em cada subexpressão.

Essas considerações nos levariam a um esquema recursivo, lembrando a regra de derivada de soma:

simp (A OP B) = simp (A) OP simp (B)

que funcionaria corretamente para (a + b), mas falharia na simplificação de (0 + (b + 0)), já que produziria o resultado (0 + b), mas deixaria de reconhecer que esse resultado pode ser simplificado.

Um esquema recursivo satisfatório simplificaria a expressão x aplicando as regras às subexpressões de x. Informalmente, primeiro penetramos em x até atingir suas subexpressões mais internas (átomos, sobre os quais nenhuma regra se aplica), para em seguida “subir” aplicando as regras a cada nível sucessivo.

Esse esquema envolveria duas funções: a primeira, simp, encarregada de penetrar nas subexpressões e de invocar a segunda função, s2, que contém as regras para efetuar as simplificações à medida que simp retorna para níveis sucessivamente mais altos.

A chamada da função > **(simp (der ‘ (a * (x + b)) ‘x))** produzirá o resultado **a**

Pede-se: implemente a função simp