


Nome: Guilherme Santos de Godoy

RA: 758710

Curso: Ciência da Computação (BCC 018)

Trabalho 1 - Paradigmas de Linguagens de Programação - Prof. Sergio D. Zorzo

A princípio, foi criada a função *der*, conforme a especificação do trabalho, que será utilizada para expandir a expressão de entrada em sub-expressões que permitam a utilização das regras de simplificação estabelecidas:

 GNU CLISP 2.48

```
i i i i i i i      ooooo  o      oooooooo  ooooo  ooooo
I I I I I I I      8      8  8      8      8      o  8      8
I \ \ `+' / I      8      8      8      8      8      8      8
\ \ \ -+-' /      8      8      8      ooooo  8oooo
  -_ | _- '      8      8      8      8      8      8
    |      8      o  8      8      o      8      8
-----+-----      ooooo  8ooooooo  ooo8ooo  ooooo  8

Welcome to GNU CLISP 2.48 (2009-07-28) <http://clisp.cons.org/>

Copyright (c) Bruno Haible, Michael Stoll 1992, 1993
Copyright (c) Bruno Haible, Marcus Daniels 1994-1997
Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998
Copyright (c) Bruno Haible, Sam Steingold 1999-2000
Copyright (c) Sam Steingold, Bruno Haible 2001-2009

Type :h and hit Enter for context help.

[1]> (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)
          ))
  ))
DER
[2]>
```

Em seguida, a partir das regras de simplificação estabelecidas na especificação do trabalho, foi criada a função *s2*, utilizada nas chamadas recursivas da função principal para verificar as possibilidades de simplificação:

```
[2]> (defun s2 (lista)
  (cond
    ((null lista) nil)
    ((null (cdr lista)) (car lista))
    ((and (equal (car '+)) (car (cdr lista))) (equal 0 (car lista))) (list (car (cddr lista))))
    ((and (equal (car '+)) (car (cdr lista))) (equal 0 (car (cddr lista)))) (list (car lista))
    ((and (equal (car '*)) (car (cdr lista))) (equal 1 (car lista))) (list (car (cddr lista)))
    ((and (equal (car '*)) (car (cdr lista))) (equal 1 (car (cddr lista)))) (list (car lista))
    ((and (equal (car '*)) (car (cdr lista))) (equal 0 (car lista))) (list (car lista))
    ((and (equal (car '*)) (car (cdr lista))) (equal 0 (car (cddr lista)))) (list (car (cddr lista))))
    (t lista)
  )
)
s2
```

Alguns pontos devem ser notados:

- Na segunda condição, que verifica se a lista atual tem um único elemento, é retornado o primeiro elemento desta lista, para que seja possível efetuar as comparações necessárias em outras partes do programa, dispensando operações entre listas com um único valor. Note que o elemento desta lista pode ser uma sub-expressão.
- As regras foram criadas de maneiras similares, utilizando a ideia de verificar se o elemento do meio é o operador de + ou *, e também verificando se uma das pontas da expressão é o valor 0 ou 1 (não foi utilizada a regra da comutatividade no desenvolvimento desta função para tornar mais clara a percepção de quais elementos estão sendo utilizados na comparação e quais estão sendo retornados como resultado).
- Ainda referente às regras criadas, é importante notar que os elementos são retornados como listas. Esta decisão foi tomada para evitar a necessidade de criar uma nova exceção que trate apenas da verificação de elementos únicos, de modo que esta definição de lista é removida pela segunda condição (explicada anteriormente) quando necessário.

Por fim, foi criada a função *simp*, responsável por lidar com as manipulações necessárias para apresentar o resultado esperado:

```
[3]> (defun simp (lista)
  (cond
    ((null lista) nil)
    ((null (cdr lista)) (car lista))
    ((atom (car lista)) (car lista))
    ((or (equal (car (cdr lista)) (car '+)) (equal (car (cdr lista)) (car '*))) (simp (car lista)))
    ((or (equal (car (cdr lista)) (car '+)) (equal (car (cdr lista)) (car '*))) (simp (car (cddr lista))))
    (t (s2 lista))
  )
)
simp
```

Algumas noções importantes para sua compreensão:

- De forma similar à função *s2*, foi estabelecida uma condição que retorna o elemento único de uma lista com apenas um elemento (segunda condição). Note que o elemento desta lista pode ser uma sub-expressão.

- A terceira condição verifica se a lista atual é composta por um elemento único, ou seja, um átomo. Neste caso, também retorna o elemento fora da lista para possibilitar as operações com outros átomos.
- A quarta e a quinta condição seguem o mesmo princípio: verificam se o elemento do meio é o operador $+$ ou $*$ (considerando que seja uma sub-expressão no formato $x + y$, por exemplo). Caso obedeça a esta condição, a função *simp* é chamada novamente, a princípio para o elemento à esquerda do operador (quarta condição) e em seguida para o elemento à direita do operador (quinta condição), gerando uma recursividade que busca átomos ou sub-expressões que possam ser simplificadas por alguma das regras estabelecidas em *s2*.
- A última condição considera que foi encontrada uma sub-expressão ou átomo válido e chama a função *s2* para efetuar a simplificação adequada.

Desta forma, com a entrada de exemplo apresentada na especificação do trabalho, recebemos o resultado esperado:

```
[4]> (simp (der '(a * (x + b)) 'x))  
A  
[5]>
```

OBS: os códigos apresentados no terminal do CLISP podem ser encontrados no seguinte repositório do GitHub caso deseje efetuar novos testes:

<https://github.com/GuilhermeSGodoy/Paradigmas-Linguagens-Programacao/tree/main/T1>