ISSN 0101-9201

MINISTÉRIO DA CIÊNCIA E TECNOLOGIA

CNPq

CBPF

CENTRO BRASILEIRO DE PESQUISAS FISICAS

Notas Técnicas

CBPF-NT-001/88

INTRODUÇÃO AO SISTEMA REDUCE DE CÁLCULO ALGÉBRICO: VERSÃO PRELIMINAR

por

Renato P. dos SANTOS

RIO DE JANEIRO
1988

A Tania, Rosa Maria, Sandra, Deise, Marcia, Neusa, Luci, Vera, Úrsula e Iza Maria.

Às Sofias, Beatrizes e Margaridas.

Pois que

...Das Ewigweibliche
Zieht uns hinan.
(...O Eterno-Feminino
Atrai-nos para o alto.)
FAUSTO II,V,7

Prefácio

Estas notas foram redigidas inicialmente baseadas no mini-curso "Computação Algébrica - Sistema REDUCE" que apresentamos na Escola de Verão/87 do Departamento de Matemática da UnB, com apoio do Centro Científico da IBM.

Com as apresentações posteriores deste mini-curso no CBPF/DRP, no IFUSP, no IFT/UESP e no IF-UFRJ, tivemos oportunidade de aperfeiçoá-lo.

Conhecendo de perto as dificuldades por que passa o iniciante em REDUCE dispondo apenas do manual, o qual, sendo de referência do sistema, não tem o intuito de ser pedagógico, esperamos que aquelas sejam minimizadas pela consulta a este material.

Gostaríamos aqui de agradecer o apoio e estímulo recebido de Rubens Marinho (IEAv/CTA), Jair Lucinda (IEAv/CTA), Alberto Santoro (CBPF/DRP), Jaime Tiomno(CBPF/DRP), Ívano D. Soares (CBPF/DRP), Prem P. Srivastava (CBPF/DRP), Marcelo Rebouças (CBPF/DRP), Luis A. Reis (CBPF/CI), Waldir L. Roque (UnB), Fernando Giorno (CC-IBM), Henrique Fleming (IFUSP), Cristina Abdalla(IFT/UESP), Stenio W. A. de Melo (IF-UFRJ).

Agradecemos, também, aos participantes dos mini-cursos por suas perguntas e sugestões que se transformaram em contribuições valiosas a estas notas.

Sugestões e críticas dos leitores serão bemvindas.

SUMÁRIO

Capítulo 1

Introdução à Computação Algébrica

"The purpose of computing is insight, not numbers."

Richard W. Hamming
Bell Laboratories

"The problem with computers is that they only give answers."

attributed to P. Picasso

Estamos acostumados a ver computadores trabalhando com números, isto é, processando números obtendo números como resultado. O fato, porém, é que o computador pode processar igualmente símbolos, isto é, obter um resultado expresso em símbolos.

Isto não é surpreendente e conhecemos algumas aplicações deste tipo, como recuperação de informações, linguística computacional, auxílio na tomada de decisões, etc. Aqui, estaremos mais interessados em manipulação algébrica.

A computação algébrica já foi utilizada em muitas áreas do conhecimento humano, tais como: mecânica celeste, óptica, física de plasmas, química, relatividade geral, geofísica, análise numérica, biofísica, mecânica dos fluidos, acústica, mecânica estatística, economia, teoria de controle, mecânica estrutural, processamento de imagens, eletrodinâmica quântica, análise de decisões, teoria dos números, física do estado sólido, etc.

Considera-se que a primeira referência à computação algébrica tenha sido já em 1844, quando Lady Lovelace, protetora de Charles Babbage, previu a possibilidade de sua Máquina Analítica manipular símbolos.

No entanto, só em 1953, através do trabalho de Kahrimanian e Nolan, conseguiu-se construir um sistema que realizasse cálculos algébricos simples: diferenciando-se $\sin(x^2)$, obteve-se $2x\cos(x^2)$. Note-se que x é uma quantidade indefinida, sendo o cálculo acima válido para qualquer valor de x.

Em 1957, Newel, Simon e Shaw desenvolveram a linguagem IPL-V que operava com dados em forma de listas e dispunha do recurso de alocação dinâmica de memória.

Em 1960, McCarthy apesenta a linguagem LISP ("LISt Processing"), que possui, também, os recursos de recursão e *coletor de lixo* (*"garbage collector"*).

É interessante ressaltar que SAINT, o primeiro sistema capaz de realizar integrações indefinidas (uma tarefa muito mais difícil que diferen-

ciação, uma vez que não há uma estratégia definida a priori para aquela), foi elaborado em 1961 por Slagle, um pesquisador interessado em estudar os processos da simulação da inteligência humana.

Em 1965, Trufyn apresenta a linguagem FORMAC, desenvolvida a partir da FORTRAN, à qual foram acrescidas instruções especificas para a manipulação algébrica. Esta linguagem foi considerada interessante por permitir ao pesquisador familiarizado com cálculos numéricos em FORTRAN, apenas pelo aprendizado de alguma instruções novas, ser capaz de realizar, também, cálculos algébricos.

De 1970 para cá, surgiram muitos outros sistemas algébricos, tais como REDUCE, SHCOONSHIP, MACSYMA, SMP, muMATH, etc.

1.1 Computação Algébrica versus Computação Numérica

Na introdução, apareceu um exemplo de computação algébrica, uma diferencia ção. Vejamos, agora, mais alguns exemplos para que fique clara a diferença com relação à computação numérica.

No primeiro exemplo abaixo[1], expandimos uma expressão que resulta num polinomio de alto grau.

```
5:      (x-1)**20*(2*x-1)**20*(3*x-1)**30;

                 30          20
        (3*X - 1)  *(2*X - 1)  *

               20
        (X - 1)

6:      ws;

                                     70
        215892499727278669824*X        - ...

                                                        45
 ...    418692282395140768550832249605496*X        + ...

                   5              4               3
 ...    534866040*X   + 19078420*X   - 535080*X

                2
         + 11065*X   - 150*X + 1
```

[1]Aqui e nos exemplos futuros indicaremos com letra *minúscula* o que foi digitado pelo usuário e com letra *maiúscula* o que foi escrito pelo sistema

O procedimento consistiu em abrir os parênteses, efetuando as exponenciações e multiplicações correspondentes, simplificar os termos de potência semelhante e escrever o resultado em ordem decrescente das potências da variável. Tudo isso foi realizado automaticamente pelo programa algébrico, bastando apenas digitar a expressão de entrada, respondendo ele com a expressão simplificada. Na verdade, ele imprime todos os 71 termos do polinomio. Por edição, deixamos apenas alguns termos típicos. Note, também, que os coeficientes são números arbitrariamente grandes.

No exemplo seguinte, apresentamos alguns exemplos de diferenciação simbólica. As funções DILOG e ERF são as funções matemáticas dilogaritmo e função erro.

```
21:  df(cos(x),x);

       - SIN(X)

22:  df(atan(x),x);

                2
           1/(X  + 1)

23:  df(expint(x),x);

            X
           E /X

24:  df(dilog(x),x);

          ( - LOG(X))/(X - 1)

25:  df(erf(x),x);

                                 2
                               (X )
          (2*SQRT(PI))/(E      *PI)
```

No terceiro exemplo, temos uma integração simbólica. Note o tempo de CPU que o sistema levou para esse cálculo : *menos de meio segundo!*

```
2:       log(x)**5;

                5
         LOG(X)

3:       int(ws,x);
```

```
                    5             4             3
X*(LOG(X)  - 5*LOG(X)  + 20*LOG(X)  - 60

                  2
  *LOG(X)  + 120*LOG(X) - 120)

TIME:  490 MS
```

No primeiro exemplo acima, obtivemos um polinomio com coeficientes que eram quocientes de números inteiros maiores que os que usualmente se obteriam numa linguagem numérica convencional. De fato, conforme se vê do exemplo abaixo, quando obtemos um número com 202 dígitos, pode-se trabalhar, em geral, com inteiros bastante grandes utilizando-se linguagens algébricas.

```
4:      2**1000;

      107150860718626732094842504906000181056140481
      170553360744375038837035105112493612249319837
      881569585812759467291755314682518714528569231
      404359845775746985748039345677748242309854210
      746050623711418779541821530464749835819412673
      987675591655439460770629145711964776865421676
      604298316526243868372056680693 76
```

É interessante esse recurso de se representar números racionais como quocientes de inteiros pois tem-se uma precisão absoluta, ao invés da aproximação decimal feita pelas linguagens numéricas convencionais.

A importância disso pode ser exemplificada na geração dos polinômios de Legendre (usando a fórmula de Rodrigues) como abaixo

```
      REDUCE 3.1, 15-Apr-84 ...

1:  linelength(45);

      72

2:  procedure pl(n,x);
2:  df((x**2-1)**n,x,n)/2**n/factorial(n);

      PL

3:  factor x;on div,rat;
```

```
5:  pl(16,x);

                      16                          14
300540195/32768*I     - 145422675/4096*I

                     12
+ 456326325/8192*I    - 185910725/4096*

  6                   4                   2
*I  + 4349845/8192*I   - 109395/4096*I
```

O ponto é que se os coeficientes do polinômio P_{16} acima houvessem sido arredondados, este não seria mais um polinomio de Legendre (as propriedades de ortogonalidade, por exemplo, não mais valeriam).

No exemplo seguinte, demonstramos que as linguagens simbólicas podem também, em geral, trabalhar com números. Esses sistemas, via de regra, são mais lentos, para esse tipo de cálculo, que linguagens essencialmente numéricas como **FORTRAN**, mas permitem uma precisão muito grande.

```
1:  pi;

    PI

2:  on bigfloat,numval;  ws;

    3.141 59265 4

4:  precision 50; pi;

    3.141 59265 35897 93238 46264 33832 7950

    2 88419 71693 99375 1

6:  cos(ws/6);

    0.866 02540 37844 33646 76372 31707 5293

    6 18347 14026 26905 19

7:  ws**2;

    0.75
```

Como exemplo, temos o número π com 50 digitos de precisão (e poderíamos apresentá-lo com precisão ainda maior!). Em seguida, calculamos $\cos(\pi/6)$, também com precisão de 50 dígitos. Para verificar a exatidão do cálculo, cujo resultado deveria ser $\sqrt{3/2}$, elevamos o resultado obtido ao quadrado. Note que, como o resultado é exato, temos apenas dois dígitos, embora a precisão admitida continue sendo de até 50 digitos.

Um ponto importante a se levar em conta em computação algébrica é a ocupação de memória.

Em computação numérica, uma posição de memória com 4 *bytes* pode armazenar um número inteiro de magnitude máxima 2.147.483.647 ou um número em ponto flutuante de magnitude entre 10^{-78} e 10^{76}, com 7 dígitos de precisão, o que é suficiente para a maioria dos cálculos numéricos.

Em LISP, expressões algébricas, contrariamente a linguagens como FORTRAN, são escritas em notação prefixada, isto é, o operador vem à frente dos operandos. Por exemplo, temos

notação usual	notação FORTRAN	notação LISP
$x^2 + x + 3$	x*x+3*x+3	(+ (+ (* x x) x) 3)

Esta expressão, por demais simples, ocuparia, no código mais comum dos usados para armazenamento de expressões, no qual se utiliza 1 *byte* para cada caracter, 6 *bytes*. Em LISP seriam necessários no mínimo 18 *bytes*, considerando-se que cada átomo necessite de 2 *bytes* para ser armazenado.

Por isso, não é incomum um cálculo algébrico necessitar de 2 *megabytes* ou mais. Além disso, durante um cálculo, a ocupação de memória por cálculos intermediários pode ser desastrosa. Por exemplo, seja a expansão de $(x+a)^3$:

$$\begin{aligned}
(x+a)^3 &= (x+a)(xx+xa+ax+aa) \\
&= (xxx+xxa+xax+xaa+axx+axx+axa+aax+aaa) \\
&= (xxx+axx+axx+aax+axx+axx+aax+aax+aaa) \\
&= (x^3+ax^2+ax^2+a^2x+ax^2+a^2x+a^2x+a^3) \\
&= (a^3+a^2x+a^2x+a^2x+ax^2+ax^2+ax^2+x^3) \\
&= (a^3+3a^2x+3ax^2+x^3)
\end{aligned}$$

Temos, nas primeiras duas linhas, a abertura dos parenteses, efetuando as multiplicações correspondentes. Em seguida, alfabetiza-se[1] cada termo, colocando os termos de cada produto em ordem. Efetuam-se os produtos e alfabetizam-se, então, os termos da expressão. Finalmente, simplificam-se os termos semelhantes, obtendo o resultado desejado.

O ponto a se ressaltar aqui é que as listas contendo os resultados inter-

[1] a é considerado alfabeticamente anterior a x, assim como aax o é com relação a axx

mediários só serão descartadas após a obtenção do resultado final, por um dispositivo chamado *coletor de lixo* (*"garbage collector"*). Considerando-se a memória necessária para se armazenar cada termo de cada lista, podemos calcular a quantidade de memória utilizada para um cálculo tão simples como este. Imagine, então a quantidade necessária para um problema real bem mais complexo!

Outro aspecto relevante dos sistemas algébricos baseados em linguagens como LISP e C é a característica da *recursão*, isto é, um programa ou função referir-se a si mesmo na sua definição. O exemplo clássico de recursão é a definição do fatorial:

$$fatorial(0) = 1$$

$$fatorial(n) = n \times fatorial(n-1) \quad (n > 0).$$

Isso permite, frequentemente, a codificação de programas mais elegantes e, por vezes, só se possui uma definição recursiva para a função desejada.

No entanto, em alguns casos, esse recurso não é conveniente pois forma-se uma cadeia de resultados pendentes da definição recursiva até se encontrar um resultado que pode ser retornado no caminho inverso, obtendo-se o resultado final. Esquematicamente, teríamos, por exemplo, para o fatorial de 4:

4!	*(cálculo proposto)*
4 × 3!	*(aplicação recursiva)*
4 × 3 × 2!	"
4 × 3 × 2 × 1!	"
4 × 3 × 2 × 1 × 0!	*(encontrado o cálculo definido* 0!)
4 × 3 × 2 × 1 × 1	*(retorno da recursão)*
4 × 3 × 2 × 1	"
4 × 3 × 2	"
4 × 6	"
24	*(fim do cálculo)*

Até ser encontrado o resultado definido 0!, foi sendo construída a cadeia de resultados pendentes intermediários, consumindo memória. No caso de uma definição não recursiva do fatorial, ao estilo FORTRAN, teremos uma sequência de multiplicações, usando sempre a mesma quantidade de memória, sem resultados pendentes.

Essa utilização maior de memória pode ser catastrófica. No exemplo abaixo, apresentamos definições recursiva e iterativa do fatorial e aplicações dos mesmos, verificando-se a sobrecarga de memória no caso recursivo.

```
8:  procedure factrecur(n);
8:        if n=0 then 1 else n*factrecur(n-1);
```

```
      FACTRECUR

4:  procedure factinter(n);
4:        if n=0 then 1 else for i:=1:n product i;

      FACTINTER

5:  factrecur(3);

      6

6:  factinter(3);

      6

7:  factrecur(70);
  *** PUSHDOWN STACK OVERFLOW
  *   39

8:  factinter(70);


      11978571669969891796072783721689098736458938l
      42546425857555362864628009582789845319680000
      00000000000
```

1.2 Heurística : o contato com a Inteligência Artificial

Certos problemas em Computação são muito complexos, envolvendo uma quantidade enorme de decisões na busca de sua solução.

Os possíveis caminhos, decorrentes das várias possibilidades em cada ponto de decisão conformam uma "árvore de possibilidades", que pode se tornar bastante grande. O programa deve incluir, então, recursos para pesquisar essa árvore e encontrar um caminho que leve à solução do problema.

No caso de um programa algorítimico, seria feita uma busca intensiva até o fim através de cada caminho, de cada encruzilhada, em busca da meta. Todavia, como é fácil imaginar, se o número de possibilidades for muito grande, esse método torna-se inviável.

Um exemplo disso é o jogo de xadrez, no qual, em cada lance, existem vários movimentos possíveis para cada jogada. Os estudiosos calculam que o número de combinações de movimentos possíveis numa partida de xadrez chega a 10^{120}, estando, dessa forma, fora do alcance de qualquer computador.

O método heurístico consiste em um critério, frequentemente empírico, fornecido pelo programador, para avaliar, em cada encruzilhada, qual caminho *provavelmente* levará a um resultado mais próximo da meta. Quando, por meio desse critério, conclui-se que se está afastando da meta, retorna-se à encruzilhada anterior e tenta-se o seguinte caminho mais provável de solução.

Um exemplo simples de programa heurístico é o do jogo da velha, que tem cerca de 91 combinações.

Por exemplo, o critério de avaliação poderia ser um parâmetro que avaliaria a vantagem do computador com relação a seu oponente. Uma proposta de parâmetro seria

$$y = y_1 + 5y_2 + 25y_3$$

onde y_1 seria o número de linhas, colunas e diagonais com uma marca não bloqueada por marca adversária, menos o número correspondente ao adversário, y_2, o número de linhas, colunas e diagonais com duas marcas e y_3, idem com três marcas.

Os pesos 1, 5 e 25 referem-se à importância relativa de cada situação (quando y_3 é diferente de 0, ocorre a vitória de um dos jogadores), são, de fato, um pouco arbitrários e sua alteração influi na eficiência do programa como jogador.

Numa situação como a abaixo

		×
o	×	
o		

teríamos a marca × superior única na primeira linha e na última coluna, a outra, na diagonal principal e na segunda coluna, a marca o inferior única na última linha, i. e.,

$$y_1 = 4 - 1 = 3$$

Da mesma forma,

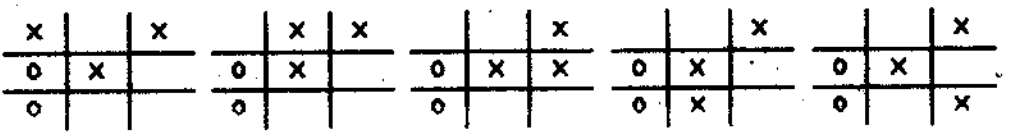
$$y_2 = 0 - 1 = -1$$
$$y_3 = 0$$

e, assim,

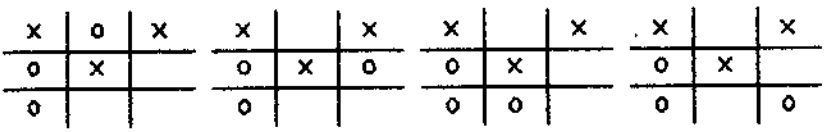
$$y = -2$$

Como o parâmetro mede a vantagem de um jogador com relaçãao ao outro, o sinal negativo indica que o jogador × está em desvantagem. De fato, o jogador o está prestes a vencer o jogo, completando a primeira coluna.

Sendo a vez do sistema, ele verifica haver cinco possíveis movimentos:

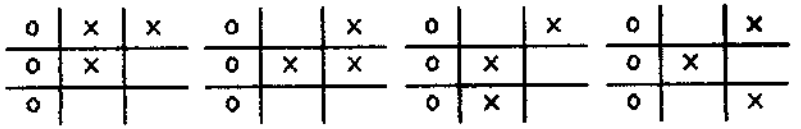


A partir do primeiro, teríamos quatro possíveis respostas do adversário:



Avaliando cada uma das quatro situações acima pelo parâmetro, teríamos os valores 5, 10, 6 e 1, respectivamente. Como o oponente joga para vencer e é, supostamente, inteligente, pode-se supor que escolherá, das quatro, a última opção, por ser a mais vantajosa, o que é indicado pelo valor menor de y. Podemos, assim, considerar esta opção como consequência lógica da primeira das cinco opções que o sistema dispõe e, portanto, associar o seu valor 1 àquela opção.

Considerando, agora, as outras quatro opções que o sistema dispõe, consideremos uma possível resposta a cada uma delas:



Temos, respectivamente, os valores -20, -20, -19 e -19. Mesmo sem analisar as outras opções que o oponente dispõe, podemos afirmar que escolherá, no mínimo tão boas como essas (na verdade, provavelmente, escolherá exatamente essas, de vez que significam vitória) e, assim, associar os valores mínimos -20, -20, -19 e -19, respectivamente às opções do sistema.

Como o sistema tem, também, o intuito de vencer, escolherá a melhor para si, qual seja, a primeira, que tem o valor mais alto do parâmetro.

Desta forma, a heurística apresentada acima levou a uma escolha acertada, sem precisar pesquisar todas as possíveis combinações de movimentos.

O exemplo acima foi talvez simples em excesso pois o jogo estava já próximo do final. Num jogo mais complicado como xadrez, a economia proporcionada seria mais flagrante.

Na página 20 veremos o método heurístico aplicado à integração analítica.

1.3 A Linguagem LISP

A linguagem LISP trabalha com listas de elementos, os quais podem ser também listas ou elementos simples (chamados *átomos*). Exemplos de listas são:

```
(A B C D)
((RENATO PIRES) (CBPF RIO))
```

Tem-se em LISP funções para manipulação de listas. Por exemplo, a *função CAR dá, como resultado, o primeiro elemento de uma lista, CDR* retorna o restante da lista, retirado o primeiro elemento; estas funções podem ser combinadas como CADR (CAR do CDR), CDDR (CDR do CDR), etc.

Tem-se também predicados lógicos, tais como: EQ que devolve o valor T (verdadeiro) se os átomos comparados são iguais e NIL (falso) se diferentes, ATOM que devolve T se o elemento analisado é um átomo e NIL caso contrário.

Tem-se ainda IF que testa uma condição e retorna o primeiro argumento fornecido se a condição for verdadeira e o segundo se falsa e COND que testa uma série de condições devolvendo o valor do argumento correspondente a primeira condição que se verificar verdadeira. Note que os comandos de LISP são, por sua vez, também listas.

Como exemplos do funcionamento dessas funções, temos

(CAR (A B C D)) resulta A (CDR (A B C D)) resulta (B C D)

(CADR (A B C D)) resulta B (CDDR (A B C D)) resulta (C D)

(CAR ((RENATO PIRES) (CBPF RIO))) resulta (RENATO PIRES)

(CDAR ((RENATO PIRES) (CBPF RIO))) resulta (PIRES)

(EQ (A A)) resulta T (ATOM (A B C D)) resulta NIL

(IF (ATOM A) A (CAR A)) resulta A

(COND (NIL v1) (NIL v2) (T v3)...) resulta v3

Como exemplo de programa em **LISP**, temos abaixo a codificação do fatorial

```
(DEFUN fact(n)
  (COND ((EQ n 0) 1)
        (T (* n (fact (SUB1 n)))) ) )
```

A declaração **DEFUN** define **fact** como uma função, de forma que o argumento **n** representa o argumento atual que será indicado pelo usuário.

O valor do argumento é testado e se for igual a 0, **fact** retornará o resultado 1, correspondendo ao fatorial de 0. Caso contrário, será usada a definição recursiva. **T** é uma condição sempre verdadeira, de forma que a definição recursiva é aplicada sempre que **n** for diferente de **0**. **SUB1** é o operador que subtrai 1 do seu argumento.

Vejamos, agora, como é possivel fazer-se uma manipulação mais sofisticada de um expressão algébrica. Consideremos um pequeno programa LISP que deriva polinomios, implementando as regras mais simples de derivação:

$$\partial(constante) = 0, \qquad \partial(variável) = 1$$

$$\partial(expr1 + expr2 + \cdots) = \partial(expr1) + \partial(expr2) + \cdots$$

$$\partial(expr1 \times expr2) = \partial(expr1) \times expr2 + expr1 \times \partial(expr2)$$

Temos, então, usando as funções **LISP** anteriormente vistas, um pequeno programa **LISP** que realiza derivações sobre polinomios:

```
(DEFUN deriva (expr var)
 (COND ((ATOM expr) (IF (EQ expr var) 1 0))
       ((EQ (CAR expr) '+) (LIST '+
             (deriva (CADR expr) var)
             (deriva (CADDR expr) var) ))
       ((EQ (CAR expr) '*) (LIST '+
             (LIST '* (deriva (CADR expr) var)
                      (CADDR expr) )
             (LIST '* (CADR expr)
                      (deriva (CADDR expr) var) ) )) ) )
```

Se aplicado este programa à expressão LISP anteriormente comentada, obteríamos

notação LISP	notação usual
(+ (+ (+ (* 1 x) (* x 1)) 1) 0)	1x + 1x + 1 + 0

Note que o resultado, embora correto, não está simplificado. A simplificação em computação algébrica está, na verdade, longe de ser um problema trivial. Não é facil caracterizar-se o procedimento de simplificação e, além disso, não há um senso comum sobre quando uma expressão está na sua forma mais simples. A "forma mais simples" pode variar, segundo o contexto do cálculo e o gosto do usuário.

Certas regras básicas, tais como

$$x + 0 \longrightarrow x \quad 0 \times x \longrightarrow 0$$
$$1 \times x \longrightarrow x$$

devem, quase sempre, ser utilizadas pelo sistema.

Em geral, deve-se ter um certo discernimento na aplicação de uma regra de simplificação, pois é frequente sua aplicação impensada levar a uma expressão mais complicada que a original.

Por exemplo, considere-se a situação abaixo,

```
11:   (x**4-y**4)/(x-y);

       3    2         2    3
      X  + X *Y + X*Y  + Y

13:   on factor;

14:   ws;

        2    2
      (X  + Y )*(X + Y)
```

O sistema algébrico simplificou a expressão original, dividindo o numerador pelo denominador, fornecendo a resposta numa forma expandida. Sob comando do usuário, o sistema reescreveu a expressão de forma fatorada.

Não se pode afirmar, porém, qual é a mais simples. Se essa expressão deve ser somada a outro polinomio, certamente a primeira é mais simples mas se deve ser dividida por outra expressão, então a segunda pode ser preferível devido a possíveis futuras simplificações entre os novos numeradores e denominadores.

Por exemplo, na expressão abaixo,

$$\frac{\left(\sqrt{x^2+a^2}+a\right)\left(\sqrt{r^2+b^2}+b\right)}{r^2} - \frac{\sqrt{r^2+b^2}+\sqrt{r^2+a^2}+b+a}{\sqrt{r^2+b^2}\sqrt{r^2+a^2}-b-a}$$

há uma simplificação possível, mas não evidente, entre os dois termos. Como se vê abaixo, quando o usuário manda abrir os parênteses e colocar toda a expressão sob um mesmo denominador, a simplificação leva a um resultado nulo.

```
(C18) exp: (sqrt(r^2+a^2)+a)*(sqrt(r^2+b^2)+b)/r^2
      -(sqrt(r^2+b^2)+sqrt(r^2+a^2)+b+a)
      /(sqrt(r^2+b^2)+sqrt(r^2+a^2)-b-a);

                 2    2                  2    2
      (SQRT(R  + A ) + A) (SQRT(R  + B ) + B)
(D18) ---------------------------------------
                          2

                         R

                              2    2          2    2
                    SQRT(R  + B ) + SQRT(R  + A ) + B + A
                  - ---------------------------------
                              2    2          2    2
                    SQRT(R  + B ) + SQRT(R  + A ) - B - A

(C19) ratsimp(exp);
TIME= 138 msec.
(D19)                          0
```

Veja-se ainda o exemplo abaixo, uma situação frequente de simplificação trigonométrica. A introdução da regra usual não é recomendável pois seu uso será a todo momento, podendo levar a expressões mais complexas. Em REDUCE não há um sistema eficiente de simplificação trigonométrica. Note-se que uma tentativa ingênua de sistema, como a do sistema abaixo, não garante bons resultados, não conseguindo, no caso, operar sobre expressões maiores.

```
1:  cos(f)**3+cos(g);

                3
          COS(F)  + COS(G)

2:  for all x let sin(x)**2+cos(x)**2=1;

3:  ws;
```

```
                                  2
      - COS(F)*SIN(F)  + COS(F) + COS(G)

4:  for all x clear sin(x)**2+cos(x)**2;

5:  operator trigsimp;

6:  forall a,c,x let
6:     trigsimp(x) = x,
6:     trigsimp(a*sin(x)**2+a*cos(x)**2+c)=
6:                        a+trigsimp(c),
6:     trigsimp(a*sin(x)**2+a*cos(x)**2)=a,
6:     trigsimp(sin(x)**2+cos(x)**2+c)=
6:                        1+trigsimp(c),
6:     trigsimp(sin(x)**2+cos(x)**2)=1;

7:  trigsimp(ws);

           3
     COS(F)  + COS(G)

8:  f*cos(g)**2+f*sin(g)**2+g*sin(g)**2
    +g*cos(g)**2+5;

           2              2              2
     COS(G) *F + COS(G) *G + SIN(G) *F

           2
     + SIN(G) *G + 5


9:  trigsimp(ws);


     F + G + 5

10:  (cos(x)**2+sin(x)**2)**2;

           4              2       2              4
     COS(X)  + 2*COS(X) *SIN(X)  + SIN(X)

11:  trigsimp(ws);

           4              2       2              4
     COS(X)  + 2*COS(X) *SIN(X)  + SIN(X)
```

```
12:  f*cos(g)**2+f*sin(g)**2+g*sin(g)**2
12:  +g*cos(g)**2+h*cos(f)**2+h*sin(f)**2;

                  2           2           2
     COS(F) *H + COS(G) *F + COS(G) *G +
```

Método geral Esta é a abordagem mais interessante, tanto do ponto de vista computacional como matemático. Quando se iniciou o desenvolvimento dos sistemas de integração analítica em computador, considerava-se que a integração não fosse um processo algoritmico como a diferenciação. Todavia, num trabalho que se inicia com Laplace, no começo do século passado, passando por Abel e Liouville, foi possível chegar- se ao algoritmo de Risch e Norman que é capaz de decidir se a integral de uma função f pertencente a um campo de funções elementares $\mathcal{F}$ pode ser escrita como

$$\int f\,dx = V_0 + \sum_i C_i \log V_i,\ (V_0, V_i \in \mathcal{F},\ C_i\ cte.),$$

e também determinar V_0, V_i e C_i.

A integral que puder ser resolvida por uma das três primeiras abordagens, o será eficientemente pois o sistema passa rapidamente por elas, tendo sucesso ou não. Mas quando não tiverem sucesso, não informarão se a integral pode ou não ser escrita na forma de uma combinação de funções elementares do campo delimitado.

Já a última abordagem permite um procedimento de decisão: se a integral não puder ser expressa em termos de funções elementares, o sistema constata essa situação e a reporta.

É interessante notar, porém, que o algoritmo de Risch e Norman, embora seja capaz de obter a expressão da integral, quando possível, exigirá considerável trabalho, exceto nos casos mais simples que podem ser resolvidos mais eficientemente pelas outras abordagens. Sendo, todavia, um processo algoritmico, essa abordagem presta-se muito bem para ser implementada em computador.

Para ilustrar as três primeiras abordagens temos os dois exemplos abaixo. O primeiro consiste na implementação (em REDUCE) de um padrão de integração, qual seja, integrais do tipo $\int x^k e^{-x}\,dx$ para $k \geq 0$

```
4:  linear intpm; operator intpm;

5:  let intpm(1,x) = x, intpm(x,x) = x**2/2;

6:  forall k such that df(k,x)=0
6:    let intpm(x**k,x) = x**(k+1)/(k+1);

7:  let intpm(e**(-x),x) = -e**(-x),
        intpm(x*e**(-x),x) =
```

```
                2              2              2
SIN(F) *H + SIN(G) *F + SIN(G) *G
```

```
13:  trigsimp(%%);

                2              2              2
COS(F) *H + COS(G) *F + COS(G) *G +

                2              2              2
SIN(F) *H + SIN(G) *F + SIN(G) *G
```

Hoje em dia, trabalha-se no desenvolvimento de procedimentos heurísticos de simplificação, que avaliem se e onde uma simplificação é possível. De qualquer forma, a pessoa mais qualificada para decidir qual a forma mais conveniente de uma expressão é o próprio usuário. O sistema algébrico deve oferecer vários recursos de simplificação, decidindo a utilização de alguns e permitindo ao usuário a utilização dos que julgar convenientes, ou mesmo impedir a sua aplicação automática.

1.4 A integração analítica por computador

Certamente, uma das áreas de desenvolvimento mais interessantes em computação algébrica é a integração. Muitas pessoas, quando ouvem falar que o computador pode realizar integração analítica, imaginam que ele tenha sido programado com uma tabela de integrais e que o sistema simplesmente faça uma espécie de busca em tabelas. Mas o que se tem, é algo bem mais sofisticado e eficiente.

Existem quatro abordagens básicas a uma integral, por parte de um sistema algébrico:

Busca a padrões O sistema faz uma inspeção da integral, procurando identificar integrais básicas que possam ser realizadas imediatamente. Esta primeira aboradagem é de certa forma, uma pesquisa em tabela.

Métodos heurísticos O sistema tenta, aqui, recursos como mudanças de variáveis e integrações por partes, até que se caia em integrais básicas que possam ser tratadas pela abordagem anterior.

Métodos especializados Aqui, utilizam-se procedimentos específicos, tais como expansão em frações parciais, no caso de funções racionais, etc.

```
                - x*e**(-x)  - e**(-x);

8:  forall k such that df(k,x)=0
8:    let intpm(x**k*e**(-x),x)  =
8:      - x**k*e**(-x)
8:      + k*intpm(x**(k-1)*e**(-x),x);

9:  intpm(x**4,x);

       5
      X  /5

10:  intpm(x**2*e**(-x),x);

           2                  X
     ( - X  - 2*X - 2)/E
```

No segundo exemplo, apresentamos a resolução de uma integral pelo programa SAINT (*"Symbolic Automatic INTegrator"*) escrito em 1963 por Slagle.

Dada a integral

$$\int \frac{x^4}{(1-x^2)^{5/2}}\,dx$$

SAINT reconhece o padrão e faz a substituição $y = \arcsin x$, obtendo

$$\int \frac{\sin^4 y}{\cos^4 y}\,dy$$

Em seguida, verifica haver três caminhos possíveis:

$$\int \tan^4 y\,dy \quad \int \cot^{-4} y\,dy \quad 32\int \frac{z^4}{(1+z^2)(1-z^2)}\,dz$$

sendo o terceiro obtido pela substituição $z = \tan y/2$.

Aqui SAINT avalia heuristicamente as opções e conclui que o caminho mais adequado é o primeiro dos três acima. Reconhecendo o padrão, faz a substituição $z = \tan y$ obtendo

$$\int \frac{z^4}{(1+z^2)}\,dz$$

Em seguida, SAINT reconhece a forma de função racional imprópria e separa a parte inteira do resto, obtendo

$$\int \left(-1 + z^2 + \frac{1}{1+z^2}\right) dz$$

e integra imediatamente, obtendo

$$-z + z^3/3 + \int \frac{1}{1+z^2}\, dz$$

Agora, SAINT considera (erradamente) que a integral restante acima é mais difícil que o segundo caminho (abandonado) da encruzilhada acima. Faz, então, a substituição $z = \cot y$, obtendo

$$-\int \frac{dz}{z^4(1+z^2)}$$

Considerando que a integral acima é mais difícil que a integral do caminho abandonado acima, SAINT retorna àquele, fazendo agora a substituição $w = \arctan z$ que resulta

$$\int dw$$

tendo sido atingida a meta.

SAINT faz as substituições inversas resolvendo a integral original como

$$\int \frac{x^4}{(1-x^2)^{5/2}}\, dx = \arcsin x + \frac{1}{3}\tan^3 \arcsin x - \tan \arcsin x$$

Capítulo 2

Sistemas de Computação Algébrica

> "Houve uma época em que a humanidade encarava o universo sozinha, sem um amigo. Agora, o homem possui criaturas para ajudá-lo; criaturas mais fortes do que ele - mais fiéis, mais úteis e absolutamente devotadas a ele. A espécie humana já não está sozinha."
>
> "Eu, robô"
> *Isaac Asimov*

Embora a computação algébrica tenha dado seus primeiros passos há cerca de 30 anos, só recentemente vem ela se disseminando e se consolidando, com aplicações nos mais variados campos de pesquisa em Ciência e Tecnologia.

Existem, hoje, vários sistemas para processar cálculos algébricos, escritos em diferentes linguagens de computador. Mais de 40 sistemas são conhecidos com esta finalidade, tendo sido, em sua maioria, criados visando aplicações específicas em problemas de Astronomia, Relatividade Geral e Física das Partículas Elementares.

Faremos, neste capítulo, uma breve descrição de alguns deles e.

2.1 Características gerais de um sistema algébrico

Como já dito, há vários sistemas algébricos, projetados para diferentes aplicações específicas. Estes sistemas se distinguem, também, pelo nível da linguagem em que estão escritos.

As linguagens de programação podem ser divididas em pelo menos três níveis:

Linguagem de Máquina Esta linguagem é constituída de instruções que são diretamente assimiladas pelo computador (máquina). Desta forma, elas são *máquina-dependentes*, isto é, dependem da marca e do tipo do computador.

Linguagem Montadora (*Assembler*) Esta linguagem, embora ainda próxima da linguagem de máquina, é mais inteligível pelo usuário precisando, no entanto, ser traduzida para aquela, por um programa chamado *compilador*, de forma que possa ser entendida pelo computador. Note-se que esta linguagem também é máquina-dependente.

Linguagem de Alto-Nível Esta é a que mais se assemelha a nossa linguagem coloquial. Na linguagem de alto- nível, os programas podem ser escritos, por exemplo, em português e com notação matemática, devendo, porém, ser traduzidos por um compilador. Em princípio, as linguagens de alto-nível são máquina-*independentes* e padronizadas internacionalmente. Como devem ser traduzidas por compiladores máquina-dependentes, são necessárias, na prática, ligeiras modificações quando se transfere um programa de uma máquina para outra, de fabricante diferente.

Como exemplo desses três níveis de linguagem, considere-se a expressão matemática

$$E = \frac{2A + 108B}{C} - D$$

Numa linguagem de alto-nível como FORTRAN, ela seria codificada como

```
E = (2*A + 108*B)/C - D
```

Numa de nível médio (*assembler*), por exemplo, como

```
LD    A      (armazene A)
MPI   =2     (multiplique por 2)
ST    TMP    (armazene temporariamente)
LD    B      (armazene B)
MPI   =108   (multiplique por 108)
ADD   TMP    (some o temporário)
DIV   C      (divida por C)
SUB   D      (subtraia D)
ST    E      (armazene em E)
```

Note que são utilizadas várias instruções elementares, sendo necessário que o programador conheça um repertório bastante grande delas. No IBM 370, por exemplo, há 256 dessas instruções.

Em linguagem de baixo nível, a codificação poderia ser, por exemplo,

```
14      1000
12      =2
15      6000
14      1009
12      =108
10      6000
13      2015
11      5282
15      6981
```

Note-se que além dos códigos das instruções não serem mais mnemônicos, não se podem mais utilizar nomes simbólicos para as variáveis, devendo o programador referenciá-las por seus endereços na memória do computador, sendo por isso necessário um conhecimento da estrutura e processos de memória da máquina em questão.

Os primeiros sistemas de computação algébrica foram baseados em linguagens montadoras e, posteriormente, nas linguagens de alto-nível FORMAC (uma extensão do FORTRAN para manipulação algébrica) e LISP. Em particular, os sistemas com linguagem de implementação LISP vêm recebendo, nos últimos anos, uma atenção muito maior do que os sistemas com base em FORMAC.

Outra característica que diferencia os sistemas é sua forma de exibição de resultados. Abaixo temos exemplos de saída de alguns sistemas no cálculo da conexão Γ^1_{01} para a métrica de Bondi:

```
GRAD        (GAM + - - 1 1 4)=-0.5E0*V(T,R,E)*R**(-2)+(0.5E0*
ASSISTANT   (D V 0 1 0)(T,R,E)+(D B 0 1 0)(T,R,E)*V(T,R,E)
            )*R**(-1)-0.5E0*EXP(-2*B(T,R,E))*EXP(2*G(T,R,E
            ))*U(T,R,E)*(D U 0 1 0)(T,R,E)*R**2-(D B 0 0 1
            )(T,R,E)*U(T,R,E))
```

```
LAM,          1              2 2G-2B           -1
 ALAM,  GAM    = -(1/2)UR  E     U   + (1/2)R   V
 CLAM,     01                   1             1
 SHEEP
                -1             -2
               +VR   B  - (1/2)VR    - UB
                     1                2
```

```
                                                  (-1)
REDUCE     GAM(1  0  1)  = V(X)*DF(B(X),R)*R

                                 (-1)                (-2)
           + 1/2*DF(V(X),R)*R          - 1/2*V(X)*R

                                                          (2*G(X)-2*B(X))
2
           - U*DF(B(X),TH)  - 1/2*U*DF(U(X),R)*E                        *R
```

```
CAMAL      K[101]  vdb[t,r,e]/dr/r
                     +1/2dv[t,r,e]/dr/r
                  -u[t,r,e]db[t,r,e]/de
                  -1/2v[t,r,e]/r^2
                  -1/2u[t,r,e]du[t,r,e]/drr^2exp[2g[t,r,e]
                  -2b[t,r,e]]
```

```
FORMAC     114
           U*((U*R*FMCEXP(G*2)*2+U*R**2FMCEXP(G*2)*FMCDIF(G,
           ------------------------------------------------
           (R,1))*2+R**2*FMCEXP(G*2)*FMCDIF(U,(R,1)))*2**(-1
           ------------------------------------------------
           )-FMCEXP(B*2)*FMCDIF(B,(E,1)))*FMCEXP(B*(-2))+(U*
           ------------------------------------------------
           R**2*FMCEXP(G*2)*FMCDIF(U,(R,1))*(-2)+U**2*R*FMCE
           ------------------------------------------------
           XP(G*2)*(-2)+U**2*R**2*FMCEXP(G*2)*FMCDIF(G,(R,1)
           ------------------------------------------------
           )*(-2)-V*R**(-2)*FMCEXP(B*2)+V*R**(-1)*FMCEXP(B*2
           ------------------------------------------------
           )*FMCDIF(B,(R,1))*2+R**(-1)*FMCEXP(B*2)*FMCDIF(V,
           ------------------------------------------------
           (R,1))*FMCEXP(B*(-2))*2**(-1)$
           ----------------------------
```

Note a grande variação nas formas de saída destes vários sistemas. A que mais se aproxima da que um pesquisador utilizaria nos seus cálculos à mão é, sem dúvida, a do SHEEP, que tem recursos para índices tensoriais covariantes e contravariantes, além de indicar derivadas por subscritos.

A forma do sistema GRAD-ASSISTANT é bem inconveniente, sem a possibilidade do uso de mais de uma linha para uma notação bidimensional. A indicação de índices superiores e inferiores é feita com sinais de + e -, a derivação é feita através do operador D, sendo necessário indicar, pelo valor 0, em quais variáveis não se está derivando. Além disso, o fator 1/2 aparece como 0.5E0, isto é, 0.5×10^0 e as potências das funções são indicadas pelo operador EXP.

O sistema REDUCE não tem tantos recursos como SHEEP mas tem uma forma razoável, bidimensional, embora não disponha de índices. Também o operador de derivação DF permite uma compreensão melhor que o do GRAD-ASSISTANT[1].

O sistema CAMAL também não dispõe de índices mas tem uma forma interessante de representar a derivação, embora seja, também, unidimensional.

Dos apresentados acima, certamente o sistema com saída mais deficiente é o FORMAC. Unidimensional, sem índices, necessita de inconvenientes operadores FMCDIF para diferenciação e FMCEXP para potências.

Talvez o sistema que tem a melhor formatação de saída seja o MACSYMA. Abaixo temos um exemplo de uma integração

```
(C27)  (LOG(I) - 1)/(LOG(I)^2 - I^2);

                                LOG(I) - 1
(D27)                           ----------
                                  2     2
                                 LOG (I) - I

(C28)  INTEGRATE(%,I);
TIME= 744 MSEC.
                      LOG(LOG(I) + I)   LOG(LOG(I) - I)
(D28)                 --------------- - ---------------
                            2                 2
```

Este sistema tem uma notação bidimensional, podendo representar com grande naturalidade funções racionais, por exemplo. Além disso, desenha símbolos de integral e somatório.

[1] O pacote EXCALC para cálculo exterior possibilita o uso de índices tensoriais e uma indicação mais conveniente para derivadas parciais, semelhante à de SHEEP

Um ponto a se ressaltar é que sistemas flexíveis como REDUCE podem ter sua forma de saída modificada pelo usuário. Um exemplo simples disso é a abolição do operador DF, substituindo-o por uma vírgula. Assim, teríamos o resultado acima escrito como

```
4:  in "procedures";
      OFF ECHO$


5:  f1(v,r,v!,r,v!,rr);

      0

6:  f1(u,r,u!,r,u!,rr);


      0

7:  f2(b,r,t,b!,r,b!,t,b!,rr,b!,rt,b!,tt);

      0

8:  gam(1,0,1):=v*df(b,r)/r+df(v,r)/2/r-v/2/r**2
8:                -u*df(b,t)-u*df(u,r)/2*e**(2*g-2*b)
8:                                              *r**2;

                                 (-1)
     GAM(1,0,1)  := 1/2*V.R*R

                                ( - 2*B + 2*G)       2
                          - 1/2*E                *U.R*R  *U

                                 (-1)
                          + B.R*R     *V - B.T*U

                                 (-2)
                          - 1/2*R     *V
```

No arquivo PROCEDURES estão os comandos necessários para essa mudança. Os operadores **F1** e **F2** são utilizados para funções de uma e duas variáveis respectivamente. O sinal ! aparece para a introdução da vírgula como parte do novo símbolo **u,r**, por exemplo, representando $\partial u/\partial r$.

2.2 Os sistemas algébricos mais em voga

Os sistemas de computação algébrica com maior atividade, no presente, são:

SCHOONSCHIP , escrito em **Assembler** é, por isso mesmo, muito rápido. É interativo, possui toda a álgebra das matrizes gama de Dirac, trabalha com espinores e é especializado para Teoria Quântica de Campos, sendo, também, utilizado para cálculos em Gravitação Quântica. A sintaxe dos seus comandos é, porém bastante difícil e dispõe de poucos recursos.

SHEEP Criado I. Frick, resultado da evolução dos sistemas LAM, ALAM, CLAM e ILAM, é um sistema interativo, bastante conversacional, tendo LISP como linguagem de implementação. Desenvolvido para ser um sistema especializado em cálculos em Relatividade Geral, SHEEP é, certamente, o mais eficiente sistema para essa aplicação.

Dada uma métrica, SHEEP pode calcular, em poucos segundos, todos os tensores de interesse da teoria, tais como os tensores de Ricci, Einstein e Weyl, independentemente da base admitida. SHEEP permite, ainda, identificar o tipo de Petrov da métrica e possibilita encontrar novas soluções das equações de Einstein.

Como sistema especializado, SHEEP emprega uma notação muito similar à utilizada pelos pesquisadores da área. Entretanto, tem a presente desvantagem de ser máquina-dependente (é implementável apenas em computadores da linha VAX, os quais não são os mais encontrados no Brasil).

CAMAL (*"CAMbridge ALgebra"*). Este sistema foi criado pela Universidade de Cambridge, Inglaterra, visando originariamente cálculos específicos em Mecânica Celeste. Recentemente, J. Wainwright, da Universidade de Waterloo, desenvolveu um estudo mais detalhado de aplicação desse sistema para o uso na Relatividade Geral e Cosmologia.

A linguagem de implementação do CAMAL é BCPL e não é interativo, i. e., seus programas só são executados em "batch". A sua versão mais difundida é para computadores da linha IBM. CAMAL é, em princípio, máquina-independente e sua notação não é tão natural quanto a de SHEEP.

REDUCE Criado, originalmente, por A. C. Hearn, na Universidade de Stanford, com linguagem de implementação LISP, é um sistema muito poderoso e em constante desenvolvimento. Difundiu-se, por isso, muito rapidamente, sendo, provavelmente, o sistema que conta hoje com o maior número de adeptos.

Como um sistema de múltiplos propósitos, REDUCE é capaz de:

- Ordenar e expandir polinomios e funções racionais
- Substituir uma grande variedade de formas algébricas
- Simplificar expressões automaticamente ou sob o controle do usuário
- Realizar cálculos simbólicos com matrizes
- Aceitar definições de novas funções e extensões de sua sintaxe
- Diferenciar e integrar analiticamente
- Resolver sistemas de equações lineares e não-lineares
- Gerar programas FORTRAN e LISP, a partir de expressões de REDUCE

Uma das vantagens do sistema REDUCE sobre os demais sistemas é a existência de várias versões para diferentes máquinas e sistemas operacionais, estando sendo preparada uma versão para micros da linha PC . Além disso, REDUCE pode operar interativamente e emprega uma notação, nas expressões matemáticas, próxima da usual.

MACSYMA (*"MAC's SYmbolic MAnipulator"*). Desenvolvido pelo Laboratório de Computação Científica do MIT, sua linguagem de implementação é, em grande parte, baseada em LISP. Como REDUCE, MACSYMA é um sistema de múltiplos propósitos bastante sofisticado, podendo ainda:

- Calcular limites e integrais definidas
- Resolver equações diferenciais simbolicamente
- Expandir funções em séries de Laurent e Taylor e calcular séries de Poisson
- Manipular vetores e tensores
- Calcular transformações de Laplace e suas inversas
- Obter formas fechadas para somas indefinidas
- Desenhar curvas e superfícies

MACSYMA é, também, um sistema interativo e emprega uma notação matemática bidimensional para os resutados, bastante próxima da usual. O uso deste sistema estava muito restrito aos membros do MIT e só recentemente tem sido comercializado pela SYMBOLICS, podendo ser adquirido por outros centros de pesquisa.

Existem, ainda, vários outros sistemas de computação algébrica em uso corrente e desenvolvimento. Para citar alguns, temos:

ORTOCARTAN , especializado para Relatividade Geral, é codificado em LISP.

POLYNOM , implementado em ALGOL, é especializado na manipulação de polinomios.

muMATH , escrito em LISP, é bastante geral e poderoso, sendo adaptado para uso em micros da linha Z80.

AVTO-ANALITIK , especializado em Física- matemática, escrito em linguagem de máquina.

SMP , geral, implementado na linguagem C.

SCRATCHPAD , geral, desenvolvido pela IBM.

ADS , implementado em FORMAC.

GRATOS , escrito em FORTRAN.

ALTRAN , geral, implementado em FORTRAN.

Capítulo 3

Introdução ao sistema REDUCE

> "Esta descoberta trará o esquecimento às almas dos discípulos porque não utilizarão suas memórias; confiarão nos caracteres escritos externos e não se recordarão por si mesmos. O método que você descobriu auxilia não a memória, mas a reminiscência. Você deu aos seus discípulos não a verdade, mas apenas o arremedo da verdade. Eles ouvirão falar de muitas coisas e nada terão aprendido; parecerão ser oniscientes e de um modo geral nada saberão; serão uma companhia aborrecida, demonstrando uma sabedoria vazia de realidade."
>
> Fedro
> *Platão*

REDUCE é um sistema algébrico bastante poderoso e versátil, capaz de:

- expandir e ordenar polinomios e expressões algébricas
- fazer substituições numa variedade de formas algébricas
- fazer simplificações automaticamente e sob controle do usuário
- trabalhar com matrizes simbólicas
- trabalhar com números com precisão arbitrária
- aceitar definições de novas funções e extensões de sintaxe

- integrar e diferenciar analiticamente
- fatorizar polinomios
- resolver sistemas de equações lineares e não- lineares
- produzir programas FORTRAN a partir de expressões REDUCE

3.1 Comandos básicos de REDUCE

Vejamos, agora, alguns comandos básicos de REDUCE.

```
1:  (3487696746599B7/64320113243)**2+67676879;

      401624743311867162765182790240/41370769675
      92343977049

2:  356*x+23*y-45*x+31*(x+y)-57*(x*y)**2/x**2/y;

      3*(114*X - Y)

3:  x:=6;

      X := 6

4:  y:=22$

5:  ws(2);

      1986

6:  clear x;

7:  ws(2);

      6*(57*X - 11)

8:  c:=(a+b)**4;

                4      3        2  2        3    4
      C := A  + 4*A *B + 6*A *B  + 4*A*B  + B

9:  d:=a**2(a+b)**2;

      D:=A**2(A+B)**$$$2;

      ***** Missing Operator

10: d:=a**2*(a+b**2;
```

```
D:=A**2*(A+B**2$$$;

***** Too few right parentheses

11:  d:=a**2*(a+b)**2;

                2   2              2
D := A *(A   + 2*A*B + B )

12:  c+d;

    4       3         2  2         3    4
2*A  + 6*A *B + 7*A *B  + 4*A*B  + B
```

Inicialmente, temos um simples cálculo numérico. Note que todo comando em REDUCE deve ser finalizado por um terminador, neste caso, um ; .

Cada comando REDUCE recebe um número; o resultado do cálculo correspondente aparece na linha seguinte (sem número nem terminador).

Vejamos, agora, um cálculo com variável indefinida. Introduzimos uma expressão algébrica que REDUCE simplifica, em seguida, deixando, porém as variáveis X e Y indefinidas.

Em seguida, atribuímos valor a uma dessas variáveis, no caso, X. A atribuição é feita através do operador :=. REDUCE responde com o resultado da atribuição, isto é, o valor atribuído, simplificado.

Atribuímos, agora, um valor a Y. Como o valor já é simplificado, não nos interessa ver o resultado da atribuição. Podemos evitar a impressão do resultado de um cálculo, usando o terminador $.

Podemos, também, recuperar o resultado de um cálculo anterior usando o comando WS (*"workspace"*), indicando o número do comando cujo resultado desejamos recuperar. Note que esse resultado sofre o efeito de quaisquer atribuições presentemente em efeito.

Podemos desfazer uma atribuição através do comando CLEAR.

Retirando a atribuição anteriormente feita a X, podemos recuperar o resultado anterior, levando em conta apenas a atribuição feita a Y.

Podemos, também, é claro, atribuir valores algébricos a uma variável.

Na linha 9, cometemos um erro de sintaxe, esquecendo de digitar o operador de multiplicação * entre o 2 e o (. REDUCE indica essa condição, repetindo o comando indicando com $$$ o ponto onde o comando deixou de fazer sentido, isto é, o erro estará à esquerda desse sinal.

Na linha 10, cometemos outro erro, esquecendo de fechar o parêntese. Isto é indicado pela mensagem correspondente.

Atribuídos valores a identificadores, podemos usá-los em expressões algébricas.

Também temos a possibilidade de repetir comandos automaticamente. Abaixo, temos um exemplo do comando FOR

```
36:  a:=0;

      A := 0

37:  for i:= 1 step 1 until 10 do a:=a+i;

38:  a;

       55

39:  for i:= 1:10 sum i;

       55

40:  array b(3);

41:  for i:=0:3 do write b(i):=2**i;

       B(0) := 1

       B(1) := 2

       B(2) := 4

       B(3) := 8
```

O índice I é local, isto é, não se relaciona a uma eventual variável chamada I fora do contexto do comando **FOR**. Obtivemos na linha 28 a soma dos inteiros de 1 a 10.

Quando o incremento do índice for 1, podemos utilizar a construção abreviada. Temos, também, a opção **SUM** que realiza a soma dos valores retornados pelo comando (no exemplo, simplesmente I), sendo desnecessária assim, a introdução da variável A.

Podemos, ainda, armazenar os valores calculados num *arranjo* (*"array"*) para futura referência, definindo-o com o comando **ARRAY**, indicando o valor máximo do seu índice (com o valor mínimo 0).

Incluímos, também, o comando **WRITE** para que os resultados calculados sejam impressos.

Já vimos, em exemplos anteriores, aparecerem algumas das funções matemáticas conhecidas por **REDUCE**. Ele conhece também algumas das suas propriedades como abaixo

```
18:  a*sin(-x)+cos(pi)*b+c*sin(5*pi)+d*tan(0);

      - (SIN(X)*A + B)

14:  a*cot(0)+b*sec(0)+c*cosec(0);
```

```
      Declare SEC operator ?  (Y or N)

? y

      Declare COSSEC operator ?  (Y or N)

? y

      COSSEC(0)*C  + SEC(0)*B

15:  a*acos(0)+b*asin(0)+c*asin(1)+d*atan(0);

      ACOS(0)*A  + ASIN(1)*C

16:  a*sinh(0)+b*cosh(0)+c*atanh(0)+d*sinh(-i*x);

        - (SINH(I*X)*D  - B)
```

REDUCE sabe, por exemplo, que $\sin(-x) = -\sin(x)$ e que $\tan(0) = 0$, conforme se pode verificar do exemplo. Mas da linha seguinte, verifica-se que não conhece as funções secante e cossecante. A menos que o modo numérico de precisão arbitrária esteja ativado, REDUCE não conhece, também, casos menos triviais das funções nem, por exemplo, quanto vale arccos(0) .

```
17:  a1*log(e)b1*log(1)+log(e**c2)+d2*exp(3/2*i*pi);

      Declare E operator ?  (Y or N)

? n

18:  a1*log(e)+b1*log(1)+log(e**c2)+d2*exp(3/2*i*pi);

      - I*D2  + C2  + A1

19:  a*erf(0)+b*dilog(0)+c*expint(0);

                                  2
      (6*EXPINT(0)*C  + B*PI  )/6

20:  forall x let sec(x) = 1/cos(x),
20:              cosec(x) = 1/sin(x);

21:  input(14);

      ***** Zero denominator
```

Na linha 17, esquecemos de digitar o operador +, tornando a expressão sem sentido. Como, em REDUCE, o argumento de uma função pode ser delimitado por parênteses ou simplesmente deixando um espaço à frente do operador, às vezes, como no caso acima, REDUCE interpreta uma sintaxe errada como um operador. Quando isso acontece, solicita confirmação.

É importante que o usuário (especialmente o principiante) se acostume a analisar cuidadosamente todas as mensagens de REDUCE. Uma mensagem ignorada pode levar a erros sérios no decorrer do cálculo.

REDUCE conhece poucas propriedades das funções **erf**, **dilog** e **expint**, como se verifica na linha 19.

Podem-se introduzir novas funções muito facilmente em REDUCE. Como exemplo disso, introduzimos, na linha 20, as funções secante e cossecante pelas suas relações com as funções seno e cosseno.

Quando tentamos refazer o cálculo da linha 14 tivemos uma divisão por zero, em função da definição da cossecante. A mensagem usual nesses casos é a que aparece acima.

3.2 Diferenciação e integração

Temos, agora, alguns exemplos simples de diferenciação e integração.

```
4:    x**(x**x)

                X
              (X  )
             X

5:    df(ws,x);

                X
              (X  + X)          2                  (-1)
             X          *(LOG(X)   + LOG(X) + X     )

      TIME: 191 MS


35:  (2*x**3+x)*(e**(x**2))**2*e**(1-x*e**(x**2))/
35:  (1-x*e**(x**2))**2;

                2
              (2*X )       2
            (E       *(2*X  + 1)*E*X)/(
```

```
                2
              (X )
            (E    *X)
          E          *

              2
            (X )          2
          (E    *X - 1) )

TIME:  1167 MS

86:  int(ws,x);

                        2
                      (X )          2
                    (E    *X)    (X )
        ( - E)/(E             *(E    *X - 1))

TIME:  2850 MS
```

A sintaxe para a diferenciação é

DF(*expressão, variável, ordem, variável, ordem,...*)

(ordem pode ser omitida se igual a um) e para integração,

INT(*expressão, variável*)

A menos que se indique que a expressão a ser derivada depende de alguma forma da variável, REDUCE supõe que é constante em relação a ela, como no exemplo abaixo

```
1:  df(x**4,y,z,2);

      0

2:  depend x,y,z;

3:  input(1);

                                                    2
    4*X*(DF(X,Y,Z,2)*X    + 6*DF(X,Y,Z)*DF(X,Z

        )*X  + 3*DF(X,Y)*DF(X,Z,2)*X   + 6*DF(

                                  2
        X,Y)*DF(X,Z)  )
```

```
4:  df(x,y,z)-df(x,z,y);

      0

5:  nodepend x,z;

6:  ws(3);

      0
```

Uma maneira de se fazer isso é com o comando DEPEND, em que indicamos que a primeira variável depende funcionalmente das seguintes no comando. No exemplo acima, indicamos que x depende de y e de z.

Note-se que REDUCE assume a comutação da ordem das derivadas.

É importante ressaltar que o módulo de integração do sistema REDUCE não é tão completo nem tão eficiente como o de outros sistemas como MACSYMA, por exemplo. Uma séria deficiência é com relação a funções envolvendo raizes quadradas.

O integrador de REDUCE também não consegue tratar de funções racionais se ele não conseguir fatorizar o denominador. No exemplo abaixo, embora o módulo fatorizador o consiga, o integrador reporta não ter conseguido fatorizar o denominador da expressão e, por conseguinte, não a integra.

```
16:  on factor;

     TIME: 62 MS

17:  (log(x)-1)/(log(x)**2-x**2);

     (LOG(X) - 1)/((LOG(X) + X)*(LOG(X) - X))


18:  int(ws,x);
     SQRTDF NOT COMPLETE

          2
     4*(X )

     THE FOLLOWING QUADRATIC DOES NOT SEEM TO FACTOR

           2      2
     (LOG(X) ) - (X )
```

```
                                    2    2
INT(LOG(X)/(LOG(X)    - X ),X) - INT(1/(

                 2    2
LOG(X)    - X ),X)
```

No exemplo seguinte, evidencia-se que alguns comandos de REDUCE são sensíveis ao estado das chaves que controlam a forma de exibição dos resultados.

```
1:  off allfac;

2:  1/(x**3+a*x**2+x);

                 2    3

    1/(A*X  + X  + X)

3:  factor x;

4:  1/(x**3+a*x**2+x);

               2    3
    1/(A*X  + X  + X)

5:  off mcd;

6:  int(ws,x);

      ***** Integration invalid with MCD off
```

3.3 Cálculos com matrizes

REDUCE oferece vários recursos para o cálculo com matrizes. Para introduzir-se uma matriz, usa-se o comando MAT, entrando a matriz linha por linha, entre parênteses e separadas por vírgulas, com os elementos de cada linha, também separados por vírgulas.

```
2:  matrix a,m;

3:  a:=mat((3,2,1),
           (0,2,0),
           (1,2,8))$

4:  trace(a);
```

```
5:  tp(a**2);

      MAT(1,1)  := 10

      MAT(1,2)  := 0

      MAT(1,3)  := 6

      MAT(2,1)  := 12

      MAT(2,2)  := 4

      MAT(2,3)  := 12

      MAT(3,1)  := 6

      MAT(3,2)  := 0

      MAT(3,3)  := 10


6:  solve(det(a-lambd*mat((1,0,0),
                          (0,1,0),
                          (0,0,1)),lambd)


      SOLN(1,1)  := 2

      SOLN(2,1)  := 4


      2

      TIME: 819 MS
7:  m:=mat((-2,-1,1),
           ( 1, 0,0),
           ( 0, 1,1))$


8:  m**(-1)*a*m;

      MAT(1,1)  := 2

      MAT(2,2)  := 2

      MAT(3,3)  := 4
```

```
TIME: 151 MS
```

Para comodidade, pode-se usar uma linha de comando para cada linha da matriz.

Temos, então, os comandos TRACE que calcula o traço, TP que calcula a matriz transposta e DET que calcula o determinante.

Na linha 6 usamos o comando SOLVE que resolve equações para obter os autovalores da matriz A. Foi necessário introduzir-se a matriz unidade. Foram encontrados os autovalores 2 e 4.

Na linha 7 introduzimos a matriz M que diagonaliza A. Na linha 8 diagonalizamos A, efetuando uma transformação de semelhança. Note que a inversa de M é obtida simplesmente elevando-a à potencia -1.

3.4 Solução de equações e de sistemas de equações algébricas

Temos, agora, o comando SOLVE para a solução de equações. A sintaxe é

SOLVE(*expressão*,*variável*)

onde a equação corresponde a *expressão* = 0 .

Temos abaixo um exemplo de SOLVE resolvendo uma equação transcendente. Note o aparecimento de constantes arbitrárias nas raizes encontradas. Elas podem ser inteiras (ARBINT), reais (ARBREAL) ou complexas (ARBCOMPLEX) e são distinguidas por um índice inteiro.

```
14:    (sin(x)-a)*(2**x-b)*(x**c-3);

                 0          X
(I - 3)*(2 - B)*(SIN(I) - A)


15:    solve(ws,x);

SOLN(1,1) := - ASIN(A) + 2*ARBINT(3)*PI

                       + PI

SOLN(2,1) := ASIN(A) + 2*ARBINT(3)*PI
```

```
SOLN(3,1)  := (LOG(B)  + 2*ARBINT(2)*I*PI)

                /LOG(2)

                 (1/0)
SOLN(4,1)  := 3      *(COS(ARBREAL(1))  +

                SIN(ARBREAL(1))*I)
```

4

O módulo **SOLVE** não consegue, é claro, resolver polinômios gerais de grau superior a quatro mas casos particulares podem ser resolvidos, embora à custa de muito trabalho à mão. **REDUCE** pode detetar alguns desses casos, fornecendo a resposta rapidamente, conforme o exemplo abaixo

```
11: x**12-4*x**11+12*x**10-28*x**9+45*x**8-68*x**7+
    69*x**6-68*x**5+45*x**4-28*x**3+12*x**2-4*x+1;

 12       11        10        9        8
X   - 4*X   + 12*X   - 28*X  + 45*X  -

    7        6        5        4        3
68*X  + 69*X  - 68*X  + 45*X  - 28*X  +

    2
12*X  - 4*X + 1

12:    solve(ws,x);

SOLN(1,1)  := ( - SQRT( - 3) - SQRT(( -

                  SQRT( - 3) - 9)*2) + 1)

               /4

SOLN(2,1)  := ( - SQRT( - 3) + SQRT(( -

                  SQRT( - 3) - 9)*2) + 1)

               /4

SOLN(3,1)  := (SQRT( - 3) - SQRT((SQRT(

                  - 3) - 9)*2) + 1)/4
```

```
SOLN(4,1)  := (SQRT( - 3) + SQRT((SQRT(
                  - 3) - 9)*2) + 1)/4

SOLN(5,1)  := ( - SQRT(( - 3*SQRT(5) - 1)
                + SQRT(5) + 3)/4

SOLN(8,1)  := (SQRT((3*SQRT(5) - 1)*2) +
              SQRT(5) + 3)/4

SOLN(9,1)  := ( - (SQRT( - 5) + SQRT( - 1
                  )))/2

SOLN(10,1)  := (SQRT( - 5) - SQRT( - 1))/
               2

SOLN(11,1)  := ( - SQRT( - 5) + SQRT( - 1
                ))/2

SOLN(12,1)  := (SQRT( - 5) + SQRT( - 1))/
               2


12

TIME:  4579 MS
```

No caso de sistemas de equações, a sintaxe é

SOLVE(LST(*lista de expressões*), LST(*lista de variáveis*))

O exemplo abaixo ilustra novamente a importância da precisão possibilitada pelos sistemas algébricos. Um sistema mal condicionado é um problema cuja solução, em se tratando de linguagens numéricas convencionais, envolve sérios problemas pois pode-se chegar a uma resposta muito diferente da exata. Um sistema algébrico como REDUCE fornece a resposta exata, como no exemplo.

```
73:   e1:=100000000000*y-333333333333*x-222222222222$

74:   e2:=200000000000*y-666666666665*x-444444444445$

76:   solve(lst(e1,e2),lst(x,y));
      SOLN(1,1)  := 1
      SOLN(1,2)  := 111111111111/20000000000

      1
```

Mesmo quando o módulo **SOLVE** não consegue achar as raízes de um polinômio, tem-se o recurso de fatorizá-lo, numa forma que se possa ter uma idéia da localização das raízes. Abaixo vemos um exemplo, em que fatorizamos um polinomio incompleto de sexagésimo grau

```
11:   x**60-1;

       60
      X   - 1

12:   on factor;


18:   us;

        16    14    10    8    6    2          8
      (X   + X   - X   - X  - X  + X  + 1)*(X

           7    5    4    3             8
        + X  - X  - X  - X  + X + 1)*(X  -

         7    5    4    3             8    6
        X  + X  - X  + X  - X + 1)*(X  - X

          4    2          4    3    2
        + X  - X  + 1)*(X  + X  + X  + X + 1

          4    3    2             4    2
        )*(X  - X  + X  - X + 1)*(X  - X  + 1

           2             2             2
        )*(X  + X + 1)*(X  - X + 1)*(X  + 1)*

      (X + 1)*(X - 1)
```

Se a memória alocada ao REDUCE for insuficiente, o módulo fatorizador, que é bastante grande, pode não ser carregado, aparecendo mensagens do tipo

```
*** (BPSMOVE nnnn) after CONDENSE
```

Neste caso, deve-se reiniciar a sessão de REDUCE e fornecer os seguintes comandos

```
symbolic condense (ssss, 25000)
(bpsmove nnnn)
(begin)
```

onde *nnnn* é o número que apareceu na mensagem e *ssss* é, geralmente, 10000.

Se, no entanto, esse problema é frequente, deve-se verificar a implementação[1].

Para mais detalhes sobre problemas de alocação de memória, consulte o manual *"Reduce user's guide for IBM 360 and derivative computers"* e *"O Controle de Alocação de memória em REDUCE"*, Renato P. dos Santos, CBPF-NT-001/87.

[1]No caso de MTS, é mais prático iniciar a sessão com o comando
$run new:reduce par=big

Capítulo 4

O Controle da exibição de expressões

Como já foi mencionado antes, é comum obterem-se expressões bastante grandes em cálculos algébricos. O problema é que esse resultado será sempre certo mas frequentemente inútil, a menos que se tenha a possibilidade de exibi-lo numa forma conveniente e recursos para analisá-la em detalhe.

REDUCE oferece facilidades nesse sentido e é muito importante que o usuário esteja bem familiarizado com elas.

4.1 Formas de apresentação de expressões

Há, em REDUCE uma série de *chaves* (*"switches"*) que controlam a forma de exibição de expressões. Elas estão listadas no apêndice D do manual e boa parte delas será descrita nestas notas.

A importância dessas chaves foi já sugerida por alguns exemplos que surgiram, como o da simplificação de expressões racionais no exemplo da página 15, em que apareceram as chaves EXP que abre parênteses em produtos de expressões e MCD que combina expressões com denominador na forma de expressões racionais sobre denominador comum.

As chaves são ativadas pelo comando ON e desativadas por OFF. Quando se inicializa REDUCE, algumas delas estarão já ativadas e outras desativadas. No manual, há a indicação da condição inicial de cada uma delas.

Uma outra chave importante para o cálculo com expressões racionais é a GCD que simplifica numerador e denominador de expressões racionais de que temos um exemplo abaixo

```
1: (2*(f*h)**2 - f**2*g*h - (f*g)**2 - f*h**3
    + f*h*g**2 - h**4 + g*h**3)/(f**2*h - f**2*g
    - f*h**2 + 2*f*g*h - f*g**2 - g*h**2 + g**2*h);

  2  2    2          2  2      2
(F *G  + F *G*H - 2*F *H  - F*G *H

              3      3    4    2       2
    + F*H  - G*H  + H )/(F *G - F *H

          2                    2    2
    + F*G  - 2*F*G*H + F*H  - G *H

         2
    + G*H )
```

```
2: on gcd; ws;

                              2
(F*G + 2*F*H + H )/(F + G)
```

Não é evidente mas há um fator em comum entre o numerador e o denominador da expressão inicial. Quando o módulo GCD é ativado, ele deteta esse fator comum e retorna a expressão na forma simplificada.

Esse módulo não fica ligado permanentemente pois exige um esforço computacional apreciável, sendo mais econômico que o usuário o ative, quando necessário. Além disso, sua atuação pode retardar as etapas intermediárias dos cálculos.

Em geral, é mais eficiente realizar o cálculo e solicitar a simplificação do resultado. Mas só a prática permite discernir quais chaves e quando devem ser ativadas ou desativadas.

É importante ressaltar que estado de ativação de algumas chaves pode influenciar o funcionamento de alguns módulos e mesmo de outras chaves, conforme exemplo abaixo e outros que veremos à frente.

```
4:  1/(x**3+a*x**2+x);

5:  off mcd;

6:  int(ws,x);

      ***** Integration invalid with MOD off
```

No exemplo abaixo, temos um exemplo de que nem sempre GCD melhora os resultados. É, de fato, também mais um exemplo de que a "forma mais simples" para uma expressão não é algo trivial de se definir.

```
17:   (f**10-g**10)/(f**2-f*g);

        10     10     2
(F    - G  )/(F  - F*G)

18:   on gcd,factor;

19:   ws;

     4    3       2  2       3    4    4
((F  + F *G + F *G  + F*G  + G )*(F  -

      3       2  2       3    4
 F *G + F *G  - F*G  + G )*(F + G))/F
```

No exemplo seguinte, faremos uma apresentação detalhada do efeito de várias chaves.

```
            REDUCE 3.1, 15-Apr-84 ...

1:  linelength(55);

      72

2:  f**2*(g**2+2*g)+f*(g**2+h)/(2*f1);

                  2                      2
     (F*(2*F*G  *F1 + 4*F*G*F1 + G  + H))/(2*F1)

3:  off mcd; ws;

                  2                      2
     (F*(2*F*G  *F1 + 4*F*G*F1 + G  + H))/(2*F1)

5:  input(2);

          2                  2   (-1)          (-1)
     F*(F*G  + 2*F*G + 1/2*G  *F1     + 1/2*H*F1     )

6:  off exp; input(2);

            2         (-1)
     (1/2*(G  + H)*F1     + (G + 2)*F*G)*F

8:  off allfac; input(2);

            2         (-1)
     (1/2*(G  + H)*F1     + (G + 2)*F*G)*F

10:  on exp,mcd; input(2);

       2  2          2           2
     (2*F  *G *F1 + 4*F  *G*F1 + F*G  + F*H)/(2*F1)
```

Após o cabeçalho padrão, indicando a versão em uso de REDUCE e a sua data de implementação, ajustamos o comprimento da linha de saída para 55 caracteres, para melhor apresentação dos resultados nestas notas, retornando o valor anterior (inicial) de 72 caracteres.

Em seguida introduzimos uma expressão, respondendo REDUCE com a mesma expressão na forma correspondente ao estado inicial das chaves. Observe-se que os parênteses foram abertos e que o denominador do último termo foi feito comum a toda a expressão.

Desligando a chave MCD tentamos recuperar a forma inicial, eliminando o denominador comum. Não há alteração na forma da expressão de vez que o comando WS apenas recupera o resultado anterior. Para eliminar o denominador comum, precisaríamos também ordenar a simplificação do numerador com o denominador, através do módulo GCD.

Por outro lado, se reintroduzirmos a expressão com aquela chave desligada, REDUCE não mais a colocará sob denominador comum. Fazemos isso usando o comando INPUT, referindo-nos á linha de comando número 2. Note o aparecimento do denominador como fator com potência negativa.

Agora, desligamos chave EXP para tentar manter os parênteses. Obtendo resultado, verificamos que alguns fatores, tais como F e G no segundo termo foram colocados em evidência.

Desligando a chave correspondente, ALLFAC, não obtemos resultado porque o funcionamento dessa chave sofre influência do estado das chaves EXP e MCD. Desligando-as, observamos o efeito desejado (compare os resultados 2 e 10).

Não conseguimos ainda recuperar a forma inicial da expressão mas, prosseguindo com o exemplo, verificamos o efeito de outras chaves.

```
12:  on allfac; input(2);

                    2                          2
     (F*(2*F*G  *F1 + 4*F*G*F1 + G  + H))/(2*F1)

14:  on dif; input(2);
       *** !*DIF DECLARED FLUID

                    2                          2
     (F*(2*F*G  *F1 + 4*F*G*F1 + G  + H))/(2*F1)

16:  on div; input(2);

                 2                    2   (-1)              (-1)
     F*(F*G  + 2*F*G + 1/2*G  *F1      + 1/2*H*F1      )

18:  off div; factor f; input(2);

                 2                          2
     (2*F  *G*F1*(G + 2) + F*(G  + H))/(2*F1)

21:  factor g; input(2);

                 2  2            2                2
     (2*F  *G  *F1 + 4*F  *G*F1 + F*G  + F*H)/(2*F1)
```

Ligando novamente a chave ALLFAC, retornamos à configuração inicial, como se pode constatar, comparando os resultados 12 e 2.

A chave DIV, quando ligada, faz com que fatores simples da expressão sejam divididos, de forma a aparecerem fatores com potência negativa ou fracões racionais.

Quando fomos ativá-la, erramos (propositalmente) sua digitação, digitando "dif" . Quando tentamos ativar uma chave inexistente, REDUCE responde com a mensagem acima.

Ligando agora a chave, observamos o denominador aparecendo como fator com potência negativa, de forma idêntica à obtida em 5, quando desligamos a chave MCD. Esta chave não tem o mesmo efeito que MCD. Ilustraremos este ponto com mais detalhe futuramente.

Por vezes, para se analisar a estrutura de uma expressão, é interessante escrevê-la como um polinomio em uma das variáveis. Isto pode ser feito declarando a variavel desejada através do comando FACTOR. *Não confundir com a chave* FACTOR, *que ativa o módulo fatorizador.*

Declarando a variável F, a expressão é escrita em dois termos, um na potência 2 de F e o outro na primeira potência.

Declarando a variável G também, os coeficientes das potências de F passam a ser escritos também como polinômios em G. Neste caso, isso fez com que a expressão fosse totalmente expandida, mantendo-se porém os termos na ordem decrescente das potências de cada variável, em sua ordem de precedência.

```
23:  remfac g; input(2);

                  2                          2
     (2*F *G*F1*(G  + 2) + F*(G  + H))/(2*F1)

25:  on rat; input(2);

                2                        2
     F *G*(G + 2) + F*(G  + H)/(2*F1)

27:  off allfac; input(2);

               2   2                     2
     F *(G  + 2*G) + F*(G  + H)/(2*F1)

29:  on list; ws;
```

```
 2   2
F *(G

        + 2*G)

      2
+ F*(G

          + H)/(2*F1)
```

Podemos remover a declaração de uma variável com o comando REMFAC. Retirando a declaração de G, obtemos o mesmo resultado que em 18, como seria de se esperar.

A chave RAT é interessante, em conjunção com o comando FACTOR. Quando uma variável foi declarada através deste comando, estando a chave RAT ligada, o denominador da expressão aparece dividindo cada coeficiente.

No caso, isto fez com que o denominador do segundo termo da expressão original aparecesse dividindo-o, de vez que aquela foi introduzida como um polinômio em F.

Para recuperar a forma original, basta agora desligar a chave ALLFAC para que G não seja colocada em evidência.

Outra chave de algum interesse para a análise de uma expressão, é a LIST que lista os termos de uma expressão em linhas sucessivas. Este comando seria mais interessante se se pudesse indicar o nível de profundidade para o desmembramento.

Mais à frente, retomaremos este exemplo para demonstrar outros recursos que REDUCE oferece para estudar-se a estrutura de uma expressão.

Conforme vimos na página 5 REDUCE armazena números racionais na forma de quocientes de números inteiros. Desde que a chave FLOAT esteja desligada, números racionais serão convertidos a essa forma, como no exemplo abaixo

```
1:  off float;

2:  12.35*g;

    *** 1.235000E1  REPRESENTED BY 247/20

    (247*G)/20
```

A chave BIGFLOAT permite, porém, trabalhar com números em forma decimal com precisão arbitrária. A precisão desejada é estabelecida pelo comando PRECISION. Note-se, porém que não se pode introdusir números com precisãom arbitrária.

Acreditamos, no entanto, que isto se deve a alguma falha da presente versão de REDUCE, a ser corrigida em versões futuras.

```
8:      3.405**2.296;

      16.66 24966 05397 87301 8

9:  a*sin(0.19)+b*atan(87)+c*exp(0.19)+d*log(87);

      0.188 85879 67760 76121 97*A

       + 1.559 30258 00798 66098 8*B

       + 1.209 24947 67322 97738 8*C

       + 4.465 90811 86545 83718 6*D

10:  a*sinh(0.19)+b*erf(0.87)+c*dilog(0.19)
10:  +expint(0.87);

      ERF(0.869 9999)*B

       + EXPINT(0.869 9999)

       + DILOG(0.189 9999)*C

       + SINH(0.189 9999)*A

11:  0.19;

       1.899999E-1
```

Em REDUCE, a potência 1/2 é, normalmente, indicada pelo operador SQRT, correspondendo à raiz quadrada. No exemplo abaixo, podemos observar essa mudança de notação quando definimos a variável Y como X elevado a 1/2. Note, na linha 2, que REDUCE não racionaliza o denominador.

Se, no entanto, desligamos a chave MCD, o resultado da linha 2 é escrito como potência negativa mas não do operador SQRT.

Note-se, porém, que na linha 5 o quociente não foi simplificado, o mesmo ocorrendo na linha 6, ocorrendo, ainda, o aparecimento do operador SQRT.

No exemplo abaixo, temos, inicialmente, a conversão inversa à do exemplo anterior. Em seguida, tentamos um cálculo numérico e notamos que ainda não é possível. Para isso, precisamos ligar, também, a chave NUMVAL.

Como no exemplo da página 6, a variável E que representa a base dos logaritmos naturais passa a ter seu valor numérico. Agora, o cálculo da linha 4 pode ser realizado numericamente.

```
1:  on bigfloat;

2:  precision(20);

      20

3:  247/20;

      12.35

4:  atan(sqrt(3));

            0.5
      ATAN(3   )

5:  on numval;

6:  e;

      2.718 28182 84590 45235 4

7:  input(4);

      1.047 19755 1
```

Da mesma forma, podemos elevar números reais a potências reais. Além disso, as funções trigonométricas, exponencial e logaritmo podem ser calculadas numericamente. As funções hiperbólicas, dilog, etc., porém, continuam indefinidas numericamente, salvo os casos particulares, como visto na página 38.

É curiosa a aproximação dos números decimais efetuada por REDUCE nas linhas 10 e 11, indo contra a decantada precisão absoluta dos sistemas de computação algébrica.

O produto da linha 7 também não é escrito como potência 3/2, embora, como se vê da linha 8, internamente, o valor matematico esteja correto.

```
1:  y:=x**(1/2);

      Y := SQRT(X)

2:  1/y;

      1/SQRT(X)
```

```
3:  off mcd;

4:  1/y;

          ( - 1/2)
        X

5:  x/y;

          ( - 1/2)
        X         *X

6:  y/x;

                   (-1)
       SQRT(X)*X

7:  x*y;

       SQRT(X)*X

8:  ws**2;

       X**3
```

Essa forma de apresentação é inconveniente e, frequentemente, exige maior atenção do usuário para interpretar resultados. Pode-se, entretanto, alterar isso ordenando as substituições correspondentes, conforme se vê na linha 9. Os resultados das linhas 10 e 11 parecem mais legíveis.

Por outro lado, essa substituição não é aceita se a chave MCD estiver ligada pois o comando LET, do qual falaremos mais na página 75, não aceita divisão na definição do padrão a ser substituído. Com a chave desligada, a divisão é transformada para multiplicação por potência negativa.

```
9:  let x**(1/2)/x=1/x**(1/2),x/x**(1/2)=x**(1/2);

10: y/x;

          ( - 1/2)
       X

11: x/y;

       SQRT(X)

12: clear x**(1/2)/x,x/x**(1/2);
```

```
13:  on mcd;

14:  let x**(1/2)/x=1/x**(1/2),x/x**(1/2)=x**(1/2);

15:  y/x;
  *** PUSHDOWN STACK OVERFLOW
  *   76

16:  clear x**(1/2)/x,x/x**(1/2);
  *** PUSHDOWN STACK OVERFLOW
  *** PUSHDOWN STACK OVERFLOW
  *   #000000000X
```

Ainda com relação ao operador **SQRT**, há a chave **REDUCED** que controla a raiz quadrada de produtos. Como se vê do exemplo abaixo, quando ligada, a raiz de produtos é escrita como produto de raizes.

Considerando-se agora a sequência iniciada na linha 7, vê-se, na linha 10, que $\sqrt{-1}$ foi substituída por i e não foi levada em conta a raiz $-i$.

Da mesma forma, nas linhas 14 e 16 não foram consideradas as segundas raízes, chegando a resultados diferentes para a mesma expressão.

É, portanto, necessário muito cuidado com cálculos envolvendo raízes pares em REDUCE por este não levar em conta ambiguidades de sinal.

```
2:  let c=sqrt(a*b);

3:  off reduced; c;

      SQRT(A*B)

5:  on reduced; c;

      SQRT(B)*SQRT(A)

7:  let a=-1;

8:  off reduced; c;

      SQRT( - B)

10:  on reduced; c;

      SQRT(B)*I
```

```
12:  clear a; let b=-1;

14:  off reduced; us(9);

      1

16:  on reduced; us(11);

      (-1)
```

Como vimos na página 44, o módulo SOLVE introduz constantes arbitrárias nas soluções encontradas. Temos, abaixo, mais um exemplo.

```
1: solve((sin x-a)*(2**x-b)*(x**c-8),x);

     SOLN(1,1) :=  - ASIN(A) + 2*ARBINT(6)*PI

                   + PI

     SOLN(2,1) := ASIN(A) + 2*ARBINT(6)*PI

     SOLN(3,1) := (LOG(B) + 2*ARBINT(5)*I*PI)

                  /LOG(2)

                   (1/C)
     SOLN(4,1) := 3     *(COS(ARBREAL(4)) +

                    SIN(ARBREAL(4))*I)

      4
```

Caso o usuário queira apenas os ramos principais das raízes, basta desligar a chave ALLBRANCH, conforme abaixo. Desaparecem, então, as constantes arbitrárias e, no caso, reduz-se a três raízes.

```
2: off allbranch;

3: solve((sin x-a)*(2**x-b)*(x**c-3),x);
```

```
SOLN(1,1)  := ASIN(A)

SOLN(2,1)  := LOG(B)/LOG(2)

                 (1/0)
SOLN(3,1)  := 3


3
```

Esse recurso de introdução de constantes permite a REDUCE resolver formalmente sistemas singulares, obtendo soluções arbitrárias.

Se o usuário desejar que REDUCE interrompa a execução com mensagem de erro quando o sistema proposto for singular, é necessário desligar a chave SOLVESINGULAR, como se vê do exemplo abaixo.

```
14:  e1:=x+y$

15:  e2:=2*x+2*y$

16:  solve(lst(e1,e2),lst(x,y));
      SOLN(1,1)  :=  - ARBCOMPLEX(1)

      SOLN(1,2)  := ARBCOMPLEX(1)


      1

17:  off solvesingular;  input(16);


      ***** SINGULAR MATRIX
```

4.2 Salvando expressões para uso posterior

Em computação algébrica, frequentemente, as expressões obtidas como resultado são bastante grandes e, além disso, por vezes é necessario interromper-se um cálculo para continuá-lo em outra ocasião.

É importante, então, dispor-se de recursos para armazenar resultados em um meio permanente e para recuperar esses resultados, continuando o cálculo. Na sequência seguinte, mostraremos como isso pode ser feito em REDUCE.

Partindo do exemplo já visto na página 5 de geração do Polinomio de Legendre P_{16}, criamos anteriormente um arquivo, que denominamos *LEG16*, em disco magnético, que a armazenará.

```
      REDUCE 3.1, 15-Apr-84 ...

1:  linelength(45);

      72

2:  procedure pl(n,x);
2:  df((x**2-1)**n,x,n)/2**n/factorial(n);

      PL

3:  factor x; on div,rat;


5:  pl(16,x);

                        16                          14
   300540195/32768*I     - 145422675/4096*I

                          12
    + 456326325/8192*I     - 185910725/4096*

    10                          8
   X   + 334639305/16384*I     - 20369349/4096

     6                      4                    2
   *I  + 4849845/8192*I     - 109395/4096*I

    + 6435/32768
```

A notação usual de REDUCE para a impressão de resultados (com expoentes elevados, por exemplo), embora confortável para o usuário, é incompreensível para REDUCE, o qual só aceita entrada em notação unidimensional.

Para que a expressão seja armazenada no arquivo numa notação que faça sentido para REDUCE numa ocasião posterior, é necessário desligar a chave NAT.

Em seguida, devemos, com o comando OUT, dirigir o resultado de quaisquer comandos posteriores para o arquivo em questão.

Agora, basta recuperar o resultado obtido anteriormente e atribuí-lo a um identificador para que REDUCE possa recuperá-la convenientemente, no futuro, bem como o usuário referir-se a ela.

O resultado da atribuição é escrito no arquivo especificado e não aparece, por isso, na tela.

Por questões internas de REDUCE, é, também, aconselhável gravar, no fim do arquivo, a sentença ;**end**; .

Para que REDUCE volte a escrever na tela, devemos dirigir a impressão a ela com o comando OUT referindo-se a T (*terminal*).

```
6:  off nat; out "leg16";

8:  pl16:=ws(5);
9:  write ";end;";
10:  out t; on nat;


12:  quit;
```

Saímos temporariamente ao sistema operacional, com o comando QUIT e verificamos o conteúdo do arquivo. Note que além dos expoentes terem aparecido em notação normal de entrada em REDUCE, a expressão foi, também, por efeito do desligamento da chave NAT, terminada por $.

```
PL16 := 300540195/32768*I**16 -
45422675/4096*I**14 + 456326325/8192*I
**12 - 185910725/4096*I**10 + 334639305/
6384*I**8 - 20369349/4096*I**6 +
849845/8192*I**4 - 109395/4096*I**2 +
435/32768$

;END;$
```

Poderíamos, também, estar interessados em utilizar a expressão obtida de P_{16} para fazer algum cálculo numérico em FORTRAN. Talvez desejássemos transformá-la numa *"function"* para ser incorporada a algum programa que, por exemplo, faça gráficos.

Programas FORTRAN podem ser facilmente obtidos de REDUCE, conforme se verá abaixo.

Criado um arquivo para conter a expressão desejada, denominado *LEG16.FOR*, retornamos a REDUCE, da mesma forma que antes.

Para se obter, automaticamente, uma expressão na notação FORTRAN, basta ligar a chave FORT.

A versão de FORTRAN para a qual essa interface de REDUCE foi codificada admite linhas de comando com comprimento grande (132 caracteres) e também um número maior de linhas de continuação que versões mais comuns.

Devemos, então ajustar o número de linhas de continuação atribuindo o valor maximo desejado à variável CARDNO* e o comprimento máximo a FORTWIDTH* (o sinal ! é necessário para introduzir-se o caráter *

como parte do nome da variável).

Podemos incluir no arquivo, também, os demais comandos necessários, de forma a gerar programas FORTRAN "automaticamente".

```
14:  on fort; out "leg16.for";

16:  cardno!* := 4 $ fortwidth!* := 45 $
18:  write "      FUNCTION PL16(X)" ;
19:  pl16:=ws(5);
20:  write "      RETURN" ;
21:  write "      END" ;
22:  out t; off fort;


24:  quit;
```

Examinando o arquivo, vemos que REDUCE partiu a expressão inicial em subexpressões intermediárias menores, para conformar aquela ao limite imposto de linhas de continuação (compare com o arquivo anterior).

Nota-se, também, que REDUCE iniciou cada linha na coluna 7, exceto as linhas de continuação que têm o caráter . na coluna 6 e são deslocadas para facilitar a visualização. Além disso, colocou o ponto decimal, transformando os inteiros em reais.

```
>       FUNCTION PL16(X)
>       ANS2=4849845./8192.*X**4-109395./
>      . 4096.*X**2+6435./32768.
>       ANS1=-185910725./4096.*X**10+
>      . 334639305./16384.*X**8-20369349./
>      . 4096.*X**6+ANS2
>       PL16=300540195./32768.*X**16-
>      . 145422675./4096.*X**14  + 456326325.
>      . /8192.*X**12+ANS1
>       RETURN
>       END
```

A forma armazenada da expressão corresponde exatamente à original mas é muito ineficiente numéricamente. Todavia, a forma em que uma expressão será gravada é influenciada por todas as chaves de controle usuais e uma forma melhor pode ser obtida, bastando ajustá-las antes de enviá-la para o arquivo.

Apesar disso, nem sempre se chega à forma mais eficiente computacionalmente. Outros sistemas algébricos têm interfaces mais eficientes que otimizam, de fato, a expressão FORTRAN.

4.3 Analisando a estrutura da expresssão

Retornando, agora, ao exemplo da página 55, apresentaremos alguns comandos que permitem alguma análise de expressões algébricas.

Desligando a chave LIST, que não foi de muita ajuda, comandamos STRUCTR que exibe a expressão na forma de uma árvore, conforme se vê da sequência abaixo

```
31:  off list,exp; on structr(input(2));

      (F*ANS3)/(2*F1)

        WHERE

           ANS3 := 2*F*G*ANS2*F1  +ANS1

           ANS2 := G + 2

                     2
           ANS1 := G  + H

33:  off mcd; structr(input(2));

     F*ANS3

        WHERE

                                          (-1)
           ANS3 := F*G*ANS2 + 1/2*F1     *ANS1

           ANS2 := G + 2

                     2
           ANS1 := G  + H

35:  ans2;

     ANS2

36:  on savestructr; array a(6); structr(input(2),a);

      F*A(3)

         WHERE
```

```
                                                    (-1)
A(3) := F*A(2)*G + 1/2*A(1)*F1

A(2) := G + 2

                  2
A(1) := G  + H

39:  a(2);

      G + 2
```

Como se vê da linha 33, a estrutura é dependente da presente configuração de chaves. Além disso, da linha 35, vê-se que os elementos da estrutura não são, normalmente, armazenados e, por isso, não podem ser referenciados.

Fazendo uso da chave SAVESTRUCTUR, temos a estrutura preservada. Podemos, também exigir que os elementos sejam armazenados num arranjo, fornecendo o nome do arranjo ao comando.

Temos, também, comandos que fornecem o numerador (NUM) ou o denominador (DEN) de uma expressão.

```
40:  on mcd; num(input(2));

                2
     F*((G  + H) + 2*(G + 2)*F*G*F1)

42:  den(input(2));

      2*F1

43:  f*f1*g;

      F*G*F1

44:  order f,f1,g; ws;

      F*F1*G

46:  ws(41);

                2
     F*((G  + H) + 2*(G + 2)*F*G*F1)
```

```
47:  order nil; ws(45);

     F*G*F1

49:  factor x; x**3+a*x**2+b*x+c;

       3    2
     X  + X *A + X*B + C

51:  coeff(ws,x,coex);

     *** COEX3 COEX2 COEX1 COEX0 are non zero

     3

52:  coex2;

     A
```

Um ponto que não havíamos levantado antes é a questão da ordem em que fatores aparecem num produto na impresão de um resultado por REDUCE.

A ordem é alfabética, por ordem crescente de caracteres no identificador. Assim, o produto da linha 43 tem os dois últimos fatores permutados pois G tem um caráter e F1, dois.

Podemos, porém, escolher uma ordem específica para os fatores que desejarmos através do comando **ORDER**. Assim, impondo a ordem da linha 44, temos a expressão da linha 43 reescrita na ordem desejada.

Note-se, porém, que em alguns casos essa ordem é ignorada, como na linha 46. Removemos um ordem através do parâmetro NIL para o comando ORDER.

Outro comando interessante é o **COEFF** que fornece os coeficientes de um polinômio, numa variável especificada, armazenando-os em variáveis cujos identificadores iniciam pelo nome especificado. O comando indica quais desses coeficientes não são nulos e fornece, ainda, a potência máxima verificada na variável.

Esse comando é, porém, dependente da configuração de chaves. Por exemplo, considere-se a mudança de variável abaixo.

```
53:  factor y; on rat; sub(x=y-a/3,input(3));
```

```
        3          2                  3
     Y  + Y*( - A  + 3*B)/3 + (2*A  - 9*A*B

         + 27*C)/27
```

```
56:  coeff(ws,y,coey);

                3      2
     ***** (2*A  - 9*A *Y - 9*A*B + 27*B*Y +

                          3
           27*C + 27*Y )/27 invalid as

     POLYNOMIAL
```

```
57:  off mcd; sub(x=y-a/3,input(3));

      3              2              3

     Y  + Y*( - 1/3*A  + B) + 2/27*A  - 1/3*A

     *B + C
```

```
59:  coeff(ws,y,coey);

     *** COEY3 COEY1 COEY0 are non zero
```

```
60:  coey0;

           3
     2/27*A  - 1/3*A*B + C
```

O resultado tem a forma de um polinomio. No entanto, o comando COEFF não funciona para essa expressão, dizendo que não é um polinomio.

Observando-se a forma em que a expressão foi impressa na mensagem, verifica-se que é a forma de uma expresão racional, com denominador comum. Isso se deve ao fato que a forma exibida de uma expressão pode não corresponder à forma "interna" de armazenamento.

Há chaves que alteram a forma apenas externamente, para facilitar a leitura, e outras que alteram-na internamente, mudando mesmo o caráter de função racional para polinomio, etc. Tal é o caso da chave MCD.

Não deixa de ser útil para analisar a estrutura de uma expressão, fatorizá-la.

O módulo fatorizador de REDUCE já apareceu no exemplo da página 47, onde foi ativado pela chave FACTOR. Essa forma de ativação faz com

que todas os resultados futuros sejam apresenteados na forma fatorizada.

Para fatorizar localmente uma expressão, utilizamos o comando FACTORIZE, que, de forma pareciada ao comando COEFF, armazena os resultados em variáveis com identificadores iniciados pelo nome indicado e reporta quantos fatores foram encontrados. Note-se que, frequentemente, o último fator é, simplesmente, igual a 1 .

```
2:  factorize(a**4-b**4,fat);
      *** LOGAND REDEFINED
      *** LOGOR REDEFINED

      *** FAT3 FAT2 FAT1 FAT0 are non zero

      3


3:  fat3;

       A - B

4:  fat1;

         2    2
       A  + B

5:  fat0;

       1
```

Capítulo 5

Atribuições e substituições

Para quem já tem alguma experiência em computação, ainda que numérica, a atribuição de valores a uma variável não é, com certeza, novidade.

Já o conceito de substituição pode ser novidade e é, de fato um recurso muito útil em computação algébrica.

Neste capítulo, além de tentar transmitir o conceito de ambas e caracterizar bem sua distinção (coisa que pode não ficar bem clara ao iniciante em REDUCE), tentaremos capacitar o usuário a tirar bom proveito delas.

5.1 Atribuições e substituições simples

Exemplos de atribuições apareceram já à página 35. Lá se viu que a sintaxe é

identificador := *expressão terminador*

onde *identificador* é o "nome" da variável, cujo conteúdo passará a ser o resultado simplificado da avaliação da expressão (levando-se em conta todas as chaves substituições e outras atribuições em efeito,) e *terminador* pode ser ;, fazendo com que seja impresso na linha seguinte

identificador := *resultado simplificado*

ou $, quando nada será impresso, sendo, porém, a atribuição realizada de qualquer forma.

Antes de ser atribuído valor a uma variável, situação em que é chamada "indefinida", seu conteúdo estará associado ao seu nome. É claro que, em computação algébrica, ao contrário da numérica, uma variável indefinida é perfeitamente válida numa expressão.

O identificador é constituido por caracteres alfanuméricos, sendo o primeiro alfabético, e pode conter até 24 caracteres, em geral. Caracteres especiais podem ser incluídos, até mesmo como iniciais, desde que precedidos pelo sinal !. Exemplo disso apareceu à página 28, onde criamos o identificador dv,r . Caracteres minúsculos são transformados em maiúsculos, a menos que a chave RAISE esteja desligada.

Como já foi visto, também, a atribuição de um novo valor substitui o conteúdo anterior e uma variável pode voltar a ser indefinida pelo comando CLEAR (vide exemplo da seção 3.1).

É preciso certo cuidado, porém, quando se faz uma atribuição. Considere-se o exemplo acima citado, em que a variável C tinha o conteúdo

```
   4       3          2  2         3     4
  A  + 4*A *B + 6*A *B  +4*A*B   + B
```

Se, agora, realizamos a substituição abaixo

```
13:  a:=a+1;

     A := A + 1

14:  c;
  *** PUSHDOWN STACK OVERFLOW
  *   A

15:  clear a; sub(a=a+1,c);

      4       3          3       2  2         2
     A  + 4*A *B + 4*A  + 6*A *B  + 12*A *B

           2          3          2
      + 6*A  + 4*A*B  + 12*A*B  + 12*A*B + 4*

          4       3       2
     A + B  + 4*B  + 6*B  + 4*B + 1
```

de vez que a variável A é indefinida, a atribuição é repetida no conteúdo de C até a exaustão da memória.

Se desejamos, realmente, obter o resultado da substituição de A por A+1 no contúdo de C, devemos usar o comando SUB que a realiza apenas uma vez e somente sobre a expressão ou conteúdo indicado.

A sintaxe desse comando é

SUB(*substituição, substituição,..., expressão*)

e o resultado do comando é o das substituiçoes, simplificações e chaves em efeito.

No exemplo abaixo, fazemos uso deste comando numa tentativa ingênua de simulação de integração definida.

```
50:  indef:=int(cos(x)**2,x);

     INDEF  := (COS(X)*SIN(X)  + X)/2

51:  sub(x=pi  , indef) - sub(x=0  , indef);

     PI/2
```

Note que, se usamos em *expressão* o nome de uma variável, as substituições levam em conta o seu conteúdo mas não o alteram, isto é, após a execução do comando, o conteúdo da variável continua o mesmo.

As substituições podem, também, ser introduzidas como "regras", a serem utilizadas em todas as expressões futuras onde couberem.

Isso é feito através do comando LET, conforme já visto à página 38, onde implementamos uma regra de substituição para as funções SEC e COSSEC, na prática, definido-as.

O comando LET, introduzindo regras de substituição, é um dos mais importantes e úteis de REDUCE e sua utilização deve ser bem compreendida por quem pretender tirar o máximo proveito deste sistema algébrico.

Na sequência seguinte, procuramos deixar claro o que vem a ser uma substituição, em oposição a uma atribuição.

```
1:  termo2:=cont1;

      TERMO2  := CONT1

2:  somaatr:=termo1+termo2;

      SOMAATR  := TERMO1 + CONT1

3:  let somasub=termo1+termo2;

4:  somasub;

      TERMO1 + CONT1

5:  termo2:=cont2;

      TERMO2  := CONT2

6:  somaatr;

      TERMO1 + CONT1

7:  somasub;

      CONT2 + TERMO1
```

Consideremos a soma de dois termos **TERMO1** e **TERMO2**, tendo sido o segundo definido, atribuindo-se a ele o conteúdo **CONT1**. Essa soma foi atribuída a uma variável **SOMAATR** e definida como o valor a ser substituído para a variável **SOMASUB**.

Observe-se dos resultados das linhas 2 e 4 que o conteúdo de **TERMO2**, **CONT1**, foi utilizado e, como **TERMO1** é indefinido, foi utilizado seu nome.

Na linha 5, alteramos o conteúdo de **TERMO2** mas, das linhas 6 e 7, vemos que o conteúdo de **SOMAATR** não se altera, enquanto o de **SOMASUB**, sim.

Isso ocorre, naturalmente, porque, na atribuição, calcula-se o valor da expressão, utilizando-se os conteudos atuais das variáveis envolvidas, e atribui-se-o à variável.

Mesmo que haja uma alteração posterior no conteúdo de alguma das variáveis que foram utilizadas naquela expressão, esta não será avaliada novamente para atualizar o conteúdo da variável que foi atribuída.

Já no outro caso, tem-se uma substituição, que é apenas simplificada e associada á variável, mas que só será utilizada quando e a cada vez que o conteúdo da variável for solicitado. Somente então os valores das variáveis envolvidas serão utilizados. Isto faz com que os valores atualizados sejam os usados. Assim, na linha 4 foi **CONT1** e na linha 7, **CONT2**.

Prosseguimos com o exemplo, atribuindo o valor **DETERM1** a **TERMO1**, que passa, agora, a ser determinado.

```
8:  termo1:=determ1;

      TERMO1  := DETERM1

9:  somaatr;

      DETERM1 + CONT1

10: somasub;

      DETERM1 + CONT2

11: termo1:=determ2;

      TERMO1  := DETERM2

12: somaatr;

      DETERM2 + CONT1

13: somasub;

      DETERM2 + CONT2
```

Como, na época da atribuição a **SOMAATR**, **TERMO1** era indeterminado, no valor calculado da expressão, a referência a **TERMO1** levou

ao seu próprio nome, ao qual seu conteúdo estava associado.

Agora, mesmo tendo sido atribuido um valor a **TERMO1**, a referência a **TERMO1** no valor calculado e atribuído a **SOMAATR** é levada sempre ao valor atual de **TERMO1**. Essa referência não é substituída por um valor definido.

Assim, na linha 9 a referência a **TERMO1** encontrou o valor **DETERM1** e na linha 12, **DETERM2**.

Como seria de esperar, **SOMASUB** levou sempre em conta o valor atual de **TERMO1** (e, também, o de **TERMO2**, é claro).

Resumindo, substituições sempre levam em conta os conteúdos atuais das variáveis, enquanto que atribuições utilizam os conteúdos do momento da atribuição. No entanto, como indeterminados não tem conteúdo definido, estando este associado inicialmente ao seu nome, atribuições que os envolvam, fornecerão, no futuro, resultados dependentes do valor atual do indeterminado via essa referência indireta.

O comando **LET**, além de introduzir regras de substituição, é, também, frequentemente, usado para definir novos operadores, como já mencionado. No exemplo citado da página 38, introduzimos o operador **SEC** pela sua definição matemática, em termos do **COS**.

Há casos, porém, em que é mais interessante definir um operador através de algumas de suas propriedades, seja porque é a única definição de que dispomos, ou porque este procedimento se torna mais eficiente.

No exemplo abaixo, precisávamos apenas das propriedeades de derivação do operador **SEC** e as introduzimos através do comando **LET** apropriado.

A cláusula **FORALL X** é necessária para que a regra se torne geral, não se restringindo à derivada com relação a **X**

```
5:  operator sec;

6:  df(sec(x),x);

      DF(SEC(X),X)

7:  forall x let df(sec(x),x)=sec(x)*tan(x);

8:  df(3*sec(x*y),y);

      3*TAN(X*Y)*SEC(X*Y)*X

9:  % forall x,y let df(sec(y),x)=sec(y)*tan(y)*df(y,x)
```

Na linha 6, o comando de derivação reconhece **SEC** como operador, devido à afirmação nesse sentido dada na linha 5 (**REDUCE** perguntaria se era operador, caso essa informação não houvesse sido dada, como ocorreu no exemplo da página 37), mas não sabe derivá-lo, deixando indicado, como sempre faz nesses casos.

Após a regra na linha 7, efetuamos outra derivação envolvendo SEC e, agora, obtém-se o resultado correto.

Note que REDUCE conhece a regra da cadeia de derivações, caso contrário, teríamos de introduzir a propriedade da linha 9 (um comando iniciado por COMMENT ou por REDUCE e tem apenas o objetivo de documentação).

Quando se introduzem regras de substituição em REDUCE, é necessário cuidado para que elas não se tornem inconsistentes. É possível que interfiram entre si, levando a efeitos imprevisíveis.

Também é preciso cuidado para que uma regra não leve a um "loop" infinito, como no exemplo abaixo, em que se pretende simular na versão 2 de REDUCE, a qual não realizava integrações, integração polinomial.

Aqui o sistema faz as seguintes substituições sucessivas

$$x^3 \to x^3/3 \to x^4/4! \to x^5/5! \to \ldots$$

levando a memória à exaustão.

Tem-se porém a chave RESUBS que, quando desligada, faz com que cada regra introduzida seja utilizada apenas uma vez para cada expressão.

Desligando-se esta chave, a regra funciona perfeitamente neste exemplo

```
      REDUCE 2 (APR-15-79 (MTS DEC-10-82)) ...

* forall n let x**n=x**(n+1)/(n+1);

* x**2;
  *** G2-PUSHDOWN STACK OVERFLOW.
      ***** ERROR TERMINATION


* off resubs;

* x**2;

        3
       X /3
```

Quando tentamos este mesmo procedimento na versão 3.1 de REDUCE, a chave RESUBS parece não ter efeito, embora exista (não houve mensagem do tipo !*RESUBS DECLARED FLUID como no exemplo da página 52).

```
        REDUCE 3.1, 15-Apr-84 ...

1:  forall n let x**n=x**(n+1)/(n+1);

2:  off resubs;
```

```
3:  x**2;
  *** PUSHDOWN STACK OVERFLOW
  *   I
```

Acreditamos que esta falha será corrigida em alguma versão próxima de REDUCE.

Alguns pontos importantes sobre o comando LET precisam ainda ser levantados.

Suponhamos que queremos introduzir uma regra de substituição do tipo da da linha 2 do exemplo abaixo.

Do resultado da linha 3, somos levados a concluir que a regra está funcionando como desejamos mas tal não é verdade.

```
1:  array a(5); let x+a(2)=3;

8:  x+a(2)+c;

      0 + 3

4:  a(2);

      0

5:  x;

      3

6:  a(2):=1$ x+a(2)+c;

      0 + 4

8:  x;

      3

9:  let a(3)=4;

      ***** Substitution for 0 not allowed

10:  operator f; forall x,y let f(x)+f(y)=x*y;

      ***** Unmatched free variable(s)  =I
```

Em REDUCE, todos os "arrays" são inicializados nulos, como se vê da linha 4; além disso, regras que envolvem expressões com operações entre

mais de um termo no lado esquerdo de um comando LET são interpretadas passando-se todos os termos menos um para o lado direito da regra, tornando-se, assim, uma definição para aquele termo que ficou.

Assim, a regra acima é interpretada como

$$\text{let } x = 3 - a(2);$$

e, como $a(2) = 0$, isso significa, na prática, $x = 3$, como se vê da linha 5.

Da mesma forma, fazendo-se a atribuição da linha 6, o resultado da aplicação da regra se altera correspondentemente.

Também a regra da linha 9 não é aceita, pois é interpretada como

$$0 = 3$$

Pelo mesmo problema, a regra da linha 10 não é aceita por se referir, na prática, não a uma definição da soma $f(x) + f(y)$, mas da forma

$$\text{let } f(y) = x^*y - f(x);$$

aparecendo a variável X apenas do lado direito, o que não é permitido, após a cláusula FORALL X,Y.

Por vezes, desejamos introduzir regras de substituição referindo-se a potências especificas de variáveis e que não devem ser aplicadas a potências diferentes, como no exemplo abaixo

```
12:  let u**2*v=3*w;

13:  u**2*v**2;

      3*V*W

14:  clear u**2*v;

15:  match u**2*v=3*w;

16:  u**2*v**2;

       2  2
      U *V

17:  u**2*v*w;

       2
     3*W
```

Da linha 13 vê-se que a regra não foi corretamente aplicada. Para esses casos, existe o comando MATCH com sintaxe idêntica ao LET, que, como se vê das linhas 16 e 17 tem a ação desejada.

5.2 Uso avancado de substituições

Conforme já deve ser sido possível notar, o comando LET é, de fato, muito poderoso, oferecendo muitos recursos ao usuário de REDUCE.

Mencionamos, na página 59, a utilização de LET para "auxiliar" o sistema de simplificação de REDUCE que, presentemente, tem ainda algumas dificuldades com radicais.

No mesmo exemplo, salientamos, também, a influência do estado das chaves sobre o funcionamento do comando LET.

Na página 16, demos um exemplo de definição de um procedimento de simplificação trigonométrica, usando vários comandos LET.

No exemplo da página 19, demonstramos em REDUCE o conceito de integração por busca de padrões pela construção de um operador de integração que reconhece um padrão específico.

Há recursos para tornar mais especifico o campo de atuação de uma regra. Isso é feito incluindo-se um teste, precedido da expressão SUCH THAT, de forma que a regra só é aplicada se as condições forem satisfeitas, como no exemplo abaixo

```
18:  operator factorial;

19:  forall x such that numberp x and x>=1
19:   let factorial(x) = x*factorial(x-1);

20:  factorial(n);

      FACTORIAL(N)

21:  let factorial(0)=1;

22:  factorial(8);

      6

23:  factorial(2.5);

      *** 2.500000E0 represented by 5/2

      (15*FACTORIAL(1/2))/4
```

A condição NUMBERP acima restringe a aplicação da fórmula dada para quando x é número (não literal) e a outra tem o objetivo de excluir os valores negativos.

Fornecemos o caso particular **0!** e vemos o cálculo correto de **2.5!**, em termos de **(1/2)!**.

Conforme já mencionamos antes, frequentemente o resultado de um cálculo algébrico resulta numa expressão bastante grande, ocupando uma porção crescente da memória, com o decorrer do cálculo.

Tal não costuma acontecer em cálculo numérico pois, em ponto flutuante, embora a magnitude do número cresça, conserva-se apenas uma certa quantidade de dígitos significativos, ocupando uma área fixa da memória.

Algo semelhante se pode fazer em cálculo algébrico, em certos casos. No exemplo seguinte, por considerações externas, tem-se que potências iguais ou superiores a 8 da variável x podem ser desprezadas. O comando LET abaixo faz com que apenas as potências significativas permanecam.

Essa técnica simples não funciona em qualquer situação, porém, como se vê das linhas 3 e 4. Os problemas aí surgiram porque REDUCE tenta, primeiro implementar quaisquer regras de substituição pertinentes e só então simplifica a expressão.

Com isso, na linha 3, primeiro fez x^{10} nulo e só depois dividiu por x^5. Na linha 4, fez x^{20} e x^{10} nulos e depois tentou realizar a divisão, resultando uma indeterminação.

```
1:  let x**3=0;

2:  (3+2*x**3)**4;

                6          3
      216*X   + 216*X   + 81

3:  x**10/x**5;

      0

4:  x**20/x**10;

      ***** Zero denominator
```

Se houvesse feito primeiro as simplificações, obtendo x^5 e x^{10} respectivamente, teria chegado aos resultados corretos.

Numa situação mais complexa, simultaneamente a variável y é tal que as potências iguais ou superiores a 4 devem ser desprezadas.

Na expansão do binômio em y, o cálculo se faz corretamente, como no exemplo anterior mas quando se multiplicam ambos os resultados, deve se considerar que também são nulos os termos do tipo $x^m y^n$ com $m + 2n \geq 8$.

Para estes casos, REDUCE possui os comandos WEIGHT que atribui pesos às variáveis do problema e WTLEVEL que estabelece qual o valor máximo da soma ponderada de potências de variáveis num produto a ser mantido.

Assim, em vista das considerações acima, os pesos relativos de y e x estão na proporção 8:4 ou 2:1 e estabelecemos o valor máximo 7.

Desta forma, todos os termos considerados nulos, acima, serão apagados da expressão resultante.

```
5:  let y**4=0;

6:  (2+3*y**2)**3;

          4       2
     54*Y  + 36*Y  + 8

7:  input(2)*input(6);

           6  2          6          3  2
     7776*X  *Y  + 1728*X  + 7776*X  *Y  + 1728

       3          2
     *X  + 2916*Y  + 648

8:  clear x**3,y**4;

9:  weight x=1, y=2; wtlevel 7;

11: input(7);

          6          3  2          3          2
     1728*X  + 7776*X  *Y  + 1728*X  + 2916*Y

      + 648
```

Infelizmente, estes comandos só funcionam bem para potências inteiras.

Por exemplo, se introduzimos a expressão abaixo, deveriam ser mantidas apenas as potências 6, 6.5, 7 e talvez 7.5. No entanto, na linha 12, a potência 7 é apagada durante a expansão do binômio, enquanto na linha 13, é mantida.

Além disso, aparece o estranho fator K*, variável interna de REDUCE, que parece ser utilizada para tomar conta da multiplicidade do sinal da raiz.

Na linha 14 aparece, da mesma forma, o fator K* também ao quadrado.

Na linha 15 tentamos verificar se esse fator decorre da multiplicidade da raiz quadrada, utilizando a chave REDUCED.

```
12: x**6*(1+x**(1/2))**4;
```

```
                    7                      6     6
4*SQRT(X*K*)*X    + 4*SQRT(X*K*)*X    + X
```

```
13: x**6+x**6.5+x**7+x**7.5+x**8;

*** 6.500000E0 represented by 13/2

*** 7.500000E0 represented by 15/2

                   7                    6     7     6
SQRT(X*K*)*X    + SQRT(X*K*)*X    + X   + X
```

```
14: y**2*(1+y**(1/4))**4;

          2 (3/4)   2           2 (1/4)   2
4*(Y*K*  )      *Y   + 4*(Y*K*  )      *Y

                     2     2
  + 6*SQRT(Y)*Y    + Y
```

```
15: on reduced; input(14);

                3   (3/4)   2         (1/4)   2
4*SQRT(K*)  *Y      *Y    + 4*Y       *Y

                     2     2
  + 6*SQRT(Y)*Y    + Y
```

Capítulo 6

Operadores e procedimentos

Algumas funções matemáticas são já conhecidas por REDUCE e alguns exemplos apareceram no exemplo da página 38, tais como SIN, COSH, ATAN, LOG, DILOG e ERF.

Lá foi visto, também, que REDUCE, no modo normal, só conhece algumas propriedades básicas dessas funções, tais como suas derivadas, integrais (não de todas) e valores numéricos triviais.

Na página 56, vimos que a chave NUMVAL faz com que REDUCE forneça os valores numéricos dessas funções, com precisão arbitrária, sob controle da chave BIGFLOAT e do comando PRECISION.

Também são conhecidos vários operadores numéricos, tais como MAX (máximo de uma lista de valores), MIN (mínimo), ABS (valor absoluto).

Têm-se ainda uma série de operadores prefixados, tais como DF, INT, NUM, DET, SUB, etc. e operadores infixados, como *, +, OR, MEMBER, EQUAL, :=, etc.

Para uma lista completa destes operadores, recomenda-se a consulta ao manual de REDUCE, onde se descreve também a ação de cada um deles.

6.1 Introduzindo novos operadores

É muito interessante a possibilidade que REDUCE oferece ao usuário de definir novos operadores.

Tivemos já exemplo disso nas páginas 38 e 78, quando definimos a função SEC. Outros exemplos surgiram nas páginas 82 (fatorial) e 16 (simplificação trigonométrica).

Uma função pode ser definida precisamente através de sua descrição (vide exemplo da página 82), como foi o caso do fatorial e da primeira definição de SEC (pag. 38), ou de forma abstrata, apenas por algumas propriedades, como foi o caso do simplificador trigonométrico e da segunda definição de SEC pag. 16 e 78).

É importante ressaltar que podemos definir, também, novos operadores infixados fornecendo, além de sua definição, o comando

INFIX *nome do operador;*

O acesso ao modo simbólico de REDUCE (vide próximo capítulo) permite ainda a construção de operadores sofisticados com os recursos da linguagem LISP e outros fornecidos por esse modo.

6.2 Operadores lineares, antisimétricos e antico mutativos

Vários operadores em REDUCE são lineares nos seus argumentos, como por exemplo DF e INT.

Pode-se fazer linear um operador criado pelo usuário, através da declaração LINEAR, como foi feito no exemplo de integrador da página 19. Na sequência abaixo, detalhamos o efeito dessa declaração.

O operador F definido na linha 1 não é, em princípio, linear e por isso, não atua linearmente sobre a expressão dada como primeiro argumento.

Tornando-o linear, mesmo sem conhecer qualquer outra propriedade particular deste operador, REDUCE utilizará as de linearidade *com relação ao segundo parâmetro*, sendo as variáveis A, B e C consideradas constantes.

Da mesma forma, na linha 5, Y é considerado constante, mas pode ser definido como dependente de X através do comandoDEPEND, já visto na página 40.

```
1:  operator f;

2:  f(a*x**5+b*x+c,x);

                 5
    F(A*X  + B*X + C,X)

3:  linear f; input(2);

                 5
    F(X ,X)*A + F(X,X)*B + F(1,X)*C

5:  f(a*x**5*y+b*x*y**2+c,x,y);

                 5                      2
    F(X ,X,Y)*A*Y + F(X,X,Y)*B*Y  + F(1,X,Y)*C

6:  depend y,x; input(5);

                 5                    2
    F(X *Y,X,Y)*A + F(X*Y ,X,Y)*B + F(1,X,Y)*C
```

As propriedades de linearidade são explicitadas na sequência seguinte, onde z é suposto não depender de x.

```
8:  f(-z,x);

        - F(1,X)*Z

9:  f(-y,x);

        - F(Y,X)

10: f(y+z,x);

      F(Y,X) + F(1,X)*Z

11: f(y*z,x);

      F(Y,X)*Z

12: f(y/z,x);

      F(Y,X)/Z

13: f(x*y/z,x);

      F(X*Y,X)/Z

14: f(0,x);

  0
```

Podemos, também, definir operadores como simétricos ou antisimétricos, como no exemplo abaixo.

De vez que REDUCE tem uma ordem interna para variáveis, conforme mencionado na página 69, os argumentos dos operadores serão reordenados, ocorrendo mudança de sinal se o operador for antisimétrico, como nas linhas 5 e 6.

```
1:  operator sym,asym,asym2;

2:  symmetric sym;

3:  antisymmetric asym,asym2;

4:  sym(b,a);

      SYM(A,B)

5:  asym(b,a);
```

```
        - ASYM(A,B)

6:  asym(c,asym2(b,a));

        - ASYM( - ASYM2(A,B),C)

7:  asym(x,x);

        0
```

Operadores podem ainda ser declarados anticomutativos. Assim, nas linhas 10 e 11, abaixo, a ordem dos termos na soma foi invertida, mas a dos produtos não.

Podem-se, é claro, introduzir regras de comutação através do comando LET. Na linha 12, informamos que os operadores TETA1 e TETA2 anticomutam.

Operadores com número nulo de argumentos também são permitidos mas é necessário introduzir regras específicas para este caso, como na linha 15.

Alertamos o leitor interessado que uma mudança de notação do tipo da produzida pelo comando LET da linha 17 não basta para dispor-se de *variáveis* anticomutativas.

Como já foi mencionado antes, no resultado da linha 16 será feita a substituição correspondente a esse comando LET, que as transformará em variáveis ordinárias, sendo possível a mudança de ordem dos termos do produto e o cancelamento dos termos da soma.

Por outro lado, o comando LET da linha 20 transforma as variáveis ordinárias nos operadores anticomutativos antes de qualquer simplificação.

```
8:  operator teta1,teta2;

9:  noncom teta1,teta2;

10: teta1(z1)*teta1(z2)-teta1(z2)*teta1(z1);

      - TETA1(Z2)*TETA1(Z1)  + TETA1(Z1)*TETA1(Z2)

11: teta1(z1)*teta2(z2)-teta2(z2)*teta1(z1);

      - TETA2(Z2)*TETA1(Z1)  + TETA1(Z1)*TETA2(Z2)

12: forall x,y let teta2(x)*teta1(y)=
12:                  -teta1(y)*teta2(x);

13: input(11);
```

```
      2*(TETA1(Z1)*TETA2(Z2))

14:  teta1(z1)*teta2(z1)-teta2(z1)*teta1(z1);

      2*(TETA1(Z1)*TETA2(Z1))

15:  let teta2()*teta1()=-teta1()*teta2();

16:  teta1()*teta2()-teta2()*teta1();

      2*(TETA1()*TETA2())

17:  let teta1()=teta1, teta2()=teta2;

18:  input(16);

      0

19:  clear teta1(), teta2();

20:  let teta1=teta1(), teta2=teta2();

21:  teta1*teta2-teta2*teta1;

      2*(TETA1()*TETA2())
```

6.3 Procedimentos

À medida que o usuário faz uso cada vez mais frequente de REDUCE, verifica que certo número de comandos aparecem repetidamente em seus cálculos, na mesma sequência, talvez diferindo apenas pelos parâmetros.

Tal como outras linguagens de computação, REDUCE permite definir esse conjunto de comandos como um procedimento com parâmetros formais, os quais serão substituidos, quando da sua execução, pelos parâmetros atuais do usuário.

No exemplo abaixo, definimos o fatorial na forma de um *procedimento*, ao contrário do visto na página 82, quando o definimos abstratamente, por meio de comandos LET.

O procedimento é definido pela declaração PROCEDURE, seguida pelo nome do procedimento e dos parâmetros entre parêntes ou, no caso de um único, separado por um espaço.

Após essa declaração, vem o comando de definição, propriamente dita.

```
8:    procedure factorial(n);
       if n=0 then 1 else n*factorial(n-1);
```

```
FACTORIAL

9:      factorial(3);

6

10:     factorial(30);

265252859812191058636308480000000
```

Se a sintaxe de ambos estiver correta, REDUCE responde com o nome do procedimento, indicando que ele agora está disponível. Caso o procedimento estiver sendo redefinido, aparece a mensagem

*** *nome do procedure* REDEFINED

O usuário deve verificar se se trata realmente de uma redefinição de um procedimento seu, anteriormente definido. Os procedimentos internos de REDUCE *não* são protegidos, para permitir maior flexibilidade e domínio ao usuário.

Outros exemplos de procedimento apareceram na página 5.

Frequentemente, a definição do procedimento envolve mais de um comando. Neste caso, deve-se se transformá-los num comando composto, atrav'es das palavras BEGIN e END, ou colocando-os entre ‼ e ‼.

No exemplo seguinte, fazemos uso de BEGIN e END para fornecer um comando composto como definição de um operador. O resultado do comando composto é aqui usado como valor para o comando LET.

```
1:  operator factorial;

2:  forall n let factorial(n) =
2:              begin scalar m,s;
2:              m:=1; s:=n;
2:          l1: if s=0 then return m;
2:              m:=m*s;
2:              s:=s-1;
2:              go to l1
2:          end;

3:  factorial(4);
```

Agora usamos o comando composto na definição de um procedimento para cálculo de limites, usando a regra de L'Hôpital até cinco vezes seguidas.

Nesse procedimento, aparece, também, a forma ¡¡ ¿¿ de comando composto dentro de um comando iterativo **FOR**.

```
1:    algebraic procedure limit(ex, indet, pnt);
1:       begin integer iteration;
1:       scalar n, d, nlim, dlim;
1:       iteration := 0;
1:       n := num(ex);
1:       d := den(ex);
1:       nlim := sub(indet=pnt, n);
1:       dlim := sub(indet=pnt, d);
1:       while nlim=0 and dlim=0 and iteration<5 do
1:     <<
1:          n := df(n, indet);
1:          d := df(d, indet);
1:          nlim := sub(indet=pnt, n);
1:          dlim := sub(indet=pnt, d);
1:          iteration := iteration + 1 >>;
1:       return (if nlim=0 then
1:                   if dlim=0 then unknown
1:                   else 0
1:                else if dlim=0 then undefined
1:                     else nlim/dlim)
1:       end;
```

```
LIMIT
```

É interessante comparar o funcionamento desse procedimento com o do comando SUB para alguns casos de limites que levam a indeterminação.

```
2:    (e**x-1)/x;
```

```
      X
(E  - 1)/X
```

```
3:    sub(x=0, ws);
```

```
***** Zero denominator
```

```
4:    limit(ws(2), x, 0);

                1
5:    ((1-x)/log(x))**2;

                  2                    2
               (X  - 2*X + 1)/LOG(X)

6:    sub(x=1, ws);

      ***** Zero denominator

7:    limit(ws(5), x, 1);

      1
```

Uma vez construido um procedimento ou um programa e testado, podemos desejar guardá-lo em meio permanente, de forma que possamos chamá-lo sempre que dele necessitemos, sem ser necessário digitá-lo novamente, de maneira semelhante à pela qual armazenamos uma expressão no exemplo da página 63.

Uma maneira de se fazer isso é, uma vez criado o arquivo de saída, usar-se o comando **DISPLAY** o qual imprimiria na tela todos os comandos digitados durante a sessão *na sequência inversa*.

```
8:    off nat; out "limit.rdc";
10:    display(all);
11:    out t; bye;
```

Tendo sido previamente digitado o comando **OUT** com o nome do arquivo de saída, os comandos serão escritos aí, conforme se vê, examinando-se o conteúdo do arquivo.

```
ALGEBRAIC PROCEDURE LIMIT(EX, INDET, PNT);
    BEGIN INTEGER ITERATION;
    SCALAR N, D, NLIM, DLIM;
    ITERATION := 0;
    N := NUM(EX);
    D := DEN(EX);
    NLIM := SUB(INDET=PNT, N);
    DLIM := SUB(INDET=PNT, D);
    WHILE NLIM=0 AND DLIM=0 AND ITERATION<5
  <<
      N := DF(N, INDET);
      D := DF(D, INDET);
      NLIM := SUB(INDET=PNT, N);
      DLIM := SUB(INDET=PNT, D);
      ITERATION := ITERARION +1 >>;
    RETURN (IF NLIM=0 THEN
                IF DLIM=0 THEN UNKNOWN
                ELSE 0
            ELSE IF DLIM=0 THEN UNDEFINED
                 ELSE NLIM/DLIM)
    END;
```

É necessário, então, editar-se o arquivo para remover os comandos incluidos no arquivo e que não se refiram ao procedimento (os das linhas 2 a 7, no caso), incluídos pelo comando **display** (representados pelos pontos acima).

Para programas e procedimentos grandes, este recurso é conveniente produzir uma cópia fiel, sem o perigo de erro na sua digitação.

Capítulo 7

O modo simbólico

Todos os recursos de REDUCE até agora exibidos referiram-se ao chamado *modo algébrico* de REDUCE, que é o ambiente mais apropriado para a programaçãao no estilo matemático natural.

No entanto, REDUCE oferece ao usuário também os recursos da linguagem LISP, através do seu chamado *modo simbólico*, também conhecido como RLISP ("REDUCE LISP").

Este modo pode tanto ser utilizado para simplesmente se programar em LISP como para se desenvolverem procedimentos e operadores sofisticados, para o que o modo algébrico não dispõe de recursos suficientes.

O usuário interessado apenas em LISP tem, também, acesso a um dialeto chamado "Standard LISP" ou SLISP, através do seu compilador correspondente.

Neste capítulo faremos uma breve introdução ao modo simbólico e à maneira de se utilizar dos seus recursos a partir do modo algébrico.

7.1 A estrutura de REDUCE

O sistema REDUCE se articula ao modo simbólico, a SLISP, e ao sistema operacional (OS) da seguinte forma:

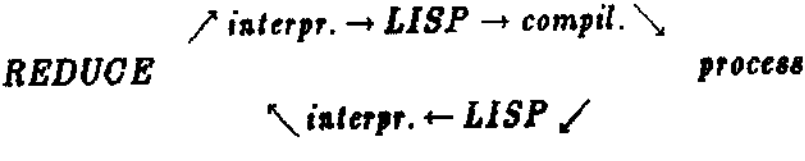
REDUCE
RLISP
SLISP
OS

REDUCE consiste num pacote de programas, procedimentos, operadores, etc. codificados em um seu subconjunto chamado RLISP, os quais são analisados por um interpretador codificado principalmente em SLISP e em parte em Assembler.

Os comandos de REDUCE são analisados pelo interpretador e transformados em um programa LISP, o qual é executado pelo processador, sob o controle do sistema operacional.

O resultado é novamente analisado e transformado numa expressão com a sintaxe de REDUCE.

Esquematicamente, teríamos o procedimento abaixo



RLISP não é o ambiente mais apropriado para o uso rotineiro dos algoritmos algébricos de **REDUCE** de forma confortável. Por isso e para evitar que o usuário acidentalmente interfira com as variáveis e funções internas, a sessão é iniciada usualmente no modo algébrico. Para isso, usa-se o comando (do sistema operacional)

```
/REDUCE                 (sistemas VMS, CMS)
ou
$RUN NEW:REDUCE         (sistema MTS)
```

Se o usuário desejar programar em **RLISP** ou em **LISP**, pode iniciar a sessão diretamente nesses ambientes, acrescentando

```
PAR=RLISP
ou
PAR=SLISP
```

respectivamente aos comandos acima.

Para passar-se de um ambiente a outro, usam-se os comandos indicados na tabela abaixo

de\para	REDUCE	RLISP	SLISP	OS
REDUCE		SYMBOLIC;	END;	BYE;/QUIT;
RLISP	ALGEBRAIC;		END;	BYE;/QUIT;
SLISP	(BEGIN)	(BEGIN)		(BYE)/(QUIT)
OS	$RUN ...(/REDUCE)	Par=RLISP	Par=SLISP	

7.2 Trabalhando no modo simbólico

Estando no modo algébrico de REDUCE, pode-se executar um comando em simbólico, prefixando-o por **SYMBOLIC**. O comando é executado no modo simbólico e o resultado convertido ao modo algébrico, continuando o cálculo no modo algébrico, conforme o exemplo abaixo.

Para conveniencia do usuário, qualquer operador cujo primeiro argumento seja precedido do sinal ' , será calculado no modo simbólico.

```
2:  symbolic car '(a1 a2 a3);

      A1

3:  (x+y)**2;
```

```
     2            2
    X  + 2*X*Y + Y
```

```
4:  cdr '(a1 a2 a3);

    (A2 A3)
```

Quando se passa ao modo simbólico, REDUCE responde com NIL. Observe-se que a sintaxe do modo simbólico é quase a mesma do algébrico, i. e., não se usam parênteses delimitadores como em LISP e o terminador continua sendo ; . As atribuições continuam também sendo indicadas por := .

Um recurso interessante do modo simbólico é a chave DEFN que, quando destivada, ao invés de executar o cálculo indicado, responde com o equivalente LISP do comando.

```
5:  symbolic;

    NIL

6:  on defn;
7:  x:=y;
      (SETQ X Y)

8:   f:= '((cbpf  .  cnpq) (rua xavier sigaud 150)


         rio de janeiro);
      (SETQ F
         '((CBPF  . CNPQ)
            (RUA XAVIER SIGAUD 150)
            RIO DE JANEIRO))

9:  lambda (x,y); car x .cdr y ;
      (LAMBDA (X Y) (CONS (CAR X) (CDR Y)))

10: off defn;

    NIL
```

Uma vez no modo simbólico, o usuário dispõe dos operadores usuais de LISP, fazendo uso, porém, da sintaxe simplificada de RLISP.

Para explorar listas em profundidade, iterativamente, existe o comando FOR EACH com a sintaxe

FOR EACH *identificador* IN *ou* ON *lista* DO , COLLECT *ou* CONC *expressão envolvendo identificador*;

como no exemplo abaixo.

```
11: f:='((cbpf.cnpq)  (rua xavier sigaud 150)
        (rio de janeiro));
    *** F DECLARED FLUID
    ((CBPF . CNPQ) (RUA XAVIER SIGAUD 150) (RIO DE JANEIRO))
12: cdar(f);
    CNPQ
13: cddr(f);
    ((RIO DE JANEIRO))
14: for each x on '(a b c) collect  x;
    ((A B C) (B C) (C))
```

A opção IN faz com que a interação seja feita sobre cada elemento da lista (representado por *identificador*, enquanto ON , sobre a lista restante a cada iteração.

A opção DO faz com que a expressão seja calculada mas seu valor não seja retornado, e sim o do termo sobre a qual atuou (dependendo de IN ou ON); COLLECT, produz uma lista, cujos elementos serão os resultados calculados e CONC, uma lista fruto da concatenação desses resultados.

Desta forma, essas opções correspondem aos vários operadores aplicativos de LISP. Sua correspondência é indicada na tabela abaixo

	DO	COLLECT	CONC
IN	MAPC	MAPCAR	MAPCAN
ON	MAP	MAPLIST	MAPCON

7.3 Comunicando entre os modos

Para fazer uso dos recursos de RLISP no modo algébrico, está disponível um meio de comunicação entre os dois modos.

Se o usuário deseja manipular uma expressão definida no modo algébrico com as técnicas do modo simbólico, deve atribuí-la a uma variável e declará-la compartilhável pelo comando **SHARE**.

Como já foi dito na página 7, polinomios podem ser representados na forma de listas. Na forma de lista usual, um termo univariado de um polinomio seria representado como

(TIMES (EXPT *variável potência*) *coeficiente*)

Na notação comumente usada por REDUCE, cada variável é representada por um par ordenado da forma

((*variável* . *potência*) . *coeficiente*)

se o coeficiente for numérico (lembrar que um identificador numérico representa sempre o próprio número),

Note, porém, que, no modo algébrico, as expressões são passadas internamente, de um comando a outro, na forma prefixada de LISP, mas o processador algébrico não converte formas e quocientes padrão à notação prefixada, por economia de esforço.

Assim, as formas e quocientes padrão são armazenados como uma lista com a sintaxe

(*SQ *quociente padrão* T *ou* NIL)

onde *SQ indica ao processador algébrico que se trata de uma forma forma prefixada que é, na realidade, um quociente padrão e o terceiro parâmetro indica se ele ainda precisa de processamento (simplificação) ou não.

Para se obter, então, o quociente padrão a partir da forma acima, toma-se o CADR dela.

```
6:  share x,y;

7:  x:=(a+b)**2;

                2              2
      I := A  + 2*A*B + B

8:  symbolic;

      NIL

9:  x;
```

```
(!*SQ ((((A . 2) . 1) ((A . 1) ((B . 1) . 2))
 ((B . 2) . 1)) . 1) T)

10: z:=cadr x;
*** Z DECLARED FLUID

((((A . 2) . 1) ((A . 1) ((B . 1) . 2)) ((B .
 2) . 1)) . 1)

11: lt numr z;

((A . 2) . 1)

12: T:=mk!*sq !*t2q lt numr z;

(!*SQ ((((A . 2) . 1)) . 1) T)

13: algebraic;

14: y**2;


  4
 A
```

Uma vez obtido o quociente padrão no modo simbólico, pode-se manipulá-lo fazendo uso de vários seletores e construtores disponíveis, tais como (vide a lista completa no manual)

DENR	denominador do quociente padrão
NUMR	numerador do quociente padrão
LT	primeiro termo do polinomio
MVAR	variável principal do polinomio
PDEG	potência da variável
TC	coeficiente da variável
.+	adiciona um termo ao polinomio
./	forma um quociente de dois polinomios

Pode-se, ainda, converter entre as várias estruturas de dados existentes através das funções

!*A2F	converte uma expressão algébrica à forma padrão.
!*F2A	converte uma forma padrão à expressão algébrica correspondente.
!*T2Q	converte um termo padrão a quociente padrão.

Para passar de volta expressões ao modo algébrico, deve-se certificar de que ela está numa notação prefixada. Se se estava manipulando quo-

cientes padrão, porém, ela estará na segunda forma acima. Neste caso, será necessária conversão.

Para isso, transforma-se a expressão em quociente padrão, usando as funções de conversão acima e usa-se **PREPSQ** ou **MK!*SQ** sobre ela para colocá-la numa das duas notações prefixadas acima.

Note que a segunda opção exige menos trabalho de conversão.

Capítulo 8

Aplicações

Até agora descrevemos vários recursos do REDUCE, os quais permitem uma variedade de cálculos algébricos bastante úteis.

A forma da apresentação, até aqui, foi usando REDUCE como uma "calculadora algébrica", i. e., fazendo-se no computador um cálculo que envolve grande quantidade de trabalho algébrico mas consiste em apenas alguns comandos. Cálculos mais complexos, porém, podem necessitar vários comandos, na forma de um programa.

Neste capítulo, além de enfatizar a construção de programas, apresentaremos exemplos de programas para cálculos em algumas áreas de aplicação.

8.1 Cálculos em Física de Altas Energias

Embora o sistema SCHOONSCHIP mencionado na página 29 seja mais eficiente para cálculos em Física de Altas Energias que quase qualquer outro sistema, estes têm um interesse histórico em REDUCE pois foi para essa aplicação que este sistema foi inicialmente concebido.

A partir daí, REDUCE evoluiu para um sistema algébrico geral e hoje esses cálculos são referidos num único capítulo do manual (já o foram num apêndice) e efetuados por um módulo separado chamado **HEPHYS**.

Este módulo introduz a declaração **VECTOR** que permite variáveis representando quadrivetores de momento, com a operação especifica produto interno, indicada pelo sinal . e massas definidas pelo comando **MASS**.

Introduz, também, as matrizes de Dirac, representadas pelo operador G. Seu primeiro parâmetro é o identificador da linha fermiônica do diagrama de Feynman. O segundo, o momento com o qual a matriz está contraída.

Produtos de matrizes gama associadas com a mesma linha fermiônica e momentos diferentes podem ser representados por um operador com mais

de dois parâmetros, representando os diversos momentos.

Sempre que uma expressão envolvendo matrizes de Dirac é avaliada, o traço de cada produto de matrizes é calculado, exceto para aquelas linhas cujos identificadores foram declarados pela declaração NOSPUR.

Apresentamos aqui um simples cálculo da seção de choque do espalhamento Compton. Seguimos a notação de Bjorken e Drell, eqs. 7.72 a 7.73. O resultado obtido confere com o da eq. 7.74.

Os momentos dos fótons incidentes e emergentes são **ki** e **kf**, respectivamente; os dos elétrons, **pi** e **pf**. Os vetores e e **ep** são os vetores de referência.

As polarizações dos fótons são definidas através de comandos LET apropriados.

```
1:  on div;

2:  mass ki= 0, kf= 0, pi= m, pf= m;

3:  vector e,ep; mshell ki,kf,pi,pf;

5:  let pi.e= 0, pi.ep= 0, pi.pf= m**2+ki.kf,
        pi.ki= m*k, pi.kf= m*kp, pf.e= -kf.e,
        pf.ep= ki.ep, pf.ki= m*kp, pf.kf= m*k,
        ki.e= 0, ki.kf= m*(k-kp), kf.ep= 0,
        e.e= -1, ep.ep= -1;

6:  (g(1,pf)+m)*(g(1,ep,e,ki)/(2*ki.pi)
          + g(1,e,ep,kf)/(2*kf.pi))
    *(g(1,pi)+m)*(g(1,ki,e,ep)/(2*ki.pi)
          + g(1,kf,ep,e)/(2*kf.pi))$

7:  write "THE COMPTON CROSS-SECTION IS " ,ws;

 THE COMPTON CROSS-SECTION IS

          2              (-1)          (-1)
 2*EP.E  + 1/2*K*KP      + 1/2*K     *KP - 1
```

8.2 Cálculos em Relatividade Geral

Apresentamos aqui um exemplo de programa para cálculo de quantidades físicas interessantes na Relatividade Geral.

A computação algébrica se desenvolveu muito nesta área da Física especialmente por esta necessitar de volumes enormes de cálculo facilmente algoritmizável.

O programa que apresentamos abaixo está longe de ser eficiente pois consiste na simples codificação das fórmulas usuais dos livros.

Não faz uso de técnicas mais sofisticadas como base de tetradas, formalismo de Newman-Penrose, etc. nem foi elaborado com preocupações de economia de esforço computacional.

Embora tenha sido codificado com a única preocupação de ser didático, pode ser, e de fato foi, útil pois é, em geral, mais rápido que um cálculo à mão.

Quem pretende usar frequentemente REDUCE para este tipo de cálculo deveria, no entanto, procurar obter um programa mais eficiente. É interessante, também verificar a possibilidade de fazer uso de sistemas especializados como SHEEP (vide página 30).

Após a mudança de notação mencionada na página 28, introduzem-se o sistema de coordenadas e a métrica em questão representada pelo arranjo gg.

Calculam-se, em seguida, da maneira usual, a métrica contravariante, os símbolos de Cristoffel, os tensores de curvatura e de Ricci, o escalar de curvatura e o tensor de Einstein.

```
on nero;
in "andy:procedures";
f1(q1,x,q1!,x,q1!,xx)$
f1(p1,x,p1!,x,p1!,xx)$
operator x;
let x(0)=t, x(1)=x, x(2)=y, x(3)=z;
array gg(3,3),h(3,3)$
gg(0,0):=e**q1$
gg(1,1):=-e**p1$
gg(2,2):=-x(1)**2$
gg(3,3):=-x(1)**2*sin(x(2))**2$
for i:=0:3 do h(i,i):=1/gg(i,i)$
array cs1(3,3,3),cs2(3,3,3)$
for i:=0:3 do for j:=i:3 do begin
    for k:=0:3 do

   cs1(j,i,k):= cs1(i,j,k):=
     (df(gg(i,k),x(j))+df(gg(j,k),x(i))
     -df(gg(i,j),x(k))/2;
for k:=0:3 do cs2(j,i,k):=cs2(i,j,k):=
     for p:=0:3 sum h(k,p)*cs1(i,j,p)
```

```
  end;
array r(3,3,3,3)$
for i:=0:3 do for j:=i+1:3 do for k:=i:3 do
   for l:=k+1:if k=i then j else 3 do begin
        r(j,i,l,k) := r(i,j,k,l) :=
          for q := 0:3  sum gg(i,q)*
             (df(cs2(k,j,q),x( l))-df(cs2(j,l,q),x(k))
        + for p:=0:3 sum (cs2(p,l,q)*cs2(k,j,p)
                          -cs2(p,k,q)*cs2(l,j,p)))$
        r(i,j,l,k) := -r(i,j,k,l);
        r(j,i,k,l) := -r(i,j,k,l);
        if i neq k or j>l
          then <<r(k,l,i,j) := r(l,k,j,i)
                            := r(i,j,k,l);
                 r(l,k,i,j) := -r(i,j,k,l);
                 r(k,l,j,i) := -r(i,j,k,l)>>
    end$
array ricci(3,3)$
for i:=0:3 do for j:=0:3 do
    write ricci(j,i) := ricci(i,j) :=
      for p : = 0:3 sum for q := 0:3
               sum h (p,q)*r(q,i,p,j);
rs := for i:= 0:3 sum for j:= 0:3 sum h(i,j)*
                                     ricci(i,j);
array einstein(3,3);
for i:=0:3 do for j:=0:3 do
         write einstein(i,j):=ricci(i,j)-
                                    rs*gg(i,j)/2;
```

Os resultados são obtidos como

```
                                          Q1
RICCI(0,0)  := RICCI(0,0)  := ( - E  *

              P1          Q1     2
   Q1,XX)/(2*E  ) + ( - E  *Q1,X )/(4*


    P1      Q1                    P1
   E  ) + (E  *Q1,X*P1,X)/(4*E  ) + ( -

    Q1         P1
   E  *Q1,X)/(E  *X)

RICCI(1,1)  := RICCI(1,1)  := Q1,XX/2 +
```

```
    2
Q1,X /4 + ( - Q1,X*P1,X)/4 + ( - P1,X)/X

RICCI(2,2)  := RICCI(2,2)  := (Q1,X*X)/(2*

   P1                        P1          P1
  E  ) + ( - P1,X*X)/(2*E   ) + ( - E

        P1
 + 1)/E

RICCI(3,3)  := RICCI(3,3)  := (Q1,X*

       2        P1                      2
  SIN(Y) *X)/(2*E   ) + ( - P1,X*SIN(Y)

         P1          2      P1
 *X)/(2*E   ) + SIN(Y) *( - E    + 1)/

 P1
E

TIME: 6386 MS


                    P1           2      P1
RS := ( - Q1,XX)/E    + ( - Q1,X )/(2*E

                                 P1
          ) + (Q1,X*P1,X)/(2*E    ) + ( - 2

                    P1                    P1
         *Q1,X)/(E   *X) + (2*P1,X)/(E   *

               P1         P1  2
    X) + 2*(E    - 1)/(E   *X )

TIME: 1006 MS

                      Q1          P1
EINSTEIN(0,0)  := ( - E  *P1,X)/(E   *X)

                    Q1      P1
                + E  *( - E    + 1)/(

                  P1  2
                 E  *X )
```

```
                                                    P1
EINSTEIN(1,1)  := (- Q1,X)/X + (E   - 1)

                  2
                 /X

                                    2       P1
EINSTEIN(2,2)  := ( - Q1,XX*X  )/(2*E   )

                                 2  2       P1
                  + ( - Q1,X  *X  )/(4*E

                                          2
                    ) + (Q1,X*P1,X*X  )/(

                       P1
                    4*E   ) + ( - Q1,X*X)

                      P1
                   /(2*E   ) + (P1,X*X)/(2*

                       P1
                    E   )


                                          2  2


EINSTEIN(3,3)  := ( - Q1,XX*SIN(Y)  *X  )/(

                      P1                  2
                   2*E   ) + ( - Q1,X  *

                            2  2       P1
                   SIN(Y)  *X  )/(4*E   )

                                         2
                  + (Q1,X*P1,X*SIN(Y)  *

                    2       P1
                   X  )/(4*E   ) + ( -

                                  2
                   Q1,X*SIN(Y)  *X)/(2*
```

```
 P1                  2
E  ) + (P1,X*SIN(Y)

          P1
*X)/(2*E  )
```

```
TIME:  2342 MS
```

8.3 Cálculos em Supersimetria

Esta área, aparentemente, não tem aproveitado os recursos da computação alg'ebrica que estão a seu dispor, embora geralmente necessite de cálculos um volume bastante grande.

Os cálculos, em Supersimetria, tendem a se tornar mais trabalhosos devido ao caráter fermiônico (não- comutativo) dos seus geradores.

Para facilitar o cálculo, comumente se faz uso do formalismo de superespaço e de supercampos, onde às quatro coordenadas do espaço-tempo x^μ , se acrescentam quatro coordenadas anticomutativas θ_α e $\bar{\theta}_{\dot{\alpha}}$.

No decorrer dos cálculos, tem-se que levar em conta, então, propriedades tais como

$$(\theta^\alpha \theta'_\alpha)(\theta^\beta \theta'_\beta) = -1/2(\theta^\alpha \theta_\alpha)(\theta'^\beta \theta'_\beta)$$

$$(\theta^\alpha \sigma^\mu_{\alpha\dot{\alpha}} \bar{\theta}^{\dot{\alpha}} q_\mu)(\theta^\beta \sigma^\nu_{\beta\dot{\beta}} \bar{\theta}^{\dot{\beta}} q_\nu) = -1/2(\theta^\alpha \theta_\alpha)(\bar{\theta}_{\dot{\beta}} \bar{\theta}^{\dot{\beta}}) q \cdot q$$

e é, também, conveniente a notação compacta

$$\theta^2 = \theta^\alpha \theta_\alpha$$

Poderíamos representar as coordenadas extras do superespaço por variáveis tornadas não-comutativas usando a declaração NOCOM vista na página 91. Isto não seria, porém, conveniente pois não teríamos a possibilidade de manter a notação compacta acima.

Ao invés disso, usamos variáveis tais como TE1, TEB1, etc., declaradas vetores, e comandos LET apropriados para introduzir as propriedades necessárias de anticomutação.

Como a operação . é diferente da *, uma vez que a primeira se refere a vetores, enquanto a segunda a escalares, a ordem dos fatores numa expressão não é alterada arbitrariamente pelo REDUCE, mas somente de acordo com as regras fornecidas.

Introduzimos, ainda, o operador SIG para simular as propriedades das matrizes de Pauli. A integração sobre as coordenadas não-comutativas,

que corresponde na prática a uma diferenciação, é feita derivando-se com relação ás massas atribuidas aos vetores correspondentes.

Abaixo temos o programa que usamos para calcular um (super)gráfico de Feynman a 2-loop[1]

```
vector p,q1,q2,q3,q4,x1,x2,x3,q,t,tb,tp,tbp,psi,te1,
      teb1,te2,teb2,te3,teb3;
mass te1=t1,teb1=tb1,te2=t2,teb2=tb2,te3=t3,teb3=tb3
       ,t=0,tb=0;
forall tb,q let sig(te1,q,tb)*t1**2=0,
                sig(te2,q,tb)*t2**2=0,
                sig(te3,q,tb)*t3**2=0;
forall t,q let sig(t,q,teb1)*tb1**2=0,
               sig(t,q,teb2)*tb2**2=0,
               sig(t,q,teb3)*tb3**2=0;
forall t,tb,tbp,q let sig(t,q,tb)*(tb.tbp)=
                      -(tb.tb)*sig(t,q,tbp)/2;
forall t,q1,q2,tb,tbp let sig(t,q1,tb)*sig(t,q2,tbp)=
                          -(t.t)*(tb.tbp)*q1.q2/2;
forall t,tp,tb,q let sig(t,q,tb)*(t.tp)=
                    -(t.t)*sig(tp,q,tb)/2;
forall t,tp,q1,q2,tb let sig(t,q1,tb)*sig(tp,q2,tb)=
                         -(t.tp)*(tb.tb)*q1.q2/2;
forall t,tb,q let sig(t,q,tb)*sig(t,q,tb)=
                  -(t.t)*(tb.tb)*q.q/2;
forall t,tp let (t.tp)*(t.tp)=-(t.t)*(tp.tp)/2;
forall t,tp,tb let (t.tb)*(tp.tb)=-(tb.tb)*(t.tp)/2;
forall t let t1**2*(te1.t)=0,
             t2**2*(te2.t)=0,
             t3**2*(te3.t)=0,
         expg(te1,-p,teb1)*expg(te2,p,teb2)*
         e**(i*p.(x1-x2)),
    dddelta12(p,x1,te1,teb1,x2,te2,teb2)=
         i*(fp(p)+gp(p)*(te1.te1)+hp(p)*(teb2.teb2)
          +ip(p)*sig(te1,p,teb2)
          +jp(p)*(te1.te1)*(teb2.teb2))*
         expg(te1,-p,teb1)*expg(te2,-p,teb2)*
         e**(i*p.(x1-x2)),
   dddelta21(p,x1,te1,teb1,x2,te2,teb2)=
         i*(fps(p)+gps(p)*(teb1.teb1)
           +hps(p)*(te2.te2)-ip(p)*sig(te2,p,teb1)
```

[1]vide Renato P. dos Santos, *Using REDUCE in Supersymmetry*, preprint CBPF- NF-047/87

```
          +jp(p)*(teb1.teb1)*(te2.te2))*
         expg(te1,p,teb1)*expg(te2,p,teb2)*
         e**(i*p.(x1-x2))
v1:=-2*i/3*g**2**dddelta12(q1,x1,te1,teb1,x2,te2,teb2)
       *dddelta12(q2,x1,te1,teb1,x2,te2,teb2)
       *dddelta12(q3,x1,te1,teb1,x2,te2,teb2)$
v2:=df(v1,t1,2,tb2,2)/4;
```

E em seguida, o resultado obtido

```
v2 :=

        (X1.Q3*I  + X1.Q2*I  + X1.Q1*I  - X2
       E

         .Q3*I  - X2.Q2*I  - X2.Q1*I)
                                    *(1/3*

         Q3.Q2*IP(Q3)*IP(Q2)*FP(Q1)  + 1/

       3*Q3.Q1*IP(Q3)*IP(Q1)*FP(Q2)  +

       1/3*Q2.Q1*IP(Q2)*IP(Q1)*FP(Q3)

        - 2/3*GP(Q3)*HP(Q2)*FP(Q1)  - 2

       /3*GP(Q3)*HP(Q1)*FP(Q2)  - 2/3*

       GP(Q2)*HP(Q3)*FP(Q1)  - 2/3*GP(

       Q2)*HP(Q1)*FP(Q3)  - 2/3*GP(Q1)*

       HP(Q3)*FP(Q2)  - 2/3*GP(Q1)*HP(

       Q2)*FP(Q3)  - 2/3*FP(Q3)*FP(Q2)*

       JP(Q1)  - 2/3*FP(Q3)*FP(Q1)*JP(

       Q2)  - 2/3*FP(Q2)*FP(Q1)*JP(Q3))

      *EXPG(TE2,  - Q3,TEB2)*EXPG(TE2,  -

      Q2,TEB2)*EXPG(TE2,  - Q1,TEB2)*EXPG

      (TE1,  - Q3,TEB1)*EXPG(TE1,  - Q2,

                                        2
      TEB1)*EXPG(TE1,  - Q1,TEB1)*G
```