راهنمای استفاده از نرم افزار REDUCE

فهرست

مقدمه

نرم افزار REDUCE یک ابزار محاسبات جبری در ریاضی, فیزیک و مهندسی است. تولید این نرم افزار حاصل تلاش مشترک افراد بسیاری است ولی منشأ اصلی شکل گیری آن کار آنتونی هرن[1] در ۱۹۶۳ است. REDUCE برای اولین بار در همین سال توزیع شد و از این زمان به بعد صدها نفر از راه های مختلف در توسعه‌ی این نرم افزار دخیل بوده اند. برای مثال, جان فیچ[2], هربرت ملنک[3], وین فرید نون[4], آرتور نورمن[5] و اِبرهارد شروفر[6] از افرادی هستند که در شکل گیری هسته ی REDUCE و بسته ها (package) ی مرتبط شرکت مداوم داشته اند.

برخی قابلیت های نرم افزار REDUCE عبارتند از:

- مرتب سازی و بسط توابع چندجمله ای و توابع گویا
- انجام انواع جاگذاری ها
- ساده سازی به صورت اتوماتیک یا تحت کنترل کاربر
- انجام محاسبات ماتریسی
- تعیین دقت دلخواه در محاسبه ی اعداد صحیح یا حقیقی
- امکان تعریف توابع جدید و توسعه ی syntax برنامه
- مشتق گیری و انتگرال گیری تحلیلی
- فاکتورگیری از چندجمله ای ها
- حل معادلات جبری مختلف
- امکان کنترل شکل خروجی برنامه
- انجام محاسبات مربوط به انواع توابع خاص (بسل, لژاندر, گاما, زتا و ...)
- انجام محاسبات مربوط به ماتریس های دیراک, قابل استفاده برای محاسبات فیزیک انرژی های بالا

[1] Anthony Hearn
[2] John Fitch
[3] Herbert Melenk
[4] Winfried Neun
[5] Arthur Norman
[6] Eberhard Schrüfer

آشنایی با محیط نرم افزار *REDUCE*

برای آشنایی با محیط نرم افزار REDUCE و شیوه ی به کارگیری آن در حل مسائل متداول فیزیک و نسبیت عام, قبل از پرداختن به اصل راهنما, مثال زیر را برای محاسبه ی نرده ای کرچمن با استفاده از متریک شوارتس شیلد مطرح می کنیم. در این مثال از روابط زیر استفاده کرده ایم:

$$g = \begin{pmatrix} \frac{r - r_s}{r} & 0 & 0 \\ 0 & \frac{-r}{r - r_s} & 0 \\ 0 & 0 & -r^2 \sin\theta^2 \end{pmatrix}$$

$$\Gamma^i_{kl} = \frac{1}{2} g^{im} (g_{mk,l} + g_{ml,k} - g_{kl,m})$$

$$R^\rho_{\sigma\mu\nu} = \partial_\mu \Gamma^\rho_{\nu\sigma} - \partial_\nu \Gamma^\rho_{\mu\sigma} + \Gamma^\rho_{\mu\lambda}\Gamma^\lambda_{\nu\sigma} - \Gamma^\rho_{\nu\lambda}\Gamma^\lambda_{\mu\sigma}$$

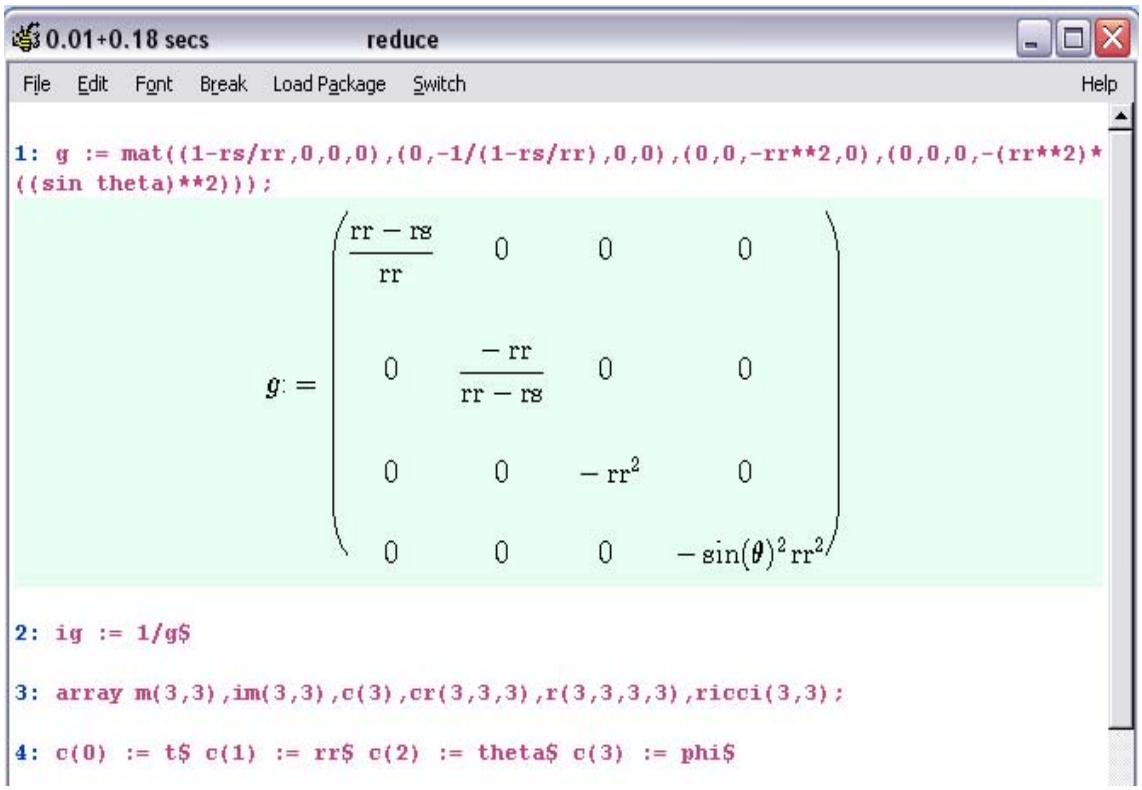
انجام این محاسبه با استفاده از بسته ی atensor که برای محاسبات تانسوری در نرم افزار REDUCE تعبیه شده است ساده تر خواهد بود. ولی برای آشنایی با دستورهای اصلی و اولیه ی نرم افزار, این محاسبه را با استفاده از آرایه ها انجام می دهیم.

ابتدا به کد زیر توجه کنید:

```
1: g := mat((1-rs/rr,0,0,0),(0,-1/(1-rs/rr),0,0),(0,0,-rr**2,0),(0,0,0,-(rr**2)*((sin
theta)**2)));
2: ig: = 1/g$
3: array m(3,3),im(3,3),c(3),cr(3,3,3),r(3,3,3,3),ricci(3,3);
4: c(0) := t$ c(1) := rr$ c(2) := theta$ c(3) := phi$
8: for i := 0:3 do<< for j := 0:3 do<< m(i,j) := g(i+1,j+1);  im(i,j) := ig(i+1,j+1)
>>;>>;
9: procedure christoffel(a,b,y,x);
9: for i := 0:3 do<< for j := 0:3 do<< for k := 0:3 do<< for s := 0:3 do<< y(i,j,k) :=
y(i,j,k)+(b(k,s)/2)*(df(a(j,s),x(i))+df(a(i,s),x(j))-df(a(i,j),x(s))) >>;>>;>>;>>;
```

```
10: christoffel(m,im,cr,c);
11: for i := 0:3 do<< for j := 0:3 do<< for k := 0:3 do<< for l := 0:3 do<< for n := 0:3
do<< r(i,j,k,l) := r(i,j,k,l)+cr(k,n,i)* cr(l,j,n)-cr(l,n,i)*cr(k,j,n) >>; r(i,j,k,l) :=
r(i,j,k,l)+df(cr(l,j,i),c(k))-df(cr(k,j,i),c(l)) >>;>>;>>;>>;
12: array r1(3,3,3,3),r2(3,3,3,3);
13: for i := 0:3 do<< for j := 0:3 do<< for k := 0:3 do<< for l := 0:3 do<< for n := 0:3
do<<  r1(i,j,k,l) := r1(i,j,k,l) + m(i,n)*r(n,j,k,l) >>;>>;>>;>>;>>;
14: for i := 0:3 do<< for j := 0:3 do<< for k := 0:3 do<< for l := 0:3 do<< for n := 0:3
do<< for s := 0:3 do<< for p := 0:3 do<< r2(i,j,k,l) := r2(i,j,k,l) +
im(j,n)*im(k,s)*im(l,p)*r(i,n,s,p) >>;>>;>>;>>;>>;>>;>>;
15: kretchman := 0$
16: for i := 0:3 do<< for j := 0:3 do<< for k := 0:3 do<< for l := 0:3 do<< kretchman
:= kretchman + r1(i,j,k,l)*r2(i,j,k,l) >>;>>;>>;>>;
```

شیوه ی اجرای این کد را در شکل زیر می بینید.



```
8: for i := 0:3 do<< for j := 0:3 do<< m(i,j) := g(i+1,j+1);  im(i,j) := ig(i+1,
j+1) >>;>>;

9: procedure christoffel(a,b,y,x);
9: for i := 0:3 do<< for j := 0:3 do<< for k := 0:3 do<< for s := 0:3 do<< y(i,j
,k) := y(i,j,k)+(b(k,s)/2)*(df(a(j,s),x(i))+df(a(i,s),x(j))-df(a(i,j),x(s))) >>;
>>;>>;>>;
```

christoffel

```
10: christoffel(m,im,cr,c);

11: for i := 0:3 do<< for j := 0:3 do<< for k := 0:3 do<< for l := 0:3 do<< for
n := 0:3 do<< r(i,j,k,l) := r(i,j,k,l)+cr(k,n,i)* cr(l,j,n)-cr(l,n,i)*cr(k,j,n)
>>; r(i,j,k,l) := r(i,j,k,l)+df(cr(l,j,i),c(k))-df(cr(k,j,i),c(l)) >>;>>;>>;>>;

12: for i := 0:3 do<< for j := 0:3 do<< for k := 0:3 do<< ricci(i,j) := ricci(i,
j)+r(k,i,j,k) >>;>>;>>;

13: array r1(3,3,3,3),r2(3,3,3,3);

14: for i := 0:3 do<< for j := 0:3 do<< for k := 0:3 do<< for l := 0:3 do<< for
n := 0:3 do<<  r1(i,j,k,l) := r1(i,j,k,l) + m(i,n)*r(n,j,k,l) >>;>>;>>;>>;>>;

15: for i := 0:3 do<< for j := 0:3 do<< for k := 0:3 do<< for l := 0:3 do<< for
n := 0:3 do<< for s := 0:3 do<< for p := 0:3 do<< r2(i,j,k,l) := r2(i,j,k,l) + i
m(j,n)*im(k,s)*im(l,p)*r(i,n,s,p) >>;>>;>>;>>;>>;>>;>>;

16: kretchman := 0$

17: for i := 0:3 do<< for j := 0:3 do<< for k := 0:3 do<< for l := 0:3 do<< kret
chman := kretchman + r1(I,j,k,l)*r2(I,j,k,l) >>;>>;>>;>>;

18: kretchman;
```

$$\frac{12\,rs^2}{rr^6}$$

خط اول تعریف ماتریس g است که همان متریک شوارتس شیلد است. ; در انتهای خط نتیجه را چاپ می کند. ولی در بقیه ی خط ها وجود $ باعث می شود که نتیجه روی صفحه چاپ نشود.

خط دوم تعریف ماتریس معکوس متریک است. در خط سوم آرایه های مورد نیاز با ابعاد مناسب تعریف شده اند که به ترتیب متریک, معکوس متریک, مختصات, نمادهای کریستوفل و تانسور انحنا را نمایش می دهند. در خط چهارم (که در واقع شامل چهار خط دستور است) نام مختصات در آرایه ی c تعیین شده و در خط هشتم مقداردهی آرایه ی متریک و معکوس آن با استفاده از دارایه های ماتریس g و ig انجام می شود.

در این مرحله آماده ایم تا تابع مورد نیاز برای محاسبه ی نمادهای کریستوفل را تعریف کنیم. این تابع را در خط نهم تحت عنوان christoffel(a,b,y,x) تعریف کرده ایم. ورودی های این تابع به ترتیب آرایه های

متریک, معکوس متریک, کریستوفل (که قرار است در تابع مقداردهی شود) و مختصات هستند ولی توجه کنید لزومی ندارد که ورودی تابع به هنگام تعریف آن هم نام با کمیت های مورد نظر ما باشند.

با صدا کردن این تابع, بعد از تعریف آن, اجزای آرایه ی cr بر اساس رابطه ی موجود برای محاسبه ی نمادهای کریستوفل در یک حلقه ی for چهار متغیره مقداردهی می شوند. اکنون می توان با استفاده از نمادهای کریستوفل تانسور انحنا را محاسبه کرد. این کار را در خط ۱۱ با استفاده از یک حلقه ی دیگر و با توجه به رابطه ی تانسور انحنا با نمادهای کریستوفل انجام داده ایم.

برای محاسبه ی نرده ای کرچمن باید عبارت $R_{ijkl}R^{ijkl}$ را محاسبه کنیم. ولی باید دقت کنید که در تعریف آرایه ی انحنا چون با آرایه سروکار داریم و نه با یک تانسور, بین اندیس بالا و پایین هیچ تفاوتی وجود ندارد. پس با توجه به اینکه در برنامه R^{i}_{jkl} را حساب کرده ایم باید با استفاده از متریک اندیس ها را تنظیم کنیم. در خطوط ۱۲ تا ۱۴ در واقع همین کار را انجام داده ایم. در بسته ی atensor قانون جمع انیشتین تعریف شده است, بنابراین این مشکل در آنجا وجود نخواهد داشت.

در نهایت, در خط سیزدهم محاسبه ی نرده ای کرچمن با استفاده از یک حلقه ی for کامل شده است. همان طور که مشاهده می کنید نرده ای کرچمن به درستی محاسبه شده است.

$$Kretchman\ Scalar = \frac{12r_s^2}{r^6}$$

۱.تعریف *identifier*

برای نام متغیرها, برچسب عبارت دستور *go to* , نام آرایه ها, ماتریس ها, عملگرها و *procedure* ها از *identifier* استفاده می کنیم که می تواند شامل حروف و اعداد باشد ولی اولین حرف آن نباید عدد باشد.

بیشترین تعداد حروف مجاز برای یک *identifier* در سیستم های مختلف متفاوت است ولی معمولاً حدود ۱۰۰ حرف است. هرچند بهتر است برای چاپ راحت تر تعداد حروف کمتر از ۲۵ انتخاب شود.

۲.دستورها (*Commands*)

Syntax:

- *; and $*

این دو حد گزاره ها را تعیین می کنند. که نتیجه را چاپ می کند ولی $ نتیجه را چاپ نمی کند.

✓ داخل یک گروه <<...>> یا *block* هر دو برای جدا کردن گزاره ها به کار می روند و فرقی با هم ندارند. چون از داخل *block* بدون وجود دستور *return* چیزی چاپ نمی شود.

- *% or comment*

برای نوشتن توضیح در داخل برنامه استفاده می شوند.

✓ ٪: هر چیزی بعد از ٪ تا انتهای خط اجرا نمی شود از جمله ; و $

✓ *Comment*: از کلمه ی *comment* تا اولین ; و یا $ اجرا نمی شود. *comment* برای گذاشتن توضیح چند خطی مناسب است :

*1: df (x^**3+y,x)% this is a comment;*

1: :

$3x^2$

; خط اول اجرا نشده است.

*1: x:a^** comment-a is positive*

1: a is an integer ;;

$X:=a^2$

اولین ; برای پایان *comment* است و دومی برای چاپ نتیجه.

- *group <<...>>*

در جایی که *REDUCE* انتظار یک گزاره را دارد، چند گزاره را با هم یک گروه می کند.

مثلاً در *if... then* یا *repeat... until* یا *while...do* داخل حلقه یا عبارت شرطی باید یگ گزاره داشته باشیم.

یا قبل از *do* که یک گزاره لازم داریم.

✓ گزاره های داخل گروه با *;* یا *$* جدا می شوند و فرقی با هم ندارند.

✓ گروه مقدار آخرین گزاره را بر می گرداند. پس اگر بین آخرین گزاره و براکت بسته (>>) *;* یا *$* قرار بگیرد، مقدار صفر یا *nill* (آخرین گزاره) برگردانده می شود. پس در حالت کلی بعد از آخرین گزاره *;* یا *$* نمی گذاریم.

- *begin ... end (block)*

یک *block* می تواند عملیاتی را انجام دهد مثل

1: begin for i:=1:3 do write I end;

1

2

3

ولی بر خلاف *group* تا وقتی دستور *return* داخل آن نباشد مقداری برنمی گرداند.

✓ دستور *return* باید آخرین دستور قبل از *end* باشد.

1: begin scalar n; n:=1; b:=for i:=1:4 product (x-i); return n end;

1

2: b;

$X^4-10x^3+35x^2-50x+24$

✓ متغیرهای محلی *(local)* در داخل *block* بلافاصله بعد از *begin* باید تعریف شوند و می‌توانند *scalar* *real* یا *integer* باشند. ولی آرایه ای که داخل *block* تعریف می شود، در هر حالت *global* است.

✓ دستور *let* هم در *block* تأثیر *global* دارد.

✓ قبل از *end* ، *;* یا *$* لازم نیست ولی بعد از آن *;* یا *$* یا براکت بسته ای متناظر با براکتی که قبلاً باز شده، باید باشد.

- *Goto or go to*

<label>: <statement>

go to <labeled-statement>;

✓ دستور *goto* فقط داخل یک *block* استفاده می شود و برنامه به خطی که برچسب (*label*)مورد نظر را دارد می رود.

✓ بهتر است به شکل *go to* استفاده شود.

✓ داخل *block* فقط گزاره های بالاترین سطح می توانند برچسب گذاری شوند نه گزاره های مثلاً داخل *<<…>>* یا *while…do* و … که در *block* وجود دارند.

- *for*

1) *for <var>:= <number> step <number> until <number> <action> <exprn>*
2) *for <var>:= <number> : <number> <action> <exprn>*
3) *for each <var> in <list> <action> <exprn>*

✓ *<var>* می تواند هر متغیری به غیر از *t* و *nil* باشد.

✓ *<action>* می تواند عملیاتی مانند *join, collect, sum, product, do* و … باشد و به دنبال آنها باید یک گزاره ی تنها یا *group* یا *block* بیاید.

چند مثال:

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

55

1: for a:=-2 step 3 until 6 product a;

-8

1: m:=0$

2: for s:=10 step -1 until 3 do << d:=10^s; m:=m+d>>;

3:m;

520

1: for i:=1:4 collect 1/i ;

$$\left\{1,\frac{1}{2},\frac{1}{3},\frac{1}{4}\right\}$$

*1: for each x in {q,r,s} sum x**2;*

$q^2+r^2+s^2$

- ✓ اگر حد بالا از حد پایین کمتر باشد هیچ عملیاتی انجام نمی شود.
- ✓ متغیر تکرار کننده *(i,a,s,…)* در گزاره ی *for* , *local* است و مقدار یک متغیر با همان نام را تغییر نمی دهد
- ✓ می توان چند گزاره ی *for* را به صورت تو در تو استفاده کرد به شرطی که چیزی که گزاره‌های داخلی برمی گردانند برای گزاره های خارجی معنی دار باشد.

- *if*

if <condition> then <statement> else <statement>

- ✓ *<condition>* باید یک گزاره منطقی یا ترکیبی از آنها (به وسیله ی *and* , *or*) باشد.
- ✓ گزاره ی منطقی: مقدار *Boolean* برمی گرداند یعنی اگر صفر یا *nil* باشد *false* برمی‌گرداند و اگر هر مقدار دیگری باشد، *true* برمی گرداند.
- ✓ *<statement>* باید یک گزاره ی تنها یا یک *group* یا *block* باشد.

چند مثال:

1: x:=9$

2: if number p x and x<20 then y:=sqrt (x) else write "illegal";

3

1: x: =12$

2: a:=if x<5 then 100 else 150;

a:=150

*3: b:=u**(if x<10 then 2);*

b:1

*4: bb:=u**(if x>10 then 2);*

bb:=u²

- ✓ اگر *else* نداشته باشیم و شرط *if* برقرار نشود، مقدار گزاره صفر خواهد بود مثل خط *3* در مثال دوم
- ✓ اگر به جای *<undition>* گزاره ی غیر شرطی بگذاریم، مثل یک ثابت، *true* در نظر گرفته می شود.
- ✓ این دستور تو در تو هم می تواند استفاده شود:

If <condition> then if <condition> then <action> else <action>

If <condition> then <action> else if <condition> then <action> else <action>

- *Repeat*

Repeat <statement> until <condition>

- ✓ *<statement>* باید یک گزاره ی تنها یا یک *group* یا *block* باشد.
- ✓ *<condition>* باید یک گزاره ی منطقی یا ترکیبی از آنها (به وسیله ی *and* و *or*) باشد.
- ✓ *<statement>* حداقل یک بار اجرا می شود.
- ✓ مواظب *loop* بی نهایت باشید!

```
1: m:=3$
2: repeat <<write 100*x*m; m:=m-1>> until m=0;
        300x
        200x
        100x
1: m:=-1$
2:repeat <<write m; m:=m+1>> til m>=0;
        -1
```

- *Return*

این دستور یک مقدار را از داخل *block* برمی گرداند.

begin <statement> return <expression> end;

مقدار *<experession>* برگردانده می شود.

- ✓ چند مثال:

```
1: begin write “yes”; return a end;
        yes
        a

1: procedure dump(a);
1: begin if number(a) then return a else return 10 end;
        Dump
2: dump(-5);
```

-5

3: dump(x);

10

- ✓ *Return* (در صورت وجود) همیشه آخرین گزاره قبل از خارج شدن از *block* است.
- ✓ بدون *return* هیچ مقداری برگردانده نمی شود ولی عملیات *block* انجام می شود.
- ✓ اگر *return* در داخل *group* یا *if…then…else* استفاده شود, اینها باید داخل *block* باشندو برای *for* و *repeat…until* و *while…do* نمی توان از *return* استفاده کرد حتی اگر داخل *block* باشند.
- ✓ در *block* های تو در تو , *return* در *block* داخلی, مقدار را به *block* بعدی بر می گرداند نه به خارج از همه ی *block* ها.

- *Procedure*

از این دستور برای تعریف کردن یک عملگر ریاضی به عنوان یک تابع با متغیرهای آن استفاده می‌شود.

Procedure <identifier>(<arg>,<arg>,…); <body>

- ✓ *<body>* باید یک گزاره ی تنها یا یک *group* یا *block* باشد.

1: procedure fac(n);

1: if not (fixp (n) and n>=0)

1: then rederr "choose non neg-integer only"

1: else for i:=o:n-1 productu i+1 ;

Fac

2: fac(0);

1

3: fac(-5);

Choose nonneg-integer only

وقتی *REDUCE* ، *procedure* را تجزیه کرده و با موفقیت به فرم قابل استفاده خود در آورد اسم آن را چاپ می کند.

- ✓ داخل تعریف یک *procedure* نمی توان یک *procedure* دیگر تعریف کرد ولی می توان یک *procedure* دیگر یا خود *procedure* را به شکل بازگشتی صدا کرد.
- ✓ متغیرهای *procedure* صوری هستند یعنی مثلاً متغیر *x* ربطی به *x* ای که در برنامه تعریف شده ندارد. و مقدار آن را تغییر نمی دهد.

✓ برای متغیری که *procedure* را صدا می کند، اگر داخل *procedure* چیزی به آن *assign* شده باشد *(:=)* مقدارش فقط داخل *procedure* تغییر می کند ولی *let* مقدار آن به صورت *global* تغییر می دهد. دستور *clear* هم برای متغیرهای صوری تأثیر *global* دارند.

✓ اپراتور و آرایه می توانند متغیر *procedure* باشند و در هر حال مقدارشان به صورت *global* تغییر می کند ولی ماتریس نمی تواند روی *procedure* باشد.

✓ هر پارامتری که صوری نباشد (صدا زننده ی *procedure* نباشد) و *local* هم نباشد (که بلافاصله بعد از *begin* در *block* تعریف شود) *global* در نظر گرفته می شود.

✓ اگر قرار باشد *procedure* یک مقدار را برگرداند دستور *return* باید آخرین گزاره قبل از خروج از *procedure* باشد.

- *write*

write <item> , <item> , ….

1: write a," ", sin(x), "this is a string" ;
A sin(x)this is a string
1: array m(10);
*2: for i:=1:3 do write m(i):=2*i ;*
m(1)=2
m(2)=4
m(3)=6
3: m(3);
6

✓ *Item* های یک دستور *write* همه در یک خط چاپ می شون مگر اینکه خیلی طولانی باشد.

- *define*

کاربرد دستور *define* در مثال زیر واضح است:

1: define is = :=, xx=y+z ;
2: a is 10 ;
a :=10

*3: xx**2 ;*

$y^2+2yz+z^2$

4: xx :=10 ;

y+z :=10

✓ در به کار بردن *define* باید مراقب بود چون به راحتی نمی توان آن را بدون بستن برنامه به حال اول برگرداند.

- *on and off*

off برای خاموش کردن یک *switch* و *on* برای به کار انداختن یک *switch* استفاده می شود:

on switch-name;

off switch-name;

✓ خاموش و روشن کردن *switch* ها در *toolbar* قسمت *switch* هم امکان پذیر است.

- *while*

while <condition> do <statement>

✓ *<condition>* باید یک گزاره ی منطقی یا ترکیبی از آنها (به وسیله ی *and* و *or*) باشد.

✓ *<statement>* باید یک گزاره ی تنها یا یک *group* یا *block* باشد.

1: a:=10 ;

2: while a<=12 do << write a; a:=a+1>> ;

10

11

12

- *let*

برای تعریف قوانین جایگزینی

Let <identifier> = <expression>

✓ *<identifier>* می تواند هر چیزی به غیر از آرایه باشد از جمله یک *<expression>*

1: let a=sin(x);

2: b: a;

b:sin(x)

3: exp(a) ;

$e^{sin(x)}$

✓ فرق *let* با اپراتور := این است که =: مقدار عبارت سمت چپ خود را *fix* می کند ولی با *let*، متغیر تغییرات بعدی را دنبال می کند:

4: let c=a ;

5: a:= x^n2 $

6: exp(a) ;

e^{x^2}

7: exp^(b) ;

$e^{sin(x)}$

8: exp^(c) ;

e^{x^2}

می بینیم که *c* تغییر *a* را دنبال کرده است ولی *b* که ابتدا مقدار *a* به آن *assign* شده بود، همان مقدار اول را حفظ کرده است.

1: operator h;

2: let h(u,v)=u-v;

3: h(u,v);

u-v

4: h(x,y);

h(x,y)

1:array q(10);

2:let q(1)=15;

***** substitution for o not allowed*

✓ وقتی در تعریف عملکرد اپراتور از *let* استفاده می کنیم، این عملکرد فقط برای متغیرهای دقیقاً به همان نامی که استفاده شده معتبر است.

✓ آرایه به هنگام تعریف با صفر مقداردهی شده، بنابراین *let q(1)=15;* منجر به پیغام خطا می شود. *(let o=15;)* ولی اگر اعضای آرایه یا درایه های های ماتریس محتوی یک متغیر یا عبارت معتبر برای *let* باشند تأثیر *let* به صورت *global* روی آن متغیر و عبارت اعمال می شود، یعنی نه فقط روی متغیر داخل ماتریس یا آرایه بلکه متغیر در همه جای برنامه. برای جلوگیری از این عوارض بهتر است *let* (و همین طور *clear*) برای آرایه و ماتریس استفاده نشود و به جای آن از := استفاده شود.

✓ فقط تکه تک اجزای آرایه می توانند برای *let* سمت چپ تساوی قرار بگیرند. ولی ماتریس می تواند به عنوان یک کل سمت چپ تساوی باشد به شرطی که سمت راست هم ماتریس باشد.

✓ متغیرهای *local* (که داخل *block* تعریف می شوند) نمی توانند سمت چپ تساوی *let* قرار بگیرند ولی کل *block* می تواند سمت راست تساوی *let* باشد با این *syntax*:

for all (vars) let <operator> (<vars>) = <block>

✓ *let* برای *active* کردن یک *rule list* هم به کار می رود:

```
1: trigl :={cos(~x)*cos(~y) => (cos(x+y)+cos(x-y))/2 ,
        Cos(~x)*sin(~y) => (sin(x+y)-sin(x-y))/2}$
2: let trigl ;
3: cos(a)*cos(b) ;
```

$$\frac{\cos(a-b)+\cos(a+b)}{2}$$

✓ چند نکته در استفاده از *let*:

1) *let A*B=C : A*B* را مساوی *C* قرار می دهد.

2) *let A+C=C : A* را مساوی *C-B* قرار می دهد.

3) *let A-B=C : A* را مساوی *C+B* قرار می دهد.

4) *let A/B=C : A* را مساوی *B*C* قرار می دهد.

- *clear*

برای حذف کردن تساوی ها یا قوانین جایگزینی

Clear <identifier>

✓ *<identifier>* می تواند اسکالر، ماتریس، آرایه یا نام *procedure* باشد.

```
1: array a(2,3) ;
2: a(2,2) :=15$
3: clear a;
4: a(2,2);
        Declare A operator? (Y or N)
5: let x=y+z;
6: sin(x);
        Sin(y+z)
7: clear x;
8: sin(x);
```

Sin(x)

9: let x^5=7;

10: clear x;

11: x^5;

7

12: clear x^5;

13: x^5;

x^5

✓ اگر *clear* برای اجزای آرایه استفاده شود, در صورتی که این جزء محتوی عبارت غیر معتبر برای *clear* باشد (مثل عدد ثابت) پیغام *error* می دهدو در غیر این صورت متغیر موجود در این جزء پاک می شود. برای ماتریس ها اگر عبارت غیر معتبر باشد *error* ظاهر می شود و در غیر این صورت هیچ عملی انجام نمی شود.

✓ برای اینکه اجزای آرایه یا ماتریس را مساوی صفر قرار دهید از *clear* استفاده نکنید. این کار نتیجه ی مورد انتظار شما را نخواهد داشت. چون اجزا نه محتوی صفر خواهد بود و نه محتوی هیچ چیز دیگر.

✓ همان طور که در مثال آخر می بینید وقتی یک عبارت (توانی, ضربی و ...) با چیزی جایگزین شده, برای پاک کردن این جایگزینی کل عبارت تا قبل از = باید در آرگومان *clear* نوشته شود.

- *Clearrules*

برای پاک کردن یک *rule list* از *clearrules* استفاده می کنیم:

1:trig 1: ={cos(~x)cos(~y) => (cos(x+y)+cos(x-y))/2;

cos(~x)sin(~y) => (sin(x+y)-sin(x-y)/2}$

2: let trigl ;

*3: cos(a)*cos(b);*

$$\frac{\cos(a+b)+\cos(a-b)}{2}$$

4: clearrules trigl ;

*5: cos(a)*cos(b);*

cos(a)cos(b)

- *for all*

for all <identifier> let <let statement>

یا:

for all < identifier> such that <condition> let <let statement>

1: for all x let f(x)=sin(x^n2);

Declare f operator? (Y or N)

2: Y

3: f(a);

Sin (a^2)

4: operator pos;

5: for all x such that x>=0 let pos(x) = sqrt(x+1) ;

6: pos (5);

$\sqrt{6}$

7: pos(-5);

Pos(-5)

8: clear pos;

9: pos(5);

declare pos operator? (Y or N)

✓ برای پاک کردن جایگزینی توسط *for all* به این مثال توجه کنید:

1: for all a such that a>o let x^na =1;

2: x^n4;

1

3: clear x^na ;

**** x^na Not Found*

4: for all a clear x^na;

5: x^n4;

1

6: for all a such that a>0 clear x^na ;

7: x^n4;

x^4

✓ *for all …let* معادل تعریف یک *procedure* است با این تفاوت که در *procedure* مقدار پارامتر صوری (صدا زننده ی *procedure*) به صورت *global* عوض نمی شود حتی اگر داخل *procedure* مقداری به آن *assigne* شده باشد (در این رابطه آرایه ها استثنا هستند)

دستورهای دیگری هم تحت عنوان *general commands* داریم که توضیح آنها به اختصار چنین است:

1) *bye* ← از یک سطح برنامه خارج می شود و به سطح *(level)* بعدی می رود و اگر در بالاترین سطح باشیم از *REDUCE* خارج می شود. این دستور معادل *quit* است

2) *display(<n>)* ← *<n>* خط قبل را چاپ می کند و اگر پرانتز خای باشد یا *<n>* از تعداد خطوط بیشتر باشد همه ی خطوط را از اول چاپ می کند.

3) *rederr <message>* ← یک پیغام *error* از داخل *procedure* یا *block* چاپ می کند.

4) *retry* ← آخرین گزاره ای که منجر به *error* شده است را دوباره امتحان می کند.

5) *load-package "<package-name>"* ← *package* مورد نظر را *load* می کند و در قسمت *toolbar* برنامه هم قابل دسترسی است.

۳.عملگرها (*Operators*)

Syntax

- *Assign :=*

مقدار سمت راست را به سمت چپ *assign* می کند.

✓ *:=* از سمت راست اعمال می شود:

```
1: a:=b:=c $
2: a;
        c
3: b;
        c
```

همه ی عبارتها در این زنجیره به غیر از عبارت آخر برابر با *c* قرار داده می شوند.

✓ سمت چپ ممکن است یک *expression* باشد:

```
1: y+b:=c &
2: y;
```

-(b-c)

✓ = : برای اجزای آرایه ها کاربرد دارد ولی برای کل آرایه نه. ولی برای درایه های ماتریس یا کل یک ماتریس می توان از آن استفاده کرد.

✓ استفاده از ساختار بازگشتی مثل *a:a+b* هم مانعی ندارد.

- *equal sign* =

در گزاره های شرطی (مثل *if…then…else* یا *repeat…until*) و در تعریف *equation* به کار می‌رود.

```
1: a:=4 $
2: if a=10 then write "yes" lese write "no";
        No
```

- / * – +

کاربرد + و – واضح است. استفاده از آنها برای ماتریس ها و *equation* ها هم ممکن است.

اپراتور * می تواند برای ضرب ماتریسها با بعد مناسب هم استفاده شود.

✓ در عبارتی مثل *2x ,REDUCE* به طور ضمنی اپراتور * را در نظر می گیرد چون *identifier* نمی تواند با عدد شروع شود.

اپراتور / هم برای ماتریس های مربعی می تواند استفاده شود: *A/B=A*B^-1* به شرطی که *B* وارون پذیر باشد. *1/A* هم معکوس *A* را می دهد.

```
1: 100/6 ;
```

$$\frac{50}{3}$$

```
2: 16/2/x ;
```

$$\frac{8}{x}$$

```
3: on rounded ;
4: 55/4;
        8.75
```

- ** or ^

```
1: x**y**z;
```

x^{yz}

*2: x**(Y**Z);*

x^{y^z}

3: on rounded;

*4: 2**pi ;*

8-82497782708

- ✓ ماتریس مربعی می تواند به توان مثبت یا منفی (به شرط معکوس پذیری) برسد.
- ✓ عبارت های اسکالر و *equation* ها می توانند به توان کسری و اعشاری برسند.

- *<= < > = >*

این اپراتورها فقط برای مقایسه بین متغیرهایی که متناظر با یک عدد هستند استفاده می شوند و در صورت برقراری شرط، *true* برمی گردانند. *true* ای که برمی گردانند فقط به عنوان عبارات شرطی در *while…do* یا *repeat…until* یا *if…then…else* استفاده می شود.

- *~*

اپراتور ~ *(tild)* یک *free variable* را نشان می دهد.

- *and & or*

and : اگر هر دو *argument* درست باشد، *true* برمی گرداند.

or : اگر حداقل یکی از *argument* ها درست باشد، *true* برمی گرداند.

- ✓ فقط در عبارتهای شرطی استفاده می شوند.
- ✓ از سمت چپ اعمال می شوند: *x and y and z = (x and y) and z*

- *Where*

مثال:

*1: x**2+17*x*y+4*y**2 where x=1, y=2;*

51

*2: for i:=1:5 collect x**i*q where q=for j:= product j;*

$\{x,2x^2,6x^3,24x^4,120x^5\}$

*3: x**2+y+z where z=y**3, y=3;*

X^2+y^3+3

✓ مقداردهی یک متغیر به وسیله ی اپراتور *where* تأثیری بر مقدار این متغیر خارج از عبارت موردنظر ندارد.

- *list*

اپراتور *list* از آرگومانهای خود یک لیست می سازد:

list (item, item,…);

*1: liss := list (c,b,c, {xx,yy} ,3x**2+7x+3, df (sin(2*x),x));*

liss:= {c,b,c,{xx,yy},3x^2+7x+3,2cos(2x)}

برای کار کردن با یک *list* اپراتورهای مختلفی وجود دارد:

1) دستورات *first (list) , second (list) , third (list)* به ترتیب المان اول، دوم و سوم *list* را برمی‌گردانند.

دستور کلی تر برای دسترسی به المان های *list* , *part* است:

*1: alist := list (c,b,c, {xx, yy} , x**z)$*

2: part (a list, 4) ;

{xx, yy}

3: part (a list, 4, 1);

xx

2) item ← cons (item, list) را به اول *list* اضافه می کند و معادل عملگر . *(dot)* است.

1: a list := {b,c}$

2: a. a list;

{a,b,c}

3: liss := cons(a,{b}); % also liss := a cons{b};

new liss := {{a,x}, {b,x}}

5: for each y in new liss sum (first(y))(second(y));*

x(a+b+c)

3) length (list) ← تعداد المان های یک *list* را می دهد.

4) append (list, list) ← دو *list* را یکی می کند (با همان ترتیب آرگومانها)

5) rest (list) ← لیستی را برمی گرداند که المان اول *list* از آن حذف شده است.

6) reverse(list) ← برعکس *list* را برمی گرداند. *Reverse* و *cons* می توانند برای اضافه کردن یک المان به انتهای یک لیست استفاده شوند:

1: reverse (q cons reverse (a,b,c));

{a,b,c,q}

7) sort (list , comp) ← اعضای *list* را بر حسب قاعده ای که با *comp* تعیین می شود *sort* می‌کند:

1: for i:=1:10 collect random(50)$

2: sort (ws, > =);

{41,38,33,30,28,25,20,17,8,5}

- *ws*

ws آخرین نتیجه را برمی گرداند و *ws(number)* نتیجه ی خطی که شماره ی آن برابر با *number* است را برمی گرداند.

1: df (siny,y);

cos(y)

2: ws 2;

$Cos(y)^2$

3: df (ws 1, y);

-sin(y)

۳.۱.عملگرهای ریاضی (*Arithmatic operators*):

- *ln , log*

هر دو اپراتور لگاریتم طبیعی را حساب می کنند.

✓ برای محاسبه باید سویچ *rounded* روشن باشد. محدودیت های *ln* در مقابل *log* در مثال‌های زیر واضح است:

1: ln(4);

ln(4)

2: ln(e)

ln(e)

3: on rounded;

4: ln(e);

1

1: log(4);

log(4)

2: log(e);

1

3: on rounded;

4: log(e);

1

5: ln(4);

1.38629436112

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

$$\frac{\partial \ln(x)}{\partial x}$$

5: log(4);

1.38629436112

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

$$\frac{1}{x}$$

✓ استفاده از *log* توصیه می شود.

✓ *log(expression, integer)* لگاریتم *expr.* را در مبنای *int.* حساب می کند و محدودیت‌های آن مثل *ln* است.

- *Sqrt*

1: sqrt (16 a^3);*

$4a\sqrt{a}$

2: sqrt (17);

$\sqrt{17}$

3: on rounded;

4: sqrt (17);

4: 1231056562

*5: sqrt (a*b*c c^5*d d^3*27);*

5: 1961524271 $(d^3c^5ba)^{0.5}$

6: off rounded;

7: ws;

$abs(d)c^2\ 3\sqrt{abcd}\sqrt{3}$

✓ در شرایطی که سویچ *precise*، روشن باشد *sqrt(d^2)* را به صورت *abs(d)* نمایش می‌دهد و اگر *off* باشد، فقط خود *d* را چاپ می کند.

اپراتورهای ریاضی دیگری هم هستند که چون ساده اند یا کاربرد زیادی ندارند آنها را به اختصار شرح می دهیم:

1) abs (expression) ← اندازه ی *expression* را برمی گرداند.

2) choose (m,n) ← انتخاب *m* از *n* ← $\binom{n}{m}$

3) *factorial number* ← فاکتوریل *number* را حساب می کند.

4) *gcd(a,b)* ← ب.م.م دو عدد *a* و *b* را حساب می کند.

5) *max (expression, expression,…)* ← ماکسیمم چند *expression* را می دهد.

6) *next prime (expression)* ← اولین عدد اول بعد از *expression* را می دهد.

7) *norm (expression)* ← با روشن بودن سوئیچ *rounded* اندازه ی *exp.* را حساب می کند. اگر سوئیچ *complex* هم روشن باشد، اندازه ی عدد مختلط *(a+bi)* را هم حساب می کند.

8) *perm (m,n)* ← ترکیب *m* از *n* را حساب می کند.

9) *remainder (expression1, expression2)* ← باقیمانده ی تقسیم *expr.1* بر *expr.2* را حساب می کند. *expression* ها می توانند عدد یا چند جمله ای باشند.

10) *round (number)* ← *number* را به نزدیک ترین *integer* گرد می کند.

11) *sign (expression)* ← ۱ یا ۰ یا -۱ برمی گرداند و علامت *expr.* را تعیین می کند.

***Boolean Operators.3.2*:**

1) = ← اگر دو طرف برابر باشند *true* برمی گرداند.

2) *even p (integer)* ← اگر *integer* مثبت و زوج باشد *true* برمی گرداند.

3) *<expression> member <list>* ← اگر *<expr.>* عنصر *<list>* باشد *true* است.

4) *<expression> neq <expression>* ← اگر دو *<expr.>* برابر نباشند، *true* برمی گرداند.

5) *not (logical expression)* ← اگر *logical expr.* *(false) nil* باشد *true* برمی گرداند.

6) *number p (expression)* اگر *expr.* عدد باشد *true* برمی گرداند.

7) *Primep (expression)* ← اگر *expr.* عدد اول باشد، *true* برمی گرداند.

8) *free of (expression, kernel)* ← اگر *argument* اول شامل *argument* دوم نباشد، *true* برمی گرداند.

✓ برای این اپراتورها *0* یا *nil*، *false* محسوب می شود و هر چیزی غیر از اینها *true* به حساب می آید.

۳.۳.عملگرهای جبری(*Algebraic Operators*):

- *df*

df (expression, var1, number, var2, number2,…)

df از *expression* به ترتیب نسبت به *var1* به تعداد *number1* و نسبت به *var2* به تعداد *number2* و …

مشتق گیری جزئی انجام می دهد:

```
1: df (x**4*y+sin(y),y,x,3);
```

$$24x$$

```
2: for all x let df (tan(x),x)=sec(x)**2;
3: df (tan(3*x),x);
```

$$3\ \sec(3x)^2$$

✓ اگر بعد از نام متغیر، مرتبه مشتق گیری را مشخص نکنید، مرتبه یک در نظر گرفته می‌شود (مثل مثال اول)

✓ می توانید مثل مثال دوم با استفاده از دستور *let* قواعد مشتق گیری مورد نظر خودتان را تعریف کنید. در این صورت این تعریف بر قاعده ی مشتق گیری خود *REDUCE* اولویت دارد و با پاک کردن قوانین هم (*clear* *rules*) به حالت قبل برنمی گردد. فقط با پایان برنامه این قاعده پاک می شود.

✓ در تعریف *depend* می توانید ببینید که چطور مشتق های ضمنی را می توان وارد کرد.

- *int*

اپراتور انتگرال گیری:

```
1: int (sin(x)*exp(2*x),x);
```

$$\frac{e^{2x}(-\cos(x)+2\sin(x))}{5}$$

```
2: int (1. x^ 2-2),x);
```

$$\frac{\sqrt{2}\left(\log\left(-\sqrt{2}+x\right)-\log\left(\sqrt{2}+x\right)\right)}{4}$$

```
3: int (1/sqrt( x^ 2-x),x);
```

$$\int\left(\frac{\sqrt{x}\sqrt{x-1}}{x^2-x},x\right)$$

✓ انتگرال هایی که شامل عبارت های جبری مثل ریشه ی دوم هستند نمی توانند محاسبه شوند. مگر اینکه *package* به نام *algint* , *load* شده باشد که خود به خود سویچ *algint* را هم روشن می کند:

4: load-package "algint";

5: int (1/sqrt(x^ 2-x), x);

$$2\log\left(\sqrt{x-1}+\sqrt{x}\right)$$

✓ مثل اپراتور مشتق، اینجا هم می توانید با استفاده از *let* قواعد خود را برای انتگرال گیری تعریف کنید.

✓ برای انتگرال معین، عملگر جداگانه ای وجود ندارد. باید (با روشن بودن سوئیچ *rounded*) مقدار انتگرال را برای دو سر بازه حساب کرده و از هم کم کنید.

✓ اگر سوئیچ *failhard* روشن باشد، *REDUCE* در صورتی که نتواند انتگرال را حساب کند به جای چاپ به شکل *int(…)* ، یک پیغام *error* چاپ می کند.

- *limit*

با استفاده از امکانات *package* ای به نام *limits* می توان از اپراتور *limit(expr., var, limpoint)* حد *expr.* را نسبت به *var* در نقطه ی *limpoint* محاسبه کرد.

*1: limit (x*cot(x),x,o);*

0

2: limit $\left((2x+5)/(3x-2),x,\inf inity\right)$;

$$\frac{2}{3}$$

✓ حد راست و چپ را به طور جداگانه می توان با استفاده از *limit!-* و *limit!+* حساب کرد.

- *Solve*

این عملگر یک معادله ی جبری یا دستگاهی از معادلات را حل می کند.

Solve (<equation>,<var>)

یا:

Solve({<equation1>,<equation2>,…},{<var1>,<var2>,…})

✓ لیست دوم نشان می دهد که معادله بر حسب کدام متغیرها باید حل شود و اگر تعداد معادلات و متغیر ها برابر باشد می توان آن را حذف کرد.

1: sss := solve(x^2+7);

$$x$$

$$\left\{x=\sqrt{7}i, x=-\sqrt{7}i\right\}$$

2: rhs first sss;

$$-\sqrt{7}i$$

*3: solve(sin(x^2*y),y);*

$$\left\{y=\frac{\pi arb\,\mathrm{int}(1)+1}{x^2}, y=\frac{2arb\,\mathrm{int}(1)\pi}{x^2}\right\}$$

4: off allbranch;

*5: solve(sin(x^2*y),y);*

{y=0}

6: solve({3x+5y=-4,2x+y=-10},{x.y});

$$\left\{\left\{x=-\frac{46}{7}, y=\frac{22}{7}\right\}\right\}$$

*7: solve({x+a*y+z,2x+5},{x,y});*

$$\left\{\left\{x=-\frac{5}{2}, y=\frac{-2z+5}{2a}\right\}\right\}$$

8: ab := (x+2)^2(x^6+17x+1)$*

9: solve(ab,x);

$$www := \left\{x=-2, x=root\_of\left(x^6+17x+1\right)\right\}$$

10: root_multiplicities;

{2,1}

✓ اگر عملگر *solve* نتواند یک معادله را حل کند, بخش حل نشده را به عنوان آرگومان *root_of* برمی گرداند.

✓ درجه ی ریشه های معادله در متغیر *root_multiplicities* ذخیره می شود و با صدا کردن این متغیر در لیستی نمایش داده می شود.

✓ اگر یک دستگاه معادلات خطی داشته باشیم با روشن کردن سوئیچ *cramer* سرعت حل معادلات افزایش می یابد.

✓ اگر معادلات یک دستگاه متناقض باشند *solve* یک لیست خالی برمی گرداند و اگر این معادلات پارامتری باشند, متغیر *requirements* شرایطی را که تحت آن معادلات سازگار خواهندشد را نشان می دهد. شرط سازگاری باید مساوی صفر قرار بگیرد:

```
1: solve({x-a,x-y,y-1},{x,y});
        {}
2: requirements;
        {a-1}
```

✓ برای یک دستگاه معادلات خطی و پارامتری متغیر *assumptions* لیستی از روابط برای این پارامترها را برمی گرداند و جواب معادله تا وقتی معتبر است که هیچ کدام از این روابط مساوی صفر نشوند:

```
1: solve({a*x-b*y+x,y-c},{x,y});
```

$$\left\{\left\{x=\frac{bc}{a+1}, y=c\right\}\right\}$$

```
2: assumptions;
        {a+1}
```

✓ وقتی سوئیچ *allbranch* روشن است (*defult*) تمام ریشه های معادله نشان داده می‌شوند. ولی اگر خاموش باشد, فقط ریشه های اصلی نشان داده می شوند. مثل خط ۴و۵ مثال اول.

- *Ode solve*

ode solve package برای حل *(ordinary differential equation) ode* است و محدودیت هایی دارد: -*1* فقط یک معادله را می تواند حل کند. -*2* فقط قادر به حل معادله ساده ی مرتبه اول، معادله ی خطی با ضرایب ثابت و معادلات اویلر است.

ode solve (expr., var1, var2)

expr. =0 معادله دیفرانسیل مورد نظر است. *var1* متغیر وابسته و *var2* متغیر مستقل است.

✓ در *expr.*، مشتق را با عملگر *df* نمایش می دهیم.

```
1: depend y,x ;
2: ode: = df (y,x) +y*sinx/cosx-1/cosx $
3: ode solve (ode, y, x);
        {y=arbconst(1)*cos(x)+sin(x)}
```

✓ اپراتور *arbconst* مخفف *arbitrary constant* است برای نمایش ثابت دلخواه در حل معادله دیفرانسیل

- *Show rules*

تمام قوانین مربوط به آرگومان خود را نمایش می دهد:

1: show rules log;

$$\left\{\log(l)\Rightarrow 0\ ,\ \log(e)\Rightarrow 1\ ,\ \log\left(e^{\forall(x)}\right)\Rightarrow x,\frac{\partial\log(\forall(x))}{\partial\forall(x)}\Rightarrow\frac{1}{x},\ \frac{\partial\log\left(\frac{\forall(x)}{\forall(y)}\right)}{\partial\forall(z)}\Rightarrow\frac{\partial\log(x)}{\partial z}-\frac{\partial\log(y)}{\partial z}\right\}$$

✓ این قواعد مثل اعضای یک *list* قابل دسترس هستند:

2: rhs second ws;

1

برخی اپراتورهای جبری دیگر:

1) Conj (expression) ← مزدوج مختلط *expression* را برمی گرداند.

2) Hypot (n,m) ← اگر *m, n* عدد باشند $\sqrt{m^2+n^2}$ را برمی گرداند.

3) Import (expression) ← بخش موهومی *expression* را برمی گرداند.

4) Repart (expression) ← بخش حقیقی *expression* را برمی گرداند.

5) num (expression) ← بعد از ساده کردن *expression* صورت آن را برمی گرداند.

6) prod (expression, k, lolim, uplim) ← معادل $\prod_{k=lo\lim}^{up\lim}(\exp ression)$ است.

7) sum(expression, k, lolim, uplim) ← معادل $\prod_{k=lo\lim}^{up\lim}(\exp ression)$ است.

۴.توابع ریاضی

- *exp*

*exp (expression) or e^ or e***

1: exp (sin(x)) ;

$e^{sin(x)}$

2: exp (11) ;

e^{11}

3: on nonded

4: exp (sin(pi/3));

2.37744267524

✓ برای گرفتن جواب صریح عددی، سویچ *rounded* روشن باشد.

- *erf*

تابع خطا:

*erf(x)= (2/sqrt(pi))*int(e^ (- x^ 2),x)*

خواص محدودی از این تابع برای *REDUCE* شناخته شده است از جمله فـرد بـودن و مشـتق آن و از روی اینهـا بعضی انتگرالها قابل محاسبه است.

1:erf(0) ;

0

2: erf(-a) ;

-erf(a)

3: df (erf(x^ 2),x) ;

$$\frac{4\sqrt{\pi}x}{e^{x^4}\pi}$$

4: int (erf(x),x) ;

$$\frac{e^{x^2}erf(x)\pi x+\sqrt{\pi}}{e^{x^2}\pi}$$

- توابع مثلثاتی

توابع مثلثاتی با اپراتور هم نام خود قابل دسترسی هستند:

cos(expr), cosh(expr), cot(expr), coth(expr), csc(expr), csch(expr), ses(expr), sech(expr), sin(expr), sinh(expr), tan(expr), tanh(expr)

معکوس این توابع *(sin^{-1}, cos^{-1}, …)* با اضافه کردن *"a"* به اول نام اپراتورها به کار می روند:

acos(expr), acosh(expr), acot(expr), acoth(expr), acsc(expr), acsch(expr), asec(expr), asech(expr), asin(expr), asinh(expr), atan(expr), atanh(expr)

برای گرفتن جواب دقیق و صریح عددی، سوئیچ *rounded* باید روشن باشد.

- توابع خاص ریاضی در *Special Function Package* با دستور *specfn* *load-package* قابل دسترسی هستند.

اطلاعات موجود در این *package*: برخی ثابت های معروف، تابع بسل، چند جمله ای های خاص، تابع گاما، تابع زتا *(Zeta)*، توابع و انتگرال های بیضوی، ضرایب *Clebsch-Gordan* و … برای هر کدام از اینها چند اپراتور وجود دارد که معرفی همه ی آنها در این راهنمای خلاصه نمی گنجد. با استفاده از این *package* بعضی محاسبات هم راحتتر می شود. مثل محاسبه ی سریهایی که جواب آنها شامل توابع خاصی است.

5.Declaration

- *array*

تعریف آرایه به این شکل است:

array identifier (dimension, dimension, …)

- ✓ هر جزء آرایه بلافاصله بعد از تعریف آن، با صفر مقداردهی می شود.

```
1: array a(2,5), b(3,3,3);
2: a(3,4);
        0
3: length a;
        {3,6}
```

- ✓ تعریف آرایه همیشه *global* است. حتی اگر داخل *procedure* یا *block* تعریف شود.
- ✓ اجزای آرایه به صورت *a(i,j)* قابل دسترسی هستند.
- ✓ اپراتور := یا دستور *let* فقط برای تک تک اجزای آرایه کاربرد دارد، نه کل آرایه.
- ✓ اندیس آرایه یا شماره ی خانه های آن از صفر شروع می شود. بنابراین مثلا خانه های *a(5)* از صفر تا *5* شماره می خورند. پس طول آرایه ی *a* *(length a)*, *6* است.

- *Precision*

دقت اعشار را در حالتی که سوئیچ *rounded* روشن است تعیین می کند (پیش فرض *12* است)

```
1: on rounded;
2: 7/9;
        0.777777777778
3: precision 20;
        20
4: 7/9;
        0.77777777777777777778
```

- *depend*

وقتی در تعریف از *depend* استفاده می کنیم، یعنی *argument* اول بستگی به بقیه ی *argument* ها دارد. این بستگی با استفاده از *nodepend* از بین می رود.

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

$$2\frac{\partial y}{\partial x}y$$

```
1: depend z, cos(x), y ;
2: df (sin(z) , cos(x) ;
```

$$\cos(z)\frac{\partial z}{\partial \cos(x)}$$

3: nodepend y,x ;

*4: df (y**z, x) ;*

0

3: depend y, x ;

4: df (z^z , x);

$$2\frac{\partial y}{\partial x}z$$

✓ اپراتورهای خطی، در اعمال خطی بودن وابستگی متغیرها را در نظر می گیرند (به تعریف خطی اپراتورها مراجعه کنید)

✓ با استفاده از اپراتور *free of* می توانید وابستگی دو متغیر را چک کنید.

✓ بیش ترین کاربرد *depend* در حل معادله دیفرانسیل با استفاده از عملگر *ode solve* است.

- *Operator*

با تعریف کردن یک اپراتور می توانید عملگر دلخواه خود را بسازید.

1: operator dis ;

2: let dis (~x, ~y) = sqrt (x^2 + y^2) ;

3: dis (1,2) ;

$\sqrt{5}$

4: dis (a,10);

$\sqrt{a^2+100}$

1: operator fac;

2: let fac(~n) = if n=0 then 1

2: else if not (fixp n and n>0)

2: then rederr "choose non-negative integer"

3: fac (5);

120

✓ مواظب باشید یکی از اپراتورهای خود سیستم را دوباره تعریف نکنید.

✓ اپراتورها می توانند داخل *procedure* تعریف شوند و در این صورت *global* هستند.

✓ پارامتر صوری (صدا زننده ی *procedure*) می تواند یک اپراتور باشد.

اپراتورها می توانند با خواص مختلفی تعریف شوند:

1) *infix* ← برای تعریف عملگر میانوندی.

1: infix aa;

*2: for all x, y let aa(x,y). cos(x)*cos(y)-sin(x)*sin(y);*

3: x aa y;

cos(x)cos(y)-sin(x)sin(y)

4: pi/3 aa pi/2 ;

$$-\frac{\sqrt{3}}{2}$$

- ✓ دقت کنید که در عبارت *let*، اپراتور پیشوندی تعریف شده است.
- ✓ بعد از تعریف اپراتور می تواند هم به صورت پیشوندی و هم میانوندی استفاده شود.

even (2: اپراتور نسبت به آرگومان اولش فرد تعریف می کند:

1: even f;

2: f(-a, -b);

f(a, -b)

linear (3 ← یک اپراتور را بر حسب متغیر اول آن خطی تعریف می کند.

- ✓ اول اپراتور باید تعریف شده باشد و بعد خطی تعریف شود.

1: operator f;

2: linear f ;

3: f(o,x) ;

0

4: f(-y,x) ;

-f(l,x)y

5: f(y+z,x) ;

f(1,x)(y+z)

6: depend z,x ;

*7: f(y*z, x) ;*

f(z, x)y

8: depend y,x ;

9: $f(\frac{y}{z}, x)$;

$$f(\frac{y}{z}, x)$$

10: nondepend z,x ;

11: $f(\frac{y}{z},x)$ *;*

$$\frac{f(y,x)}{z}$$

دقت کنید که اضافه کردن وابستگی متغیرها چگونه در اعمال خطی بودن تأثیر می گذارند.

✓ اگر *argument* اول شامل *argument* دوم باشد هم تأثیر این وابستگی به همین شکل اعمال می شود.

noncom (4 ← اگر یک اپراتور را *nancom* تعریف کنیم به این معنی است که این اپراتور تحت عمل ضرب جا به جا پذیر نیست:

1: operator f, h ;
2: noncom f ;
3: $h(a)^* h(b)-h(b)^* h(a)$ *;*
0
4: $f(u)^* f(b)-f(b)^* f(a)$*;*
f(a)f(b)-f(b)f(a)

✓ در پیش فرض *REDUCE* یک اپراتور، غیرخطی، پیشوندی و جابه جا پذیر تعریف می‌شود.

- *scalar, real, integer*

تعریف متغیر به صورت *integer, real* یا *scalar* باید بلافاصله بعد از *begin* در یک *block* انجام شود. تعریف، به صورت *local* داخل *block* است و بلافاصله با صفر مقداردهی می شود. بعد از پایان *block* این متغیر حذف می شود. تغییرات مقدار آن داخل *block* متغیر هم نام آن خارج از *block* را تغییر نمی دهد. استفاده از *real* و *integer* به معنی اینکه *REDUCE* نوع متغیر را چک می کند نیست و فقط برای اطلاع خود کاربر است.

- *equation*

equation عبارتی است که در آن دو عبارت جبری به وسیله ی عملگر = به هم وصل شده اند و به شکل زیر است:

left-hand side=right-hand side

✓ عبارت سمت چپ معادله به وسیله ی عملگر *lhs(<equation>)* و عبارت سمت راست آن به وسیله ی عملگر *rhs(<equation>)* قابل دسترسی هستند.

✓ عبارت سمت راست همیشه محاسبه می شود. ولی برای محاسبه شدن عبارت سمت چپ, سوئیچ *evallhseqp* باید روشن باشد.

✓ وقتی یک معادله بخشی از یک عبارت منطقی است, مثل عبارت *if* یا *while* , دو طرف آن از هم کم شده و مقدار آن با صفر مقایسه می شود.

✓ معادله می تواند یک عضو *list* باشد. همچنین می توان یک متغیر را برابر با یک *equation* قرار داد.

✓ اگر سوئیچ *evallhseqp* روشن باشد, می توان دو معادله را با هم جمع یا از هم کم کرد یا یک معادله را به توان رساند.

1: on evallhseqp;

2: u := x+y=1$

3: v := 2x-y=0$

*4: 2*u-v;*

$$-3y=-2$$

5: ws/3;

$$\frac{2}{3}$$

6: lhs(u);

$$x+y$$

۶.سوئیچ ها (*switches*)

سویچ ها می توانند به وسیله ی دستورهای *on* و *off* روشن و خاموش شوند:

on switch-name ;

off switch-name ;

روشن یا خاموش بودن یک سوئیچ کارایی سیستم را تغییر می دهد.

تعداد سوئیچ های تعریف شده در *REDUCE* بسیار زیاد است. در اینجا فقط به سوئیچ هایی که در مثال ها به آنها اشاره کردیم یا مهم تر هستند می پردازیم:

- *algint*

کاربرد سوئیچ *algint* را در استفاده از اپراتور *int* ببینید.

- *combineexpt*

در صورت روشن بودن، توانایی *REDUCE* را در ساده کردن عبارتهای توانی افزایش می دهد:

```
1: 3^(1/2)*3^(1/3)*3^(1/6) ;
```

$$\sqrt{3}\sqrt[3]{3}\sqrt[6]{3}$$

```
2: on combineexpt ;
3: ws ;
```

3

- *complex*

با روشن بودن *complex* قوانین ریاضیات مختلط در ساده سازی، محاسبه ی توابع، فاکتورگیری و ... استفاده می شود بدون آن، *REDUCE* تنها می داند که i برابر با $\sqrt{-1}$ است:

```
1: (x^ + y^ 2)/(x+i*y) ;
```

$(iy+x)^{-1}(x^2+y^2)$

```
2: on complex ;
3: ws ;
```

x-iy

- *cramer*

با روشن بودن سوئیچ *cramer*، عملیات معکوس کردن ماتریس و حل معادلات خطی (*solve*) با استفاده از روش کرامرز انجام می شود که سریع تر است. البته اگر درایه های ماتریس عدد باشند عملیات کندتر میشود بنابراین قبل از روشن کردن این سوئیچ به نوع ماتریس توجه کنید.

- *failhard*

در صورت روشن بودن این سوئیچ، در صورتی که *REDUCE* نتواند یک انتگرال را به فرم بسته حساب کند، پیغام خطا می دهد.

- *gcd*

اگر این سوئیچ روشن باشد، فاکتورهای مشترک صورت و مخرج هنگام چاپ جواب ساده می شوند.

- *listargs*

در حالت کلی، اگر آرگومان یک اپراتور *list* باشد ولی این اپراتور برای *list* تعریف نشده باشد، جواب، لیستی است که در عضو آن تأثیر اپراتور روی عضو متناظر آن در آرگومان تابع است. روشن بود سوئیچ *listargs* از این عمل جلوگیری می کند:

```
1: log {a,b,c} ;
        {log(a), log(b), log(c)}
2: on listargs ;
3: log {a,b,c} ;
        log({a,b,c})
```

- *multiplicities*

در حل یک معادله (*solve*) اگر یک ریشه درجه ای بیشتر از یک داشته باشد، صریحاً در جواب این درجه نشان داده نمی شود. بلکه در متغیر *root-multiplicities* ذخیره می شود، ولی با روشن بودن سوئیچ *multiplicities*، درجه ی ریشه در خود جواب مشخص است:

```
1: solve( x^ 2=2x-1, x) ;
        x=1
2: root-multiplicities ;
        2
3: on multiplicities;
4: solve ( x^ 2=2x-1, x);
        x=1 , x=1
```

- *nero*

اگر سوئیچ *nero* روشن باشد، متغیری که مقدار صفر دارد چاپ نمی شود. این کار به خصوص وقتی با ماتریس های بزرگ سر و کار داریم مفید است که می توان فقط درایه های غیر صفر را در صفحه دید.

- *output*

اگر سوئیچ *output* خاموش باشد، هیچ نتیجه ای روی صفحه چاپ نمی شود حتی اگر انتهای خط دستور ; باشد. چون چاپ نتیجه زمان بر است، در صورتی که با محاسبات مفصل و نتایج پیچیده سروکار دارید، خاموش کردن این سوئیچ می تواند برنامه را سریع تر پیش ببرد. در صورت احتیاج به نتیجه ی یک خط خاص می توانید از *ws(number)* استفاده کنید.

- *Precise*

1: sqrt (x^ 2);

x

2:(x^ 2)^ (1/4);

$\sqrt{x}$

3: on precise;

4: sqrt (x^ 2)^ ;

abs(s)

5: (x^ 2)^ (1/4);

$\sqrt{abs(x)}$

- *Rounded, roundall, roundbf*

✓ با روشن بودن سوئیچ *rounded* ریاضیات اعشاری فعال می شود.

1: pi ;

π

2: 35/217 ;

$\frac{5}{31}$

3: on rounded ;

4: pi ;

3.141592359

5: 35/217 ;

0.161

- ✓ برای تغییر دقت اعشار از دستور *precision* استفاده کنید. به طور پیش فرض، دقت اعشار *12* رقم است.
- ✓ اگر بخواهید اعداد کسری به شکل اعشاری در نیایند، سوئیچ *round all* را خاموش کنید:

1: on rounded ;

2: sqrt (3);

1.73205080756

3: 1/2 ;

0.5

4: off round all ;

5: 1/2 ;

$$\frac{1}{2}$$

- ✓ اگر بخواهید *REDUCE* اعداد بسیار کوچک را صفر چاپ نکند باید سوئیچ *roundbf* را روشن کنید.

1: on rounded;

2: exp (-100000.1^ 2);

0

3: on round bf ;

4: exp (-100000. 1^ 2);

1.18441281937e-4342953505

- *Time*

اگر سوئیچ *time* روشن باشد، زمان سیستم که برای اجرا کردن هر گزاره ی *REDUCE* استفاده شده است بعد از چاپ جواب، نمایش داده می شود.

1: on time ;

time : 281 ms

2: df (sin (x^ 2+y), y);

cos(x^ +y)

time: 0 ms

*3: solve (x^ 2-6*y, x);*

$$\left\{x=\sqrt{y}\sqrt{6}, x=-\sqrt{y}\sqrt{6}\right\}$$

Time: 16 ms

✓ زمانی که بعد از روشن کردن *time* چاپ می شود، زمانی است که از ابتدای شروع برنامه محاسبه شده است.

✓ بعد از هر دستور، زمان محاسبه چاپ می شود که شامل زمان تایپ کردن دستور نیست.

✓ در جاهایی که ادعا کردیم روشن بودن یک سوئیچ می تواند سرعت اجرا و محاسبه را افزایش دهد (مثل سوئیچ *cramer* و *output*) این سرعت را می توانیم با روشن کردن *time* چک کنیم.

۷.ماتریس ها(***Matrix***)

- **تعریف ماتریس:**

یک *identifier* را می توانیم از نوع *matrix* تعریف کنیم.

```
1: matrix a, b(1,4);
2: b(1,1) ;
        0
3: a(1,1) ;
        ***** Matrix A not set
```

✓ تعیین مرتبه ی ماتریس هنگام تعریف آن اجباری نیست. ولی در صورت تعریف مرتبه همه‌ی درایه ها با صفر مقداردهی می شوند و در صورت عدم تعریف مرتبه، درایه ها قابل دسترسی نیستند.

```
4: a:= mat[(x0, y0), (x1, y1))
5: length a;
        {2,2}
6: b: = a^n 2 ;
```

$$b := \begin{pmatrix} x_o^2 + x_1 y_o & y_o(x_o + y_1) \\ x_1(x_o + y_1) & x_1 y_o + y_1^2 \end{pmatrix}$$

✓ در صورت نیاز، مرتبه ی ماتریس با محاسبات هماهنگ می شود (مثل ماتریس *b*)

✓ *let* را می توان به کل یک ماتریس اعمال کرد، ولی اگر *let* و *clear* به یک درایه ی ماتریس بخواهد اعمال شود، حتماً این درایه باید محتوی یک ثابت باشد.

✓ عملیات جبری را می توان برای ماتریس ها به کار برد در صورتی که مرتبه ها درست باشد (+ - * /) *1/A* یا */A* عکس ماتریس *A* را می دهد (در صورت معکوس پذیری). *A/B* هم به صورت $A^*B^{\wedge}-1$ تعریف مـی شـود. هم چنین ماتریس های مربعی را می توان به توان صحیح مثبت و منفی (در صورت معکوس پذیری) رساند.

- اپراتور *mat* : برای نمایش ماتریس

1: mat ((1,2), (3,4)) ;

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

2: matrix q;

3: a;= ws ;

$$q := \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

4: a: = mat ((sin(x), cos(x), 1);

a:= (sin(x) cos(x) 1)

✓ برای استفاده از *mat* احتیاجی نیست که ماتریس قبلاً تعریف شده باشد و مرتبه آن مشخص باشد.

✓ *Mat* در صورت نیاز مرتبه ی تعریف شده برای ماتریس را تغییر می دهد.

✓ اگر بخواهیم به درایه ها دسترسی داشته باشیم *(a(i,j))* حتماً ماتریس باید تعریف شده باشد.

- اپراتور *cofactor*

cofactor (matrix-expression, row, column)

1: cofactor (mat ((a,b,c), (d,e,f), (p,q,r)), 2, 2);

ar- cp

2: cofactor (mat((a,b,c), (d,e,f)), 2, 2);

***** non-square matrix*

- اپراتور *def* : برای محاسبه ی دترمینان ماتریس

1: m: = mat (a,b,c), (d,e,f), (p,q,r)) $

2: def m; ,% also det(m);

aer-afq-bdr+bfp+cdq-cep

3: def (5);

5

- اپراتور *mateigen*: برای به دست آوردن معادله ی ویژه مقدار و ویژه بردارهای یک ماتریس

mateigen (matrix-expression, var)

✓ *Var* نام متغییر معادله ی ویژه مقدار است.

1: aa: = mat (2,5), (1,0)) $

2: mateigen (aa, alpha);

$$\left\{\alpha^2-2\alpha-5,\ 1,\ \begin{pmatrix}\frac{5arbcomplex(1)}{\alpha-2}\\ arbcomplex(1)\end{pmatrix}\right\}$$

3: bb: = mat ((1,2,3), (4,5,6), (7,8,9))$

4: mateigen (bb, alpha);

$$\left\{\alpha,1,\begin{pmatrix}arbcomplex(2)\\ -2arbcomplex(2)\\ arbcomplex(2)\end{pmatrix}\right\},\left\{\alpha^2-15\alpha-18,\ 1,\ \begin{pmatrix}\frac{arbcomplex(3)(\alpha-1)}{3\alpha+5}\\ \frac{2arbcomplex(3)(\alpha+1)}{3\alpha+5}\\ arbcomplex(3)\end{pmatrix}\right\}$$

✓ در لیست هایی که برگردانده می شود، عضو اول معادله ی ویژه مقدار است که مساوی صفرقرار می گیـرد، عضو دوم درجه ی ویژه مقدار و عضو سوم ویژه بردار متناظر است.

✓ اپراتور *arbcomplex* در یک *expression* بیانگر یک بخش اسکالر دلخواه در آن *expression* است.

- اپراتور *nullspace*: برای ماتریس a لیستی از بردارهای مستقل خطی را حسـاب مـی کنـد کـه ترکیـب خطی آنها معادله ی $ax=0$ را ارضا می کند.

nullspace(matrix-expression)

1: nullspace(mat((1,2,3,4), (5,6,7,8)));

$$\left\{\begin{pmatrix}1\\0\\-3\\2\end{pmatrix},\begin{pmatrix}0\\1\\-2\\1\end{pmatrix}\right\}$$

- اپراتور *smithex*: فرم نرمال *smith* را برای ماتریس A حساب می کند و $\{S,P,P^{\wedge}-1\}$ را برمی گرداند که : $P^{*}S^{*}P^{\wedge}-1=A$

smithex (<matrix>,<variable>)

✓ *<matrix>* ماتریسی بر حسب *<variable>* است.

1: a: = mat ((x,x+1), (0,3 x^ 2))$*

2: smithex (a,x);

$$\left\{\begin{pmatrix}1 & 0\\ 0 & x^3\end{pmatrix},\begin{pmatrix}1 & 0\\ 3x^2 & 1\end{pmatrix},\begin{pmatrix}x & x+1\\ -3 & -3\end{pmatrix}\right\}$$

اپراتورهای دیگری از جمله *smithex-int(<matrix>)* ، *frobenius(<matrix>)*، *jordan(<matrix>)* و ... نیز کارکرد مشابهی دارند که به ترتیب فرم نرمال *smith* برای ماتریس با درایه های *integer*، فرم نرمال *frobenius* و فرم نرمال *Jordan* را برای ماتریس A حساب می کنند و به ترتیب $\{S,P,P^{\wedge}-1\}$، $\{F,P,P^{\wedge}-1\}$ و $\{J,P,P^{\wedge}-1\}$ را برمی گردانند که $P^{*}J^{*}P^{\wedge}-1=A,$. $P^{*}F^{*}P^{\wedge}-1=A,\ P^{*}S^{*}P^{\wedge}-1=A,$

- *add-columns (<matrix>, <c1>, <c2>, <expr>)* : به ستون *c2* از *<matrix>* *c1*<exper>* را اضافه می کند. *add-rows* هم کارکرد مشابهی برای سطرها دارد.

- *add-to-columns (<matrix>, <column-list>, <expr>)* : به ستون هایی که در لیست *<column-list>* مشخص شده اند، *<expr>* را اضافه می کند. *add-to-rows* هم کارکرد مشابهی برای سطره دارد. دو اپراتور *mult-columns* و *mult-rows* با *syntax* مشابه همین کار را با عملیات ضرب (به جای جمع) انجام می دهند.

- *augment-comlumns (<matrix>, <column-list>)*: ستون های مشخص شده در لسیت *<column-list>* از *<matrix>* را نگه می دارد و در ماتریس جدیدی کنار هم قرار می دهد. *Stack-rows* همین کار را برای سطرها انجام می دهد.
- *band-matrix (<expr-list>,<size>)* : *<expr-list>* سینی با تعداد اعضای فرد است و *band matrix* یک ماتریس مربعی با سایز *<size>* می سازد و این لیست را طوری در سطرهای آن قرار می دهد که عضو وسط همیشه روی قطر باشد:

1: band-matrixs ({x,y,z},4);

$$\begin{pmatrix} y & z & 0 & 0 \\ x & y & z & 0 \\ 0 & x & y & z \\ 0 & 0 & x & y \end{pmatrix}$$

- *Char-matrix(<matrix>,<lambda>)* : ماتریس مشخصه ی *<matrix>* را بر حسب متغیر *<lambda>* می سازد یعنی *c=<lambda>*Id-<matrix>* که *Id* ماتریس واحد است.

- *Char-poly (<matrix>, <lambda>)*: معادله ی مشخصه ی *<matrix>* را بر حسب متغیر *<lambda>* پیدا می کند یعنی دترمینان *c=<lambd>*Id-<matrix>* که *Id* ماتریس واحد است.

- *Cholesky (<matrix>)* : تجزیه ی *cholesky* را برای *<matrix>* نمایش می دهد. چیزی که برمی گرداند *{L,U}* است به طوری که *A=LU* و *U = L^T*. در این اپراتور درایه های ماتریس باید حقیقی باشند. برای ماتریس با درایه های مختلط دستور *lu-decom(<matrix>)* را به کار می‌بریم.

1: f: = mat((1,1,0), (1,3,1), (0,1,1)) $
2: on rounded;
3: cholesky (f);

$$\left\{\begin{pmatrix}1 & 0 & 0\\ 1 & 1.41421356237 & 0\\ 0 & 0.707106781187 & 0.707106781187\end{pmatrix},\begin{pmatrix}1 & 0 & 0\\ 1 & 1.41421356237 & 0.707106781187\\ 0 & 0 & 0.707106781187\end{pmatrix}\right\}$$

- *Coeff-matrix ({<lineq-list>})* : <*lineq-list*> لیستی از معادلات است. این اپراتور *{C,X,B}* را برمی گرداند به طوری که $CX=B$

*1: coeff-matrix ({x+y+4*z=10, y+x-z=20, x+y+4});*

$$\left\{\begin{pmatrix}4 & 1 & 1\\ -1 & 1 & 1\\ 0 & 1 & 1\end{pmatrix},\begin{pmatrix}z\\ y\\ x\end{pmatrix},\begin{pmatrix}10\\ 20\\ -4\end{pmatrix}\right\}$$

- *Row-dim(<matrix>), column-dim (<matrix>)* ، ابعاد ستون و سطر <*matrix*> را حساب می کند.

- *diagonal (<mat-list>)* : ماتریسی می سازد که اعضای <*mat-list*> روی قطر آن قرار دارند. اعضای <*mat-list*> ممکن است اسکالر یا ماتریس باشند.

- *extend (<matrix>, <r>, <c>, <expr>)* : ماتریس را به اندازه ی <*r*> سطر و <*c*> ستون گسترش می دهد و درایه های جدید را برابر با <*expr*> قرار می دهد.

- *get-columns (<matrix>, <column-list>)*: ستون هایی که شماره ی آنها در لیست <*column-list*> مشخص شده را به صورت لیستی از ماتریس های ستونی برمی گرداند. *Ger-rows(matrix>,<row-list>)* همین کار را برای سطرها می کند.

- *tp(<matrix>)*: ترانهاده ی <*matrix*> را حساب می کند.

- *hermitian-tp(<matrix>)*: برای ماتریس A، A^+ را حساب می کند.
- *hesian(<expr>, <var-list>)*: ماتریس *hessian* را برای *<expr>* محاسبه می کند و آن ماتریس $n \times n$ است (*n*: تعداد اعضای *<var-list>*) که درایه ی *(i,j)* آن برابر است با *df(<expr>,<var-list>(i), <var-list> (i)*.

- *hilbert(<size>,<expr>)*: ماتریس مربعی *Hilbert* را برای *<expr>* محاسبه می کند و آن ماتریس متقارنی است که درایه ی *(i,j)* آن برابر است با $\frac{1}{i+j-\langle \exp r \rangle}$

- *jacobian (<expr-list>,<var-list>)*: ماتریس ژاکوبی را برای *<expr-list>* (بر حسب متغیرهای *<var-list>*) محاسبه می کند که یک ماتریس $n \times m$ است: n تعداد متغیرها و m تعداد *expression* ها. درایه ی *(i,j)* آن برابر است با: *df(<expr-list>(i),<var-list>(j))*

1: jacobian$\left(\{x\hat{}4, x^*y\hat{}2, x^*y^*z\hat{}3\}, \{w,x,y,z\}\right)$

$$\begin{pmatrix} 0 & 4x^3 & 0 & 0 \\ 0 & y^2 & 2xy & 0 \\ 0 & yz^3 & xz^3 & 3xyz^2 \end{pmatrix}$$

- *Jordan-block (<expr>,(size>)* : ماتریس مربعی *Jordan block* را با ابعاد *<size>* می‌سازد درایه های این ماتریس به این شکل هستند:

J(i,j) = <expr> for i=1,…,n , J(I,i+1)=1 for i=1,…,n-1

و بقیه ی درایه ها صفر هستند.

- *make-identity (<size>)*: ماتریس واحد را با ابعاد *<size>* × *<size>* می سازد.
- *matrix-augment(<matrix-list>)* : ماتریس های *<matrix-list>* را افقی کنار هم قرار می دهد و یک ماتریس می سازد. *matrix-stack* همین کار را به صورت عمودی انجام می دهد.

Matrix(<input>): اگر *<input>* ماتریس باشد *true* بر می گرداند و در غیر این صورت *nil*.

- *Minor (<matrix>, <r>, <c>)* : ماتریس جدیدی می سازد سطر *<r>* و ستون *<c>* از *<matrix>* را در آن حذف کرده است.

- *random-matrix(<r>,<c>,<limit>)* : یک ماتریس *<r>×<c>* می سازد که درایه‌های آن اعداد رندوم بین *<limit>*- و *<limit>* هستند. سویچ های زیر نوع درایه ها را تعیین می کنند:

1) *imaginary* : اگر روشن باشد درایه ها به صورت *x+iy* خواهند بود که *–limit<x,y<limit*

2) *not-negative*: اگر روشن باشد درایه ها (یا *x,y* در حالت *imaginary*) بین *0* و *limit* خواهندبود.

3) *only-integer*: اگر روشن باشد درایه ها (یا *x,y* در حالت *imaginary*) *integer* خواهند بود.

4) *symmetric* : اگر روشن باشد، ماتریس متقارن خواهد بود.

5) *upper-matrix*: اگر روشن باشد، ماتریس بالا مثلثی خواهد بود.

6) *lower-matrix* : اگر روشن باشد، ماتریس پایین مثلثی خواهد بود.

- *remove-columns (<matrix>,<column-list>)* : ماتریس جدیدی می سازد که ستون‌های مشخص شده در *<column-list>* از *<matrix>* را در آن حذف کرده است. *remove-rows* عملیات مشابه را برای سطرها انجام می دهد.

- *Squarep (<matrix>)* : اگر *<matrix>* مربعی باشد *true* برمی گرداند و در غیر این صورت *nil*

- *sub-matrix(<matrix>, <row-list>, <column-list>)* : ستون های *<c1>, <c2>* از *<matrix>* را با هم جا به جا می کند. *swap-rows* همین کار را برای سطرها انجام می دهد.

Swap-entries (<matrix>, <r1,c1> , <r2,c2>) : درایه های *(<r1>, <c1>) , (<r2> , <c2>)* را با هم جا به جا می کند.

- *Symmetricp (<matrix>)* : اگر *<matrix>* متقارن باشد *true* برمی گرداند و در غیر ایـن صـورت *nil*

- *toeplitz (<expr-list>)* : از لیست *<expr-list>* ماتریس *toeplitz* را می سازد. به این شکل کـه اولین *expression* روی قطر قرار می گیرد و *i*- امین *expression* روی *(i-1)* ردیف بالاتر و پایین تر از قطر:

1: toeplitz ({w,x,y,z});

$$\begin{pmatrix} w & x & y & z \\ x & w & x & y \\ y & x & w & x \\ z & y & x & w \end{pmatrix}$$

- *Vandermonde({<expr-list>})* : از روی *<expr-list>* ماتریس *vandermonde* را مـی‌سـازد به این شکل که درایه ی *(i,j)* را برابر $(<\exp r-list>(i))\hat{}(j-1)$ قرار می دهد. ابعـاد مـاتریس $n\times n$ اسـت که n تعداد *expression* است:

*1: vander monde ({x,z*y, 3*z});*

$$\begin{pmatrix} 1 & x & x^2 \\ 1 & 2y & 4y^2 \\ 1 & 3y & 9z^2 \end{pmatrix}$$

- *Trace (<square-matrix>)* : رد یک ماتریس مربعی را محاسبه می کند.

- *Pseudo-inverse (<matrix>)* : *pseudo inverse* (شـبه وارون) *<matrix>* را محاسـبه مـی کند که: *<matrix>*pseudo-inverse (<matrix>)=Id* و لزومـی نـدارد کـه *<matrix>* یـک مـاتریس مربعی باشد.

یک نکته:

به نظر می رسد در یک *group* یا *block* یا یک عبارت تنها (مثلاً بدنه ی *procedure*) تعداد محدودی از حروف جای می گیرد. بنابراین در شرایطی که با عبارت های طولانی کار می‌کنید, اگر *REDUCE* عملیات را انجام نداد, این مسئله یکی از اولین گزینه های محتمل است که در این صورت باید تعداد حروف را کم کنید.

از آنجا که در توضیحات نرم افزار اشاره ای به این مسئله نشده است, این احتمال وجود دارد که این محدودیت مربوط به یک *version* خاص باشد. در هر حال بهتر است مد نظر گرفته شود!