Основные понятия квантовой механики (продолжение)

Сложение и умножение операторов

Механика, которой подчиняются атомные явления, - так называемая квантовая или волновая механика - основана на представлениях о движении, принципиально отличных от представлений классической механики. Рассмотрим некоторую физическую величину A , характеризующую состояние квантовой системы. Значения, которые может принимать данная физическая величина, называют ее собственными значениями, а об их совокупности говорят как о спектре собственных значений данной величины. Каждой физической величине в квантовой механике приводится в соответствие линейный оператор. Пусть A и B - одновременно измеримые величины. Наряду с их суммой, можно ввести понятие и об их произведении, как о величине, собственные значения которой равны произведениям собственных значений величин A и B . Таким образом, если две величины могут иметь одновременно определенные значения, то их операторы коммутируют друг с другом $AB-BA=0$. Если две величины не могут иметь одновременно определенных значений, то понятие об их произведении не может быть определено указанным выше способом. Отметим, что из произведений двух некоммутативных эрмитовых операторов можно составить антиэрмитов оператор $[A,B]=AB-BA$ - так называемый коммутатор, и эрмитов оператор $\{A,B\}=AB-BA$ - так называемый антикоммутатор. Таким образом, не коммутативное умножение и вычисления, использующие коммутаторы и антикоммутаторы, являются одними из основных элементов квантовой механики. В силу этого, в данной главе мы рассмотрим возможность подобных вычислений в системах символьных вычислений, например, в системе Maple. Отметим сразу, что имеющаяся в ранних версиях системы Maple, встроенная процедура коммутирования в Maple6 отсутствует.

- **Коммутаторы в Maple (I)**

В Maple реализовано несколько возможностей для работы с операторами, групповыми элементами и другими сложными математическими объектами. Например, при работе с матрицами мы используем некоммутативное умножение $A\,B \neq B\,A$. Однако, в квантовой механике не всегда удобно иметь дело с матричными представлениями операторов. В силу этого, в Maple есть возможность определить некоммутативное умножение для произвольных операторов, не отождествляя их с матрицами. Произведение операторов, как и матриц, имеет вид $A\ \&*\ B$. Изучим свойства этой операции подробнее.

- **Что такое "просто" для некоммутативного умножения &***

Повторим, что главное в системах символьных вычислений - это научиться получать необходимые нам простые результаты. В силу этого, самое главное это научиться упрощать полученные выражения, так как для компьютера любое представление результата является одинаково простым.

Например, мы имеем два умножения - скалярное умножение и некоммутативное умножение

> **restart:**
(A&*B/3)-(B&*-A/3);

$$\frac{1}{3}(A\ \&*\ B)-\frac{1}{3}(B\ \&*-A)$$

Видно, что первое слагаемое упростилось, а второе нет. В данной реализации системы Maple все числа имеют тип $constant$, хотя не воспринимает $-a$ как $(-1)\,a$. Это связано с стремлением к универсальности программы. Для упрощения в этом случае необходимо применять не команду $simplify$, а команду $expand$. Например, создадим список выражений

> **ListEx:={0&*A,1&*-A,(sqrt(2)*A)&*B};**

$$ListEx := \{0\ \&*\ A,\ 1\ \&*\text{-}\ A,\ \sqrt{2}\ A\ \&*\ B\}$$

Применяя обычную команду ко всем элементам множества, искомого упрощения мы не достигнем

> **map(simplify,ListEx);**

$$\{0\ \&*\ A,\ 1\ \&*\text{-}\ A,\ \sqrt{2}\ A\ \&*\ B\}$$

Специальная команда дает лучший результат (но не всегда)

> **map(expand,ListEx);**

$$\{0,\ 1\ \&*\text{-}\ A,\ \sqrt{2}\ (A\ \&*\ B)\}$$

Все выше сказанное справедливо и для свойства ассоциативности

> **(A&*B)&*C-A&*(B&*C);**

$$((A\ \&*\ B)\ \&*\ C) - (A\ \&*\ (B\ \&*\ C))$$

> **simplify(%);**

$$((A\ \&*\ B)\ \&*\ C) - (A\ \&*\ (B\ \&*\ C))$$

> **expand(%);**

$$0$$

и дистрибутивности

> **expand((2*A+B/3)&*C);**

$$2\ (A\ \&*\ C) + \frac{1}{3}\ (B\ \&*\ C)$$

Используя операцию некоммутативного умножения, мы можем определить **коммутатор** двух операторов $[A, B] = A\ B - B\ A$ в виде простой операции

> **restart:**
Cm:=(A,B)->A&*B-B&*A;

$$Cm := (A, B) \rightarrow\ `\&*`(A, B) - `\&*`(B, A)$$

Теперь можно задать значение этого коммутатора для любой пары операторов. Например, для операторов координаты x и момента p, зададим коммутатор, используя имя h для константы Планка hbar=h/(2 Pi)

> **def1:=Cm(x,p)=I*h;**

$$def1 := (x\ \&*\ p) - (p\ \&*\ x) = I\ h$$

> **xp:=solve(def1,x&*p);**

$$xp := (p \;\&*\; x) + I\,h$$

Одним из способов вычисления и упрощения выражений, содержащих коммутаторы, является подстановка. Например

> **subs(x&*p=xp,Cm(x,p));**

$$I\,h$$

Подстановка эффективна только при наличии двух операторных сомножителей в выражении. Иногда этого недостаточно

> **expand(subs(x&*p=xp,Cm(x,Cm(x,p))));**

$$I\,(x \;\&*\; h) - I\,(h \;\&*\; x)$$

С другой стороны и этот простой метод можно использовать. Рассмотрим в качестве примера квантовые цепочки Тоды, которые являются одним из модельных примеров в современном квантовом методе обратной задачи. Мы выбрали именно этот нестандартный пример для того, чтобы показать возможность использования систем символьных вычислений в современной квантовой физике.

- **Цепочки Тоды в квантовом методе обратной задачи**

Внутреннее представление матриц отличается от представления чисел (скаляров). В тоже время, символьное представления операторов и чисел одинаковы. В силу этого нам придется отделять символы операторов от символов чисел. Более подробно мы рассмотрим этот вопрос далее. Итак, мы займемся конкретной задачей, в которой можно получить необходимый ответ, используя подстановку для вычисления коммутаторов.

> **restart:**

Пусть константа Планка и два спектральных параметра u и v являются константами(скалярами)

> **constants:=constants,h,u,v;**

$$constants := false, \gamma, \infty, true, Catalan, FAIL, \pi, h, u, v$$

Определим ассоциативное умножение с фиксированными свойствами дистрибутивности, с заданными правилами умножения для скаляров, с единицей и нулем (более подробно это определение разобрано в следующем параграфе).

> **prop:=`&.`(-a::anything,-b::anything)=`&.`(a,b):**

Описав все необходимые нам свойства, зададим операцию умножения **&.**

> **define(`&.`, flat,multilinear, identity=1,zero=0,prop);**

Теперь введем квантовый одноузельный оператор для цепочки Тоды

> **with(linalg):**

Warning, the protected names norm and trace have been redefined and unprotected

> **Tu:=matrix(2,2,[u-p[1],exp(x[1]),-exp(-x[1]),0]);**

$$Tu := \begin{bmatrix} & & x_1 \\ u - p_1 & \mathbf{e} & \\ & (-x_1) & \\ -\mathbf{e} & & 0 \end{bmatrix}$$

Теперь мы хотим проверить фундаментальные коммутационные соотношения

$$R(u - v)\, T_1(u)\, T_2(v) = T_2(v)\, T_1(u)\, R(u - v)$$

$$T(v)$$

Для этого, во-первых, зададим матрицу

> **Tv:=evalm(map(d->subs({u=v},d),Tu));**

$$Tv := \begin{bmatrix} & & x_1 \\ v - p_1 & \mathbf{e} & \\ & (-x_1) & \\ -\mathbf{e} & & 0 \end{bmatrix}$$

Далее напишем процедуру тензорного произведения матриц

```
> TenMul:=proc(A,B) local i,j,n,nn,m,mm,T,S,C:
options operator, arrow:
m:=linalg[rowdim](A):n:=linalg[coldim](A):
mm:=linalg[rowdim](B):nn:=linalg[coldim](B):
C:=matrix(m*mm,n*nn):
T:=[]: T:=[seq(seq([i,j], j=1..m),i=1..mm)];
S:=[]: S:=[seq(seq([i,j], j=1..n),i=1..nn)];
for i from 1 to n^2 do for j from 1 to n^2 do
C[i,j]:=A[op(1,T[i]),op(1,S[j])]*B[op(2,T[i]),op(2,S[j])];
od;od; evalm(C);
end:
```

Теперь построим матрицы $T_1(u) = T(u)\, x\, I$ и $T_2(v) = I\, x\, T(v)$

```
> ed := eval(array(identity, 1..2,1..2)):
T1:=TenMul(Tu,ed);
```

$$Tl := \begin{bmatrix} & & & x_1 & \\ u - p_1 & 0 & \mathbf{e} & 0 & \\ & & & & x_1 \\ 0 & u - p_1 & 0 & \mathbf{e} & \\ & (-x_1) & & & \\ -\mathbf{e} & 0 & 0 & 0 & \\ & & (-x_1) & & \\ 0 & -\mathbf{e} & 0 & 0 & \end{bmatrix}$$

> **T2:=TenMul(ed,Tv);**

$$T2 := \begin{bmatrix} v - p_1 & \mathbf{e}^{x_1} & 0 & 0 \\ -\mathbf{e}^{(-x_1)} & 0 & 0 & 0 \\ 0 & 0 & v - p_1 & \mathbf{e}^{x_1} \\ 0 & 0 & -\mathbf{e}^{(-x_1)} & 0 \end{bmatrix}$$

Затем определим рациональную R-матрицу

```
> ed := eval(array(identity, 1..4,1..4)):
P:=matrix(4,4,[1,0,0,0,
0,0,1,0,
0,1,0,0,
0,0,0,1]):

R:=evalm(scalarmul(ed,-(u-v))+scalarmul(P,I*h) );
```

$$R := \begin{bmatrix} -u + v + I h & 0 & 0 & 0 \\ 0 & -u + v & I h & 0 \\ 0 & I h & -u + v & 0 \\ 0 & 0 & 0 & -u + v + I h \end{bmatrix}$$

Заметим, что матричные элементы матриц $T_{1,2}$ являются квантовыми операторами и, поэтому, нам придется переписать операцию матричного умножения, используя некоммутативное произведение элементов матриц.

```
> NcMul:=proc(A,B) local i,j,n,C:
options operator, arrow:
n:=linalg[rowdim](A):
C:=matrix(n,n):
for i from 1 to n do
for j from 1 to n do
C[i,j]:=sum(A[i,k]&.B[k,j],k=1..n);
od;od;
evalm(C);
end:
```

Теперь у нас все готово для проверки коммутационных соотношений.

```
> otv:=evalm( R&*NcMul(T1,T2)-NcMul(T2,T1)&*R):
```

Явно мы не выписываем ответ. Можете проверить, что в данном выражении нулей не так много и, поэтому, этот ответ выглядит недостаточно убедительным. В силу этого, приступим к упрощению.

Зададим коммутационные соотношения между оператором импульса и экспонентами от оператора координаты. Также мы вынуждены дополнительно определить коммутатор между функциями e^{x_1} и $e^{(-x_1)}$.

```
> p1:=p[1]&.exp(x[1])+I*h*exp(x[1]):
m1:=p[1]&.exp(-x[1])-I*h*exp(-x[1]):
x1:=exp(x[1])&.exp(-x[1]):
```

Осталось подставить эти коммутационные соотношения в ответ и упростить полученное выражение

```
> map(d->subs({exp(x[1])&.p[1]=p1},d),otv):
map(d->subs({exp(-x[1])&.p[1]=m1},d),%):
map(d->subs({exp(-x[1])&.exp(x[1])=x1},d),%):
otv:=map(simplify,%);
```

$$otv := \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

В итоге мы получили искомый ответ. Заметим, для большего числа частиц такой простой метод подстановки не приведет к упрощению результата и сокращению слагаемых. Эту проблему мы обсудим далее во втором параграфе, где мы докажем фундаментальные коммутационные соотношения для двухчастичной цепочки Тоды.

Мы увидим далее, что это недостаточно эффективный путь для работы с коммутаторами. В силу этого, перейдем к дальнейшему изучению возможностей системы Maple .

Очистим память и снова определим операцию коммутирования. Затем мы перейдем к изучению свойств данной операции. Все примеры далее выполняются последовательно!

```
> restart:
Cm:=(A,B)->A&*B-B&*A;
```

$$Cm := (A, B) \rightarrow `&*`(A, B) - `&*`(B, A)$$

```
> xp:=p&*x+I*h;
```

$$xp := (p \ \&* \ x) + I \, h$$

- **Свойства процедуры вычисления коммутаторов Cm**

Проверим свойства коммутатора. Начнем с тривиального

```
> Cm(A,A);
```

$$0$$

Однако, далее при рассмотрении следующего равенства **[** A, B **] - [** B, A **] = 2 [** A, B **]** мы уже испытываем затруднения

> **Cm(A,B)-Cm(B,A);**

$$2\,(A\,\&{*}\,B)-2\,(B\,\&{*}\,A)$$

> **factor(%);**

$$2\,(A\,\&{*}\,B)-2\,(B\,\&{*}\,A)$$

Все бы хорошо, но я не знаю достаточно простого пути конвертировать данное выражение к **2 Cm(A,B)** !

Другие проблемы связаны с представлением чисел и констант. Т.е. нам надо сказать Maple, что один символ обозначает оператор, а другой символ отвечает числу. Например

> **expand(Cm(g*A,A));**

$$(g\,A\,\&{*}\,A)-(A\,\&{*}\,g\,A)$$

Мы не можем упростить данное выражение, пока не подставим **значение** вместо **символа**

> **expand(Cm(3*A,A));**

$$0$$

Попробуем определить, что h это константа, а не оператор

> **constants:=constants,h;**

$$constants := false, \gamma, \infty, true, Catalan, FAIL, \pi, h$$

> **expand(Cm(h*A,A));**

$$0$$

Как видим, встроенное определение помогло нам в данном случае.

- **Изменение порядка операторов в выражении**

Один из стандартных приемов квантовой механики - изменение порядка следования операторов при помощи коммутационных соотношений. Рассмотрим данную операцию в Maple .

Попробуем использовать операцию **подстановки** ($subs$) для нашего определения коммутатора

> **subs(x&*p=xp,Cm(x,p));**

$$I\,h$$

Однако этот прием не работает для нескольких операторов

> **C1:=Cm(p&*p,x);**

$$C1 := ((p \,\&^* p)\,\&^* x) - (x \,\&^* (p \,\&^* p))$$

> **subs(x&*p=xp,expand(C1));**

$$\&^*(p, p, x) - \&^*(x, p, p)$$

Другой способ подстановки для вычисления коммутаторов так же работает только для пары операторов

> **C2:=c(p*I/h,x);**

$$C2 := \mathrm{c}\left(\frac{I p}{h}, x\right)$$

> **R1p:=c(p,x)=-I*h;**

$$R1p := \mathrm{c}(p, x) = -I h$$

> **subs(R1p,C2);**

$$\mathrm{c}\left(\frac{I p}{h}, x\right)$$

Однако для трех операторов это не работает

> **C3:=c(p/(-I*h),x&*x);**

$$C3 := \mathrm{c}\left(\frac{I p}{h}, x \,\&^* x\right)$$

> **subs(R1p,C3);**

$$\mathrm{c}\left(\frac{I p}{h}, x \,\&^* x\right)$$

Далее мы сами напишем процедуру, которая могла бы изменять порядок сомножителей в выражении.

- **Изменение порядка в произведении координаты и момента**

Рассмотрим пример процедуры, которая анализирует выражение вида &*(a,b,c,...) и выносит все операторы типа *x* налево, используя значение коммутатора [*x, p*]=I h . Заметим, что процедура умеет анализировать только операторы *x* , *p* и скаляры, но не функции от операторов!

> **txp:=proc() local i,n,is,i1,ct,m,r1,r2; global h;**
n:=nops(args[1]);
if n=1 then RETURN(op(1,args[1]));
elif n=2 then
if op(1,args[1])='p' and op(2,args[1])='x' then
RETURN(&*(x,p)-I*h); else RETURN(args[1]); fi; fi;
for i from 1 to n do:
ct[i]:=op(i,args[1]); od:
is:=0: for i from 1 to n-1 do:m:=n+1-i:
if ct[m]='x' and ct[m-1]='p' then
is:=m-1; break; fi; od:
if is=0 then RETURN(args[1]); fi;

```
i1:=n-is-1;
r1:=[]: r2:=[]:
for i from 1 to is-1 do:
r1:=[op(r1),ct[i]]: r2:=[op(r2),ct[i]]: od:
r1:=[op(r1),ct[is+1],ct[is]]:
for i from is+2 to n do:
r1:=[op(r1),ct[i]]: r2:=[op(r2),ct[i]]: od:
&*(op(r1))+(-I*h)*&*(op(r2));
end:
```

Данная процедура производит только одну перестановку в объекте типа &*() за время работы. Проверим это

> **txp(&*(p,x,x,p,x));**

$$\&*(p, x, x, x, p) - I\,h\,\&*(p, x, x)$$

> **txp(%);**

$$\&*(p, x, x, x, p) - I\,h\,\&*(p, x, x)$$

Как видим, во второй раз написанная нами процедура не приводит к перестановке сомножителей. Для того, чтобы понять почему так происходит - посмотрите тип аргумента операции и получаемого ответа. Другой пример:

> **txp(&*(p,x,x,x,p));**

$$\&*(x, p, x, x, p) - I\,h\,\&*(x, x, p)$$

Итак, мы имеем операцию, которая один раз изменяет порядок сомножителей, используя явное определение [p, x]=I h . Далее нам необходимо научиться применять данную операцию к линейной комбинации произведений операторов.

- **Подготовка (можно пропустить)**

В целях изучения Maple, мы не сразу реализуем наиболее эффективный алгоритм. Подойдем к необходимому нам ответу постепенно.

Итак, мы хотим построить процедуру pre1 такую, что:
1. Если аргумент процедуры есть функция, то предполагается что эта функция типа &* . Это легко проверить операцией op(0,args[1])=`&*` .
2. Если аргумент процедуры есть сумма, то мы используем нашу процедуру рекуррентно.

В этой реализации есть один существенный недостаток - мы не можем менять местами функции от

$$[p, \mathrm{f}(x)] = I\,h\left(\frac{\partial}{\partial x}\mathrm{f}(x)\right)$$

операторов. Например, мы не можем вычислить следующий коммутатор .

Реализуем принципы выписанные выше

```
> pre1:=proc() local i,n,res;
if type(args[1],function) then txp(args[1]);
elif type(args[1],`+`) then
n:=nops(args[1]);
res:=add(pre1(op(i,args[1])),i=1..n);
RETURN(expand(subs(&*()=1,res)));
else
print(`Тип данного аргумента ни функция, ни сумма`); fi;
end:
```

Данная процедура работает только для простых сумм

> **res:=pre1(&*(p,x,x,p)+&*(p,x,p));**

$$res := \&*(x,p,x,p) - lh(x\ \&*p) + \&*(x,p,p) - lhp$$

и не работает для разностей (так как выражение типа -&*() не воспринимается как единый операнд)

> **pre1(&*(p,x,x,p)-&*(p,x,p));**

Тип данного аргумента ни функция, ни сумма

Error, (in pre1) invalid terms in sum

Займемся теперь произведением, хотя бы для того, чтобы определить (-1)*&*()

> **pre2:=proc() local i,n;**
if type(args[1],function) then txp(args[1]);
elif type(args[1],`+`) then
n:=nops(args[1]);
add(pre2(op(i,args[1])),i=1..n);
RETURN(expand(subs(&*()=1,%)));
elif type(args[1],`*`) then
n:=nops(args[1]);
mul(pre2(op(i,args[1])),i=1..n);
RETURN(expand(subs(&*()=1,%)));
elif typematch(args[1],constant) then RETURN(args[1]);
elif type(args[1],symbol) then RETURN(args[1]);
else print(`The type was not function and not + `);
fi;
end:

Проверим возможности данной процедуры

> **res:=pre2(&*(p,x,x,p)+&*(p,x,p));**

$$res := \&*(x,p,x,p) - lh(x\ \&*p) + \&*(x,p,p) - lhp$$

Не так плохо, но придется применить процедуру еще раз

> **pre2(res);**

$$\&*(x,x,p,p) - 2\,lh(x\ \&*p) + \&*(x,p,p) - lhp$$

На этом пока закончим обучение.

Мы приобрели некоторый опыт, который позволит нам построить более эффективную процедуру. Рассмотрим одну из реализаций данной операции перестановки и изучим ее свойства.

- **Операция перестановки сомножителей**

Используя общие принципы анализа аргумента процедуры, мы написали следующую реализацию операции перестановки:

> **Perm:=proc() local i,n;**
if type(args[1],function) then txp(args[1]);
elif typematch(args[1],constant) then RETURN(args[1]);

```
elif type(args[1],name) then RETURN(args[1]);
elif type(args[1],`^`) then RETURN(args[1]);
elif type(args[1],`+`) then n:=nops(args[1]);
add(Perm(op(i,args[1])),i=1..n);
expand(subs(&*()=1,&*(x)=x,&*(p)=p,&*(x,x)=x^2,&*(p,p)=p^2,%));
RETURN(%);
elif type(args[1],`*`) then
n:=nops(args[1]);
mul(Perm(op(i,args[1])),i=1..n);
expand(subs(&*()=1,&*(x)=x,&*(p)=p,&*(x,x)=x^2,&*(p,p)=p^2,%));
RETURN(%);
else print(`The type was not function, not string, and not + or * `);
fi;
end:
```

Посмотрим на работу процедуры

> **otv:=Perm(&*(p,x,p,x));**

$$otv := \&*(p, x, x, p) - I h (p \&* x)$$

> **Perm(otv);**

$$\&*(x, p, x, p) - 2 I h (x \&* p) - h^2$$

> **Perm(%);**

$$\&*(x, x, p, p) - 3 I h (x \&* p) - h^2$$

Напомним, что коммутатор двух функций можно сравнить с операцией дифференцирования. Например, для x^2

> **Perm(&*(p,x,x)-&*(x,x,p));**

$$\&*(x, p, x) - I h x - \&*(x, x, p)$$

> **Perm(%);**

$$-2 I h x$$

Однако

> **Perm(&*(p,x^2)-&*(x^2,p));**

$$(p \&* x^2) - (x^2 \&* p)$$

> **Perm(%);**

$$(p \&* x^2) - (x^2 \&* p)$$

Как видим, объекты &*(x,x,x...,x) и x^n имеют разные типы и разное представление в Maple. В силу этого нам необходимо переписать процедуры.

Если мы ограничим число N сомножителей в выражении, то мы можем делать итерации автоматически , пока не перенесем все операторы *x* налево. Например, пусть N=100

```
> push1:=proc() local i,res_old,res_new;
res_old:=args[1];
for i from 1 to 100 do: res_new:=Perm(res_old);
if res_old=res_new then RETURN(res_new): fi;
res_old:=res_new: od:
end:
```

Проверим работу процедуры

```
> push1(Cm(p/(-I*h),x&*x&*x&*x));
```

$$\frac{I\,\&*(p,x^2,x,x)}{h}+\frac{-I\,\&*(x^2,x,x,p)}{h}$$

```
> push1(Cm(p/(-I*h),x^4));
```

$$\frac{I\,(p\,\&*\,x^4)}{h}+\frac{-I\,(x^4\,\&*\,p)}{h}$$

Так как мы имеем дело с операторами, то формально произведение оператора отличается от его степени. x^n Рассмотрим, как одно выражение типа можно перевести в выражение типа $x\,\&\ldots\,\&(x)$.

- **Как конвертировать &*(x,x,x...) в степени от x и наоборот**

Здесь мы приведем только один из возможных способов.

Итак, перевести степень оператора в произведение операторов можно, например, с помощью следующей процедуры

```
> powoff:=proc() local i,n,nu,opr,nop,xop,res,resi;
if type(args[1],function) then
n:=nops(args[1]); res:=[];
for i from 1 to n do
opr:=op(i,args[1]);
if type(opr,`^`) then
xop:=op(1,opr); nop:=op(2,opr);
resi:=[];
for nu from 1 to nop do
resi:=[op(resi),xop]:
od:
else resi:=[opr] fi: res:=[op(res),op(resi)]
od:
&*(op(res));
elif type(args[1],`*`) then mul(powoff(op(nu,args[1])),nu=1..nops(args[1]));
elif type(args[1],`+`) then add(powoff(op(nu,args[1])),nu=1..nops(args[1]));
elif type(args[1],`^`) and type(op(1,args[1]),constant)=false then
RETURN(&*(seq(op(1,args[1]),nu=1..op(2,args[1]))));
else RETURN(args[1]) fi;
end:
```

Проверим как работает эта процедура

> **powoff(&*(x,p^3));**

$$\&*(x,p,p,p)$$

> **powoff(x&*p&*x^9);**

$$\&*(x\ \&*p,x,x,x,x,x,x,x,x,x)$$

> **powoff(I*&*(x^3,p^4));**

$$I\ \&*(x,x,x,p,p,p,p)$$

> **powoff(I*&*(x^3,p^4)+h*(x^2&*p^5));**

$$I\ \&*(x,x,x,p,p,p,p)+h\ \&*(x,x,p,p,p,p,p)$$

> **powoff(x^2);**

$$x\ \&*\ x$$

Теперь напишем процедуру перевода произведения операторов в степень оператора

> **powon:=proc() local i,n,nu,opr,nop,ic,xop,res,opr1;**
if type(args[1],function) then
n:=nops(args[1]); if n=1 then RETURN(args[1]) fi;
res:=[]; ic:=1;
for i from 1 to n-1 do:
opr:=op(i,args[1]); opr1:=op(i+1,args[1]);
if opr=opr1 then ic:=ic+1;
else
if ic>1 then res:=[op(res),opr^ic]; ic:=1;
else res:=[op(res),opr];
fi;
fi;
od:
if ic=1 then res:=[op(res),opr1];
else res:=[op(res),opr^ic];
fi;
&*(op(res));
elif type(args[1],`*`) then mul(powon(op(nu,args[1])),nu=1..nops(args[1]));
elif type(args[1],`+`) then add(powon(op(nu,args[1])),nu=1..nops(args[1]));
else RETURN(args[1])
fi;
end:

Посмотрим как работает данная процедура

> **powon(&*(x,x,x,p));**

$$x^3\ \&*\ p$$

> **powon(&*(p,x,x,x,p,p,x));**

$$\&*(p,x^3,p^2,x)$$

> **powon(&*(p,x,x,x,p,p,x,x,x));**

$$\&*(p, x^3, p^2, x^3)$$

> powon(&*(p,p,p,x,x)/h+I*&*(x,p,p,x,x,x));

$$\frac{p^3\,\&*\,x^2}{h}+I\,\&*(x, p^2, x^3)$$

Теперь объединим процедуры перестановки, повторения перестановок несколько раз и преобразования степени в произведение. Позвольте объяснить, зачем мы хотим объединить эти процедуры. Как только во всех частях выражения мы вынесем все операторы импульса направо, а координаты налево, мы сможем заменить символ некоммутативного выражения на символ обычного умножения. В свою очередь это позволит привести подобные члены в выражении без использования операции подстановки.

- **Процедура переноса операторов координаты налево, используя выражение [x,p] = I h**

Напомним, что мы ограничили общее число сомножителей в выражении N=100

Напишем теперь подпрограмму, которая использует все введенные нами ранее процедуры

> **push:=proc() local i,res_old,res_new;**
res_old:=powoff(args[1]);
for i from 1 to 100 do:
res_new:=Perm(res_old);
if res_old=res_new then RETURN(powon(res_new)): fi;
res_old:=res_new:
od:
powon(res_new);
end:

Проверим работу данной процедуры

> **push(Cm(p/(-I*h),x&*x&*x&*x));**

$$\frac{I\,\&*(p, x^2, x^2)}{h}+\frac{-I\,\&*(x^2, x^2, p)}{h}$$

> push(Cm(p/(-I*h),x^4));

$$4\,\&*(x^3)$$

> ***push(Cm(p/(-I*h),x^12));***

$$12\,\&*(x^{11})$$

Заметим, что предела совершенству нет, и данная процедура имеет много недостатков, которые достаточно легко найти и, поэтому, не очень эффективна.