## FullKroneckerExpand

- `FullKroneckerExpand[δ][expr]` will fully expand tensors with label $\delta$, assumed to be generalized Kroneckers, as first order Kroneckers.

In Tensorial all Kronecker symbols must have one Void in each slot just like all other indexed objects. Many texts use indices in both the up and down positions, taking advantage of the fact that Kroneckers must be even order with equal number of up and down indices.

Labels other that $\delta$ can be used to represent the Kronecker.

See also: `KroneckerContract`, `PartialKroneckerExpand`, `KroneckerAbsorb`, `KroneckerEvaluate`.

### Examples

```
In[1]:= Needs["TensorCalculus4`Tensorial`"]

In[2]:= DefineTensorShortcuts[δ, #] & /@ {2, 4, 6};
        DefineTensorShortcuts[κ, 6]
```

The following routine fully expands a generalized Kronecker.

```
In[4]:= δuuuddd[i, j, k, r, s, t]
        % // FullKroneckerExpand[δ]
```

$$\text{Out[4]=}\quad \delta^{i\,j\,k}{}_{r\,s\,t}$$

$$\text{Out[5]=}\quad -\delta^{i}{}_{t}\,\delta^{j}{}_{s}\,\delta^{k}{}_{r}+\delta^{i}{}_{s}\,\delta^{j}{}_{t}\,\delta^{k}{}_{r}+\delta^{i}{}_{t}\,\delta^{j}{}_{r}\,\delta^{k}{}_{s}-\delta^{i}{}_{r}\,\delta^{j}{}_{t}\,\delta^{k}{}_{s}-\delta^{i}{}_{s}\,\delta^{j}{}_{r}\,\delta^{k}{}_{t}+\delta^{i}{}_{r}\,\delta^{j}{}_{s}\,\delta^{k}{}_{t}$$

```
In[6]:= δuudd[i, j, r, s]
        % // FullKroneckerExpand[δ]
```

$$\text{Out[6]=}\quad \delta^{i\,j}{}_{r\,s}$$

$$\text{Out[7]=}\quad -\delta^{i}{}_{s}\,\delta^{j}{}_{r}+\delta^{i}{}_{r}\,\delta^{j}{}_{s}$$

The Kronecker does not have to be in standard form.

```
In[8]:= δududud[i, s, k, r, j, t]
        % // FullKroneckerExpand[δ]
```

$$\text{Out[8]=}\quad \delta^{i}{}_{s}{}^{k}{}_{r}{}^{j}{}_{t}$$

$$\text{Out[9]=}\quad -\delta^{i}{}_{t}\,\delta^{j}{}_{s}\,\delta^{k}{}_{r}+\delta^{i}{}_{s}\,\delta^{j}{}_{t}\,\delta^{k}{}_{r}+\delta^{i}{}_{t}\,\delta^{j}{}_{r}\,\delta^{k}{}_{s}-\delta^{i}{}_{r}\,\delta^{j}{}_{t}\,\delta^{k}{}_{s}-\delta^{i}{}_{s}\,\delta^{j}{}_{r}\,\delta^{k}{}_{t}+\delta^{i}{}_{r}\,\delta^{j}{}_{s}\,\delta^{k}{}_{t}$$

The Kronecker can be represented by a label different than $\delta$

```
In[10]:= κuuuddd[i, j, k, r, s, t]
         % // FullKroneckerExpand[κ]
```

$$\text{Out[10]=}\quad \kappa^{i\,j\,k}{}_{r\,s\,t}$$

$$\text{Out[11]=}\quad -\kappa^{i}{}_{t}\,\kappa^{j}{}_{s}\,\kappa^{k}{}_{r}+\kappa^{i}{}_{s}\,\kappa^{j}{}_{t}\,\kappa^{k}{}_{r}+\kappa^{i}{}_{t}\,\kappa^{j}{}_{r}\,\kappa^{k}{}_{s}-\kappa^{i}{}_{r}\,\kappa^{j}{}_{t}\,\kappa^{k}{}_{s}-\kappa^{i}{}_{s}\,\kappa^{j}{}_{r}\,\kappa^{k}{}_{t}+\kappa^{i}{}_{r}\,\kappa^{j}{}_{s}\,\kappa^{k}{}_{t}$$

```
In[12]:= ClearTensorShortcuts[δ, #] & /@ {2, 4, 6};
         ClearTensorShortcuts[κ, 6]
```

## GenerateBasisTensors

- `GenerateBasisTensors[`*e, configuration*`]` will generate the set of basis tensors on basis vectors e with a a given configuration of up and down basis vectors.

The configuration is specified by a String of u's and d's for up and down.

It is assumed that tensor shortcuts have been defined for e.

The basis is generated as an array of `CircleTimes` expressions of the basis vectors.

See also: `CircleEvalRule`, `EvaluateSlots`.

### Examples

```
In[1]:= Needs["TensorCalculus4`Tensorial`"]

In[2]:= DefineTensorShortcuts[e, 1]
```

With the basis vectors themselves we may or may not wish to eliminate the `CircleTimes`, depending on how we are going to use them.

```
In[3]:= basis1 = GenerateBasisTensors[e, "d"]
        basis2 = GenerateBasisTensors[e, "u"] /. CircleTimes → Identity
        (basis1.#)[basis2.#] &[{a, b, c}]
        % // EvaluateSlots[e, g]

Out[3]= {⊗e_1, ⊗e_2, ⊗e_3}

Out[4]= {e^1, e^2, e^3}

Out[5]= (a (⊗e_1) + b (⊗e_2) + c (⊗e_3))[a e^1 + b e^2 + c e^3]

Out[6]= a^2 + b^2 + c^2
```

The various 2nd order basis dyads.

```
In[7]:= Print[GenerateBasisTensors[e, #]] & /@ {"uu", "ud", "du", "dd"};

        {e^1 ⊗ e^1, e^1 ⊗ e^2, e^1 ⊗ e^3, e^2 ⊗ e^1, e^2 ⊗ e^2, e^2 ⊗ e^3, e^3 ⊗ e^1, e^3 ⊗ e^2, e^3 ⊗ e^3}

        {e^1 ⊗ e_1, e^1 ⊗ e_2, e^1 ⊗ e_3, e^2 ⊗ e_1, e^2 ⊗ e_2, e^2 ⊗ e_3, e^3 ⊗ e_1, e^3 ⊗ e_2, e^3 ⊗ e_3}

        {e_1 ⊗ e^1, e_1 ⊗ e^2, e_1 ⊗ e^3, e_2 ⊗ e^1, e_2 ⊗ e^2, e_2 ⊗ e^3, e_3 ⊗ e^1, e_3 ⊗ e^2, e_3 ⊗ e^3}

        {e_1 ⊗ e_1, e_1 ⊗ e_2, e_1 ⊗ e_3, e_2 ⊗ e_1, e_2 ⊗ e_2, e_2 ⊗ e_3, e_3 ⊗ e_1, e_3 ⊗ e_2, e_3 ⊗ e_3}
```

A set of 3rd order basis triads.

```
In[8]:= GenerateBasisTensors[e, "uuu"]

Out[8]= {e^1 ⊗ e^1 ⊗ e^1, e^1 ⊗ e^1 ⊗ e^2, e^1 ⊗ e^1 ⊗ e^3, e^1 ⊗ e^2 ⊗ e^1, e^1 ⊗ e^2 ⊗ e^2, e^1 ⊗ e^2 ⊗ e^3,
         e^1 ⊗ e^3 ⊗ e^1, e^1 ⊗ e^3 ⊗ e^2, e^1 ⊗ e^3 ⊗ e^3, e^2 ⊗ e^1 ⊗ e^1, e^2 ⊗ e^1 ⊗ e^2, e^2 ⊗ e^1 ⊗ e^3, e^2 ⊗ e^2 ⊗ e^1,
         e^2 ⊗ e^2 ⊗ e^2, e^2 ⊗ e^2 ⊗ e^3, e^2 ⊗ e^3 ⊗ e^1, e^2 ⊗ e^3 ⊗ e^2, e^2 ⊗ e^3 ⊗ e^3, e^3 ⊗ e^1 ⊗ e^1, e^3 ⊗ e^1 ⊗ e^2,
         e^3 ⊗ e^1 ⊗ e^3, e^3 ⊗ e^2 ⊗ e^1, e^3 ⊗ e^2 ⊗ e^2, e^3 ⊗ e^2 ⊗ e^3, e^3 ⊗ e^3 ⊗ e^1, e^3 ⊗ e^3 ⊗ e^2, e^3 ⊗ e^3 ⊗ e^3}
```

```
In[9]:= ClearTensorShortcuts[e, 1]
        Clear[basis1, basis2]
```

## GetBaseIndices

■ `GetBaseIndices[`*index*`]` will return the list of base indices associated with index.

The list of indices will be the standard `BaseIndices` unless the flavor of the index is one of the special ones declared in `DeclareBaseIndices`.

See also: `DeclareBaseIndices`, `NDim`, `GetBaseIndices`, `DeclareIndexFlavor`, `BaseIndexQ`.

### Examples

```
In[1]:= Needs["TensorCalculus4`Tensorial`"]

In[2]:= oldindices = CompleteBaseIndices;
        oldflavors = IndexFlavors;
        DeclareIndexFlavor[{red, Red}, {space, SuperStar}, {blue, Blue}]
```

The following statement associates separate sets of base indices with red and space flavored indices.

```
In[5]:= DeclareBaseIndices[{0, 1, 2, 3}, {red, {A, B}}, {space, {1, 2, 3}}]
```

Now red and space flavored indices have their own sets of base indices, but black and blue indices have the standard base indices.

```
In[6]:= {a, red@a, space@a, blue@a}
        GetBaseIndices /@ %

Out[6]= {a, a, a*, a}

Out[7]= {{0, 1, 2, 3}, {A, B}, {1, 2, 3}, {0, 1, 2, 3}}
```

Restore the initial values...

```
In[8]:= DeclareBaseIndices @@ oldindices
        DeclareIndexFlavors @@ oldflavors;
        Clear[oldindices, oldflavors]
```

## GetIndexFlavor

■ GetIndexFlavor[*indexl*] returns the flavor on an index.

This is principally a service routine for programming other routines.

Identity is returned if the index has no flavor.

$Failed is returned if the flavor is not a currently declared flavor.

See also: IndexFlavors, DeclareIndexFlavor, ClearIndexFlavor, ToFlavor.

### Examples

```
In[1]:= Needs["TensorCalculus4`Tensorial`"]
```

The following saves the current state.

```
In[2]:= oldflavors = IndexFlavors;
        ClearIndexFlavor /@ oldflavors;
```

This declares a set of index flavors...

```
In[4]:= DeclareIndexFlavor[{red, Red}, {green, Cerulean}, {rocket, SuperStar}];
```

Here is a list of indices with various flavors.

```
In[5]:= {i, red@i, green@i, rocket@i, blue[i]}
        GetIndexFlavor /@ %

Out[5]= {i, i, i, i*, blue[i]}

Out[6]= {Identity, red, green, rocket, $Failed}
```

This resets to the original state.

```
In[7]:= ClearIndexFlavor[IndexFlavors];
        DeclareIndexFlavor /@ oldflavors;
        Clear[oldflavors]
```

## HoldOp

■ `HoldOp[`*operation*`][`*expr*`]` will prevent the given operation from being evaluated in expr.

Nevertheless, other operations within expr, such as tensor shortcuts, will be evaluated. .

Operation should be the Head of an expression. It may be a pattern and even a pattern that includes alternatives..

The Hold will be released by `ReleaseHold`.

HoldOp is useful when there are formatted operations that have definitions that perform automatic evaluations. HoldOp will prevent these evaluations and thus allow one to show the starting point of a calculation.

See also: `SymbolsToPatterns`.

### Examples

```
In[1]:= Needs["TensorCalculus4`Tensorial`"]

In[2]:= DefineTensorShortcuts[{{p, q}, 1}]
```

The various derivative operators are formatted. They also have a number of definitions, such as linear and Liebnizian breakouts, that are automatically applied. For didactic purposes we may wish to see the starting expression. This can be achieved by using the `HoldOp` command.

```
In[3]:= PartialD[pu[a] qd[b], c] // HoldOp[PartialD]
        % // ReleaseHold
```

$$\text{Out[3]=}\quad (p^a\, q_b)_{,c}$$

$$\text{Out[4]=}\quad q_{b,c}\, p^a + p^a{}_{,c}\, q_b$$

Notice that the tensor shortcuts were evaluated even though the derivative is held.

```
In[5]:= CovariantD[pu[a] qd[b] + pd[b] qu[a], c] // HoldOp[CovariantD]
        % // ReleaseHold
```

$$\text{Out[5]=}\quad (p_b\, q^a + p^a\, q_b)_{;c}$$

$$\text{Out[6]=}\quad q_{b;c}\, p^a + q^a{}_{;c}\, p_b + p_{b;c}\, q^a + p^a{}_{;c}\, q_b$$

```
In[7]:= TotalD[pu[a] qd[b], t] // HoldOp[TotalD]
        % // ReleaseHold
```

$$\text{Out[7]=}\quad \frac{dp^a\, q_b}{dt}$$

$$\text{Out[8]=}\quad q_b \frac{dp^a}{dt} + p^a \frac{dq_b}{dt}$$

`HoldOp` can also be used to maintain an order in standard expressions such as `Plus` or `Times`.

```
In[9]:= (d + c) (c b + a) // HoldOp[Plus | Times]
        % // ReleaseHold

Out[9]= (d + c) (c b + a)

Out[10]= (a + b c) (c + d)

In[11]:= qd[i] pu[i] // HoldOp[Times]
         % // ReleaseHold

Out[11]= q_i p^i

Out[12]= p^i q_i
```

The head may be an expression.

```
In[13]:= foo[a_][b_] := a b
```

Here the HoldForm was wrapped around the entire foo expression.

```
In[14]:= foo[a][3^2] + foo[3^2][b] // HoldOp[foo[_]]
         % // FullForm
         % // ReleaseHold

Out[14]= foo[9][b] + foo[a][9]

   Out[15]//FullForm=
        Plus[HoldForm[foo[9][b]], HoldForm[foo[a][9]]]

Out[16]= 9 a + 9 b
```

In the following the HoldForm was wrapped only around the Head.

```
In[17]:= foo[a][3^2] + foo[3^2][b] // HoldOp[foo]
         % // FullForm
         % // ReleaseHold

Out[17]= foo[9][b] + foo[a][9]

   Out[18]//FullForm=
        Plus[HoldForm[foo[9]][b], HoldForm[foo[a]][9]]

Out[19]= 9 a + 9 b
```

Restore state

```
In[20]:= ClearTensorShortcuts[{{p, q}, 1}]
```