Free Lie Algebras Routines

Pensieve header: A free-Lie calculator, pensieve://Projects/WKO4/ branch (most current), continues pensieve://2014-01/. Now understands the Drinfel'd-Kohno algebra.

Non-current versions of this package are available at http://drorbn.net/AcademicPensieve/Projects/WKO4/Archive/.

To Do

ToDo

* Add "?" help lines and/or tooltip popups.

ToDo

* Revise LW and CW I/O; have declarable "generators".

ToDo

* Consider "trees and wheels printing".

Prolog

```
BeginPackage["FreeLie`"];
Print["FreeLie` implements / extends ",
  Sort@{"*", "+", "**", "$SeriesShowDegree", "⟨⟩", "∫", "≡", ad, Ad, adSeries, AllCyclicWords,
    AllLyndonWords, AllWords, Arbitrator, ASeries, AW, b, BCH, BooleanSequence, BracketForm,
    BS, CC, Crop, cw, CW, CWS, CWSeries, D, Deg, DegreeScale, DerivationSeries, div, DK, DKS,
    DKSeries, EulerE, Exp, Inverse, j, J, JA, LieDerivation, LieMorphism, LieSeries, LS, LW,
    LyndonFactorization, Morphism, New, RandomCWSeries, Randomizer, RandomLieSeries, RC, SeriesSolve,
    Support, t, tb, TopBracketForm, tr, UndeterminedCoefficients, αMap, Γ, ι, Λ, σ, ℏ, ⎵, ⌢},
  "."];
Print["FreeLie` is in the public domain. Dror Bar-Natan is committed
    to support it within reason until July 15, 2022. This is version 150814."];
$SeriesShowDegree = 3;
Begin["`Private`"];
```

"New"

```
SetAttributes[New, HoldAll];
(*New[type_[srn_], def_] := Module[   (* "srn" for "self-reference name" *)
   {un}, (* "un" for "unique name" *)
   un=Unique[ToString[type]<>"$"];
   ReleaseHold[Hold[def] /. srn → un];
   type[un]
  ];*)
New[type_[srn_], def_] := ( (* "srn" for "self-reference name" *)
   (*Print["new object of type ", type//ToString, " and srn ", srn];*)
   ReleaseHold[Hold[def; type[srn]] /. srn → Unique[ToString[type] <> "$"]]
  );
```

Words and Lyndon Words

A Lyndon word is a word lexicographically smaller than all of its proper right factors.

```
AllWords[n_, ab_List] := AW @@@ Tuples[ab, n];
LW /: LW[] < LW[__] = True;
LW /: _LW < LW[] = False;
LW /: LW[x_, xs___] < LW[x_, ys___] := LW[xs] < LW[ys];
LW /: LW[x_, ___] < LW[y_, ___] /; (x =!= y) := OrderedQ[{x, y}];
LW /: x_LW > y_LW := y < x;
LW /: x_LW ≤ y_LW := ! (y < x);
LW /: x_LW ≥ y_LW := ! (x < y);
LyndonQ[w_AW] := And @@ (
    (LW @@ w < LW @@ #) & /@ Table[Drop[w, i], {i, 1, Length[w] - 1}]);
AllLyndonWords[n_Integer, ab_List] := AllLyndonWords[n, ab] =
   LW @@@ Select[AllWords[n, ab /. LW[w_] ⧴ w], LyndonQ];
AllLyndonWords[{n_}, ab_List] := Join @@ Table[AllLyndonWords[k, ab], {k, n}];
Deg[w_LW] := Length[w];
LyndonFactorization[w_LW /; Deg[w] == 1] := w;
LyndonFactorization[w_LW /; Deg[w] > 1] := Module[{rf},
   rf = First[Sort[Table[Drop[w, i], {i, 1, Length[w] - 1}], Less]];
   LW /@ {Drop[w, -Length[rf]], rf}];
LW[w_LW] := w;
BracketForm[LW[c_]] := ToString[c];
BracketForm[w_LW] := BracketForm[w] = StringJoin[Flatten[{
      "[", BracketForm /@ LyndonFactorization[w], "]"
     }]];
BracketForm[expr_] := expr /. w_LW ⧴ BracketForm[w];
topbracketform[LW[c_]] := c;
topbracketform[w_LW] := topbracketform[w] = Overscript[
    Row[Riffle[topbracketform /@ LyndonFactorization[w], " "]], ⎴];
TopBracketForm[LW[c_]] := Overscript[c, ⎴];
TopBracketForm[w_LW] := topbracketform[w];
TopBracketForm[w_CW] := Overscript[StringJoin @@ (ToString /@ w), ⏜];
TopBracketForm[expr_] := expr /. w_LW | w_CW | w_DK ⧴ TopBracketForm[w];
Format[w_LW] := TopBracketForm[w];
```

The Bracket for Lie Elements

```
b[0, _] = 0; b[_, 0] = 0;
b[c_ * (x_AW | x_LW), y_] := Expand[c b[x, y]];
b[x_, c_ * (y_AW | y_LW)] := Expand[c b[x, y]];
b[x_Plus, y_] := b[#, y] & /@ x;
b[x_, y_Plus] := b[x, #] & /@ y;
b[w_LW, z_LW] := LWAdjoint[w][z];
ad[x_][y_] := b[x, y];
```

```
LWAdjoint[w_] := LWAdjoint[w] = Module[{u},
    u = Unique[LWAct];
    u[z_] := u[z] = Which[
       w === z, 0,
       z < w, Expand[-b[z, w]],
       Deg[w] == 1, Join[w, z],
       True, Module[{x, y},
        {x, y} = LyndonFactorization[w];
        If[y ≥ z,
         Join[w, z],
         b[x, LWAdjoint[y][z]] + b[LWAdjoint[x][z], y]
        ]
       ]
      ];
    u
   ];
```

BooleanSequence

```
BS[v : (True | False)] := BS[v] = New[BooleanSequence[ser], ser[_] = v];
BooleanSequence[bs_Symbol][{dd_Integer}] := TopBracketForm[Append[
    BS @@ (Table[bs[d], {d, 0, dd}] //. {n_. True, True, rest___} :> {(n + 1) True, rest}),
    "..."
   ]];
BooleanSequence[bs_Symbol][e___] := bs[e];
Format[bs_BooleanSequence, StandardForm] := bs[{$SeriesShowDegree}];
BooleanSequence /: And[bss__BooleanSequence] := New[BooleanSequence[bs],
   bs[d_Integer] := bs[d] = And @@ ((#[d]) & /@ {bss})
  ];
BooleanSequence /: ħ^k_. * BooleanSequence[bs_Symbol] := New[BooleanSequence[nbs],
   nbs[d_Integer] := bs[d - k]
  ];
BooleanSequence /: TrueQ[bs_BooleanSequence] := New[BooleanSequence[nbs],
   nbs[d_Integer] := nbs[d] = TrueQ[bs[d]]
  ];
```

LieSeries

```
LieSeries[ser_Symbol][{dd_Integer}] := Append[TopBracketForm[LS @@ Table[ser[d], {d, dd}]], "..."];
LieSeries[ser_Symbol][e___] := ser[e];
Format[s_LieSeries, StandardForm] := TopBracketForm[s[{$SeriesShowDegree}]];
ShowLieSeries[d_Integer][s_LieSeries] := s[{d}];
LS[s_] := MakeLieSeries[s];
MakeLieSeries[s_LieSeries] := s;
MakeLieSeries[expr_] := MakeLieSeries[expr] = New[LieSeries[ser],
    ser[d_Integer] := ser[d] = Expand[expr /. w_LW /; Deg[w] ≠ d → 0]
   ];
s1_LieSeries ≡ s2_LieSeries := New[BooleanSequence[bs],
   bs[0] = True;
   bs[d_Integer] := bs[d - 1] && (s1[d] == s2[d]);
  ];
Crop[s_LieSeries, d_Integer] := Crop[s, d] = New[LieSeries[ser],
    ser[dd_Integer] := If[dd ≤ d, s[dd], 0]
   ];
RandomLieSeries[ab_List, opts___Rule] := Module[
   {rand = Randomizer /. {opts} /. Randomizer → (RandomInteger[{-2 Deg[#]!, 2 Deg[#]!}] / Deg[#]! &)},
   New[LieSeries[ser],
    ser[d_Integer] := ser[d] = Plus @@ ((rand[#] * #) & /@ AllLyndonWords[d, ab])
   ]
  ];
```

```
AddLieSeries[ss___LieSeries] := AddLieSeries[ss] = New[LieSeries[ser],
    ser[d_Integer] := ser[d] = Plus @@ ((#[d]) & /@ {ss})
   ];
LieSeries /: Plus[ss__LieSeries] := AddLieSeries[ss];
ScaleLieSeries[c_, s_LieSeries] := ScaleLieSeries[c, s] = New[LieSeries[ser],
    ser[d_Integer] := ser[d] = Expand[c * s[d]]
   ];
LieSeries /: c_ * s_LieSeries := ScaleLieSeries[c, s];
LieSeries /: D[ls_LieSeries, vars__] := New[LieSeries[ser],
   ser[d_Integer] := ser[d] = D[ls[d], vars]
  ];
IntegrateLieSeries[ls_LieSeries, {s_, s0_, s1_}] :=
  IntegrateLieSeries[ls, {s, s0, s1}] = New[LieSeries[ser],
    ser[d_Integer] := ser[d] = Expand[∫_{s0}^{s1} ls[d] ⅆs]
   ];
LieSeries /: Integrate[ls_LieSeries, {s_, s0_, s1_}] := IntegrateLieSeries[ls, {s, s0, s1}];
```

```
b[s1_LieSeries, s2_LieSeries] := b[s1, s2] = New[LieSeries[ser],
    ser[d_Integer] := ser[d] = Sum_{k=1}^{d-1} b[s1[k], s2[d - k]]
   ];
b[s_LieSeries, y_] := b[s, MakeLieSeries[y]];
b[x_, s_LieSeries] := b[MakeLieSeries[x], s];
```

EulerE, DegreeScale

```
LieSeries /: EulerE[s_LieSeries] := New[LieSeries[ser],
   ser[d_Integer] := ser[d] = Expand[d * s[d]]
  ];
DegreeScale[h_][s_LieSeries | s_CWSeries] := New[Head[s][ser],
   ser[d_Integer] := ser[d] = Expand[h^d s[d]]
  ];
```

adSeries, and Ad

Convention: ad(x)(z)=[x,z]. Satisfies ad([x,y])=[ad(x),ad(y)].

```
adSeries[1, x_LieSeries][ψ_LieSeries] := adSeries[1, x][ψ] = ψ;
adSeries[ad^n_., x_LieSeries][ψ_LieSeries] := adSeries[ad^n, x][ψ] = New[LieSeries[ser],
    ser[d_Integer] := ser[d] = b[x, adSeries[ad^(n-1), x][ψ]][d]
   ];
adSeries[f_, x_LieSeries][ψ_LieSeries] := adSeries[f, x][ψ] = New[LieSeries[ser],
    ser[d_Integer] := ser[d] = Module[{c},
       Expand[Sum[
         c = SeriesCoefficient[f, {ad, 0, k}];
         If[c === 0, 0, c * adSeries[ad^k, x][ψ][d]],
         {k, 0, d - 1}
        ]]
      ]
   ];
adSeries[f_, x_][ψ_] := adSeries[f, MakeLieSeries[x]][MakeLieSeries[ψ]];
Ad[x_] := adSeries[E^ad, x];
```

LieDerivation, DerivationPower, DerivationSeries

```
LieDerivation[der_][es___] := der[es];
LieDerivation[rules___Rule] := LieDerivation[{rules}];
LieDerivation[rules_List] := LieDerivation[rules] = New[LieDerivation[der],
    der[Support] = First /@ rules;
    (der[w_LW] /; Deg[w] == 1) := (der[w] = MakeLieSeries[w /. Append[rules, _LW → 0]]);
    der[w_LW] := der[w] = Module[{x, y},
       {x, y} = LyndonFactorization[w];
       AddLieSeries[b[der[x], y], b[x, der[y]]]
      ];
    der[s_LieSeries] := der[s] = New[LieSeries[ser],
       ser[d_] := ser[d] = Sum[der[s[k]][d], {k, 1, d}]];
    der[as_ASeries] := der[as] = New[ASeries[ser],
       ser[d_] := ser[d] = Sum[
          Expand[as[k] /. w_AW :> Sum[
              NonCommutativeMultiply[
               Take[w, j - 1],
               ι[der[LW[w[[j]]]][d - k + 1]],
               Drop[w, j]
              ],
              {j, k}
             ]],
          {k, 1, d}
         ]
      ];
    der[cws_CWSeries] := der[cws] = New[CWSeries[ser],
       ser[d_] := ser[d] = Sum[
          Expand[cws[k] /. w_CW :> Sum[
              tr[NonCommutativeMultiply[
                 AW @@ Take[w, j - 1],
                 ι[der[LW[w[[j]]]][d - k + 1]],
                 AW @@ Drop[w, j]
                ]],
              {j, k}
             ]],
          {k, 1, d}
         ]
      ];
    der[expr_][d_] := Expand[expr /. {w_LW :> der[w][d], s_LieSeries :> der[s][d]}];
   ];
```

```
LieDerivation /: Plus[ders__LieDerivation] := LieDerivation[Table[
    u → Total[#[u] & /@ {ders}],
    {u, Union @@ (#[Support] & /@ {ders})}
   ]];
LieDerivation /: c_ * der_LieDerivation := LieDerivation[Table[
    u → (c der[u]),
    {u, der@Support}
   ]];
```

```
b[der1_LieDerivation, der2_LieDerivation] := LieDerivation[Table[
    u → der1[der2[u]] - der2[der1[u]],
    {u, (der1@Support) ⋃ (der2@Support)}
   ]];
```

```
DerivationPower[0, der_LieDerivation][ψ_LieSeries | ψ_CWSeries] :=
  DerivationPower[0, der][ψ] = New[Head[ψ][ser],
    ser[d_Integer] := ser[d] = ψ[d]
   ];
DerivationPower[n_Integer, der_LieDerivation][ψ_LieSeries | ψ_CWSeries] :=
  DerivationPower[n, x][ψ] = New[Head[ψ][ser],
    ser[d_Integer] := ser[d] = der[DerivationPower[n - 1, der][ψ]][d]
   ];
DerivationSeries[__][0] = 0;
DerivationSeries[f_, ld_LieDerivation][ψ_LieSeries | ψ_CWSeries] :=
  DerivationSeries[f, ld][ψ] = New[Head[ψ][ser],
    ser[d_Integer] := ser[d] = Module[{c},
       Expand[Sum[
         c = SeriesCoefficient[f, {der, 0, k}];
         If[c == 0, 0, c * DerivationPower[k, ld][ψ][d]],
         {k, 0, d}
        ]]
      ]
   ];
DerivationExp[ld_LieDerivation] := DerivationSeries[E^der, ld];
LieDerivation /: Exp[ld_LieDerivation] := DerivationExp[ld];
LieDerivation /: ⅇ^ld_LieDerivation := DerivationExp[ld];
```

LieMorphism

```
LieMorphism[mor_][es___] := mor[es];
LieMorphism[rules__Rule] := LieMorphism[{rules}];
LieMorphism[rules_List] := LieMorphism[rules] = New[LieMorphism[mor],
    mor[Support] = First /@ rules;
    (mor[w_LW] /; Deg[w] == 1) := (mor[w] = MakeLieSeries[w /. rules]);
    mor[w_LW] := (mor[w] = b @@ (mor /@ LyndonFactorization[w]));
    mor[AW[]] = MakeASeries[AW[]];
    mor[AW[w_]] := mor[AW[w]] = ι[MakeLieSeries[LW[w] /. rules]];
    mor[w_AW] := mor[w] = Module[{w1, w2},
       w1 = Take[w, Floor[Length[w] / 2]];
       w2 = Drop[w, Floor[Length[w] / 2]];
       mor[w1] ** mor[w2]
      ];
    mor[w_CW] := tr[mor[AW @@ w]];
    mor[s_LieSeries] := mor[s] = New[LieSeries[ser],
       ser[d_] := ser[d] = Sum[mor[s[k]][d], {k, 1, d}]];
    mor[cws_CWSeries] := mor[cws] = New[CWSeries[ser],
       ser[d_] := ser[d] = Sum[mor[cws[k]][d], {k, 1, d}]];
    mor[expr_][d_] := Expand[expr /. (w_LW | w_AW | w_CW) :> mor[w][d]];
    (* Added 150217, commented out later that day *)
    (*mor[expr_] := Expand[expr /. (w_LW|w_AW|w_CW) :> mor[w]]*)
   ];
```

```
LieMorphism /: Inverse[mor_LieMorphism] := InvertLieMorphism[mor];
InvertLieMorphism[mor_LieMorphism] := InvertLieMorphism[mor] =
  LieMorphism[Table[
    LW@u → New[LieSeries[uimg],
      uimg[1] = LW@u;
      uimg[d_Integer] /; d > 1 := uimg[d] = -Sum[(mor[uimg[k]][d]), {k, 1, d-1}]
     ],
    {u, mor[Support]}
   ]]
```

StableApply

```
StableApply[mor_LieMorphism, (type : (LieSeries | ASeries | CWSeries))[s_]] := (
   StableApply[mor, type[s]] = New[type[ser],
     ser[d_] := ser[d] = Nest[mor, type[s], d][d]
    ]
  );
```

BCH

```
BCHBase = New[LieSeries[bch],
   bch[1] = LW@"x" + LW@"y";
   bch[d_Integer] := bch[d] = Expand[Plus[
        adSeries[e^-ad, MakeLieSeries[LW@"y"]][MakeLieSeries[LW@"x"]][d],
        -adSeries[(1 - e^-ad)/ad - 1, LieSeries[bch]][EulerE[LieSeries[bch]]][d]
       ]/d]
  ];
BCH[x_, y_] := LieMorphism[{LW["x"] → x, LW["y"] → y}][BCHBase];
```

AW, ASeries, ι, τ, tr_y

```
Unprotect[NonCommutativeMultiply];
x_ ** 0 = 0; 0 ** y_ = 0;
(c_ * x_AW) ** y_ := Expand[c (x ** y)];
x_ ** (c_ * y_AW) := Expand[c (x ** y)];
x_Plus ** y_ := (# ** y) & /@ x;
x_ ** y_Plus := (x ** #) & /@ y;
Deg[w_AW] := Length[w];
AW[AW[w_]] := AW[w];
AW[w1___] ** AW[w2___] := AW[w1, w2];
b[w_AW, z_AW] := w ** z - z ** w;
```

```
ASeries[ser_Symbol][{dd_Integer}] := AS @@ Table[ser[d], {d, 0, dd}];
ASeries[as_Symbol][es___] := as[es];
Format[s_ASeries, StandardForm] := s[{$SeriesShowDegree}];
MakeASeries[as_CWSeries] := as;
MakeASeries[expr_] := MakeASeries[expr] = New[ASeries[ser],
    ser[d_Integer] := ser[d] = Expand[expr /. w_AW /; Deg[w] ≠ d → 0]
   ];
(s1_ASeries ** s2_ASeries) := (s1 ** s2) = New[ASeries[ser],
    ser[d_Integer] := ser[d] = Sum[s1[k] ** s2[d - k], {k, 0, d}]];
```

```
ι[w_LW] /; Deg[w] == 1 := AW @@ w;
ι[w_LW] := ι[w] = b @@ (ι /@ LyndonFactorization[w]);
ι[expr_] := Expand[expr /. w_LW :→ ι[w]];
ι[ls_LieSeries] := ι[ls] = New[ASeries[as],
    as[0] = 0;
    as[d_] /; d > 0 := as[d] = ι[ls[d]]
   ];
```

```
τ[y_LW, w_LW] /; Deg[y] == 1 := τ[y, w] = Which[
    y === w, AW[],
    Deg[w] === 1, 0,
    True, Module[{w1, w2},
     {w1, w2} = LyndonFactorization[w];
     ι[w1] ** τ[y, w2] - ι[w2] ** τ[y, w1]
    ]
   ];
τ[y_, ls_LieSeries] := τ[y, ls] = New[ASeries[as],
    as[d_] := as[d] = τ[LW[y], ls[d + 1]]
   ];
τ[y_, expr_] := Expand[expr /. w_LW ↦ τ[LW[y], w]];
```

```
tr_y_[expr_] /; Deg[LW@y] == 1 := tr[τ[LW@y, expr]];
```

CW, cw, CWSeries, tr, div

```
RotateToMinimal[l_] := Module[
   {bestl = l, rotatedl = RotateLeft[l]},
   While[rotatedl =!= l,
    bestl = First[Sort[{bestl, rotatedl}]];
    rotatedl = RotateLeft[rotatedl]
   ];
   bestl
  ];
```

```
Deg[w_CW] := Length[w];
Format[w_CW] := TopBracketForm[w];
AllCyclicWords[d_Integer, ab_List] := AllCyclicWords[d, ab] = Union[tr[AllWords[d, ab]]];
CWSeries[cws_Symbol][es___] := cws[es];
CWSeries[ser_Symbol][{dd_Integer}] :=
  Append[TopBracketForm[CWS @@ Table[ser[d], {d, dd}]], "..."];
Format[s_CWSeries, StandardForm] := TopBracketForm[s[{$SeriesShowDegree}]];
CWS[cws_] := MakeCWSeries[cws];
MakeCWSeries[cws_CWSeries] := cws;
MakeCWSeries[expr_] := MakeCWSeries[expr] = New[CWSeries[ser],
    ser[d_Integer] := ser[d] = Expand[
       expr /. w_CW :> If[Deg[w] == d, RotateToMinimal@w, 0]
      ]
   ];
RandomCWSeries[ab_List, opts___Rule] := New[CWSeries[ser],
   Module[
    {rand = Randomizer /. {opts} /. Randomizer → (RandomInteger[{-2 Deg[#]!, 2 Deg[#]!}] / Deg[#]! &)},
    ser[d_Integer] := ser[d] = Plus @@ ((rand[#] * #) & /@ AllCyclicWords[d, ab])
   ]];
s1_CWSeries ≡ s2_CWSeries := New[BooleanSequence[bs],
   bs[0] = True;
   bs[d_Integer] := bs[d - 1] && (s1[d] == s2[d]);
  ];
AddCWSeries[ss___CWSeries] := AddCWSeries[ss] = New[CWSeries[ser],
    ser[d_Integer] := ser[d] = Plus @@ ((#[d]) & /@ {ss})
   ];
CWSeries /: Plus[ss__CWSeries] := AddCWSeries[ss];
ScaleCWSeries[c_, s_CWSeries] := ScaleCWSeries[c, s] = New[CWSeries[ser],
    ser[d_Integer] := ser[d] = Expand[c * s[d]]
   ];
CWSeries /: c_ * s_CWSeries := ScaleCWSeries[c, s];
IntegrateCWSeries[cws_CWSeries, {s_, s0_, s1_}] :=
  IntegrateCWSeries[cws, {s, s0, s1}] = New[CWSeries[ser],
    ser[d_Integer] := ser[d] = Expand[∫_s0^s1 cws[d] ⅆs]];
CWSeries /: Integrate[cws_CWSeries, {s_, s0_, s1_}] := IntegrateCWSeries[cws, {s, s0, s1}];
```

```
cw[gs___] := CW @@ Replace[RotateToMinimal@{gs}, LW[g_] :> g, {1}];
```

```
tr[w_AW] := tr[w] = cw @@ w;
tr[expr_] := expr /. aw_AW :> tr[aw];
tr[as_ASeries] := tr[as] = New[CWSeries[cws], cws[d_] := cws[d] = tr[as[d]]];
```

```
div[LW[y_], w_LW] := div[LW@y, w] = tr[(AW@y) ** τ[LW@y, w]];
div[y_, ls_LieSeries] := div[y, ls] = New[CWSeries[cws],
    cws[d_] := cws[d] = div[LW[y], ls[d]]
   ];
div[y_, expr_] := Expand[expr /. w_LW :> div[LW[y], w]];
div[y_, p_Plus] := Total[div[LW[y], List @@ p]];
div_y_[expr_] := div[y, expr];
```

The Meta-Cocycle JA

```
JA[-1, ___] = MakeCWSeries[0];
JA[n_, y_LW, μ_LieSeries, ss_] := JA[n, y, μ, ss] = Module[
    {s, sμ, μs},
    sμ = ScaleLieSeries[s, μ];
    μs = StableApply[LieMorphism[{y → Ad[ScaleLieSeries[1, sμ]][LW[z]]}], μ];
    μs = μs // LieMorphism[{LW[z] → y}];
    IntegrateCWSeries[
     AddCWSeries[
      JA[n - 1, y, μ, s] // LieDerivation[{y → b[μs, y]}],
      div[y, μs]
     ],
     {s, 0, ss}
    ]
   ];
JA[y_LW, μ_LieSeries] := JA[y, μ] = Module[{s}, New[CWSeries[cws],
     cws[d_Integer] := cws[d] = JA[d - 1, y, μ, s][d] /. s → 1
    ]];
```

⟨…⟩

```
SetAttributes[AngleBracket, Orderless];
⟨{λ___}⟩ := ⟨λ⟩;
(*⟨x_ → 0, rest___⟩ := ⟨x→LS[0], rest⟩;*)
Support[⟨λ___⟩] := First /@ {λ};
λ_ \ key_ := DeleteCases[λ, key → _];
λ_ \ keys_List := Fold[#1 \ #2 &, λ, keys];
⟨λ___⟩_s_ := s /. {λ};
(λ_AngleBracket)[d_] := ⟨Table[u → λ_u[d], {u, Support[λ]}]⟩;
AngleBracket /: EulerE[λ_AngleBracket] := MapAt[EulerE, λ, {All, 2}];
Crop[λ_AngleBracket, d_] := MapAt[Crop[#, d] &, λ, {All, 2}];
AngleBracket /: λ1_AngleBracket ≡ λ2_AngleBracket /; Support[λ1] == Support[λ2] :=
  And @@ Table[λ1_s ≡ λ2_s, {s, Support[λ1]}];
AngleBracket /: Plus[λs__AngleBracket] := ⟨Table[
    u → Total[#_u & /@ {λs}],
    {u, Union @@ (Support[#] & /@ {λs})}
   ]⟩;
AngleBracket /: c_ * λ_AngleBracket := ⟨Table[
    u → Expand[c λ_u],
    {u, Support[λ]}
   ]⟩;
AngleBracket /: λ_AngleBracket // der_LieDerivation := MapAt[der, λ, {All, 2}];
AngleBracket /: λ_AngleBracket // DerivationSeries[f_, ld_LieDerivation] :=
  MapAt[DerivationSeries[f, ld], λ, {All, 2}];
AngleBracket /: λ_AngleBracket // mor_LieMorphism := MapAt[mor, λ, {All, 2}];
AngleBracket /: Integrate[λ_AngleBracket, {s_, s0_, s1_}] :=
  ⟨Table[u → Integrate[λ_u, {s, s0, s1}], {u, Support[λ]}]⟩;
λ_AngleBracket // DegreeScale[h_] := MapAt[DegreeScale[h], λ, {All, 2}];
b[λ1_AngleBracket, λ2_AngleBracket] := ⟨Table[s → b[λ1_s, λ2_s], {s, Support[λ1] ⋃ Support[λ2]}]⟩;
AngleBracket /: D[λ_AngleBracket, vars__] := ⟨Table[s → D[λ_s, vars], {s, Support[λ]}]⟩;
```

Tangential Derivations D_λ, the tangential bracket tb

```
AngleBracket /: D[λ_AngleBracket] := LieDerivation[List @@ λ /. (s_ → λs_) ⧴ (LW[s] → b[LW[s], λs])];
AngleBracket /: D_{λ_AngleBracket} := D[λ];
```

```
tb[λ1_AngleBracket, λ2_AngleBracket] := ⟨Table[
    s → b[λ1_s, λ2_s] + D_λ1[λ2_s] - D_λ2[λ1_s],
    {s, Support[λ1] ⋃ Support[λ2]}
   ]⟩;
```

CC, RC, ad_u

```
AngleBracket /: CC[⟨λ___⟩] := LieMorphism[{λ} /. (s_ → λs_) ⧴ (LW[s] → Ad[λs][LW[s]])];
CC[u_, γ_] := CC[⟨u → γ⟩];
CC_u_[γ_] := CC[u, γ];
RC[λ_AngleBracket] := Inverse[CC[-λ]];
RC[u_, γ_] := RC[⟨u → γ⟩];
RC_u_[γ_] := RC[u, γ];
ad[u_, γ_LieSeries] := LieDerivation[u → b[γ, u]];
ad_u_[γ_] := ad[u, γ];
```

J, div, j

JDef

```
J_u_[γ_] := J_u[γ] = Module[{s}, ∫_0^1 (γ // RC_u[s γ] // div_u // CC_u[-s γ]) ⅆs];
```

```
J[u_, γ_] := J_u[γ];
```

divDef

```
div[λ_AngleBracket] := Sum[div_a[λ_a], {a, Support[λ]}];
j[λ_AngleBracket] := div[λ] // DerivationSeries[(ⅇ^der - 1)/der, D_λ];
```

Evaluating Lie Series in ⟨…⟩

```
LieMorphism[rules__Rule, keys_AngleBracket, br_] := LieMorphism[{rules}, keys, br];
LieMorphism[rules_List, keys_AngleBracket, br_] := New[LieMorphism[mor],
   mor[Support] = First /@ rules;
   mor[0] := mor[0] = AngleBracket @@ Table[s → LS[0], {s, List @@ keys}];
   (mor[w_LW] /; Deg[w] == 1) := (mor[w] = w /. rules);
   mor[w_LW] := (mor[w] = br @@ (mor /@ LyndonFactorization[w]));
   mor[expr_][d_] := Expand[expr /. w_LW :→ mor[w][d]];
   mor[ls_LieSeries] := mor[ls] = Module[{ser},
      ⟨Table[
        ReleaseHold[Hold[
           ser[] = Hold[AngleBracketFromLieMorphism[mor, ls, ss]];
           ser[d_Integer] := ser[d] = (Sum[mor[ls[k]][d], {k, 1, d}])_ss ;
           ss → LieSeries[ser]
          ] /. {ss → s, ser → Unique[LieSeriesInAngleBracket]}],
        {s, List @@ keys}
       ]⟩]
  ];
```

```
BCH[x_, y_, keys_AngleBracket, br_] := LieMorphism[{LW["x"] → x, LW["y"] → y}, keys, br][BCHBase];
BCH_br_[λ1_AngleBracket, λ2_AngleBracket] := BCH[λ1, λ2, ⟨Support[λ1] ⋃ Support[λ2]⟩, br];
adSeries[f_, λ1_AngleBracket, br_][λ2_AngleBracket] :=
  LieMorphism[{LW["x"] → λ1, LW["y"] → λ2}, ⟨Support[λ1] ⋃ Support[λ2]⟩, br][
   adSeries[f, LW["x"]][LW["y"]]];
adSeries[f_, λ1_AngleBracket][λ2_AngleBracket] := adSeries[f, λ1, b][λ2];
```

Γ and Λ

```
Γ_{0,t_}[λ_AngleBracket] := t λ;
Γ_{n_,t_}[λ_AngleBracket] := Γ_{n,t}[λ] = ∫_0^t (λ // ⅇ^(-τ D_λ) // adSeries[ad/(ⅇ^ad - 1), Γ_{n-1,τ}[λ]]) ⅆτ;
Γ_{t_}[λ_AngleBracket] := Γ_t[λ] = 〈Table[
      s → New[LieSeries[ser],
        With[{s = s}, ser[d_Integer] := ser[d] = (Γ_{d-1,t}[λ]_s)[d]];
       ],
      {s, Support[λ]}
     ]〉;
Γ[λ_AngleBracket] := Γ_1[λ];
(*Γ_{-1,_}[λ_AngleBracket] := 〈Table[s→MakeLieSeries[0], {s, Support[λ]}]〉;*)
(*Γ_{n_,t_}[λ_AngleBracket] := Γ_{n,t}[λ] = Module[{τ},
     ∫_0^t (λ // ⅇ^(-τ D_λ) // adSeries[ad/(ⅇ^ad-1), Γ_{n-1,τ}[λ]]) ⅆτ
    ];*)
(*Γ_{n_,t_}[λ_AngleBracket] := Γ_{n,t}[λ] = Module[{τ,Γ0},
     Γ0=Γ_{n-1,τ}[λ];
     〈Table[
       s→∫_0^t (λ_s // ⅇ^(-τ D_λ) // adSeries[ad/(ⅇ^ad-1), Γ0_s]) ⅆτ,
       {s, Support[λ]}
      ]〉
    ];*)
```

```
Λ_{0,t_}[λ_AngleBracket] := t λ;
Λ_{n_,t_}[λ_AngleBracket] := Λ_{n,t}[λ] = Module[{τ, Λ0},
     Λ0 = Λ_{n-1,τ}[λ];
     ∫_0^t (λ // ⅇ^(D_Λ0) // adSeries[ad/(ⅇ^ad - 1), Λ0, tb]) ⅆτ
    ];
Λ_{t_}[λ_AngleBracket] := Λ_t[λ] = 〈Table[
      s → New[LieSeries[ser],
        With[{s = s}, ser[d_Integer] := ser[d] = (Λ_{d-1,t}[λ]_s)[d]];
       ],
      {s, Support[λ]}
     ]〉;
Λ[λ_AngleBracket] := Λ_1[λ];
```

SeriesSolve

```
LS[ab_List, coefs_] := New[LieSeries[ser],
   ser[setter] = Null;
   ser[d_Integer, UndeterminedCoefficients] := Cases[coefs @@@ AllLyndonWords[d, ab], _coefs];
   ser[d_Integer] := If[ser[setter] =!= Null,
     ser[setter][d]; ser[d],
     Plus @@ ((#*coefs @@ #) & /@ AllLyndonWords[d, ab])
    ];
  ];
CWS[ab_List, coefs_] := New[CWSeries[ser],
   ser[setter] = Null;
   ser[d_Integer, UndeterminedCoefficients] := Cases[coefs @@@ AllCyclicWords[d, ab], _coefs];
   ser[d_Integer] := If[ser[setter] =!= Null,
     ser[setter][d]; ser[d],
     Plus @@ ((#*coefs @@ #) & /@ AllCyclicWords[d, ab])
    ];
  ];
```

```
Options[SeriesSolve] = {
   Arbitrator → 0 (* should be 0 or a pure function that takes a
    list of unsettled variables and returns a list of their arbitrated values *)
  };
SeriesSolve::ArbitrarilySetting = "In degree `1` arbitrarily setting `2`.";
SeriesSolve::NoSolution = "No solution in degree `1`.";
SeriesSolve[unknown_, eqns_, opts___] /; Head[unknown] =!= List :=
  SeriesSolve[{unknown}, eqns, opts];
SeriesSolve[unknowns_List, eqns_, opts___] := Module[{
    arbitrator = Arbitrator /. {opts} /. Options[SeriesSolve],
    msgs, MessageStream, solver, lineqs, gens, vars, sol, fvars, avals, d
   },
   If[arbitrator === 0, arbitrator = Replace[#, _ → 0, 1] &];
   msgs = {};
   MessageStream /: Read[MessageStream] := msgs;
   solver[0] = Null;
   solver[n_] := (
     solver[n - 1];
     (#[setter] = Null) & /@ (First /@ unknowns);
     lineqs = eqns[n];
     lineqs = If[Head[lineqs] === And, List @@ lineqs, List@lineqs];
     gens = Union[Cases[lineqs, _LW | _CW, ∞]];
     lineqs = Flatten[Replace[lineqs ,
        lhs_ == rhs_ :→ ((Coefficient[lhs, #] == Coefficient[rhs, #]) & /@ gens),
        {1}]];
     vars = Union @@ ((#[n, UndeterminedCoefficients]) & /@ unknowns);
     sol = Quiet[Solve[lineqs, vars], {Solve::svars}];
     If[sol === {},
      AppendTo[msgs, {"NoSolution", n}]; Message[SeriesSolve::NoSolution, n]; Abort[],
      (* Else *) sol = sol[[1]];
      fvars = Complement[vars, First /@ sol];
      avals = arbitrator[fvars];
      AppendTo[msgs, {"ArbitrarilySetting", n, Thread[(Hold /@ fvars) → avals]}];
      If[fvars =!= {}, Message[SeriesSolve::ArbitrarilySetting, n, Thread[fvars → avals]]];
      MapThread[(#1 = #2) &, {fvars, avals}];
      sol /. (Rule → Set);
      (#[n] = #[n]) & /@ (First /@ unknowns);
      (#[setter] = solver) & /@ (First /@ unknowns);
      solver[n] = Null
     ]
    );
   (#[setter] = solver) & /@ (First /@ unknowns);
   MessageStream
  ];
```

DK (Drinfel’d-Kohno Elements)

```
DK[_, 0] = 0;
DK /: DK[k_, x1_] + DK[k_, x2_] := DK[k, x1 + x2];
DK /: c_ * DK[k_, x_] := DK[k, Expand[c x]];
DK /: b[DK[k1_, c_ * x1_LW], DK[k2_, x2_]] := Expand[c b[DK[k1, x1], DK[k2, x2]]];
DK /: b[DK[k1_, x1_LW], DK[k2_, c_ * x2_LW]] := Expand[c b[DK[k1, x1], DK[k2, x2]]];
DK /: b[DK[k1_, x1_Plus], DK[k2_, x2_]] := b[DK[k1, #], DK[k2, x2]] & /@ x1;
DK /: b[DK[k1_, x1_], DK[k2_, x2_Plus]] := b[DK[k1, x1], DK[k2, #]] & /@ x2;
DK /: b[DK[k_, x1_], DK[k_, x2_]] := DK[k, b[x1, x2]];
DK /: b[DK[k1_, x1_], DK[k2_, x2_]] /; k1 > k2 := b[DK[k2, Expand[-x2]], DK[k1, x1]];
DK /: b[DK[k1_, LW@i1_], DK[k2_, LW@i2_]] /; k1 < k2 := b[DK[k1, LW@i1], DK[k2, LW@i2]] = Which[
    i1 == i2, DK[k2, -b[LW@k1, LW@i2]],
    k1 == i2, DK[k2, -b[LW@i1, LW@i2]],
    True, 0
   ];
DK /: b[DK[k1_, w1_LW], DK[k2_, w2_LW]] /; Deg[w1] > 1 := b[DK[k1, w1], DK[k2, w2]] = Module[{x, y},
    {x, y} = LyndonFactorization[w1];
    b[b[DK[k1, x], DK[k2, w2]], DK[k1, y]] + b[DK[k1, x], b[DK[k1, y], DK[k2, w2]]]
   ];
DK /: b[DK[k1_, w1_LW], DK[k2_, w2_LW]] /; Deg[w2] > 1 := b[DK[k1, w1], DK[k2, w2]] = Module[{x, y},
    {x, y} = LyndonFactorization[w2];
    b[b[DK[k1, w1], DK[k2, x]], DK[k2, y]] + b[DK[k2, x], b[DK[k1, w1], DK[k2, y]]]
   ];
t[i_, j_] := DK[Max[i, j], LW@Min[i, j]];
TopBracketForm[DK[k_, x_]] := x // LieMorphism[Table[LW@i → LW@t_{10 i+k}, {i, k - 1}]];
(*Format[dk_DK] := TopBracketForm[dk];*)
```

σ

```
σ /: expr_^{s_σ} := expr // s;
σ[lft___, i_Integer, rgt___] := σ[lft, IntegerDigits[i], rgt];
_σ[0] = 0;
x_Plus // s_σ := s[#] & /@ x;
DK[k_, x_Plus] // s_σ := s[DK[k, #]] & /@ x;
DK[k_, c_ * w_LW] // s_σ := Expand[c * s[DK[k, w]]];
DK[k_, LW@i_] // s_σ := Sum[t[α, β], {α, s[[i]]}, {β, s[[k]]}];
DK[k_, w_LW] // s_σ := b @@ (s[DK[k, #]] & /@ LyndonFactorization[w]);
```

DKSeries

```
DKSeries[ser_Symbol][{dd_Integer}] := TopBracketForm[Append[
    DKS @@ Table[
      ser[d] /. DK[k_, x_] :→ (x // LieMorphism[Table[LW@i → LW@t_{10 i+k}, {i, k - 1}]]) @d,
      {d, dd}
     ],
    "..."]];
Format[dks_DKSeries, StandardForm] := dks[{$SeriesShowDegree}];
DKSeries[ser_Symbol][e___] := ser[e];
b[dk1_DKSeries, dk2_DKSeries] := b[dk1, dk2] = New[DKSeries[ser],
    ser[d_Integer] := ser[d] = Sum_{j=1}^{d-1} b[dk1[j], dk2[d - j]]
   ];
DKS[dks_DKSeries] := dks;
DKS[expr_] := DKS[expr] = New[DKSeries[ser],
    ser[d_Integer] := ser[d] = Expand[expr /. w_LW /; Deg[w] ≠ d → 0]
   ];
DKS[k_Integer, coefs_] := New[DKSeries[ser],
   ser[setter] = Null;
   ser[d_Integer, UndeterminedCoefficients] := Cases[
     Join @@ Table[coefs[j, Sequence @@ #] & /@ AllLyndonWords[d, Range[j - 1]], {j, 2, k}],
     _coefs];
   ser[d_Integer] := If[ser[setter] =!= Null,
     ser[setter][d]; ser[d],
     Sum[
      Plus @@ ((DK[j, #] * coefs[j, Sequence @@ #]) & /@ AllLyndonWords[d, Range[j - 1]]),
      {j, 2, k}
     ]
    ];
  ];
AddDKSeries[ss___DKSeries] := AddDKSeries[ss] = New[DKSeries[ser],
    ser[d_Integer] := ser[d] = Plus @@ ((#[d]) & /@ {ss})
   ];
DKSeries /: Plus[ss__DKSeries] := AddDKSeries[ss];
ScaleDKSeries[c_, s_DKSeries] := ScaleDKSeries[c, s] = New[DKSeries[ser],
    ser[d_Integer] := ser[d] = Expand[c * s[d]]
   ];
DKSeries /: c_ * s_DKSeries := ScaleDKSeries[c, s];
s1_DKSeries ≡ s2_DKSeries := New[BooleanSequence[bs],
   bs[0] = True;
   bs[d_Integer] := bs[d - 1] &&
     (And @@ Cases[{Expand[s1[d] - s2[d]]}, DK[_, x_] :→ (x == 0), ∞]);
  ];
dks_DKSeries // s_σ := dks // s = New[DKSeries[ser],
    ser[d_Integer] := ser[d] = dks[d] // s
   ];
(dks_DKSeries)_{k_Integer} := New[LieSeries[ser],
   ser[d_Integer] := ser[d] = dks[d] /. {DK[k, x_] :→ x, DK[__] → 0}
  ];
```

Morphisms LS → DKS

```
Morphism[mor_][es___] := mor[es];
Morphism[LS, DKS, rules__Rule] := Morphism[LS, DKS, {rules}];
Morphism[LS, DKS, rules_List] := New[Morphism[mor],
   mor[Support] = First /@ rules;
   (mor[w_LW] /; Deg[w] == 1) := (mor[w] = w /. rules);
   mor[w_LW] := (mor[w] = b @@ (mor /@ LyndonFactorization[w]));
   mor[expr_][d_] := Expand[expr /. w_LW :→ mor[w][d]];
   mor[ls_LieSeries] := mor[ls] = New[DKSeries[ser],
      ser[d_] := ser[d] = Sum[mor[ls[k]][d], {k, 1, d}]];
  ];
BCH[x_DKSeries, y_DKSeries] := BCH[LW@"x", LW@"y"] // Morphism[LS, DKS, LW@"x" → x, LW@"y" → y];
DKSeries /: x_DKSeries ** y_DKSeries := BCH[x, y];
```

The α Map

```
αMap[S_List][dks_DKSeries] := Sum[
   LieMorphism[
     Table[
      LW@S[[i]] → AngleBracket @@ Table[
         S[[j]] → LS[Switch[j,
            i, LW@S[[k]],
            k, LW@S[[i]],
            _, 0
           ]],
         {j, Length[S]}],
      {i, k - 1}
     ],
     AngleBracket @@ S, tb
    ][dks_{S[[k]]}],
   {k, 2, Length[S]}
  ];
αMap[S___][dks_DKSeries] := αMap[{S}][dks];
```

Epilog

```
End[]; EndPackage[];
```