Free Lie Algebras Routines

Lazy Evaluation Version

Pensieve header: A free-Lie calculator with lazy evaluation for series; pensieve://2014-01/ version, continues pensieve://2013-05/, continued pensieve://Projects/WKO4/.

Global Definitions

```
$SeriesShowDegree = 3; $SeriesCompareDegree = 3;
```

Words and Lyndon Words

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

```
AllWords[n_, ab_List] := AW /@ StringJoin @@@ Tuples[ab, n];
LyndonQ[AW[w_String]] := And @@ (
    OrderedQ[{w, #}] & /@ Table[StringDrop[w, i], {i, 1, StringLength[w] - 1}]);
AllLyndonWords[n_Integer, ab_List] := AllLyndonWords[n, ab] =
   LW @@@ Select[AllWords[n, ab /. LW[w_String] :> w], LyndonQ];
AllLyndonWords[{n_}, ab_List] := Join @@ Table[AllLyndonWords[k, ab], {k, n}];
Deg[LW[x_]] := StringLength[x];
LyndonFactorization[w_LW /; Deg[w] == 1] := w;
LyndonFactorization[LW[w_String] /; Deg[LW@w] > 1] := Module[{rf},
   rf = First[Sort[Table[StringDrop[w, i], {i, 1, StringLength[w] - 1}]]];
   LW /@ {StringDrop[w, -StringLength[rf]], rf}];
LW[s_Symbol] := LW[ToString[s]];
LW[LW[w_]] := LW[w];
LW /: LW[x_] ≤ LW[y_] := OrderedQ[{x, y}];
LW /: x_LW ≥ y_LW := y ≤ x; LW /: x_LW > y_LW := ! (x ≤ y);
LW /: x_LW < y_LW := ! (y ≤ x);
Format[LW[w_], StandardForm] := Defer[⟨w⟩];
BracketForm[w_LW] /; Deg[w] == 1 := w[[1]];
BracketForm[w_LW] := BracketForm[w] = StringJoin[Flatten[{
      "[", BracketForm /@ LyndonFactorization[w], "]"
     }]];
topbracketform[w_LW] /; Deg[w] == 1 := w[[1]];
topbracketform[w_LW] := topbracketform[w] = Overscript[
    Row[Riffle[topbracketform /@ LyndonFactorization[w], " "]], ⟌];
TopBracketForm[w_LW] /; Deg[w] == 1 := Overscript[w[[1]], ⟌];
TopBracketForm[w_LW] := topbracketform[w];
TopBracketForm[CW[w_String]] := Overscript[w, ⏜];
TopBracketForm[expr_] := expr /. w_LW | w_CW :> TopBracketForm[w];
⟨w__⟩ := LW[w];
LW[is__Integer] := LW[StringJoin @@
    (StringTake["0123456789abcdefghijklmnopqrstuvwxyz", {#}] & /@ (1 + {is}))];
```

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, LW[First[w] <> First[z]],
       True, Module[{x, y},
        {x, y} = LyndonFactorization[w];
        If[y ≥ z,
         LW[First[w] <> First[z]],
         b[x, LWAdjoint[y][z]] + b[LWAdjoint[x][z], y]
        ]
       ]
      ];
    u
   ];
```

LieSeries

```
LieSeries[ser_Symbol][{dd_Integer}] :=
  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}];
MakeLieSeries[s_LieSeries] := s;
MakeLieSeries[expr_] :=
  MakeLieSeries[expr] = MakeLieSeries[Unique[MakeLieSeries], expr];
MakeLieSeries[ser_Symbol, expr_] := (
   ser[] = Hold[MakeLieSeries[ser, expr]];
   ser[d_Integer] := ser[d] = Expand[expr /. w_LW /; Deg[w] ≠ d → 0];
   LieSeries[ser]
  );
MakeLieSeries[ab_List, coefs_] :=
  MakeLieSeries[Unique[MakeLieSeries], ab, coefs];
MakeLieSeries[ser_Symbol, ab_List, coefs_Symbol] := (
   ser[] = Hold[MakeLieSeries[a, coefs]];
   ser[d_Integer] := ser[d] =
     Plus @@ MapIndexed[((coefs @@ Prepend[#2, d]) * #1) &, AllLyndonWords[d, ab]];
   LieSeries[ser]
  );
s1_LieSeries ≡ s2_LieSeries := Module[{res = True, k},
   For[k = 1, res && k <= $SeriesCompareDegree, ++k, res = res && (s1[k] == s2[k])];
   res
  ];
RandomLieSeries[ab_List, opts___Rule] :=
  RandomLieSeries[Unique[RandomLieSeries], ab, opts];
RandomLieSeries[ser_Symbol, ab_List, opts___Rule] := Module[
   {rand = Randomizer /. {opts} /.
      Randomizer → (RandomInteger[{-2 Deg[#]!, 2 Deg[#]!}] / Deg[#]! &)},
   ser[] = Hold[RandomLieSeries[a, opts]];
   ser[d_Integer] := ser[d] = Plus @@ ((rand[#] * #) & /@ AllLyndonWords[d, ab]);
   LieSeries[ser]
  ];
```

```
AddLieSeries[ss___LieSeries] := AddLieSeries[ss] = Module[{ser},
    ser = Unique[AddLieSeries];
    ser[] = Hold[AddLieSeries[ss]];
    ser[d_Integer] := ser[d] = Plus @@ ((#[d]) & /@ {ss});
    LieSeries[ser]
   ];
LieSeries /: Plus[ss__LieSeries] := AddLieSeries[ss];
ScaleLieSeries[c_, s_LieSeries] := ScaleLieSeries[c, s] = Module[{ser},
    ser = Unique[ScaleLieSeries];
    ser[] = Hold[ScaleLieSeries[c, s]];
    ser[d_Integer] := ser[d] = Expand[c * s[d]];
    LieSeries[ser]
   ];
LieSeries /: c_ * s_LieSeries := ScaleLieSeries[c, s];

b[s1_LieSeries, s2_LieSeries] := b[s1, s2] = Module[{ser},
    ser = Unique[b];
    ser[] = Hold[b[s1, s2]];
    ser[d_Integer] := ser[d] = Sum[
       b[s1[k], s2[d - k]],
       {k, 1, d - 1}
      ];
    LieSeries[ser]
   ];
b[s_LieSeries, y_] := b[s, MakeLieSeries[y]];
b[x_, s_LieSeries] := b[MakeLieSeries[x], s];

LieSeries /: EulerE[s_LieSeries] := Module[{ser},
   ser = Unique[EulerE];
   ser[] = Hold[EulerE[s]];
   ser[d_Integer] := ser[d] = Expand[d * s[d]];
   LieSeries[ser]
  ];
```

adPower, adSeries, and Ad

```
adPower[0, x_LieSeries][ψ_LieSeries] := adPower[0, x][ψ] = Module[{ser},
    ser = Unique[adPower];
    ser[] = Hold[adPower[0, x][ψ]];
    ser[d_Integer] := ser[d] = ψ[d];
    LieSeries[ser]
   ];
adPower[n_Integer, x_LieSeries][ψ_LieSeries] := adPower[n, x][ψ] = Module[{ser},
    ser = Unique[adPower];
    ser[] = Hold[adPower[n, x][ψ]];
    ser[d_Integer] := ser[d] = b[x, adPower[n - 1, x][ψ]][d];
    LieSeries[ser]
   ];
adSeries[f_, x_LieSeries][ψ_LieSeries] := adSeries[f, x][ψ] = Module[{ser},
    ser = Unique[adSeries];
    ser[] = Hold[adSeries[f, x][ψ]];
    ser[d_Integer] := ser[d] = Module[{c},
       Expand[Sum[
         c = SeriesCoefficient[f, {ad, 0, k}];
         If[c === 0, 0, c * adPower[k, x][ψ][d]],
         {k, 0, d - 1}
        ]]
      ];
    LieSeries[ser]
   ];
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] = LieDerivation[Unique[LieDerivation], rules];
LieDerivation[der_Symbol, rules_List] := (
   der[] = Hold[LieDerivation[der, rules]];
   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] = Module[{ser},
      ser = Unique[LieDerivationOnLieSeries];
      ser[] = Hold[der[s]];
```

```
      ser[d_] := ser[d] = Sum[
         der[s[k]][d],
         {k, 1, d}
        ];
      LieSeries[ser]
     ];
   der[as_ASeries] := der[as] = Module[{ser},
      ser = Unique[LieDerivationOnASeries];
      ser[] = Hold[der[as]];
      ser[d_] := ser[d] = Sum[
         Expand[as[k] /. AW[w_] :> Sum[
              NonCommutativeMultiply[
               AW[StringTake[w, j - 1]],
               ι[der[LW[StringTake[w, {j}]]][d - k + 1]],
               AW[StringDrop[w, j]]
              ],
              {j, k}
             ]],
         {k, 1, d}
        ];
      ASeries[ser]
     ];
   der[cws_CWSeries] := der[cws] = Module[{ser},
      ser = Unique[LieDerivationOnCWSeries];
      ser[] = Hold[der[cws]];
      ser[d_] := ser[d] = Sum[
         Expand[cws[k] /. CW[w_] :> Sum[
              tr[NonCommutativeMultiply[
                AW[StringTake[w, j - 1]],
                ι[der[LW[StringTake[w, {j}]]][d - k + 1]],
                AW[StringDrop[w, j]]
               ]],
              {j, k}
             ]],
         {k, 1, d}
        ];
      CWSeries[ser]
     ];
   der[expr_][d_] :=
    Expand[expr /. {w_LW :> der[w][d], s_LieSeries :> der[s][d]}];
   LieDerivation[der]
  );

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

```
DerivationPower[0, der_LieDerivation][ψ_LieSeries] :=
  DerivationPower[0, der][ψ] = Module[{ser},
    ser = Unique[DerivationPower];
    ser[] = Hold[DerivationPower[0, der][ψ]];
    ser[d_Integer] := ser[d] = ψ[d];
    LieSeries[ser]
   ];
DerivationPower[n_Integer, der_LieDerivation][ψ_LieSeries] :=
  DerivationPower[n, x][ψ] = Module[{ser},
    ser = Unique[DerivationPower];
    ser[] = Hold[DerivationPower[n, der][ψ]];
    ser[d_Integer] := ser[d] = der[DerivationPower[n - 1, der][ψ]][d];
    LieSeries[ser]
   ];
DerivationSeries[__][0] = 0;
DerivationSeries[f_, ld_LieDerivation][ψ_LieSeries] :=
  DerivationSeries[f, ld][ψ] = Module[{ser},
    ser = Unique[DerivationSeries];
    ser[] = Hold[DerivationSeries[f, ld][ψ]];
    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}
        ]]
      ];
    LieSeries[ser]
   ];
DerivationExp[ld_LieDerivation] := DerivationSeries[E^der, ld];
```

LieMorphism

```
LieMorphism[mor_][es___] := mor[es];
LieMorphism[rules_List] :=
  LieMorphism[rules] = LieMorphism[Unique[LieMorphism], rules];
LieMorphism[rules__Rule] := LieMorphism[{rules}];
LieMorphism[mor_Symbol, rules_List] := (
   mor[] = Hold[LieMorphism[mor, rules]];
   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_]] /; StringLength[w] == 1) :=
    (mor[AW[w]] = ι[MakeLieSeries[LW[w] /. rules]]);
   mor[AW[w_]] := mor[AW[w]] = Module[{w1, w2},
      w1 = StringTake[w, Floor[StringLength[w] / 2]];
      w2 = StringDrop[w, Floor[StringLength[w] / 2]];
      (mor[AW[w1]]) ** (mor[AW[w2]])
     ];
   mor[CW[w_]] := tr[mor[AW[w]]];
   mor[s_LieSeries] := mor[s] = Module[{ser},
      ser = Unique[LieMorphismOnLieSeries];
      ser[] = Hold[mor[s]];
      ser[d_] := ser[d] = Sum[
         mor[s[k]][d],
         {k, 1, d}
        ];
      LieSeries[ser]
     ];
   mor[cws_CWSeries] := mor[cws] = Module[{ser},
      ser = Unique[LieMorphismOnCWSeries];
      ser[] = Hold[mor[s]];
      ser[d_] := ser[d] = Sum[
         mor[cws[k]][d],
         {k, 1, d}
        ];
      CWSeries[ser]
     ];
   mor[expr_][d_] := Expand[expr /. (w_LW | w_AW | w_CW) :> mor[w][d]];
   LieMorphism[mor]
  );
```

StableApply

```
StableApply[mor_LieMorphism, (type : (LieSeries | ASeries | CWSeries))[s_]] := (
   StableApply[mor, type[s]] = Module[{ser},
     ser = Unique[StableApply];
     ser[] = Hold[StableApply[mor, type[s]]];
     ser[d_] := ser[d] = Nest[mor, type[s], d][d];
     type[ser]
    ]
  );
```

BCH

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

AW, ASeries, ι, σ

```
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[AW[w_]] := StringLength[w];
AW[AW[w_]] := AW[w];
AW[w1_String] ** AW[w2_String] := 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] = MakeASeries[Unique[MakeASeries], expr];
MakeASeries[ser_Symbol, expr_] := (
   ser[] = Hold[MakeASeries[ser, expr]];
   ser[d_Integer] := ser[d] = Expand[expr /. w_AW /; Deg[w] ≠ d → 0];
   ASeries[ser]
  );
(s1_ASeries ** s2_ASeries) := (s1 ** s2) = Module[{ser},
    ser = Unique[NonCommutativeMultiply];
    ser[] = Hold[s1 ** s2];
    ser[d_Integer] := ser[d] = Sum[
       s1[k] ** s2[d - k],
       {k, 0, d}
      ];
    ASeries[ser]
   ];

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] = Module[{as},
    as = Unique[ι];
    as[] = Hold[ι[ls]];
    as[0] = 0;
    as[d_] /; d > 0 := as[d] = ι[ls[d]];
    ASeries[as]
   ];

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

CW, CWSeries, tr, div

```
Deg[CW[w_]] := StringLength[w];
AllCyclicWords[d_Integer, ab_List] :=
  AllCyclicWords[d, ab] = Union[tr[AW[StringJoin@@#] & /@ Tuples[ab, d]]];
CWSeries[cws_Symbol][es___] := cws[es];
CWSeries[ser_Symbol][{dd_Integer}] :=
  TopBracketForm[CWS@@Table[ser[d], {d, dd}]];
Format[s_CWSeries, StandardForm] := TopBracketForm[s[{$SeriesShowDegree}]];
MakeCWSeries[cws_CWSeries] := cws;
MakeCWSeries[expr_] :=
  MakeCWSeries[expr] = MakeCWSeries[Unique[MakeCWSeries], expr];
MakeCWSeries[ser_Symbol, expr_] := (
   ser[] = Hold[MakeCWSeries[ser, expr]];
   ser[d_Integer] := ser[d] = Expand[expr /. w_CW /; Deg[w] ≠ d → 0];
   CWSeries[ser]
  );
MakeCWSeries[ab_List, coefs_] := MakeCWSeries[Unique[MakeCWSeries], ab, coefs];
MakeCWSeries[ser_Symbol, ab_List, coefs_Symbol] := (
   ser[] = Hold[MakeCWSeries[a, coefs]];
   ser[d_Integer] := ser[d] =
     Plus @@ MapIndexed[((coefs@@Prepend[#2, d])*#1) &, AllCyclicWords[d, ab]];
   CWSeries[ser]
  );
RandomCWSeries[ab_List, opts___Rule] :=
  RandomCWSeries[Unique[RandomCWSeries], ab, opts];
RandomCWSeries[ser_Symbol, ab_List, opts___Rule] := Module[
   {rand = Randomizer /. {opts} /.
      Randomizer → (RandomInteger[{-2 Deg[#]!, 2 Deg[#]!}]/Deg[#]! &)},
   ser[] = Hold[RandomCWSeries[a, opts]];
   ser[d_Integer] := ser[d] = Plus @@ ((rand[#]*#) & /@ AllCyclicWords[d, ab]);
   CWSeries[ser]
  ];
s1_CWSeries ≡ s2_CWSeries := Module[{res = True, k},
   For[k = 1, res&&k <= $SeriesCompareDegree, ++k, res = res&&(s1[k] == s2[k])];
   res
  ];
AddCWSeries[ss___CWSeries] := AddCWSeries[ss] = Module[{ser},
    ser = Unique[AddCWSeries];
    ser[] = Hold[AddCWSeries[ss]];
    ser[d_Integer] := ser[d] = Plus @@ ((#[d]) & /@ {ss});
    CWSeries[ser]
   ];
CWSeries /: Plus[ss__CWSeries] := AddCWSeries[ss];
```

```
ScaleCWSeries[c_, s_CWSeries] := ScaleCWSeries[c, s] = Module[{ser},
    ser = Unique[ScaleCWSeries];
    ser[] = Hold[ScaleCWSeries[c, s]];
    ser[d_Integer] := ser[d] = Expand[c * s[d]];
    CWSeries[ser]
   ];
CWSeries /: c_ * s_CWSeries := ScaleCWSeries[c, s];
IntegrateCWSeries[cws_CWSeries, {s_, s0_, s1_}] :=
  IntegrateCWSeries[cws, {s, s0, s1}] = Module[{ser},
    ser = Unique[IntegrateCWSeries];
    ser[] = Hold[IntegrateCWSeries[cws, {s, s0, s1}]];
    ser[d_Integer] := ser[d] = Expand[Integrate[cws[d], {s, s0, s1}]];
    CWSeries[ser]
   ];
CWSeries /: Integrate[cws_CWSeries, {s_, s0_, s1_}] :=
  IntegrateCWSeries[cws, {s, s0, s1}];

tr[w_AW] := tr[w] = CW[RotateToMinimal@@w];
tr[expr_] := expr /. aw_AW :> tr[aw];
tr[as_ASeries] := tr[as] = Module[{cws},
    cws = Unique[tr];
    cws[] = Hold[tr[as]];
    cws[d_] := cws[d] = tr[as[d]];
    CWSeries[cws]
   ];

div[y_LW, w_LW] /; Deg[y] == 1 := div[y, w] = tr[(AW@@y) ** σ[y, w]];
div[y_, ls_LieSeries] := div[y, ls] = Module[{cws},
    cws = Unique[div];
    cws[] = Hold[div[y, ls]];
    cws[d_] := cws[d] = div[LW[y], ls[d]];
    CWSeries[cws]
   ];
div[y_, expr_] := Expand[expr /. w_LW :> div[LW[y], w]];
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[{cws, s},
    cws = Unique[JA];
    cws[] = Hold[JA[y, μ]];
    cws[d_Integer] := cws[d] = JA[d - 1, y, μ, s][d] /. s → 1;
    CWSeries[cws]
   ];
```