Semisimple.jl

Pieter Belmans

May 29, 2026

Contents

Part I

Home

A Julia package for computations with finite-dimensional complex semisimple Lie algebras via their root data: root systems, Weyl groups, weight lattices, and highest-weight representation-theoretic operations. It is heavily optimized and uses Julia's type system to specialize many finite root-data computations.

0.1 Features

- **Dynkin types** — Type-level classification (TypeA{N}, TypeB{N}, ..., TypeG2, products) with text Dynkin diagrams
- **Cartan matrices** — Compile-time @generated Cartan matrices, symmetrisers, bilinear forms
- **Root systems** — Positive roots, coroots, reflection tables (immutable singletons)
- **Weight lattice** — Fundamental weights, Weyl vector, dominance, conjugation
- **Weyl groups** — Reduced words, multiplication via reflection tables, orbits, dimension formula
- **Characters** — Weyl characters (representation ring), Freudenthal formula, Brauer–Klimyk tensor products, Littlewood–Richardson (Type A), Adams operators, symmetric/exterior powers

0.2 Installation

```
using Pkg
Pkg.add(url="https://github.com/HomogeneousTools/Semisimple.jl")
```

0.3 Quick start

```
julia> using Semisimple

julia> ω1 = fundamental_weight(TypeA{3}, 1)
ω1

julia> degree(ω1)   # dimension of the standard representation
4

julia> V = WeylCharacter(ω1);

julia> tensor_product(V, V)   # V(ω1) ⊗ V(ω1) = Sym²V ⊕ □²V
A3(2, 0, 0) + A3(0, 1, 0)

julia> Sym(2, V) + □(2, V) == V * V   # Newton identity
true

julia> length(weyl_orbit(TypeA{3}, ω1))
4
```

0.4 Contents

`Semisimple.Semisimple` – Module.

```
Semisimple
```

Julia package for computations with semisimple Lie algebras over $\mathbb{C}$.

Provides root systems, Weyl groups, and weight-lattice arithmetic for all classical and exceptional Dynkin types (A, B, C, D, E_6, E_7, E_8, F_4, G_2) as well as their direct products.

See the online documentation for usage examples and mathematical background.

Part II

Dynkin types and Cartan matrices

Chapter 1

Dynkin types and Cartan matrices

1.1 Dynkin types

Over $\mathbb{C}$, every finite-dimensional semisimple Lie algebra is classified up to isomorphism by its **Dynkin type**, a combinatorial datum that encodes the root system. Semisimple.jl represents types at the Julia type level — `TypeA{3}`, `TypeB{4}`, etc. — so that rank and root counts are compile-time constants, enabling zero-cost dispatch and `@generated` specialisation.

Simple types

The classical families `TypeA{N}`, `TypeB{N}`, `TypeC{N}`, `TypeD{N}`, and the exceptional types `TypeE{6}`, `TypeE{7}`, `TypeE{8}`, `TypeF4`, `TypeG2`:

```
julia> using Semisimple

julia> rank(TypeA{3})
3

julia> rank(TypeG2)
2

julia> n_positive_roots(TypeA{3})
6

julia> n_positive_roots(TypeE{8})
120
```

`Semisimple.DynkinType` – Type.

```
DynkinType
```

Abstract supertype for finite Dynkin types (simple and semisimple).

Examples

```
julia> using Semisimple
```

```
julia> TypeA{2} <: DynkinType
true
```

Semisimple.SimpleDynkinType – Type.

```
SimpleDynkinType <: DynkinType
```

Abstract supertype for simple (irreducible) finite Dynkin types.

Examples

```
julia> using Semisimple

julia> TypeG2 <: SimpleDynkinType
true
```

Semisimple.TypeA – Type.

```
TypeA{N} <: SimpleDynkinType
```

Dynkin type A_N: the root-system type of $\mathfrak{sl}_{N+1}(\mathbb{C})$ and groups isogenous to SL_{N+1}. Valid for $N \geq 1$.

Examples

```
julia> using Semisimple

julia> rank(TypeA{3})
3
```

Semisimple.TypeB – Type.

```
TypeB{N} <: SimpleDynkinType
```

Dynkin type B_N: the root-system type of $\mathfrak{so}_{2N+1}(\mathbb{C})$ and groups isogenous to Spin_{2N+1} or SO_{2N+1}. Valid for $N \geq 2$.

Examples

```
julia> using Semisimple

julia> rank(TypeB{3})
3
```

Semisimple.TypeC – Type.

```
TypeC{N} <: SimpleDynkinType
```

Dynkin type C_N: the root-system type of $\mathfrak{sp}_{2N}(\mathbb{C})$ and groups isogenous to Sp_{2N}. Valid for $N \geq 2$.

Examples

```
julia> using Semisimple

julia> rank(TypeC{3})
3
```

Semisimple.TypeD – Type.

```
TypeD{N} <: SimpleDynkinType
```

Dynkin type D_N: the root-system type of $\mathfrak{so}_{2N}(\mathbb{C})$ and groups isogenous to Spin_{2N} or SO_{2N}. Valid for $N \geq 3$.

Examples

```
julia> using Semisimple

julia> rank(TypeD{4})
4
```

Semisimple.TypeE – Type.

```
TypeE{N} <: SimpleDynkinType
```

Exceptional Dynkin type E_N for $N \in \{6, 7, 8\}$.

Examples

```
julia> using Semisimple

julia> n_positive_roots(TypeE{6})
36
```

Semisimple.TypeF4 – Type.

```
TypeF4 <: SimpleDynkinType
```

Exceptional Dynkin type F_4.

Examples

```
julia> using Semisimple

julia> rank(TypeF4)
4
```

`Semisimple.TypeG2` – Type.

```
TypeG2 <: SimpleDynkinType
```

Exceptional Dynkin type G_2.

Examples

```
julia> using Semisimple

julia> rank(TypeG2)
2
```

`Semisimple.rank` – Function.

```
rank(::Type{DT}) where DT<:DynkinType -> Int
```

Return the rank (dimension of the Cartan subalgebra) of the Dynkin type `DT`. This is a compile-time constant.

Examples

```
julia> using Semisimple

julia> rank(TypeA{3})
3

julia> rank(TypeE{8})
8
```

`Semisimple.n_positive_roots` – Function.

```
n_positive_roots(::Type{DT}) -> Int
```

Number of positive roots for a simple Dynkin type.

Examples

```
julia> using Semisimple

julia> n_positive_roots(TypeA{3})
6

julia> n_positive_roots(TypeE{8})
120
```

```
n_positive_roots(RS::RootSystem) -> Int
```

Return the number of positive roots.

Examples

```
julia> using Semisimple

julia> n_positive_roots(RootSystem(TypeA{2}))
3
```

Dynkin diagrams

The function `dynkin_diagram` produces a text rendering of the Dynkin diagram following Bourbaki labelling conventions. This includes the correct arrow directions for non-simply-laced types and the branching for types D and E.

The return value is a `DynkinDiagram` object that pretty-prints automatically in the REPL without requiring `println`:

```
julia> dynkin_diagram(TypeA{4})
○———○———○———○
1   2   3   4

julia> dynkin_diagram(TypeG2)
○≡<≡○
1   2
```

For types with branching (D and E), the diagram shows the fork:

```
dynkin_diagram(TypeD{5})
```

```
          ○ 5
          /
○———○———○———○
1   2   3   4
```

```
dynkin_diagram(TypeE{6})
```

```
        ○ 2
        |
○———○———○———○———○
1   3   4   5   6
```

`Semisimple.DynkinDiagram` – Type.

```
DynkinDiagram
```

A pretty-printable wrapper around the text rendering of a Dynkin diagram.

Displaying a `DynkinDiagram` in the REPL renders the diagram automatically without requiring `println`. Call `string` to recover the raw string.

See also: `dynkin_diagram`.

Examples

```
julia> using Semisimple

julia> occursin(Char(10), string(dynkin_diagram(TypeA{2})))
true
```

`Semisimple.dynkin_diagram` – Function.

```
dynkin_diagram(::Type{DT}) -> DynkinDiagram
dynkin_diagram(dt::DynkinType) -> DynkinDiagram
```

Return the Dynkin diagram for the given type as a `DynkinDiagram`, following Bourbaki conventions. The result pretty-prints automatically in the REPL; call `string` to recover the raw multi-line string.

Examples

```
julia> using Semisimple

julia> dynkin_diagram(TypeA{4})
○———○———○———○
1   2   3   4

julia> dynkin_diagram(TypeB{3})
○———○=>=○
1   2   3

julia> dynkin_diagram(TypeG2)
○≡<≡○
1   2
```

Product types

Direct products of simple types are represented by `ProductDynkinType`. The Dynkin diagram shows each component separately:

```
julia> PT = ProductDynkinType{Tuple{TypeA{2}, TypeB{2}}};

julia> rank(PT)
4
```

```
julia> n_components(PT)
2

julia> component_ranks(PT)
(2, 2)

julia> component_offsets(PT)
(0, 2)
```

```
PT = ProductDynkinType{Tuple{TypeA{2}, TypeB{2}}}
dynkin_diagram(PT)
```

```
A2:
○——○
1   2

B2:
○=>=○
1   2
```

`Semisimple.ProductDynkinType` – Type.

```
ProductDynkinType{Ts} <: DynkinType
```

Product of simple Dynkin types, representing a semisimple Lie algebra. `Ts` is a `Tuple` type of `SimpleDynkinType` subtypes.

Examples

```
julia> using Semisimple

julia> ProductDynkinType{Tuple{TypeA{3}, TypeD{5}, TypeE{6}}}()   # A3 × D5 × E6
A3 × D5 × E6
```

`Semisimple.n_components` – Function.

```
n_components(::Type{ProductDynkinType{Ts}}) -> Int
```

Number of simple factors in a product type.

Examples

```
julia> using Semisimple

julia> n_components(ProductDynkinType{Tuple{TypeA{2}, TypeB{2}}})
2
```

`Semisimple.component_type` – Function.

```
component_type(::Type{ProductDynkinType{Ts}}, i::Integer) -> Type
```

Return the `i`-th simple Dynkin type in a product.

Examples

```
julia> using Semisimple

julia> PT = ProductDynkinType{Tuple{TypeA{2}, TypeB{2}}};

julia> component_type(PT, 1)
TypeA{2}

julia> component_type(PT, 2)
TypeB{2}
```

`Semisimple.component_ranks` – Function.

```
component_ranks(::Type{ProductDynkinType{Ts}}) -> Tuple
```

Return a tuple of ranks of the components.

Examples

```
julia> using Semisimple

julia> component_ranks(ProductDynkinType{Tuple{TypeA{2}, TypeB{3}}})
(2, 3)
```

`Semisimple.component_offsets` – Function.

```
component_offsets(::Type{ProductDynkinType{Ts}}) -> Tuple
```

Return a tuple of starting index offsets for each component in the product type. The i-th component occupies indices offset[i]+1 : offset[i]+rank(component_i).

Examples

```
julia> using Semisimple

julia> component_offsets(ProductDynkinType{Tuple{TypeA{2}, TypeB{3}}})
(0, 2)
```

Validation

Invalid ranks are caught at construction time:

```
julia> TypeA{0}()
ERROR: ArgumentError: TypeA{N} requires N ≥ 1, got N=0
[...]

julia> TypeD{2}()
ERROR: ArgumentError: TypeD{N} requires N ≥ 3, got N=2
[...]
```

1.2 Cartan matrices

The Cartan matrix $C_{ij} = \langle \alpha_i^\vee, \alpha_j \rangle$ is computed at compile time via `@generated` functions, returning a `StaticArrays.SMatrix`. The conventions follow Bourbaki. With this convention, column i of C gives the simple root α_i in the fundamental weight basis: $\alpha_i = \sum_j C_{ji} \omega_j$.

```
julia> cartan_matrix(TypeA{3})
3×3 StaticArraysCore.SMatrix{3, 3, Int64, 9} with indices SOneTo(3)×SOneTo(3):
  2  -1   0
 -1   2  -1
  0  -1   2

julia> cartan_matrix(TypeG2)
2×2 StaticArraysCore.SMatrix{2, 2, Int64, 4} with indices SOneTo(2)×SOneTo(2):
  2  -3
 -1   2

julia> cartan_matrix(TypeB{2})
2×2 StaticArraysCore.SMatrix{2, 2, Int64, 4} with indices SOneTo(2)×SOneTo(2):
  2  -1
 -2   2
```

Symmetriser

The vector d such that $\mathrm{diag}(d) \cdot C$ is symmetric. For simply-laced types (A, D, E) all entries are 1:

```
julia> cartan_symmetrizer(TypeA{3})
3-element StaticArraysCore.SVector{3, Int64} with indices SOneTo(3):
 1
 1
 1

julia> cartan_symmetrizer(TypeG2)
2-element StaticArraysCore.SVector{2, Int64} with indices SOneTo(2):
 1
 3

julia> cartan_symmetrizer(TypeB{2})
2-element StaticArraysCore.SVector{2, Int64} with indices SOneTo(2):
 2
 1
```

Bilinear form and inverse

The symmetric bilinear form $B = \mathrm{diag}(d) \cdot C$ and the rational inverse C^{-1}:

```
julia> cartan_bilinear_form(TypeA{3})
3×3 StaticArraysCore.SMatrix{3, 3, Int64, 9} with indices SOneTo(3)×SOneTo(3):
  2  -1   0
 -1   2  -1
  0  -1   2

julia> cartan_matrix_inverse(TypeA{3})
3×3 StaticArraysCore.SMatrix{3, 3, Rational{Int64}, 9} with indices SOneTo(3)×SOneTo(3):
 3//4  1//2  1//4
 1//2   1    1//2
 1//4  1//2  3//4
```

`Semisimple.cartan_matrix` – Function.

```
cartan_matrix(::Type{ProductDynkinType{Ts}})
```

Block-diagonal Cartan matrix for a product of simple types.

Examples

```
julia> using Semisimple, StaticArrays

julia> cartan_matrix(TypeA{2}) == SMatrix{2,2}(2, -1, -1, 2)
true

julia> cartan_matrix(TypeG2) == SMatrix{2,2}(2, -1, -3, 2)
true
```

`Semisimple.cartan_symmetrizer` – Function.

```
cartan_symmetrizer(::Type{DT}) -> SVector
```

Return the symmetrizer `d` such that `diag(d) * C` is symmetric, where `C` is the Cartan matrix of `DT`. Entries are positive integers with gcd 1.

Examples

```
julia> using Semisimple, StaticArrays

julia> cartan_symmetrizer(TypeB{3}) == SVector(2, 2, 1)
true

julia> cartan_symmetrizer(TypeG2) == SVector(1, 3)
true
```

`Semisimple.cartan_bilinear_form` – Function.

```
cartan_bilinear_form(::Type{DT}) -> SMatrix
```

Return the symmetrized Cartan matrix `diag(d) * C`, which is a symmetric positive-definite matrix defining the inner product on the root space.

Examples

```
julia> using Semisimple

julia> cartan_bilinear_form(TypeB{2})
2×2 StaticArraysCore.SMatrix{2, 2, Int64, 4} with indices SOneTo(2)×SOneTo(2):
  4  -2
 -2   2
```

`Semisimple.cartan_matrix_inverse` – Function.

```
cartan_matrix_inverse(::Type{DT}) -> SMatrix{R,R,Rational{Int}}
```

Return the inverse of the Cartan matrix over the rationals.

Examples

```
julia> using Semisimple

julia> cartan_matrix_inverse(TypeA{2})
2×2 StaticArraysCore.SMatrix{2, 2, Rational{Int64}, 4} with indices SOneTo(2)×SOneTo(2):
 2//3  1//3
 1//3  2//3
```

`Semisimple.cartan_determinant` – Function.

```
cartan_determinant(::Type{DT}) -> Int
cartan_determinant(dt::DT) -> Int
```

Compute the determinant of the Cartan matrix of the Dynkin type `DT`.

For semisimple Lie algebras, this determinant equals the **connection index**, which measures the index of the root lattice in the weight lattice.

This is a compile-time constant based on hardcoded values for simple types.

Examples

```
julia> using Semisimple

julia> cartan_determinant(TypeA{3})
```

```
4

julia> cartan_determinant(TypeB{3})
2

julia> cartan_determinant(TypeG2)
1
```

1.3 Dimension

The dimension of the semisimple Lie algebra equals $r + 2n$ where r is the rank and n is the number of positive roots:

```
julia> dimension(TypeA{3})   # dim sl_4(C) = 15
15

julia> dimension(TypeE{8})   # dim(E_8) = 248
248

julia> dimension(ProductDynkinType{Tuple{TypeA{1}, TypeA{1}}}())   # dim(sl_2(C) + sl_2(C)) = 6
6
```

`Semisimple.dimension` – Function.

```
dimension(::Type{DT}) -> Int
dimension(dt::DynkinType) -> Int
```

Return the dimension of the semisimple Lie algebra of type `DT`.

For a semisimple Lie algebra of rank r with n positive roots, the dimension is $r + 2n$ (Cartan subalgebra plus positive and negative root spaces).

Note: for the adjoint representation dimension (same number), use `degree` on `adjoint_representation`.

Examples

```
julia> using Semisimple

julia> dimension(TypeA{3})  # sl_4(C) has dimension 15
15

julia> dimension(TypeE{8})  # e_8(C) has dimension 248
248

julia> dimension(ProductDynkinType{Tuple{TypeA{1}, TypeA{1}}}())  # sl_2(C) ⊕ sl_2(C)
6
```

Scaled bilinear form

The scaled bilinear form on the fundamental weight basis, used internally by Freudenthal's formula:

`Semisimple.omega_bilinear_form_scaled` – Function.

```
omega_bilinear_form_scaled(::Type{DT}) -> Tuple{Int, SMatrix{R,R,Int}}
```

Return $(S, B_{\omega,S})$ where $B_{\omega,S} = S(C^{-1})^{\mathsf{T}} B C^{-1}$ is the bilinear form in the fundamental weight basis, scaled by the smallest positive integer S that makes all entries integral. This is a compile-time constant.

Examples

```
julia> using Semisimple

julia> first(omega_bilinear_form_scaled(TypeA{2}))
3
```

Connection index

The determinant of the Cartan matrix is the **connection index**, which gives the index of the root lattice Q in the weight lattice P:

$$\det(C) = [P : Q]$$

For simply-laced types (A, D, E), the index varies by type. For multiply-laced types, the determinant encodes how the roots and weights are related:

```
julia> using Semisimple

julia> cartan_determinant(TypeA{3})   # A3: det = 4 = n+1
4

julia> cartan_determinant(TypeB{2})   # B2 multiply-laced
2

julia> cartan_determinant(TypeG2)
1
```

Part III

Root systems

Chapter 2

Root systems

A root system is determined by its Dynkin type. Semisimple.jl caches one immutable `RootSystem` singleton per type, so constructing the same type twice returns the same object.

2.1 Creating a root system

```
julia> using Semisimple

julia> RS = RootSystem(TypeA{3})
Root system of type A3, rank 3 with 6 positive roots

julia> n_roots(RS)
12

julia> n_simple_roots(RS)
3
```

`Semisimple.RootSystem` – Type.

```
RootSystem{DT,R,N}
```

A root system for the Dynkin type `DT` of rank `R` with `N` positive roots. For small ranks the data is emitted directly by a `@generated` constructor; for larger ranks it is built once at runtime and cached per Dynkin type.

Fields:

- `positive_roots_list`: `NTuple{N, SVector{R,Int}}` of positive roots, ordered by non-decreasing height (`pos_roots[N]` is the highest root).
- `positive_coroots_list`: `NTuple{N, SVector{R,Int}}` of positive coroots, in the same order as the roots.
- `refl`: `SMatrix{R,N,UInt}` reflection table — `refl[s, i]` = index of `s_s(α_i)` among positive roots, or 0 if the result is negative.
- `highest_coroot_idx`: the index of the positive coroot with greatest height (= index of the highest short root in `positive_roots_list`).

Examples

```
julia> using Semisimple

julia> RootSystem(TypeA{2})
Root system of type A2, rank 2 with 3 positive roots
```

`Semisimple.n_roots` – Function.

```
n_roots(RS::RootSystem) -> Int
```

Return the total number of roots (positive and negative).

Examples

```
julia> using Semisimple

julia> n_roots(RootSystem(TypeA{2}))
6
```

`Semisimple.n_simple_roots` – Function.

```
n_simple_roots(RS::RootSystem) -> Int
```

Return the number of simple roots, equal to the rank of the root system.

Examples

```
julia> using Semisimple

julia> n_simple_roots(RootSystem(TypeA{2}))
2
```

2.2 Simple and positive roots

Roots are stored in the simple root basis as `RootSpaceElem` values. Use `coefficients` to extract the underlying coordinate vector:

```
julia> α1 = simple_root(RS, 1)
α1

julia> α2 = simple_root(RS, 2)
α2

julia> α1 + α2
α1 + α2
```

```
julia> 2 * α1
2α1

julia> coefficients(α1)
3-element StaticArraysCore.SVector{3, Int64} with indices SOneTo(3):
 1
 0
 0
```

Listing roots

Positive roots are returned in **non-decreasing order of height**: indices 1 through `rank` are the simple roots (height 1), and the last element is always the highest root.

```
julia> length(positive_roots(RS))
6

julia> positive_root(RS, 3)
α3

julia> positive_root(RS, 4)
α1 + α2

julia> highest_root(RS)
α1 + α2 + α3
```

Because the ordering is canonical, `highest_root` simply returns the last positive root — no search is performed.

Negative roots are the negations of positive roots:

```
julia> negative_root(RS, 1)
-α1
```

`Semisimple.RootSpaceElem` – Type.

```
RootSpaceElem{DT,R}
```

An element of the root space for Dynkin type `DT` of rank `R`, stored as an `SVector{R,Int}` of coordinates in the simple root basis.

Examples

```
julia> using Semisimple

julia> RootSpaceElem(TypeA{2}, [1, 1])
α1 + α2
```

`Semisimple.simple_roots` – Function.

```
simple_roots(RS::RootSystem{DT,R}) -> Vector{RootSpaceElem}
```

Return all simple roots.

Examples

```
julia> using Semisimple

julia> simple_roots(RootSystem(TypeA{2}))
2-element Vector{RootSpaceElem{TypeA{2}, 2}}:
 α1
 α2
```

`Semisimple.simple_root` – Function.

```
simple_root(RS::RootSystem{DT,R}, i) -> RootSpaceElem
```

Return the `i`-th simple root.

Examples

```
julia> using Semisimple

julia> simple_root(RootSystem(TypeA{2}), 1)
α1
```

`Semisimple.positive_roots` – Function.

```
positive_roots(RS::RootSystem{DT,R}) -> Vector{RootSpaceElem}
```

Return all positive roots.

Examples

```
julia> using Semisimple

julia> length(positive_roots(RootSystem(TypeA{2})))
3
```

`Semisimple.positive_root` – Function.

```
positive_root(RS::RootSystem{DT,R}, i) -> RootSpaceElem
```

Return the `i`-th positive root.

Examples

```
julia> using Semisimple

julia> positive_root(RootSystem(TypeA{2}), 3)
α1 + α2
```

`Semisimple.negative_roots` – Function.

```
negative_roots(RS::RootSystem{DT,R}) -> Vector{RootSpaceElem}
```

Return all negative roots.

Examples

```
julia> using Semisimple

julia> length(negative_roots(RootSystem(TypeA{2})))
3
```

`Semisimple.negative_root` – Function.

```
negative_root(RS::RootSystem{DT,R}, i) -> RootSpaceElem
```

Return the `i`-th negative root (negative of the `i`-th positive root).

Examples

```
julia> using Semisimple

julia> negative_root(RootSystem(TypeA{2}), 1)
-α1
```

`Semisimple.roots` – Function.

```
roots(RS::RootSystem) -> Vector{RootSpaceElem}
```

Return all roots (positive followed by negative).

Examples

```
julia> using Semisimple

julia> length(roots(RootSystem(TypeA{2})))
6
```

`Semisimple.root` – Function.

```
root(RS::RootSystem{DT,R}, i) -> RootSpaceElem
```

Return the i-th root. Indices 1..n*pos are positive roots, n*pos+1..2*n_pos are negative roots.

Examples

```
julia> using Semisimple

julia> root(RootSystem(TypeA{2}), 4)
-α1
```

`Semisimple.highest_root` – Function.

```
highest_root(RS::RootSystem{DT,R}) -> RootSpaceElem
```

Return the highest root. Positive roots are ordered by non-decreasing height, so the highest root is always the last positive root.

Examples

```
julia> using Semisimple

julia> highest_root(RootSystem(TypeA{2}))
α1 + α2
```

`Semisimple.highest_short_root` – Function.

```
highest_short_root(RS::RootSystem{DT,R}) -> RootSpaceElem
```

Return the highest short root: the positive root of minimal length that has greatest height among all short positive roots.

For simply-laced types (A, D, E), every root has the same length, so this coincides with `highest_root`.

The index equals `RS.highest_coroot_idx`, precomputed at compile time.

Examples

```
julia> using Semisimple

julia> RS = RootSystem(TypeB{2});

julia> coefficients(highest_short_root(RS))
2-element StaticArraysCore.SVector{2, Int64} with indices SOneTo(2):
 1
 1

julia> RS_G2 = RootSystem(TypeG2);
```

```
julia> coefficients(highest_short_root(RS_G2))
2-element StaticArraysCore.SVector{2, Int64} with indices SOneTo(2):
 2
 1
```

`Semisimple.coefficients` – Function.

```
coefficients(r::RootSpaceElem) -> SVector
coefficients(w::WeightLatticeElem) -> SVector
```

Return the coordinate vector of a root space element (in the simple root basis) or of a weight lattice element (in the fundamental weight basis).

Examples

```
julia> using Semisimple

julia> coefficients(RootSpaceElem(TypeA{2}, [1, 1])) == [1, 1]
true
```

2.3 Root queries

```
julia> is_root(RS, α1 + α2)
true

julia> is_positive_root(RS, α1 + α2)
true

julia> is_root(RS, 2 * α1)
false
```

`Semisimple.is_root` – Function.

```
is_root(RS::RootSystem{DT,R}, v::RootSpaceElem{DT,R}) -> Bool
```

Check whether v is a root.

Examples

```
julia> using Semisimple

julia> RS = RootSystem(TypeA{2});

julia> is_root(RS, RootSpaceElem(TypeA{2}, [1, 1]))
true
```

`Semisimple.is_positive_root` – Function.

```
is_positive_root(RS::RootSystem{DT,R}, v::RootSpaceElem{DT,R}) -> Bool
```

Check whether v is a positive root.

Examples

```
julia> using Semisimple

julia> RS = RootSystem(TypeA{2});

julia> is_positive_root(RS, RootSpaceElem(TypeA{2}, [-1, 0]))
false
```

2.4 Height

The height of a root is the sum of its simple root coefficients:

```
julia> height(α1)
1

julia> height(α1 + α2)
2

julia> height(highest_root(RS))
3
```

`Semisimple.height` – Function.

```
height(r::RootSpaceElem) -> Int
```

Sum of coefficients in the simple root expansion.

Examples

```
julia> using Semisimple

julia> height(RootSpaceElem(TypeA{2}, [1, 1]))
2
```

2.5 Inner product

The inner product on the root space uses the symmetrised Cartan form $(\alpha, \beta) = \alpha^T \operatorname{diag}(d)\, C\, \beta$:

```
julia> dot(α1, α1)
2//1

julia> dot(α1, α2)
-1//1
```

2.6 Coroots

For simply-laced types (A, D, E), coroots coincide with roots. For non-simply-laced types the coroots are rescaled by the symmetriser entries:

```
julia> sc = simple_coroots(RS);

julia> sc[1]
α1

julia> length(positive_coroots(RS))
6
```

`Semisimple.simple_coroots` – Function.

```
simple_coroots(RS::RootSystem{DT,R}) -> Vector{RootSpaceElem}
```

Return the simple coroots.

Examples

```
julia> using Semisimple

julia> simple_coroots(RootSystem(TypeA{2})) == simple_roots(RootSystem(TypeA{2}))
true
```

`Semisimple.positive_coroots` – Function.

```
positive_coroots(RS::RootSystem{DT,R}) -> Vector{RootSpaceElem}
```

Return all positive coroots.

Examples

```
julia> using Semisimple

julia> length(positive_coroots(RootSystem(TypeB{2})))
4
```

2.7 Coxeter invariants

The **highest root** is a fundamental invariant of the root system. When expressed in the simple root basis as $\theta = \sum_i m_i \alpha_i$, the coefficients m_i are the **Coxeter labels** or **marks** (returned by `coxeter_coefficients`). Because positive roots are sorted by height, `highest_root(RS)` is simply `positive_root(RS, N)` where `N` is the number of positive roots — no search needed.

The **highest short root** θ_s is the short root of greatest height. For simply-laced types (A, D, E) it coincides with θ. Its index in the positive root list is precomputed at compile time and stored in `RS.highest_coroot_idx`.

The **highest coroot** (or **dominant coroot**) $\theta^\vee$ is the positive coroot of greatest height. It equals the coroot of the highest short root, and is stored at the same precomputed index `RS.highest_coroot_idx`.

The **Coxeter number** $h = 1 + \sum_i m_i$ is the order of a Coxeter element in the Weyl group.

For the **dual root system** (Langlands dual, swapping B↔C), the corresponding invariants are the **dual Coxeter labels** and **dual Coxeter number** $h^\vee$.

```
julia> c_coeff = coxeter_coefficients(TypeA{3})
3-element StaticArraysCore.SVector{3, Int64} with indices SOneTo(3):
 1
 1
 1

julia> coxeter_number(TypeA{3})
4

julia> cartan_determinant(TypeA{3})
4
```

For multiply-laced types like G_2:

```
julia> c_coeff_G2 = coxeter_coefficients(TypeG2)
2-element StaticArraysCore.SVector{2, Int64} with indices SOneTo(2):
 3
 2

julia> coxeter_number(TypeG2)
6
```

`Semisimple.highest_coroot` – Function.

```
highest_coroot(RS::RootSystem{DT,R}) -> RootSpaceElem
```

Return the highest coroot $\theta^\vee$: the positive coroot of greatest height. This is the coroot of the highest short root.

The index is precomputed at compile time and stored in `RS.highest_coroot_idx`.

Examples

```
julia> using Semisimple

julia> highest_coroot(RootSystem(TypeA{2}))
α1 + α2
```

Semisimple.coxeter_coefficients – Function.

```
coxeter_coefficients(::Type{DT}) -> SVector{R,Int}
coxeter_coefficients(dt::DT) -> SVector{R,Int}
```

Return the **Coxeter labels** (also called marks): the coefficients of the highest root in the simple root basis: $\theta = \sum_i m_i \alpha_i$

These are not the Weyl group exponents; the degrees of fundamental invariants are returned by degrees_fundamental_invar and the exponents are those degrees minus 1.

Examples

```
julia> using Semisimple

julia> coxeter_coefficients(TypeA{3})
3-element StaticArraysCore.SVector{3, Int64} with indices SOneTo(3):
 1
 1
 1

julia> coxeter_coefficients(TypeB{2})
2-element StaticArraysCore.SVector{2, Int64} with indices SOneTo(2):
 1
 2
```

Semisimple.dual_coxeter_coefficients – Function.

```
dual_coxeter_coefficients(::Type{DT}) -> SVector{R,Int}
dual_coxeter_coefficients(dt::DT) -> SVector{R,Int}
```

Return the **dual Coxeter coefficients**: the coefficients of simple roots in the highest short root of the dual root system (Langlands dual). The dual Coxeter number is $h^\vee = 1 + \sum_i n_i^\vee$.

For simply-laced types (A, D, E) all roots have the same length, so these equal the Coxeter coefficients. For B, C, F4, and G2 they differ.

Examples

```
julia> using Semisimple

julia> dual_coxeter_coefficients(TypeB{2})
2-element StaticArraysCore.SVector{2, Int64} with indices SOneTo(2):
 1
 1
```

```
julia> dual_coxeter_coefficients(TypeG2)
2-element StaticArraysCore.SVector{2, Int64} with indices SOneTo(2):
 1
 2
```

`Semisimple.coxeter_number` – Function.

```
coxeter_number(::Type{DT}) -> Int
coxeter_number(dt::DT) -> Int
```

Return the **Coxeter number** h of the Dynkin type, defined as $h = 1 + \sum_i m_i$ where m_i are the Coxeter coefficients (coefficients of the highest root).

The Coxeter number is the order of a Coxeter element (product of all simple reflections) in the Weyl group.

Examples

```
julia> using Semisimple

julia> coxeter_number(TypeA{1})
2

julia> coxeter_number(TypeA{3})
4

julia> coxeter_number(TypeG2)
6
```

`Semisimple.dual_coxeter_number` – Function.

```
dual_coxeter_number(::Type{DT}) -> Int
dual_coxeter_number(dt::DT) -> Int
```

Return the **dual Coxeter number** $h^\vee$ of the Dynkin type, which is the Coxeter number of the Langlands dual root system.

Examples

```
julia> using Semisimple

julia> dual_coxeter_number(TypeA{1})
2

julia> dual_coxeter_number(TypeA{3})
4

julia> dual_coxeter_number(TypeB{2})
3
```

```
julia> dual_coxeter_number(TypeG2)
4
```

`Semisimple.degrees_fundamental_invariants` – Function.

```
degrees_fundamental_invariants(::Type{DT}) -> SVector{R,Int}
degrees_fundamental_invariants(dt::DT) -> SVector{R,Int}
```

Return the degrees of the fundamental invariants of the Weyl group action on the polynomial ring.

Examples

```
julia> using Semisimple

julia> degrees_fundamental_invariants(TypeA{2})
2-element StaticArraysCore.SVector{2, Int64} with indices SOneTo(2):
 2
 3

julia> degrees_fundamental_invariants(TypeB{3})
3-element StaticArraysCore.SVector{3, Int64} with indices SOneTo(3):
 2
 4
 6

julia> degrees_fundamental_invariants(TypeD{4})
4-element StaticArraysCore.SVector{4, Int64} with indices SOneTo(4):
 2
 4
 6
 4

julia> degrees_fundamental_invariants(TypeG2)
2-element StaticArraysCore.SVector{2, Int64} with indices SOneTo(2):
 2
 6
```

2.8 Examples

A2

```
julia> RS2 = RootSystem(TypeA{2})
Root system of type A2, rank 2 with 3 positive roots

julia> [positive_root(RS2, i) for i in 1:3]
3-element Vector{RootSpaceElem{TypeA{2}, 2}}:
 α1
 α2
```

```
 α1 + α2

julia> highest_root(RS2)
α1 + α2

julia> highest_short_root(RS2)
α1 + α2

julia> coefficients(highest_coroot(RS2))
2-element StaticArraysCore.SVector{2, Int64} with indices SOneTo(2):
 1
 1
```

A_2 is simply-laced, so the highest root, highest short root, and highest coroot all coincide.

G2

The exceptional type G_2 has short roots of squared length 2 and long roots of squared length 6. The highest short root and the highest long root are distinct:

```
julia> RS_G2 = RootSystem(TypeG2);

julia> n_positive_roots(TypeG2)
6

julia> highest_root(RS_G2)
3α1 + 2α2

julia> highest_short_root(RS_G2)
2α1 + α2

julia> coefficients(highest_coroot(RS_G2))
2-element StaticArraysCore.SVector{2, Int64} with indices SOneTo(2):
 2
 3
```

B2

```
julia> RS_B2 = RootSystem(TypeB{2});

julia> highest_root(RS_B2)
α1 + 2α2

julia> highest_short_root(RS_B2)
α1 + α2

julia> coefficients(highest_coroot(RS_B2))
2-element StaticArraysCore.SVector{2, Int64} with indices SOneTo(2):
 2
 1
```

Part IV

Weight lattice

Chapter 3

Weight lattice

Weights are elements of the weight lattice, expressed in the **fundamental weight basis** $(\omega_1, \ldots, \omega_r)$ where $\langle \alpha_i^\vee, \omega_j \rangle = \delta_{ij}$.

3.1 Creating weights

Weights are constructed with `fundamental_weight` or directly from a coefficient vector using `WeightLatticeElem`:

```
julia> using Semisimple

julia> ω1 = fundamental_weight(TypeA{3}, 1)
ω1

julia> ω2 = fundamental_weight(TypeA{3}, 2)
ω2

julia> ω3 = fundamental_weight(TypeA{3}, 3)
ω3

julia> ω1 + ω2
ω1 + ω2

julia> 2 * ω1
2ω1

julia> WeightLatticeElem(TypeA{3}, [3, 1, 0])
3ω1 + ω2
```

All fundamental weights

```
julia> fundamental_weights(TypeA{3})
3-element Vector{WeightLatticeElem{TypeA{3}, 3}}:
 ω1
 ω2
 ω3
```

Weyl vector

The Weyl vector $\rho = \omega_1 + \cdots + \omega_r$:

```
julia> weyl_vector(TypeA{3})
ω1 + ω2 + ω3
```

`Semisimple.WeightLatticeElem` – Type.

```
WeightLatticeElem{DT,R}
```

An element of the weight lattice for Dynkin type `DT` of rank `R`, stored as an `SVector{R,Int}` of coordinates in the fundamental weight basis $(\omega_1, \ldots, \omega_r)$.

The pairing with the i-th simple coroot is simply `w[i]`: $\langle \alpha_i^\vee, \lambda \rangle = \lambda_i$

Constructors

```
WeightLatticeElem(::Type{DT}, v::AbstractVector{<:Integer})
```

When `v` has fewer entries than `rank(DT)`, the remaining coordinates are silently filled with zeros. When `v` has more entries than `rank(DT)`, a warning is emitted and only the first `rank(DT)` entries are used.

Length handling

The `AbstractVector` constructor is meant as a convenience for interactive work. For library code, tests, and reproducible computations, prefer the exact-length `SVector` or `NTuple` constructors so dimension mismatches are caught immediately. Padding can change the intended weight by implicitly adding zero coordinates, while truncation discards trailing coordinates after emitting a warning.

Examples

```
julia> using Semisimple

julia> WeightLatticeElem(TypeA{3}, [1, 2])   # padded with one zero
ω1 + 2ω2

julia> WeightLatticeElem(TypeA{3}, [1, 2, 3])  # exact length
ω1 + 2ω2 + 3ω3

julia> using Test

julia> @test_logs (:warn, r"truncating to first 3 entries") WeightLatticeElem(TypeA{3}, [1, 2,
↪  3, 4])
ω1 + 2ω2 + 3ω3
```

`Semisimple.fundamental_weight` – Function.

```
fundamental_weight(::Type{DT}, i) -> WeightLatticeElem{DT}
```

Return the `i`-th fundamental weight ω_i.

Examples

```
julia> using Semisimple

julia> fundamental_weight(TypeA{3}, 1)
ω1

julia> fundamental_weight(TypeB{2}, 2)
ω2
```

`Semisimple.fundamental_weights` – Function.

```
fundamental_weights(::Type{DT}) -> Vector{WeightLatticeElem{DT}}
```

Return all fundamental weights.

Examples

```
julia> using Semisimple

julia> fundamental_weights(TypeA{2})
2-element Vector{WeightLatticeElem{TypeA{2}, 2}}:
 ω1
 ω2
```

`Semisimple.weyl_vector` – Function.

```
weyl_vector(::Type{DT}) -> WeightLatticeElem{DT}
```

Return the Weyl vector $\rho = \omega_1 + \omega_2 + \cdots + \omega_r = \frac{1}{2}\sum_{\alpha>0}\alpha$.

Examples

```
julia> using Semisimple

julia> weyl_vector(TypeA{3})
ω1 + ω2 + ω3
```

3.2 Display

Weights are printed in the fundamental weight basis by default: `ω1`, `2ω1 + ω2`, `0`, etc.

Per-call compact format

Pass `:compact => true` via `IOContext` to switch a single `show` call to the concise coordinate form `DT[c₁,c₂,…]`:

```
julia> show(IOContext(stdout, :compact => true), ω1)
A3[1,0,0]

julia> show(IOContext(stdout, :compact => true), 2*ω1 - ω3)
A3[2,0,-1]
```

The same compact form is used by `RootSpaceElem`.

Global compact toggle

Call `compact_display!` to make the compact form the session-wide default for all `WeightLatticeElem` and `RootSpaceElem` output:

```
julia> compact_display!(true)
true

julia> fundamental_weights(TypeA{3})
3-element Vector{WeightLatticeElem{TypeA{3}, 3}}:
 A3[1,0,0]
 A3[0,1,0]
 A3[0,0,1]

julia> compact_display!(false)   # restore default
false
```

`Semisimple.compact_display!` – Function.

```
compact_display!(val::Bool = true)
```

Set the global compact-display mode for `WeightLatticeElem` and `RootSpaceElem`.

When `true`, every call to `show` on these types prints the coordinate form `DT[c₁,c₂,…]` (e.g. `A3[1,0,0]` for ω_1 in type A_3) regardless of the `IOContext`. When `false` (the default), the standard symbolic form is used (ω1, α1 + α2, etc.).

The per-call `IOContext(:compact => true)` override always takes precedence.

Examples

```
julia> using Semisimple

julia> ω1 = fundamental_weight(TypeA{3}, 1);

julia> compact_display!(true)
true

julia> ω1
A3[1,0,0]

julia> fundamental_weights(TypeA{3})
3-element Vector{WeightLatticeElem{TypeA{3}, 3}}:
 A3[1,0,0]
```

```
 A3[0,1,0]
 A3[0,0,1]

julia> compact_display!(false)
false

julia> ω1
ω1
```

3.3 Dominance

A weight is **dominant** when all its fundamental weight coordinates are non-negative:

```
julia> is_dominant(ω1)
true

julia> is_dominant(ω1 - 2 * ω2)
false
```

Conjugation to the dominant chamber

Every weight is Weyl-conjugate to a unique dominant weight:

```
julia> w = WeightLatticeElem(TypeA{3}, [-1, 2, 0]);

julia> is_dominant(w)
false

julia> conjugate_dominant_weight(w)
ω1 + ω2
```

`Semisimple.is_dominant` – Function.

```
is_dominant(w::WeightLatticeElem) -> Bool
```

A weight is dominant iff all its coordinates (pairings with simple coroots) are `>= 0`.

Examples

```
julia> using Semisimple

julia> is_dominant(fundamental_weight(TypeA{2}, 1))
true

julia> is_dominant(WeightLatticeElem(TypeA{2}, [-1, 1]))
false
```

`Semisimple.conjugate_dominant_weight` – Function.

```
conjugate_dominant_weight(w::WeightLatticeElem{DT,R}) -> WeightLatticeElem{DT,R}
```

Return the unique dominant weight in the Weyl orbit of `w`.

Examples

```
julia> using Semisimple

julia> conjugate_dominant_weight(WeightLatticeElem(TypeA{2}, [-1, 1]))
ω1

julia> conjugate_dominant_weight(fundamental_weight(TypeA{3}, 1))
ω1
```

`Semisimple.conjugate_dominant_weight_with_elem` – Function.

```
conjugate_dominant_weight_with_elem(w::WeightLatticeElem{DT,R}) -> (WeightLatticeElem,
↪  Vector{Int})
```

Return the dominant weight and the sequence of simple reflections applied.

Examples

```
julia> using Semisimple

julia> conjugate_dominant_weight_with_elem(WeightLatticeElem(TypeA{2}, [-1, 1]))
(ω1, [1])
```

`Semisimple.conjugate_dominant_weight_with_length` – Function.

```
conjugate_dominant_weight_with_length(w::WeightLatticeElem{DT,R}) -> (WeightLatticeElem, Int)
```

Return the dominant weight in the Weyl orbit of `w` together with the number of simple reflections applied (i.e. the length of the Weyl group element mapping `w` into the dominant chamber).

This is faster than `conjugate_dominant_weight_with_elem` because it only tracks a counter instead of building the full word.

Examples

```
julia> using Semisimple

julia> conjugate_dominant_weight_with_length(WeightLatticeElem(TypeA{2}, [-1, 1]))
(ω1, 1)

julia> conjugate_dominant_weight_with_length(fundamental_weight(TypeA{3}, 1))
(ω1, 0)
```

3.4 Reflections

Simple reflections act on weights by the formula $s_i(\lambda) = \lambda - \langle \alpha_i^\vee, \lambda \rangle \alpha_i$, which in the fundamental weight basis simplifies to $(s_i(\lambda))_j = \lambda_j - \lambda_i C_{ji}$, because $\alpha_i = \sum_j C_{ji}\omega_j$:

```
julia> reflect(ω1, 1)   # reflection in the first simple root
-ω1 + ω2

julia> reflect(ω1, 2)   # unchanged because the pairing is zero
ω1
```

`Semisimple.reflect` – Function.

```
reflect(w::WeightLatticeElem{DT,R}, s::Integer) -> WeightLatticeElem{DT,R}
```

Reflect `w` by the `s`-th simple reflection: $s_s(\lambda) = \lambda - \langle \alpha_s^\vee, \lambda \rangle \alpha_s$

In the fundamental weight basis, $\langle \alpha_s^\vee, \lambda \rangle = \lambda_s$ and $\alpha_s = \sum_j C_{js}\omega_j$, so the new weight has coordinates: $(s_s(\lambda))_j = \lambda_j - \lambda_s C_{js}$

Examples

```
julia> using Semisimple

julia> reflect(WeightLatticeElem(TypeA{2}, [2, 1]), 1)
-2ω1 + 3ω2
```

```
reflect(w::WeightLatticeElem{DT,R}, β::RootSpaceElem{DT,R}) -> WeightLatticeElem{DT,R}
```

Reflect `w` by the root `β`: $s_\beta(\lambda) = \lambda - \langle \beta^\vee, \lambda \rangle \beta$ where $\langle \beta^\vee, \lambda \rangle = 2(\beta, \lambda)/(\beta, \beta)$.

The argument β must be an actual root of the root system.

Examples

```
julia> using Semisimple

julia> reflect(fundamental_weight(TypeA{2}, 1), simple_root(RootSystem(TypeA{2}), 1))
-ω1 + ω2
```

3.5 Inner products

Pairing of roots and weights, $\langle \alpha^\vee, \lambda \rangle$, and the weight-space inner product:

```
julia> RS = RootSystem(TypeA{2});

julia> α1 = simple_root(RS, 1);

julia> ω1 = fundamental_weight(TypeA{2}, 1);
```

```
julia> ω2 = fundamental_weight(TypeA{2}, 2);

julia> dot(α1, ω1)   # simple-coroot pairing equals 1
1//1

julia> dot(α1, ω2)   # simple-coroot pairing equals 0
0//1

julia> dot(ω1, ω1)   # invariant bilinear form
2//3

julia> dot(ω1, ω2)
1//3
```

`LinearAlgebra.dot` – Function.

```
dot(a::RootSpaceElem{DT,R}, b::RootSpaceElem{DT,R}) -> Rational{Int}
```

Inner product of two root space elements using the symmetrized Cartan form.

$(,) = \mathrm{diag}(d)C$

```
dot(r::RootSpaceElem{DT,R}, w::WeightLatticeElem{DT,R}) -> Rational{Int}
```

Compute the inner product (α, λ) between a root α (in simple-root coordinates) and a weight λ (in fundamental-weight coordinates).

Following OSCAR's convention: $(\alpha, \lambda) = \sum_i \alpha_i d_i \lambda_i$ where d is the Cartan symmetrizer.

This works because $(\alpha_i, \omega_j) = d_i \delta_{ij}$, which follows from $\langle \alpha_i^\vee, \omega_j \rangle = \delta_{ij}$ and $\alpha_i^\vee = \alpha_i / d_i$ in the bilinear-form sense.

Examples

```
julia> using Semisimple

julia> dot(simple_root(RootSystem(TypeA{2}), 1), fundamental_weight(TypeA{2}, 1))
1//1
```

```
dot(w1::WeightLatticeElem{DT,R}, w2::WeightLatticeElem{DT,R}) -> Rational{Int}
```

Compute the inner product (λ, μ) between two weights. Both are given in fundamental-weight coordinates; the implementation converts to root coordinates and applies the bilinear form there.

Examples

```
julia> using Semisimple

julia> dot(fundamental_weight(TypeA{2}, 1), fundamental_weight(TypeA{2}, 1))
2//3
```

3.6 Conversions

Weights and roots live in different coordinate systems. Convert between them:

```
julia> α1_as_weight = WeightLatticeElem(simple_root(RS, 1))
2ω1 - ω2

julia> ρ_as_root = RootSpaceElem(weyl_vector(TypeA{2}))
α1 + α2
```

Every weight lies in the rational span of the simple roots via C^{-1}. It lies in the root lattice only when those rational simple-root coordinates are integral; otherwise `RootSpaceElem(w)` throws an `ArgumentError`.

Part V

Weyl groups

Chapter 4

Weyl groups

The Weyl group is generated by simple reflections $s_1, \ldots, s_r$. Elements are stored as reduced words (sequences of generator indices).

Semisimple.jl writes Weyl group actions on the right: `λ * x` denotes the usual geometric action of x on the weight or root λ. Products are applied in the same order as written in `λ * (s1 * s2)`, matching OSCAR's right-action convention for roots and weights.

4.1 Creating the Weyl group

```
julia> using Semisimple

julia> W = weyl_group(TypeA{3})
Weyl group of type A3

julia> weyl_order(TypeA{3})
24
```

`Semisimple.WeylGroup` – Type.

```
WeylGroup{DT,R}
```

The Weyl group of a root system of Dynkin type `DT` with rank `R`.

Semisimple.jl writes Weyl group actions on the right: `λ * x` denotes the usual action of the Weyl group element x on the weight or root λ.

Examples

```
julia> using Semisimple

julia> weyl_order(TypeA{2})
6
```

`Semisimple.WeylGroupElem` – Type.

```
WeylGroupElem{DT,R}
```

An element of the Weyl group, stored as a reduced word (vector of simple reflection indices).

Examples

```
julia> using Semisimple

julia> W = weyl_group(TypeA{2});

julia> W([1, 2])
s1 * s2
```

`Semisimple.weyl_group` – Function.

```
weyl_group(::Type{DT}) -> WeylGroup{DT}
```

Construct the Weyl group for the given Dynkin type.

Examples

```
julia> using Semisimple

julia> W = weyl_group(TypeA{2})
Weyl group of type A2
```

`Semisimple.weyl_order` – Function.

```
weyl_order(::Type{DT}) -> BigInt
```

Return the order of the Weyl group of type `DT`.

Examples

```
julia> using Semisimple

julia> weyl_order(TypeA{3})
24

julia> weyl_order(TypeE{8})
696729600
```

`Semisimple.root_system` – Function.

```
root_system(W::WeylGroup) -> RootSystem
```

Return the root system underlying the Weyl group `W`.

Examples

```
julia> using Semisimple

julia> root_system(weyl_group(TypeA{2}))
Root system of type A2, rank 2 with 3 positive roots
```

4.2 Generators and multiplication

```
julia> s1 = gen(W, 1)
s1

julia> s2 = gen(W, 2)
s2

julia> s1 * s2
s1 * s2

julia> word(s1)
1-element Vector{UInt8}:
 0x01

julia> word(s1 * s2)
2-element Vector{UInt8}:
 0x01
 0x02
```

Longest element

The longest element w_0 has maximal length in the Weyl group:

```
julia> w0 = longest_element(W)
s1 * s2 * s1 * s3 * s2 * s1

julia> length(word(w0))
6
```

`Semisimple.gens` – Function.

```
gens(W::WeylGroup) -> Vector{WeylGroupElem}
```

Return all simple reflections.

Examples

```
julia> using Semisimple

julia> gens(weyl_group(TypeA{2}))
2-element Vector{WeylGroupElem{TypeA{2}, 2}}:
 s1
 s2
```

`Semisimple.gen` – Function.

```
gen(W::WeylGroup, i) -> WeylGroupElem
```

Return the `i`-th simple reflection.

Examples

```
julia> using Semisimple

julia> gen(weyl_group(TypeA{2}), 1)
s1
```

`Semisimple.longest_element` – Function.

```
longest_element(W::WeylGroup{DT,R}) -> WeylGroupElem{DT,R}
```

Compute the longest element w0 of the Weyl group. Uses the iterative algorithm: repeatedly find a simple reflection that increases length. The result is cached per Dynkin type.

Examples

```
julia> using Semisimple

julia> W = weyl_group(TypeA{2});

julia> w0 = longest_element(W);

julia> length(w0)
3
```

`Semisimple.word` – Function.

```
word(x::WeylGroupElem) -> Vector{UInt8}
```

Return the reduced word of `x`.

Examples

```
julia> using Semisimple

julia> W = weyl_group(TypeA{2});

julia> word(W([1, 2]))
2-element Vector{UInt8}:
 0x01
 0x02
```

Base.one – Method.

```
one(W::WeylGroup) -> WeylGroupElem
```

Return the identity element.

Examples

```
julia> using Semisimple

julia> one(weyl_group(TypeA{2}))
id
```

Base.length – Method.

```
Base.length(x::WeylGroupElem) -> Int
```

Return the length (number of simple reflections) of x.

Semisimple.rmul! – Function.

```
rmul!(x::WeylGroupElem, s::UInt8) -> WeylGroupElem
```

Multiply x from the right by the simple reflection s, maintaining the reduced word in short-lex normal form.

Uses the reflection table from the root system.

4.3 Action on weights

The Weyl group acts on the weight lattice. Right multiplication corresponds to the geometric reflection:

```
julia> ω1 = fundamental_weight(TypeA{3}, 1);

julia> ω1 * s1   # s1(ω1) = -ω1 + ω2
-ω1 + ω2

julia> ω1 * s2   # s2(ω1) = ω1 (orthogonal)
ω1
```

Base.:* – Method.

```
*(w::WeightLatticeElem{DT,R}, x::WeylGroupElem{DT,R}) -> WeightLatticeElem{DT,R}
```

Right action of a Weyl group element on a weight.

4.4 Action on roots

```
julia> RS = RootSystem(TypeA{3});

julia> α1 = simple_root(RS, 1);

julia> α1 * s1   # s1(α1) = -α1
-α1

julia> α1 * s2   # s2(α1) = α1 + α2
α1 + α2
```

Base.:* – Method.

```
*(r::RootSpaceElem{DT,R}, x::WeylGroupElem{DT,R}) -> RootSpaceElem{DT,R}
```

Right action of a Weyl group element on a root space element.

4.5 Weyl orbits

The orbit of a weight under the full Weyl group:

```
julia> orbit = weyl_orbit(TypeA{3}, ω1);

julia> length(orbit)   # |W/Stab(ω1)| = 4 for std rep of A3
4
```

For the adjoint representation of A_2:

```
julia> ω = fundamental_weight(TypeA{2}, 1) + fundamental_weight(TypeA{2}, 2);

julia> length(weyl_orbit(TypeA{2}, ω))
6
```

Semisimple.weyl_orbit – Function.

```
weyl_orbit(::Type{DT}, w::WeightLatticeElem{DT,R}) -> Vector{WeightLatticeElem{DT,R}}
```

Compute the full Weyl orbit of the weight w.

Examples

```
julia> using Semisimple

julia> length(weyl_orbit(TypeA{2}, fundamental_weight(TypeA{2}, 1)))
3
```

4.6 Dominant weights

All dominant weights $\mu \leq \lambda$ (in the dominance order):

```
julia> ω2 = fundamental_weight(TypeA{3}, 2);

julia> dw = dominant_weights(TypeA{3}, ω1 + ω2);

julia> length(dw)
2
```

`Semisimple.dominant_weights` – Function.

```
dominant_weights(::Type{DT}, hw::WeightLatticeElem{DT,R}) -> Vector{WeightLatticeElem{DT,R}}
```

Compute the dominant weights occurring in the irreducible representation with highest weight hw, sorted by decreasing level below hw.

The level of μ below hw is the root-lattice height of hw - μ, i.e. the sum of coefficients when hw - μ is written in the simple root basis.

Examples

```
julia> using Semisimple

julia> λ = fundamental_weight(TypeA{2}, 1) + fundamental_weight(TypeA{2}, 2);

julia> length(dominant_weights(λ))
2
```

4.7 Weyl dimension formula

The dimension of the irreducible representation $\mathrm{V}(\lambda)$:

```
julia> degree(ω1)   # standard rep of A3 (SL4)
4

julia> degree(ω2)   # □² of standard = 6-dim
6
```

```
julia> degree(ω1 + ω2)   # 20-dim rep
20

julia> degree(weyl_vector(TypeA{3}))   # ρ = ω1+ω2+ω3
64
```

A_2 examples:

```
julia> ω1_a2 = fundamental_weight(TypeA{2}, 1);

julia> degree(ω1_a2)
3

julia> degree(ω1_a2 + fundamental_weight(TypeA{2}, 2))   # adjoint
8
```

The zero weight gives the trivial representation:

```
julia> degree(WeightLatticeElem(TypeA{3}, [0, 0, 0]))
1
```

`Semisimple.degree` – Function.

```
degree(::Type{DT}, hw::WeightLatticeElem{DT,R}) -> BigInt
degree(hw::WeightLatticeElem{DT,R}) -> BigInt
```

Dimension of the irreducible representation with highest weight `hw`, computed via the Weyl dimension formula:

$$\dim V(\lambda) = \prod_{\alpha>0} \frac{\langle \lambda+\rho, \alpha^\vee\rangle}{\langle \rho, \alpha^\vee\rangle}$$

Equivalently, using the invariant bilinear form, $\prod_{\alpha>0}(\lambda+\rho,\alpha)/(\rho,\alpha)$.

The denominator and the symmetrizer-scaled root vectors are precomputed once per Dynkin type. The numerator is computed as a `BigInt` product of `Int`-valued inner products via in-place GMP arithmetic.

Examples

```
julia> using Semisimple

julia> degree(fundamental_weight(TypeA{3}, 1))
4

julia> degree(fundamental_weight(TypeB{3}, 3))
8

julia> degree(fundamental_weight(TypeE{8}, 8))
248

julia> [degree(fundamental_weight(TypeB{3}, i)) for i in 1:3]
3-element Vector{BigInt}:
```

```
 7
21
 8
```

```
degree(::Type{DT}, v::AbstractVector{<:Integer}) -> BigInt
```

Dimension of the irreducible representation with highest weight given by the integer vector v (in the fundamental weight basis).

This is a convenience wrapper: `degree(DT, v) == degree(WeightLatticeElem(DT, v))`.

Examples

```
julia> using Semisimple

julia> degree(TypeA{2}, [1, 0])  # standard representation of A2
3

julia> degree(TypeE{8}, [0, 0, 0, 0, 0, 0, 0, 1])  # adjoint of E8
248
```

```
degree(V::WeylCharacter) -> BigInt
```

Return the (signed) dimension of a character in the representation ring.

For a virtual character $V = \sum m_i V(\lambda_i)$ (with m_i possibly negative), returns:

$$\deg(V) = \sum m_i \dim(V(\lambda_i))$$

For effective characters (all $m_i \geq 0$) this equals the total dimension of the corresponding representation. For virtual characters (some $m_i < 0$) the result can be negative or zero; this is the Euler characteristic in the representation ring.

Returns a `BigInt`.

Examples

```
julia> using Semisimple

julia> ω1 = fundamental_weight(TypeA{2}, 1);

julia> V = WeylCharacter(ω1);

julia> degree(V)  # dim of standard representation
3

julia> degree(V^2)  # dim of V ⊗ V = Sym²V ⊕ □²V
9

julia> degree(Sym(2, V))  # dim of Sym²V
```

```
6

julia> # Virtual character: V(ω1) - V(ω2) has degree 0 (both dim 3)
       degree(V - WeylCharacter(fundamental_weight(TypeA{2}, 2)))
0

julia> ω8 = fundamental_weight(TypeE{8}, 8);

julia> degree(WeylCharacter(ω8))
248
```

`Semisimple.weyl_dimension` – Function.

```
weyl_dimension(λ::WeightLatticeElem) -> BigInt
weyl_dimension(::Type{DT}, λ::WeightLatticeElem) -> BigInt
weyl_dimension(::Type{DT}, v::AbstractVector{<:Integer}) -> BigInt
weyl_dimension(dt::DynkinType, v) -> BigInt
```

Synonym for `degree`. Computes the dimension of the irreducible representation via the Weyl dimension formula.

4.8 Borel–Weil–Bott theorem

Compute the cohomological degree and resulting representation for a weight on a flag variety:

```
julia> import Semisimple: borel_weil_bott

julia> borel_weil_bott(ω1)   # dominant weight → degree 0
(0, ω1)

julia> borel_weil_bott(WeightLatticeElem(TypeA{3}, [-3, 2, 1]))
(1, ω1 + ω3)
```

Singular weights give zero cohomology, and `borel_weil_bott` returns `nothing`:

```
julia> borel_weil_bott(-weyl_vector(TypeA{3})) === nothing
true
```

Note

`borel_weil_bott` is not exported. It is rather a feature for `PartialFlagVarieties.jl`. Use `import Semisimple: borel_weil_bott` to access it.

`Semisimple.borel_weil_bott` – Function.

```
borel_weil_bott(λ::WeightLatticeElem{DT,R}) -> Union{Nothing, Tuple{Int,
↪  WeightLatticeElem{DT,R}}}
```

Apply the Borel–Weil–Bott theorem to the weight λ.

Package placement

This function is a preview implementation that properly belongs to `PartialFlagVarieties.jl`, an upcoming companion package. It is included here for convenience but is **not part of the public API of `Semisimple.jl`** and is not exported. Access it via `import Semisimple: borel_weil_bott`.

Compute `μ = λ + ρ` and find the unique Weyl group element `w` such that `w(μ)` is dominant. If `μ` is singular (lies on a Weyl chamber wall), all cohomology vanishes and we return `nothing`. Otherwise, return `(d, w(μ) - ρ)` where `d = ℓ(w)` is the cohomological degree, meaning

$$\mathrm{H}^d(G/B, \mathcal{L}) \cong \mathrm{V}^*_{w\cdot\lambda}$$

and all other cohomology groups vanish.

Examples

```
julia> using Semisimple; import Semisimple: borel_weil_bott

julia> borel_weil_bott(fundamental_weight(TypeA{2}, 1))
(0, ω1)

julia> borel_weil_bott(WeightLatticeElem(TypeA{2}, [-2, 1]))
(1, 0)

julia> borel_weil_bott(-weyl_vector(TypeA{2})) === nothing
true
```

Singular weights

A weight λ is **singular** when $\langle \lambda + \rho, \alpha^\vee \rangle = 0$ for some positive root α. Singular weights give zero cohomology:

```
julia> λ = WeightLatticeElem(TypeA{2}, [-1, 0]);

julia> is_singular(λ)
true
```

`Semisimple.is_singular` – Function.

```
is_singular(w::WeightLatticeElem{DT,R}) -> Bool
```

Check whether the weight `w` is singular, i.e. lies on some wall of a Weyl chamber. Equivalently, `w` is singular iff `⟨αv, w⟩ = 0` for some positive root `α`.

For a dominant weight this simplifies to checking whether any fundamental weight coordinate is zero. For a general weight, we first conjugate to the dominant chamber.

Examples

```
julia> using Semisimple

julia> is_singular(fundamental_weight(TypeA{2}, 1))
true

julia> is_singular(weyl_vector(TypeA{2}))
false
```

4.9 Bruhat order and descent sets

The **Bruhat order** on the Weyl group is a partial order defined by subword inclusion in reduced expressions. Descent sets record which simple reflections reduce the word length.

```
julia> s3 = gen(W, 3)
s3

julia> x = s1 * s2;

julia> right_descent_set(x)
1-element Vector{Int64}:
 2

julia> bruhat_leq(s1, x)
true

julia> bruhat_leq(x, s1)
false
```

`Semisimple.right_descent_set` – Function.

```
right_descent_set(w::WeylGroupElem) -> Vector{Int}
```

Return the right descent set of `w`, i.e. the indices `i` such that `\ell(ws_i) < \ell(w)`.

Examples

```
julia> using Semisimple

julia> W = weyl_group(TypeA{2});

julia> right_descent_set(W([1, 2]))
1-element Vector{Int64}:
 2
```

`Semisimple.left_descent_set` – Function.

```
left_descent_set(w::WeylGroupElem) -> Vector{Int}
```

Return the left descent set of `w`, i.e. the indices `i` such that `\ell(s_iw) < \ell(w)`.

Examples

```
julia> using Semisimple

julia> W = weyl_group(TypeA{2});

julia> left_descent_set(W([1, 2]))
1-element Vector{Int64}:
 1
```

`Semisimple.bruhat_leq` – Function.

```
bruhat_leq(x::WeylGroupElem, y::WeylGroupElem) -> Bool
```

Return whether `x \le y` in the (strong) Bruhat order.

Examples

```
julia> using Semisimple

julia> W = weyl_group(TypeA{2});

julia> bruhat_leq(gen(W, 1), W([1, 2]))
true
```

`Semisimple.bruhat_descendants` – Function.

```
bruhat_descendants(w::WeylGroupElem) -> Vector{WeylGroupElem}
```

Return the immediate Bruhat descendants obtained by right-multiplying by simple reflections in the right descent set.

Examples

```
julia> using Semisimple

julia> W = weyl_group(TypeA{2});

julia> bruhat_descendants(W([1, 2]))
1-element Vector{WeylGroupElem{TypeA{2}, 2}}:
 s1
```

4.10 Parabolic subgroups and coset representatives

For a subset $I \subseteq \{1, \ldots, r\}$ of simple root indices, the **parabolic subgroup** W_I is generated by $\{s_i : i \in I\}$. The minimal-length coset representatives for W/W_I are the elements whose right descent sets are disjoint from I.

```
julia> reps = right_coset_reps(W, [1]);  # W/⟨s1⟩ for A3

julia> length(reps)   # |A3|/|A1×A1 ... actually just |W|/|W_{1}| = 24/2 = 12
12

julia> all(w -> !(1 in right_descent_set(w)), reps)
true
```

Semisimple.right_coset_reps – Function.

```
right_coset_reps(W::WeylGroup, I::AbstractVector{<:Integer}) -> Vector{WeylGroupElem}
```

Enumerate minimal right coset representatives for W/W_I, where W_I is the parabolic subgroup generated by simple reflections in I.

Uses a weight-orbit BFS: the weight $_I = \sum_{j \notin I}$ $_j$ has stabilizer exactly W_I, so its W-orbit has size $|W/W_I|$. Enumerates that orbit using $O(|W/W_I| \cdot R)$ weight reflections, independent of $|W|$.

Examples

```
julia> using Semisimple

julia> length(right_coset_reps(weyl_group(TypeA{2}), [1]))
3
```

Semisimple.left_coset_reps – Function.

```
left_coset_reps(W::WeylGroup, I::AbstractVector{<:Integer}) -> Vector{WeylGroupElem}
```

Enumerate minimal left coset representatives for W_I\W.

Examples

```
julia> using Semisimple

julia> length(left_coset_reps(weyl_group(TypeA{2}), [1]))
3
```

4.11 Weyl group orders

```
julia> weyl_order(TypeA{3})
24

julia> weyl_order(TypeB{3})
48
```

```
julia> weyl_order(TypeG2)
12

julia> weyl_order(TypeE{8})
696729600
```

Part VI

Characters and representations

Chapter 5

Characters and representations

A `WeylCharacter` is an element of the representation ring (Grothendieck ring) of a semisimple Lie algebra. It stores a formal $\mathbb{Z}$-linear combination of irreducible representations, indexed by their dominant highest weights.

5.1 Constructing characters

```
julia> using Semisimple, StaticArrays

julia> ω1 = fundamental_weight(TypeA{3}, 1);

julia> ω2 = fundamental_weight(TypeA{3}, 2);

julia> V = WeylCharacter(ω1)   # irreducible V(ω1)
A3(1, 0, 0)

julia> WeylCharacter(TypeA{3})   # zero character (additive identity)
0

julia> is_effective(V)
true

julia> is_irreducible(V)
true

julia> highest_weight(V)
ω1
```

`Semisimple.WeylCharacter` – Type.

```
WeylCharacter{DT,R}
```

An element of the representation ring (Grothendieck ring) of a semisimple Lie algebra of Dynkin type `DT` with rank `R`.

Computationally, this is stored as a `Dict{WeightLatticeElem{DT,R}, Int}` mapping dominant highest weights to irreducible multiplicities. Positive multiplicities correspond to actual representations; negative multiplicities arise in virtual differences.

This is distinct from the full weight-multiplicity character returned by `freudenthal_formula` and the dominant-representative weight multiplicity dictionary returned by `dominant_character`.

Examples

```
julia> using Semisimple

julia> ω1 = fundamental_weight(TypeA{2}, 1);

julia> V = WeylCharacter(ω1)
A2(1, 0)

julia> V + V == 2 * V
true
```

`Semisimple.is_effective` – Function.

```
is_effective(V::WeylCharacter) -> Bool
```

True when all multiplicities are non-negative, i.e. `V` corresponds to an actual (not merely virtual) representation.

Examples

```
julia> using Semisimple

julia> V = WeylCharacter(fundamental_weight(TypeA{2}, 1));

julia> is_effective(V - 2V)
false

julia> is_effective(V - V)
true
```

`Semisimple.is_irreducible` – Function.

```
is_irreducible(V::WeylCharacter) -> Bool
```

True when `V` is a single irreducible with multiplicity 1.

Examples

```
julia> using Semisimple

julia> V = WeylCharacter(fundamental_weight(TypeA{2}, 1));

julia> is_irreducible(V)
true

julia> is_irreducible(2V)
false
```

`Semisimple.highest_weight` – Function.

```
highest_weight(V::WeylCharacter{DT,R}) -> WeightLatticeElem{DT,R}
```

Return the highest weight of an irreducible virtual character. Throws if `V` is not irreducible.

Examples

```
julia> using Semisimple

julia> V = WeylCharacter(fundamental_weight(TypeA{2}, 1));

julia> highest_weight(V)
ω1
```

5.2 Arithmetic

Characters support addition, subtraction, and scalar multiplication:

```
julia> V2 = WeylCharacter(ω2);

julia> V + V2
A3(1, 0, 0) + A3(0, 1, 0)

julia> 2 * V
2*A3(1, 0, 0)

julia> V + V == 2 * V
true
```

Characters also support multiplication (tensor product) and exponentiation (tensor power):

```
julia> V * V2 == tensor_product(V, V2)
true

julia> V^3 == V * V * V
true
```

`Base.:*` – Method.

```
*(V::WeylCharacter, W::WeylCharacter) -> WeylCharacter
```

Tensor product of virtual characters, computed via Brauer–Klimyk. Equivalent to the ring multiplication in the representation ring.

`Base.:^` – Method.

```
^(V::WeylCharacter, n::Integer) -> WeylCharacter
```

Compute the n-th tensor power of V.

Algorithm choice: uses right-to-left sequential multiplication `V ⊗ (V ⊗ (V ⊗ ⋯))` rather than repeated squaring.

With Brauer–Klimyk, the cost of `A ⊗ B` is proportional to `|A| × Weyl-orbit-size(B)` (or vice versa). Sequential multiplication keeps one factor always equal to the original small V, so every step is `O(|V| × orbit_size(V^{r−1}))`. Repeated squaring would let both factors grow: the intermediate result `V^{n/2}` can be much larger than V itself, making each squaring step significantly more expensive.

Examples

```
julia> using Semisimple

julia> ω1 = fundamental_weight(TypeA{2}, 1);

julia> V = WeylCharacter(ω1);

julia> V^2 == tensor_product(ω1, ω1)
true

julia> V^0 == WeylCharacter(zero(ω1))
true
```

In-place mutation

For performance-critical loops, use `add!` and `addmul!`:

```
julia> result = WeylCharacter(TypeA{3});

julia> add!(result, V);

julia> result
A3(1, 0, 0)

julia> addmul!(result, V2, 3);

julia> result
A3(1, 0, 0) + 3*A3(0, 1, 0)
```

`Semisimple.add!` – Function.

```
add!(V::WeylCharacter{DT,R}, W::WeylCharacter{DT,R}) -> WeylCharacter{DT,R}
```

Add W into V in-place, modifying V. Returns V.

Examples

```
julia> using Semisimple

julia> ω1 = fundamental_weight(TypeA{2}, 1);

julia> V = WeylCharacter(ω1); W = WeylCharacter(ω1);

julia> add!(V, W) == 2 * WeylCharacter(ω1)
true
```

`Semisimple.addmul!` – Function.

```
addmul!(V::WeylCharacter{DT,R}, W::WeylCharacter{DT,R}, c::Integer) -> WeylCharacter{DT,R}
```

Add `c * W` into `V` in-place, modifying `V`. Returns `V`.

Examples

```
julia> using Semisimple

julia> ω1 = fundamental_weight(TypeA{2}, 1);

julia> V = WeylCharacter(TypeA{2}); W = WeylCharacter(ω1);

julia> addmul!(V, W, 3) == 3 * WeylCharacter(ω1)
true
```

5.3 Character data terminology

Semisimple.jl uses three related but distinct character representations:

Name	Julia representation	Meaning
`WeylCharacter`	`Dict{WeightLatticeElem,Int}` in the `terms` field	irreducible decomposition in the representation ring
Full weight character	`Dict{SVector{R,Int},Int}` from `freudenthal_formula`	multiplicity of every weight
Dominant character	`Dict{SVector{R,Int},Int}` from `dominant_character`	multiplicity on dominant representatives of Weyl orbits

5.4 Character polynomials

A **character polynomial** of a finite-dimensional representation is a formal sum

$$\chi(\lambda) = \sum_{\mu \in P} m_\lambda(\mu) e^\mu$$

where P is the weight lattice, $m_\lambda(\mu)$ is the multiplicity of weight μ in the representation with highest weight λ, and e^μ is a formal exponential. This encodes the weight multiplicities in a single object that respects the representation ring structure (tensor product → multiplication of characters).

Since character polynomials are **Weyl group invariant** (i.e., $\chi_\lambda(w \cdot \mu) = \chi_\lambda(\mu)$ for all $w \in W$), the full character is determined by its values on dominant weights. More precisely: each weight orbit under the Weyl group contains exactly one dominant weight, and all weights in an orbit have equal multiplicity.

```
julia> ω1_a2 = fundamental_weight(TypeA{2}, 1);

julia> full_mults = freudenthal_formula(ω1_a2);

julia> length(full_mults)   # includes all W-orbit members
3

julia> dom_mults = dominant_character(ω1_a2);

julia> length(dom_mults)   # only dominant weights
1
```

5.5 Dominant character polynomials

The **dominant character polynomial** (or sometimes just "dominant character") is a more compact representation:

$$\chi_{\lambda,\text{dom}} = \sum_{\mu \in P^+ : m_\lambda(\mu) > 0} m_\lambda(\mu) e^\mu$$

where P^+ is the set of dominant weights. By W-invariance, this omits all non-dominant weights while retaining *all information* about the full character.

Explicitly: since each W-orbit contains a unique dominant representative, reconstructing the full character from the dominant character is straightforward — just apply the Weyl group orbit operator. This is what `freudenthal_formula` does internally.

`dominant_character` computes this using Freudenthal's recursion formula and returns a `Dict{SVector{R,Int}, Int}` mapping dominant weight coordinates to multiplicities. This is the cached building block behind `freudenthal_formula`, `weight_multiplicity`, tensor products, Adams operators, and plethysms.

This dominant character corresponds to the `domchar` output in LiE.

```
julia> ω1_a2 = fundamental_weight(TypeA{2}, 1);

julia> dc = dominant_character(ω1_a2);

julia> length(dc)   # only 1 dominant weight for V(ω1) of A2
1

julia> dc[SVector(1, 0)]  # the highest weight itself
1
```

`Semisimple.dominant_character` – Function.

```
dominant_character(λ::WeightLatticeElem{DT,R}) -> Dict{SVector{R,Int}, BigInt}
```

Compute the **dominant character** of the irreducible representation $\mathrm{V}()$ using Freudenthal's recursion.

The result records the weight multiplicity on the unique dominant representative of each Weyl orbit of weights in $\mathrm{V}()$. It is a compact form of the full weight character, not an irreducible decomposition.

This function computes the dominant character using Freudenthal's recursion formula and returns a `Dict{SVector{R,Int}, BigInt}` with dominant weight coordinates as keys and multiplicities as values. Multiplicities are stored as `BigInt` because the Freudenthal recursion accumulates intermediates that exceed `typemax(Int64)` for large representations such as $\mathrm{V}()$ of E_8.

Results are cached per highest weight; clear with `clear_all_caches!`.

Expanding the dominant character:

- Use `freudenthal_formula` to expand the dominant character into the full character polynomial (all W-orbit members listed with their multiplicities).
- Use `weight_multiplicity` to query the multiplicity of a single weight without computing the entire W-orbit.

Examples

```
julia> using Semisimple; using StaticArrays

julia> ω1 = fundamental_weight(TypeA{2}, 1);

julia> dc = dominant_character(ω1);

julia> length(dc)   # V(ω1) of A2 has 1 dominant weight
1

julia> dc[SVector(1, 0)]   # multiplicity of ω1 itself
1

julia> adj = ω1 + fundamental_weight(TypeA{2}, 2);

julia> dc_adj = dominant_character(adj);

julia> dc_adj[SVector(0, 0)]   # zero weight multiplicity in adjoint
2
```

5.6 Freudenthal's formula

Compute the full weight multiplicity dictionary of an irreducible representation $\mathrm{V}(\lambda)$. Returns a `Dict{SVector{R,Int}, Int}` mapping weight coordinates (in the fundamental weight basis) to their multiplicities:

```
julia> m = freudenthal_formula(ω1);

julia> length(m)   # V(ω1) of A3 has 4 weights
4
```

```
julia> sum(values(m)) == degree(ω1)   # total = dimension
true
```

The convenience function `weight_multiplicity` returns the multiplicity of a single weight. For example, the A_2 adjoint representation $V(\omega_1 + \omega_2)$ has dimension 8, and the zero weight has multiplicity 2:

```
julia> adj = fundamental_weight(TypeA{2}, 1) + fundamental_weight(TypeA{2}, 2);

julia> weight_multiplicity(adj, zero(adj))
2
```

`Semisimple.freudenthal_formula` – Function.

```
freudenthal_formula(λ::WeightLatticeElem{DT,R}) -> Dict{SVector{R,Int}, BigInt}
```

Compute the full weight multiplicity dictionary of the irreducible representation $V()$.

Returns a dictionary mapping *every* weight (in fundamental weight coordinates) to its multiplicity. Only weights with non-zero multiplicity are included.

Multiplicities are returned as `BigInt` because they can exceed `typemax(Int64)` for large representations (e.g. $V()$ of E_8).

Internally, this calls `dominant_character` to compute multiplicities of dominant weights via Freudenthal's recursion, then expands each dominant weight to its full Weyl orbit.

Examples

```
julia> using Semisimple; using StaticArrays

julia> ω1 = fundamental_weight(TypeA{2}, 1);

julia> mults = freudenthal_formula(ω1);

julia> mults[SVector(1, 0)]
1

julia> sum(values(mults))  # = dim V(ω1) = 3
3

julia> ω2 = fundamental_weight(TypeA{2}, 2);

julia> mults = freudenthal_formula(ω1 + ω2);  # adjoint of A2

julia> length(mults)  # 7 distinct weights
7

julia> sum(values(mults))  # dim = 8
8
```

`Semisimple.weight_multiplicity` – Function.

```
weight_multiplicity(λ::WeightLatticeElem{DT,R}, μ::WeightLatticeElem{DT,R}) -> BigInt
```

Return the multiplicity of weight μ in the irreducible representation $V(\lambda)$. This is a convenience wrapper around `freudenthal_formula`. The return type is `BigInt` because multiplicities in large representations (e.g. $V(\rho)$ of E_8) can exceed `typemax(Int64)`.

Examples

```
julia> using Semisimple

julia> ω1 = fundamental_weight(TypeA{2}, 1); ω2 = fundamental_weight(TypeA{2}, 2);

julia> weight_multiplicity(ω1 + ω2, zero(ω1))   # zero weight of adjoint
2
```

5.7 Tensor products

Brauer–Klimyk (general)

The default tensor product algorithm works for all types:

```
julia> tensor_product(ω1, ω1)   # V ⊗ V = Sym²V ⊕ □²V
A3(2, 0, 0) + A3(0, 1, 0)

julia> tensor_product(ω1, ω2)
A3(1, 1, 0) + A3(0, 0, 1)
```

Tensor product of a weight with the trivial representation:

```
julia> ω3 = fundamental_weight(TypeA{3}, 3);

julia> tensor_product(ω1, ω3)   # V(ω1) ⊗ V(ω3) contains trivial
A3(1, 0, 1) + A3(0, 0, 0)
```

Littlewood–Richardson (type A)

For type A, the Littlewood–Richardson rule provides a faster tensor product algorithm. It is used automatically by `tensor_product` for type A inputs:

```
julia> ω1_a4 = fundamental_weight(TypeA{4}, 1);

julia> ω2_a4 = fundamental_weight(TypeA{4}, 2);

julia> lr_tensor_product(ω1_a4, ω2_a4)
A4(1, 1, 0, 0) + A4(0, 0, 1, 0)
```

```
julia> λ = WeightLatticeElem(TypeA{4}, [2, 1, 0, 0]);

julia> r = tensor_product(λ, λ);

julia> length(r.terms)   # number of irreducible components
11

julia> sum(m * degree(μ) for (μ, m) in r.terms) == degree(λ)^2
true
```

`Semisimple.tensor_product` – Function.

```
tensor_product(V::WeylCharacter{DT,R}, W::WeylCharacter{DT,R}) -> WeylCharacter{DT,R}
```

Tensor product decomposition of two virtual characters. Iterates over all pairs `(λ, m)` in `V` and `(μ, n)` in `W`, computes `tensor_product(λ, μ)` via Brauer–Klimyk, and accumulates `m * n * result`.

Examples

```
julia> using Semisimple

julia> ω1 = fundamental_weight(TypeA{2}, 1);

julia> V = WeylCharacter(ω1);

julia> V * V == Sym(2, V) + □(2, V)
true

julia> V^3  # right-to-left tensor power
A2(3, 0) + 2*A2(1, 1) + A2(0, 0)
```

```
tensor_product(λ::WeightLatticeElem{DT,R}, μ::WeightLatticeElem{DT,R}) -> WeylCharacter{DT,R}
```

Decompose the tensor product $\mathrm{V}()\otimes\mathrm{V}()$ into irreducibles using the Brauer–Klimyk algorithm. The Freudenthal formula is applied to whichever factor has smaller dimension, for efficiency.

Examples

```
julia> using Semisimple

julia> ω1 = fundamental_weight(TypeA{2}, 1); ω2 = fundamental_weight(TypeA{2}, 2);

julia> tensor_product(ω1, ω1)
A2(2, 0) + A2(0, 1)

julia> ω1 = fundamental_weight(TypeA{3}, 1); tensor_product(ω1, ω1)
A3(2, 0, 0) + A3(0, 1, 0)
```

```
tensor_product(λ::WeightLatticeElem{TypeA{N},N}, μ::WeightLatticeElem{TypeA{N},N}) ->
↪  WeylCharacter{TypeA{N},N}
```

Specialization for type A: uses the Littlewood–Richardson rule instead of Brauer–Klimyk, which is typically much faster.

`Semisimple.lr_tensor_product` – Function.

```
lr_tensor_product(λ::WeightLatticeElem{TypeA{N},N}, μ::WeightLatticeElem{TypeA{N},N}) ->
↪  WeylCharacter{TypeA{N},N}
```

Decompose $\mathrm{V}() \otimes \mathrm{V}()$ into irreducibles using the Littlewood–Richardson rule. This is specific to type A and is typically much faster than the general Brauer–Klimyk algorithm.

The algorithm converts highest weights to partitions, enumerates LR skew tableaux of shape $/$ with content (where and are the partitions for and), and reads off the multiplicities c.

Examples

```
julia> using Semisimple

julia> ω1 = fundamental_weight(TypeA{2}, 1); ω2 = fundamental_weight(TypeA{2}, 2);

julia> lr_tensor_product(ω1, ω1)
A2(2, 0) + A2(0, 1)

julia> lr_tensor_product(ω1, ω2)
A2(1, 1) + A2(0, 0)

julia> lr_tensor_product(ω1 + ω2, ω1)  # 8 ⊗ 3 in A2
A2(2, 1) + A2(1, 0) + A2(0, 2)
```

5.8 Duality

The dual (contragredient) representation:

```
julia> dual(WeylCharacter(ω1))
A3(0, 0, 1)

julia> dual(WeylCharacter(ω3))
A3(1, 0, 0)

julia> dual(WeylCharacter(ω2))   # self-dual for A3
A3(0, 1, 0)
```

`Semisimple.dual` – Function.

```
dual(λ::WeightLatticeElem{DT,R}) -> WeightLatticeElem{DT,R}
```

Highest weight of the contragredient (dual) representation: `λ* = -w₀(λ)`.

Examples

```
julia> using Semisimple

julia> ω1 = fundamental_weight(TypeA{2}, 1); ω2 = fundamental_weight(TypeA{2}, 2);

julia> dual(ω1) == ω2
true

julia> [dual(fundamental_weight(TypeA{3}, i)) for i in 1:3]
3-element Vector{WeightLatticeElem{TypeA{3}, 3}}:
 ω3
 ω2
 ω1
```

```
dual(V::WeylCharacter{DT,R}) -> WeylCharacter{DT,R}
```

Dual of a virtual character: each summand $\mathrm{V}()$ maps to $\mathrm{V}(^*)$.

Examples

```
julia> using Semisimple

julia> ω1 = fundamental_weight(TypeA{2}, 1);

julia> dual(WeylCharacter(ω1))
A2(0, 1)
```

5.9 Adams operators

The Adams operator ψ^k scales every weight of $\mathrm{V}(\lambda)$ by k. The result is returned as a `Dict{SVector{R,Int}, Int}` mapping weight coordinates to multiplicities (not decomposed into irreducibles). Use `character_from_weights` to convert it to a `WeylCharacter`:

```
julia> ω1_a2 = fundamental_weight(TypeA{2}, 1);

julia> ψ2 = adams_operator(ω1_a2, 2);

julia> length(ψ2)   # number of distinct weights
3
```

The Newton identity connects Adams operators to symmetric/exterior powers: $\psi^2(\mathrm{V}) = \mathrm{Sym}^2(\mathrm{V}) - \bigwedge^2(\mathrm{V})$:

```
julia> ψ2_char = character_from_weights(TypeA{2}, ψ2);

julia> ψ2_char == Sym(2, ω1_a2) - □(2, ω1_a2)
true
```

`Semisimple.adams_operator` – Function.

```
adams_operator(λ::WeightLatticeElem{DT,R}, k::Integer) -> Dict{SVector{R,Int}, BigInt}
```

Compute the k-th Adams operator $^k(\mathrm{V}())$, returned as a dictionary of weight multiplicities (not decomposed into irreducibles).

The Adams operator scales every weight by k: if $\mathrm{V}()$ has weight multiplicity $m()$, then $^k(\mathrm{V}())$ has $m()$ at weight k. Multiplicities are stored as `BigInt` (propagated from `freudenthal_formula`).

Examples

```
julia> using Semisimple, StaticArrays

julia> m = adams_operator(fundamental_weight(TypeA{2}, 1), 2);

julia> length(m), m[SVector(2, 0)]
(3, 1)
```

`Semisimple.character_from_weights` – Function.

```
character_from_weights(::Type{DT}, multiplicities::Dict{SVector{R,Int}, <:Integer}) ->
↪  WeylCharacter{DT,R}
```

Given a dictionary of weight multiplicities (as from Freudenthal), decompose the representation into a formal sum of irreducibles (a virtual character if some multiplicities are negative).

Uses the "peeling" algorithm: at each step find the dominant weight with the largest root-basis height `ht_root(λ) = (col-sums of C⁻¹)·λ` (with ties broken in favour of dominant weights — necessary for minuscule representations of simply-laced types), subtract the Freudenthal multiplicities of that irreducible, and repeat until `mults` is empty.

The root-basis height is the unique correct linear height for the peeling order: it is strictly positive on positive roots, strictly larger on the dominant weight of each Weyl orbit than on any non-dominant orbit member, and handles all Lie types (including G_2 where the Cartan symmetrizer gives the wrong order).

Examples

```
julia> using Semisimple

julia> ω1 = fundamental_weight(TypeA{2}, 1);

julia> character_from_weights(TypeA{2}, freudenthal_formula(ω1))
A2(1, 0)
```

5.10 Exterior powers

The k-th exterior power $\bigwedge^k \mathrm{V}$ is computed via the **Newton–Girard recurrence**, which relates exterior powers to Adams operators (power-sum symmetric functions):

$$k \cdot \bigwedge^{k}(\mathrm{V}) = \sum_{r=1}^{k} (-1)^{r-1}\, \psi^{r}(\mathrm{V}) \cdot \bigwedge^{k-r}(\mathrm{V})$$

This is the representation-ring analogue of the classical identity $k\, e_k = \sum_{r=1}^{k} (-1)^{r-1} p_r\, e_{k-r}$ relating elementary symmetric polynomials e_k to power-sum polynomials p_r.

Both `WeightLatticeElem` and `WeylCharacter` are accepted:

```
julia> □(2, V)   # □²V(ω1) of A3 = V(ω2)
A3(0, 1, 0)

julia> □(3, V)   # □³V(ω1) of A3 = V(ω3)
A3(0, 0, 1)

julia> □(4, V)   # top exterior power = trivial (det)
A3(0, 0, 0)

julia> □(5, V)   # exceeds dim = 0
0
```

Dimension identity

$\dim \bigwedge^{k} \mathrm{V} = \binom{\dim \mathrm{V}}{k}$:

```
julia> r = □(2, V);

julia> sum(m * degree(μ) for (μ, m) in r.terms) == binomial(4, 2)
true
```

Type A fundamental representations

For A_n, $\bigwedge^{k} \mathrm{V}(\omega_1) = \mathrm{V}(\omega_k)$:

```
julia> ω = [fundamental_weight(TypeA{5}, i) for i in 1:5];

julia> all(□(k, WeylCharacter(ω[1])) == WeylCharacter(ω[k]) for k in 1:5)
true
```

Non-minuscule exterior powers

```
julia> adj = ω1 + ω3;   # adjoint of A3 (15-dim)

julia> r = □(2, adj);

julia> is_effective(r)
true
```

```
julia> sum(m * degree(μ) for (μ, m) in r.terms) == binomial(15, 2)
true
```

Semisimple.exterior_power – Function.

```
exterior_power(λ::WeightLatticeElem{DT,R}, k::Integer) -> WeylCharacter{DT,R}
```

Compute the k-th exterior power $\bigwedge^k \mathrm{V}()$ of the irreducible representation with highest weight λ, using the Newton–Girard recurrence:

$k \cdot \bigwedge^k(\mathrm{V}) = \sum_{r=1}^{k} (-1)^{r-1} {}^r(\mathrm{V}) \cdot \bigwedge^{k-r}(\mathrm{V})$

This is the representation-ring analogue of the classical identity $k\, e_k = \sum_{r=1}^{k} (-1)^{r-1} p_r\, e_{k-r}$ relating the elementary symmetric polynomials e_k to the power-sum polynomials p_r. Here the Adams operator ^{r} plays the role of p_r.

Results are memoized for efficiency in recursive calls.

Examples

```
julia> using Semisimple

julia> spin = fundamental_weight(TypeB{3}, 3);  # B3 spin rep, dim 8

julia> □(6, spin)
B3(1, 0, 0) + B3(0, 1, 0)
```

```
exterior_power(V::WeylCharacter{DT,R}, k::Integer) -> WeylCharacter{DT,R}
```

Compute the k-th exterior power $\bigwedge^k(V)$ of a virtual character `V`.

For an irreducible character $V = \mathrm{V}()$ this delegates to the weight-level `exterior_power(λ, k)`. For a general virtual character $V = \sum_i m_i \mathrm{V}(_i)$ the Newton–Girard recurrence is applied at the level of characters, with the Adams operator applied component-wise:

$k \cdot \bigwedge^k(V) = \sum_{r=1}^{k} (-1)^{r-1} {}^r(V) \cdot \bigwedge^{k-r}(V)$

Results are memoized.

Examples

```
julia> using Semisimple

julia> ω1 = fundamental_weight(TypeA{3}, 1);

julia> V = WeylCharacter(ω1);

julia> exterior_power(V, 2)
A3(0, 1, 0)

julia> exterior_power(V, 4)   # top exterior power = trivial (det)
A3(0, 0, 0)
```

```
julia> exterior_power(V, 5)   # exceeds dimension = 0
0

julia> exterior_power(V, 2) == exterior_power(ω1, 2)
true
```

Semisimple.□ – Function.

```
□(k::Integer, λ::WeightLatticeElem) -> WeylCharacter
□(k::Integer, V::WeylCharacter) -> WeylCharacter
```

Compute the k-th exterior power. Shorthand for exterior_power.

Examples

```
julia> using Semisimple

julia> ω1 = fundamental_weight(TypeA{3}, 1);

julia> □(2, ω1) == WeylCharacter(fundamental_weight(TypeA{3}, 2))
true

julia> V = WeylCharacter(ω1);

julia> □(2, V) == WeylCharacter(fundamental_weight(TypeA{3}, 2))
true
```

5.11 Symmetric powers

The k-th symmetric power $\mathrm{Sym}^k \mathrm{V}$ is likewise computed via the **Newton–Girard recurrence**:

$$k \cdot \mathrm{Sym}^k(\mathrm{V}) = \sum_{r=1}^{k} \psi^r(\mathrm{V}) \cdot \mathrm{Sym}^{k-r}(\mathrm{V})$$

This is the representation-ring analogue of the identity $k\,h_k = \sum_{r=1}^{k} p_r\,h_{k-r}$ for complete homogeneous symmetric polynomials h_k.

Both `WeightLatticeElem` and `WeylCharacter` are accepted:

```
julia> Sym(2, V)   # Sym² of std rep of A3
A3(2, 0, 0)

julia> Sym(0, V)   # Sym⁰ = trivial
A3(0, 0, 0)

julia> Sym(1, V)   # Sym¹ = identity
A3(1, 0, 0)
```

Type A: always irreducible for ω₁

For A_n, $\mathrm{Sym}^k \mathrm{V}(\omega_1) = \mathrm{V}(k\omega_1)$:

```
julia> all(Sym(k, V) == WeylCharacter(k * ω1) for k in 1:5)
true
```

Dimension identity

$\dim \mathrm{Sym}^k \mathrm{V} = \binom{\dim \mathrm{V}+k-1}{k}$:

```
julia> r = Sym(3, V);

julia> sum(m * degree(μ) for (μ, m) in r.terms) == binomial(4 + 2, 3)
true
```

Newton identity: V ⊗ V = Sym²V ⊕ ⋀²V

```
julia> V * V == Sym(2, V) + □(2, V)
true

julia> ω1_g2 = fundamental_weight(TypeG2, 1);

julia> W_g2 = WeylCharacter(ω1_g2);

julia> W_g2 * W_g2 == Sym(2, W_g2) + □(2, W_g2)
true
```

Reducible character example

`Sym` and `□` also accept general `WeylCharacter` values (not just irreducibles). Scaling $V \mapsto 2V$ adds a copy, so: $\mathrm{Sym}^2(2V) = 3\,\mathrm{Sym}^2(V) \oplus \bigwedge^2(V)$:

```
julia> symmetric_power(2 * V, 2) == 3 * Sym(2, V) + □(2, V)
true
```

`Semisimple.symmetric_power` – Function.

```
symmetric_power(λ::WeightLatticeElem{DT,R}, k::Integer) -> WeylCharacter{DT,R}
```

Compute the k-th symmetric power $\mathrm{Sym}^k \mathrm{V}()$ of the irreducible representation with highest weight λ, using the Newton–Girard recurrence:

$k \cdot \mathrm{Sym}^k(\mathrm{V}) = \sum_{r=1}^{k} {}^r(\mathrm{V}) \cdot \mathrm{Sym}^{k-r}(\mathrm{V})$

This is the representation-ring analogue of the classical identity $k\,h_k = \sum_{r=1}^{k} p_r\,h_{k-r}$ relating the complete homogeneous symmetric polynomials h_k to the power-sum polynomials p_r. Here the Adams operator ^{r} plays the role of p_r.

Results are memoized for efficiency in recursive calls.

Examples

```
julia> using Semisimple

julia> spin = fundamental_weight(TypeB{3}, 3);  # B3 spin rep, dim 8

julia> Sym(6, spin)
B3(0, 0, 6) + B3(0, 0, 4) + B3(0, 0, 2) + B3(0, 0, 0)
```

```
symmetric_power(V::WeylCharacter{DT,R}, k::Integer) -> WeylCharacter{DT,R}
```

Compute the k-th symmetric power $\mathrm{Sym}^k(V)$ of a virtual character `V`.

For an irreducible character $V = \mathrm{V}()$ this delegates to the weight-level `symmetric_power(λ, k)`. For a general virtual character $V = \sum_i m_i \mathrm{V}(_i)$ the Newton–Girard recurrence is applied at the level of characters, with the Adams operator applied component-wise:

$k \cdot \mathrm{Sym}^k(V) = \sum_{r=1}^{k} {}^{r}(V) \cdot \mathrm{Sym}^{k-r}(V)$

Results are memoized.

Examples

```
julia> using Semisimple

julia> ω1 = fundamental_weight(TypeA{2}, 1);

julia> V = WeylCharacter(ω1);

julia> symmetric_power(V, 2)
A2(2, 0)

julia> symmetric_power(V, 3)
A2(3, 0)

julia> symmetric_power(V, 0) == WeylCharacter(zero(ω1))
true

julia> # Sym²(V ⊕ V) = 3·Sym²V ⊕ □²V (dimension identity: C(6+1,2) = 21)
       symmetric_power(2 * V, 2) == 3 * symmetric_power(V, 2) + exterior_power(V, 2)
true
```

`Semisimple.Sym` – Function.

```
Sym(k::Integer, λ::WeightLatticeElem) -> WeylCharacter
Sym(k::Integer, V::WeylCharacter) -> WeylCharacter
```

Compute the k-th symmetric power. Shorthand for `symmetric_power`.

Examples

```
julia> using Semisimple

julia> ω1 = fundamental_weight(TypeA{2}, 1);

julia> Sym(2, ω1)
A2(2, 0)

julia> V = WeylCharacter(ω1);

julia> Sym(2, V)
A2(2, 0)

julia> Sym(2, V) == Sym(2, ω1)
true

julia> degree(highest_weight(Sym(3, V)))
10
```

5.12 Plethysm (Schur functors)

The **plethysm** $s_\lambda(\mathrm{V}(\mu))$ applies the Schur functor associated to a partition $\lambda \vdash n$ to an irreducible representation $\mathrm{V}(\mu)$. Symmetric and exterior powers are special cases:

- $s_{(n)} = \mathrm{Sym}^n$
- $s_{(1^n)} = \bigwedge^n$

The general formula uses the **Murnaghan–Nakayama rule** for S_n characters and **Adams operators** (power-sum symmetric functions):

$$s_\lambda(\mathrm{V}) = \frac{1}{n!} \sum_{\kappa \vdash n} \chi^\lambda(\kappa) \cdot |\mathrm{Cl}(\kappa)| \cdot \psi^{\kappa_1}(\mathrm{V}) \otimes \cdots \otimes \psi^{\kappa_m}(\mathrm{V})$$

```
julia> plethysm([2], ω1) == Sym(2, V)   # one-row partition = Sym
true

julia> plethysm([1, 1], ω1) == □(2, V)  # one-column partition = □
true

julia> plethysm([2, 1], ω1)               # mixed symmetry S_{(2,1)}
A3(1, 1, 0)

julia> degree(plethysm([2, 1], ω1))       # mixed-symmetry representation of A3
20
```

Plethysm on non-simply-laced types

```
julia> plethysm([2], ω1_g2) == Sym(2, W_g2)
true

julia> plethysm([1, 1], ω1_g2) == □(2, W_g2)
true
```

`Semisimple.plethysm` – Function.

```
plethysm(λ::Vector{<:Integer}, μ::WeightLatticeElem{DT,R}) -> WeylCharacter{DT,R}
```

Compute the plethysm $s(\mathrm{V}())$, where is a partition of n (parts in weakly decreasing order) and $\mathrm{V}()$ is the irreducible representation with highest weight .

The Schur functor s generalises symmetric and exterior powers:

- $s_{(n)} = \mathrm{Sym}^n$
- $s_{(1,1,\ldots,1)} = \bigwedge^n$

Uses the Murnaghan–Nakayama rule for S_n characters and the formula:

$$s(\mathrm{V}) = \frac{1}{n!} \sum_{\vdash n} () \cdot |\mathrm{Cl}()| \cdot {}^{1}(\mathrm{V}) \otimes \cdots \otimes {}^{m}(\mathrm{V})$$

Examples

```
julia> using Semisimple

julia> ω1 = fundamental_weight(TypeA{3}, 1);

julia> plethysm([2], ω1) == Sym(2, ω1)
true

julia> plethysm([1, 1], ω1) == □(2, ω1)
true

julia> plethysm([2, 1], ω1)  # Mixed symmetry: S_{(2,1)} functor
A3(1, 1, 0)
```

5.13 Reconstructing characters from weights

Given a weight multiplicity dictionary (e.g. from an Adams operator), recover the Weyl character decomposition:

```
julia> m = Dict(SVector(1, 0, 0) => 1, SVector(-1, 1, 0) => 1,
                SVector(0, -1, 1) => 1, SVector(0, 0, -1) => 1);

julia> V_rec = character_from_weights(TypeA{3}, m);
```

```
julia> is_irreducible(V_rec)
true

julia> highest_weight(V_rec) == ω1
true
```

5.14 Cross-type examples

B_3: spin representation

```
julia> ω3_b3 = fundamental_weight(TypeB{3}, 3);

julia> degree(ω3_b3)   # 8-dim spin rep
8

julia> spin = WeylCharacter(ω3_b3);

julia> r = □(2, spin);

julia> sum(m * degree(μ) for (μ, m) in r.terms) == binomial(8, 2)
true
```

G_2: 7-dimensional representation

```
julia> degree(ω1_g2)
7

julia> r = Sym(2, W_g2);

julia> is_effective(r)
true

julia> sum(m * degree(μ) for (μ, m) in r.terms) == binomial(7 + 1, 2)
true
```

E_8: adjoint representation

```
julia> ω8_e8 = fundamental_weight(TypeE{8}, 8);

julia> degree(ω8_e8)   # 248-dim adjoint
248

julia> r = □(2, WeylCharacter(ω8_e8));

julia> length(r.terms)   # 2 irreducible components
2
```

5.15 Representation invariants

Semisimple.jl provides several classical invariants attached to irreducible representations.

Dynkin index

The **Dynkin index** of $\mathrm{V}()$ is the proportionality constant between the trace form on $\mathrm{V}()$ and the Killing form:

$$I() = \frac{\dim \mathrm{V}() \cdot (,\ +2)}{2\ \dim \mathfrak{g}}$$

where the inner product is normalized so that long roots have squared length 2. Equivalently, Semisimple.jl divides its internal bilinear form by $(\theta, \theta)/2$. For the adjoint representation, $I() = h$ (the dual Coxeter number).

```
julia> dynkin_index(fundamental_weight(TypeA{3}, 1))
1//2

julia> adj = adjoint_representation(TypeA{3});

julia> dynkin_index(highest_weight(adj)) == dual_coxeter_number(TypeA{3})
true
```

Casimir eigenvalue

The eigenvalue of the quadratic Casimir element on $\mathrm{V}()$ is $c() = 2(, +2)/(\theta, \theta)$, where θ is the highest root. This normalizes long roots to have squared length 2:

```
julia> casimir_eigenvalue(fundamental_weight(TypeA{3}, 1))
15//4

julia> casimir_eigenvalue(fundamental_weight(TypeB{3}, 3))
21//4
```

Congruency class

```
julia> congruency_class(fundamental_weight(TypeA{3}, 1))
1

julia> congruency_class(fundamental_weight(TypeD{4}, 3))
(1, 0)
```

Self-duality and Frobenius–Schur indicator

```
julia> is_self_dual(fundamental_weight(TypeB{3}, 1))
true
```

```
julia> frobenius_schur_indicator(fundamental_weight(TypeB{3}, 1))
1

julia> frobenius_schur_indicator(fundamental_weight(TypeC{2}, 1))
-1

julia> frobenius_schur_indicator(fundamental_weight(TypeA{2}, 1))
0
```

Adjoint representation

```
julia> adj = adjoint_representation(TypeE{8});

julia> degree(adj)
248
```

Semisimple.dynkin_index – Function.

```
dynkin_index(λ::WeightLatticeElem{DT,R}) -> Rational{BigInt}
```

Compute the Dynkin index of the irreducible representation $\mathrm{V}()$:

$$\ell() = \frac{\dim \mathrm{V}()}{2 \dim \mathfrak{g}} \cdot (,\ +2)$$

where ρ is the Weyl vector, `\dim \mathfrak{g}` is the dimension of the Lie algebra, and the bilinear form is normalized so long roots have squared length

1.

For the adjoint representation, the Dynkin index equals the dual Coxeter number. The result is always a non-negative half-integer.

Examples

```
julia> using Semisimple

julia> dynkin_index(fundamental_weight(TypeA{2}, 1))
1//2

julia> dynkin_index(fundamental_weight(TypeE{8}, 8))
30//1
```

Semisimple.casimir_eigenvalue – Function.

```
casimir_eigenvalue(λ::WeightLatticeElem{DT,R}) -> Rational{Int}
```

Compute the eigenvalue of the quadratic Casimir operator on the irreducible representation $\mathrm{V}()$:

$$c() = \frac{2\,(,\ +2)}{(,)}$$

where $(\cdot,\cdot)$ is the Cartan bilinear form and is the highest root. The normalization ensures long roots have squared length 2.

Examples

```
julia> using Semisimple

julia> casimir_eigenvalue(fundamental_weight(TypeA{2}, 1))
8//3

julia> casimir_eigenvalue(fundamental_weight(TypeB{3}, 3))
21//4
```

`Semisimple.congruency_class` – Function.

```
congruency_class(λ::WeightLatticeElem{DT,R}) -> Union{Int, Tuple{Int,Int}}
```

Compute the congruency class of the irreducible representation $\mathrm{V}()$. Two representations belong to the same congruency class if and only if their difference of highest weights lies in the root lattice.

The return value depends on the Lie algebra type:

- A_N: an integer in $0, 1, \ldots, N$ (mod $N+1$)
- B_N: an integer in $\{0, 1\}$ (mod 2)
- C_N: an integer in $\{0, 1\}$ (mod 2)
- D_N (N even): a tuple (a, b) with $a, b \in \{0, 1\}$ (center $\mathbb{Z}/2 \times \mathbb{Z}/2$)
- D_N (N odd): an integer in $\{0, 1, 2, 3\}$ (center $\mathbb{Z}/4$)
- E_6: an integer in $\{0, 1, 2\}$ (mod 3)
- E_7: an integer in $\{0, 1\}$ (mod 2)
- $\mathrm{E}_8, \mathrm{F}_4, \mathrm{G}_2$: always 0 (trivial center)

Examples

```
julia> using Semisimple

julia> congruency_class(fundamental_weight(TypeA{2}, 1))
1

julia> congruency_class(fundamental_weight(TypeD{4}, 1))
(1, 1)
```

`Semisimple.is_self_dual` – Function.

```
is_self_dual(λ::WeightLatticeElem{DT,R}) -> Bool
```

Return `true` if the irreducible representation $\mathrm{V}(\lambda)$ is isomorphic to its dual $\mathrm{V}(\lambda)^*$, i.e. if $\lambda = -w_0(\lambda)$.

Examples

```
julia> using Semisimple

julia> is_self_dual(fundamental_weight(TypeA{2}, 1))
false

julia> is_self_dual(WeightLatticeElem(TypeA{2}, [1, 1]))
true

julia> is_self_dual(fundamental_weight(TypeB{3}, 1))
true
```

`Semisimple.frobenius_schur_indicator` – Function.

```
frobenius_schur_indicator(λ::WeightLatticeElem{DT,R}) -> Int
```

Compute the Frobenius–Schur indicator of the irreducible representation $\mathrm{V}(\lambda)$:

- `1` if $\mathrm{V}(\lambda)$ is **real** (orthogonal): admits an invariant symmetric bilinear form
- `-1` if $\mathrm{V}(\lambda)$ is **pseudoreal** (quaternionic/symplectic): admits an invariant skew-symmetric bilinear form
- `0` if $\mathrm{V}(\lambda)$ is **complex**: not self-dual

For a self-dual representation, the indicator is $(-1)^{\langle \rho^\vee, \lambda \rangle}$ where $\langle \rho^\vee, \lambda \rangle = \sum_{i>0} \langle \alpha_i^\vee, \lambda \rangle$ is twice the dual Weyl vector and $\langle \alpha_i^\vee, \lambda \rangle$ is the natural pairing.

Examples

```
julia> using Semisimple

julia> frobenius_schur_indicator(fundamental_weight(TypeA{2}, 1))
0

julia> frobenius_schur_indicator(WeightLatticeElem(TypeA{2}, [1, 1]))
1

julia> frobenius_schur_indicator(fundamental_weight(TypeC{2}, 1))
-1
```

`Semisimple.adjoint_representation` – Function.

```
adjoint_representation(::Type{DT}) -> WeylCharacter{DT}
```

Return the character of the adjoint representation of the Lie algebra of type DT. The highest weight is the highest root θ.

For a simple type, the adjoint is a single irreducible. For a product type DT = $T_1 \times T_2 \times \cdots$, the adjoint is the direct sum of the factor adjoints, each embedded in the product weight space with zeros in the other factor slots.

Examples

```
julia> using Semisimple

julia> degree(adjoint_representation(TypeA{3}))
15

julia> degree(adjoint_representation(TypeE{8}))
248

julia> PDT = ProductDynkinType{Tuple{TypeA{2},TypeB{2}}};

julia> degree(adjoint_representation(PDT))
18

julia> adjoint_representation(PDT)
A2 × B2(1, 1, 0, 0) + A2 × B2(0, 0, 0, 2)
```

Part VII

Implementation details

Chapter 6

Implementation details

This page covers performance considerations, caching mechanisms, precompilation, and other implementation details of Semisimple.jl.

6.1 Caching

Semisimple.jl uses several internal caches to avoid recomputing expensive results. Understanding these caches is important for benchmarking and memory management.

Available caches

Semisimple.jl maintains ten internal caches. Six are unbounded `Dict` caches for small singletons and lookup tables; four are bounded LRU caches (from LRUCache.jl) whose total memory budget is configurable at runtime via `configure_caches!`.

Cache	Variable	Type	Purpose
Root system	`Semisimple._root_system_cache`	`Dict`	Singleton `RootSystem` instances per Dynkin type
Positive roots set	`Semisimple._positive_roots_set_cache`	`Dict`	Fast `is_positive_root` lookup sets
Longest Weyl element	`Semisimple._longest_element_cache`	`Dict`	Cached longest element w_0 per Dynkin type
Coset representatives	`Semisimple._coset_reps_cache`	`Dict`	Weyl orbit coset reps for exceptional types
Dominant character (type)	`Semisimple._dominant_character_type_cache`	`Dict`	Type-level Freudenthal intermediates
Weyl dimension data	`Semisimple._weyl_dimension_data_cache`	`Dict`	Dimension formula denominator and scaled roots
Dominant character	`Semisimple._dominant_character_cache`	LRU	Dominant weight multiplicities from Freudenthal's formula
Tensor product	`Semisimple._tensor_cache`	LRU	Tensor product decompositions
Symmetric power	`Semisimple._symmetric_power_cache`	LRU	Symmetric power decompositions
Exterior power	`Semisimple._exterior_power_cache`	LRU	Exterior power decompositions

The six `Dict` caches are unbounded and persist for the lifetime of the Julia session. The four LRU caches have a configurable memory budget (default: 25 % of system RAM, minimum 256 MiB) and automatically evict least-recently-used entries when the budget is exceeded.

Why the dominant character cache matters

The dominant character cache is the main performance lever for downstream operations. Tensor products, symmetric/exterior powers, and plethysms call `dominant_character` repeatedly for the same highest weights.

Inspecting caches

Use `cache_info` to get a snapshot of cache occupancy:

```
using Semisimple

# Snapshot before any work
info = cache_info()
println("Tensor cache: ", info.tensor.length, " entries (max ", info.tensor.maxsize, " bytes)")

# Populate some caches by doing computations
ω₁ = fundamental_weight(TypeE{8}, 1)
freudenthal_formula(ω₁)
tensor_product(ω₁, ω₁)

# Snapshot after
info = cache_info()
println("Dominant character cache: ", info.dominant_character.length, " entries")
println("Tensor cache: ", info.tensor.length, " entries")
```

Clearing caches

Use `clear_caches!` (or its alias `clear_all_caches!`) to empty every cache at once:

```
using Semisimple

# Do some computations
ω₁ = fundamental_weight(TypeA{2}, 1)
tensor_product(ω₁, ω₁)
freudenthal_formula(ω₁)
symmetric_power(ω₁, 3)

# Clear everything
clear_caches!()
```

This is particularly useful for:

- **Benchmarking cold-start performance** — measure how long operations take without cached results
- **Memory management** — free memory after large computations (e.g., after computing many E_8 tensor products)
- **Reproducible testing** — ensure tests start from a clean state

Individual cache variables have underscored names and are internal implementation details. Prefer the public `clear_caches!` and `configure_caches!` APIs.

Configuring cache budgets

Use `configure_caches!` to resize the LRU caches at runtime. The `budget` (in bytes) controls the total memory envelope; the four fraction arguments determine how it is divided:

```
using Semisimple

# Give caches 512 MiB total
configure_caches!(budget = 512 * 1024^2)

# Custom split: 50 % tensor, 30 % dominant, 10 % each for Sym/□
configure_caches!(
  budget = 512 * 1024^2,
  dominant_frac = 0.30,
  tensor_frac = 0.50,
  sym_power_frac = 0.10,
  ext_power_frac = 0.10,
)
```

Default fractions: dominant 30 %, tensor 40 %, symmetric 15 %, exterior 15 %. The default total budget is 25 % of system RAM (minimum 256 MiB). These defaults can also be set persistently via Julia's `Preferences.jl` (keys: `cache_budget`, `dominant_frac`, `tensor_frac`, `sym_power_frac`, `ext_power_frac`).

Cache invalidation

Caches are **never invalidated by code changes** — all cached functions are pure (same inputs always produce same outputs). However, cached entries can disappear in three ways:

- You explicitly clear a cache (via `clear_caches!` or `empty!(...)`)
- An LRU cache evicts least-recently-used entries when its memory budget is exceeded
- Your Julia session ends

Automatic eviction only affects the four bounded LRU caches. The six unbounded `Dict` caches persist until cleared or session end.

This design is safe because:

- Dynkin types are immutable compile-time constants
- Weights are immutable `SVector` objects
- All cached functions are pure — re-computing an evicted entry always gives the same result

6.2 Precompilation

Semisimple.jl precompiles many commonly-used methods to reduce first-call latency. When you load the package with `using Semisimple`, the precompilation work has already been done.

What gets precompiled

The package precompiles the following operations for all simple Dynkin types up to rank 9 (plus the exceptional types):

Dynkin types precompiled:

- `TypeA{1}` through `TypeA{9}`
- `TypeB{2}` through `TypeB{9}`
- `TypeC{2}` through `TypeC{9}`
- `TypeD{4}` through `TypeD{9}`
- `TypeE{6}`, `TypeE{7}`, `TypeE{8}`
- `TypeF4`
- `TypeG2`

Operations precompiled:

- `cartan_matrix`, `cartan_symmetrizer`, `cartan_bilinear_form`, `cartan_matrix_inverse`
- `_make_root_system` (internal root system construction)
- `_weyl_denominator`, `_weyl_dim_scaled_roots` (Weyl dimension formula internals)
- `degree` (representation dimension)
- `conjugate_dominant_weight` (dominant weight conjugation)
- `conjugate_dominant_weight_with_length` (Borel–Weil–Bott hot path)
- `weyl_orbit` (Weyl orbit generation)
- Weyl group actions (* operator for roots and weights with Weyl elements)
- `freudenthal_formula` (weight multiplicities)
- `dot_reduce` (weight normalization)
- `lr_tensor_product` (Littlewood–Richardson rule for Type A)

Why precompilation matters

Without precompilation, the first call to a method triggers just-in-time (JIT) compilation, which can take hundreds of milliseconds. With precompilation, these methods are ready to use immediately:

```
using Semisimple

# First call is fast due to precompilation
@time degree(fundamental_weight(TypeE{8}, 1))  # ~0.001s

# Without precompilation, this would take ~0.5s for the first call
```

What is NOT precompiled

Operations involving:

- **Product Dynkin types** (e.g., `ProductDynkinType{Tuple{TypeA{2}, TypeB{3}}}`)
- **Rank ≥ 10 simple types** (e.g., `TypeA{15}`)
- **Specific high-dimensional computations** (e.g., `tensor_product(ω₇, ω₇)` for E_8)

These will experience first-call latency but will be fast on subsequent calls (after JIT compilation).

6.3 Performance characteristics

Compile-time vs. run-time

Semisimple.jl leverages Julia's type system and `@generated` functions to move many computations to compile time:

Compile-Time (Type-Level)	Run-Time
Dynkin type classification	Weight coordinate values
Rank of Dynkin type	Weight lattice arithmetic
Cartan matrix entries	Weyl orbit traversal
Root system enumeration	Freudenthal recursion
Weyl denominator product	Character multiplication

This means that `cartan_matrix(TypeE{8})` produces a compile-time constant `SMatrix` that is embedded directly into your compiled code — there's no matrix allocation at runtime.

Memory usage

Operation	Memory Footprint	Notes
`RootSystem{TypeE{8}}`	~15 KB	Singleton, cached per type
`WeightLatticeElem`	8R bytes	R = rank; stored as `SVector{R,Int}`
`WeylGroupElem`	~40 + L bytes	Word stored as `Vector{UInt8}`; L = word length
`WeylCharacter`	~24 + 40N bytes	N = number of terms in the character
Freudenthal cache (E_8 adjoint)	~40 KB	3,875 weight multiplicities

For large-scale computations (e.g., thousands of E_8 tensor products), the character-related LRU caches will automatically evict old entries once their memory budget is reached. Use `configure_caches!` to increase the budget, or `clear_caches!` to free memory immediately.

Asymptotic complexity

For reproducible performance measurements, see the benchmark scripts in `benchmark/`.

6.4 Type stability

Semisimple.jl is designed for **complete type stability**:

Operation	Time Complexity	Notes
`degree(λ)`	$O(N^2)$	N = number of positive roots
`freudenthal_formula(λ)`	$O(M \cdot N)$	M =
`tensor_product(λ, μ)` (BK)	$O(M \cdot W \cdot d)$	W = Weyl group order, d = dim V(smaller weight)
`tensor_product(λ, μ)` (LR, Type A)	$O(n^3)$	n = max(
`symmetric_power(λ, k)`	$O(k^2 \cdot T)$	T = cost of one tensor product
`weyl_orbit(λ)`	$O(W \cdot R \cdot R)$	W = orbit size ≤ Weyl order, R = rank

```
using Semisimple

ω₁ = fundamental_weight(TypeE{8}, 1)
typeof(ω₁)  # WeightLatticeElem{TypeE{8}, 8} — concrete type

ch = freudenthal_formula(ω₁)
typeof(ch)  # Dict{SVector{8, Int64}, Int64} — concrete type

result = tensor_product(ω₁, ω₁)
typeof(result)  # WeylCharacter{TypeE{8}, 8} — concrete type
```

All public APIs return concrete types, enabling aggressive compiler optimizations. There are **no type instabilities** in hot paths.

6.5 Numerical precision

All computations use **exact integer arithmetic** — there are no floating-point operations:

- Weights are `SVector{R, Int}` — exact integer vectors
- Multiplicities are `Int` — exact integer counts
- Dimensions are computed exactly (Weyl dimension formula uses `BigInt` for large products)
- Inner products use scaled integer forms to avoid division

This means:

- **No numerical stability concerns** — safe for arbitrarily large representations
- **Overflow protection** — dimension computations automatically promote to `BigInt` when needed

Example:

```
julia> ω7 = fundamental_weight(TypeE{8}, 7);

julia> degree(ω7)  # 147,250 × 2⁶⁰ — too large for Int64
170141183460469137866240
```

```
julia> typeof(degree(ω7))
BigInt
```

6.6 Thread safety

Some caches are not thread-safe

The small `Dict` singleton/type-data caches are protected by locks where they are populated through public APIs. The bounded LRU character caches are not synchronized, so concurrent cache-populating calls such as `dominant_character`, `tensor_product`, `symmetric_power`, or `exterior_power` can race.

Safe: Using Semisimple.jl from a single thread (the default)

Safe: Read-only operations from multiple threads after warming up caches

Unsafe: Calling LRU-cache-populating representation operations from multiple threads simultaneously

If you need parallel computation, populate caches in a single-threaded warm-up phase, then perform read-only operations in parallel.

6.7 Comparison with LiE

Semisimple.jl reimplements many algorithms from the LiE computer algebra system. Key differences:

Aspect	LiE (C)	Semisimple.jl (Julia)
Language	C (CWEB literate programming)	Julia (pure Julia)
Type system	Runtime `group` structs	Compile-time Dynkin type parameters
Cartan matrices	Runtime matrix allocation	Compile-time `SMatrix` constants
Caching	Permanent "long-life" objects	Bounded LRU caches + `Dict` singletons
Hot performance	Fast (compiled C)	Fast (JIT-compiled, with caching)
Cold performance	Instant (no compilation)	Slow first call (JIT overhead)

For hot operations (cached, precompiled), Semisimple.jl matches or exceeds LiE's performance. For cold operations, LiE is faster due to no JIT compilation delay.

6.8 Implementation philosophy

Semisimple.jl follows these design principles:

1. **Type-level dispatch** — Use Julia's type system to specialize code for each Dynkin type
2. **Compile-time constants** — Leverage `@generated` functions to embed mathematical data
3. **Immutability** — All core types are immutable for thread safety and optimization
4. **Caching** — Trade memory for speed by memoizing expensive computations
5. **Minimal dependencies** — StaticArrays.jl, LRUCache.jl, PrecompileTools.jl, Preferences.jl, and LinearAlgebra (stdlib)
6. **Pure Julia** — No C/Fortran, enabling introspection and compilation to other targets

These principles enable aggressive compiler optimizations while maintaining mathematical rigor.

Weyl orbit traversal

Weyl orbits are computed by the internal module `Weylloop.jl`, which implements LiE-style systematic orbit traversal. Rather than a hash-set BFS that scales with orbit size, it converts weight coordinates to the **ε-basis** where classical Weyl subgroups act as permutations (type A) or permutations + sign flips (types B/C/D). Orbits are enumerated via lexicographic permutation generation and Gray-code sign flips, eliminating the $O(|\text{orbit}|)$ hash-set overhead that would otherwise dominate for large orbits (e.g., E_8 orbits with millions of elements). For exceptional types, precomputed coset representatives reduce the problem to the classical case.

6.9 API reference

`Semisimple.clear_all_caches!` – Function.

```
clear_all_caches!()
```

Clear all internal caches used by Semisimple.jl. Alias for `clear_caches!`.

Examples

```
julia> using Semisimple

julia> clear_all_caches!()
```

`Semisimple.clear_caches!` – Function.

```
clear_caches!()
```

Empty every internal cache in Semisimple.jl (both bounded Dict caches and LRU caches).

Examples

```
julia> using Semisimple

julia> ω1 = fundamental_weight(TypeA{2}, 1);

julia> tensor_product(ω1, ω1);  # populates caches

julia> clear_caches!()
```

`Semisimple.configure_caches!` – Function.

```
configure_caches!(; budget=nothing, dominant_frac=nothing, tensor_frac=nothing,
                    sym_power_frac=nothing, ext_power_frac=nothing)
```

Resize the LRU caches at runtime. Unspecified keyword arguments retain their current values. The `budget` is in **bytes** and controls the total memory envelope; the four fraction arguments determine how it is divided among the caches (they need not sum to 1 — each cache is sized independently as `budget * frac`).

Examples

```
julia> using Semisimple

julia> configure_caches!(budget = 512 * 1024^2)  # 512 MiB total
```

`Semisimple.cache_info` – Function.

```
cache_info() -> NamedTuple
```

Return a snapshot of the current cache occupancy. Each entry is a `NamedTuple` with fields `length` (number of entries) and `maxsize` (capacity in bytes).

Examples

```
julia> using Semisimple

julia> info = cache_info();

julia> info.tensor.length >= 0
true
```

6.10 Internals reference

These are internal functions not part of the public API. They are documented here for contributors and advanced users.

Root system internals

`Semisimple._root_system_cache` – Constant.

```
RootSystem(::Type{DT}) -> RootSystem{DT,R,N}
```

Return the root system for Dynkin type `DT`. A single instance is cached per Dynkin type, with small ranks using fully generated literals and larger ranks using a compact runtime builder.

Examples

```
julia> using Semisimple

julia> RootSystem(TypeA{2}) === RootSystem(TypeA{2})
true
```

`Semisimple._compute_positive_roots_and_reflections` – Function.

Compute positive roots, coroots, and the reflection table from a Cartan matrix.

This uses the standard algorithm: start with simple roots and iteratively apply simple reflections to discover new positive roots.

Returns:

- `pos_roots::Vector{SVector{R,Int}}` — positive roots in simple root coordinates
- `pos_coroots::Vector{SVector{R,Int}}` — positive coroots
- `refl::Matrix{UInt}` — reflection table

`Semisimple._make_root_system_runtime` – Function.

```
_make_root_system_runtime(::Type{DT}) -> RootSystem{DT,R,N}
```

Compact runtime builder for root systems. This is used directly for higher ranks and via a small generated wrapper for larger mid-rank types to avoid emitting enormous tuple literals into method bodies.

Weyl group internals

`Semisimple._weyl_denominator` – Function.

```
_weyl_denominator(::Type{DT}) -> BigInt
```

Compute the Weyl dimension denominator ∏_{α>0} ⟨ρ, α⟩.

`Semisimple._weyl_dim_scaled_roots` – Function.

```
_weyl_dim_scaled_roots(::Type{DT}) -> NTuple{N, SVector{R,Int}}
```

Return the symmetrizer-scaled positive root vectors `d .* α` for Dynkin type `DT`.

`Semisimple._explain_rmul` – Function.

Internal: determines what right-multiplication by `s` does to word `x`.

Returns `(insert::Bool, position::Int, letter::UInt8)`:

- if `insert=true`: insert `letter` at `position`
- if `insert=false`: delete the element at `position`

`Semisimple.weylloop` – Function.

```
weylloop(action!, ::Type{DT}, v::AbstractVector{<:Integer})
```

Call `action!(w)` once for each weight `w` in the Weyl orbit of `v`, where `v` and `w` are in the fundamental weight (ω) basis.

Uses LiE's ε-basis algorithm: converts to ε-coordinates where a classical subgroup acts by permutations (type A) or permutations + sign changes (types B/C/D), then enumerates orbits systematically via lexicographic permutation generation and Gray-code sign flips — with no hash set or BFS.

For exceptional types, coset representatives W / W_classical are precomputed as matrices.

All transforms dispatch on the Dynkin type, enabling the compiler to inline them and unroll fixed-size loops. The hot per-orbit workspace uses stack-allocated `MVector` from StaticArrays; the deduplicated suborbit representatives still live in a small heap vector because their count depends on the Dynkin type and the input weight.

`action!` receives a mutable workspace vector; it must NOT hold a reference to this vector across calls (copy if needed).

`Semisimple._positive_roots_runtime` – Function.

```
_positive_roots_runtime(C, R) -> Vector{Vector{Int}}
```

Compute positive roots using plain arrays (no StaticArrays).

Cache internals

`Semisimple._apply_cache_preferences!` – Function.

```
_apply_cache_preferences!()
```

Read Preferences-based cache settings and resize caches accordingly. Called from `__init__()` so that user preferences take effect on load.

Recognised preferences (set via `Preferences.set_preferences!`):

Key	Type	Default
`cache_budget`	`Int`	25% of RAM (≥256 MiB)
`dominant_frac`	`Float64`	0.30
`tensor_frac`	`Float64`	0.40
`sym_power_frac`	`Float64`	0.15
`ext_power_frac`	`Float64`	0.15

Character internals

`Semisimple.dot_reduce` – Function.

```
dot_reduce(λ::WeightLatticeElem{DT,R}) -> Tuple{Int, WeightLatticeElem{DT,R}}
```

Compute the "dot-action reduction" of λ:

Return `(ε, μ)` where:

- μ is the dominant weight such that `w · λ = μ` under the dot action `w · λ = w(λ + ρ) - ρ` for some Weyl group element `w`
- `ε = (-1)^{□(w)}` is the sign of `w`, or `ε = 0` if `λ + ρ` is singular (lies on a Weyl chamber wall)

This is the key ingredient in the Brauer–Klimyk algorithm.

`Semisimple.brauer_klimyk` – Function.

```
brauer_klimyk(char::Dict{SVector{R,Int}, <:Integer}, μ::WeightLatticeElem{DT,R}) ->
↪  WeylCharacter{DT,R}
```

Tensor the representation with weight multiplicities `char` (as from `freudenthal_formula`) with the irreducible representation $\mathrm{V}()$, using the Brauer–Klimyk formula:

$$\mathrm{V} \otimes \mathrm{V}() = \sum_{\text{weights of } \mathrm{V}} m() \cdot (\,+\,) \cdot \mathrm{V}((\,+\,))$$

where `(ε, ν) = dot_reduce(μ + λ)`. Accepts `BigInt`-valued multiplicities (as returned by `freudenthal_formula`); the resulting irreducible multiplicities are stored as `Int` in the returned `WeylCharacter`, so this will throw `InexactError` if any of them exceeds `typemax(Int64)`.

`Semisimple._brauer_klimyk_dominant` – Function.

```
_brauer_klimyk_dominant(dom_char::Dict{SVector{R,Int}, <:Integer}, μ::WeightLatticeElem{DT,R})
↪  -> WeylCharacter{DT,R}
```

Like `brauer_klimyk`, but takes a **dominant-only** character dict (as returned by `dominant_character`) and expands Weyl orbits on-the-fly using `weylloop`. This avoids materializing the full weight system and eliminates hash-set overhead for large orbits.

The Brauer–Klimyk formula is: $\mathrm{V} \otimes \mathrm{V}() = \sum_{\text{dom. wts } d} m(d) \sum_{w \in W \cdot d} (w(d) + \,) \cdot \mathrm{V}((w(d) + \,))$

Accepts `BigInt`-valued multiplicities; the resulting irreducible multiplicities are stored as `Int` in the returned `WeylCharacter` and will throw `InexactError` if any of them exceeds `typemax(Int64)`.

`Semisimple._vdecomp` – Function.

```
_vdecomp(::Type{DT}, dom_char::Dict{SVector{R,Int},<:Integer}) -> WeylCharacter{DT,R}
```

Virtual decomposition: given a character as a dict of weight multiplicities (dominant weights only, as produced by Adams operators), decompose into irreducibles using the Weyl orbit / alternating-dominant method.

This is the analogue of LiE's `Vdecomp` function.

`Semisimple._tensor_characters` – Function.

```
_tensor_characters(V::WeylCharacter{DT,R}, W::WeylCharacter{DT,R}) -> WeylCharacter{DT,R}
```

Tensor product of two virtual characters (each decomposed into irreducibles).

Littlewood–Richardson internals (type A)

`Semisimple._weight_to_partition` – Function.

```
_weight_to_partition(λ::WeightLatticeElem{TypeA{N},N}) -> Vector{Int}
```

Convert a dominant weight in fundamental weight coordinates to a partition with $N+1$ parts (for GL_{N+1}).

For A_N, the dominant weight $= (_1, \ldots, _N)$ in the fundamental weight basis corresponds to the partition $= (_1 \ldots _N 0)$ where $_i = _i + _{i+1} + \cdots + _N$ (partial sums from right to left), with $_{N+1} = 0$.

`Semisimple._partition_to_weight` – Function.

```
_partition_to_weight(::Type{TypeA{N}}, p::Vector{Int}) -> WeightLatticeElem{TypeA{N},N}
```

Convert a partition back to a dominant weight in the fundamental weight basis for SL_{N+1}. First reduces the partition by subtracting the minimum part (to pass from GL to SL), then computes successive differences: $_i = _i - _{i+1}$.

`Semisimple._lr_coefficients` – Function.

```
_lr_coefficients(α::Vector{<:Integer}, β::Vector{<:Integer}, n::Integer) -> Dict{Vector{Int},
↪  Int}
```

Compute all Littlewood–Richardson coefficients c for partitions α and β, where partitions have at most n parts.

Returns a dictionary mapping each partition ν (as `Vector{Int}`) to the LR coefficient c.

The algorithm fills the skew shape $/$ with content β row by row, enforcing:

1. **Semistandard**: entries weakly increase along rows, strictly increase down columns.
2. **Ballot (lattice word) condition**: reading the filling right-to-left, top-to-bottom, at every prefix the count of j ≥ count of j+1.

We enumerate valid fillings recursively row by row, which implicitly determines the partition ν.

`Semisimple._n_tableaux` – Function.

```
_n_tableaux(λ::Vector{<:Integer}, l::Integer) -> BigInt
```

Compute the number of standard Young tableaux of shape using the hook-length formula:

$f = \frac{n!}{\prod_{\text{boxes}} h(b)}$

`l` is the number of (nonzero) rows.

Plethysm internals (Murnaghan–Nakayama)

`Semisimple._partitions` – Function.

```
_partitions(n::Integer) -> Vector{Vector{Int}}
```

Generate all partitions of n in decreasing order.

`Semisimple._classord` – Function.

```
_classord(κ::Vector{Int}) -> BigInt
```

Compute the size of the conjugacy class in S_n corresponding to cycle type :

$|\mathrm{Cl}()| = \frac{n!}{\prod_{k>0} k^{c_k()} \cdot c_k()!}$

where $c_k()$ is the number of parts of equal to k. The parts of κ must be in weakly decreasing order.

`Semisimple._mn_char_val` – Function.

```
_mn_char_val(λ::Vector{Int}, μ::Vector{Int}) -> BigInt
```

Compute the irreducible S_n-character value $()$ using the Murnaghan–Nakayama rule.

Both λ and μ are partitions of n with parts in weakly decreasing order.

The algorithm represents the Young diagram of as a Maya diagram (edge sequence of horizontal/vertical steps), then recursively removes rim hooks of the sizes given by the parts of (largest first).

`Semisimple._mn_recurse!` – Function.

Recursive Murnaghan–Nakayama: remove rim hooks of sizes μ[i], μ[i+1], ... from the Maya diagram edge, tracking leg-length parity in k.