```julia
# Superselection sectors (quantum numbers):
# for defining graded vector spaces and invariant subspaces of tensor products
#==============================================================================#
"""
    abstract type Sector end

Abstract type for representing the (isomorphism classes of) simple objects in
(unitary
and pivotal) (pre-)fusion categories, e.g. the irreducible representations of a
finite or
compact group.

Every new `G<:Sector` should implement the following methods:
*   `one(::Type{G})`: unit element of `G`
*   `conj(a::G)`: ``ā``, conjugate or dual label of ``a``
*   `⊗(a::G, b::G)`: iterable with unique fusion outputs of ``a ⊗ b``
    (i.e. don't repeat in case of multiplicities)
*   `Nsymbol(a::G, b::G, c::G)`: number of times `c` appears in `a ⊗ b`, i.e. the
    multiplicity
*   `FusionStyle(::Type{G})`: `Abelian()`, `SimpleNonAbelian()` or
    `DegenerateNonAbelian()`
*   `BraidingStyle(::Type{G})`: `Bosonic()`, `Fermionic()`, `Anyonic()`, ...
*   `Fsymbol(a::G, b::G, c::G, d::G, e::G, f::G)`: F-symbol: scalar (in case of
    `Abelian`/`SimpleNonAbelian`) or matrix (in case of `DegenerateNonAbelian`)
*   `Rsymbol(a::G, b::G, c::G)`: R-symbol: scalar (in case of
    `Abelian`/`SimpleNonAbelian`) or matrix (in case of `DegenerateNonAbelian`)
and optionally
*   `dim(a::G)`: quantum dimension of sector `a`
*   `frobeniusschur(a::G)`: Frobenius-Schur indicator of `a`
*   `Bsymbol(a::G, b::G, c::G)`: B-symbol: scalar (in case of
    `Abelian`/`SimpleNonAbelian`) or matrix (in case of `DegenerateNonAbelian`)
*   `twist(a::G)` -> twist of sector `a`
and optionally, if `FusionStyle(G) isa DegenerateNonAbelian`
*   `vertex_ind2label(i::Int, a::G, b::G, c::G)` -> a custom label for the `i`th
copy of
    `c` appearing in `a ⊗ b`

Furthermore, `iterate` and `Base.IteratorSize` should be made to work for the
singleton type
[`SectorValues{G}`](@ref).
"""
abstract type Sector end

# iterator over the values in the sector
"""
    struct SectorValues{G<:Sector}

Singleton type to represent an iterator over the possible values of type `G`,
whose instance is obtained as `values(G)`. For a new `G::Sector`, the following
should be defined
*   `Base.iterate(::SectorValues{G}[, state])`: iterate over the values
*   `Base.IteratorSize(::Type{SectorValues{G}})`: `HasLenght()`, `SizeUnkown()`
    or `IsInfinite()` depending on whether the number of values of type `G` is
finite (and sufficiently small) or infinite; for a large number of values,
```

```
`SizeUnknown()` is recommend because this will trigger the use of
`GenericRepresentationSpace`.
If `IteratorSize(G) == HasLength()`, also the following must be implemented:
*   `Base.length(::SectorValues{G})`: the number of different values
*   `Base.getindex(::SectorValues{G}, i::Int)`: a mapping between an index `i` and
an
    instance of `G`
*   `findindex(::SectorValues{G}, c::G)`: reverse mapping between a value `c::G`
and an
    index `i::Integer ∈ 1:length(values(G))`
"""
struct SectorValues{G<:Sector} end
Base.IteratorEltype(::Type{<:SectorValues}) = HasEltype()
Base.eltype(::Type{SectorValues{G}}) where {G<:Sector} = G
Base.values(::Type{G}) where {G<:Sector} = SectorValues{G}()

# Define a sector for ungraded vector spaces
struct Trivial <: Sector
end
Base.show(io::IO, ::Trivial) = print(io, "Trivial()")
Base.show(io::IO, ::Type{Trivial}) = print(io, "Trivial")

"""
    one(::Sector) -> Sector
    one(::Type{<:Sector}) -> Sector

Return the unit element within this type of sector.
"""
Base.one(a::Sector) = one(typeof(a))
Base.one(::Type{Trivial}) = Trivial()

"""
    dual(a::Sector) -> Sector

Return the conjugate label `conj(a)`.
"""
dual(a::Sector) = conj(a)
Base.conj(::Trivial) = Trivial()

"""
    isreal(::Type{<:Sector}) -> Bool

Return whether the topological data (Fsymbol, Rsymbol) of the sector is real or
not (in
which case it is complex).
"""
Base.@pure function Base.isreal(G::Type{<:Sector})
    u = one(G)
    return (eltype(Fsymbol(u,u,u,u,u,u))<:Real) && (eltype(Rsymbol(u,u,u))<:Real)
end
Base.@pure Base.isreal(::Type{Trivial}) = true

Base.isless(::Trivial, ::Trivial) = false
```

```julia
# FusionStyle: the most important aspect of Sector
#---------------------------------------------
"""
    ⊗(a::G, b::G) where {G<:Sector}

Return an iterable of elements of `c::G` that appear in the fusion product `a ⊗ b`.

Note that every element `c` should appear at most once, fusion degeneracies (if
`FusionStyle(G) == DegenerateNonAbelian()`) should be accessed via
`Nsymbol(a,b,c)`.
"""
⊗(::Trivial, ::Trivial) = (Trivial(),)

"""
    Nsymbol(a::G, b::G, c::G) where {G<:Sector} -> Integer

Return an `Integer` representing the number of times `c` appears in the fusion
product
`a ⊗ b`. Could be a `Bool` if `FusionStyle(G) == Abelian()` or
`SimpleNonAbelian()`.
"""
function Nsymbol end
Nsymbol(::Trivial, ::Trivial, ::Trivial) = true

# trait to describe the fusion of superselection sectors
abstract type FusionStyle end
struct Abelian <: FusionStyle
end
abstract type NonAbelian <: FusionStyle end
struct SimpleNonAbelian <: NonAbelian # non-abelian fusion but multiplicity free
end
struct DegenerateNonAbelian <: NonAbelian # non-abelian fusion with multiplicities
end

"""
    FusionStyle(a::Sector) -> ::FusionStyle
    FusionStyle(G::Type{<:Sector}) -> ::FusionStyle

Return the type of fusion behavior of sectors of type G, which can be either
*   `Abelian()`: single fusion output when fusing two sectors;
*   `SimpleNonAbelian()`: multiple outputs, but every output occurs at most one,
    also known as multiplicity free (e.g. irreps of ``SU(2)``);
*   `DegenerateNonAbelian()`: multiple outputs that can occur more than once (e.g.
irreps
    of ``SU(3)``).
There is an abstract supertype `NonAbelian` of which both `SimpleNonAbelian` and
`DegenerateNonAbelian` are subtypes.
"""
FusionStyle(a::Sector) = FusionStyle(typeof(a))
FusionStyle(::Type{Trivial}) = Abelian()

# NOTE: the following inline is extremely important for performance, especially
# in the case of Abelian, because ⊗(...) is computed very often
@inline function ⊗(a::G, b::G, c::G, rest::Vararg{G}) where {G<:Sector}
```

```julia
    if FusionStyle(G) isa Abelian
        return a ⊗ first(⊗(b, c, rest...))
    else
        s = Set{G}()
        for d in ⊗(b, c, rest...)
            for e in a ⊗ d
                push!(s, e)
            end
        end
        return s
    end
end

"""
    Fsymbol(a::G, b::G, c::G, d::G, e::G, f::G) where {G<:Sector}

Return the F-symbol ``F^{abc}_d`` that associates the two different fusion orders
of sectors
`a`, `b` and `c` into an ouput sector `d`, using either an intermediate sector ``a
⊗ b → e``
or ``b ⊗ c → f``:
```
a-<-μ-<-e-<-ν-<-d   a-<-λ-<-d
    ∨   ∨   -> Fsymbol(a,b,c,d,e,f)[μ,ν,κ,λ]   ∨
    b   c   b-<-κ
                                                         ∨
                                                         c
```
If `FusionStyle(G)` is `Abelian` or `SimpleNonAbelian`, the F-symbol is a number.
Otherwise
it is a rank 4 array of size
`(Nsymbol(a,b,e), Nsymbol(e,c,d), Nsymbol(b,c,f), Nsymbol(a,f,d))`.
"""
function Fsymbol end
Fsymbol(::Trivial, ::Trivial, ::Trivial, ::Trivial, ::Trivial, ::Trivial) = 1

"""
    Rsymbol(a::G, b::G, c::G) where {G<:Sector}

Returns the R-symbol ``R^{ab}_c`` that maps between ``a ⊗ b → c`` and ``b ⊗ a →
c`` as in
```
a -<-μ-<- c   b -<-ν-<- c
     ∨   -> Rsymbol(a,b,c)[μ,ν]   ∧
     b   a
```
If `FusionStyle(G)` is `Abelian()` or `SimpleNonAbelian()`, the R-symbol is a
number.
Otherwise it is a square matrix with row and column size `Nsymbol(a,b,c) ==
Nsymbol(b,a,c)`.
"""
function Rsymbol end
Rsymbol(::Trivial, ::Trivial, ::Trivial) = 1
```

```
# If a G::Sector with `fusion(G) == DegenerateNonAbelian` fusion wants to have
  custom vertex
# labels, a specialized method for `vertindex2label` should be added
"""
    vertex_ind2label(i::Int, a::G, b::G, c::G) where {G<:Sector}

Convert the index i of the fusion vertex (a,b)->c into a label. For
`FusionStyle(G) == Abelian()` or `FusionStyle(G) == NonAbelian()`, where every
fusion
output occurs only once and `i == 1`, the default is to suppress vertex labels by
setting
them equal to `nothing`. For `FusionStyle(G) == DegenerateNonAbelian()`, the
default is to
just use `i`, unless a specialized method is provided.
"""
vertex_ind2label(i::Int, s1::G, s2::G, sout::G) where {G<:Sector}=
    _ind2label(FusionStyle(G), i::Int, s1::G, s2::G, sout::G)
_ind2label(::Abelian, i, s1, s2, sout) = nothing
_ind2label(::SimpleNonAbelian, i, s1, s2, sout) = nothing
_ind2label(::DegenerateNonAbelian, i, s1, s2, sout) = i

"""
    vertex_labeltype(G::Type{<:Sector}) -> Type

Return the type of labels for the fusion vertices of sectors of type `G`.
"""
Base.@pure vertex_labeltype(G::Type{<:Sector}) =
    typeof(vertex_ind2label(1, one(G), one(G), one(G)))

# combine fusion properties of tensor products of sectors
Base.:&(f::F, ::F) where {F<:FusionStyle} = f
Base.:&(f1::FusionStyle, f2::FusionStyle) = f2 & f1

Base.:&(::SimpleNonAbelian, ::Abelian) = SimpleNonAbelian()
Base.:&(::DegenerateNonAbelian, ::Abelian) = DegenerateNonAbelian()
Base.:&(::DegenerateNonAbelian, ::SimpleNonAbelian) = DegenerateNonAbelian()

# properties that can be determined in terms of the F symbol
# TODO: find mechanism for returning these numbers with custom type
  T<:AbstractFloat
"""
    dim(a::Sector)

Return the (quantum) dimension of the sector `a`.
"""
function dim(a::Sector)
    if FusionStyle(a) isa Abelian
        1
    elseif FusionStyle(a) isa SimpleNonAbelian
        abs(1/Fsymbol(a,conj(a),a,a,one(a),one(a)))
    else
        abs(1/Fsymbol(a,conj(a),a,a,one(a),one(a))[1])
    end
end
```

```julia
"""
    frobeniusschur(a::Sector)

Return the Frobenius-Schur indicator of a sector `a`.
"""
function frobeniusschur(a::Sector)
    if FusionStyle(a) isa Abelian || FusionStyle(a) isa SimpleNonAbelian
        sign(Fsymbol(a,conj(a),a,a,one(a),one(a)))
    else
        sign(Fsymbol(a,conj(a),a,a,one(a),one(a))[1])
    end
end

"""
    twist(a::Sector)

Return the twist of a sector `a`
"""
function twist(a::Sector)
    if FusionStyle(a) isa Abelian || FusionStyle(a) isa SimpleNonAbelian
        θ = sum(dim(b)/dim(a)*Rsymbol(a,a,b) for b in a ⊗ a)
    else
        # TODO: is this correct?
        # θ = sum(dim(b)/dim(a)*tr(Rsymbol(a,a,b)) for b in a ⊗ a)
        throw(MethodError(twist, (a,)))
    end
    return θ
end



"""
    Bsymbol(a::G, b::G, c::G) where {G<:Sector}

Return the value of ``B^{ab}_c`` which appears in transforming a splitting vertex
into a fusion vertex using the transformation
```
a -<-μ-<- c   a -<-ν-<- c
     ∨   -> √(dim(c)/dim(a)) * Bsymbol(a,b,c)[μ,ν]   ∧
     b   dual(b)
```
If `FusionStyle(G)` is `Abelian()` or `SimpleNonAbelian()`, the B-symbol is a
number.
Otherwise it is a square matrix with row and column size
`Nsymbol(a, b, c) == Nsymbol(c, dual(b), a)`.
"""
function Bsymbol(a::G, b::G, c::G) where {G<:Sector}
    if FusionStyle(G) isa Abelian || FusionStyle(G) isa SimpleNonAbelian
        sqrt(dim(a)*dim(b)/dim(c))*Fsymbol(a, b, dual(b), a, c, one(a))
    else
        reshape(sqrt(dim(a)*dim(b)/dim(c))*Fsymbol(a,b,dual(b),a,c,one(a)),
            (Nsymbol(a,b,c), Nsymbol(c,dual(b),a)))
    end
end
```

```julia
# Not necessary
function Asymbol(a::G, b::G, c::G) where {G<:Sector}
    if FusionStyle(G) isa Abelian || FusionStyle(G) isa SimpleNonAbelian

sqrt(dim(a)*dim(b)/dim(c))*conj(frobeniusschur(a)*Fsymbol(dual(a),a,b,b,one(a),c))
    else
        reshape(sqrt(dim(a)*dim(b)/dim(c))*
                conj(frobeniusschur(a)*Fsymbol(dual(a),a,b,b,one(a),c)),
                (Nsymbol(a,b,c), Nsymbol(dual(a),c,b)))
    end
end

# Braiding:
#-------------------------------------------------
# trait to describe type to denote how the elementary spaces in a tensor product
  space
# interact under permutations or actions of the braid group
abstract type BraidingStyle end # generic braiding
abstract type SymmetricBraiding <: BraidingStyle end # symmetric braiding =>
    actions of permutation group are well defined
struct Bosonic <: SymmetricBraiding end # trivial under permutations
struct Fermionic <: SymmetricBraiding end
struct Anyonic <: BraidingStyle end

Base.:&(b::B,::B) where {B<:BraidingStyle} = b
Base.:&(B1::BraidingStyle, B2::BraidingStyle) = B2 & B1

Base.:&(::Bosonic,::Fermionic) = Fermionic()
Base.:&(::Bosonic,::Anyonic) = Anyonic()
Base.:&(::Fermionic,::Anyonic) = Anyonic()

"""
    BraidingStyle(::Sector) -> ::BraidingStyle
    BraidingStyle(G::Type{<:Sector}) -> ::BraidingStyle

Return the type of braiding and twist behavior of sectors of type `G`, which can
be either
*   `Bosonic()`: symmetric braiding with trivial twist (i.e. identity)
*   `Fermionic()`: symmetric braiding with non-trivial twist (squares to identity)
*   `Anyonic()`: general ``R_(a,b)^c`` phase or matrix (depending on
`SimpleNonAbelian` or
    `DegenerateNonAbelian` fusion) and arbitrary twists

Note that `Bosonic` and `Fermionic` are subtypes of `SymmetricBraiding`, which
means that
braids are in fact equivalent to crossings (i.e. braiding twice is an identity:
`Rsymbol(b,a,c)*Rsymbol(a,b,c) = I`) and permutations are uniquely defined.
"""
BraidingStyle(a::Sector) = BraidingStyle(typeof(a))
BraidingStyle(::Type{Trivial}) = Bosonic()

# SectorSet:
#--------------------------------------------------------------------------
```

```julia
# Custum generator to represent sets of sectors with type inference
struct SectorSet{G<:Sector,F,S}
    f::F
    set::S
end
SectorSet{G}(::Type{F}, set::S) where {G<:Sector,F,S} = SectorSet{G,Type{F},S}(F,
set)
SectorSet{G}(f::F, set::S) where {G<:Sector,F,S} = SectorSet{G,F,S}(f, set)
SectorSet{G}(set) where {G<:Sector} = SectorSet{G}(identity, set)

Base.IteratorEltype(::Type{<:SectorSet}) = HasEltype()
Base.IteratorSize(::Type{SectorSet{G,F,S}}) where {G<:Sector,F,S} =
Base.IteratorSize(S)

Base.eltype(::SectorSet{G}) where {G<:Sector} = G
Base.length(s::SectorSet) = length(s.set)
Base.size(s::SectorSet) = size(s.set)

function Base.iterate(s::SectorSet{G}, args...) where {G<:Sector}
    next = iterate(s.set, args...)
    next === nothing && return nothing
    val, state = next
    return convert(G, s.f(val)), state
end

# possible sectors
include("irreps.jl") # irreps of symmetry groups, with bosonic braiding
# include("fermions.jl") # irreps with defined fermionparity and fermionic braiding
include("anyons.jl") # non-group sectors
include("product.jl") # direct product of different sectors
```