```julia
# Sectors corresponding to irreducible representations of compact groups
#==============================================================================#
# Irreps of Abelian groups
#------------------------------------------------------------------------------#
"""
    abstract type Irrep <: Sector end

Abstract supertype for sectors which corresponds to irreps (irreducible
representations) of
groups. As we assume unitary representations, these would be finite groups or
compact Lie
groups. Note that this could also include projective rather than linear
representations.

All irreps have [`BraidingStyle`](@ref) equal to `Bosonic()` and thus trivial
twists.
A fermionic sector can be created using [`Fermion`](@ref).
"""
abstract type Irrep <: Sector end # irreps have integer quantum dimensions
Base.@pure BraidingStyle(::Type{<:Irrep}) = Bosonic()

"""
    abstract type AbelianIrrep <: Irrep end

Abstract supertype for sectors which corresponds to irreps (irreducible
representations) of
abelian groups. They all have `FusionStyle` equal to `Abelian()` and thus trivial
topological data, which is real valued.
"""
abstract type AbelianIrrep <: Irrep end
Base.@pure FusionStyle(::Type{<:AbelianIrrep}) = Abelian()
Base.isreal(::Type{<:AbelianIrrep}) = true

Nsymbol(a::G, b::G, c::G) where {G<:AbelianIrrep} = c == first(a ⊗ b)
Fsymbol(a::G, b::G, c::G, d::G, e::G, f::G) where {G<:AbelianIrrep} =
    Int(Nsymbol(a,b,e)*Nsymbol(e,c,d)*Nsymbol(b,c,f)*Nsymbol(a,f,d))
frobeniusschur(a::AbelianIrrep) = 1
Bsymbol(a::G, b::G, c::G) where {G<:AbelianIrrep} = Float64(Nsymbol(a, b, c))
Rsymbol(a::G, b::G, c::G) where {G<:AbelianIrrep} = Float64(Nsymbol(a, b, c))

fusiontensor(a::G, b::G, c::G, v::Nothing = nothing) where {G<:AbelianIrrep} =
    fill(Float64(Nsymbol(a,b,c)), (1,1,1))

# ZNIrrep: irreps of Z_N are labelled by integers mod N; do we ever want N > 127?
"""
    struct ZNIrrep{N} <: AbelianIrrep
    ZNIrrep(n::Integer)

Represents irreps of the group ``ℤ_N`` for some value of `N<64`. Unicode synonyms
are
available for the cases `N=2,3,4` as `ℤ₂`, `ℤ₃`, `ℤ₄`. Also the name `Parity` can
be used
as synonym for `ℤ₂`. An arbitrary `Integer` `n` can be provided to the
constructor, but
```

```julia
only the value `mod(n, N)` is relevant.
"""
struct ZNIrrep{N} <: AbelianIrrep
    n::Int8
    function ZNIrrep{N}(n::Integer) where {N}
        @assert N < 64
        new{N}(mod(n, N))
    end
end

Base.IteratorSize(::Type{SectorValues{ZNIrrep{N}}}) where N = HasLength()
Base.length(::SectorValues{ZNIrrep{N}}) where N = N
Base.iterate(::SectorValues{ZNIrrep{N}}, i = 0) where N =
    return i == N ? nothing : (ZNIrrep{N}(i), i+1)
Base.getindex(::SectorValues{ZNIrrep{N}}, i::Int) where N =
    1 <= i <= N ? ZNIrrep{N}(i-1) : throw(BoundsError(values(ZNIrrep{N}), i))
findindex(::SectorValues{ZNIrrep{N}}, c::ZNIrrep{N}) where N = c.n + 1

Base.one(::Type{ZNIrrep{N}}) where {N} =ZNIrrep{N}(0)
Base.conj(c::ZNIrrep{N}) where {N} = ZNIrrep{N}(-c.n)
⊗(c1::ZNIrrep{N}, c2::ZNIrrep{N}) where {N} = (ZNIrrep{N}(c1.n+c2.n),)

ZNIrrep{N}(n::Real) where {N} = convert(ZNIrrep{N}, n)
Base.convert(Z::Type{<:ZNIrrep}, n::Real) = Z(convert(Int, n))

const ℤ₂ = ZNIrrep{2}
const ℤ₃ = ZNIrrep{3}
const ℤ₄ = ZNIrrep{4}
const Parity = ZNIrrep{2}
Base.show(io::IO, ::Type{ZNIrrep{2}}) = print(io, "ℤ₂")
Base.show(io::IO, ::Type{ZNIrrep{3}}) = print(io, "ℤ₃")
Base.show(io::IO, ::Type{ZNIrrep{4}}) = print(io, "ℤ₄")
Base.show(io::IO, c::ZNIrrep{2}) =
    get(io, :typeinfo, nothing) === ZNIrrep{2} ? print(io, c.n) : print(io, "ℤ₂(",
c.n, ")")
Base.show(io::IO, c::ZNIrrep{3}) =
    get(io, :typeinfo, nothing) === ZNIrrep{3} ? print(io, c.n) : print(io, "ℤ₃(",
c.n, ")")
Base.show(io::IO, c::ZNIrrep{4}) =
    get(io, :typeinfo, nothing) === ZNIrrep{4} ? print(io, c.n) : print(io, "ℤ₄(",
c.n, ")")
Base.show(io::IO, c::ZNIrrep{N}) where {N} =
    get(io, :typeinfo, nothing) === ZNIrrep{N} ?
        print(io, c.n) : print(io, "ZNIrrep{", N, "}(" , c.n, ")")

# U1Irrep: irreps of U1 are labelled by integers
"""
    struct U1Irrep <: AbelianIrrep
    U1Irrep(j::Real)

Represents irreps of the group `U₁ == SO₂`, both of which are valid unicode
synonyms.
The irrep is labelled by a charge, which should be an integer for a linear
representation.
```

```julia
However, it is often useful to allow half integers to represent irreps of `U₁`
subgroups of
``SU₂``. Hence, the charge is stored as a `HalfInt` from the package
HalfIntegers.jl, but
can be entered as arbitrary `Real`.
"""
struct U1Irrep <: AbelianIrrep
    charge::HalfInt
end

Base.IteratorSize(::Type{SectorValues{U1Irrep}}) = IsInfinite()
Base.iterate(::SectorValues{U1Irrep}, i = 0) =
    return i <= 0 ? (U1Irrep(half(i)), (-i + 1)) : (U1Irrep(half(i)), -i)
function Base.getindex(::SectorValues{U1Irrep}, i::Int)
    i < 1 && throw(BoundsError(values(U1Irrep), i))
    return U1Irrep(iseven(i) ? half(i>>1) : -half(i>>1))
end
findindex(::SectorValues{U1Irrep}, c::U1Irrep) = (n = twice(c.charge);
2*abs(n)+(n<=0))

Base.one(::Type{U1Irrep}) = U1Irrep(0)
Base.conj(c::U1Irrep) = U1Irrep(-c.charge)
⊗(c1::U1Irrep, c2::U1Irrep) = (U1Irrep(c1.charge+c2.charge),)

Base.convert(::Type{U1Irrep}, c::Real) = U1Irrep(c)

const U₁ = U1Irrep
const SO₂ = U1Irrep
Base.show(io::IO, ::Type{U1Irrep}) = print(io, "U₁")
Base.show(io::IO, c::U1Irrep) =
    get(io, :typeinfo, nothing) === U1Irrep ? print(io, c.charge) :
        print(io, "U₁(", c.charge, ")")

Base.hash(c::ZNIrrep{N}, h::UInt) where {N} = hash(c.n, h)
Base.isless(c1::ZNIrrep{N}, c2::ZNIrrep{N}) where {N} = isless(c1.n, c2.n)
Base.hash(c::U1Irrep, h::UInt) = hash(c.charge, h)
@inline Base.isless(c1::U1Irrep, c2::U1Irrep) where {N} =
    isless(abs(c1.charge), abs(c2.charge)) || zero(HalfInt) < c1.charge ==
-c2.charge

# Nob-abelian groups
#------------------------------------------------------------------------------#
# SU2Irrep: irreps of SU2 are labelled by half integers j
struct SU2IrrepException <: Exception end
Base.show(io::IO, ::SU2IrrepException) =
    print(io, "Irreps of (bosonic or fermionic) `SU₂` should be labelled by
non-negative half integers, i.e. elements of `Rational{Int}` with denominator 1 or
2")

"""
    struct SU2Irrep <: Irrep
    SU2Irrep(j::Real)

Represents irreps of the group `SU₂`, which is also a valid unicode synonym. The
```

```
irrep is
labelled by a half integer `j` which can be entered as an abitrary `Real`, but is
stored as
a `HalfInt` from the HalfIntegers.jl package. Half-integer and integer irreps of
`SU₂` are
also projective and linear representation of `SO₃`, which is another valid unicode
synonym.
"""
struct SU2Irrep <: Irrep
    j::HalfInt
    function SU2Irrep(j)
        j >= zero(j) || error("Not a valid SU₂ irrep")
        new(j)
    end
end

Base.IteratorSize(::Type{SectorValues{SU2Irrep}}) = IsInfinite()
Base.iterate(::SectorValues{SU2Irrep}, i = 0) = (SU2Irrep(half(i)), i+1)
Base.getindex(::SectorValues{SU2Irrep}, i::Int) =
    1 <= i ? SU2Irrep(half(i-1)) : throw(BoundsError(values(SU2Irrep), i))
findindex(::SectorValues{SU2Irrep}, s::SU2Irrep) = twice(s.j)+1

const _su2one = SU2Irrep(zero(HalfInt))
Base.one(::Type{SU2Irrep}) = _su2one
Base.conj(s::SU2Irrep) = s
⊗(s1::SU2Irrep, s2::SU2Irrep) = SectorSet{SU2Irrep}(abs(s1.j-s2.j):(s1.j+s2.j))

# SU2Irrep(j::Real) = convert(SU2Irrep, j)
Base.convert(::Type{SU2Irrep}, j::Real) = SU2Irrep(j)

dim(s::SU2Irrep) = twice(s.j)+1

Base.@pure FusionStyle(::Type{SU2Irrep}) = SimpleNonAbelian()
Base.isreal(::Type{SU2Irrep}) = true

Nsymbol(sa::SU2Irrep, sb::SU2Irrep, sc::SU2Irrep) = WignerSymbols.δ(sa.j, sb.j,
sc.j)
function Fsymbol(s1::SU2Irrep, s2::SU2Irrep, s3::SU2Irrep,
                    s4::SU2Irrep, s5::SU2Irrep, s6::SU2Irrep)
    if all(==(_su2one), (s1, s2, s3, s4, s5, s6))
        return 1.0
    else
        return sqrt(dim(s5) * dim(s6)) * WignerSymbols.racahW(Float64, s1.j, s2.j,
                                                                s4.j, s3.j, s5.j,
s6.j)
    end
end
function Rsymbol(sa::SU2Irrep, sb::SU2Irrep, sc::SU2Irrep)
    Nsymbol(sa, sb, sc) || return 0.
    iseven(convert(Int, sa.j+sb.j-sc.j)) ? 1.0 : -1.0
end

function fusiontensor(a::SU2Irrep, b::SU2Irrep, c::SU2Irrep, v::Nothing = nothing)
    C = Array{Float64}(undef, dim(a), dim(b), dim(c))
```

```julia
    ja, jb, jc = a.j, b.j, c.j

    for kc = 1:dim(c), kb = 1:dim(b), ka = 1:dim(a)
        C[ka,kb,kc] = WignerSymbols.clebschgordan(ja, ja+1-ka, jb, jb+1-kb, jc,
jc+1-kc)
    end
    return C
end

const SU₂ = SU2Irrep
const SO₃ = SU2Irrep
Base.show(io::IO, ::Type{SU2Irrep}) = print(io, "SU₂")
Base.show(io::IO, s::SU2Irrep) =
    get(io, :typeinfo, nothing) === SU2Irrep ? print(io, s.j) : print(io, "SU₂(",
s.j, ")")

Base.hash(s::SU2Irrep, h::UInt) = hash(s.j, h)
Base.isless(s1::SU2Irrep, s2::SU2Irrep) = isless(s1.j, s2.j)

# U₁ ⋉ C (U₁ and charge conjugation)
"""
    struct CU1Irrep <: Irrep
        j::HalfInt # value of the U1 charge
        s::Int # rep of charge conjugation:
    end

Represents irreps of the group ``U₁ ⋉ C`` (``U₁`` and charge conjugation or
reflection),
which is also known as just `O₂`. Unicode synomyms are thus `CU₁` or `O₂`. The
irrep is
labelled by a positive half integer `j` (the ``U₁`` charge) and an integer `s`
indicating
the behaviour under charge conjugation. They take values:
*   if `j == 0`, `s = 0` (trivial charge conjugation) or
    `s = 1` (non-trivial charge conjugation)
*   if `j > 0`, `s = 2` (two-dimensional representation)
"""
struct CU1Irrep <: Irrep
    j::HalfInt # value of the U1 charge
    s::Int # rep of charge conjugation:
    # if j == 0, s = 0 (trivial) or s = 1 (non-trivial),
    # else s = 2 (two-dimensional representation)
    # Let constructor take the actual half integer value j
    function CU1Irrep(j::Real, s::Int = ifelse(j>zero(j), 2, 0))
        if ((j > zero(j) && s == 2) || (j == zero(j) && (s == 0 || s == 1)))
            new(j, s)
        else
            error("Not a valid CU₁ irrep")
        end
    end
end

Base.IteratorSize(::Type{SectorValues{CU1Irrep}}) = IsInfinite()
function Base.iterate(::SectorValues{CU1Irrep}, state = (0, 0))
```

```
    j, s = state
    if iszero(j) && s == 0
        return CU1Irrep(j, s), (j, 1)
    elseif iszero(j) && s == 1
        return CU1Irrep(j, s), (j+1, 2)
    else
        return CU1Irrep(half(j), s), (j+1, 2)
    end
end
function Base.getindex(::SectorValues{CU1Irrep}, i::Int)
    i < 1 && throw(BoundsError(values(CU1Irrep), i))
    if i == 1
        return CU1Irrep(0, 0)
    elseif i == 2
        return CU1Irrep(0, 1)
    else
        return CU1Irrep(half(i-2), 2)
    end
end
findindex(::SectorValues{CU1Irrep}, c::CU1Irrep) = twice(c.j) + iszero(c.j) + c.s

Base.hash(c::CU1Irrep, h::UInt) = hash(c.s, hash(c.j, h))
Base.isless(c1::CU1Irrep, c2::CU1Irrep) =
    isless(c1.j, c2.j) || (c1.j == c2.j == zero(HalfInt) && c1.s < c2.s)

# CU1Irrep(j::Real, s::Int = ifelse(j>0, 2, 0)) = CU1Irrep(convert(HalfInteger,
  j), s)

Base.convert(::Type{CU1Irrep}, j::Real) = CU1Irrep(j)
Base.convert(::Type{CU1Irrep}, js::Tuple{Real,Int}) = CU1Irrep(js...)

Base.one(::Type{CU1Irrep}) = CU1Irrep(zero(HalfInt), 0)
Base.conj(c::CU1Irrep) = c

struct CU1ProdIterator
    a::CU1Irrep
    b::CU1Irrep
end
function Base.iterate(p::CU1ProdIterator, s::Int = 1)
    if s == 1
        if p.a.j == p.b.j == zero(HalfInt)
            return CU1Irrep(zero(HalfInt), xor(p.a.s, p.b.s)), 4
        elseif p.a.j == zero(HalfInt)
            return p.b, 4
        elseif p.b.j == zero(HalfInt)
            return p.a, 4
        elseif p.a == p.b # != zero
            return one(CU1Irrep), 2
        else
            return CU1Irrep(abs(p.a.j - p.b.j)),  3
        end
    elseif s == 2
        return CU1Irrep(zero(HalfInt), 1), 3
    elseif s == 3
```

```
        CU1Irrep(p.a.j + p.b.j), 4
    else
        return nothing
    end
end
function Base.length(p::CU1ProdIterator)
    if p.a.j == zero(HalfInt) || p.b.j == zero(HalfInt)
        return 1
    elseif p.a == p.b
        return 3
    else
        return 2
    end
end

⊗(a::CU1Irrep, b::CU1Irrep) = CU1ProdIterator(a, b)

dim(c::CU1Irrep) = ifelse(c.j == zero(HalfInt), 1, 2)

Base.@pure FusionStyle(::Type{CU1Irrep}) = SimpleNonAbelian()
Base.isreal(::Type{CU1Irrep}) = true

function Nsymbol(a::CU1Irrep, b::CU1Irrep, c::CU1Irrep)
    ifelse(c.s == 0, (a.j == b.j) & ((a.s == b.s == 2) | (a.s == b.s)),
        ifelse(c.s == 1, (a.j == b.j) & ((a.s == b.s == 2) | (a.s != b.s)),
        (c.j == a.j + b.j) | (c.j == abs(a.j - b.j)) ))
end
function Fsymbol(a::CU1Irrep, b::CU1Irrep, c::CU1Irrep,
        d::CU1Irrep, e::CU1Irrep, f::CU1Irrep)
    Nabe = convert(Int, Nsymbol(a,b,e))
    Necd = convert(Int, Nsymbol(e,c,d))
    Nbcf = convert(Int, Nsymbol(b,c,f))
    Nafd = convert(Int, Nsymbol(a,f,d))

    Nabe*Necd*Nbcf*Nafd == 0 && return 0.

    op = CU1Irrep(0,0)
    om = CU1Irrep(0,1)

    if a == op || b == op || c == op
        return 1.
    end
    if (a == b == om) || (a == c == om) || (b == c == om)
        return 1.
    end
    if a == om
        if d.j == zero(HalfInt)
            return 1.
        else
            return (d.j == c.j - b.j) ? -1. : 1.
        end
    end
    if b == om
        return (d.j == abs(a.j - c.j)) ? -1. : 1.
```

```
    end
    if c == om
        return (d.j == a.j - b.j) ? -1. : 1.
    end
    # from here on, a,b,c are neither 0+ or 0-
    s = sqrt(2)/2
    if a == b == c
        if d == a
            if e.j == 0
                if f.j == 0
                    return f.s == 1 ? -0.5 : 0.5
                else
                    return e.s == 1 ? -s : s
                end
            else
                return f.j == 0 ? s : 0.
            end
        else
            return 1.
        end
    end
    if a == b # != c
        if d == c
            if f.j == b.j + c.j
                return e.s == 1 ? -s : s
            else
                return s
            end
        else
            return 1.
        end
    end
    if b == c
        if d == a
            if e.j == a.j + b.j
                return s
            else
                return f.s == 1 ? -s : s
            end
        else
            return 1.
        end
    end
    if a == c
        if d == b
            if e.j == f.j
                return 0.
            else
                return 1.
            end
        else
            return d.s == 1 ? -1. : 1.
        end
    end
```

```
    if d == om
        return b.j == a.j + c.j ? -1. : 1.
    end
    return 1.
end
function Rsymbol(a::CU1Irrep, b::CU1Irrep, c::CU1Irrep)
    R = convert(Float64, Nsymbol(a, b, c))
    return c.s == 1 && a.j > 0 ? -R : R
end

function fusiontensor(a::CU1Irrep, b::CU1Irrep, c::CU1Irrep, ::Nothing = nothing)
    C = fill(0., dim(a), dim(b), dim(c))
    !Nsymbol(a,b,c) && return C
    if c.j == 0
        if a.j == b.j == 0
            C[1,1,1] = 1.
        else
            if c.s == 0
                C[1,2,1] = 1. / sqrt(2)
                C[2,1,1] = 1. / sqrt(2)
            else
                C[1,2,1] = 1. / sqrt(2)
                C[2,1,1] = -1. / sqrt(2)
            end
        end
    elseif a.j == 0
        C[1,1,1] = 1.
        C[1,2,2] = a.s == 1 ? -1. : 1.
    elseif b.j == 0
        C[1,1,1] = 1.
        C[2,1,2] = b.s == 1 ? -1. : 1.
    elseif c.j == a.j + b.j
        C[1,1,1] = 1.
        C[2,2,2] = 1.
    elseif c.j == a.j - b.j
        C[1,2,1] = 1.
        C[2,1,2] = 1.
    elseif c.j == b.j - a.j
        C[2,1,1] = 1.
        C[1,2,2] = 1.
    end
    return C
end
frobeniusschur(::CU1Irrep) = 1

const CU₁ = CU1Irrep
Base.show(io::IO, ::Type{CU1Irrep}) = print(io, "CU₁")
function Base.show(io::IO, c::CU1Irrep)
    if c.s == 1
        if get(io, :typeinfo, nothing) === CU1Irrep
            print(io, "(", c.j, ", ", c.s, ")")
        else
            print(io, "CU₁(", c.j, ", ", c.s, ")")
        end
```

```julia
        else
            if get(io, :typeinfo, nothing) === CU1Irrep
                print(io, "(", c.j, ")")
            else
                print(io, "CU₁(", c.j, ")")
            end
        end
    end
```