```
# Direct product of different sectors
#==============================================================================#
const SectorTuple = Tuple{Vararg{Sector}}

struct ProductSector{T<:SectorTuple} <: Sector
    sectors::T
end
_sectors(::Type{Tuple{}}) = ()
_sectors(::Type{T}) where {T<:SectorTuple} =
    (Base.tuple_type_head(T),  _sectors(Base.tuple_type_tail(T))...)

Base.IteratorSize(::Type{SectorValues{ProductSector{T}}}) where {T<:SectorTuple} =
    Base.IteratorSize(Base.Iterators.product(map(values, _sectors(T))...))
Base.@pure Base.size(::SectorValues{ProductSector{T}}) where {T<:SectorTuple} =
    map(s->length(values(s)), _sectors(T))
Base.@pure Base.length(P::SectorValues{<:ProductSector}) = *(size(P)...)

function Base.iterate(::SectorValues{ProductSector{T}}, args...) where
{T<:SectorTuple}
    next = iterate(product(values.(_sectors(T))...), args...)
    next === nothing && return nothing
    val, state = next
    return ProductSector{T}(val), state
end
function Base.getindex(P::SectorValues{ProductSector{T}}, i::Int) where
{T<:SectorTuple}
    Base.IteratorSize(P) isa IsInfinite &&
        throw(ArgumentError("cannot index into infinite product sector"))
    ProductSector{T}(getindex.(values.(_sectors(T)),
Tuple(CartesianIndices(size(P))[i])))
end
function findindex(P::SectorValues{ProductSector{T}}, c::ProductSector{T}) where

{T<:SectorTuple}
    Base.IteratorSize(P) isa IsInfinite &&
        throw(ArgumentError("cannot index into infinite product sector"))
    LinearIndices(size(P))[CartesianIndex(findindex.(values.(_sectors(T)),
c.sectors))]
end

ProductSector{T}(args...) where {T<:SectorTuple} = ProductSector{T}(args)
Base.convert(::Type{ProductSector{T}}, t::Tuple) where {T<:SectorTuple} =
    ProductSector{T}(convert(T, t))

Base.one(::Type{ProductSector{T}}) where {G<:Sector, T<:Tuple{G}} =
ProductSector((one(G),))
Base.one(::Type{ProductSector{T}}) where {G<:Sector, T<:Tuple{G,Vararg{Sector}}} =
    one(G) × one(ProductSector{Base.tuple_type_tail(T)})

Base.conj(p::ProductSector) = ProductSector(map(conj, p.sectors))
function ⊗(p1::P, p2::P) where {P<:ProductSector}
    if FusionStyle(P) isa Abelian
        (P(first(product(map(⊗, p1.sectors, p2.sectors)...))),)
    else
```

```julia
        return SectorSet{P}(product(map(⊗,p1.sectors,p2.sectors)...))
    end
end

Nsymbol(a::P, b::P, c::P) where {P<:ProductSector} =
    prod(map(Nsymbol, a.sectors, b.sectors, c.sectors))
function Fsymbol(a::P, b::P, c::P, d::P, e::P, f::P) where {P<:ProductSector}
    if FusionStyle(P) isa Abelian || FusionStyle(P) isa SimpleNonAbelian
        return prod(map(Fsymbol, a.sectors, b.sectors, c.sectors,
                                    d.sectors, e.sectors, f.sectors))
    else
        # TODO: DegenerateNonAbelian case, use kron ?
        throw(MethodError(Fsymbol,(a,b,c,d,e,f)))
    end
end
function Rsymbol(a::P, b::P, c::P) where {P<:ProductSector}
    if FusionStyle(P) isa Abelian || FusionStyle(P) isa SimpleNonAbelian
        return prod(map(Rsymbol, a.sectors, b.sectors, c.sectors))
    else
        # TODO: DegenerateNonAbelian case, use kron ?
        throw(MethodError(Rsymbol,(a,b,c)))
    end
end
function Asymbol(a::P, b::P, c::P) where {P<:ProductSector}
    if FusionStyle(P) isa Abelian || FusionStyle(P) isa SimpleNonAbelian
        return prod(map(Asymbol, a.sectors, b.sectors, c.sectors))
    else
        # TODO: DegenerateNonAbelian case, use kron ?
        throw(MethodError(Asymbol,(a,b,c)))
    end
end
function Bsymbol(a::P, b::P, c::P) where {P<:ProductSector}
    if FusionStyle(P) isa Abelian || FusionStyle(P) isa SimpleNonAbelian
        return prod(map(Bsymbol, a.sectors, b.sectors, c.sectors))
    else
        # TODO: DegenerateNonAbelian case, use kron ?
        throw(MethodError(Bsymbol,(a,b,c)))
    end
end
frobeniusschur(p::ProductSector) = prod(map(frobeniusschur, p.sectors))

fusiontensor(a::ProductSector{T}, b::ProductSector{T}, c::ProductSector{T},
                v::Nothing = nothing) where {T<:SectorTuple} =
    _kron(fusiontensor(a.sectors[1], b.sectors[1], c.sectors[1]),
            fusiontensor(ProductSector(tail(a.sectors)),
ProductSector(tail(b.sectors)),
                            ProductSector(tail(c.sectors))))

fusiontensor(a::ProductSector{T}, b::ProductSector{T}, c::ProductSector{T},
                v::Nothing = nothing) where {T<:Tuple{<:Sector}} =
    fusiontensor(a.sectors[1], b.sectors[1], c.sectors[1])

FusionStyle(::Type{<:ProductSector{T}}) where {T<:SectorTuple} =
    Base.:&(map(FusionStyle, _sectors(T))...)
```

```
BraidingStyle(::Type{<:ProductSector{T}}) where {T<:SectorTuple} =
    Base.:&(map(BraidingStyle, _sectors(T))...)
Base.isreal(::Type{<:ProductSector{T}}) where {T<:SectorTuple} = all(isreal,
_sectors(T))

fermionparity(P::ProductSector) = mapreduce(fermionparity, xor, P.sectors)

dim(p::ProductSector) = *(dim.(p.sectors)...)

Base.isequal(p1::ProductSector, p2::ProductSector) = isequal(p1.sectors,
p2.sectors)
Base.hash(p::ProductSector, h::UInt) = hash(p.sectors, h)
Base.isless(p1::ProductSector{T}, p2::ProductSector{T}) where {T} =
    isless(reverse(p1.sectors), reverse(p2.sectors))

# Default construction from tensor product of sectors
#-----------------------------------------------------
×(S1, S2, S3, S4...) = ×(×(S1, S2), S3, S4...)

×(S1::Sector, S2::Sector) = ProductSector((S1, S2))
×(P1::ProductSector, S2::Sector) = ProductSector(tuple(P1.sectors..., S2))
×(S1::Sector, P2::ProductSector) = ProductSector(tuple(S1, P2.sectors...))
×(P1::ProductSector, P2::ProductSector) =
    ProductSector(tuple(P1.sectors..., P2.sectors...))

×(G1::Type{ProductSector{Tuple{}}},
                    G2::Type{ProductSector{T}}) where {T<:SectorTuple} = G2
×(G1::Type{ProductSector{T1}},
                    G2::Type{ProductSector{T2}}) where
{T1<:SectorTuple,T2<:SectorTuple} =
    tuple_type_head(T1) × (ProductSector{tuple_type_tail(T1)} × G2)
×(G1::Type{ProductSector{Tuple{}}}, G2::Type{<:Sector}) =
    ProductSector{Tuple{G2}}
×(G1::Type{ProductSector{T}}, G2::Type{<:Sector}) where {T<:SectorTuple} =
    Base.tuple_type_head(T) × (ProductSector{Base.tuple_type_tail(T)} × G2)
×(G1::Type{<:Sector}, G2::Type{ProductSector{T}}) where {T<:SectorTuple} =
    ProductSector{Base.tuple_type_cons(G1,T)}
×(G1::Type{<:Sector}, G2::Type{<:Sector}) = ProductSector{Tuple{G1,G2}}

function Base.show(io::IO, P::ProductSector)
    sectors = P.sectors
    compact = get(io, :typeinfo, nothing) === typeof(P)
    sep = compact ? ", " : " × "
    print(io,"(")
    for i = 1:length(sectors)
        i == 1 || print(io, sep)
        io2 = compact ? IOContext(io, :typeinfo => typeof(sectors[i])) : io
        print(io2, sectors[i])
    end
    print(io,")")
end

function Base.show(io::IO, ::Type{ProductSector{T}}) where {T<:SectorTuple}
    sectors = T.parameters
```

```
    print(io,"(")
    for i = 1:length(sectors)
        i == 1 || print(io, " × ")
        print(io, sectors[i])
    end
    print(io,")")
end
```