```julia
# abstracttensor.jl
#
# Abstract Tensor type
#----------------------
"""
    abstract type AbstractTensorMap{S<:IndexSpace, N₁, N₂} end

Abstract supertype of all tensor maps, i.e. linear maps between tensor products
of vector spaces of type `S<:IndexSpace`. An `AbstractTensorMap` maps from
an input space of type `ProductSpace{S, N₂}` to an output space of type
`ProductSpace{S, N₁}`.
"""
abstract type AbstractTensorMap{S<:IndexSpace, N₁, N₂} end
"""
    AbstractTensor{S<:IndexSpace, N} = AbstractTensorMap{S, N, 0}

Abstract supertype of all tensors, i.e. elements in the tensor product space
of type `ProductSpace{S, N}`, built from elementary spaces of type `S<:IndexSpace`.

An `AbstractTensor{S, N}` is actually a special case `AbstractTensorMap{S, N, 0}`,
i.e. a tensor map with only a non-trivial output space.
"""
const AbstractTensor{S<:IndexSpace, N} = AbstractTensorMap{S, N, 0}

# tensor characteristics
Base.eltype(t::AbstractTensorMap) = eltype(typeof(t))
Base.eltype(T::Type{<:AbstractTensorMap}) = eltype(storagetype(T))

storagetype(t::AbstractTensorMap) = storagetype(typeof(t))

similarstoragetype(t::AbstractTensorMap, T) = similarstoragetype(typeof(t), T)
similarstoragetype(TT::Type{<:AbstractTensorMap}, ::Type{T}) where {T} =
    Core.Compiler.return_type(similar, Tuple{storagetype(TT), Type{T}})

spacetype(t::AbstractTensorMap) = spacetype(typeof(t))
spacetype(::Type{<:AbstractTensorMap{S}}) where {S<:IndexSpace} = S

sectortype(t::AbstractTensorMap) = sectortype(typeof(t))
sectortype(::Type{<:AbstractTensorMap{S}}) where {S<:IndexSpace} = sectortype(S)

field(t::AbstractTensorMap) = field(typeof(t))
field(::Type{<:AbstractTensorMap{S}}) where {S<:IndexSpace} = field(S)

numout(t::AbstractTensorMap) = numout(typeof(t))
numout(::Type{<:AbstractTensorMap{<:IndexSpace, N₁, N₂}}) where {N₁, N₂} = N₁

numin(t::AbstractTensorMap) = numin(typeof(t))
numin(::Type{<:AbstractTensorMap{<:IndexSpace, N₁, N₂}}) where {N₁, N₂} = N₂

numind(t::AbstractTensorMap) = numind(typeof(t))
numind(::Type{<:AbstractTensorMap{<:IndexSpace, N₁, N₂}}) where {N₁, N₂} = N₁ + N₂
const order = numind

# tensormap implementation should provide codomain(t) and domain(t)
```

```julia
codomain(t::AbstractTensorMap, i) = codomain(t)[i]
domain(t::AbstractTensorMap, i) = domain(t)[i]
source(t::AbstractTensorMap) = domain(t) # categorical terminology
target(t::AbstractTensorMap) = codomain(t) # categorical terminology
space(t::AbstractTensorMap) = HomSpace(codomain(t), domain(t))
space(t::AbstractTensorMap, i::Int) = space(t)[i]
dim(t::AbstractTensorMap) = dim(space(t))

# some index manipulation utilities
codomainind(t::AbstractTensorMap) = codomainind(typeof(t))
codomainind(::Type{<:AbstractTensorMap{<:IndexSpace, N₁, N₂}}) where {N₁, N₂} =
    ntuple(n->n, N₁)

domainind(t::AbstractTensorMap) = domainind(typeof(t))
domainind(::Type{<:AbstractTensorMap{<:IndexSpace, N₁, N₂}}) where {N₁, N₂} =
    ntuple(n-> N₁+n, N₂)

allind(t::AbstractTensorMap) = allind(typeof(t))
allind(::Type{<:AbstractTensorMap{<:IndexSpace, N₁, N₂}}) where {N₁, N₂} =
    ntuple(n->n, N₁+N₂)

adjointtensorindex(t::AbstractTensorMap{<:IndexSpace, N₁, N₂}, i) where {N₁, N₂} =
    ifelse(i<=N₁, N₂+i, i-N₁)
adjointtensorindices(t::AbstractTensorMap, indices::IndexTuple) =
    map(i->adjointtensorindex(t, i), indices)

# Equality and approximality
#-----------------------------
function Base.:(==)(t1::AbstractTensorMap, t2::AbstractTensorMap)
    (codomain(t1) == codomain(t2) && domain(t1) == domain(t2)) || return false
    for c in blocksectors(t1)
        block(t1, c) == block(t2, c) || return false
    end
    return true
end
function Base.hash(t::AbstractTensorMap, h::UInt)
    h = hash(codomain(t), h)
    h = hash(domain(t), h)
    for (c, b) in blocks(t)
        h = hash(c, hash(b, h))
    end
    return h
end

function Base.isapprox(t1::AbstractTensorMap, t2::AbstractTensorMap;
                atol::Real=0, rtol::Real=Base.rtoldefault(eltype(t1), eltype(t2),
atol))
    d = norm(t1 - t2)
    if isfinite(d)
        return d <= max(atol, rtol*max(norm(t1), norm(t2)))
    else
        return false
    end
end
```

```
# Conversion to Array:
#----------------------
# probably not optimized for speed, only for checking purposes
function Base.convert(::Type{Array}, t::AbstractTensorMap{S, N₁, N₂}) where {S,
N₁, N₂}
    I = sectortype(t)
    if I === Trivial
        convert(Array, t[])
    else
        cod = codomain(t)
        dom = domain(t)
        local A
        for (f1, f2) in fusiontrees(t)
            F1 = convert(Array, f1)
            F2 = convert(Array, f2)
            sz1 = size(F1)
            sz2 = size(F2)
            d1 = TupleTools.front(sz1)
            d2 = TupleTools.front(sz2)
            F = reshape(reshape(F1, TupleTools.prod(d1), sz1[end])*reshape(F2,
TupleTools.prod(d2), sz2[end])', (d1..., d2...))
            if !(@isdefined A)
                if eltype(F) <: Complex
                    T = complex(float(eltype(t)))
                elseif eltype(F) <: Integer
                    T = eltype(t)
                else
                    T = float(eltype(t))
                end
                A = fill(zero(T), (dims(cod)..., dims(dom)...))
            end
            Aslice = StridedView(A)[axes(cod, f1.uncoupled)..., axes(dom,
f2.uncoupled)...]
            axpy!(1, StridedView(_kron(convert(Array, t[f1, f2]), F)), Aslice)
        end
        return A
    end
end
```