````julia
# methods/simple.jl
#
# Method-based access to tensor operations using simple definitions.

# type unstable; better use tuples instead
tensorcopy(A, IA, IC=IA) = tensorcopy(A, tuple(IA...), tuple(IC...))
tensoradd(A, IA, B, IB, IC=IA) = tensoradd(A, tuple(IA...), B, tuple(IB...),
tuple(IC...))
tensortrace(A, IA, IC = unique2(IA)) = tensortrace(A, tuple(IA...), tuple(IC...))
tensorcontract(A, IA, B, IB, IC = symdiff(IA, IB)) =
    tensorcontract(A, tuple(IA...), B, tuple(IB...), tuple(IC...))
tensorproduct(A, IA, B, IB, IC = vcat(IA, IB)) = tensorproduct(A, tuple(IA...), B,
tuple(IB...), tuple(IC...))

"""
    tensorcopy(A, IA, IC = IA)

Creates a copy of `A`, where the dimensions of `A` are assigned indices from the
iterable `IA` and the indices of the copy are contained in `IC`. Both iterables
should contain the same elements in a different order.

The result of this method is equivalent to `permutedims(A, p)` where p is the
permutation
such that `IC = IA[p]`. The implementation of `tensorcopy` is however more
efficient on
average, especially if `Threads.nthreads() > 1`.
"""
function tensorcopy(A, IA::Tuple, IC::Tuple=IA)
    indCinA = add_indices(IA, IC)
    C = similar_from_indices(eltype(A), indCinA, (), A, :N)
    add!(1, A, :N, 0, C, indCinA)
    return C
end

"""
    tensoradd(A, IA, B, IB, IC = IA)

Returns the result of adding arrays `A` and `B` where the iterabels `IA` and `IB`
denote how the array data should be permuted in order to be added. More
specifically,
the result of this method is equivalent to

```julia
tensorcopy(A, IA, IC) + tensorcopy(B, IB, IC)
```

but without creating the temporary permuted arrays.
"""
function tensoradd(A, IA::Tuple, B, IB::Tuple, IC::Tuple=IA)
    T = promote_type(eltype(A), eltype(B))
    indCinA = add_indices(IA, IC)
    C = similar_from_indices(T, indCinA, (), A, :N)
    add!(1, A, :N, 0, C, indCinA)
    indCinB = add_indices(IB, IC)
````

```julia
    add!(1, B, :N, 1, C, indCinB)
    return C
end

"""
    tensortrace(A, IA [, IC])

Trace or contract pairs of indices of array `A`, by assigning them an identical
indices in the iterable `IA`. The untraced indices, which are assigned a unique
index,
can be reordered according to the optional argument `IC`. The default value
corresponds
to the order in which they appear. Note that only pairs of indices can be
contracted,
so that every index in `IA` can appear only once (for an untraced index) or twice
(for an index in a contracted pair).
"""
function tensortrace(A, IA::Tuple, IC::Tuple)
    indCinA, cindA1, cindA2 = trace_indices(IA, IC)
    C = similar_from_indices(eltype(A), indCinA, (), A, :N)
    trace!(1, A, :N, 0, C, indCinA, cindA1, cindA2)
    return C
end

"""
    tensorcontract(A, IA, B, IB[, IC])

Contract indices of array `A` with corresponding indices in array `B` by assigning
them identical labels in the iterables `IA` and `IB`. The indices of the resulting
array correspond to the indices that only appear in either `IA` or `IB` and can be
ordered by specifying the optional argument `IC`. The default is to have all open
indices of array `A` followed by all open indices of array `B`. Note that inner
contractions of an array should be handled first with `tensortrace`, so that every
label can appear only once in `IA` or `IB` seperately, and once (for open
index) or twice (for contracted index) in the union of `IA` and `IB`.

The contraction can be performed by a native Julia algorithm without creating any
temporaries, or by first permuting the arrays such that the contraction becomes
equivalent
to a matrix product, which is then performed by BLAS. The latter is typically
faster for
large arrays. The choice of method is globally controlled by the methods
[`enable_blas()`](@ref) and [`disable_blas()`](@ref).
"""
function tensorcontract(A, IA::Tuple, B, IB::Tuple, IC::Tuple)
    oindA, cindA, oindB, cindB, indCinoAB = contract_indices(IA, IB, IC)

    T = promote_type(eltype(A), eltype(B))
    C = similar_from_indices(T, oindA, oindB, indCinoAB, (), A, B, :N, :N)

    contract!(1, A, :N, B, :N, 0, C, oindA, cindA, oindB, cindB, indCinoAB)
    return C
end

"""
```

```
    tensorproduct(A, IA, B, IB, IC = (IA..., IB...))

Computes the tensor product of two arrays `A` and `B`, i.e. returns a new array `C`
with `ndims(C) = ndims(A)+ndims(B)`. The indices of the output tensor are related to
those of the input tensors by the pattern specified by the indices. Essentially,
this is a special case of `tensorcontract` with no indices being contracted over.
This method checks whether the indices indeed specify a tensor product instead of
a genuine contraction.
"""
function tensorproduct(A, IA::Tuple, B, IB::Tuple, IC::Tuple = (IA..., IB...))
    isempty(intersect(IA, IB)) || throw(IndexError("not a valid tensor product"))
    tensorcontract(A, IA, B, IB, IC)
end
```