```julia
const defaultparser = TensorParser()

"""
    @notensor(block)

Marks a block which should be ignored within an `@tensor` environment. Has no
effect outside of `@tensor`.
"""
macro notensor(ex::Expr)
    return esc(ex)
end

"""
    @tensor(block)

Specify one or more tensor operations using Einstein's index notation. Indices can
be chosen to be arbitrary Julia variable names, or integers. When contracting
several
tensors together, this will be evaluated as pairwise contractions in left to right
order, unless the so-called NCON style is used (positive integers for contracted
indices and negative indices for open indices).

A second argument to the `@tensor` macro can be provided of the form `order=(...)`,
where
the list specifies the contraction indices in the order in which they will be
contracted.
"""
macro tensor(ex::Expr)
    return esc(defaultparser(ex))
end

macro tensor(ex::Expr, orderex::Expr)
    parser = TensorParser()
    if !(orderex.head == :(=) && orderex.args[1] == :order &&
            orderex.args[2] isa Expr && orderex.args[2].head == :tuple)
        throw(ArgumentError("unkown first argument in @tensor, should be `order =
(...,)`"))
    end
    indexorder = map(normalizeindex, orderex.args[2].args)
    parser.contractiontreebuilder = network->indexordertree(network, indexorder)
    return esc(parser(ex))
end

"""
    @tensoropt(optex, block)
    @tensoropt(block)

Specify one or more tensor operations using Einstein's index notation. Indices can
be chosen to be arbitrary Julia variable names, or integers. When contracting
several
tensors together, the macro will determine (at compile time) the optimal contraction
order depending on the cost associated to the individual indices. If no `optex` is
provided, all indices are assumed to have an abstract scaling `χ` which is optimized
in the asympotic limit of large `χ`.
```

````julia
The cost can be specified in the following ways:

```julia
@tensoropt (a=>χ,b=>χ^2,c=>2*χ,e=>5) C[a,b,c,d] := A[a,e,c,f,h]*B[f,g,e,b]*C[g,d,h]
# asymptotic cost as specified for listed indices, unlisted indices have cost 1
(any symbol for χ can be used)
@tensoropt (a,b,c,e) C[a,b,c,d] := A[a,e,c,f,h]*B[f,g,e,b]*C[g,d,h]
# asymptotic cost χ for indices a,b,c,e, other indices (d,f) have cost 1
@tensoropt !(a,b,c,e) C[a,b,c,d] := A[a,e,c,f,h]*B[f,g,e,b]*C[g,d,h]
# cost 1 for indices a,b,c,e; other indices (d,f) have asymptotic cost χ
@tensoropt C[a,b,c,d] := A[a,e,c,f,h]*B[f,g,e,b]*C[g,d,h]
# asymptotic cost χ for all indices (a,b,c,d,e,f)
```

Note that `@tensoropt` will optimize any tensor contraction sequence it encounters
in the (block of) expressions. It will however not break apart expressions that have
been explicitly grouped with parenthesis, i.e. in
```julia
@tensoroptC[a,b,c,d] := A[a,e,c,f,h]*(B[f,g,e,b]*C[g,d,h])
```
it will always contract `B` and `C` first. For a single tensor contraction sequence,
the optimal contraction order and associated (asymptotic) cost can be obtained using
`@optimalcontractiontree`.
"""
macro tensoropt(expressions...)
    if length(expressions) == 1
        ex = expressions[1]
        optdict = optdata(ex)
    elseif length(expressions) == 2
        optex = expressions[1]
        ex = expressions[2]
        optdict = optdata(optex, ex)
    end

    parser = TensorParser()
    parser.contractiontreebuilder = network->optimaltree(network, optdict)[1]
    return esc(parser(ex))
end

"""
    @tensoropt_verbose(optex, block)
    @tensoropt_verbose(block)

Similar to `@tensoropt`, but prints information details regarding the optimization
process to the standard output.
"""
macro tensoropt_verbose(expressions...)
    if length(expressions) == 1
        ex = expressions[1]
        optdict = optdata(ex)
    elseif length(expressions) == 2
        optex = expressions[1]
        ex = expressions[2]
        optdict = optdata(optex, ex)
````

```julia
    end

    parser = TensorParser()
    parser.contractiontreebuilder =
        network->optimaltree(network, optdict; verbose = true)[1]
    return esc(parser(ex))
end

macro optimalcontractiontree(expressions...)
    if length(expressions) == 1
        ex = expressions[1]
        optdict = optdata(ex)
    elseif length(expressions) == 2
        optex = expressions[1]
        ex = expressions[2]
        optdict = optdata(optex, ex)
    end

    if isassignment(ex) || isdefinition(ex)
        ex = getrhs(ex)
    end
    if !(ex.head == :call && ex.args[1] == :*)
        error("cannot compute optimal contraction tree for this expression")
    end
    network = [getindices(ex.args[k]) for k = 2:length(ex.args)]
    tree, cost = optimaltree(network, optdict)
    return tree, cost
end

"""
    @ncon(tensorlist, indexlist; order = ..., output = ...)

Contract the tensors in `tensorlist` (of type `Vector` or `Tuple`) according to the
network
as specified by `indexlist`. Here, `indexlist` is a list (i.e. a `Vector` or
`Tuple`) with
the same length as `tensorlist` whose entries are themselves lists (preferably
`Vector{Int}`) where every integer entry provides a label for corresponding
index/dimension
of the corresponding tensor in `tensorlist`. Positive integers are used to label
indices
that need to be contracted, and such thus appear in two different entries within
`indexlist`, whereas negative integers are used to label indices of the output
tensor, and
should appear only once.

By default, contractions are performed in the order such that the indices being
contracted
over are labelled by increasing integers, i.e. first the contraction corresponding
to label
`1` is performed. The output tensor had an index order corresponding to decreasing
(negative, so increasing in absolute value) index labels. The keyword arguments
`order` and
`output` allow to change these defaults.
```

```julia
The advantage of the macro `@ncon` over the function call `ncon` is that the former
automatically generates a unique symbol that hooks into the cache. Furthermore, if
`tensorlist` is not just some variable but an actual list (as a tuple with
parentheses or a
vector with square brackets) at the call site, the `@ncon` macro will scan for
conjugation
calls, e.g. `conj(A)`, and replace this with just `A` but build a matching list of
conjugation flags to be specified to `ncon`. This makes it more convenient to
specify
tensor conjugation, without paying the cost of actively performing the conjugation
beforehand.

See also the function [`ncon`](@ref).
"""
macro ncon(args...)
    if length(args) == 2
        return _nconmacro(args[1], args[2])
    else
        return _nconmacro(args[2], args[3], args[1])
    end
end
function _nconmacro(tensors, indices, kwargs = nothing)
    if !(tensors isa Expr) # there is not much that we can do
        if kwargs === nothing
            ex = Expr(:call, :ncon, tensors, indices,
                        Expr(:call, :fill, false, Expr(:call, :length, tensors)),
                        QuoteNode(gensym()))
        else
            ex = Expr(:call, :ncon, kwargs, tensors, indices,
                        Expr(:call, :fill, false, Expr(:call, :length, tensors)),
                        QuoteNode(gensym()))
        end
        return esc(ex)
    end
    if tensors.head == :vect || tensors.head == :tuple
        tensorargs = tensors.args
    elseif tensors.head == :ref
        tensorargs = tensors.args[2:end]
    else
        throw(ArgumentError("invalid @ncon syntax"))
    end
    if any(isa(ta, Expr) && ta.head === :... for ta in tensorargs)
        throw(ArgumentError("@ncon does not support splats (...) in tensor lists."))
    end
    conjlist = fill(false, length(tensorargs))
    for i = 1:length(tensorargs)
        if tensorargs[i] isa Expr
            if tensorargs[i].head == :call && tensorargs[i].args[1] == :conj
                tensorargs[i] = tensorargs[i].args[2]
                conjlist[i] = true
            end
        end
    end
    if tensors.head == :ref
        tensorex = Expr(:ref, tensors.args[1], tensorargs...)
```

```
    else
        tensorex = Expr(:ref, :Any, tensorargs...)
    end
    if kwargs === nothing
        ex = Expr(:call, :ncon, tensorex, indices, conjlist, QuoteNode(gensym()))
    else
        ex = Expr(:call, :ncon, kwargs, tensorex, indices, conjlist,
QuoteNode(gensym()))
    end
    return esc(ex)
end
```