```
mutable struct TensorParser
    preprocessors::Vector{Any} # any preprocessing steps
    contractiontreebuilder::Any # determine a contraction tree for a contraction
        involving multiple tensors
    contractiontreesorter::Any # transforms the contraction expression into an
        expression of nested binary contractions using the tree output from the
        contractiontreebuilder

    postprocessors::Vector{Any}
    function TensorParser()
        preprocessors = [normalizeindices,
                         expandconj,
                         nconindexcompletion,
                         extracttensorobjects]
        contractiontreebuilder = defaulttreebuilder
        contractiontreesorter = defaulttreesorter
        postprocessors = [_flatten, removelinenumbernode, addtensoroperations]
        return new(preprocessors,
                   contractiontreebuilder,
                   contractiontreesorter,
                   postprocessors)
    end
end

function (parser::TensorParser)(ex::Expr)
    if ex isa Expr && ex.head == :function
        return Expr(:function, ex.args[1], parser(ex.args[2]))
    end
    for p in parser.preprocessors
        ex = p(ex)::Expr
    end
    treebuilder = parser.contractiontreebuilder
    treesorter = parser.contractiontreesorter
    ex = processcontractions(ex, treebuilder, treesorter)::Expr
    ex = tensorify(ex)::Expr
    for p in parser.postprocessors
        ex = p(ex)::Expr
    end
    return ex
end

function processcontractions(ex::Expr, treebuilder, treesorter)
    if ex.head == :macrocall && ex.args[1] == Symbol("@notensor")
        return ex
    end
    ex = Expr(ex.head, map(e->processcontractions(e, treebuilder, treesorter),
ex.args)...)
    if istensorcontraction(ex) && length(ex.args) > 3
        args = ex.args[2:end]
        network = map(getindices, args)
        for a in getallindices(ex)
            count(a in n for n in network) <= 2 ||
                throw(ArgumentError("invalid tensor contraction: $ex"))
        end
```

```
        tree = treebuilder(network)
        ex = treesorter(args, tree)
    end
    return ex
end
processcontractions(ex, treebuilder, treesorter) = ex

function defaulttreesorter(args, tree)
    if isa(tree, Int)
        return args[tree]
    else
        return Expr(:call, :*,
                    defaulttreesorter(args, tree[1]),
                    defaulttreesorter(args, tree[2]))
    end
end

function defaulttreebuilder(network)
    if isnconstyle(network)
        tree = ncontree(network)
    else
        tree = Any[1,2]
        for k = 3:length(network)
            tree = Any[tree, k]
        end
    end
    return tree
end

# functions for parsing and processing tensor expressions
function tensorify(ex::Expr)
    if ex.head == :macrocall && ex.args[1] == Symbol("@notensor")
        return ex.args[3]
    end
    # assignment case
    if isassignment(ex) || isdefinition(ex)
        lhs, rhs = getlhs(ex), getrhs(ex)
        if isa(rhs, Expr) && rhs.head == :call && rhs.args[1] == :throw
            return rhs
        end

        # process left hand side
        if istensor(lhs) && istensorexpr(rhs)
            indices = getindices(rhs)
            if hastraceindices(lhs)
                err = "left hand side of an assignment should have unique indices:
$lhs"
                return :(throw(IndexError($err)))
            end
            dst, leftind, rightind = decomposetensor(lhs)
            if Set(vcat(leftind,rightind)) != Set(indices)
                err = "non-matching indices between left and right hand side: $ex"
                return :(throw(IndexError($err)))
            end
            if isassignment(ex)
```

```
                if ex.head == :(=)
                    return instantiate(dst, false, rhs, true, leftind, rightind)
                elseif ex.head == :(+=)
                    return instantiate(dst, true, rhs, 1, leftind, rightind)
                else
                    return instantiate(dst, true, rhs, -1, leftind, rightind)
                end
            else
                return Expr(:(=), dst, instantiate(nothing, false, rhs, true,
leftind, rightind, false))
            end
        elseif isassignment(ex) && isscalarexpr(lhs)
            if istensorexpr(rhs) && isempty(getindices(rhs))
                return Expr(ex.head, instantiate_scalar(lhs), Expr(:call, :scalar,
instantiate(nothing, false, rhs, true, [], [], true)))
            elseif isscalarexpr(rhs)
                return Expr(ex.head, instantiate_scalar(lhs),
instantiate_scalar(rhs))
            end
        else
            return ex # likely an error
        end
    end
    if ex.head == :block
        return Expr(ex.head, map(tensorify, ex.args)...)
    end
    if ex.head == :for
        return Expr(ex.head, ex.args[1], tensorify(ex.args[2]))
    end
    if ex.head == :function
        return Expr(ex.head, ex.args[1], tensorify(ex.args[2]))
    end
    # constructions of the form: a = @tensor ...
    if isscalarexpr(ex)
        return instantiate_scalar(ex)
    end
    if istensorexpr(ex)
        if !isempty(getindices(ex))
            err = "cannot evaluate $ex to a scalar: uncontracted indices"
            return :(throw(IndexError($err)))
        end
        return Expr(:call, :scalar, instantiate(nothing, false, ex, true, [], [],
true))
    end
    error("invalid syntax in @tensor macro: $ex")
end
tensorify(ex) = ex
```