```julia
@noinline not_planar_err() = throw(ArgumentError("not a planar diagram expression"))
@noinline not_planar_err(ex) = throw(ArgumentError("not a planar diagram expression: $ex"))

@nospecialize

macro planar(ex::Expr)
    return esc(planar_parser(ex))
end

function planar_parser(ex::Expr)
    parser = TO.TensorParser()

    pop!(parser.preprocessors) # remove TO.extracttensorobjects
    push!(parser.preprocessors, _conj_to_adjoint)
    treebuilder = parser.contractiontreebuilder
    treesorter = parser.contractiontreesorter
    push!(parser.preprocessors, ex->TO.processcontractions(ex, treebuilder, treesorter))
    push!(parser.preprocessors, ex->_check_planarity(ex))
    push!(parser.preprocessors, _extract_tensormap_objects)
    temporaries = Vector{Symbol}()
    push!(parser.preprocessors, ex->_decompose_planar_contractions(ex, temporaries))

    pop!(parser.postprocessors) # remove TO.addtensoroperations
    push!(parser.postprocessors, ex->_update_temporaries(ex, temporaries))
    push!(parser.postprocessors, ex->_annotate_temporaries(ex, temporaries))
    push!(parser.postprocessors, _add_modules)
    return parser(ex)
end

macro planar2(ex::Expr)
    return esc(planar2_parser(ex))
end

function planar2_parser(ex::Expr)
    parser = TO.TensorParser()

    braidingtensors = Vector{Any}()

    pop!(parser.preprocessors) # remove TO.extracttensorobjects
    push!(parser.preprocessors, _conj_to_adjoint)
    push!(parser.preprocessors, _extract_tensormap_objects2)
    push!(parser.preprocessors, _construct_braidingtensors)
    treebuilder = parser.contractiontreebuilder
    treesorter = parser.contractiontreesorter
    push!(parser.preprocessors, ex->TO.processcontractions(ex, treebuilder, treesorter))
    push!(parser.preprocessors, ex->_check_planarity(ex))
    temporaries = Vector{Symbol}()
    push!(parser.preprocessors, ex->_decompose_planar_contractions(ex, temporaries))

    pop!(parser.postprocessors) # remove TO.addtensoroperations
    push!(parser.postprocessors, ex->_update_temporaries(ex, temporaries))
    push!(parser.postprocessors, ex->_annotate_temporaries(ex, temporaries))
    push!(parser.postprocessors, _add_modules)
    return parser(ex)
end

function _conj_to_adjoint(ex::Expr)
    if ex.head == :call && ex.args[1] == :conj && TO.istensor(ex.args[2])
        obj, leftind, rightind = TO.decomposetensor(ex.args[2])
        return Expr(:typed_vcat, Expr(TO.prime, obj),
                    Expr(:tuple, rightind...), Expr(:tuple, leftind...))
    else
        return Expr(ex.head, [_conj_to_adjoint(a) for a in ex.args]...)
    end
end
_conj_to_adjoint(ex) = ex

function get_possible_planar_indices(ex::Expr)
```

```julia
    @assert TO.istensorexpr(ex)
    if TO.isgeneraltensor(ex)
        _,leftind,rightind = TO.decomposegeneraltensor(ex)
        ind = planar_unique2(vcat(leftind, reverse(rightind)))
        return length(ind) == length(unique(ind)) ? Any[ind] : Any[]
    elseif ex.head == :call && (ex.args[1] == :+ || ex.args[1] == :-)
        inds = get_possible_planar_indices(ex.args[2])
        keep = fill(true, length(inds))
        for i = 3:length(ex.args)
            inds′ = get_possible_planar_indices(ex.args[i])
            keepᵢ = fill(false, length(inds))
            for (j, ind) in enumerate(inds), ind′ in inds′
                if iscyclicpermutation(ind′, ind)
                    keepᵢ[j] = true
                end
            end
            keep .&= keepᵢ
            any(keep) || break # give up early if keep is all false
        end
        return inds[keep]
    elseif ex.head == :call && ex.args[1] == :*
        @assert length(ex.args) == 3
        inds1 = get_possible_planar_indices(ex.args[2])
        inds2 = get_possible_planar_indices(ex.args[3])
        inds = Any[]
        for ind1 in inds1, ind2 in inds2
            for (oind1, oind2, cind1, cind2) in possible_planar_complements(ind1, ind2)
                push!(inds, vcat(oind1, oind2))
            end
        end
        return inds
    else
        return Any[]
    end
end

# remove double indices (trace indices) from cyclic set
function planar_unique2(allind)
    oind = collect(allind)
    removing = true
    while removing
        removing = false
        i = 1
        while i <= length(oind) && length(oind) > 1
            j = mod1(i+1, length(oind))
            if oind[i] == oind[j]
                deleteat!(oind, i)
                deleteat!(oind, mod1(i, length(oind)))
                removing = true
            else
                i += 1
            end
        end
    end
    return oind
end

# remove intersection (contraction indices) from two cyclic sets
function possible_planar_complements(ind1, ind2)
    # quick return path
    (isempty(ind1) || isempty(ind2)) && return Any[(ind1, ind2, Any[], Any[])]
    # general case:
    j1 = findfirst(in(ind2), ind1)
    if j1 === nothing # disconnected diagrams, can be made planar in various ways
        return Any[(circshift(ind1, i-1), circshift(ind2, j-1), Any[], Any[])
                    for i ∈ eachindex(ind1), j ∈ eachindex(ind2)]
    else # genuine contraction
        N1, N2 = length(ind1), length(ind2)
```

```julia
        j2 = findfirst(==(ind1[j1]), ind2)
        jmax1 = j1
        jmin2 = j2
        while jmax1 < N1 && ind1[jmax1+1] == ind2[mod1(jmin2-1, N2)]
            jmax1 += 1
            jmin2 -= 1
        end
        jmin1 = j1
        jmax2 = j2
        if j1 == 1 && jmax1 < N1
            while ind1[mod1(jmin1-1, N1)] == ind2[mod1(jmax2 + 1, N2)]
                jmin1 -= 1
                jmax2 += 1
            end
        end
        if jmax2 > N2
            jmax2 -= N2
            jmin2 -= N2
        end
        indo1 = jmin1 < 1 ? ind1[(jmax1+1):mod1(jmin1-1, N1)] :
                    vcat(ind1[(jmax1+1):N1], ind1[1:(jmin1-1)])
        cind1 = jmin1 < 1 ? vcat(ind1[mod1(jmin1, N1):N1], ind1[1:jmax1]) : ind1[jmin1:jmax1]
        indo2 = jmin2 < 1 ? ind2[(jmax2+1):mod1(jmin2-1, N2)] :
                    vcat(ind2[(jmax2+1):N2], ind2[1:(jmin2-1)])
        cind2 = reverse(cind1)
        return isempty(intersect(indo1, indo2)) ? Any[(indo1, indo2, cind1, cind2)] : Any[]
    end
end

function _check_planarity(ex::Expr)
    if ex.head == :macrocall && ex.args[1] == Symbol("@notensor")
    elseif TO.isassignment(ex) || TO.isdefinition(ex)
        lhs, rhs = TO.getlhs(ex), TO.getrhs(ex)
        if TO.istensorexpr(rhs)
            if TO.istensorexpr(lhs)
                @assert TO.istensor(lhs)
                indlhs = only(get_possible_planar_indices(lhs)) # should have only one element
            else
                indlhs = Any[]
            end
            indsrhs = get_possible_planar_indices(rhs)
            isempty(indsrhs) && not_planar_err(rhs)
            i = findfirst(ind -> iscyclicpermutation(ind, indlhs), indsrhs)
            i === nothing && not_planar_err(ex)
        end
    else
        foreach(ex.args) do a
            _check_planarity(a)
        end
    end
    return ex
end
_check_planarity(ex) = ex

# decompose contraction trees in order to fix index order of temporaries
# to ensure that planarity is guaranteed
_decompose_planar_contractions(ex, temporaries) = ex
function _decompose_planar_contractions(ex::Expr, temporaries)
    if ex.head == :macrocall && ex.args[1] == Symbol("@notensor")
        return ex
    end
    if TO.isassignment(ex) || TO.isdefinition(ex)
        lhs, rhs = TO.getlhs(ex), TO.getrhs(ex)
        if TO.istensorexpr(rhs)
            pre = Vector{Any}()
            rhs = _extract_contraction_pairs(rhs, lhs, pre, temporaries)
            return Expr(:block, pre..., Expr(ex.head, lhs, rhs))
        else
```

```
            return ex
        end
    end
    if TO.istensorexpr(ex)
        pre = Vector{Any}()
        rhs = _extract_contraction_pairs(ex, (Any[], Any[]), pre, temporaries)
        return Expr(:block, pre..., rhs)
    end
    if ex.head == :block
        return Expr(ex.head,
                    [_decompose_planar_contractions(a, temporaries) for a in ex.args]...)
    end
    if ex.head == :for || ex.head == :function
        return Expr(ex.head, ex.args[1],
                        _decompose_planar_contractions(ex.args[2], temporaries))
    end
    return ex
end

# decompose a contraction into elementary binary contractions of tensors without inner traces
# if lhs is an expression, it contains the existing lhs and thus the index order
# if lhs is a tuple, the result is a temporary object and the tuple (lind, rind) gives a suggestion for the
  preferred index order
function _extract_contraction_pairs(rhs, lhs, pre, temporaries)
    if TO.isgeneraltensor(rhs)
        if TO.hastraceindices(rhs) && lhs isa Tuple
            s = gensym()
            newlhs = Expr(:typed_vcat, s, Expr(:tuple, lhs[1]...), Expr(:tuple, lhs[2]...))
            push!(temporaries, s)
            push!(pre, Expr(:(:=), newlhs, rhs))
            return newlhs
        else
            return rhs
        end
    elseif rhs.head == :call && rhs.args[1] == :*
        @assert length(rhs.args) == 3

        if lhs isa Expr
            _, leftind, rightind = TO.decomposetensor(lhs)
        else
            leftind, rightind = lhs
        end
        lhs_ind = vcat(leftind, reverse(rightind))

        # find possible planar order
        rhs_inds = Any[]
        for ind1 in get_possible_planar_indices(rhs.args[2])
            for ind2 in get_possible_planar_indices(rhs.args[3])
                for (oind1, oind2, cind1, cind2) in possible_planar_complements(ind1, ind2)
                    if iscyclicpermutation(vcat(oind1, oind2), lhs_ind)
                        push!(rhs_inds, (ind1, ind2, oind1, oind2, cind1, cind2))
                    end
                    isempty(rhs_inds) || break
                end
                isempty(rhs_inds) || break
            end
            isempty(rhs_inds) || break
        end
        ind1, ind2, oind1, oind2, cind1, cind2 = only(rhs_inds) # inds_rhs should hold exactly one match
        if all(in(leftind), oind2) && all(in(rightind), oind1) # reverse order
            a1 = _extract_contraction_pairs(rhs.args[3], (oind2, reverse(cind2)), pre, temporaries)
            a2 = _extract_contraction_pairs(rhs.args[2], (cind1, reverse(oind1)), pre, temporaries)
            oind1, oind2 = oind2, oind1
            cind1, cind2 = cind2, cind1
        else
            a1 = _extract_contraction_pairs(rhs.args[2], (oind1, reverse(cind1)), pre, temporaries)
            a2 = _extract_contraction_pairs(rhs.args[3], (cind2, reverse(oind2)), pre, temporaries)
        end
```

```
        # note that index order in _extract... is only a suggestion, now we have actual index order
        _, l1, r1, = TO.decomposegeneraltensor(a1)
        _, l2, r2, = TO.decomposegeneraltensor(a2)
        if all(in(r1), oind1) && all(in(l2), oind2) # reverse order
            a1, a2 = a2, a1
            ind1, ind2 = ind2, ind1
            oind1, oind2 = oind2, oind1
        end
        if lhs isa Tuple
            rhs = Expr(:call, :*, a1, a2)
            s = gensym()
            newlhs = Expr(:typed_vcat, s, Expr(:tuple, oind1...),
                                            Expr(:tuple, reverse(oind2)...))
            push!(temporaries, s)
            push!(pre, Expr(:(:=), newlhs, rhs))
            return newlhs
        else
            if leftind == oind1 && rightind == reverse(oind2)
                rhs = Expr(:call, :*, a1, a2)
                return rhs
            elseif leftind == oind2 && rightind == reverse(oind1) # probably this can not happen anymore
                rhs = Expr(:call, :*, a2, a1)
                return rhs
            else
                rhs = Expr(:call, :*, a1, a2)
                s = gensym()
                newlhs = Expr(:typed_vcat, s, Expr(:tuple, oind1...),
                                                Expr(:tuple, reverse(oind2)...))
                push!(temporaries, s)
                push!(pre, Expr(:(:=), newlhs, rhs))
                return newlhs
            end
        end
    elseif rhs.head == :call && rhs.args[1] ∈ (:+, :-)
        args = [_extract_contraction_pairs(a, lhs, pre, temporaries) for
                    a in rhs.args[2:end]]
        return Expr(rhs.head, rhs.args[1], args...)
    else
        throw(ArgumentError("unknown tensor expression"))
    end
end

# replacement of TensorOperations functionality:
# adds checks for matching number of domain and codomain indices
# special cases adjoints so that t and t' are considered the same object
function _extract_tensormap_objects(ex)
    inputtensors = _remove_adjoint.(TO.getinputtensorobjects(ex))
    outputtensors = _remove_adjoint.(TO.getoutputtensorobjects(ex))
    newtensors = TO.getnewtensorobjects(ex)
    @assert !any(_is_adjoint, newtensors)
    existingtensors = unique!(vcat(inputtensors, outputtensors))
    alltensors = unique!(vcat(existingtensors, newtensors))
    tensordict = Dict{Any,Any}(a => gensym() for a in alltensors)
    pre = Expr(:block, [Expr(:(=), tensordict[a], a) for a in existingtensors]...)
    pre2 = Expr(:block)
    ex = TO.replacetensorobjects(ex) do obj, leftind, rightind
        _is_adj = _is_adjoint(obj)
        if _is_adj
            leftind, rightind = rightind, leftind
            obj = _remove_adjoint(obj)
        end
        newobj = get(tensordict, obj, obj)
        if (obj in existingtensors)
            nl = length(leftind)
            nr = length(rightind)
            nlsym = gensym()
            nrsym = gensym()
            objstr = string(obj)
```

```julia
            errorstr1 = "incorrect number of input-output indices: ($nl, $nr) instead of "
            errorstr2 = " for $objstr."
            checksize = quote
                $nlsym = numout($newobj)
                $nrsym = numin($newobj)
                ($nlsym == $nl && $nrsym == $nr) ||
                    throw(IndexError($errorstr1 * string(($nlsym, $nrsym)) * $errorstr2))
            end
            push!(pre2.args, checksize)
        end
        return _is_adj ? _restore_adjoint(newobj) : newobj
    end
    post = Expr(:block, [Expr(:(=), a, tensordict[a]) for a in newtensors]...)
    pre = Expr(:macrocall, Symbol("@notensor"), LineNumberNode(@__LINE__, Symbol(@__FILE__)), pre)
    pre2 = Expr(:macrocall, Symbol("@notensor"), LineNumberNode(@__LINE__, Symbol(@__FILE__)), pre2)
    post = Expr(:macrocall, Symbol("@notensor"), LineNumberNode(@__LINE__, Symbol(@__FILE__)), post)
    return Expr(:block, pre, pre2, ex, post)
end
_is_adjoint(ex) = isa(ex, Expr) && ex.head == TO.prime
_remove_adjoint(ex) = _is_adjoint(ex) ? ex.args[1] : ex
_restore_adjoint(ex) = Expr(TO.prime, ex)

function _extract_tensormap_objects2(ex)
    inputtensors = collect(filter(!=(:τ), _remove_adjoint.(TO.getinputtensorobjects(ex))))
    outputtensors = _remove_adjoint.(TO.getoutputtensorobjects(ex))
    newtensors = TO.getnewtensorobjects(ex)
    (any(==(:τ), newtensors) || any(==(:τ), outputtensors)) &&
        throw(ArgumentError("The name τ is reserved for the braiding, and should not be assigned to."))
    @assert !any(_is_adjoint, newtensors)
    existingtensors = unique!(vcat(inputtensors, outputtensors))
    alltensors = unique!(vcat(existingtensors, newtensors))
    tensordict = Dict{Any,Any}(a => gensym() for a in alltensors)
    pre = Expr(:block, [Expr(:(=), tensordict[a], a) for a in existingtensors]...)
    pre2 = Expr(:block)
    ex = TO.replacetensorobjects(ex) do obj, leftind, rightind
        _is_adj = _is_adjoint(obj)
        if _is_adj
            leftind, rightind = rightind, leftind
            obj = _remove_adjoint(obj)
        end
        newobj = get(tensordict, obj, obj)
        if (obj in existingtensors)
            nl = length(leftind)
            nr = length(rightind)
            nlsym = gensym()
            nrsym = gensym()
            objstr = string(obj)
            errorstr1 = "incorrect number of input-output indices: ($nl, $nr) instead of "
            errorstr2 = " for $objstr."
            checksize = quote
                $nlsym = numout($newobj)
                $nrsym = numin($newobj)
                ($nlsym == $nl && $nrsym == $nr) ||
                    throw(IndexError($errorstr1 * string(($nlsym, $nrsym)) * $errorstr2))
            end
            push!(pre2.args, checksize)
        end
        return _is_adj ? _restore_adjoint(newobj) : newobj
    end
    post = Expr(:block, [Expr(:(=), a, tensordict[a]) for a in newtensors]...)
    pre = Expr(:macrocall, Symbol("@notensor"), LineNumberNode(@__LINE__, Symbol(@__FILE__)), pre)
    pre2 = Expr(:macrocall, Symbol("@notensor"), LineNumberNode(@__LINE__, Symbol(@__FILE__)), pre2)
    post = Expr(:macrocall, Symbol("@notensor"), LineNumberNode(@__LINE__, Symbol(@__FILE__)), post)
    return Expr(:block, pre, pre2, ex, post)
end

function _construct_braidingtensors(ex::Expr)
    if TO.isdefinition(ex) || TO.isassignment(ex)
```

```
        lhs, rhs = TO.getlhs(ex), TO.getrhs(ex)
        if TO.istensorexpr(rhs)
            list = TO.gettensors(rhs)
            if TO.isassignment(ex) && istensor(lhs)
                obj, l, r = TO.decomposetensor(lhs)
                lhs_as_rhs = Expr(:typed_vcat, Expr(TO.prime, obj), Expr(:tuple, r...), Expr(:tuple, l...))
                push!(list, lhs_as_rhs)
            end
        else
            return ex
        end
    elseif TO.istensorexpr(ex)
        list = TO.gettensors(ex)
    else
        return Expr(ex.head, map(_construct_braidingtensors, ex.args)...)
    end

    i = 1
    translatebraidings = Dict{Any,Any}()
    while i <= length(list)
        t = list[i]
        if _remove_adjoint(TO.gettensorobject(t)) == :τ
            translatebraidings[t] = Expr(:call, GlobalRef(TensorKit, :BraidingTensor))
            deleteat!(list, i)
        else
            i += 1
        end
    end
    pre = Expr(:block)
    for (t, construct_expr) in translatebraidings
        obj, leftind, rightind = TO.decomposetensor(t)
        length(leftind) == length(rightind) == 2 ||
            throw(ArgumentError("The name τ is reserved for the braiding, and should have two input and two output
indices."))
        if _is_adjoint(obj)
            i1b, i2b, = leftind
            i2a, i1a, = rightind
        else
            i2b, i1b, = leftind
            i1a, i2a, = rightind
        end
        obj_and_pos = _findindex(i1a, list)
        if !isnothing(obj_and_pos)
            push!(construct_expr.args, Expr(:call, :space, obj_and_pos...))
        else
            obj_and_pos = _findindex(i1b, list)
            isnothing(obj_and_pos) &&
                throw(ArgumentError("cannot determine space of index $i1a and $i1b of braiding tensor"))
            push!(construct_expr.args, Expr(TO.prime, Expr(:call, :space, obj_and_pos...)))
        end

        obj_and_pos = _findindex(i2a, list)
        if !isnothing(obj_and_pos)
            push!(construct_expr.args, Expr(:call, :space, obj_and_pos...))
        else
            obj_and_pos = _findindex(i2b, list)
            isnothing(obj_and_pos) &&
                throw(ArgumentError("cannot determine space of index $i2a and $i2b of braiding tensor"))
            push!(construct_expr.args, Expr(TO.prime, Expr(:call, :space, obj_and_pos...)))
        end
        s = gensym()
        push!(pre.args, Expr(:(=), s, construct_expr))
        ex = TO.replacetensorobjects(ex) do o, l, r
            if o == obj && l == leftind && r == rightind
                return obj  == :τ ? s : Expr(TO.prime, s)
            else
                return o
            end
```

```
        end
    end
    return Expr(:block, pre, ex)
end
_construct_braidingtensors(x) = x

function _findindex(i, list) # finds an index i in a list of tensor expressions
    for t in list
        obj, l, r = TO.decomposetensor(t)
        pos = findfirst(==(i), l)
        isnothing(pos) || return (obj, pos)
        pos = findfirst(==(i), r)
        isnothing(pos) || return (obj, pos + length(l))
    end
    return nothing
end

# since temporaries were taken out in preprocessing, they are not identified by the parsing
# step of TensorOperations, and we have to manually fix this
# Step 1: we have to find the new name that TO.tensorify assigned to these temporaries
# since it parses `tmp[] := a[] * b[]` as `newtmp = similar...; tmp = contract!(.... , newtmp, ...)`
function _update_temporaries(ex, temporaries)
    if ex isa Expr && ex.head == :(=)
        lhs = ex.args[1]
        i = findfirst(==(lhs), temporaries)
        if i !== nothing
            rhs = ex.args[2]
            if !(rhs isa Expr && rhs.head == :call && rhs.args[1] == :contract!)
                @error "lhs = $lhs , rhs = $rhs"
            end
            newname = rhs.args[8]
            temporaries[i] = newname
        end
    elseif ex isa Expr
        for a in ex.args
            _update_temporaries(a, temporaries)
        end
    end
    return ex
end
# Step 2: we find `newtmp = similar_from_...` and replace with `newtmp = cached_similar_from...`
function _annotate_temporaries(ex, temporaries)
    if ex isa Expr && ex.head == :(=)
        lhs = ex.args[1]
        i = findfirst(==(lhs), temporaries)
        if i !== nothing
            rhs = ex.args[2]
            if !(rhs isa Expr && rhs.head == :call && rhs.args[1] == :similar_from_indices)
                @error "lhs = $lhs , rhs = $rhs"
            end
            newrhs = Expr(:call, :cached_similar_from_indices,
                                QuoteNode(lhs), rhs.args[2:end]...)
            return Expr(:(=), lhs, newrhs)
        end
    elseif ex isa Expr
        return Expr(ex.head, [_annotate_temporaries(a, temporaries) for a in ex.args]...)
    end
    return ex
end

const _TOFUNCTIONS = (:similar_from_indices, :cached_similar_from_indices,
                        :scalar, :IndexError)

function _add_modules(ex::Expr)
    if ex.head == :call && ex.args[1] in _TOFUNCTIONS
        return Expr(ex.head, GlobalRef(TensorOperations, ex.args[1]),
                        (_add_modules(ex.args[i]) for i in 2:length(ex.args))...)
    elseif ex.head == :call && ex.args[1] == :add!
```

```julia
        @assert ex.args[4] == :(:N)
        argind = [2,3,5,6,7,8]
        return Expr(ex.head, GlobalRef(TensorKit, Symbol(:planar_add!)),
                        (_add_modules(ex.args[i]) for i in argind)...)
    elseif ex.head == :call && ex.args[1] == :trace!
        @assert ex.args[4] == :(:N)
        argind = [2,3,5,6,7,8,9,10]
        return Expr(ex.head, GlobalRef(TensorKit, Symbol(:planar_trace!)),
                        (_add_modules(ex.args[i]) for i in argind)...)
    elseif ex.head == :call && ex.args[1] == :contract!
        @assert ex.args[4] == :(:N) && ex.args[6] == :(:N)
        argind = vcat([2,3,5], 7:length(ex.args))
        return Expr(ex.head, GlobalRef(TensorKit, Symbol(:planar_contract!)),
                        (_add_modules(ex.args[i]) for i in argind)...)
    else
        return Expr(ex.head, (_add_modules(e) for e in ex.args)...)
    end
end
_add_modules(ex) = ex

@specialize

planar_add!(α, tsrc::AbstractTensorMap{S},
            β, tdst::AbstractTensorMap{S, N₁, N₂},
            p1::IndexTuple{N₁}, p2::IndexTuple{N₂}) where {S, N₁, N₂} =
    add_transpose!(α, tsrc, β, tdst, p1, p2)

function planar_trace!(α, tsrc::AbstractTensorMap{S},
                        β, tdst::AbstractTensorMap{S, N₁, N₂},
                        p1::IndexTuple{N₁}, p2::IndexTuple{N₂},
                        q1::IndexTuple{N₃}, q2::IndexTuple{N₃}) where {S, N₁, N₂, N₃}
    if BraidingStyle(sectortype(S)) == Bosonic()
        return trace!(α, tsrc, β, tdst, p1, p2, q1, q2)
    end

    @boundscheck begin
        all(i->space(tsrc, p1[i]) == space(tdst, i), 1:N₁) ||
            throw(SpaceMismatch("trace: tsrc = $(codomain(tsrc))←$(domain(tsrc)),
                    tdst = $(codomain(tdst))←$(domain(tdst)), p1 = $(p1), p2 = $(p2)"))
        all(i->space(tsrc, p2[i]) == space(tdst, N₁+i), 1:N₂) ||
            throw(SpaceMismatch("trace: tsrc = $(codomain(tsrc))←$(domain(tsrc)),
                    tdst = $(codomain(tdst))←$(domain(tdst)), p1 = $(p1), p2 = $(p2)"))
        all(i->space(tsrc, q1[i]) == dual(space(tsrc, q2[i])), 1:N₃) ||
            throw(SpaceMismatch("trace: tsrc = $(codomain(tsrc))←$(domain(tsrc)),
                    q1 = $(q1), q2 = $(q2)"))
    end

    if iszero(β)
        fill!(tdst, β)
    elseif β != 1
        rmul!(tdst, β)
    end
    pdata = (p1..., p2...)
    for (f1, f2) in fusiontrees(tsrc)
        for ((f1′, f2′), coeff) in planar_trace(f1, f2, p1, p2, q1, q2)
            TO._trace!(α*coeff, tsrc[f1, f2], true, tdst[f1′, f2′], pdata, q1, q2)
        end
    end
    return tdst
end

_cyclicpermute(t::Tuple) = (Base.tail(t)..., t[1])
_cyclicpermute(t::Tuple{}) = ()
function reorder_indices(codA, domA, codB, domB, oindA, oindB, p1, p2)
    N₁ = length(oindA)
    N₂ = length(oindB)
    @assert length(p1) == N₁ && all(in(p1), 1:N₁)
    @assert length(p2) == N₂ && all(in(p2), N₁ .+ (1:N₂))
```

```julia
    oindA2 = TupleTools.getindices(oindA, p1)
    oindB2 = TupleTools.getindices(oindB, p2 .- N₁)
    indA = (codA..., reverse(domA)...)
    indB = (codB..., reverse(domB)...)
    # cycle indA to be of the form (oindA2..., reverse(cindA2)...)
    while length(oindA2) > 0 && indA[1] != oindA2[1]
        indA = _cyclicpermute(indA)
    end
    # cycle indA to be of the form (cindB2..., reverse(oindB2)...)
    while length(oindB2) > 0 && indB[end] != oindB2[1]
        indB = _cyclicpermute(indB)
    end
    for i = 2:N₁
        @assert indA[i] == oindA2[i]
    end
    for j = 2:N₂
        @assert indB[end + 1 - j] == oindB2[j]
    end
    Nc = length(indA) - N₁
    @assert Nc == length(indB) - N₂
    pc = ntuple(identity, Nc)
    cindA2 = reverse(TupleTools.getindices(indA, N₁ .+ pc))
    cindB2 = TupleTools.getindices(indB, pc)
    return oindA2, cindA2, oindB2, cindB2
end
function reorder_indices(codA, domA, codB, domB, oindA, cindA, oindB, cindB, p1, p2)
    oindA2, cindA2, oindB2, cindB2 =
        reorder_indices(codA, domA, codB, domB, oindA, oindB, p1, p2)

    #if oindA or oindB are empty, then reorder indices can only order it correctly up to a cyclic permutation!
    if isempty(oindA2) && !isempty(cindA)
         # isempty(cindA) is a cornercase which I'm not sure if we can encounter
        hit = cindA[findfirst(==(first(cindB2)), cindB)];
        while hit != first(cindA2)
            cindA2 = _cyclicpermute(cindA2)
        end
    end
    if isempty(oindB2) && !isempty(cindB)
        hit = cindB[findfirst(==(first(cindA2)), cindA)]
        while hit != first(cindB2)
            cindB2 = _cyclicpermute(cindB2)
        end
    end
    @assert TupleTools.sort(cindA) == TupleTools.sort(cindA2)
    @assert TupleTools.sort(tuple.(cindA2, cindB2)) == TupleTools.sort(tuple.(cindA, cindB))
    return oindA2, cindA2, oindB2, cindB2
end

function planar_contract!(α, A::AbstractTensorMap{S}, B::AbstractTensorMap{S},
                            β, C::AbstractTensorMap{S},
                            oindA::IndexTuple{N₁}, cindA::IndexTuple,
                            oindB::IndexTuple{N₂}, cindB::IndexTuple,
                            p1::IndexTuple, p2::IndexTuple,
                            syms::Union{Nothing, NTuple{3, Symbol}} = nothing) where {S, N₁, N₂}

    codA = codomainind(A)
    domA = domainind(A)
    codB = codomainind(B)
    domB = domainind(B)
    oindA, cindA, oindB, cindB =
        reorder_indices(codA, domA, codB, domB, oindA, cindA, oindB, cindB, p1, p2)

    if oindA == codA && cindA == domA
        A′ = A
    else
        if isnothing(syms)
            A′ = TO.similar_from_indices(eltype(A), oindA, cindA, A, :N)
        else
```

```julia
            A′ = TO.cached_similar_from_indices(syms[1], eltype(A), oindA, cindA, A, :N)
        end
        add_transpose!(true, A, false, A′, oindA, cindA)
    end
    if cindB == codB && oindB == domB
        B′ = B
    else
        if isnothing(syms)
            B′ = TO.similar_from_indices(eltype(B), cindB, oindB, B, :N)
        else
            B′ = TO.cached_similar_from_indices(syms[2], eltype(B), cindB, oindB, B, :N)
        end
        add_transpose!(true, B, false, B′, cindB, oindB)
    end
    mul!(C, A′, B′, α, β)
    return C
end
```