```
@noinline not_planar_err() = throw(ArgumentError("not a planar diagram"))

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

@nospecialize

function planar_parser(ex::Expr)
    parser = TO.TensorParser()
    parser.preprocessors[end] = _extract_tensormap_objects
    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))
    temporaries = Vector{Symbol}()
    push!(parser.preprocessors, ex->_decompose_planar_contractions(ex,
temporaries))
    deleteat!(parser.postprocessors, length(parser.postprocessors))
    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(:call, :adjoint, 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_planar_indices(ex::Expr)
    @assert TO.istensorexpr(ex)
    if TO.isgeneraltensor(ex)
        _,leftind,rightind = TO.decomposegeneraltensor(ex)
        ind = planar_unique2(vcat(leftind, reverse(rightind)))
        length(ind) == length(unique(ind)) || not_planar_err()
        return ind
    elseif ex.head == :call && (ex.args[1] == :+ || ex.args[1] == :-)
        ind = get_planar_indices(ex.args[2])
        for i = 3:length(ex.args)
            ind′ = get_planar_indices(ex.args[i])
            (length(ind) == length(ind′) && iscyclicpermutation(indexin(ind′,
ind))) ||
                not_planar_err()
        end
        return ind
```

```
    elseif ex.head == :call && ex.args[1] == :*
        @assert length(ex.args) == 3
        ind1 = get_planar_indices(ex.args[2])
        ind2 = get_planar_indices(ex.args[3])
        indo1, indo2 = planar_complement(ind1, ind2)
        isempty(intersect(indo1, indo2)) || not_planar_err()
        return vcat(indo1, indo2)
    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 planar_complement(ind1, ind2)
    j1 = findfirst(in(ind2), ind1)
    if j1 === nothing
        return ind1, ind2
    else
        N1, N2 = length(ind1), length(ind2)
        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)])
        indo2 = jmin2 < 1 ? ind2[(jmax2+1):mod1(jmin2-1, N2)] :
                    vcat(ind2[(jmax2+1):N2], ind2[1:(jmin2-1)])
        return indo1, indo2
    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)
            indlhs = TO.istensorexpr(lhs) ? get_planar_indices(lhs) : []
            indrhs = get_planar_indices(rhs)
            (length(indlhs) == length(indrhs) &&
                iscyclicpermutation(indexin(indrhs, indlhs))) || not_planar_err()
        end
    else
        foreach(ex.args) do a
            _check_planarity(a)
        end
    end
    return ex
end
_check_planarity(ex, leftind = nothing, rightind = nothing) = ex

_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 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

function _extract_contraction_pairs(rhs, lhs, pre, temporaries)
    if TO.isgeneraltensor(rhs)
        if TO.hastraceindices(rhs) && lhs === nothing
            s = gensym()
            ind = get_planar_indices(rhs)
            lhs = Expr(:typed_vcat, s, Expr(:tuple, ind...), Expr(:tuple))
            push!(temporaries, s)
            push!(pre, Expr(:(:=), lhs, rhs))
            return lhs
        else
            return rhs
        end
    elseif rhs.head == :call && rhs.args[1] == :*
        @assert length(rhs.args) == 3
        a1 = _extract_contraction_pairs(rhs.args[2], nothing, pre, temporaries)
        a2 = _extract_contraction_pairs(rhs.args[3], nothing, pre, temporaries)
        ind1 = get_planar_indices(a1)
        ind2 = get_planar_indices(a2)
        oind1, oind2 = planar_complement(ind1, ind2)
        _, l1, r1, = TO.decomposegeneraltensor(a1)
        _, l2, r2, = TO.decomposegeneraltensor(a2)
        if all(in(r1), oind1) && all(in(l2), oind2)
            a1, a2 = a2, a1
            ind1, ind2 = ind2, ind1
            oind1, oind2 = oind2, oind1
        end
        if lhs === nothing
            rhs = Expr(:call, :*, a1, a2)
            s = gensym()
            lhs = Expr(:typed_vcat, s, Expr(:tuple, oind1...),
                                       Expr(:tuple, reverse(oind2)...))
            push!(temporaries, s)
            push!(pre, Expr(:(:=), lhs, rhs))
            return lhs
        else
            _, leftind, rightind = TO.decomposetensor(lhs)
            if leftind == oind1 && rightind == reverse(oind2)
                rhs = Expr(:call, :*, a1, a2)
                return rhs
            elseif leftind == oind2 && rightind == reverse(oind1)
                rhs = Expr(:call, :*, a2, a1)
                return rhs
            else
                rhs = Expr(:call, :*, a1, a2)
                s = gensym()
                lhs = Expr(:typed_vcat, s, Expr(:tuple, oind1...),
                                           Expr(:tuple, reverse(oind2)...))
                push!(temporaries, s)
                push!(pre, Expr(:(:=), lhs, rhs))
```

```
                return lhs
            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

function _extract_tensormap_objects(ex)
    inputtensors = TO.getinputtensorobjects(ex)
    outputtensors = TO.getoutputtensorobjects(ex)
    newtensors = TO.getnewtensorobjects(ex)
    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
        newobj = get(tensordict, obj, obj)
        obj == newobj && return obj
        if !(obj in newtensors)
            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 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 _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

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]),
                    (ex.args[i] for i in 2:length(ex.args))...)
    elseif ex.head == :call && ex.args[1] == :add!
        @assert ex.args[4] == :(:N)
        argind = [2,3,5,6,7,8]
        return Expr(ex.head, GlobalRef(TensorKit, Symbol(:planar_add!)),
                    (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!)),
                        (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!)),
                        (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, cindA, oindB, cindB, p1,
p2)
    N₁ = length(oindA)
    N₂ = length(oindB)
    @assert all(x->x in p1, 1:N₁)
    @assert all(x->x in p2, N₁ .+ (1:N₂))
    oindA2 = TupleTools.getindices(oindA, p1)
    oindB2 = TupleTools.getindices(oindB, p2 .- N₁)
    indA = (codA..., reverse(domA)...)
    indB = (codB..., reverse(domB)...)
    while length(oindA2) > 0 && indA[1] != oindA2[1]
        indA = _cyclicpermute(indA)
    end
    while length(oindB2) > 0 && indB[1] != oindB2[end]
        indB = _cyclicpermute(indB)
    end
    cindA2 = reverse(TensorOperations.tsetdiff(indA, oindA2))
    cindB2 = TensorOperations.tsetdiff(indB, reverse(oindB2))
    @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}}) 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
        A′ = TO.cached_similar_from_indices(syms[1], eltype(A), oindA, cindA, A,
:N)
        add_transpose!(true, A, false, A′, oindA, cindA)
    end
    if cindB == codB && oindB == domB
```

```
        B′ = B
    else
        B′ = TO.cached_similar_from_indices(syms[2], eltype(B), cindB, oindB, B,
:N)
        add_transpose!(true, B, false, B′, cindB, oindB)
    end
    mul!(C, A′, B′, α, β)
    return C
end
```