//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

template <class ResultType, class Context>

struct WalkerPolicyBase

{

typedef ResultType ResultType;

typedef Context Context;

inline bool ContinueWalk(ResultType) { return true; }

inline ResultType DefaultResult() { return ResultType(); }

inline ResultType WalkNode(ParseNode \*pnode, Context context) { return DefaultResult(); }

inline ResultType WalkListNode(ParseNode \*pnode, Context context) { return DefaultResult(); }

virtual ResultType WalkChild(ParseNode \*pnode, Context context) { return DefaultResult(); }

inline ResultType WalkFirstChild(ParseNode \*pnode, Context context) { return WalkChild(pnode, context); }

inline ResultType WalkSecondChild(ParseNode \*pnode, Context context) { return WalkChild(pnode, context); }

inline ResultType WalkNthChild(ParseNode \*pparentnode, ParseNode \*pnode, Context context) { return WalkChild(pnode, context); }

inline void WalkReference(ParseNode \*\*ppnode, Context context) { }

};

template <class Context>

struct WalkerPolicyBase<bool, Context>

{

typedef bool ResultType;

typedef Context Context;

inline bool ContinueWalk(ResultType) { return true; }

inline bool DefaultResult() { return true; }

inline ResultType WalkNode(ParseNode \*pnode, Context context) { return DefaultResult(); }

inline ResultType WalkListNode(ParseNode \*pnode, Context context) { return DefaultResult(); }

virtual ResultType WalkChild(ParseNode \*pnode, Context context) { return DefaultResult(); }

inline ResultType WalkFirstChild(ParseNode \*pnode, Context context) { return WalkChild(pnode, context); }

inline ResultType WalkSecondChild(ParseNode \*pnode, Context context) { return WalkChild(pnode, context); }

inline ResultType WalkNthChild(ParseNode \*pparentnode, ParseNode \*pnode, Context context) { return WalkChild(pnode, context); }

inline void WalkReference(ParseNode \*\*ppnode, Context context) { }

};

template <typename WalkerPolicy>

class ParseNodeWalker : public WalkerPolicy

{

public:

typedef typename WalkerPolicy::Context Context;

protected:

typedef typename WalkerPolicy::ResultType ResultType;

private:

ResultType WalkList(ParseNode \*pnodeparent, ParseNode \*&pnode, Context context)

{

ResultType result = DefaultResult();

bool first = true;

if (pnode)

{

result = WalkListNode(pnode, context);

if (!ContinueWalk(result)) return result;

ParseNodePtr current = pnode;

ParseNodePtr \*ppnode = &pnode;

// Skip list nodes and nested VarDeclList nodes

while ((current->nop == knopList && (current->grfpn & PNodeFlags::fpnDclList) == 0) ||

(current->nop == pnode->nop && (current->grfpn & pnode->grfpn & PNodeFlags::fpnDclList)))

{

WalkReference(&current->sxBin.pnode1, context);

result = first ? WalkFirstChild(current->sxBin.pnode1, context) : WalkNthChild(pnodeparent, current->sxBin.pnode1, context);

first = false;

if (!ContinueWalk(result)) return result;

ppnode = &current->sxBin.pnode2;

current = \*ppnode;

}

WalkReference(ppnode, context);

result = first ? WalkFirstChild(\*ppnode, context) : WalkNthChild(pnodeparent, \*ppnode, context);

}

// Reset the reference back.

WalkReference(nullptr, context);

return result;

}

ResultType WalkLeaf(ParseNode \*pnode, Context context)

{

return WalkNode(pnode, context);

}

ResultType WalkPreUnary(ParseNode \*pnode, Context context)

{

ResultType result = WalkNode(pnode, context);

if (ContinueWalk(result) && pnode->sxUni.pnode1) result = WalkFirstChild(pnode->sxUni.pnode1, context);

return result;

}

ResultType WalkPostUnary(ParseNode \*pnode, Context context)

{

ResultType result = WalkFirstChild(pnode->sxUni.pnode1, context);

if (ContinueWalk(result)) result = WalkNode(pnode, context);

return result;

}

ResultType WalkBinary(ParseNode \*pnode, Context context)

{

ResultType result = WalkFirstChild(pnode->sxBin.pnode1, context);

if (ContinueWalk(result))

{

result = WalkNode(pnode, context);

if (ContinueWalk(result)) result = WalkSecondChild(pnode->sxBin.pnode2, context);

}

return result;

}

ResultType WalkTiernary(ParseNode \*pnode, Context context)

{

ResultType result = WalkFirstChild(pnode->sxTri.pnode1, context);

if (ContinueWalk(result))

{

result = WalkNode(pnode, context);

if (ContinueWalk(result))

{

result = WalkSecondChild(pnode->sxTri.pnode2, context);

if (ContinueWalk(result)) result = WalkNthChild(pnode, pnode->sxTri.pnode3, context);

}

}

return result;

}

ResultType WalkCall(ParseNode \*pnode, Context context)

{

ResultType result = WalkFirstChild(pnode->sxBin.pnode1, context);

if (ContinueWalk(result))

{

result = WalkNode(pnode, context);

if (ContinueWalk(result)) result = WalkList(pnode, pnode->sxBin.pnode2, context);

}

return result;

}

ResultType WalkStringTemplate(ParseNode \*pnode, Context context)

{

ResultType result;

if (!pnode->sxStrTemplate.isTaggedTemplate)

{

if (pnode->sxStrTemplate.pnodeSubstitutionExpressions == nullptr)

{

// If we don't have any substitution expressions, then we should only have one string literal and not a list

result = WalkNode(pnode->sxStrTemplate.pnodeStringLiterals, context);

}

else

{

result = WalkList(pnode, pnode->sxStrTemplate.pnodeSubstitutionExpressions, context);

if (ContinueWalk(result))

{

result = WalkList(pnode, pnode->sxStrTemplate.pnodeStringLiterals, context);

}

}

}

else

{

// Tagged template nodes are call nodes

result = WalkCall(pnode, context);

}

return result;

}

ResultType WalkVar(ParseNode \*pnode, Context context)

{

ResultType result = WalkNode(pnode, context);

if (ContinueWalk(result) && pnode->sxVar.pnodeInit) result = WalkFirstChild(pnode->sxVar.pnodeInit, context);

return result;

}

ResultType WalkFnc(ParseNode \*pnode, Context context)

{

ResultType result;

// For ordering, arguments are considered prior to the function and the body after.

for (ParseNode\*\* argNode = &(pnode->sxFnc.pnodeArgs); \*argNode != nullptr; argNode = &((\*argNode)->sxVar.pnodeNext))

{

result = \*argNode == pnode->sxFnc.pnodeArgs ? WalkFirstChild(\*argNode, context) : WalkNthChild(pnode, \*argNode, context);

if (!ContinueWalk(result)) return result;

}

if (pnode->sxFnc.pnodeRest != nullptr)

{

result = WalkSecondChild(pnode->sxFnc.pnodeRest, context);

if (!ContinueWalk(result)) return result;

}

result = WalkNode(pnode, context);

if (ContinueWalk(result)) result = WalkNthChild(pnode, pnode->sxFnc.pnodeBody, context);

return result;

}

ResultType WalkProg(ParseNode \*pnode, Context context)

{

ResultType result = WalkNode(pnode, context);

if (ContinueWalk(result)) result = WalkList(pnode, pnode->sxFnc.pnodeBody, context);

return result;

}

ResultType WalkFor(ParseNode \*pnode, Context context)

{

ResultType result = WalkFirstChild(pnode->sxFor.pnodeInit, context);

if (ContinueWalk(result))

{

result = WalkNthChild(pnode, pnode->sxFor.pnodeCond, context);

if (ContinueWalk(result))

{

result = WalkNthChild(pnode, pnode->sxFor.pnodeIncr, context);

if (ContinueWalk(result))

{

result = WalkNode(pnode, context);

if (ContinueWalk(result))

{

result = WalkSecondChild(pnode->sxFor.pnodeBody, context);

}

}

}

}

return result;

}

ResultType WalkIf(ParseNode \*pnode, Context context)

{

ResultType result = WalkFirstChild(pnode->sxIf.pnodeCond, context);

if (ContinueWalk(result))

{

result = WalkNode(pnode, context);

if (ContinueWalk(result))

{

result = WalkSecondChild(pnode->sxIf.pnodeTrue, context);

if (ContinueWalk(result) && pnode->sxIf.pnodeFalse)

result = WalkNthChild(pnode, pnode->sxIf.pnodeFalse, context);

}

}

return result;

}

ResultType WalkWhile(ParseNode \*pnode, Context context)

{

ResultType result = WalkFirstChild(pnode->sxWhile.pnodeCond, context);

if (ContinueWalk(result))

{

result = WalkNode(pnode, context);

if (ContinueWalk(result)) result = WalkSecondChild(pnode->sxWhile.pnodeBody, context);

}

return result;

}

ResultType WalkDoWhile(ParseNode \*pnode, Context context)

{

ResultType result = WalkFirstChild(pnode->sxWhile.pnodeBody, context);

if (ContinueWalk(result))

{

result = WalkNode(pnode, context);

if (ContinueWalk(result))

{

result = WalkSecondChild(pnode->sxWhile.pnodeCond, context);

}

}

return result;

}

ResultType WalkForInOrForOf(ParseNode \*pnode, Context context)

{

ResultType result = WalkFirstChild(pnode->sxForInOrForOf.pnodeLval, context);

if (ContinueWalk(result))

{

result = WalkNthChild(pnode, pnode->sxForInOrForOf.pnodeObj, context);

if (ContinueWalk(result))

{

result = WalkNode(pnode, context);

if (ContinueWalk(result)) result = WalkSecondChild(pnode->sxForInOrForOf.pnodeBody, context);

}

}

return result;

}

ResultType WalkReturn(ParseNode \*pnode, Context context)

{

ResultType result = WalkNode(pnode, context);

if (ContinueWalk(result) && pnode->sxReturn.pnodeExpr) result = WalkFirstChild(pnode->sxReturn.pnodeExpr, context);

return result;

}

ResultType WalkBlock(ParseNode \*pnode, Context context)

{

ResultType result = WalkNode(pnode, context);

if (ContinueWalk(result) && pnode->sxBlock.pnodeStmt)

result = WalkList(pnode, pnode->sxBlock.pnodeStmt, context);

return result;

}

ResultType WalkWith(ParseNode \*pnode, Context context)

{

ResultType result = WalkFirstChild(pnode->sxWith.pnodeObj, context);

if (ContinueWalk(result))

{

result = WalkNode(pnode, context);

if (ContinueWalk(result))

{

result = WalkSecondChild(pnode->sxWith.pnodeBody, context);

}

}

return result;

}

ResultType WalkSwitch(ParseNode \*pnode, Context context)

{

ResultType result = WalkFirstChild(pnode->sxSwitch.pnodeVal, context);

if (ContinueWalk(result))

{

for (ParseNode\*\* caseNode = &(pnode->sxSwitch.pnodeCases); \*caseNode != nullptr; caseNode = &((\*caseNode)->sxCase.pnodeNext))

{

result = \*caseNode == pnode->sxSwitch.pnodeCases ? WalkFirstChild(\*caseNode, context) : WalkNthChild(pnode, \*caseNode, context);

if (!ContinueWalk(result)) return result;

}

result = WalkNode(pnode, context);

}

return result;

}

ResultType WalkCase(ParseNode \*pnode, Context context)

{

ResultType result = WalkFirstChild(pnode->sxCase.pnodeExpr, context);

if (ContinueWalk(result))

{

result = WalkNode(pnode, context);

if (ContinueWalk(result)) result = WalkSecondChild(pnode->sxCase.pnodeBody, context);

}

return result;

}

ResultType WalkTryFinally(ParseNode \*pnode, Context context)

{

ResultType result = WalkFirstChild(pnode->sxTryFinally.pnodeTry, context);

if (ContinueWalk(result))

{

result = WalkNode(pnode, context);

if (ContinueWalk(result)) result = WalkSecondChild(pnode->sxTryFinally.pnodeFinally, context);

}

return result;

}

ResultType WalkFinally(ParseNode \*pnode, Context context)

{

ResultType result = WalkNode(pnode, context);

if (ContinueWalk(result)) result = WalkFirstChild(pnode->sxFinally.pnodeBody, context);

return result;

}

ResultType WalkCatch(ParseNode \*pnode, Context context)

{

ResultType result = WalkFirstChild(pnode->sxCatch.pnodeParam, context);

if (ContinueWalk(result))

{

result = WalkNode(pnode, context);

if (ContinueWalk(result)) result = WalkSecondChild(pnode->sxCatch.pnodeBody, context);

}

return result;

}

ResultType WalkTryCatch(ParseNode \*pnode, Context context)

{

ResultType result = WalkFirstChild(pnode->sxTryCatch.pnodeTry, context);

if (ContinueWalk(result))

{

result = WalkNode(pnode, context);

if (ContinueWalk(result)) result = WalkSecondChild(pnode->sxTryCatch.pnodeCatch, context);

}

return result;

}

ResultType WalkTry(ParseNode \*pnode, Context context)

{

ResultType result = WalkNode(pnode, context);

if (ContinueWalk(result)) result = WalkFirstChild(pnode->sxTry.pnodeBody, context);

return result;

}

ResultType WalkClass(ParseNode \*pnode, Context context)

{

// First walk the class node itself

ResultType result = WalkNode(pnode, context);

if (!ContinueWalk(result)) return result;

// Walk extends expr

result = WalkFirstChild(pnode->sxClass.pnodeExtends, context);

if (!ContinueWalk(result)) return result;

// Walk the constructor

result = WalkNthChild(pnode, pnode->sxClass.pnodeConstructor, context);

if (!ContinueWalk(result)) return result;

// Walk all non-static members

result = WalkList(pnode, pnode->sxClass.pnodeMembers, context);

if (!ContinueWalk(result)) return result;

// Walk all static members

result = WalkList(pnode, pnode->sxClass.pnodeStaticMembers, context);

return result;

}

public:

ResultType Walk(ParseNode \*pnode, Context context)

{

if (!pnode) return DefaultResult();

switch (pnode->nop) {

// Handle all special cases first.

// Post-fix unary operators.

//PTNODE(knopIncPost , "++ post" ,Inc ,Uni ,fnopUni|fnopAsg)

//PTNODE(knopDecPost , "-- post" ,Dec ,Uni ,fnopUni|fnopAsg)

case knopIncPost:

case knopDecPost:

return WalkPostUnary(pnode, context);

// Call and call like

//PTNODE(knopCall , "()" ,None ,Bin ,fnopBin)

//PTNODE(knopNew , "new" ,None ,Bin ,fnopBin)

//PTNODE(knopIndex , "[]" ,None ,Bin ,fnopBin)

case knopCall:

case knopNew:

case knopIndex:

return WalkCall(pnode, context);

// Tierinary operator

//PTNODE(knopQmark , "?" ,None ,Tri ,fnopBin)

case knopQmark:

return WalkTiernary(pnode, context);

// General nodes.

//PTNODE(knopList , "<list>" ,None ,Bin ,fnopNone)

case knopList:

return WalkList(NULL, pnode, context);

//PTNODE(knopVarDecl , "varDcl" ,None ,Var ,fnopNone)

case knopVarDecl:

case knopConstDecl:

case knopLetDecl:

case knopTemp:

return WalkVar(pnode, context);

//PTNODE(knopFncDecl , "fncDcl" ,None ,Fnc ,fnopLeaf)

case knopFncDecl:

return WalkFnc(pnode, context);

//PTNODE(knopProg , "program" ,None ,Fnc ,fnopNone)

case knopProg:

return WalkProg(pnode, context);

//PTNODE(knopFor , "for" ,None ,For ,fnopBreak|fnopContinue)

case knopFor:

return WalkFor(pnode, context);

//PTNODE(knopIf , "if" ,None ,If ,fnopNone)

case knopIf:

return WalkIf(pnode, context);

//PTNODE(knopWhile , "while" ,None ,While,fnopBreak|fnopContinue)

case knopWhile:

return WalkWhile(pnode, context);

//PTNODE(knopDoWhile , "do-while" ,None ,While,fnopBreak|fnopContinue)

case knopDoWhile:

return WalkDoWhile(pnode, context);

//PTNODE(knopForIn , "for in" ,None ,ForIn,fnopBreak|fnopContinue|fnopCleanup)

case knopForIn:

return WalkForInOrForOf(pnode, context);

case knopForOf:

return WalkForInOrForOf(pnode, context);

//PTNODE(knopReturn , "return" ,None ,Uni ,fnopNone)

case knopReturn:

return WalkReturn(pnode, context);

//PTNODE(knopBlock , "{}" ,None ,Block,fnopNone)

case knopBlock:

return WalkBlock(pnode, context);

//PTNODE(knopWith , "with" ,None ,With ,fnopCleanup)

case knopWith:

return WalkWith(pnode, context);

//PTNODE(knopSwitch , "switch" ,None ,Switch,fnopBreak)

case knopSwitch:

return WalkSwitch(pnode, context);

//PTNODE(knopCase , "case" ,None ,Case ,fnopNone)

case knopCase:

return WalkCase(pnode, context);

//PTNODE(knopTryFinally,"try-finally",None,TryFinally,fnopCleanup)

case knopTryFinally:

return WalkTryFinally(pnode, context);

case knopFinally:

return WalkFinally(pnode, context);

//PTNODE(knopCatch , "catch" ,None ,Catch,fnopNone)

case knopCatch:

return WalkCatch(pnode, context);

//PTNODE(knopTryCatch , "try-catch" ,None ,TryCatch ,fnopCleanup)

case knopTryCatch:

return WalkTryCatch(pnode, context);

//PTNODE(knopTry , "try" ,None ,Try ,fnopCleanup)

case knopTry:

return WalkTry(pnode, context);

//PTNODE(knopThrow , "throw" ,None ,Uni ,fnopNone)

case knopThrow:

return WalkPostUnary(pnode, context);

case knopStrTemplate:

return WalkStringTemplate(pnode, context);

//PTNODE(knopClassDecl , "classDecl" ,None ,Class ,fnopLeaf)

case knopClassDecl:

return WalkClass(pnode, context);

default:

{

uint fnop = ParseNode::Grfnop(pnode->nop);

if (fnop & fnopLeaf || fnop && fnopNone)

{

return WalkLeaf(pnode, context);

}

else if (fnop & fnopBin)

{

return WalkBinary(pnode, context);

}

else if (fnop & fnopUni)

{

// Prefix unary operators.

return WalkPreUnary(pnode, context);

}

// Some node types are both fnopNotExprStmt and something else. Try the above cases first and fall back to this one.

if (fnop & fnopNotExprStmt)

{

return WalkLeaf(pnode, context);

}

Assert(false);

\_\_assume(false);

}

}

}

};