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

// 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

#include <crtdefs.h>

#include "ParseFlags.h"

// Operator precedence levels

enum

{

koplNo, // not an operator

koplCma, // ,

koplSpr, // ...

koplAsg, // = += etc

koplQue, // ?:

koplLor, // ||

koplLan, // &&

koplBor, // |

koplXor, // ^

koplBan, // &

koplEqu, // == !=

koplCmp, // < <= > >=

koplShf, // << >> >>>

koplAdd, // + -

koplExpo, // \*\*

koplMul, // \* / %

koplUni, // unary operators

koplLim

};

enum ParseType

{

ParseType\_Upfront,

ParseType\_Deferred,

ParseType\_Reparse

};

enum DestructuringInitializerContext

{

DIC\_None,

DIC\_ShouldNotParseInitializer, // e.g. We don't want to parse the initializer even though we found assignment

DIC\_ForceErrorOnInitializer, // e.g. Catch param where we explicitly want to raise an error when the initializer found

};

enum ScopeType;

enum SymbolType : byte;

// Representation of a label used when no AST is being built.

struct LabelId

{

IdentPtr pid;

struct LabelId\* next;

};

typedef ArenaAllocator ParseNodeAllocator;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Parser object.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

class CompileScriptException;

class Parser;

class SourceContextInfo;

struct BlockIdsStack;

class Span;

class BackgroundParser;

struct BackgroundParseItem;

struct PnClass;

class HashTbl;

typedef void (\*ParseErrorCallback)(void \*data, charcount\_t position, charcount\_t length, HRESULT hr);

struct PidRefStack;

struct CatchPidRef

{

IdentPtr pid;

PidRefStack \*ref;

};

typedef SListBase<CatchPidRef> CatchPidRefList;

struct DeferredFunctionStub;

DeferredFunctionStub \* BuildDeferredStubTree(ParseNode \*pnodeFnc, Recycler \*recycler);

struct StmtNest;

struct BlockInfoStack;

struct ParseContext

{

LPCUTF8 pszSrc;

size\_t offset;

size\_t length;

charcount\_t characterOffset;

int nextBlockId;

ULONG grfscr;

ULONG lineNumber;

ParseNodePtr pnodeProg;

SourceContextInfo\* sourceContextInfo;

BlockInfoStack\* currentBlockInfo;

bool strictMode;

bool fromExternal;

};

template <bool nullTerminated> class UTF8EncodingPolicyBase;

typedef UTF8EncodingPolicyBase<false> NotNullTerminatedUTF8EncodingPolicy;

template <typename T> class Scanner;

namespace Js

{

class ParseableFunctionInfo;

class FunctionBody;

};

class Parser

{

typedef Scanner<NotNullTerminatedUTF8EncodingPolicy> Scanner\_t;

private:

template <OpCode nop> static int GetNodeSize();

#define PTNODE(nop,sn,pc,nk,ok,json) template <> static int GetNodeSize<nop>() { return kcbPn##nk; };

#include "ptlist.h"

template <OpCode nop> static ParseNodePtr StaticAllocNode(ArenaAllocator \* alloc)

{

ParseNodePtr pnode = (ParseNodePtr)alloc->Alloc(GetNodeSize<nop>());

Assert(pnode != nullptr);

return pnode;

}

public:

#if DEBUG

Parser(Js::ScriptContext\* scriptContext, BOOL strictMode = FALSE, PageAllocator \*alloc = nullptr, bool isBackground = false, size\_t size = sizeof(Parser));

#else

Parser(Js::ScriptContext\* scriptContext, BOOL strictMode = FALSE, PageAllocator \*alloc = nullptr, bool isBackground = false);

#endif

~Parser(void);

Js::ScriptContext\* GetScriptContext() const { return m\_scriptContext; }

void ClearScriptContext() { m\_scriptContext = nullptr; }

bool IsBackgroundParser() const { return m\_isInBackground; }

bool IsDoingFastScan() const { return m\_doingFastScan; }

static IdentPtr PidFromNode(ParseNodePtr pnode);

ParseNode\* CopyPnode(ParseNode\* pnode);

IdentPtr GenerateIdentPtr(\_\_ecount(len) wchar\_t\* name,long len);

ArenaAllocator \*GetAllocator() { return &m\_nodeAllocator;}

size\_t GetSourceLength() { return m\_length; }

size\_t GetOriginalSourceLength() { return m\_originalLength; }

static ULONG GetDeferralThreshold(bool isProfileLoaded);

BOOL DeferredParse(Js::LocalFunctionId functionId);

BOOL IsDeferredFnc();

void ReduceDeferredScriptLength(size\_t chars);

void RestorePidRefForSym(Symbol \*sym);

HRESULT ValidateSyntax(LPCUTF8 pszSrc, size\_t encodedCharCount, bool isGenerator, bool isAsync, CompileScriptException \*pse, void (Parser::\*validateFunction)());

// Should be called when the UTF-8 source was produced from UTF-16. This is really CESU-8 source in that it encodes surragate pairs

// as 2 three byte sequences instead of 4 bytes as required UTF-8. It also is is loss-less converison of invalid UTF-16 sequences.

// This is important in Javascript because Javascript engines are required not report invalid UTF-16 sequences and to consider

// the UTF-16 characters pre-canonacalized. Converting this UTF-16 with invalid sequences to valid UTF-8 and back would cause

// all invalid UTF-16 seqences to be replace by one or more Unicode replacement characters (0xFFFD), losing the original

// invalid sequences.

HRESULT ParseCesu8Source(\_\_out ParseNodePtr\* parseTree, LPCUTF8 pSrc, size\_t length, ULONG grfsrc, CompileScriptException \*pse,

Js::LocalFunctionId \* nextFunctionId, SourceContextInfo \* sourceContextInfo);

// Should be called when the source is UTF-8 and invalid UTF-8 sequences should be replaced with the unicode replacement character

// (0xFFFD). Security concerns require externally produced UTF-8 only allow valid UTF-8 otherwise an attacker could use invalid

// UTF-8 sequences to fool a filter and cause Javascript to be executed that might otherwise have been rejected.

HRESULT ParseUtf8Source(\_\_out ParseNodePtr\* parseTree, LPCUTF8 pSrc, size\_t length, ULONG grfsrc, CompileScriptException \*pse,

Js::LocalFunctionId \* nextFunctionId, SourceContextInfo \* sourceContextInfo);

// Used by deferred parsing to parse a deferred function.

HRESULT ParseSourceWithOffset(\_\_out ParseNodePtr\* parseTree, LPCUTF8 pSrc, size\_t offset, size\_t cbLength, charcount\_t cchOffset,

bool isCesu8, ULONG grfscr, CompileScriptException \*pse, Js::LocalFunctionId \* nextFunctionId, ULONG lineNumber,

SourceContextInfo \* sourceContextInfo, Js::ParseableFunctionInfo\* functionInfo, bool isReparse);

protected:

HRESULT ParseSourceInternal(

\_\_out ParseNodePtr\* parseTree, LPCUTF8 pszSrc, size\_t offsetInBytes,

size\_t lengthInCodePoints, charcount\_t offsetInChars, bool fromExternal,

ULONG grfscr, CompileScriptException \*pse, Js::LocalFunctionId \* nextFunctionId, ULONG lineNumber, SourceContextInfo \* sourceContextInfo);

ParseNodePtr Parse(LPCUTF8 pszSrc, size\_t offset, size\_t length, charcount\_t charOffset, ULONG grfscr, ULONG lineNumber,

Js::LocalFunctionId \* nextFunctionId, CompileScriptException \*pse);

private:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Core members.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

ParseNodeAllocator m\_nodeAllocator;

long m\_cactIdentToNodeLookup;

ulong m\_grfscr;

size\_t m\_length; // source length in characters excluding comments and literals

size\_t m\_originalLength; // source length in characters excluding comments and literals

Js::LocalFunctionId \* m\_nextFunctionId;

SourceContextInfo\* m\_sourceContextInfo;

CatchPidRefList \*m\_catchPidRefList;

ParseErrorCallback m\_errorCallback;

void \* m\_errorCallbackData;

BOOL m\_uncertainStructure;

bool m\_hasParallelJob;

bool m\_doingFastScan;

Span m\_asgToConst;

int m\_nextBlockId;

// RegexPattern objects created for literal regexes are recycler-allocated and need to be kept alive until the function body

// is created during byte code generation. The RegexPattern pointer is stored in the script context's guest

// arena for that purpose. This list is then unregistered from the guest arena at the end of parsing/scanning.

SList<UnifiedRegex::RegexPattern \*, ArenaAllocator> m\_registeredRegexPatterns;

protected:

Js::ScriptContext\* m\_scriptContext;

HashTbl \* m\_phtbl;

ErrHandler m\_err;

static const uint HASH\_TABLE\_SIZE = 256;

\_\_declspec(noreturn) void Error(HRESULT hr);

private:

\_\_declspec(noreturn) void Error(HRESULT hr, ParseNodePtr pnode);

\_\_declspec(noreturn) void Error(HRESULT hr, charcount\_t ichMin, charcount\_t ichLim);

\_\_declspec(noreturn) static void OutOfMemory();

void GenerateCode(ParseNodePtr pnode, void \*pvUser, long cbUser,

LPCOLESTR pszSrc, long cchSrc, LPCOLESTR pszTitle);

void EnsureStackAvailable();

void IdentifierExpectedError(const Token& token);

bool CheckForDirective(bool\* pIsUseStrict, bool\* pIsUseAsm, bool\* pIsOctalInString);

bool CheckStrictModeStrPid(IdentPtr pid);

bool CheckAsmjsModeStrPid(IdentPtr pid);

void InitPids();

CatchPidRefList \*GetCatchPidRefList() const { return m\_catchPidRefList; }

void SetCatchPidRefList(CatchPidRefList \*list) { m\_catchPidRefList = list; }

CatchPidRefList \*EnsureCatchPidRefList();

// True if we need to create PID's and bind names to decls in deferred functions.

// Do this if we need to support early let/const errors.

bool BindDeferredPidRefs() const;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Members needed just for parsing.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

protected:

Token m\_token;

Scanner\_t\* m\_pscan;

public:

// create nodes using arena allocator; used by AST transformation

template <OpCode nop>

static ParseNodePtr StaticCreateNodeT(ArenaAllocator\* alloc, charcount\_t ichMin = 0, charcount\_t ichLim = 0);

static ParseNodePtr StaticCreateBinNode(OpCode nop, ParseNodePtr pnode1,ParseNodePtr pnode2,ArenaAllocator\* alloc);

static ParseNodePtr StaticCreateBlockNode(ArenaAllocator\* alloc, charcount\_t ichMin = 0, charcount\_t ichLim = 0, int blockId = -1, PnodeBlockType blockType = PnodeBlockType::Regular);

ParseNodePtr CreateNode(OpCode nop, charcount\_t ichMin,charcount\_t ichLim);

ParseNodePtr CreateDummyFuncNode(bool fDeclaration);

ParseNodePtr CreateTriNode(OpCode nop, ParseNodePtr pnode1,

ParseNodePtr pnode2, ParseNodePtr pnode3,

charcount\_t ichMin,charcount\_t ichLim);

ParseNodePtr CreateTempNode(ParseNode\* initExpr);

ParseNodePtr CreateTempRef(ParseNode\* tempNode);

ParseNodePtr CreateNode(OpCode nop) { return CreateNode(nop, m\_pscan? m\_pscan->IchMinTok() : 0); }

ParseNodePtr CreateDeclNode(OpCode nop, IdentPtr pid, SymbolType symbolType, bool errorOnRedecl = true);

Symbol\* AddDeclForPid(ParseNodePtr pnode, IdentPtr pid, SymbolType symbolType, bool errorOnRedecl);

ParseNodePtr CreateNameNode(IdentPtr pid)

{

ParseNodePtr pnode = CreateNode(knopName);

pnode->sxPid.pid = pid;

pnode->sxPid.sym=NULL;

pnode->sxPid.symRef=NULL;

return pnode;

}

ParseNodePtr CreateBlockNode(PnodeBlockType blockType = PnodeBlockType::Regular)

{

ParseNodePtr pnode = CreateNode(knopBlock);

InitBlockNode(pnode, m\_nextBlockId++, blockType);

return pnode;

}

// Creating parse nodes.

ParseNodePtr CreateNode(OpCode nop, charcount\_t ichMin);

ParseNodePtr CreateTriNode(OpCode nop, ParseNodePtr pnode1, ParseNodePtr pnode2, ParseNodePtr pnode3);

ParseNodePtr CreateIntNode(long lw);

ParseNodePtr CreateStrNode(IdentPtr pid);

ParseNodePtr CreateUniNode(OpCode nop, ParseNodePtr pnodeOp);

ParseNodePtr CreateBinNode(OpCode nop, ParseNodePtr pnode1, ParseNodePtr pnode2);

ParseNodePtr CreateCallNode(OpCode nop, ParseNodePtr pnode1, ParseNodePtr pnode2);

// Create parse node with token limis

template <OpCode nop>

ParseNodePtr CreateNodeT(charcount\_t ichMin,charcount\_t ichLim);

ParseNodePtr CreateUniNode(OpCode nop, ParseNodePtr pnode1, charcount\_t ichMin,charcount\_t ichLim);

ParseNodePtr CreateBlockNode(charcount\_t ichMin,charcount\_t ichLim, PnodeBlockType blockType = PnodeBlockType::Regular);

ParseNodePtr CreateNameNode(IdentPtr pid,charcount\_t ichMin,charcount\_t ichLim);

ParseNodePtr CreateBinNode(OpCode nop, ParseNodePtr pnode1, ParseNodePtr pnode2,

charcount\_t ichMin,charcount\_t ichLim);

ParseNodePtr CreateCallNode(OpCode nop, ParseNodePtr pnode1, ParseNodePtr pnode2,

charcount\_t ichMin,charcount\_t ichLim);

void PrepareScanner(bool fromExternal);

void PrepareForBackgroundParse();

void AddFastScannedRegExpNode(ParseNodePtr const pnode);

void AddBackgroundRegExpNode(ParseNodePtr const pnode);

void AddBackgroundParseItem(BackgroundParseItem \*const item);

void FinishBackgroundRegExpNodes();

void FinishBackgroundPidRefs(BackgroundParseItem \*const item, bool isOtherParser);

void WaitForBackgroundJobs(BackgroundParser \*bgp, CompileScriptException \*pse);

HRESULT ParseFunctionInBackground(ParseNodePtr pnodeFunc, ParseContext \*parseContext, bool topLevelDeferred, CompileScriptException \*pse);

void CheckPidIsValid(IdentPtr pid, bool autoArgumentsObject = false);

void AddVarDeclToBlock(ParseNode \*pnode);

// Add a var declaration. Only use while parsing. Assumes m\_ppnodeVar is pointing to the right place already

ParseNodePtr CreateVarDeclNode(IdentPtr pid, SymbolType symbolType, bool autoArgumentsObject = false, ParseNodePtr pnodeFnc = NULL, bool checkReDecl = true);

// Add a var declaration, during parse tree rewriting. Will setup m\_ppnodeVar for the given pnodeFnc

ParseNodePtr AddVarDeclNode(IdentPtr pid, ParseNodePtr pnodeFnc);

// Add a 'const' or 'let' declaration.

ParseNodePtr CreateBlockScopedDeclNode(IdentPtr pid, OpCode nodeType);

void RegisterRegexPattern(UnifiedRegex::RegexPattern \*const regexPattern);

bool IsReparsing() const { return m\_parseType == ParseType\_Reparse; }

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

WCHAR\* GetParseType() const

{

switch(m\_parseType)

{

case ParseType\_Upfront:

return L"Upfront";

case ParseType\_Deferred:

return L"Deferred";

case ParseType\_Reparse:

return L"Reparse";

}

Assert(false);

return NULL;

}

#endif

void CaptureContext(ParseContext \*parseContext) const;

void RestoreContext(ParseContext \*const parseContext);

int GetLastBlockId() const { Assert(m\_nextBlockId > 0); return m\_nextBlockId - 1; }

private:

template <OpCode nop> ParseNodePtr CreateNodeWithScanner();

template <OpCode nop> ParseNodePtr CreateNodeWithScanner(charcount\_t ichMin);

ParseNodePtr CreateStrNodeWithScanner(IdentPtr pid);

ParseNodePtr CreateIntNodeWithScanner(long lw);

static void InitNode(OpCode nop,ParseNodePtr pnode);

static void InitBlockNode(ParseNodePtr pnode, int blockId, PnodeBlockType blockType);

private:

ParseNodePtr m\_currentNodeNonLambdaFunc; // current function or NULL

ParseNodePtr m\_currentNodeNonLambdaDeferredFunc; // current function or NULL

ParseNodePtr m\_currentNodeFunc; // current function or NULL

ParseNodePtr m\_currentNodeDeferredFunc; // current function or NULL

ParseNodePtr m\_currentNodeProg; // current programm

DeferredFunctionStub \*m\_currDeferredStub;

long \* m\_pCurrentAstSize;

ParseNodePtr \* m\_ppnodeScope; // function list tail

ParseNodePtr \* m\_ppnodeExprScope; // function expression list tail

ParseNodePtr \* m\_ppnodeVar; // variable list tail

bool m\_inDeferredNestedFunc; // true if parsing a function in deferred mode, nested within the current node

bool m\_isInBackground;

uint \* m\_pnestedCount; // count of functions nested at one level below the current node

struct WellKnownPropertyPids

{

IdentPtr arguments; // m\_pidArguments; // 'arguments' identifier

IdentPtr async; // m\_pidAsync;

IdentPtr eval; // m\_pidEval;

IdentPtr setter; // m\_pidSetter;

IdentPtr getter; // m\_pidGetter;

IdentPtr let; //m\_pidLet;

IdentPtr constructor; //m\_pidConstructor;

IdentPtr prototype; //m\_pidPrototype;

IdentPtr \_\_proto\_\_; // m\_pid\_\_proto\_\_;

IdentPtr of; //m\_pidOf;

IdentPtr target; // m\_pidTarget;

};

WellKnownPropertyPids wellKnownPropertyPids;

charcount\_t m\_sourceLim; // The actual number of characters parsed.

Js::ParseableFunctionInfo\* m\_functionBody; // For a deferred parsed function, the function body is non-null

ParseType m\_parseType;

uint m\_parsingDuplicate;

uint m\_arrayDepth;

uint m\_funcInArrayDepth; // Count func depth within array literal

charcount\_t m\_funcInArray;

uint m\_scopeCountNoAst;

/\*

\* Parsing states for super restriction

\*/

static const uint ParsingSuperRestrictionState\_SuperDisallowed = 0;

static const uint ParsingSuperRestrictionState\_SuperCallAndPropertyAllowed = 1;

static const uint ParsingSuperRestrictionState\_SuperPropertyAllowed = 2;

uint m\_parsingSuperRestrictionState;

friend class AutoParsingSuperRestrictionStateRestorer;

// Used for issuing spread and rest errors when there is ambiguity with parameter list and parenthesized expressions

uint m\_parenDepth;

bool m\_deferEllipsisError;

RestorePoint m\_EllipsisErrLoc;

uint m\_tryCatchOrFinallyDepth; // Used to determine if parsing is currently in a try/catch/finally block in order to throw error on yield expressions inside them

StmtNest \*m\_pstmtCur; // current statement or NULL

BlockInfoStack \*m\_currentBlockInfo;

Scope \*m\_currentScope;

BackgroundParseItem \*currBackgroundParseItem;

BackgroundParseItem \*backgroundParseItems;

typedef DList<ParseNodePtr, ArenaAllocator> NodeDList;

NodeDList\* fastScannedRegExpNodes;

BlockIdsStack \*m\_currentDynamicBlock;

int GetCurrentDynamicBlockId() const;

void AppendFunctionToScopeList(bool fDeclaration, ParseNodePtr pnodeFnc);

// block scoped content helpers

void SetCurrentStatement(StmtNest \*stmt);

ParseNode\* GetCurrentBlock();

ParseNode\* GetFunctionBlock();

BlockInfoStack\* GetCurrentBlockInfo();

BlockInfoStack\* GetCurrentFunctionBlockInfo();

ParseNode \*GetCurrentFunctionNode();

ParseNode \*GetCurrentNonLamdaFunctionNode();

bool IsNodeAllowedForDeferParse(OpCode op) {return !this->m\_deferringAST ||

(op == knopBlock || op == knopVarDecl || op == knopConstDecl || op == knopLetDecl || op == knopFncDecl); }

bool NextTokenConfirmsLetDecl() const { return m\_token.tk == tkID || m\_token.tk == tkLBrack || m\_token.tk == tkLCurly || m\_token.IsReservedWord(); }

template<bool buildAST>

void PushStmt(StmtNest \*pStmt, ParseNodePtr pnode, OpCode op, ParseNodePtr pnodeLab, LabelId\* pLabelIdList)

{

AssertMem(pStmt);

if (buildAST)

{

AssertNodeMem(pnode);

AssertNodeMemN(pnodeLab);

pnode->sxStmt.grfnop = 0;

pnode->sxStmt.pnodeOuter = (NULL == m\_pstmtCur) ? NULL : m\_pstmtCur->pnodeStmt;

pStmt->pnodeStmt = pnode;

pStmt->pnodeLab = pnodeLab;

}

else

{

// Assign to pnodeStmt rather than op so that we initialize the whole field.

pStmt->pnodeStmt = 0;

pStmt->isDeferred = true;

pStmt->op = op;

pStmt->pLabelId = pLabelIdList;

}

pStmt->pstmtOuter = m\_pstmtCur;

SetCurrentStatement(pStmt);

}

void PopStmt(StmtNest \*pStmt);

BlockInfoStack \*PushBlockInfo(ParseNodePtr pnodeBlock);

void PopBlockInfo();

void PushDynamicBlock();

void PopDynamicBlock();

ParseNodePtr PnodeLabel(IdentPtr pid, ParseNodePtr pnodeLabels);

void MarkEvalCaller()

{

if (m\_currentNodeFunc)

{

ParseNodePtr pnodeFunc = GetCurrentFunctionNode();

pnodeFunc->sxFnc.SetCallsEval(true);

}

ParseNode \*pnodeBlock = GetCurrentBlock();

if (pnodeBlock != NULL)

{

pnodeBlock->sxBlock.SetCallsEval(true);

PushDynamicBlock();

}

}

public:

WellKnownPropertyPids\* names(){ return &wellKnownPropertyPids; }

IdentPtr CreatePid(\_\_in\_ecount(len) LPCOLESTR name, charcount\_t len)

{

return m\_phtbl->PidHashNameLen(name, len);

}

bool KnownIdent(\_\_in\_ecount(len) LPCOLESTR name, charcount\_t len)

{

return m\_phtbl->Contains(name, len);

}

template <typename THandler>

static void ForEachItemRefInList(ParseNodePtr \*list, THandler handler)

{

ParseNodePtr \*current = list;

while (current != nullptr && (\*current) != nullptr)

{

if ((\*current)->nop == knopList)

{

handler(&(\*current)->sxBin.pnode1);

// Advance to the next node

current = &(\*current)->sxBin.pnode2;

}

else

{

// The last node

handler(current);

current = nullptr;

}

}

}

template <typename THandler>

static void ForEachItemInList(ParseNodePtr list, THandler handler)

{

ForEachItemRefInList(&list, [&](ParseNodePtr \* item) {

Assert(item != nullptr);

handler(\*item);

});

}

template <class THandler>

static void MapBindIdentifierFromElement(ParseNodePtr elementNode, THandler handler)

{

ParseNodePtr bindIdentNode = elementNode;

if (bindIdentNode->nop == knopAsg)

{

bindIdentNode = bindIdentNode->sxBin.pnode1;

}

else if (bindIdentNode->nop == knopEllipsis)

{

bindIdentNode = bindIdentNode->sxUni.pnode1;

}

if (bindIdentNode->IsPattern())

{

MapBindIdentifier(bindIdentNode, handler);

}

else if (bindIdentNode->IsVarLetOrConst())

{

handler(bindIdentNode);

}

else

{

AssertMsg(bindIdentNode->nop == knopEmpty, "Invalid bind identifier");

}

}

template <class THandler>

static void MapBindIdentifier(ParseNodePtr patternNode, THandler handler)

{

if (patternNode->nop == knopAsg)

{

patternNode = patternNode->sxBin.pnode1;

}

Assert(patternNode->IsPattern());

if (patternNode->nop == knopArrayPattern)

{

ForEachItemInList(patternNode->sxArrLit.pnode1, [&](ParseNodePtr item) {

MapBindIdentifierFromElement(item, handler);

});

}

else

{

ForEachItemInList(patternNode->sxUni.pnode1, [&](ParseNodePtr item) {

Assert(item->nop == knopObjectPatternMember);

MapBindIdentifierFromElement(item->sxBin.pnode2, handler);

});

}

}

private:

struct IdentToken

{

tokens tk;

IdentPtr pid;

charcount\_t ichMin;

charcount\_t ichLim;

IdentToken()

: tk(tkNone), pid(NULL)

{

}

};

void CheckArguments(ParseNodePtr pnode);

void CheckArgumentsUse(IdentPtr pid, ParseNodePtr pnodeFnc);

void CheckStrictModeEvalArgumentsUsage(IdentPtr pid, ParseNodePtr pnode = NULL);

void CheckStrictModeFncDeclNotSourceElement(const bool isSourceElement, const BOOL isDeclaration);

// environments on which the strict mode is set, if found

enum StrictModeEnvironment

{

SM\_NotUsed, // StrictMode environment is don't care

SM\_OnGlobalCode, // The current environment is a global code

SM\_OnFunctionCode, // The current environment is a function code

SM\_DeferedParse // StrictMode used in defered parse cases

};

template<bool buildAST> ParseNodePtr ParseArrayLiteral();

template<bool buildAST> ParseNodePtr ParseStatement(bool isSourceElement = false);

template<bool buildAST> ParseNodePtr ParseVariableDeclaration(

tokens declarationType,

charcount\_t ichMin,

BOOL fAllowIn = TRUE,

BOOL\* pfForInOk = nullptr,

BOOL singleDefOnly = FALSE,

BOOL allowInit = TRUE,

BOOL isTopVarParse = TRUE,

BOOL isFor = FALSE);

BOOL TokIsForInOrForOf();

template<bool buildAST>

void ParseStmtList(

ParseNodePtr \*ppnodeList,

ParseNodePtr \*\*pppnodeLast = NULL,

StrictModeEnvironment smEnvironment = SM\_NotUsed,

const bool isSourceElementList = false,

bool\* strictModeOn = NULL);

bool FastScanFormalsAndBody();

bool ScanAheadToFunctionEnd(uint count);

bool DoParallelParse(ParseNodePtr pnodeFnc) const;

// TODO: We should really call this StartScope and separate out the notion of scopes and blocks;

// blocks refer to actual curly braced syntax, whereas scopes contain symbols. All blocks have

// a scope, but some statements like for loops or the with statement introduce a block-less scope.

template<bool buildAST> ParseNodePtr StartParseBlock(PnodeBlockType blockType, ScopeType scopeType, ParseNodePtr pnodeLabel = NULL, LabelId\* pLabelId = NULL);

template<bool buildAST> ParseNodePtr StartParseBlockWithCapacity(PnodeBlockType blockType, ScopeType scopeType, int capacity);

template<bool buildAST> ParseNodePtr StartParseBlockHelper(PnodeBlockType blockType, Scope \*scope, ParseNodePtr pnodeLabel, LabelId\* pLabelId);

void PushFuncBlockScope(ParseNodePtr pnodeBlock, ParseNodePtr \*\*ppnodeScopeSave, ParseNodePtr \*\*ppnodeExprScopeSave);

void PopFuncBlockScope(ParseNodePtr \*ppnodeScopeSave, ParseNodePtr \*ppnodeExprScopeSave);

template<bool buildAST> ParseNodePtr ParseBlock(ParseNodePtr pnodeLabel, LabelId\* pLabelId);

void FinishParseBlock(ParseNode \*pnodeBlock, bool needScanRCurly = true);

void FinishParseFncExprScope(ParseNodePtr pnodeFnc, ParseNodePtr pnodeFncExprScope);

template<const bool backgroundPidRefs>

void BindPidRefs(BlockInfoStack \*blockInfo, uint maxBlockId = (uint)-1);

void BindPidRefsInScope(IdentPtr pid, Symbol \*sym, int blockId, uint maxBlockId = (uint)-1);

void BindConstPidRefsInScope(IdentPtr pid, Symbol \*sym, int blockId, uint maxBlockId = (uint)-1);

template<const bool constBinding>

void BindPidRefsInScopeImpl(IdentPtr pid, Symbol \*sym, int blockId, uint maxBlockId = (uint)-1);

void PushScope(Scope \*scope);

void PopScope(Scope \*scope);

template<bool buildAST> ParseNodePtr ParseArgList(bool \*pCallOfConstants, uint16 \*pSpreadArgCount, uint16 \* pCount);

template<bool buildAST> ParseNodePtr ParseArrayList(bool \*pArrayOfTaggedInts, bool \*pArrayOfInts, bool \*pArrayOfNumbers, bool \*pHasMissingValues, uint \*count, uint \*spreadCount);

template<bool buildAST> ParseNodePtr ParseMemberList(LPCOLESTR pNameHint, ulong \*pHintLength, tokens declarationType = tkNone);

template<bool buildAST> ParseNodePtr ParseSuper(ParseNodePtr pnode, bool fAllowCall);

// Used to determine the type of JavaScript object member.

// The values can be combined using bitwise OR.

// specifically, it is valid to have getter and setter at the same time.

enum MemberType

{

MemberTypeDataProperty = 1 << 0, // { foo: 1 },

MemberTypeGetter = 1 << 1, // { get foo() }

MemberTypeSetter = 1 << 2, // { set foo(arg) {} }

MemberTypeMethod = 1 << 3, // { foo() {} }

MemberTypeIdentifier = 1 << 4 // { foo } (shorthand for { foo: foo })

};

// Used to map JavaScript object member name to member type.

typedef JsUtil::BaseDictionary<WCHAR\*, MemberType, ArenaAllocator, PrimeSizePolicy> MemberNameToTypeMap;

static MemberNameToTypeMap\* CreateMemberNameMap(ArenaAllocator\* pAllocator);

template<bool buildAST> void ParseComputedName(ParseNodePtr\* ppnodeName, LPCOLESTR\* ppNameHint, LPCOLESTR\* ppFullNameHint = nullptr, ulong \*pNameLength = nullptr, ulong \*pShortNameOffset = nullptr);

template<bool buildAST> ParseNodePtr ParseMemberGetSet(OpCode nop, LPCOLESTR\* ppNameHint);

template<bool buildAST> ParseNodePtr ParseFncDecl(ushort flags, LPCOLESTR pNameHint = NULL, const bool isSourceElement = false, const bool needsPIDOnRCurlyScan = false, bool resetParsingSuperRestrictionState = true, bool fUnaryOrParen = false);

template<bool buildAST> bool ParseFncNames(ParseNodePtr pnodeFnc, ParseNodePtr pnodeFncParent, ushort flags, ParseNodePtr \*\*pLastNodeRef);

template<bool buildAST> void ParseFncFormals(ParseNodePtr pnodeFnc, ushort flags);

template<bool buildAST> bool ParseFncDeclHelper(ParseNodePtr pnodeFnc, ParseNodePtr pnodeFncParent, LPCOLESTR pNameHint, ushort flags, bool \*pHasName, bool fUnaryOrParen, bool noStmtContext, bool \*pNeedScanRCurly);

template<bool buildAST> void ParseExpressionLambdaBody(ParseNodePtr pnodeFnc);

bool FncDeclAllowedWithoutContext(ushort flags);

void FinishFncDecl(ParseNodePtr pnodeFnc, LPCOLESTR pNameHint, ParseNodePtr \*lastNodeRef);

void ParseTopLevelDeferredFunc(ParseNodePtr pnodeFnc, ParseNodePtr pnodeFncParent, LPCOLESTR pNameHint);

void ParseNestedDeferredFunc(ParseNodePtr pnodeFnc, bool fLambda, bool \*pNeedScanRCurly, bool \*pStrictModeTurnedOn);

void CheckStrictFormalParameters();

void AddArgumentsNodeToVars(ParseNodePtr pnodeFnc);

void UpdateOrCheckForDuplicateInFormals(IdentPtr pid, SList<IdentPtr> \*formals);

void TransformAsyncFncDeclAST(ParseNodePtr \*pnodeBody, bool fLambda);

ParseNodePtr CreateAsyncSpawnGenerator();

LPCOLESTR GetFunctionName(ParseNodePtr pnodeFnc, LPCOLESTR pNameHint);

uint CalculateFunctionColumnNumber();

template<bool buildAST> ParseNodePtr GenerateEmptyConstructor(bool extends = false);

IdentPtr ParseClassPropertyName(IdentPtr \* hint);

template<bool buildAST> ParseNodePtr ParseClassDecl(BOOL isDeclaration, LPCOLESTR pNameHint, ulong \*pHintLength, ulong \*pShortNameOffset);

template<bool buildAST> ParseNodePtr ParseStringTemplateDecl(ParseNodePtr pnodeTagFnc);

// This is used in the es6 class pattern.

LPCOLESTR ConstructFinalHintNode(IdentPtr pClassName, IdentPtr pMemberName, IdentPtr pGetSet, bool isStatic, ulong\* nameLength, ulong\* pShortNameOffset, bool isComputedName = false, LPCOLESTR pMemberNameHint = nullptr);

// Construct the name from the parse node.

LPCOLESTR FormatPropertyString(LPCOLESTR propertyString, ParseNodePtr pNode, ulong \*fullNameHintLength, ulong \*pShortNameOffset);

LPCOLESTR ConstructNameHint(ParseNodePtr pNode, ulong\* fullNameHintLength, ulong \*pShortNameOffset);

LPCOLESTR AppendNameHints(IdentPtr left, IdentPtr right, ulong \*pNameLength, ulong \*pShortNameOffset, bool ignoreAddDotWithSpace = false, bool wrapInBrackets = false);

LPCOLESTR AppendNameHints(IdentPtr left, LPCOLESTR right, ulong \*pNameLength, ulong \*pShortNameOffset, bool ignoreAddDotWithSpace = false, bool wrapInBrackets = false);

LPCOLESTR AppendNameHints(LPCOLESTR left, IdentPtr right, ulong \*pNameLength, ulong \*pShortNameOffset, bool ignoreAddDotWithSpace = false, bool wrapInBrackets = false);

LPCOLESTR AppendNameHints(LPCOLESTR left, LPCOLESTR right, ulong \*pNameLength, ulong \*pShortNameOffset, bool ignoreAddDotWithSpace = false, bool wrapInBrackets = false);

LPCOLESTR AppendNameHints(LPCOLESTR leftStr, ulong leftLen, LPCOLESTR rightStr, ulong rightLen, ulong \*pNameLength, ulong \*pShortNameOffset, bool ignoreAddDotWithSpace = false, bool wrapInBrackets = false);

WCHAR \* AllocateStringOfLength(ulong length);

void FinishFncNode(ParseNodePtr pnodeFnc);

template<bool buildAST> bool ParseOptionalExpr(

ParseNodePtr\* pnode,

bool fUnaryOrParen = false,

int oplMin = koplNo,

BOOL \*pfCanAssign = NULL,

BOOL fAllowIn = TRUE,

BOOL fAllowEllipsis = FALSE,

\_Inout\_opt\_ IdentToken\* pToken = NULL);

template<bool buildAST> ParseNodePtr ParseExpr(

int oplMin = koplNo,

BOOL \*pfCanAssign = NULL,

BOOL fAllowIn = TRUE,

BOOL fAllowEllipsis = FALSE,

LPCOLESTR pHint = NULL,

ulong \*pHintLength = nullptr,

ulong \*pShortNameOffset = nullptr,

\_Inout\_opt\_ IdentToken\* pToken = NULL,

bool fUnaryOrParen = false,

\_Inout\_opt\_ bool\* pfLikelyPattern = nullptr);

template<bool buildAST> ParseNodePtr ParseTerm(

BOOL fAllowCall = TRUE,

LPCOLESTR pNameHint = nullptr,

ulong \*pHintLength = nullptr,

ulong \*pShortNameOffset = nullptr,

\_Inout\_opt\_ IdentToken\* pToken = nullptr,

bool fUnaryOrParen = false,

\_Out\_opt\_ BOOL\* pfCanAssign = nullptr,

\_Inout\_opt\_ BOOL\* pfLikelyPattern = nullptr);

template<bool buildAST> ParseNodePtr ParsePostfixOperators(ParseNodePtr pnode,

BOOL fAllowCall, BOOL fInNew, BOOL \*pfCanAssign, \_Inout\_ IdentToken\* pToken);

void ThrowNewTargetSyntaxErrForGlobalScope();

template<bool buildAST> ParseNodePtr ParseMetaProperty(

tokens metaParentKeyword,

charcount\_t ichMin,

\_Out\_opt\_ BOOL\* pfCanAssign = nullptr);

BOOL NodeIsIdent(ParseNodePtr pnode, IdentPtr pid);

BOOL NodeIsEvalName(ParseNodePtr pnode);

BOOL IsJSONValid(ParseNodePtr pnodeExpr)

{

OpCode jnop = (knopNeg == pnodeExpr->nop) ? pnodeExpr->sxUni.pnode1->nop : pnodeExpr->nop;

if (knopNeg == pnodeExpr->nop)

{

return (knopInt == jnop || knopFlt == jnop);

}

else

{

return (knopInt == jnop || knopFlt == jnop ||

knopStr == jnop || knopNull == jnop ||

knopTrue == jnop || knopFalse == jnop ||

knopObject == jnop || knopArray == jnop);

}

}

BOOL IsConstantInFunctionCall(ParseNodePtr pnode);

BOOL IsConstantInArrayLiteral(ParseNodePtr pnode);

ParseNodePtr CreateParamPatternNode(ParseNodePtr pnode1);

ParseNodePtr ConvertMemberToMemberPattern(ParseNodePtr pnodeMember);

ParseNodePtr ConvertObjectToObjectPattern(ParseNodePtr pnodeMemberList);

ParseNodePtr GetRightSideNodeFromPattern(ParseNodePtr pnode);

ParseNodePtr ConvertArrayToArrayPattern(ParseNodePtr pnode);

ParseNodePtr ConvertToPattern(ParseNodePtr pnode);

void AppendToList(ParseNodePtr \* node, ParseNodePtr nodeToAppend);

bool IsES6DestructuringEnabled() const;

bool IsPossiblePatternStart() const { return m\_token.tk == tkLCurly || m\_token.tk == tkLBrack; }

bool IsPostFixOperators() const

{

return m\_token.tk == tkLParen ||

m\_token.tk == tkLBrack ||

m\_token.tk == tkDot ||

m\_token.tk == tkStrTmplBasic ||

m\_token.tk == tkStrTmplBegin;

}

template<bool buildAST> ParseNodePtr ParseTryCatchFinally();

template<bool buildAST> ParseNodePtr ParseTry();

template<bool buildAST> ParseNodePtr ParseCatch();

template<bool buildAST> ParseNodePtr ParseFinally();

template<bool buildAST> ParseNodePtr ParseCase(ParseNodePtr \*ppnodeBody);

template<bool buildAST> ParseNodePtr ParseRegExp();

template <bool buildAST>

ParseNodePtr ParseDestructuredArrayLiteral(tokens declarationType, bool isDecl, bool topLevel = true);

template <bool buildAST>

ParseNodePtr ParseDestructuredObjectLiteral(tokens declarationType, bool isDecl, bool topLevel = true);

template <bool buildAST>

ParseNodePtr ParseDestructuredLiteral(tokens declarationType,

bool isDecl,

bool topLevel = true,

DestructuringInitializerContext initializerContext = DIC\_None,

bool allowIn = true,

BOOL \*forInOfOkay = nullptr);

template <bool buildAST>

ParseNodePtr ParseDestructuredVarDecl(tokens declarationType, bool isDecl, bool \*hasSeenRest, bool topLevel = true);

template <bool buildAST>

ParseNodePtr ParseDestructuredInitializer(ParseNodePtr lhsNode,

bool isDecl,

bool topLevel,

DestructuringInitializerContext initializerContext,

bool allowIn,

BOOL \*forInOfOkay);

template<bool CheckForNegativeInfinity> static bool IsNaNOrInfinityLiteral(LPCOLESTR str);

void ParseDestructuredLiteralWithScopeSave(tokens declarationType,

bool isDecl,

bool topLevel,

DestructuringInitializerContext initializerContext = DIC\_None,

bool allowIn = true);

public:

void ValidateSourceElementList();

void ValidateFormals();

bool IsStrictMode() const;

BOOL ExpectingExternalSource();

IdentPtr GetArgumentsPid() const { return wellKnownPropertyPids.arguments; }

IdentPtr GetEvalPid() const { return wellKnownPropertyPids.eval; }

IdentPtr GetTargetPid() const { return wellKnownPropertyPids.target; }

BackgroundParseItem \*GetCurrBackgroundParseItem() const { return currBackgroundParseItem; }

void SetCurrBackgroundParseItem(BackgroundParseItem \*item) { currBackgroundParseItem = item; }

void Release()

{

RELEASEPTR(m\_pscan);

RELEASEPTR(m\_phtbl);

}

private:

void DeferOrEmitPotentialSpreadError(ParseNodePtr pnodeT);

template<bool buildAST> void TrackAssignment(ParseNodePtr pnodeT, IdentToken\* pToken, charcount\_t ichMin, charcount\_t ichLim);

PidRefStack\* PushPidRef(IdentPtr pid);

PidRefStack\* FindOrAddPidRef(IdentPtr pid, int blockId, int maxScopeId = -1);

void RemovePrevPidRef(IdentPtr pid, PidRefStack \*lastRef);

void SetPidRefsInScopeDynamic(IdentPtr pid, int blockId);

void RestoreScopeInfo(Js::FunctionBody\* functionBody);

void FinishScopeInfo(Js::FunctionBody\* functionBody);

BOOL PnodeLabelNoAST(IdentToken\* pToken, LabelId\* pLabelIdList);

LabelId\* CreateLabelId(IdentToken\* pToken);

void AddToNodeList(ParseNode \*\* ppnodeList, ParseNode \*\*\* pppnodeLast, ParseNode \* pnodeAdd);

void AddToNodeListEscapedUse(ParseNode \*\* ppnodeList, ParseNode \*\*\* pppnodeLast, ParseNode \* pnodeAdd);

void ChkCurTokNoScan(int tk, int wErr)

{

if (m\_token.tk != tk)

{

Error(wErr);

}

}

void ChkCurTok(int tk, int wErr)

{

if (m\_token.tk != tk)

{

Error(wErr);

}

else

{

m\_pscan->Scan();

}

}

void ChkNxtTok(int tk, int wErr)

{

m\_pscan->Scan();

ChkCurTok(tk, wErr);

}

template <class Fn>

void VisitFunctionsInScope(ParseNodePtr pnodeScopeList, Fn fn);

void FinishDeferredFunction(ParseNodePtr pnodeScopeList);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Misc

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

bool m\_UsesArgumentsAtGlobal; // "arguments" used at global code.

BOOL m\_fUseStrictMode; // ES5 Use Strict mode. In AST mode this is a global flag; in NoAST mode it is pushed and popped.

bool m\_InAsmMode; // Currently parsing Asm.Js module

bool m\_deferAsmJs;

BOOL m\_fExpectExternalSource;

BOOL m\_deferringAST;

BOOL m\_stoppedDeferredParse;

enum FncDeclFlag : ushort

{

fFncNoFlgs = 0,

fFncDeclaration = 1 << 0,

fFncNoArg = 1 << 1,

fFncOneArg = 1 << 2, //Force exactly one argument.

fFncNoName = 1 << 3,

fFncLambda = 1 << 4,

fFncMethod = 1 << 5,

fFncClassMember = 1 << 6,

fFncGenerator = 1 << 7,

fFncSetter = 1 << 8,

fFncAsync = 1 << 9,

};

//

// If we need the scanner to force PID creation temporarily, use this auto object

// to turn scanner deferred parsing off temporarily and restore at destructor.

//

class AutoTempForcePid

{

private:

Scanner\_t\* m\_scanner;

BOOL m\_forcePid;

BYTE m\_oldScannerDeferredParseFlags;

public:

AutoTempForcePid(Scanner\_t\* scanner, BOOL forcePid)

: m\_scanner(scanner), m\_forcePid(forcePid)

{

if (forcePid)

{

m\_oldScannerDeferredParseFlags = scanner->SetDeferredParse(FALSE);

}

}

~AutoTempForcePid()

{

if (m\_forcePid)

{

m\_scanner->SetDeferredParseFlags(m\_oldScannerDeferredParseFlags);

}

}

};

public:

charcount\_t GetSourceIchLim() { return m\_sourceLim; }

static BOOL NodeEqualsName(ParseNodePtr pnode, LPCOLESTR sz, ulong cch);

};

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

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

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

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

#include "ParserPch.h"

#include "FormalsUtil.h"

#include "..\Runtime\Language\SourceDynamicProfileManager.h"

#if DBG\_DUMP

void PrintPnodeWIndent(ParseNode \*pnode,int indentAmt);

#endif

const char\* const nopNames[knopLim]= {

#define PTNODE(nop,sn,pc,nk,grfnop,json) sn,

#include "ptlist.h"

};

void printNop(int nop) {

printf("%s\n",nopNames[nop]);

}

const uint ParseNode::mpnopgrfnop[knopLim] =

{

#define PTNODE(nop,sn,pc,nk,grfnop,json) grfnop,

#include "ptlist.h"

};

bool Parser::BindDeferredPidRefs() const

{

return m\_scriptContext->GetConfig()->BindDeferredPidRefs();

}

bool Parser::IsES6DestructuringEnabled() const

{

return m\_scriptContext->GetConfig()->IsES6DestructuringEnabled();

}

struct DeferredFunctionStub

{

RestorePoint restorePoint;

uint fncFlags;

uint nestedCount;

DeferredFunctionStub \*deferredStubs;

#if DEBUG

charcount\_t ichMin;

#endif

};

struct StmtNest

{

union

{

struct

{

ParseNodePtr pnodeStmt; // This statement node.

ParseNodePtr pnodeLab; // Labels for this statement.

};

struct

{

bool isDeferred : 1;

OpCode op; // This statement operation.

LabelId\* pLabelId; // Labels for this statement.

};

};

StmtNest \*pstmtOuter; // Enclosing statement.

};

struct BlockInfoStack

{

StmtNest pstmt;

ParseNode \*pnodeBlock;

ParseNodePtr \*m\_ppnodeLex; // lexical variable list tail

BlockInfoStack \*pBlockInfoOuter; // containing block's BlockInfoStack

BlockInfoStack \*pBlockInfoFunction; // nearest function's BlockInfoStack (if pnodeBlock is a function, this points to itself)

};

#if DEBUG

Parser::Parser(Js::ScriptContext\* scriptContext, BOOL strictMode, PageAllocator \*alloc, bool isBackground, size\_t size)

#else

Parser::Parser(Js::ScriptContext\* scriptContext, BOOL strictMode, PageAllocator \*alloc, bool isBackground)

#endif

: m\_nodeAllocator(L"Parser", alloc ? alloc : scriptContext->GetThreadContext()->GetPageAllocator(), Parser::OutOfMemory),

// use the GuestArena directly for keeping the RegexPattern\* alive during byte code generation

m\_registeredRegexPatterns(scriptContext->GetGuestArena())

{

AssertMsg(size == sizeof(Parser), "verify conditionals affecting the size of Parser agree");

Assert(scriptContext != nullptr);

m\_isInBackground = isBackground;

m\_phtbl = nullptr;

m\_pscan = nullptr;

m\_deferringAST = FALSE;

m\_stoppedDeferredParse = FALSE;

m\_hasParallelJob = false;

m\_doingFastScan = false;

m\_scriptContext = scriptContext;

m\_pCurrentAstSize = nullptr;

m\_parsingDuplicate = 0;

m\_arrayDepth = 0;

m\_funcInArrayDepth = 0;

m\_parenDepth = 0;

m\_funcInArray = 0;

m\_tryCatchOrFinallyDepth = 0;

m\_UsesArgumentsAtGlobal = false;

m\_currentNodeFunc = nullptr;

m\_currentNodeDeferredFunc = nullptr;

m\_currentNodeNonLambdaFunc = nullptr;

m\_currentNodeNonLambdaDeferredFunc = nullptr;

m\_currentNodeProg = nullptr;

m\_currDeferredStub = nullptr;

m\_pstmtCur = nullptr;

m\_currentBlockInfo = nullptr;

m\_currentScope = nullptr;

m\_currentDynamicBlock = nullptr;

m\_catchPidRefList = nullptr;

m\_grfscr = fscrNil;

m\_length = 0;

m\_originalLength = 0;

m\_nextFunctionId = nullptr;

m\_errorCallback = nullptr;

m\_uncertainStructure = FALSE;

currBackgroundParseItem = nullptr;

backgroundParseItems = nullptr;

fastScannedRegExpNodes = nullptr;

m\_fUseStrictMode = strictMode;

m\_InAsmMode = false;

m\_deferAsmJs = true;

m\_scopeCountNoAst = 0;

m\_fExpectExternalSource = 0;

m\_parseType = ParseType\_Upfront;

m\_deferEllipsisError = false;

m\_parsingSuperRestrictionState = ParsingSuperRestrictionState\_SuperDisallowed;

}

Parser::~Parser(void)

{

if (m\_scriptContext == nullptr || m\_scriptContext->GetGuestArena() == nullptr)

{

// If the scriptContext or guestArena have gone away, there is no point clearing each item of this list.

// Just reset it so that destructor of the SList will be no-op

m\_registeredRegexPatterns.Reset();

}

if (this->m\_hasParallelJob)

{

#if ENABLE\_BACKGROUND\_PARSING

// Let the background threads know that they can decommit their arena pages.

BackgroundParser \*bgp = m\_scriptContext->GetBackgroundParser();

Assert(bgp);

if (bgp->Processor()->ProcessesInBackground())

{

JsUtil::BackgroundJobProcessor \*processor = static\_cast<JsUtil::BackgroundJobProcessor\*>(bgp->Processor());

bool result = processor->IterateBackgroundThreads([&](JsUtil::ParallelThreadData \*threadData)->bool {

threadData->canDecommit = true;

return false;

});

Assert(result);

}

#endif

}

Release();

}

void Parser::OutOfMemory()

{

throw ParseExceptionObject(ERRnoMemory);

}

void Parser::Error(HRESULT hr)

{

Assert(FAILED(hr));

m\_err.Throw(hr);

}

void Parser::Error(HRESULT hr, ParseNodePtr pnode)

{

if (pnode && pnode->ichLim)

{

Error(hr, pnode->ichMin, pnode->ichLim);

}

else

{

Error(hr);

}

}

void Parser::Error(HRESULT hr, charcount\_t ichMin, charcount\_t ichLim)

{

m\_pscan->SetErrorPosition(ichMin, ichLim);

Error(hr);

}

void Parser::IdentifierExpectedError(const Token& token)

{

Assert(token.tk != tkID);

HRESULT hr;

if (token.IsReservedWord())

{

if (token.IsKeyword())

{

hr = ERRKeywordNotId;

}

else

{

Assert(token.IsFutureReservedWord(true));

if (token.IsFutureReservedWord(false))

{

// Future reserved word in strict and non-strict modes

hr = ERRFutureReservedWordNotId;

}

else

{

// Future reserved word only in strict mode. The token would have been converted to tkID by the scanner if not

// in strict mode.

Assert(IsStrictMode());

hr = ERRFutureReservedWordInStrictModeNotId;

}

}

}

else

{

hr = ERRnoIdent;

}

Error(hr);

}

CatchPidRefList \*Parser::EnsureCatchPidRefList()

{

if (this->m\_catchPidRefList == nullptr)

{

this->m\_catchPidRefList = Anew(&m\_nodeAllocator, CatchPidRefList);

}

return this->m\_catchPidRefList;

}

HRESULT Parser::ValidateSyntax(LPCUTF8 pszSrc, size\_t encodedCharCount, bool isGenerator, bool isAsync, CompileScriptException \*pse, void (Parser::\*validateFunction)())

{

AssertPsz(pszSrc);

AssertMemN(pse);

if (this->IsBackgroundParser())

{

PROBE\_STACK\_NO\_DISPOSE(m\_scriptContext, Js::Constants::MinStackDefault);

}

else

{

PROBE\_STACK(m\_scriptContext, Js::Constants::MinStackDefault);

}

HRESULT hr;

SmartFPUControl smartFpuControl;

DebugOnly( m\_err.fInited = TRUE; )

BOOL fDeferSave = m\_deferringAST;

try

{

hr = NOERROR;

this->PrepareScanner(false);

m\_length = encodedCharCount;

m\_originalLength = encodedCharCount;

// make sure deferred parsing is turned off

ULONG grfscr = fscrNil;

// Give the scanner the source and get the first token

m\_pscan->SetText(pszSrc, 0, encodedCharCount, 0, grfscr);

m\_pscan->SetYieldIsKeyword(isGenerator);

m\_pscan->SetAwaitIsKeyword(isAsync);

m\_pscan->Scan();

uint nestedCount = 0;

m\_pnestedCount = &nestedCount;

ParseNodePtr pnodeScope = nullptr;

m\_ppnodeScope = &pnodeScope;

m\_ppnodeExprScope = nullptr;

uint nextFunctionId = 0;

m\_nextFunctionId = &nextFunctionId;

m\_inDeferredNestedFunc = false;

m\_deferringAST = true;

m\_nextBlockId = 0;

if (this->BindDeferredPidRefs())

{

ParseNode \*pnodeFnc = CreateNode(knopFncDecl);

pnodeFnc->sxFnc.ClearFlags();

pnodeFnc->sxFnc.SetDeclaration(false);

pnodeFnc->sxFnc.astSize = 0;

pnodeFnc->sxFnc.pnodeVars = nullptr;

pnodeFnc->sxFnc.pnodeArgs = nullptr;

pnodeFnc->sxFnc.pnodeBody = nullptr;

pnodeFnc->sxFnc.pnodeName = nullptr;

pnodeFnc->sxFnc.pnodeRest = nullptr;

pnodeFnc->sxFnc.deferredStub = nullptr;

pnodeFnc->sxFnc.SetIsGenerator(isGenerator);

pnodeFnc->sxFnc.SetIsAsync(isAsync);

m\_ppnodeVar = &pnodeFnc->sxFnc.pnodeVars;

m\_currentNodeFunc = pnodeFnc;

m\_currentNodeDeferredFunc = NULL;

AssertMsg(m\_pstmtCur == NULL, "Statement stack should be empty when we start parse function body");

ParseNodePtr block = StartParseBlock<false>(PnodeBlockType::Function, ScopeType\_FunctionBody);

(this->\*validateFunction)();

FinishParseBlock(block);

pnodeFnc->ichLim = m\_pscan->IchLimTok();

pnodeFnc->sxFnc.cbLim = m\_pscan->IecpLimTok();

pnodeFnc->sxFnc.pnodeVars = nullptr;

if (m\_asgToConst)

{

Error(ERRAssignmentToConst, m\_asgToConst.GetIchMin(), m\_asgToConst.GetIchLim());

}

}

else

{

(this->\*validateFunction)();

}

// there should be nothing after successful parsing for a given construct

if (m\_token.tk != tkEOF)

Error(ERRsyntax);

RELEASEPTR(m\_pscan);

m\_deferringAST = fDeferSave;

}

catch(ParseExceptionObject& e)

{

m\_deferringAST = fDeferSave;

m\_err.m\_hr = e.GetError();

hr = pse->ProcessError( m\_pscan, m\_err.m\_hr, /\* pnodeBase \*/ NULL);

}

return hr;

}

HRESULT Parser::ParseSourceInternal(

\_\_out ParseNodePtr\* parseTree, LPCUTF8 pszSrc, size\_t offsetInBytes, size\_t encodedCharCount, charcount\_t offsetInChars,

bool fromExternal, ULONG grfscr, CompileScriptException \*pse, Js::LocalFunctionId \* nextFunctionId, ULONG lineNumber, SourceContextInfo \* sourceContextInfo)

{

AssertMem(parseTree);

AssertPsz(pszSrc);

AssertMemN(pse);

double startTime = m\_scriptContext->GetThreadContext()->ParserTelemetry.Now();

if (this->IsBackgroundParser())

{

PROBE\_STACK\_NO\_DISPOSE(m\_scriptContext, Js::Constants::MinStackDefault);

}

else

{

PROBE\_STACK(m\_scriptContext, Js::Constants::MinStackDefault);

}

#ifdef PROFILE\_EXEC

m\_scriptContext->ProfileBegin(Js::ParsePhase);

#endif

JS\_ETW(EventWriteJSCRIPT\_PARSE\_START(m\_scriptContext,0));

\*parseTree = NULL;

m\_sourceLim = 0;

m\_grfscr = grfscr;

m\_sourceContextInfo = sourceContextInfo;

ParseNodePtr pnodeBase = NULL;

HRESULT hr;

SmartFPUControl smartFpuControl;

DebugOnly( m\_err.fInited = TRUE; )

try

{

this->PrepareScanner(fromExternal);

if ((grfscr & fscrEvalCode) != 0)

{

// This makes the parser to believe when eval() is called, it accept any super access in global scope.

this->m\_parsingSuperRestrictionState = Parser::ParsingSuperRestrictionState\_SuperCallAndPropertyAllowed;

}

// parse the source

pnodeBase = Parse(pszSrc, offsetInBytes, encodedCharCount, offsetInChars, grfscr, lineNumber, nextFunctionId, pse);

AssertNodeMem(pnodeBase);

// Record the actual number of words parsed.

m\_sourceLim = pnodeBase->ichLim - offsetInChars;

// TODO: The assert can be false positive in some scenarios and chuckj to fix it later

// Assert(utf8::ByteIndexIntoCharacterIndex(pszSrc + offsetInBytes, encodedCharCount, fromExternal ? utf8::doDefault : utf8::doAllowThreeByteSurrogates) == m\_sourceLim);

#if DBG\_DUMP

if (Js::Configuration::Global.flags.Trace.IsEnabled(Js::ParsePhase))

{

PrintPnodeWIndent(pnodeBase,4);

fflush(stdout);

}

#endif

\*parseTree = pnodeBase;

hr = NOERROR;

}

catch(ParseExceptionObject& e)

{

m\_err.m\_hr = e.GetError();

hr = pse->ProcessError( m\_pscan, m\_err.m\_hr, pnodeBase);

}

if (this->m\_hasParallelJob)

{

#if ENABLE\_BACKGROUND\_PARSING

///// Wait here for remaining jobs to finish. Then look for errors, do final const bindings.

// pleath TODO: If there are remaining jobs, let the main thread help finish them.

BackgroundParser \*bgp = m\_scriptContext->GetBackgroundParser();

Assert(bgp);

CompileScriptException se;

this->WaitForBackgroundJobs(bgp, &se);

BackgroundParseItem \*failedItem = bgp->GetFailedBackgroundParseItem();

if (failedItem)

{

CompileScriptException \*bgPse = failedItem->GetPSE();

Assert(bgPse);

\*pse = \*bgPse;

hr = failedItem->GetHR();

bgp->SetFailedBackgroundParseItem(nullptr);

}

if (this->fastScannedRegExpNodes != nullptr)

{

this->FinishBackgroundRegExpNodes();

}

for (BackgroundParseItem \*item = this->backgroundParseItems; item; item = item->GetNext())

{

Parser \*parser = item->GetParser();

parser->FinishBackgroundPidRefs(item, this != parser);

}

#endif

}

// done with the scanner

RELEASEPTR(m\_pscan);

#ifdef PROFILE\_EXEC

m\_scriptContext->ProfileEnd(Js::ParsePhase);

#endif

JS\_ETW(EventWriteJSCRIPT\_PARSE\_STOP(m\_scriptContext, 0));

ThreadContext \*threadContext = m\_scriptContext->GetThreadContext();

threadContext->ParserTelemetry.LogTime(threadContext->ParserTelemetry.Now() - startTime);

return hr;

}

#if ENABLE\_BACKGROUND\_PARSING

void Parser::WaitForBackgroundJobs(BackgroundParser \*bgp, CompileScriptException \*pse)

{

// The scan of the script is done, but there may be unfinished background jobs in the queue.

// Enlist the main thread to help with those.

BackgroundParseItem \*item;

if (!\*bgp->GetPendingBackgroundItemsPtr())

{

// We're done.

return;

}

// Save parser state, since we'll need to restore it in order to bind references correctly later.

this->m\_isInBackground = true;

this->SetCurrBackgroundParseItem(nullptr);

uint blockIdSave = this->m\_nextBlockId;

uint functionIdSave = \*this->m\_nextFunctionId;

StmtNest \*pstmtSave = this->m\_pstmtCur;

if (!bgp->Processor()->ProcessesInBackground())

{

// No background thread. Just walk the jobs with no locking and process them.

for (item = bgp->GetNextUnprocessedItem(); item; item = bgp->GetNextUnprocessedItem())

{

bgp->Processor()->RemoveJob(item);

bool succeeded = bgp->Process(item, this, pse);

bgp->JobProcessed(item, succeeded);

}

Assert(!\*bgp->GetPendingBackgroundItemsPtr());

}

else

{

// Background threads. We need to have the critical section in order to:

// - Check for unprocessed jobs;

// - Remove jobs from the processor queue;

// - Do JobsProcessed work (such as removing jobs from the BackgroundParser's unprocessed list).

CriticalSection \*pcs = static\_cast<JsUtil::BackgroundJobProcessor\*>(bgp->Processor())->GetCriticalSection();

pcs->Enter();

for (;;)

{

// Grab a job (in lock)

item = bgp->GetNextUnprocessedItem();

if (item == nullptr)

{

break;

}

bgp->Processor()->RemoveJob(item);

pcs->Leave();

// Process job (if there is one) (outside lock)

bool succeeded = bgp->Process(item, this, pse);

pcs->Enter();

bgp->JobProcessed(item, succeeded);

}

pcs->Leave();

// Wait for the background threads to finish jobs they're already processing (if any).

// TODO: Replace with a proper semaphore.

while(\*bgp->GetPendingBackgroundItemsPtr());

}

Assert(!\*bgp->GetPendingBackgroundItemsPtr());

// Restore parser state.

this->m\_pstmtCur = pstmtSave;

this->m\_isInBackground = false;

this->m\_nextBlockId = blockIdSave;

\*this->m\_nextFunctionId = functionIdSave;

}

void Parser::FinishBackgroundPidRefs(BackgroundParseItem \*item, bool isOtherParser)

{

for (BlockInfoStack \*blockInfo = item->GetParseContext()->currentBlockInfo; blockInfo; blockInfo = blockInfo->pBlockInfoOuter)

{

if (isOtherParser)

{

this->BindPidRefs<true>(blockInfo, item->GetMaxBlockId());

}

else

{

this->BindPidRefs<false>(blockInfo, item->GetMaxBlockId());

}

}

}

void Parser::FinishBackgroundRegExpNodes()

{

// We have a list of RegExp nodes that we saw on the UI thread in functions we're parallel parsing,

// and for each background job we have a list of RegExp nodes for which we couldn't allocate patterns.

// We need to copy the pattern pointers from the UI thread nodes to the corresponding nodes on the

// background nodes.

// There may be UI thread nodes for which there are no background thread equivalents, because the UI thread

// has to assume that the background thread won't defer anything.

// Note that because these lists (and the list of background jobs) are SList's built by prepending, they are

// all in reverse lexical order.

Assert(!this->IsBackgroundParser());

Assert(this->fastScannedRegExpNodes);

Assert(this->backgroundParseItems != nullptr);

BackgroundParseItem \*currBackgroundItem;

#if DBG

for (currBackgroundItem = this->backgroundParseItems;

currBackgroundItem;

currBackgroundItem = currBackgroundItem->GetNext())

{

if (currBackgroundItem->RegExpNodeList())

{

FOREACH\_DLIST\_ENTRY(ParseNodePtr, ArenaAllocator, pnode, currBackgroundItem->RegExpNodeList())

{

Assert(pnode->sxPid.regexPattern == nullptr);

}

NEXT\_DLIST\_ENTRY;

}

}

#endif

// Hook up the patterns allocated on the main thread to the nodes created on the background thread.

// Walk the list of foreground nodes, advancing through the work items and looking up each item.

// Note that the background thread may have chosen to defer a given RegEx literal, so not every foreground

// node will have a matching background node. Doesn't matter for correctness.

// (It's inefficient, of course, to have to restart the inner loop from the beginning of the work item's

// list, but it should be unusual to have many RegExes in a single work item's chunk of code. Figure out how

// to start the inner loop from a known internal node within the list if that turns out to be important.)

currBackgroundItem = this->backgroundParseItems;

FOREACH\_DLIST\_ENTRY(ParseNodePtr, ArenaAllocator, pnodeFgnd, this->fastScannedRegExpNodes)

{

Assert(pnodeFgnd->nop == knopRegExp);

Assert(pnodeFgnd->sxPid.regexPattern != nullptr);

bool quit = false;

while (!quit)

{

// Find the next work item with a RegEx in it.

while (currBackgroundItem && currBackgroundItem->RegExpNodeList() == nullptr)

{

currBackgroundItem = currBackgroundItem->GetNext();

}

if (!currBackgroundItem)

{

break;

}

// Walk the RegExps in the work item.

FOREACH\_DLIST\_ENTRY(ParseNodePtr, ArenaAllocator, pnodeBgnd, currBackgroundItem->RegExpNodeList())

{

Assert(pnodeBgnd->nop == knopRegExp);

if (pnodeFgnd->ichMin <= pnodeBgnd->ichMin)

{

// Either we found a match, or the next background node is past the foreground node.

// In any case, we can stop searching.

if (pnodeFgnd->ichMin == pnodeBgnd->ichMin)

{

Assert(pnodeFgnd->ichLim == pnodeBgnd->ichLim);

pnodeBgnd->sxPid.regexPattern = pnodeFgnd->sxPid.regexPattern;

}

quit = true;

break;

}

}

NEXT\_DLIST\_ENTRY;

if (!quit)

{

// Need to advance to the next work item.

currBackgroundItem = currBackgroundItem->GetNext();

}

}

}

NEXT\_DLIST\_ENTRY;

#if DBG

for (currBackgroundItem = this->backgroundParseItems;

currBackgroundItem;

currBackgroundItem = currBackgroundItem->GetNext())

{

if (currBackgroundItem->RegExpNodeList())

{

FOREACH\_DLIST\_ENTRY(ParseNodePtr, ArenaAllocator, pnode, currBackgroundItem->RegExpNodeList())

{

Assert(pnode->sxPid.regexPattern != nullptr);

}

NEXT\_DLIST\_ENTRY;

}

}

#endif

}

#endif

LabelId\* Parser::CreateLabelId(IdentToken\* pToken)

{

LabelId\* pLabelId;

pLabelId = (LabelId\*)m\_nodeAllocator.Alloc(sizeof(LabelId));

if (NULL == pLabelId)

Error(ERRnoMemory);

pLabelId->pid = pToken->pid;

pLabelId->next = NULL;

return pLabelId;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The following set of routines allocate parse tree nodes of various kinds.

They catch an exception on out of memory.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static const int g\_mpnopcbNode[] =

{

#define PTNODE(nop,sn,pc,nk,ok,json) kcbPn##nk,

#include "ptlist.h"

};

const Js::RegSlot NoRegister = (Js::RegSlot)-1;

const Js::RegSlot OneByteRegister = (Js::RegSlot\_OneByte)-1;

void Parser::InitNode(OpCode nop,ParseNodePtr pnode) {

pnode->nop = nop;

pnode->grfpn = PNodeFlags::fpnNone;

pnode->location = NoRegister;

pnode->emitLabels = false;

pnode->isUsed = true;

pnode->notEscapedUse = false;

pnode->isInList = false;

pnode->isCallApplyTargetLoad = false;

}

// Create nodes using Arena

template <OpCode nop>

ParseNodePtr Parser::StaticCreateNodeT(ArenaAllocator\* alloc, charcount\_t ichMin, charcount\_t ichLim)

{

ParseNodePtr pnode = StaticAllocNode<nop>(alloc);

InitNode(nop,pnode);

// default - may be changed

pnode->ichMin = ichMin;

pnode->ichLim = ichLim;

return pnode;

}

ParseNodePtr

Parser::StaticCreateBlockNode(ArenaAllocator\* alloc, charcount\_t ichMin , charcount\_t ichLim, int blockId, PnodeBlockType blockType)

{

ParseNodePtr pnode = StaticCreateNodeT<knopBlock>(alloc, ichMin, ichLim);

InitBlockNode(pnode, blockId, blockType);

return pnode;

}

void Parser::InitBlockNode(ParseNodePtr pnode, int blockId, PnodeBlockType blockType)

{

Assert(pnode->nop == knopBlock);

pnode->sxBlock.pnodeScopes = nullptr;

pnode->sxBlock.pnodeNext = nullptr;

pnode->sxBlock.scope = nullptr;

pnode->sxBlock.enclosingBlock = nullptr;

pnode->sxBlock.pnodeLexVars = nullptr;

pnode->sxBlock.pnodeStmt = nullptr;

pnode->sxBlock.pnodeLastValStmt = nullptr;

pnode->sxBlock.callsEval = false;

pnode->sxBlock.childCallsEval = false;

pnode->sxBlock.blockType = blockType;

pnode->sxBlock.blockId = blockId;

if (blockType != PnodeBlockType::Regular)

{

pnode->grfpn |= PNodeFlags::fpnSyntheticNode;

}

}

// Create Node with limit

template <OpCode nop>

ParseNodePtr Parser::CreateNodeT(charcount\_t ichMin,charcount\_t ichLim)

{

Assert(!this->m\_deferringAST);

ParseNodePtr pnode = StaticCreateNodeT<nop>(&m\_nodeAllocator, ichMin, ichLim);

Assert(m\_pCurrentAstSize != NULL);

\*m\_pCurrentAstSize += GetNodeSize<nop>();

return pnode;

}

ParseNodePtr Parser::CreateDeclNode(OpCode nop, IdentPtr pid, SymbolType symbolType, bool errorOnRedecl)

{

ParseNodePtr pnode = CreateNode(nop);

pnode->sxVar.InitDeclNode(pid, NULL);

if (symbolType != STUnknown)

{

pnode->sxVar.sym = AddDeclForPid(pnode, pid, symbolType, errorOnRedecl);

}

return pnode;

}

Symbol\* Parser::AddDeclForPid(ParseNodePtr pnode, IdentPtr pid, SymbolType symbolType, bool errorOnRedecl)

{

Assert(pnode->IsVarLetOrConst());

PidRefStack \*refForUse = nullptr, \*refForDecl = nullptr;

BlockInfoStack \*blockInfo;

bool fBlockScope = false;

if (m\_scriptContext->GetConfig()->IsBlockScopeEnabled() &&

(pnode->nop != knopVarDecl || symbolType == STFunction))

{

Assert(m\_pstmtCur);

if (m\_pstmtCur->isDeferred)

{

// Deferred parsing: there's no pnodeStmt node, only an opcode on the Stmt struct.

if (m\_pstmtCur->op != knopBlock)

{

// Let/const declared in a bare statement context.

Error(ERRDeclOutOfStmt);

}

if (m\_pstmtCur->pstmtOuter && m\_pstmtCur->pstmtOuter->op == knopSwitch)

{

// Let/const declared inside a switch block (requiring conservative use-before-decl check).

pnode->sxVar.isSwitchStmtDecl = true;

}

}

else

{

if (m\_pstmtCur->pnodeStmt->nop != knopBlock)

{

// Let/const declared in a bare statement context.

Error(ERRDeclOutOfStmt);

}

if (m\_pstmtCur->pstmtOuter && m\_pstmtCur->pstmtOuter->pnodeStmt->nop == knopSwitch)

{

// Let/const declared inside a switch block (requiring conservative use-before-decl check).

pnode->sxVar.isSwitchStmtDecl = true;

}

}

fBlockScope = pnode->nop != knopVarDecl ||

(

!GetCurrentBlockInfo()->pnodeBlock->sxBlock.scope ||

GetCurrentBlockInfo()->pnodeBlock->sxBlock.scope->GetScopeType() != ScopeType\_GlobalEvalBlock

);

}

if (fBlockScope)

{

blockInfo = GetCurrentBlockInfo();

}

else

{

blockInfo = GetCurrentFunctionBlockInfo();

}

// If we are creating an 'arguments' Sym at function block scope, create it in

// the parameter scope instead. That way, if we need to reuse the Sym for the

// actual arguments object at the end of the function, we don't need to move it

// into the parameter scope.

if (pid == wellKnownPropertyPids.arguments

&& pnode->nop == knopVarDecl

&& blockInfo->pnodeBlock->sxBlock.blockType == PnodeBlockType::Function

&& blockInfo->pBlockInfoOuter != nullptr

&& blockInfo->pBlockInfoOuter->pnodeBlock->sxBlock.blockType == PnodeBlockType::Parameter)

{

blockInfo = blockInfo->pBlockInfoOuter;

}

int maxScopeId = blockInfo->pnodeBlock->sxBlock.blockId;

// The body of catch may have let declared variable. In the case of pattern, found at catch parameter level,

// we need to search the duplication at that scope level as well - thus extending the scope lookup range.

if (IsES6DestructuringEnabled()

&& fBlockScope

&& blockInfo->pBlockInfoOuter != nullptr

&& blockInfo->pBlockInfoOuter->pnodeBlock->sxBlock.scope != nullptr

&& blockInfo->pBlockInfoOuter->pnodeBlock->sxBlock.scope->GetScopeType() == ScopeType\_CatchParamPattern)

{

maxScopeId = blockInfo->pBlockInfoOuter->pnodeBlock->sxBlock.blockId;

}

if (blockInfo->pnodeBlock->sxBlock.scope != nullptr && blockInfo->pnodeBlock->sxBlock.scope->GetScopeType() == ScopeType\_FunctionBody)

{

// Check if there is a parameter scope and try to get it first.

BlockInfoStack \*outerBlockInfo = blockInfo->pBlockInfoOuter;

if (outerBlockInfo != nullptr && outerBlockInfo->pnodeBlock->sxBlock.blockType == PnodeBlockType::Parameter)

{

maxScopeId = outerBlockInfo->pnodeBlock->sxBlock.blockId;

}

}

refForDecl = this->FindOrAddPidRef(pid, blockInfo->pnodeBlock->sxBlock.blockId, maxScopeId);

if (refForDecl == nullptr)

{

Error(ERRnoMemory);

}

if (blockInfo == GetCurrentBlockInfo())

{

refForUse = refForDecl;

}

else

{

refForUse = this->PushPidRef(pid);

}

pnode->sxVar.symRef = refForUse->GetSymRef();

Symbol \*sym = refForDecl->GetSym();

if (sym != nullptr)

{

// Multiple declarations in the same scope. 3 possibilities: error, existing one wins, new one wins.

switch (pnode->nop)

{

case knopLetDecl:

case knopConstDecl:

if (!sym->GetDecl()->sxVar.isBlockScopeFncDeclVar)

{

Assert(errorOnRedecl);

// Redeclaration error.

Error(ERRRedeclaration);

}

else

{

// (New) let/const hides the (old) var

sym->SetSymbolType(symbolType);

sym->SetDecl(pnode);

}

break;

case knopVarDecl:

if (sym->GetDecl() == nullptr)

{

Assert(symbolType == STFunction);

sym->SetDecl(pnode);

break;

}

switch (sym->GetDecl()->nop)

{

case knopLetDecl:

case knopConstDecl:

// Destructuring made possible to have the formals to be the let bind. But that shouldn't throw the error.

if (errorOnRedecl && (!IsES6DestructuringEnabled() || sym->GetSymbolType() != STFormal))

{

Error(ERRRedeclaration);

}

// If !errorOnRedecl, (old) let/const hides the (new) var, so do nothing.

break;

case knopVarDecl:

// Legal redeclaration. Who wins?

if (errorOnRedecl || sym->GetDecl()->sxVar.isBlockScopeFncDeclVar)

{

if (symbolType == STFormal ||

(symbolType == STFunction && sym->GetSymbolType() != STFormal) ||

sym->GetSymbolType() == STVariable)

{

// New decl wins.

sym->SetSymbolType(symbolType);

sym->SetDecl(pnode);

}

}

break;

}

break;

}

}

else

{

Scope \*scope = blockInfo->pnodeBlock->sxBlock.scope;

if (scope == nullptr)

{

Assert(blockInfo->pnodeBlock->sxBlock.blockType == PnodeBlockType::Regular &&

m\_scriptContext->GetConfig()->IsBlockScopeEnabled());

scope = Anew(&m\_nodeAllocator, Scope, &m\_nodeAllocator, ScopeType\_Block);

blockInfo->pnodeBlock->sxBlock.scope = scope;

PushScope(scope);

}

if (scope->GetScopeType() == ScopeType\_GlobalEvalBlock)

{

Assert(fBlockScope);

Assert(scope->GetEnclosingScope() == m\_currentNodeProg->sxProg.scope);

// Check for same-named decl in Global scope.

PidRefStack \*pidRefOld = pid->GetPidRefForScopeId(0);

if (pidRefOld && pidRefOld->GetSym())

{

Error(ERRRedeclaration);

}

}

else if (scope->GetScopeType() == ScopeType\_Global && (this->m\_grfscr & fscrEvalCode) &&

!(m\_functionBody && m\_functionBody->GetScopeInfo()))

{

// Check for same-named decl in GlobalEvalBlock scope. Note that this is not necessary

// if we're compiling a deferred nested function and the global scope was restored from cached info,

// because in that case we don't need a GlobalEvalScope.

Assert(!fBlockScope || (this->m\_grfscr & fscrConsoleScopeEval) == fscrConsoleScopeEval);

PidRefStack \*pidRefOld = pid->GetPidRefForScopeId(1);

if (pidRefOld && pidRefOld->GetSym())

{

Error(ERRRedeclaration);

}

}

if ((scope->GetScopeType() == ScopeType\_FunctionBody || scope->GetScopeType() == ScopeType\_Parameter) && symbolType != STFunction)

{

ParseNodePtr pnodeFnc = GetCurrentFunctionNode();

AnalysisAssert(pnodeFnc);

if (pnodeFnc->sxFnc.pnodeName &&

pnodeFnc->sxFnc.pnodeName->nop == knopVarDecl &&

pnodeFnc->sxFnc.pnodeName->sxVar.pid == pid)

{

// Named function expression has its name hidden by a local declaration.

// This is important to know if we don't know whether nested deferred functions refer to it,

// because if the name has a non-local reference then we have to create a scope object.

m\_currentNodeFunc->sxFnc.SetNameIsHidden();

}

}

if (!sym)

{

const wchar\_t \*name = reinterpret\_cast<const wchar\_t\*>(pid->Psz());

int nameLength = pid->Cch();

SymbolName const symName(name, nameLength);

Assert(!scope->FindLocalSymbol(symName));

sym = Anew(&m\_nodeAllocator, Symbol, symName, pnode, symbolType);

scope->AddNewSymbol(sym);

sym->SetPid(pid);

}

refForDecl->SetSym(sym);

}

return sym;

}

void Parser::RestorePidRefForSym(Symbol \*sym)

{

IdentPtr pid = m\_pscan->m\_phtbl->PidHashNameLen(sym->GetName().GetBuffer(), sym->GetName().GetLength());

Assert(pid);

sym->SetPid(pid);

PidRefStack \*ref = this->PushPidRef(pid);

ref->SetSym(sym);

}

IdentPtr Parser::GenerateIdentPtr(\_\_ecount(len) wchar\_t\* name, long len)

{

return m\_phtbl->PidHashNameLen(name,len);

}

IdentPtr Parser::PidFromNode(ParseNodePtr pnode)

{

for (;;)

{

switch (pnode->nop)

{

case knopName:

return pnode->sxPid.pid;

case knopVarDecl:

return pnode->sxVar.pid;

case knopDot:

Assert(pnode->sxBin.pnode2->nop == knopName);

return pnode->sxBin.pnode2->sxPid.pid;

case knopComma:

// Advance to the RHS and iterate.

pnode = pnode->sxBin.pnode2;

break;

default:

return nullptr;

}

}

}

#if DBG

void VerifyNodeSize(OpCode nop, int size)

{

Assert(nop >= 0 && nop < knopLim);

\_\_analysis\_assume(nop < knopLim);

Assert(g\_mpnopcbNode[nop] == size);

}

#endif

ParseNodePtr Parser::StaticCreateBinNode(OpCode nop, ParseNodePtr pnode1,

ParseNodePtr pnode2,ArenaAllocator\* alloc)

{

DebugOnly(VerifyNodeSize(nop, kcbPnBin));

ParseNodePtr pnode = (ParseNodePtr)alloc->Alloc(kcbPnBin);

InitNode(nop, pnode);

pnode->sxBin.pnodeNext = nullptr;

pnode->sxBin.pnode1 = pnode1;

pnode->sxBin.pnode2 = pnode2;

// Statically detect if the add is a concat

if (!PHASE\_OFF1(Js::ByteCodeConcatExprOptPhase))

{

// We can't flatten the concat expression if the LHS is not a flatten concat already

// e.g. a + (<str> + b)

// Side effect of ToStr(b) need to happen first before ToStr(a)

// If we flatten the concat expression, we will do ToStr(a) before ToStr(b)

if ((nop == knopAdd) && (pnode1->CanFlattenConcatExpr() || pnode2->nop == knopStr))

{

pnode->grfpn |= fpnCanFlattenConcatExpr;

}

}

return pnode;

}

// Create nodes using parser allocator

ParseNodePtr Parser::CreateNode(OpCode nop, charcount\_t ichMin)

{

bool nodeAllowed = IsNodeAllowedForDeferParse(nop);

Assert(nodeAllowed);

Assert(nop >= 0 && nop < knopLim);

ParseNodePtr pnode;

int cb = (nop >= knopNone && nop < knopLim) ? g\_mpnopcbNode[nop] : g\_mpnopcbNode[knopEmpty];

pnode = (ParseNodePtr)m\_nodeAllocator.Alloc(cb);

Assert(pnode != nullptr);

if (!m\_deferringAST)

{

Assert(m\_pCurrentAstSize != nullptr);

\*m\_pCurrentAstSize += cb;

}

InitNode(nop,pnode);

// default - may be changed

pnode->ichMin = ichMin;

if (m\_pscan!= nullptr) {

pnode->ichLim = m\_pscan->IchLimTok();

}

else pnode->ichLim=0;

return pnode;

}

ParseNodePtr Parser::CreateUniNode(OpCode nop, ParseNodePtr pnode1)

{

Assert(!this->m\_deferringAST);

DebugOnly(VerifyNodeSize(nop, kcbPnUni));

ParseNodePtr pnode = (ParseNodePtr)m\_nodeAllocator.Alloc(kcbPnUni);

Assert(m\_pCurrentAstSize != nullptr);

\*m\_pCurrentAstSize += kcbPnUni;

InitNode(nop, pnode);

pnode->sxUni.pnode1 = pnode1;

if (nullptr == pnode1)

{

// no ops

pnode->ichMin = m\_pscan->IchMinTok();

pnode->ichLim = m\_pscan->IchLimTok();

}

else

{

// 1 op

pnode->ichMin = pnode1->ichMin;

pnode->ichLim = pnode1->ichLim;

this->CheckArguments(pnode);

}

return pnode;

}

ParseNodePtr Parser::CreateBinNode(OpCode nop, ParseNodePtr pnode1, ParseNodePtr pnode2)

{

Assert(!this->m\_deferringAST);

charcount\_t ichMin;

charcount\_t ichLim;

if (nullptr == pnode1)

{

// no ops

Assert(nullptr == pnode2);

ichMin = m\_pscan->IchMinTok();

ichLim = m\_pscan->IchLimTok();

}

else

{

if (nullptr == pnode2)

{

// 1 op

ichMin = pnode1->ichMin;

ichLim = pnode1->ichLim;

}

else

{

// 2 ops

ichMin = pnode1->ichMin;

ichLim = pnode2->ichLim;

if (nop != knopDot && nop != knopIndex)

{

this->CheckArguments(pnode2);

}

}

if (nop != knopDot && nop != knopIndex)

{

this->CheckArguments(pnode1);

}

}

return CreateBinNode(nop, pnode1, pnode2, ichMin, ichLim);

}

ParseNodePtr Parser::CreateTriNode(OpCode nop, ParseNodePtr pnode1,

ParseNodePtr pnode2, ParseNodePtr pnode3)

{

charcount\_t ichMin;

charcount\_t ichLim;

if (nullptr == pnode1)

{

// no ops

Assert(nullptr == pnode2);

Assert(nullptr == pnode3);

ichMin = m\_pscan->IchMinTok();

ichLim = m\_pscan->IchLimTok();

}

else if (nullptr == pnode2)

{

// 1 op

Assert(nullptr == pnode3);

ichMin = pnode1->ichMin;

ichLim = pnode1->ichLim;

}

else if (nullptr == pnode3)

{

// 2 op

ichMin = pnode1->ichMin;

ichLim = pnode2->ichLim;

}

else

{

// 3 ops

ichMin = pnode1->ichMin;

ichLim = pnode3->ichLim;

}

return CreateTriNode(nop, pnode1, pnode2, pnode3, ichMin, ichLim);

}

ParseNodePtr Parser::CreateBlockNode(charcount\_t ichMin,charcount\_t ichLim, PnodeBlockType blockType)

{

return StaticCreateBlockNode(&m\_nodeAllocator, ichMin, ichLim, this->m\_nextBlockId++, blockType);

}

ParseNodePtr

Parser::CreateCallNode(OpCode nop, ParseNodePtr pnode1, ParseNodePtr pnode2,charcount\_t ichMin,charcount\_t ichLim)

{

Assert(!this->m\_deferringAST);

DebugOnly(VerifyNodeSize(nop, kcbPnCall));

ParseNodePtr pnode = (ParseNodePtr)m\_nodeAllocator.Alloc(kcbPnCall);

Assert(m\_pCurrentAstSize != nullptr);

\*m\_pCurrentAstSize += kcbPnCall;

InitNode(nop, pnode);

pnode->sxCall.pnodeTarget = pnode1;

pnode->sxCall.pnodeArgs = pnode2;

pnode->sxCall.argCount = 0;

pnode->sxCall.spreadArgCount = 0;

pnode->sxCall.callOfConstants = false;

pnode->sxCall.isApplyCall = false;

pnode->sxCall.isEvalCall = false;

pnode->ichMin = ichMin;

pnode->ichLim = ichLim;

return pnode;

}

ParseNodePtr Parser::CreateStrNode(IdentPtr pid)

{

Assert(!this->m\_deferringAST);

ParseNodePtr pnode = CreateNode(knopStr);

pnode->sxPid.pid=pid;

pnode->grfpn |= PNodeFlags::fpnCanFlattenConcatExpr;

return pnode;

}

ParseNodePtr Parser::CreateIntNode(long lw)

{

ParseNodePtr pnode = CreateNode(knopInt);

pnode->sxInt.lw = lw;

return pnode;

}

// Create Node with scanner limit

template <OpCode nop>

ParseNodePtr Parser::CreateNodeWithScanner()

{

Assert(m\_pscan != nullptr);

return CreateNodeWithScanner<nop>(m\_pscan->IchMinTok());

}

template <OpCode nop>

ParseNodePtr Parser::CreateNodeWithScanner(charcount\_t ichMin)

{

Assert(m\_pscan != nullptr);

return CreateNodeT<nop>(ichMin, m\_pscan->IchLimTok());

}

ParseNodePtr Parser::CreateCallNode(OpCode nop, ParseNodePtr pnode1, ParseNodePtr pnode2)

{

charcount\_t ichMin;

charcount\_t ichLim;

if (nullptr == pnode1)

{

Assert(nullptr == pnode2);

ichMin = m\_pscan->IchMinTok();

ichLim = m\_pscan->IchLimTok();

}

else

{

if (nullptr == pnode2)

{

ichMin = pnode1->ichMin;

ichLim = pnode1->ichLim;

}

else

{

ichMin = pnode1->ichMin;

ichLim = pnode2->ichLim;

}

if (pnode1->nop == knopDot || pnode1->nop == knopIndex)

{

this->CheckArguments(pnode1->sxBin.pnode1);

}

}

return CreateCallNode(nop, pnode1, pnode2, ichMin, ichLim);

}

ParseNodePtr Parser::CreateStrNodeWithScanner(IdentPtr pid)

{

Assert(!this->m\_deferringAST);

ParseNodePtr pnode = CreateNodeWithScanner<knopStr>();

pnode->sxPid.pid=pid;

pnode->grfpn |= PNodeFlags::fpnCanFlattenConcatExpr;

return pnode;

}

ParseNodePtr Parser::CreateIntNodeWithScanner(long lw)

{

Assert(!this->m\_deferringAST);

ParseNodePtr pnode = CreateNodeWithScanner<knopInt>();

pnode->sxInt.lw = lw;

return pnode;

}

ParseNodePtr Parser::CreateTempNode(ParseNode\* initExpr)

{

ParseNodePtr pnode = CreateNode(knopTemp, (charcount\_t)0);

pnode->sxVar.pnodeInit =initExpr;

pnode->sxVar.pnodeNext = nullptr;

return pnode;

}

ParseNodePtr Parser::CreateTempRef(ParseNode\* tempNode)

{

ParseNodePtr pnode = CreateUniNode(knopTempRef, tempNode);

return pnode;

}

void Parser::CheckPidIsValid(IdentPtr pid, bool autoArgumentsObject)

{

if (IsStrictMode())

{

// in strict mode, variable named 'eval' cannot be created

if (pid == wellKnownPropertyPids.eval)

{

Error(ERREvalUsage);

}

else if (pid == wellKnownPropertyPids.arguments && !autoArgumentsObject)

{

Error(ERRArgsUsage);

}

}

}

// CreateVarDecl needs m\_ppnodeVar to be pointing to the right function.

// Post-parsing rewriting during bytecode gen may have m\_ppnodeVar pointing to the last parsed function.

// This function sets up m\_ppnodeVar to point to the given pnodeFnc and creates the new var declaration.

// This prevents accidentally adding var declarations to the last parsed function.

ParseNodePtr Parser::AddVarDeclNode(IdentPtr pid, ParseNodePtr pnodeFnc)

{

AnalysisAssert(pnodeFnc);

ParseNodePtr \*const ppnodeVarSave = m\_ppnodeVar;

m\_ppnodeVar = &pnodeFnc->sxFnc.pnodeVars;

while (\*m\_ppnodeVar != nullptr)

{

m\_ppnodeVar = &(\*m\_ppnodeVar)->sxVar.pnodeNext;

}

ParseNodePtr pnode = CreateVarDeclNode(pid, STUnknown, false, 0, /\* checkReDecl = \*/ false);

m\_ppnodeVar = ppnodeVarSave;

return pnode;

}

ParseNodePtr Parser::CreateVarDeclNode(IdentPtr pid, SymbolType symbolType, bool autoArgumentsObject, ParseNodePtr pnodeFnc, bool errorOnRedecl)

{

ParseNodePtr pnode = CreateDeclNode(knopVarDecl, pid, symbolType, errorOnRedecl);

// Append the variable to the end of the current variable list.

AssertMem(m\_ppnodeVar);

pnode->sxVar.pnodeNext = \*m\_ppnodeVar;

\*m\_ppnodeVar = pnode;

if (nullptr != pid)

{

// this is not a temp - make sure temps go after this node

AssertMem(pid);

m\_ppnodeVar = &pnode->sxVar.pnodeNext;

CheckPidIsValid(pid, autoArgumentsObject);

}

return pnode;

}

ParseNodePtr Parser::CreateBlockScopedDeclNode(IdentPtr pid, OpCode nodeType)

{

Assert(nodeType == knopConstDecl || nodeType == knopLetDecl);

ParseNodePtr pnode = CreateDeclNode(nodeType, pid, STVariable, true);

if (nullptr != pid)

{

AssertMem(pid);

pid->SetIsLetOrConst();

AddVarDeclToBlock(pnode);

CheckPidIsValid(pid);

}

return pnode;

}

void Parser::AddVarDeclToBlock(ParseNode \*pnode)

{

Assert(pnode->nop == knopConstDecl || pnode->nop == knopLetDecl);

// Maintain a combined list of let and const declarations to keep

// track of declaration order.

AssertMem(m\_currentBlockInfo->m\_ppnodeLex);

\*m\_currentBlockInfo->m\_ppnodeLex = pnode;

m\_currentBlockInfo->m\_ppnodeLex = &pnode->sxVar.pnodeNext;

pnode->sxVar.pnodeNext = nullptr;

}

void Parser::SetCurrentStatement(StmtNest \*stmt)

{

m\_pstmtCur = stmt;

}

template<bool buildAST>

ParseNodePtr Parser::StartParseBlockWithCapacity(PnodeBlockType blockType, ScopeType scopeType, int capacity)

{

Scope \*scope = nullptr;

// Block scopes are created lazily when we discover block-scoped content.

if (scopeType != ScopeType\_Unknown && scopeType != ScopeType\_Block)

{

scope = Anew(&m\_nodeAllocator, Scope, &m\_nodeAllocator, scopeType, PHASE\_OFF1(Js::ParserBindPhase), capacity);

PushScope(scope);

}

return StartParseBlockHelper<buildAST>(blockType, scope, nullptr, nullptr);

}

template<bool buildAST>

ParseNodePtr Parser::StartParseBlock(PnodeBlockType blockType, ScopeType scopeType, ParseNodePtr pnodeLabel, LabelId\* pLabelId)

{

Scope \*scope = nullptr;

// Block scopes are created lazily when we discover block-scoped content.

if (scopeType != ScopeType\_Unknown && scopeType != ScopeType\_Block)

{

scope = Anew(&m\_nodeAllocator, Scope, &m\_nodeAllocator, scopeType);

PushScope(scope);

}

return StartParseBlockHelper<buildAST>(blockType, scope, pnodeLabel, pLabelId);

}

template<bool buildAST>

ParseNodePtr Parser::StartParseBlockHelper(PnodeBlockType blockType, Scope \*scope, ParseNodePtr pnodeLabel, LabelId\* pLabelId)

{

ParseNodePtr pnodeBlock = CreateBlockNode(blockType);

pnodeBlock->sxBlock.scope = scope;

BlockInfoStack \*newBlockInfo = PushBlockInfo(pnodeBlock);

PushStmt<buildAST>(&newBlockInfo->pstmt, pnodeBlock, knopBlock, pnodeLabel, pLabelId);

return pnodeBlock;

}

void Parser::PushScope(Scope \*scope)

{

Assert(scope);

scope->SetEnclosingScope(m\_currentScope);

m\_currentScope = scope;

}

void Parser::PopScope(Scope \*scope)

{

Assert(scope == m\_currentScope);

m\_currentScope = scope->GetEnclosingScope();

scope->SetEnclosingScope(nullptr);

}

void Parser::PushFuncBlockScope(ParseNodePtr pnodeBlock, ParseNodePtr \*\*ppnodeScopeSave, ParseNodePtr \*\*ppnodeExprScopeSave)

{

bool blockHasScope = m\_scriptContext->GetConfig()->IsBlockScopeEnabled();

if (blockHasScope)

{

// Maintain the scope tree.

pnodeBlock->sxBlock.pnodeScopes = nullptr;

pnodeBlock->sxBlock.pnodeNext = nullptr;

// Insert this block into the active list of scopes (m\_ppnodeExprScope or m\_ppnodeScope).

// Save the current block's "next" pointer as the new endpoint of that list.

if (m\_ppnodeExprScope)

{

\*ppnodeScopeSave = m\_ppnodeScope;

Assert(\*m\_ppnodeExprScope == nullptr);

\*m\_ppnodeExprScope = pnodeBlock;

\*ppnodeExprScopeSave = &pnodeBlock->sxBlock.pnodeNext;

}

else

{

Assert(m\_ppnodeScope);

Assert(\*m\_ppnodeScope == nullptr);

\*m\_ppnodeScope = pnodeBlock;

\*ppnodeScopeSave = &pnodeBlock->sxBlock.pnodeNext;

\*ppnodeExprScopeSave = m\_ppnodeExprScope;

}

// Advance the global scope list pointer to the new block's child list.

m\_ppnodeScope = &pnodeBlock->sxBlock.pnodeScopes;

// Set m\_ppnodeExprScope to NULL to make that list inactive.

m\_ppnodeExprScope = nullptr;

}

}

void Parser::PopFuncBlockScope(ParseNodePtr \*ppnodeScopeSave, ParseNodePtr \*ppnodeExprScopeSave)

{

bool blockHasScope = m\_scriptContext->GetConfig()->IsBlockScopeEnabled();

if (blockHasScope)

{

Assert(m\_ppnodeExprScope == nullptr || \*m\_ppnodeExprScope == nullptr);

m\_ppnodeExprScope = ppnodeExprScopeSave;

AssertMem(m\_ppnodeScope);

Assert(nullptr == \*m\_ppnodeScope);

m\_ppnodeScope = ppnodeScopeSave;

}

}

template<bool buildAST>

ParseNodePtr Parser::ParseBlock(ParseNodePtr pnodeLabel, LabelId\* pLabelId)

{

StmtNest stmt;

ParseNodePtr pnodeBlock = nullptr;

ParseNodePtr \*ppnodeScopeSave = nullptr;

ParseNodePtr \*ppnodeExprScopeSave = nullptr;

if (buildAST || BindDeferredPidRefs())

{

pnodeBlock = StartParseBlock<buildAST>(PnodeBlockType::Regular, ScopeType\_Block, pnodeLabel, pLabelId);

}

else

{

PushStmt<buildAST>(&stmt, nullptr, knopBlock, pnodeLabel, pLabelId);

}

ChkCurTok(tkLCurly, ERRnoLcurly);

ParseNodePtr \* ppnodeList = nullptr;

if (buildAST)

{

PushFuncBlockScope(pnodeBlock, &ppnodeScopeSave, &ppnodeExprScopeSave);

ppnodeList = &pnodeBlock->sxBlock.pnodeStmt;

}

ParseStmtList<buildAST>(ppnodeList);

if (buildAST)

{

PopFuncBlockScope(ppnodeScopeSave, ppnodeExprScopeSave);

}

if (buildAST || BindDeferredPidRefs())

{

FinishParseBlock(pnodeBlock);

}

else

{

PopStmt(&stmt);

}

ChkCurTok(tkRCurly, ERRnoRcurly);

return pnodeBlock;

}

void Parser::FinishParseBlock(ParseNode \*pnodeBlock, bool needScanRCurly)

{

Assert(m\_currentBlockInfo != nullptr && pnodeBlock == m\_currentBlockInfo->pnodeBlock);

if (needScanRCurly)

{

// Only update the ichLim if we were expecting an RCurly. If there is an

// expression body without a necessary RCurly, the correct ichLim will

// have been set already.

pnodeBlock->ichLim = m\_pscan->IchLimTok();

}

BindPidRefs<false>(GetCurrentBlockInfo(), m\_nextBlockId - 1);

PopStmt(&m\_currentBlockInfo->pstmt);

PopBlockInfo();

Scope \*scope = pnodeBlock->sxBlock.scope;

if (scope)

{

PopScope(scope);

}

}

void Parser::FinishParseFncExprScope(ParseNodePtr pnodeFnc, ParseNodePtr pnodeFncExprScope)

{

int fncExprScopeId = pnodeFncExprScope->sxBlock.blockId;

ParseNodePtr pnodeName = pnodeFnc->sxFnc.pnodeName;

if (pnodeName)

{

Assert(pnodeName->nop == knopVarDecl);

BindPidRefsInScope(pnodeName->sxVar.pid, pnodeName->sxVar.sym, fncExprScopeId);

}

FinishParseBlock(pnodeFncExprScope);

}

template <const bool backgroundPidRef>

void Parser::BindPidRefs(BlockInfoStack \*blockInfo, uint maxBlockId)

{

// We need to bind all assignments in order to emit assignment to 'const' error

int blockId = blockInfo->pnodeBlock->sxBlock.blockId;

Scope \*scope = blockInfo->pnodeBlock->sxBlock.scope;

if (scope)

{

auto bindPidRefs = [blockId, maxBlockId, this](Symbol \*sym)

{

ParseNodePtr pnode = sym->GetDecl();

IdentPtr pid;

#if PROFILE\_DICTIONARY

int depth = 0;

#endif

Assert(pnode);

switch (pnode->nop)

{

case knopLetDecl:

case knopVarDecl:

pid = pnode->sxVar.pid;

if (backgroundPidRef)

{

pid = this->m\_pscan->m\_phtbl->FindExistingPid(pid->Psz(), pid->Cch(), pid->Hash(), nullptr, nullptr

#if PROFILE\_DICTIONARY

, depth

#endif

);

if (pid == nullptr)

{

break;

}

}

this->BindPidRefsInScope(pid, sym, blockId, maxBlockId);

break;

case knopConstDecl:

pid = pnode->sxVar.pid;

if (backgroundPidRef)

{

pid = this->m\_pscan->m\_phtbl->FindExistingPid(pid->Psz(), pid->Cch(), pid->Hash(), nullptr, nullptr

#if PROFILE\_DICTIONARY

, depth

#endif

);

if (pid == nullptr)

{

break;

}

}

this->BindConstPidRefsInScope(pid, sym, blockId, maxBlockId);

break;

case knopName:

pid = pnode->sxPid.pid;

if (backgroundPidRef)

{

pid = this->m\_pscan->m\_phtbl->FindExistingPid(pid->Psz(), pid->Cch(), pid->Hash(), nullptr, nullptr

#if PROFILE\_DICTIONARY

, depth

#endif

);

if (pid == nullptr)

{

break;

}

}

this->BindPidRefsInScope(pid, sym, blockId, maxBlockId);

break;

default:

Assert(0);

break;

}

};

scope->ForEachSymbol(bindPidRefs);

}

}

void Parser::BindPidRefsInScope(IdentPtr pid, Symbol \*sym, int blockId, uint maxBlockId)

{

this->BindPidRefsInScopeImpl<false>(pid, sym, blockId, maxBlockId);

}

void Parser::BindConstPidRefsInScope(IdentPtr pid, Symbol \*sym, int blockId, uint maxBlockId)

{

this->BindPidRefsInScopeImpl<true>(pid, sym, blockId, maxBlockId);

}

template<const bool isConstBinding>

void Parser::BindPidRefsInScopeImpl(IdentPtr pid, Symbol \*sym, int blockId, uint maxBlockId)

{

PidRefStack \*ref, \*nextRef, \*lastRef = nullptr;

Assert(sym);

for (ref = pid->GetTopRef(); ref && ref->GetScopeId() >= blockId; ref = nextRef)

{

// Fix up sym\* on PID ref.

Assert(!ref->GetSym() || ref->GetSym() == sym);

nextRef = ref->prev;

Assert(ref->GetScopeId() >= 0);

if ((uint)ref->GetScopeId() > maxBlockId)

{

lastRef = ref;

continue;

}

ref->SetSym(sym);

if (isConstBinding && ref->IsAssignment() && !ref->IsDynamicBinding())

{

if (pid->GetTopIchMin() < this->m\_asgToConst.GetIchMin())

{

this->m\_asgToConst.Set(pid->GetTopIchMin(), pid->GetTopIchLim());

}

}

this->RemovePrevPidRef(pid, lastRef);

if (ref->IsAssignment())

{

sym->PromoteAssignmentState();

}

if (ref->GetScopeId() == blockId)

{

break;

}

}

}

void Parser::PopStmt(StmtNest \*pStmt)

{

Assert(pStmt == m\_pstmtCur);

SetCurrentStatement(m\_pstmtCur->pstmtOuter);

}

BlockInfoStack \*Parser::PushBlockInfo(ParseNodePtr pnodeBlock)

{

BlockInfoStack \*newBlockInfo = (BlockInfoStack \*)m\_nodeAllocator.Alloc(sizeof(BlockInfoStack));

Assert(nullptr != newBlockInfo);

newBlockInfo->pnodeBlock = pnodeBlock;

newBlockInfo->pBlockInfoOuter = m\_currentBlockInfo;

newBlockInfo->m\_ppnodeLex = &pnodeBlock->sxBlock.pnodeLexVars;

if (pnodeBlock->sxBlock.blockType != PnodeBlockType::Regular)

{

newBlockInfo->pBlockInfoFunction = newBlockInfo;

}

else

{

Assert(m\_currentBlockInfo);

newBlockInfo->pBlockInfoFunction = m\_currentBlockInfo->pBlockInfoFunction;

}

m\_currentBlockInfo = newBlockInfo;

return newBlockInfo;

}

void Parser::PopBlockInfo()

{

Assert(m\_currentBlockInfo);

PopDynamicBlock();

m\_currentBlockInfo = m\_currentBlockInfo->pBlockInfoOuter;

}

void Parser::PushDynamicBlock()

{

if (!m\_scriptContext->GetConfig()->IsLetAndConstEnabled())

{

// Shortcut: we only need to track dynamically-bound blocks for const reassignment.

return;

}

Assert(GetCurrentBlock());

int blockId = GetCurrentBlock()->sxBlock.blockId;

if (m\_currentDynamicBlock && m\_currentDynamicBlock->id == blockId)

{

return;

}

BlockIdsStack \*info = (BlockIdsStack \*)m\_nodeAllocator.Alloc(sizeof(BlockIdsStack));

if (nullptr == info)

{

Error(ERRnoMemory);

}

info->id = blockId;

info->prev = m\_currentDynamicBlock;

m\_currentDynamicBlock = info;

}

void Parser::PopDynamicBlock()

{

int blockId = GetCurrentDynamicBlockId();

if (GetCurrentBlock()->sxBlock.blockId != blockId || blockId == -1)

{

return;

}

Assert(m\_currentDynamicBlock);

AssertMsg(m\_scriptContext->GetConfig()->IsLetAndConstEnabled(), "Should only do this if let/const is enabled since only needed for const reassignment error checking");

for (BlockInfoStack \*blockInfo = m\_currentBlockInfo; blockInfo; blockInfo = blockInfo->pBlockInfoOuter)

{

for (ParseNodePtr pnodeDecl = blockInfo->pnodeBlock->sxBlock.pnodeLexVars;

pnodeDecl;

pnodeDecl = pnodeDecl->sxVar.pnodeNext)

{

this->SetPidRefsInScopeDynamic(pnodeDecl->sxVar.pid, blockId);

}

}

m\_currentDynamicBlock = m\_currentDynamicBlock->prev;

}

int Parser::GetCurrentDynamicBlockId() const

{

return m\_currentDynamicBlock ? m\_currentDynamicBlock->id : -1;

}

ParseNode \*Parser::GetCurrentFunctionNode()

{

if (m\_currentNodeDeferredFunc != nullptr)

{

return m\_currentNodeDeferredFunc;

}

else if (m\_currentNodeFunc != nullptr)

{

return m\_currentNodeFunc;

}

else

{

AssertMsg(GetFunctionBlock()->sxBlock.blockType == PnodeBlockType::Global,

"Most likely we are trying to find a syntax error, related to 'let' or 'const' in deferred parsing mode with disabled support of 'let' and 'const'");

return m\_currentNodeProg;

}

}

ParseNode \*Parser::GetCurrentNonLamdaFunctionNode()

{

if (m\_currentNodeNonLambdaDeferredFunc != nullptr)

{

return m\_currentNodeNonLambdaDeferredFunc;

}

return m\_currentNodeNonLambdaFunc;

}

void Parser::RegisterRegexPattern(UnifiedRegex::RegexPattern \*const regexPattern)

{

Assert(regexPattern);

// ensure a no-throw add behavior here, to catch out of memory exceptions, using the guest arena allocator

if (!m\_registeredRegexPatterns.PrependNoThrow(m\_scriptContext->GetGuestArena(), regexPattern))

{

Parser::Error(ERRnoMemory);

}

}

void Parser::AddToNodeListEscapedUse(ParseNode \*\* ppnodeList, ParseNode \*\*\* pppnodeLast,

ParseNode \* pnodeAdd)

{

AddToNodeList(ppnodeList, pppnodeLast, pnodeAdd);

pnodeAdd->SetIsInList();

}

void Parser::AddToNodeList(ParseNode \*\* ppnodeList, ParseNode \*\*\* pppnodeLast,

ParseNode \* pnodeAdd)

{

Assert(!this->m\_deferringAST);

if (nullptr == \*pppnodeLast)

{

// should be an empty list

Assert(nullptr == \*ppnodeList);

\*ppnodeList = pnodeAdd;

\*pppnodeLast = ppnodeList;

}

else

{

//

AssertNodeMem(\*ppnodeList);

AssertNodeMem(\*\*pppnodeLast);

ParseNode \*pnodeT = CreateBinNode(knopList, \*\*pppnodeLast, pnodeAdd);

\*\*pppnodeLast = pnodeT;

\*pppnodeLast = &pnodeT->sxBin.pnode2;

}

}

// Check reference to "arguments" that indicates the object may escape.

void Parser::CheckArguments(ParseNodePtr pnode)

{

if (m\_currentNodeFunc && this->NodeIsIdent(pnode, wellKnownPropertyPids.arguments))

{

m\_currentNodeFunc->sxFnc.SetHasHeapArguments();

}

}

// Check use of "arguments" that requires instantiation of the object.

void Parser::CheckArgumentsUse(IdentPtr pid, ParseNodePtr pnodeFnc)

{

if (pid == wellKnownPropertyPids.arguments)

{

if (pnodeFnc != nullptr)

{

pnodeFnc->sxFnc.SetUsesArguments(TRUE);

}

else

{

m\_UsesArgumentsAtGlobal = true;

}

}

}

void Parser::CheckStrictModeEvalArgumentsUsage(IdentPtr pid, ParseNodePtr pnode)

{

if (pid != nullptr)

{

// In strict mode, 'eval' / 'arguments' cannot be assigned to.

if ( pid == wellKnownPropertyPids.eval)

{

Error(ERREvalUsage, pnode);

}

if (pid == wellKnownPropertyPids.arguments)

{

Error(ERRArgsUsage, pnode);

}

}

}

void Parser::CheckStrictModeFncDeclNotSourceElement(const bool isSourceElement, const BOOL isDeclaration)

{

// In strict mode, only a SourceElement can expand to a FunctionDeclaration; a Statement cannot. That means a function

// declaration may only appear as a top-level statement in a program or function body, and otherwise may not be nested

// inside another statement or block.

//

// The only difference between a SourceElement and a Statement is that a SourceElement can include a FunctionDeclaration, so

// we just use ParseStmtList and ParseStatement and pass in a flag indicating whether the statements are source elements.

Assert(!(isSourceElement && !isDeclaration));

if(IsStrictMode() && !isSourceElement && isDeclaration &&

!this->GetScriptContext()->GetConfig()->IsBlockScopeEnabled())

{

Error(ERRFncDeclNotSourceElement);

}

}

void Parser::ReduceDeferredScriptLength(size\_t chars)

{

// If we're in deferred mode, subtract the given char count from the total length,

// and see if this puts us under the deferral threshold.

if (m\_grfscr & fscrDeferFncParse)

{

if (m\_length > chars)

{

m\_length -= chars;

}

else

{

m\_length = 0;

}

if (m\_length < Parser::GetDeferralThreshold(this->m\_sourceContextInfo->IsSourceProfileLoaded()))

{

// Stop deferring.

m\_grfscr &= ~fscrDeferFncParse;

m\_stoppedDeferredParse = TRUE;

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Look for an existing label with the given name.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

BOOL Parser::PnodeLabelNoAST(IdentToken\* pToken, LabelId\* pLabelIdList)

{

StmtNest\* pStmt;

LabelId\* pLabelId;

// Look in the label stack.

for (pStmt = m\_pstmtCur; pStmt != nullptr; pStmt = pStmt->pstmtOuter)

{

for (pLabelId = pStmt->pLabelId; pLabelId != nullptr; pLabelId = pLabelId->next)

{

if (pLabelId->pid == pToken->pid)

return TRUE;

}

}

// Also look in the pnodeLabels list.

for (pLabelId = pLabelIdList; pLabelId != nullptr; pLabelId = pLabelId->next)

{

if (pLabelId->pid == pToken->pid)

return TRUE;

}

return FALSE;

}

void Parser::EnsureStackAvailable()

{

if (!m\_scriptContext->GetThreadContext()->IsStackAvailable(Js::Constants::MinStackCompile))

{

Error(ERRnoMemory);

}

}

void Parser::ThrowNewTargetSyntaxErrForGlobalScope()

{

//TODO: (falotfi) we need reliably distinguish eval in global scope vs in a function

// The rule for this syntax error is any time new.target is called at global scope

// we are excluding new.target in eval at global scope for now.

if(GetCurrentNonLamdaFunctionNode() == nullptr && (this->m\_grfscr & fscrEvalCode) == 0)

{

Error(ERRInvalidNewTarget);

}

}

template<bool buildAST>

ParseNodePtr Parser::ParseMetaProperty(tokens metaParentKeyword, charcount\_t ichMin, \_Out\_opt\_ BOOL\* pfCanAssign)

{

AssertMsg(metaParentKeyword == tkNEW, "Only supported for tkNEW parent keywords");

AssertMsg(this->m\_token.tk == tkDot, "We must be currently sitting on the dot after the parent keyword");

m\_pscan->Scan();

if (this->m\_token.tk == tkID && this->m\_token.GetIdentifier(m\_phtbl) == this->GetTargetPid())

{

ThrowNewTargetSyntaxErrForGlobalScope();

if (pfCanAssign)

{

\*pfCanAssign = FALSE;

}

if (buildAST)

{

return CreateNodeWithScanner<knopNewTarget>(ichMin);

}

}

else

{

Error(ERRsyntax);

}

return nullptr;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Parse an expression term.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

template<bool buildAST>

ParseNodePtr Parser::ParseTerm(BOOL fAllowCall,

LPCOLESTR pNameHint,

ulong \*pHintLength,

ulong \*pShortNameOffset,

\_Inout\_opt\_ IdentToken\* pToken/\*= nullptr\*/,

bool fUnaryOrParen /\*= false\*/,

\_Out\_opt\_ BOOL\* pfCanAssign /\* = nullptr\*/,

\_Inout\_opt\_ BOOL\* pfLikelyPattern /\* = nullptr\*/)

{

ParseNodePtr pnode = nullptr;

charcount\_t ichMin = 0;

size\_t iecpMin = 0;

size\_t iuMin;

IdentToken term;

BOOL fInNew = FALSE;

BOOL fCanAssign = TRUE;

bool isAsyncExpr = false;

bool isLambdaExpr = false;

Assert(pToken == nullptr || pToken->tk == tkNone); // Must be empty initially

if (this->IsBackgroundParser())

{

PROBE\_STACK\_NO\_DISPOSE(m\_scriptContext, Js::Constants::MinStackParseOneTerm);

}

else

{

PROBE\_STACK(m\_scriptContext, Js::Constants::MinStackParseOneTerm);

}

switch (m\_token.tk)

{

case tkID:

{

PidRefStack \*ref = nullptr;

IdentPtr pid = m\_token.GetIdentifier(m\_phtbl);

charcount\_t ichLim = m\_pscan->IchLimTok();

size\_t iecpLim = m\_pscan->IecpLimTok();

ichMin = m\_pscan->IchMinTok();

iecpMin = m\_pscan->IecpMinTok();

m\_pscan->Scan();

// We search an Async expression (a function declaration or a async lambda expression)

if (pid == wellKnownPropertyPids.async && m\_scriptContext->GetConfig()->IsES7AsyncAndAwaitEnabled())

{

if (m\_token.tk == tkFUNCTION)

{

isAsyncExpr = true;

goto LFunction;

}

else if (m\_token.tk == tkID)

{

isLambdaExpr = true;

isAsyncExpr = true;

goto LFunction;

}

}

if (buildAST || BindDeferredPidRefs())

{

ref = this->PushPidRef(pid);

}

if (buildAST)

{

pnode = CreateNameNode(pid);

pnode->ichMin = ichMin;

pnode->ichLim = ichLim;

pnode->sxPid.SetSymRef(ref);

CheckArgumentsUse(pid, m\_currentNodeFunc);

}

else

{

// Remember the identifier start and end in case it turns out to be a statement label.

term.tk = tkID;

term.pid = pid; // Record the identifier for detection of eval

term.ichMin = static\_cast<charcount\_t>(iecpMin);

term.ichLim = static\_cast<charcount\_t>(iecpLim);

}

break;

}

case tkTHIS:

if (buildAST)

{

pnode = CreateNodeWithScanner<knopThis>();

}

fCanAssign = FALSE;

m\_pscan->Scan();

break;

case tkLParen:

ichMin = m\_pscan->IchMinTok();

iuMin = m\_pscan->IecpMinTok();

m\_pscan->Scan();

if (m\_token.tk == tkRParen)

{

// Empty parens can only be legal as an empty parameter list to a lambda declaration.

// We're in a lambda if the next token is =>.

fAllowCall = FALSE;

m\_pscan->Scan();

// If the token after the right paren is not => or if there was a newline between () and => this is a syntax error

if (!m\_doingFastScan && (m\_token.tk != tkDArrow || m\_pscan->FHadNewLine()))

{

Error(ERRsyntax);

}

if (buildAST)

{

pnode = CreateNodeWithScanner<knopEmpty>();

}

break;

}

this->m\_parenDepth++;

pnode = ParseExpr<buildAST>(koplNo, &fCanAssign, TRUE, FALSE, nullptr, nullptr /\*nameLength\*/, nullptr /\*pShortNameOffset\*/, &term, true);

this->m\_parenDepth--;

ChkCurTok(tkRParen, ERRnoRparen);

// Emit a deferred ... error if one was parsed.

if (m\_deferEllipsisError && m\_token.tk != tkDArrow)

{

m\_pscan->SeekTo(m\_EllipsisErrLoc);

Error(ERRInvalidSpreadUse);

}

else

{

m\_deferEllipsisError = false;

}

break;

case tkIntCon:

if (IsStrictMode() && m\_pscan->IsOctOrLeadingZeroOnLastTKNumber())

{

Error(ERRES5NoOctal);

}

if (buildAST)

{

pnode = CreateIntNodeWithScanner(m\_token.GetLong());

}

fCanAssign = FALSE;

m\_pscan->Scan();

break;

case tkFltCon:

if (IsStrictMode() && m\_pscan->IsOctOrLeadingZeroOnLastTKNumber())

{

Error(ERRES5NoOctal);

}

if (buildAST)

{

pnode = CreateNodeWithScanner<knopFlt>();

pnode->sxFlt.dbl = m\_token.GetDouble();

pnode->sxFlt.maybeInt = m\_token.GetDoubleMayBeInt();

}

fCanAssign = FALSE;

m\_pscan->Scan();

break;

case tkStrCon:

if (IsStrictMode() && m\_pscan->IsOctOrLeadingZeroOnLastTKNumber())

{

Error(ERRES5NoOctal);

}

if (buildAST)

{

pnode = CreateStrNodeWithScanner(m\_token.GetStr());

}

else

{

// Subtract the string literal length from the total char count for the purpose

// of deciding whether to defer parsing and byte code generation.

this->ReduceDeferredScriptLength(m\_pscan->IchLimTok() - m\_pscan->IchMinTok());

}

fCanAssign = FALSE;

m\_pscan->Scan();

break;

case tkTRUE:

if (buildAST)

{

pnode = CreateNodeWithScanner<knopTrue>();

}

fCanAssign = FALSE;

m\_pscan->Scan();

break;

case tkFALSE:

if (buildAST)

{

pnode = CreateNodeWithScanner<knopFalse>();

}

fCanAssign = FALSE;

m\_pscan->Scan();

break;

case tkNULL:

if (buildAST)

{

pnode = CreateNodeWithScanner<knopNull>();

}

fCanAssign = FALSE;

m\_pscan->Scan();

break;

case tkDiv:

case tkAsgDiv:

pnode = ParseRegExp<buildAST>();

fCanAssign = FALSE;

m\_pscan->Scan();

break;

case tkNEW:

{

ichMin = m\_pscan->IchMinTok();

m\_pscan->Scan();

if (m\_token.tk == tkDot && m\_scriptContext->GetConfig()->IsES6ClassAndExtendsEnabled())

{

pnode = ParseMetaProperty<buildAST>(tkNEW, ichMin, &fCanAssign);

m\_pscan->Scan();

}

else

{

ParseNodePtr pnodeExpr = ParseTerm<buildAST>(FALSE, pNameHint, pHintLength, pShortNameOffset);

if (buildAST)

{

pnode = CreateCallNode(knopNew, pnodeExpr, nullptr);

pnode->ichMin = ichMin;

}

fInNew = TRUE;

fCanAssign = FALSE;

}

break;

}

case tkLBrack:

{

ichMin = m\_pscan->IchMinTok();

m\_pscan->Scan();

pnode = ParseArrayLiteral<buildAST>();

if (buildAST)

{

pnode->ichMin = ichMin;

pnode->ichLim = m\_pscan->IchLimTok();

}

if (this->m\_arrayDepth == 0)

{

Assert(m\_pscan->IchLimTok() - ichMin > m\_funcInArray);

this->ReduceDeferredScriptLength(m\_pscan->IchLimTok() - ichMin - this->m\_funcInArray);

this->m\_funcInArray = 0;

this->m\_funcInArrayDepth = 0;

}

ChkCurTok(tkRBrack, ERRnoRbrack);

if (!IsES6DestructuringEnabled())

{

fCanAssign = FALSE;

}

else if (pfLikelyPattern != nullptr && !IsPostFixOperators())

{

\*pfLikelyPattern = TRUE;

}

break;

}

case tkLCurly:

{

ichMin = m\_pscan->IchMinTok();

m\_pscan->ScanForcingPid();

ParseNodePtr pnodeMemberList = ParseMemberList<buildAST>(pNameHint, pHintLength);

if (buildAST)

{

pnode = CreateUniNode(knopObject, pnodeMemberList);

pnode->ichMin = ichMin;

pnode->ichLim = m\_pscan->IchLimTok();

}

ChkCurTok(tkRCurly, ERRnoRcurly);

if (!IsES6DestructuringEnabled())

{

fCanAssign = FALSE;

}

else if (pfLikelyPattern != nullptr && !IsPostFixOperators())

{

\*pfLikelyPattern = TRUE;

}

break;

}

case tkFUNCTION:

{

LFunction :

if (m\_grfscr & fscrDeferredFncExpression)

{

// The top-level deferred function body was defined by a function expression whose parsing was deferred. We are now

// parsing it, so unset the flag so that any nested functions are parsed normally. This flag is only applicable the

// first time we see it.

//

// Normally, deferred functions will be parsed in ParseStatement upon encountering the 'function' token. The first

// token of the source code of the function may not a 'function' token though, so we still need to reset this flag

// for the first function we parse. This can happen in compat modes, for instance, for a function expression enclosed

// in parentheses, where the legacy behavior was to include the parentheses in the function's source code.

m\_grfscr &= ~fscrDeferredFncExpression;

}

ushort flags = fFncNoFlgs;

if (isLambdaExpr)

{

flags |= fFncLambda;

}

if (isAsyncExpr)

{

flags |= fFncAsync;

}

pnode = ParseFncDecl<buildAST>(flags, pNameHint, false, false, true, fUnaryOrParen);

if (isAsyncExpr)

{

pnode->sxFnc.cbMin = iecpMin;

pnode->ichMin = ichMin;

}

fCanAssign = FALSE;

break;

}

case tkCLASS:

fAllowCall = FALSE;

if (m\_scriptContext->GetConfig()->IsES6ClassAndExtendsEnabled())

{

pnode = ParseClassDecl<buildAST>(FALSE, pNameHint, pHintLength, pShortNameOffset);

}

else

{

goto LUnknown;

}

fCanAssign = FALSE;

break;

case tkStrTmplBasic:

case tkStrTmplBegin:

Assert(m\_scriptContext->GetConfig()->IsES6StringTemplateEnabled());

pnode = ParseStringTemplateDecl<buildAST>(nullptr);

fCanAssign = FALSE;

break;

case tkSUPER:

if (m\_scriptContext->GetConfig()->IsES6ClassAndExtendsEnabled())

{

pnode = ParseSuper<buildAST>(pnode, !!fAllowCall);

}

else

{

goto LUnknown;

}

break;

case tkCASE:

{

if (!m\_doingFastScan)

{

goto LUnknown;

}

ParseNodePtr pnodeUnused;

pnode = ParseCase<buildAST>(&pnodeUnused);

break;

}

case tkELSE:

if (!m\_doingFastScan)

{

goto LUnknown;

}

m\_pscan->Scan();

ParseStatement<buildAST>();

break;

default:

LUnknown :

Error(ERRsyntax);

break;

}

pnode = ParsePostfixOperators<buildAST>(pnode, fAllowCall, fInNew, &fCanAssign, &term);

// Pass back identifier if requested

if (pToken && term.tk == tkID)

{

\*pToken = term;

}

if (pfCanAssign)

{

\*pfCanAssign = fCanAssign;

}

return pnode;

}

template <bool buildAST>

ParseNodePtr Parser::ParseRegExp()

{

ParseNodePtr pnode = nullptr;

if (buildAST || m\_doingFastScan)

{

m\_pscan->RescanRegExp();

BOOL saveDeferringAST = this->m\_deferringAST;

if (m\_doingFastScan)

{

this->m\_deferringAST = false;

}

pnode = CreateNodeWithScanner<knopRegExp>();

pnode->sxPid.regexPattern = m\_token.GetRegex();

if (m\_doingFastScan)

{

this->m\_deferringAST = saveDeferringAST;

this->AddFastScannedRegExpNode(pnode);

if (!buildAST)

{

pnode = nullptr;

}

}

#if ENABLE\_BACKGROUND\_PARSING

else if (this->IsBackgroundParser())

{

Assert(pnode->sxPid.regexPattern == nullptr);

this->AddBackgroundRegExpNode(pnode);

}

#endif

}

else

{

m\_pscan->RescanRegExpNoAST();

}

Assert(m\_token.tk == tkRegExp);

return pnode;

}

BOOL Parser::NodeIsEvalName(ParseNodePtr pnode)

{

//WOOB 1107758 Special case of indirect eval binds to local scope in standards mode

return pnode->nop == knopName && (pnode->sxPid.pid == wellKnownPropertyPids.eval);

}

BOOL Parser::NodeEqualsName(ParseNodePtr pnode, LPCOLESTR sz, ulong cch)

{

return pnode->nop == knopName &&

pnode->sxPid.pid->Cch() == cch &&

!wmemcmp(pnode->sxPid.pid->Psz(), sz, cch);

}

BOOL Parser::NodeIsIdent(ParseNodePtr pnode, IdentPtr pid)

{

for (;;)

{

switch (pnode->nop)

{

case knopName:

return (pnode->sxPid.pid == pid);

case knopComma:

pnode = pnode->sxBin.pnode2;

break;

default:

return FALSE;

}

}

}

template<bool buildAST>

ParseNodePtr Parser::ParsePostfixOperators(

ParseNodePtr pnode,

BOOL fAllowCall,

BOOL fInNew,

BOOL \*pfCanAssign,

\_Inout\_ IdentToken\* pToken)

{

uint16 count = 0;

bool callOfConstants = false;

for (;;)

{

uint16 spreadArgCount = 0;

switch (m\_token.tk)

{

case tkLParen:

{

if (fInNew)

{

ParseNodePtr pnodeArgs = ParseArgList<buildAST>(&callOfConstants, &spreadArgCount, &count);

if (buildAST)

{

Assert(pnode->nop == knopNew);

Assert(pnode->sxCall.pnodeArgs == nullptr);

pnode->sxCall.pnodeArgs = pnodeArgs;

pnode->sxCall.callOfConstants = callOfConstants;

pnode->sxCall.isApplyCall = false;

pnode->sxCall.isEvalCall = false;

pnode->sxCall.argCount = count;

pnode->sxCall.spreadArgCount = spreadArgCount;

pnode->ichLim = m\_pscan->IchLimTok();

}

else

{

pToken->tk = tkNone; // This is no longer an identifier

}

fInNew = FALSE;

}

else

{

bool fCallIsEval = false;

if (!fAllowCall)

{

return pnode;

}

ParseNodePtr pnodeArgs = ParseArgList<buildAST>(&callOfConstants, &spreadArgCount, &count);

// We used to un-defer a deferred function body here if it was called as part of the expression that declared it.

// We now detect this case up front in ParseFncDecl, which is cheaper and simpler.

if (buildAST)

{

pnode = CreateCallNode(knopCall, pnode, pnodeArgs);

Assert(pnode);

// Detect call to "eval" and record it on the function.

// Note: we used to leave it up to the byte code generator to detect eval calls

// at global scope, but now it relies on the flag the parser sets, so set it here.

if (count > 0 && this->NodeIsEvalName(pnode->sxCall.pnodeTarget))

{

this->MarkEvalCaller();

fCallIsEval = true;

}

pnode->sxCall.callOfConstants = callOfConstants;

pnode->sxCall.spreadArgCount = spreadArgCount;

pnode->sxCall.isApplyCall = false;

pnode->sxCall.isEvalCall = fCallIsEval;

pnode->sxCall.argCount = count;

pnode->ichLim = m\_pscan->IchLimTok();

}

else

{

if (pToken->tk == tkID && pToken->pid == wellKnownPropertyPids.eval) // Detect eval

{

this->MarkEvalCaller();

}

pToken->tk = tkNone; // This is no longer an identifier

}

}

ChkCurTok(tkRParen, ERRnoRparen);

if (pfCanAssign)

{

\*pfCanAssign = FALSE;

}

break;

}

case tkLBrack:

{

m\_pscan->Scan();

ParseNodePtr pnodeExpr = ParseExpr<buildAST>();

if (buildAST)

{

pnode = CreateBinNode(knopIndex, pnode, pnodeExpr);

pnode->ichLim = m\_pscan->IchLimTok();

}

else

{

pToken->tk = tkNone; // This is no longer an identifier

}

ChkCurTok(tkRBrack, ERRnoRbrack);

if (pfCanAssign)

{

\*pfCanAssign = TRUE;

}

if (!buildAST)

{

break;

}

bool shouldConvertToDot = false;

if (pnode->sxBin.pnode2->nop == knopStr)

{

// if the string is empty or contains escape character, we will not convert them to dot node

shouldConvertToDot = pnode->sxBin.pnode2->sxPid.pid->Cch() > 0 && !m\_pscan->IsEscapeOnLastTkStrCon();

}

if (shouldConvertToDot)

{

LPCOLESTR str = pnode->sxBin.pnode2->sxPid.pid->Psz();

// See if we can convert o["p"] into o.p and o["0"] into o[0] since they're equivalent and the latter forms

// are faster

uint32 uintValue;

if(Js::JavascriptOperators::TryConvertToUInt32(

str,

pnode->sxBin.pnode2->sxPid.pid->Cch(),

&uintValue) &&

!Js::TaggedInt::IsOverflow(uintValue)) // the optimization is not very useful if the number can't be represented as an TaggedInt

{

// No need to verify that uintValue != JavascriptArray::InvalidIndex since all nonnegative TaggedInts are valid indexes

auto intNode = CreateIntNodeWithScanner(uintValue); // implicit conversion from uint32 to long

pnode->sxBin.pnode2 = intNode;

}

// Field optimization (see GlobOpt::KillLiveElems) checks for value being a Number,

// and since NaN/Infinity is a number it won't kill o.NaN/o.Infinity which would cause a problem

// if we decide to hoist o.NaN/o.Infinity.

// We need to keep o["NaN"] and o["+/-Infinity"] as array element access (we don't hoist that but we may hoist field access),

// so no matter if it's killed by o[x] inside a loop, we make sure that we never hoist these.

// We need to follow same logic for strings that convert to a floating point number.

else

{

bool doConvertToProperty = false; // Convert a["x"] -> a.x.

if (!Parser::IsNaNOrInfinityLiteral<true>(str))

{

const OLECHAR\* terminalChar;

double dbl = Js::NumberUtilities::StrToDbl(str, &terminalChar, m\_scriptContext);

bool convertsToFloat = !Js::NumberUtilities::IsNan(dbl);

doConvertToProperty = !convertsToFloat;

}

if (doConvertToProperty)

{

pnode->sxBin.pnode2->nop = knopName;

pnode->nop = knopDot;

pnode->grfpn |= PNodeFlags::fpnIndexOperator;

}

}

}

}

break;

case tkDot:

{

ParseNodePtr name = nullptr;

OpCode opCode = knopDot;

m\_pscan->Scan();

if (!m\_token.IsIdentifier())

{

//allow reserved words in ES5 mode

if (!(m\_token.IsReservedWord()))

{

IdentifierExpectedError(m\_token);

}

}

// Note: see comment above about field optimization WRT NaN/Infinity/-Infinity.

// Convert a.Nan, a.Infinity into a["NaN"], a["Infinity"].

// We don't care about -Infinity case here because x.-Infinity is invalid in JavaScript.

// Both NaN and Infinity are identifiers.

else if (buildAST && Parser::IsNaNOrInfinityLiteral<false>(m\_token.GetIdentifier(m\_phtbl)->Psz()))

{

opCode = knopIndex;

}

if (buildAST)

{

if (opCode == knopDot)

{

name = CreateNameNode(m\_token.GetIdentifier(m\_phtbl));

}

else

{

Assert(opCode == knopIndex);

name = CreateStrNodeWithScanner(m\_token.GetIdentifier(m\_phtbl));

}

pnode = CreateBinNode(opCode, pnode, name);

}

else

{

pToken->tk = tkNone;

}

if (pfCanAssign)

{

\*pfCanAssign = TRUE;

}

m\_pscan->Scan();

break;

}

case tkStrTmplBasic:

case tkStrTmplBegin:

{

Assert(m\_scriptContext->GetConfig()->IsES6StringTemplateEnabled());

ParseNode\* templateNode = ParseStringTemplateDecl<buildAST>(pnode);

if (!buildAST)

{

pToken->tk = tkNone; // This is no longer an identifier

}

pnode = templateNode;

if (pfCanAssign)

{

\*pfCanAssign = FALSE;

}

break;

}

default:

return pnode;

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Look for an existing label with the given name.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

ParseNodePtr Parser::PnodeLabel(IdentPtr pid, ParseNodePtr pnodeLabels)

{

AssertMem(pid);

AssertNodeMemN(pnodeLabels);

StmtNest \*pstmt;

ParseNodePtr pnodeT;

// Look in the statement stack.

for (pstmt = m\_pstmtCur; nullptr != pstmt; pstmt = pstmt->pstmtOuter)

{

AssertNodeMem(pstmt->pnodeStmt);

AssertNodeMemN(pstmt->pnodeLab);

for (pnodeT = pstmt->pnodeLab; nullptr != pnodeT;

pnodeT = pnodeT->sxLabel.pnodeNext)

{

Assert(knopLabel == pnodeT->nop);

if (pid == pnodeT->sxLabel.pid)

return pnodeT;

}

}

// Also look in the pnodeLabels list.

for (pnodeT = pnodeLabels; nullptr != pnodeT;

pnodeT = pnodeT->sxLabel.pnodeNext)

{

Assert(knopLabel == pnodeT->nop);

if (pid == pnodeT->sxLabel.pid)

return pnodeT;

}

return nullptr;

}

// Currently only ints and floats are treated as constants in function call

// TODO: Check if we need for other constants as well

BOOL Parser::IsConstantInFunctionCall(ParseNodePtr pnode)

{

if (pnode->nop == knopInt && !Js::TaggedInt::IsOverflow(pnode->sxInt.lw))

{

return TRUE;

}

if (pnode->nop == knopFlt)

{

return TRUE;

}

return FALSE;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Parse a list of arguments.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

template<bool buildAST>

ParseNodePtr Parser::ParseArgList( bool \*pCallOfConstants, uint16 \*pSpreadArgCount, uint16 \* pCount)

{

ParseNodePtr pnodeArg;

ParseNodePtr pnodeList = nullptr;

ParseNodePtr \*lastNodeRef = nullptr;

// Check for an empty list

Assert(m\_token.tk == tkLParen);

if (m\_pscan->Scan() == tkRParen)

{

return nullptr;

}

\*pCallOfConstants = true;

\*pSpreadArgCount = 0;

int count=0;

while (true)

{

// the count of arguments has to fit in an unsigned short

if (count > 0xffffU)

Error(ERRnoMemory);

// Allow spread in argument lists.

pnodeArg = ParseExpr<buildAST>(koplCma, nullptr, TRUE, /\* fAllowEllipsis \*/TRUE);

if (buildAST)

{

this->CheckArguments(pnodeArg);

if (\*pCallOfConstants && !IsConstantInFunctionCall(pnodeArg))

{

\*pCallOfConstants = false;

}

if (pnodeArg->nop == knopEllipsis)

{

(\*pSpreadArgCount)++;

}

++count;

AddToNodeListEscapedUse(&pnodeList, &lastNodeRef, pnodeArg);

}

if (m\_token.tk != tkComma)

{

break;

}

m\_pscan->Scan();

if (m\_token.tk == tkRParen && m\_scriptContext->GetConfig()->IsES7TrailingCommaEnabled())

{

break;

}

}

if (pSpreadArgCount!=nullptr && (\*pSpreadArgCount) > 0){

CHAKRATEL\_LANGSTATS\_INC\_LANGFEATURECOUNT(SpreadFeatureCount, m\_scriptContext);

}

if (buildAST)

{

\*pCount = (uint16)count;

AssertMem(lastNodeRef);

AssertNodeMem(\*lastNodeRef);

pnodeList->ichLim = (\*lastNodeRef)->ichLim;

}

return pnodeList;

}

// Currently only ints are treated as constants in ArrayLiterals

BOOL Parser::IsConstantInArrayLiteral(ParseNodePtr pnode)

{

if (pnode->nop == knopInt && !Js::TaggedInt::IsOverflow(pnode->sxInt.lw))

{

return TRUE;

}

return FALSE;

}

template<bool buildAST>

ParseNodePtr Parser::ParseArrayLiteral()

{

ParseNodePtr pnode = nullptr;

bool arrayOfTaggedInts = false;

bool arrayOfInts = false;

bool arrayOfNumbers = false;

bool hasMissingValues = false;

uint count = 0;

uint spreadCount = 0;

ParseNodePtr pnode1 = ParseArrayList<buildAST>(&arrayOfTaggedInts, &arrayOfInts, &arrayOfNumbers, &hasMissingValues, &count, &spreadCount);

if (buildAST)

{

pnode = CreateNodeWithScanner<knopArray>();

pnode->sxArrLit.pnode1 = pnode1;

pnode->sxArrLit.arrayOfTaggedInts = arrayOfTaggedInts;

pnode->sxArrLit.arrayOfInts = arrayOfInts;

pnode->sxArrLit.arrayOfNumbers = arrayOfNumbers;

pnode->sxArrLit.hasMissingValues = hasMissingValues;

pnode->sxArrLit.count = count;

pnode->sxArrLit.spreadCount = spreadCount;

if (pnode->sxArrLit.pnode1)

{

this->CheckArguments(pnode->sxArrLit.pnode1);

}

}

return pnode;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Create a ArrayLiteral node

Parse a list of array elements. [ a, b, , c, ]

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

template<bool buildAST>

ParseNodePtr Parser::ParseArrayList(bool \*pArrayOfTaggedInts, bool \*pArrayOfInts, bool \*pArrayOfNumbers, bool \*pHasMissingValues, uint \*count, uint \*spreadCount)

{

ParseNodePtr pnodeArg = nullptr;

ParseNodePtr pnodeList = nullptr;

ParseNodePtr \*lastNodeRef = nullptr;

\*count = 0;

// Check for an empty list

if (tkRBrack == m\_token.tk)

{

return nullptr;

}

this->m\_arrayDepth++;

bool arrayOfTaggedInts = buildAST;

bool arrayOfInts = buildAST;

bool arrayOfNumbers = buildAST;

bool arrayOfVarInts = false;

bool hasMissingValues = false;

for (;;)

{

(\*count)++;

if (tkComma == m\_token.tk || tkRBrack == m\_token.tk)

{

hasMissingValues = true;

arrayOfTaggedInts = false;

arrayOfInts = false;

arrayOfNumbers = false;

if (buildAST)

{

pnodeArg = CreateNodeWithScanner<knopEmpty>();

}

}

else

{

// Allow Spread in array literals.

pnodeArg = ParseExpr<buildAST>(koplCma, nullptr, TRUE, /\* fAllowEllipsis \*/ TRUE);

if (buildAST)

{

if (pnodeArg->nop == knopEllipsis)

{

(\*spreadCount)++;

}

this->CheckArguments(pnodeArg);

}

}

#if DEBUG

if(m\_grfscr & fscrEnforceJSON && !IsJSONValid(pnodeArg))

{

Error(ERRsyntax);

}

#endif

if (buildAST)

{

if (arrayOfNumbers)

{

if (pnodeArg->nop != knopInt)

{

arrayOfTaggedInts = false;

if (pnodeArg->nop != knopFlt)

{

// Not an array of constants.

arrayOfInts = false;

arrayOfNumbers = false;

}

else if (arrayOfInts && Js::JavascriptNumber::IsInt32OrUInt32(pnodeArg->sxFlt.dbl) && (!Js::JavascriptNumber::IsInt32(pnodeArg->sxFlt.dbl) || pnodeArg->sxFlt.dbl == -2147483648.0))

{

// We've seen nothing but ints, and this is a uint32 but not an int32.

// Unless we see an actual float at some point, we want an array of vars

// so we can work with tagged ints.

arrayOfVarInts = true;

}

else

{

// Not an int array, but it may still be a float array.

arrayOfInts = false;

}

}

else

{

if (Js::SparseArraySegment<int32>::IsMissingItem((int32\*)&pnodeArg->sxInt.lw))

{

arrayOfInts = false;

}

if (Js::TaggedInt::IsOverflow(pnodeArg->sxInt.lw))

{

arrayOfTaggedInts = false;

}

}

}

AddToNodeListEscapedUse(&pnodeList, &lastNodeRef, pnodeArg);

}

if (tkComma != m\_token.tk)

{

break;

}

m\_pscan->Scan();

if (tkRBrack == m\_token.tk)

{

break;

}

}

if (spreadCount != nullptr && \*spreadCount > 0){

CHAKRATEL\_LANGSTATS\_INC\_LANGFEATURECOUNT(SpreadFeatureCount, m\_scriptContext);

}

if (buildAST)

{

AssertMem(lastNodeRef);

AssertNodeMem(\*lastNodeRef);

pnodeList->ichLim = (\*lastNodeRef)->ichLim;

if (arrayOfVarInts && arrayOfInts)

{

arrayOfInts = false;

arrayOfNumbers = false;

}

\*pArrayOfTaggedInts = arrayOfTaggedInts;

\*pArrayOfInts = arrayOfInts;

\*pArrayOfNumbers = arrayOfNumbers;

\*pHasMissingValues = hasMissingValues;

}

this->m\_arrayDepth--;

return pnodeList;

}

Parser::MemberNameToTypeMap\* Parser::CreateMemberNameMap(ArenaAllocator\* pAllocator)

{

Assert(pAllocator);

return Anew(pAllocator, MemberNameToTypeMap, pAllocator, 5);

}

template<bool buildAST> void Parser::ParseComputedName(ParseNodePtr\* ppnodeName, LPCOLESTR\* ppNameHint, LPCOLESTR\* ppFullNameHint, ulong \*pNameLength, ulong \*pShortNameOffset)

{

m\_pscan->Scan();

ParseNodePtr pnodeNameExpr = ParseExpr<buildAST>(koplNo, nullptr, TRUE, FALSE, \*ppNameHint, pNameLength, pShortNameOffset);

if (buildAST)

{

\*ppnodeName = CreateNodeT<knopComputedName>(pnodeNameExpr->ichMin, pnodeNameExpr->ichLim);

(\*ppnodeName)->sxUni.pnode1 = pnodeNameExpr;

}

if (ppFullNameHint && buildAST && CONFIG\_FLAG(UseFullName))

{

\*ppFullNameHint = FormatPropertyString(\*ppNameHint, pnodeNameExpr, pNameLength, pShortNameOffset);

}

ChkCurTokNoScan(tkRBrack, ERRsyntax);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Parse a list of object set/get members, e.g.:

{ get foo(){ ... }, set bar(arg) { ... } }

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

template<bool buildAST>

ParseNodePtr Parser::ParseMemberGetSet(OpCode nop, LPCOLESTR\* ppNameHint)

{

ParseNodePtr pnodeName = nullptr;

Assert(nop == knopGetMember || nop == knopSetMember);

AssertMem(ppNameHint);

IdentPtr pid = nullptr;

bool isComputedName = false;

\*ppNameHint=nullptr;

switch(m\_token.tk)

{

default:

if (!m\_token.IsReservedWord())

{

Error(ERRnoMemberIdent);

}

// fall through

case tkID:

pid = m\_token.GetIdentifier(m\_phtbl);

\*ppNameHint = pid->Psz();

if (buildAST)

{

pnodeName = CreateStrNodeWithScanner(pid);

}

break;

case tkStrCon:

if (IsStrictMode() && m\_pscan->IsOctOrLeadingZeroOnLastTKNumber())

{

Error(ERRES5NoOctal);

}

pid = m\_token.GetStr();

\*ppNameHint = pid->Psz();

if (buildAST)

{

pnodeName = CreateStrNodeWithScanner(pid);

}

break;

case tkIntCon:

if (IsStrictMode() && m\_pscan->IsOctOrLeadingZeroOnLastTKNumber())

{

Error(ERRES5NoOctal);

}

pid = m\_pscan->PidFromLong(m\_token.GetLong());

if (buildAST)

{

pnodeName = CreateStrNodeWithScanner(pid);

}

break;

case tkFltCon:

if (IsStrictMode() && m\_pscan->IsOctOrLeadingZeroOnLastTKNumber())

{

Error(ERRES5NoOctal);

}

pid = m\_pscan->PidFromDbl(m\_token.GetDouble());

if (buildAST)

{

pnodeName = CreateStrNodeWithScanner(pid);

}

break;

case tkLBrack:

// Computed property name: get|set [expr] () { }

if (!m\_scriptContext->GetConfig()->IsES6ObjectLiteralsEnabled())

{

Error(ERRnoMemberIdent);

}

LPCOLESTR emptyHint = nullptr;

ulong offset = 0;

ParseComputedName<buildAST>(&pnodeName, &emptyHint, ppNameHint, &offset);

isComputedName = true;

break;

}

MemberType memberType;

ushort flags;

if(nop == knopGetMember)

{

memberType = MemberTypeGetter;

flags = fFncNoArg | fFncNoName;

}

else

{

Assert(nop == knopSetMember);

memberType = MemberTypeSetter;

flags = fFncOneArg | fFncNoName;

}

this->m\_parsingSuperRestrictionState = ParsingSuperRestrictionState\_SuperPropertyAllowed;

ParseNodePtr pnodeFnc = ParseFncDecl<buildAST>(flags | fFncMethod | (nop == knopSetMember ? fFncSetter : fFncNoFlgs), \*ppNameHint,

/\*isSourceElement\*/ false, /\*needsPIDOnRCurlyScan\*/ false, /\*resetParsingSuperRestrictionState\*/ false);

if (buildAST)

{

pnodeFnc->sxFnc.SetIsAccessor();

return CreateBinNode(nop, pnodeName, pnodeFnc);

}

else

{

return nullptr;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Parse a list of object members. e.g. { x:foo, 'y me':bar }

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

template<bool buildAST>

ParseNodePtr Parser::ParseMemberList(LPCOLESTR pNameHint, ulong\* pNameHintLength, tokens declarationType)

{

ParseNodePtr pnodeArg;

ParseNodePtr pnodeName = nullptr;

ParseNodePtr pnodeList = nullptr;

ParseNodePtr \*lastNodeRef = nullptr;

LPCOLESTR pFullNameHint = nullptr; // A calculated full name

ulong fullNameHintLength = pNameHintLength ? \*pNameHintLength : 0;

ulong shortNameOffset = 0;

bool isProtoDeclared = false;

// we get declaration tkLCurly - when the possible object pattern found under the expression.

bool isObjectPattern = (declarationType == tkVAR || declarationType == tkLET || declarationType == tkCONST || declarationType == tkLCurly) && IsES6DestructuringEnabled();

// Check for an empty list

if (tkRCurly == m\_token.tk)

{

return nullptr;

}

ArenaAllocator tempAllocator(L"MemberNames", m\_nodeAllocator.GetPageAllocator(), Parser::OutOfMemory);

for (;;)

{

bool isComputedName = false;

#if DEBUG

if((m\_grfscr & fscrEnforceJSON) && (tkStrCon != m\_token.tk || !(m\_pscan->IsDoubleQuoteOnLastTkStrCon())))

{

Error(ERRsyntax);

}

#endif

bool isAsyncMethod = false;

charcount\_t ichMin = 0;

size\_t iecpMin = 0;

if (m\_token.tk == tkID && m\_token.GetIdentifier(m\_phtbl) == wellKnownPropertyPids.async && m\_scriptContext->GetConfig()->IsES7AsyncAndAwaitEnabled())

{

RestorePoint parsedAsync;

m\_pscan->Capture(&parsedAsync);

ichMin = m\_pscan->IchMinTok();

iecpMin = m\_pscan->IecpMinTok();

m\_pscan->ScanForcingPid();

if (m\_token.tk == tkLParen || m\_token.tk == tkColon || m\_token.tk == tkRCurly)

{

m\_pscan->SeekTo(parsedAsync);

}

else

{

isAsyncMethod = true;

}

}

bool isGenerator = m\_scriptContext->GetConfig()->IsES6GeneratorsEnabled() &&

m\_token.tk == tkStar;

ushort fncDeclFlags = fFncNoName | fFncMethod;

if (isGenerator)

{

if (isAsyncMethod)

{

Error(ERRsyntax);

}

m\_pscan->ScanForcingPid();

fncDeclFlags |= fFncGenerator;

}

IdentPtr pidHint = nullptr; // A name scoped to current expression

Token tkHint = m\_token;

charcount\_t idHintIchMin = static\_cast<charcount\_t>(m\_pscan->IecpMinTok());

charcount\_t idHintIchLim = static\_cast< charcount\_t >(m\_pscan->IecpLimTok());

bool wrapInBrackets = false;

switch (m\_token.tk)

{

default:

if (!m\_token.IsReservedWord())

{

Error(ERRnoMemberIdent);

}

// allow reserved words

wrapInBrackets = true;

// fall-through

case tkID:

pidHint = m\_token.GetIdentifier(m\_phtbl);

if (buildAST)

{

pnodeName = CreateStrNodeWithScanner(pidHint);

}

break;

case tkStrCon:

if (IsStrictMode() && m\_pscan->IsOctOrLeadingZeroOnLastTKNumber())

{

Error(ERRES5NoOctal);

}

wrapInBrackets = true;

pidHint = m\_token.GetStr();

if (buildAST)

{

pnodeName = CreateStrNodeWithScanner(pidHint);

}

break;

case tkIntCon:

// Object initializers with numeric labels allowed in JS6

if (IsStrictMode() && m\_pscan->IsOctOrLeadingZeroOnLastTKNumber())

{

Error(ERRES5NoOctal);

}

pidHint = m\_pscan->PidFromLong(m\_token.GetLong());

if (buildAST)

{

pnodeName = CreateStrNodeWithScanner(pidHint);

}

break;

case tkFltCon:

if (IsStrictMode() && m\_pscan->IsOctOrLeadingZeroOnLastTKNumber())

{

Error(ERRES5NoOctal);

}

pidHint = m\_pscan->PidFromDbl(m\_token.GetDouble());

if (buildAST)

{

pnodeName = CreateStrNodeWithScanner(pidHint);

}

wrapInBrackets = true;

break;

case tkLBrack:

// Computed property name: [expr] : value

if (!m\_scriptContext->GetConfig()->IsES6ObjectLiteralsEnabled())

{

Error(ERRnoMemberIdent);

}

ParseComputedName<buildAST>(&pnodeName, &pNameHint, &pFullNameHint, &fullNameHintLength, &shortNameOffset);

isComputedName = true;

break;

}

if (pFullNameHint == nullptr)

{

if (CONFIG\_FLAG(UseFullName))

{

pFullNameHint = AppendNameHints(pNameHint, pidHint, &fullNameHintLength, &shortNameOffset, false, wrapInBrackets);

}

else

{

pFullNameHint = pidHint? pidHint->Psz() : nullptr;

fullNameHintLength = pidHint ? pidHint->Cch() : 0;

shortNameOffset = 0;

}

}

RestorePoint atPid;

m\_pscan->Capture(&atPid);

m\_pscan->ScanForcingPid();

if (isGenerator && m\_token.tk != tkLParen)

{

Error(ERRnoLparen);

}

if (tkColon == m\_token.tk)

{

// It is a syntax error is the production of the form \_\_proto\_\_ : <> occurs more than once. From B.3.1 in spec.

// Note that previous scan is important because only after that we can determine we have a variable.

if (!isComputedName && pidHint == wellKnownPropertyPids.\_\_proto\_\_)

{

if (isProtoDeclared)

{

Error(ERRsyntax);

}

else

{

isProtoDeclared = true;

}

}

m\_pscan->Scan();

ParseNodePtr pnodeExpr = nullptr;

if (isObjectPattern)

{

pnodeExpr = ParseDestructuredVarDecl<buildAST>(declarationType, declarationType != tkLCurly, nullptr/\* \*hasSeenRest\*/, false /\*topLevel\*/);

if (m\_token.tk != tkComma && m\_token.tk != tkRCurly)

{

if (m\_token.IsOperator())

{

Error(ERRDestructNoOper);

}

Error(ERRsyntax);

}

}

else

{

pnodeExpr = ParseExpr<buildAST>(koplCma, nullptr, TRUE, FALSE, pFullNameHint, &fullNameHintLength, &shortNameOffset);

}

#if DEBUG

if((m\_grfscr & fscrEnforceJSON) && !IsJSONValid(pnodeExpr))

{

Error(ERRsyntax);

}

#endif

if (buildAST)

{

pnodeArg = CreateBinNode(isObjectPattern ? knopObjectPatternMember : knopMember, pnodeName, pnodeExpr);

if (pnodeArg->sxBin.pnode1->nop == knopStr)

{

pnodeArg->sxBin.pnode1->sxPid.pid->PromoteAssignmentState();

}

}

}

else if (m\_token.tk == tkLParen && m\_scriptContext->GetConfig()->IsES6ObjectLiteralsEnabled())

{

if (isObjectPattern)

{

Error(ERRInvalidAssignmentTarget);

}

// Shorthand syntax: foo() {} -> foo: function() {}

// Rewind to the PID and parse a function expression.

m\_pscan->SeekTo(atPid);

this->m\_parsingSuperRestrictionState = ParsingSuperRestrictionState\_SuperPropertyAllowed;

ParseNodePtr pnodeFunc = ParseFncDecl<buildAST>(fncDeclFlags | (isAsyncMethod ? fFncAsync : fFncNoFlgs), pFullNameHint,

/\*isSourceElement\*/ false, /\*needsPIDOnRCurlyScan\*/ false, /\*resetParsingSuperRestrictionState\*/ false);

if (isAsyncMethod)

{

pnodeFunc->sxFnc.cbMin = iecpMin;

pnodeFunc->ichMin = ichMin;

}

if (buildAST)

{

pnodeArg = CreateBinNode(knopMember, pnodeName, pnodeFunc);

}

}

else if (nullptr != pidHint) //Its either tkID/tkStrCon/tkFloatCon/tkIntCon

{

Assert(pidHint->Psz() != nullptr);

if (pidHint == wellKnownPropertyPids.getter && tkHint.tk == tkID)

{

if (isObjectPattern)

{

Error(ERRInvalidAssignmentTarget);

}

LPCOLESTR pNameGet = nullptr;

pnodeArg = ParseMemberGetSet<buildAST>(knopGetMember, &pNameGet);

if (CONFIG\_FLAG(UseFullName) && buildAST && pnodeArg->sxBin.pnode2->nop == knopFncDecl)

{

if (m\_scriptContext->GetConfig()->IsES6FunctionNameEnabled())

{

// displays as get object.funcname

ulong getOffset = 0;

pFullNameHint = AppendNameHints(wellKnownPropertyPids.getter, AppendNameHints(pNameHint, pNameGet, &fullNameHintLength, &shortNameOffset), &fullNameHintLength, &getOffset, true);

shortNameOffset += getOffset;

}

else

{

// displays as object.funcname.get

pFullNameHint = AppendNameHints(pNameHint, AppendNameHints(pNameGet, wellKnownPropertyPids.getter, &fullNameHintLength, &shortNameOffset), &fullNameHintLength, &shortNameOffset);

}

}

}

else if (pidHint == wellKnownPropertyPids.setter && tkHint.tk == tkID)

{

if (isObjectPattern)

{

Error(ERRInvalidAssignmentTarget);

}

LPCOLESTR pNameSet = nullptr;

pnodeArg = ParseMemberGetSet<buildAST>(knopSetMember, &pNameSet);

if (CONFIG\_FLAG(UseFullName) && buildAST && pnodeArg->sxBin.pnode2->nop == knopFncDecl)

{

if (m\_scriptContext->GetConfig()->IsES6FunctionNameEnabled())

{

// displays as set object.funcname

ulong setOffset = 0;

pFullNameHint = AppendNameHints(wellKnownPropertyPids.setter, AppendNameHints(pNameHint, pNameSet, &fullNameHintLength, &shortNameOffset), &fullNameHintLength, &setOffset, true);

shortNameOffset += setOffset;

}

else

{

// displays as object.funcname.set

pFullNameHint = AppendNameHints(pNameHint, AppendNameHints(pNameSet, wellKnownPropertyPids.setter, &fullNameHintLength, &shortNameOffset), &fullNameHintLength, &shortNameOffset);

}

}

}

else if ((m\_token.tk == tkRCurly || m\_token.tk == tkComma || (isObjectPattern && m\_token.tk == tkAsg)) && m\_scriptContext->GetConfig()->IsES6ObjectLiteralsEnabled())

{

// Shorthand {foo} -> {foo:foo} syntax.

// {foo = <initializer>} supported only when on object pattern rules are being applied

if (tkHint.tk != tkID)

{

Assert(tkHint.IsReservedWord()

|| tkHint.tk == tkIntCon || tkHint.tk == tkFltCon || tkHint.tk == tkStrCon);

// All keywords are banned in non-strict mode.

// Future reserved words are banned in strict mode.

if (IsStrictMode() || !tkHint.IsFutureReservedWord(true))

{

IdentifierExpectedError(tkHint);

}

}

if (buildAST)

{

CheckArgumentsUse(pidHint, GetCurrentFunctionNode());

}

ParseNodePtr pnodeIdent = nullptr;

if (isObjectPattern)

{

m\_pscan->SeekTo(atPid);

pnodeIdent = ParseDestructuredVarDecl<buildAST>(declarationType, declarationType != tkLCurly, nullptr/\* \*hasSeenRest\*/, false /\*topLevel\*/);

if (m\_token.tk != tkComma && m\_token.tk != tkRCurly)

{

if (m\_token.IsOperator())

{

Error(ERRDestructNoOper);

}

Error(ERRsyntax);

}

}

if (buildAST)

{

if (!isObjectPattern)

{

pnodeIdent = CreateNameNode(pidHint, idHintIchMin, idHintIchLim);

PidRefStack \*ref = PushPidRef(pidHint);

pnodeIdent->sxPid.SetSymRef(ref);

}

pnodeArg = CreateBinNode(isObjectPattern ? knopObjectPatternMember : knopMemberShort, pnodeName, pnodeIdent);

}

}

else

{

Error(ERRnoColon);

}

}

else

{

Error(ERRnoColon);

}

if (buildAST)

{

Assert(pnodeArg->sxBin.pnode2 != nullptr);

if (pnodeArg->sxBin.pnode2->nop == knopFncDecl)

{

Assert(fullNameHintLength >= shortNameOffset);

pnodeArg->sxBin.pnode2->sxFnc.hint = pFullNameHint;

pnodeArg->sxBin.pnode2->sxFnc.hintLength = fullNameHintLength;

pnodeArg->sxBin.pnode2->sxFnc.hintOffset = shortNameOffset;

}

AddToNodeListEscapedUse(&pnodeList, &lastNodeRef, pnodeArg);

}

pidHint = nullptr;

pFullNameHint = nullptr;

if (tkComma != m\_token.tk)

{

break;

}

m\_pscan->ScanForcingPid();

if (tkRCurly == m\_token.tk)

{

break;

}

}

if (buildAST)

{

AssertMem(lastNodeRef);

AssertNodeMem(\*lastNodeRef);

pnodeList->ichLim = (\*lastNodeRef)->ichLim;

}

return pnodeList;

}

BOOL Parser::DeferredParse(Js::LocalFunctionId functionId)

{

if ((m\_grfscr & fscrDeferFncParse) != 0)

{

if (m\_stoppedDeferredParse)

{

return false;

}

if (PHASE\_OFF\_RAW(Js::DeferParsePhase, m\_sourceContextInfo->sourceContextId, functionId))

{

return false;

}

if (PHASE\_FORCE\_RAW(Js::DeferParsePhase, m\_sourceContextInfo->sourceContextId, functionId))

{

return true;

}

#if ENABLE\_PROFILE\_INFO

#ifndef DISABLE\_DYNAMIC\_PROFILE\_DEFER\_PARSE

if (m\_sourceContextInfo->sourceDynamicProfileManager != nullptr)

{

Js::ExecutionFlags flags = m\_sourceContextInfo->sourceDynamicProfileManager->IsFunctionExecuted(functionId);

return flags != Js::ExecutionFlags\_Executed;

}

#endif

#endif

return true;

}

return false;

}

//

// Call this in ParseFncDecl only to check (and reset) if ParseFncDecl is re-parsing a deferred

// function body. If a deferred function is called and being re-parsed, it shouldn't be deferred again.

//

BOOL Parser::IsDeferredFnc()

{

if (m\_grfscr & fscrDeferredFnc)

{

m\_grfscr &= ~fscrDeferredFnc;

return true;

}

return false;

}

template<bool buildAST>

ParseNodePtr Parser::ParseFncDecl(ushort flags, LPCOLESTR pNameHint, const bool isSourceElement, const bool needsPIDOnRCurlyScan, bool resetParsingSuperRestrictionState, bool fUnaryOrParen)

{

AutoParsingSuperRestrictionStateRestorer restorer(this);

if (resetParsingSuperRestrictionState)

{

// ParseFncDecl will always reset m\_parsingSuperRestrictionState to super disallowed unless explicitly disabled

this->m\_parsingSuperRestrictionState = ParsingSuperRestrictionState\_SuperDisallowed;

}

ParseNodePtr pnodeFnc = nullptr;

ParseNodePtr \*ppnodeVarSave = nullptr;

ParseNodePtr pnodeFncSave = nullptr;

ParseNodePtr pnodeFncSaveNonLambda = nullptr;

ParseNodePtr pnodeFncBlockScope = nullptr;

ParseNodePtr \*ppnodeScopeSave = nullptr;

ParseNodePtr \*ppnodeExprScopeSave = nullptr;

bool funcHasName = false;

bool fDeclaration = flags & fFncDeclaration;

bool fLambda = (flags & fFncLambda) != 0;

charcount\_t ichMin = this->m\_pscan->IchMinTok();

bool wasInDeferredNestedFunc = false;

CatchPidRefList \*catchPidRefList = nullptr;

uint tryCatchOrFinallyDepthSave = this->m\_tryCatchOrFinallyDepth;

this->m\_tryCatchOrFinallyDepth = 0;

CheckStrictModeFncDeclNotSourceElement(isSourceElement, fDeclaration);

if (this->m\_arrayDepth)

{

this->m\_funcInArrayDepth++; // Count function depth within array literal

}

// Update the count of functions nested in the current parent.

Assert(m\_pnestedCount || !buildAST);

uint \*pnestedCountSave = m\_pnestedCount;

if (buildAST || m\_pnestedCount)

{

(\*m\_pnestedCount)++;

}

uint scopeCountNoAstSave = m\_scopeCountNoAst;

m\_scopeCountNoAst = 0;

long\* pAstSizeSave = m\_pCurrentAstSize;

bool noStmtContext = false;

if (buildAST || BindDeferredPidRefs())

{

if (fDeclaration && m\_scriptContext->GetConfig()->IsBlockScopeEnabled())

{

noStmtContext =

(m\_pstmtCur->isDeferred && m\_pstmtCur->op != knopBlock) ||

(!m\_pstmtCur->isDeferred && m\_pstmtCur->pnodeStmt->nop != knopBlock);

if (noStmtContext)

{

// We have a function declaration like "if (a) function f() {}". We didn't see

// a block scope on the way in, so we need to pretend we did. Note that this is a syntax error

// in strict mode.

if (!this->FncDeclAllowedWithoutContext(flags))

{

Error(ERRsyntax);

}

pnodeFncBlockScope = StartParseBlock<buildAST>(PnodeBlockType::Regular, ScopeType\_Block);

if (buildAST)

{

PushFuncBlockScope(pnodeFncBlockScope, &ppnodeScopeSave, &ppnodeExprScopeSave);

}

}

}

// Create the node.

pnodeFnc = CreateNode(knopFncDecl);

pnodeFnc->sxFnc.ClearFlags();

pnodeFnc->sxFnc.SetDeclaration(fDeclaration);

pnodeFnc->sxFnc.astSize = 0;

pnodeFnc->sxFnc.pnodeName = nullptr;

pnodeFnc->sxFnc.pnodeScopes = nullptr;

pnodeFnc->sxFnc.pnodeRest = nullptr;

pnodeFnc->sxFnc.pid = nullptr;

pnodeFnc->sxFnc.hint = nullptr;

pnodeFnc->sxFnc.hintOffset = 0;

pnodeFnc->sxFnc.hintLength = 0;

pnodeFnc->sxFnc.isNameIdentifierRef = true;

pnodeFnc->sxFnc.pnodeNext = nullptr;

pnodeFnc->sxFnc.pnodeArgs = nullptr;

pnodeFnc->sxFnc.pnodeVars = nullptr;

pnodeFnc->sxFnc.funcInfo = nullptr;

pnodeFnc->sxFnc.deferredStub = nullptr;

pnodeFnc->sxFnc.nestedCount = 0;

pnodeFnc->sxFnc.cbMin = m\_pscan->IecpMinTok();

pnodeFnc->sxFnc.functionId = (\*m\_nextFunctionId)++;

// Push new parser state with this new function node

AppendFunctionToScopeList(fDeclaration, pnodeFnc);

// Start the argument list.

ppnodeVarSave = m\_ppnodeVar;

}

else

{

(\*m\_nextFunctionId)++;

}

if (buildAST)

{

pnodeFnc->sxFnc.lineNumber = m\_pscan->LineCur();

pnodeFnc->sxFnc.columnNumber = CalculateFunctionColumnNumber();

pnodeFnc->sxFnc.SetNested(m\_currentNodeFunc != nullptr); // If there is a current function, then we're a nested function.

pnodeFnc->sxFnc.SetStrictMode(IsStrictMode()); // Inherit current strict mode -- may be overridden by the function itself if it contains a strict mode directive.

pnodeFnc->sxFnc.firstDefaultArg = 0;

m\_pCurrentAstSize = &pnodeFnc->sxFnc.astSize;

// Make this the current function and start its sub-function list.

pnodeFncSave = m\_currentNodeFunc;

m\_currentNodeFunc = pnodeFnc;

if (!fLambda)

{

pnodeFncSaveNonLambda = m\_currentNodeNonLambdaFunc;

m\_currentNodeNonLambdaFunc = pnodeFnc;

}

m\_pnestedCount = &pnodeFnc->sxFnc.nestedCount;

catchPidRefList = this->GetCatchPidRefList();

if (catchPidRefList)

{

Assert(!m\_scriptContext->GetConfig()->IsBlockScopeEnabled());

if (fDeclaration)

{

// We're starting a function declaration, and we're inside some number

// of catches, and the catch has its own scope but the function gets hoisted

// outside it. We have to fiddle with the PidRefStack's to simulate hoisting.

// For each catch object in scope here, do the following:

// - Remove the portion of the PID ref stack that holds references inside the catch.

// - Save that portion of the stack in the catchPidRef list entry. Do this by:

// - Letting the list entry point to the current top of the PID ref stack;

// - Setting the prev pointer of the PID ref at the bottom of the removed portion to null.

// Now we can accumulate references inside the function declaration without getting them

// interspersed with the references that should bind to the catch variable.

FOREACH\_SLISTBASE\_ENTRY(CatchPidRef, catchPidRef, catchPidRefList)

{

IdentPtr pidCatch = catchPidRef.pid;

PidRefStack \*topRef = pidCatch->GetTopRef();

PidRefStack \*catchScopeRef = catchPidRef.ref;

catchPidRef.ref = topRef;

pidCatch->SetTopRef(catchScopeRef->prev);

catchScopeRef->prev = nullptr;

}

NEXT\_SLISTBASE\_ENTRY;

catchPidRefList->Reverse();

}

this->SetCatchPidRefList(nullptr);

}

}

else // if !buildAST

{

wasInDeferredNestedFunc = m\_inDeferredNestedFunc;

m\_inDeferredNestedFunc = true;

if (BindDeferredPidRefs())

{

AnalysisAssert(pnodeFnc);

if (!fLambda)

{

pnodeFncSaveNonLambda = m\_currentNodeNonLambdaDeferredFunc;

m\_currentNodeNonLambdaDeferredFunc = pnodeFnc;

}

pnodeFncSave = m\_currentNodeDeferredFunc;

m\_currentNodeDeferredFunc = pnodeFnc;

m\_pnestedCount = &pnodeFnc->sxFnc.nestedCount;

}

else

{

m\_pnestedCount = nullptr;

}

}

if (buildAST || BindDeferredPidRefs())

{

AnalysisAssert(pnodeFnc);

pnodeFnc->sxFnc.SetIsAsync((flags & fFncAsync) != 0);

pnodeFnc->sxFnc.SetIsLambda(fLambda);

pnodeFnc->sxFnc.SetIsMethod((flags & fFncMethod) != 0);

pnodeFnc->sxFnc.SetIsClassMember((flags & fFncClassMember) != 0);

}

bool needScanRCurly = true;

bool result = ParseFncDeclHelper<buildAST>(pnodeFnc, pnodeFncSave, pNameHint, flags, &funcHasName, fUnaryOrParen, noStmtContext, &needScanRCurly);

if (!result)

{

Assert(!pnodeFncBlockScope);

return pnodeFnc;

}

if (buildAST || BindDeferredPidRefs())

{

AnalysisAssert(pnodeFnc);

\*m\_ppnodeVar = nullptr;

m\_ppnodeVar = ppnodeVarSave;

// Restore the current function.

if (buildAST)

{

Assert(pnodeFnc == m\_currentNodeFunc);

m\_currentNodeFunc = pnodeFncSave;

m\_pCurrentAstSize = pAstSizeSave;

if (!fLambda)

{

Assert(pnodeFnc == m\_currentNodeNonLambdaFunc);

m\_currentNodeNonLambdaFunc = pnodeFncSaveNonLambda;

}

}

else

{

Assert(pnodeFnc == m\_currentNodeDeferredFunc);

if (!fLambda)

{

Assert(pnodeFnc == m\_currentNodeNonLambdaDeferredFunc);

m\_currentNodeNonLambdaDeferredFunc = pnodeFncSaveNonLambda;

}

m\_currentNodeDeferredFunc = pnodeFncSave;

if (m\_currentNodeFunc && pnodeFnc->sxFnc.HasWithStmt())

{

GetCurrentFunctionNode()->sxFnc.SetHasWithStmt(true);

}

}

if (m\_currentNodeFunc && (pnodeFnc->sxFnc.CallsEval() || pnodeFnc->sxFnc.ChildCallsEval()))

{

GetCurrentFunctionNode()->sxFnc.SetChildCallsEval(true);

}

// Lambdas do not have "arguments" and instead capture their parent's

// binding of "arguments. To ensure the arguments object of the enclosing

// non-lambda function is loaded propagate the UsesArguments flag up to

// the parent function

if ((flags & fFncLambda) != 0 && pnodeFnc->sxFnc.UsesArguments())

{

if (pnodeFncSave != nullptr)

{

pnodeFncSave->sxFnc.SetUsesArguments();

}

else

{

m\_UsesArgumentsAtGlobal = true;

}

}

}

if (needScanRCurly)

{

// Consume the next token now that we're back in the enclosing function (whose strictness may be

// different from the function we just finished).

#if DBG

bool expectedTokenValid = m\_token.tk == tkRCurly;

AssertMsg(expectedTokenValid, "Invalid token expected for RCurly match");

#endif

// The next token may need to have a PID created in !buildAST mode, as we may be parsing a method with a string name.

if (needsPIDOnRCurlyScan)

{

m\_pscan->ScanForcingPid();

}

else

{

m\_pscan->Scan();

}

}

m\_pnestedCount = pnestedCountSave;

Assert(!buildAST || !wasInDeferredNestedFunc);

m\_inDeferredNestedFunc = wasInDeferredNestedFunc;

if (this->m\_arrayDepth)

{

this->m\_funcInArrayDepth--;

if (this->m\_funcInArrayDepth == 0 && !this->m\_parsingDuplicate)

{

// We disable deferred parsing if array literals dominate.

// But don't do this if the array literal is dominated by function bodies.

if (flags & (fFncMethod | fFncClassMember) && m\_token.tk != tkSColon)

{

// Class member methods have optional separators. We need to check whether we are

// getting the IchLim of the correct token.

Assert(m\_pscan->m\_tkPrevious == tkRCurly && needScanRCurly);

this->m\_funcInArray += m\_pscan->IchMinTok() - /\*tkRCurly\*/ 1 - ichMin;

}

else

{

this->m\_funcInArray += m\_pscan->IchLimTok() - ichMin;

}

}

}

m\_scopeCountNoAst = scopeCountNoAstSave;

if (buildAST)

{

if (catchPidRefList)

{

if (this->GetCatchPidRefList())

{

// We may have had catches inside the function we just finished. If so, we should be done

// with them all (so the ref list should be empty), and we can throw away the list.

Assert(this->GetCatchPidRefList()->Empty());

Adelete(&m\_nodeAllocator, this->GetCatchPidRefList());

}

this->SetCatchPidRefList(catchPidRefList);

if (fDeclaration)

{

// We're finishing a function declaration inside a catch. For each catch variable that's in

// scope here, put the portion of the PID ref stack that we removed and saved back on the top

// of the stack. When we finish the catch, the references in this restored portion of the stack

// will be bound to the catch variable, but those that belong the function body will

// be left behind to be bound to the context outside the catch.

FOREACH\_SLISTBASE\_ENTRY(CatchPidRef, catchPidRef, catchPidRefList)

{

IdentPtr pidCatch = catchPidRef.pid;

PidRefStack \*oldTopRef = pidCatch->GetTopRef();

PidRefStack \*ref = catchPidRef.ref;

pidCatch->SetTopRef(ref);

while (ref->prev)

{

ref = ref->prev;

}

ref->prev = oldTopRef;

catchPidRef.ref = ref;

}

NEXT\_SLISTBASE\_ENTRY;

catchPidRefList->Reverse();

}

}

}

if (buildAST && fDeclaration && m\_scriptContext->GetConfig()->IsBlockScopeEnabled() && !IsStrictMode())

{

if (pnodeFnc->sxFnc.pnodeName != nullptr && pnodeFnc->sxFnc.pnodeName->nop == knopVarDecl &&

GetCurrentBlock()->sxBlock.blockType == PnodeBlockType::Regular)

{

// Add a function-scoped VarDecl with the same name as the function for

// back compat with pre-ES6 code that declares functions in blocks. The

// idea is that the last executed declaration wins at the function scope

// level and we accomplish this by having each block scoped function

// declaration assign to both the block scoped "let" binding, as well

// as the function scoped "var" binding.

ParseNodePtr vardecl = CreateVarDeclNode(pnodeFnc->sxFnc.pnodeName->sxVar.pid, STVariable, false, nullptr, false);

vardecl->sxVar.isBlockScopeFncDeclVar = true;

}

}

if (pnodeFncBlockScope)

{

Assert(pnodeFncBlockScope->sxBlock.pnodeStmt == nullptr);

pnodeFncBlockScope->sxBlock.pnodeStmt = pnodeFnc;

if (buildAST)

{

PopFuncBlockScope(ppnodeScopeSave, ppnodeExprScopeSave);

}

FinishParseBlock(pnodeFncBlockScope);

return pnodeFncBlockScope;

}

this->m\_tryCatchOrFinallyDepth = tryCatchOrFinallyDepthSave;

return pnodeFnc;

}

bool Parser::FncDeclAllowedWithoutContext(ushort flags)

{

// Statement context required for strict mode, async functions, and generators.

// Note that generators aren't detected yet when this method is called; they're checked elsewhere.

return !IsStrictMode() && !(flags & fFncAsync);

}

uint Parser::CalculateFunctionColumnNumber()

{

uint columnNumber;

if (m\_pscan->IchMinTok() >= m\_pscan->IchMinLine())

{

// In scenarios involving defer parse IchMinLine() can be incorrect for the first line after defer parse

columnNumber = m\_pscan->IchMinTok() - m\_pscan->IchMinLine();

if (m\_functionBody != nullptr && m\_functionBody->GetRelativeLineNumber() == m\_pscan->LineCur())

{

// Adjust the column if it falls on the first line, where the re-parse is happening.

columnNumber += m\_functionBody->GetRelativeColumnNumber();

}

}

else if (m\_currentNodeFunc)

{

// For the first line after defer parse, compute the column relative to the column number

// of the lexically parent function.

ULONG offsetFromCurrentFunction = m\_pscan->IchMinTok() - m\_currentNodeFunc->ichMin;

columnNumber = m\_currentNodeFunc->sxFnc.columnNumber + offsetFromCurrentFunction ;

}

else

{

// if there is no current function, lets give a default of 0.

columnNumber = 0;

}

return columnNumber;

}

void Parser::AppendFunctionToScopeList(bool fDeclaration, ParseNodePtr pnodeFnc)

{

if (!fDeclaration && m\_ppnodeExprScope)

{

// We're tracking function expressions separately from declarations in this scope

// (e.g., inside a catch scope in standards mode).

Assert(\*m\_ppnodeExprScope == nullptr);

\*m\_ppnodeExprScope = pnodeFnc;

m\_ppnodeExprScope = &pnodeFnc->sxFnc.pnodeNext;

}

else

{

Assert(\*m\_ppnodeScope == nullptr);

\*m\_ppnodeScope = pnodeFnc;

m\_ppnodeScope = &pnodeFnc->sxFnc.pnodeNext;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Parse a function definition.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

template<bool buildAST>

bool Parser::ParseFncDeclHelper(ParseNodePtr pnodeFnc, ParseNodePtr pnodeFncParent, LPCOLESTR pNameHint, ushort flags, bool \*pHasName, bool fUnaryOrParen, bool noStmtContext, bool \*pNeedScanRCurly)

{

bool fDeclaration = (flags & fFncDeclaration) != 0;

bool fLambda = (flags & fFncLambda) != 0;

bool fAsync = (flags & fFncAsync) != 0;

bool fDeferred = false;

StmtNest \*pstmtSave;

ParseNodePtr \*lastNodeRef = nullptr;

bool fFunctionInBlock = false;

if (buildAST)

{

fFunctionInBlock = GetCurrentBlockInfo() != GetCurrentFunctionBlockInfo() &&

(GetCurrentBlockInfo()->pnodeBlock->sxBlock.scope == nullptr ||

GetCurrentBlockInfo()->pnodeBlock->sxBlock.scope->GetScopeType() != ScopeType\_GlobalEvalBlock);

}

// Save the position of the scanner in case we need to inspect the name hint later

RestorePoint beginNameHint;

m\_pscan->Capture(&beginNameHint);

ParseNodePtr pnodeFncExprScope = nullptr;

Scope \*fncExprScope = nullptr;

if ((buildAST || BindDeferredPidRefs()) &&

!fDeclaration)

{

pnodeFncExprScope = StartParseBlock<buildAST>(PnodeBlockType::Function, ScopeType\_FuncExpr);

fncExprScope = pnodeFncExprScope->sxBlock.scope;

}

\*pHasName = !fLambda && this->ParseFncNames<buildAST>(pnodeFnc, pnodeFncParent, flags, &lastNodeRef);

if (noStmtContext && pnodeFnc->sxFnc.IsGenerator())

{

// Generator decl not allowed outside stmt context. (We have to wait until we've parsed the '\*' to

// detect generator.)

Error(ERRsyntax, pnodeFnc);

}

// switch scanner to treat 'yield' as keyword in generator functions

// or as an identifier in non-generator functions

bool fPreviousYieldIsKeyword = m\_pscan->SetYieldIsKeyword(pnodeFnc && pnodeFnc->sxFnc.IsGenerator());

bool fPreviousAwaitIsKeyword = m\_pscan->SetAwaitIsKeyword(fAsync);

if (pnodeFnc && pnodeFnc->sxFnc.IsGenerator())

{

CHAKRATEL\_LANGSTATS\_INC\_LANGFEATURECOUNT(GeneratorCount, m\_scriptContext);

}

if (fncExprScope && !\*pHasName)

{

FinishParseBlock(pnodeFncExprScope);

m\_nextBlockId--;

Adelete(&m\_nodeAllocator, fncExprScope);

fncExprScope = nullptr;

pnodeFncExprScope = nullptr;

}

if (pnodeFnc)

{

pnodeFnc->sxFnc.scope = fncExprScope;

}

// Start a new statement stack.

bool topLevelStmt =

buildAST &&

!fFunctionInBlock &&

(this->m\_pstmtCur == nullptr || this->m\_pstmtCur->pnodeStmt->nop == knopBlock);

pstmtSave = m\_pstmtCur;

SetCurrentStatement(nullptr);

RestorePoint beginFormals;

m\_pscan->Capture(&beginFormals);

BOOL fWasAlreadyStrictMode = IsStrictMode();

BOOL oldStrictMode = this->m\_fUseStrictMode;

if (fLambda)

{

// lambda formals are parsed in strict mode always

m\_fUseStrictMode = TRUE;

CHAKRATEL\_LANGSTATS\_INC\_LANGFEATURECOUNT(LambdaCount, m\_scriptContext);

}

uint uDeferSave = m\_grfscr & fscrDeferFncParse;

if ((!fDeclaration && m\_ppnodeExprScope) ||

(m\_scriptContext->GetConfig()->IsBlockScopeEnabled() && fFunctionInBlock) ||

(flags & (fFncNoName | fFncLambda)))

{

// NOTE: Don't defer if this is a function expression inside a construct that induces

// a scope nested within the current function (like a with, or a catch in ES5 mode, or

// any function declared inside a nested lexical block in ES6 mode).

// We won't be able to reconstruct the scope chain properly when we come back and

// try to compile just the function expression.

// Also shut off deferring on getter/setter or other construct with unusual text bounds

// (fFncNoName|fFncLambda) as these are usually trivial, and re-parsing is problematic.

m\_grfscr &= ~fscrDeferFncParse;

}

bool isTopLevelDeferredFunc = false;

struct AutoFastScanFlag {

bool savedDoingFastScan;

AutoFastScanFlag(Parser \*parser) : m\_parser(parser) { savedDoingFastScan = m\_parser->m\_doingFastScan; }

~AutoFastScanFlag() { m\_parser->m\_doingFastScan = savedDoingFastScan; }

Parser \*m\_parser;

} flag(this);

bool doParallel = false;

bool parallelJobStarted = false;

if (buildAST)

{

bool isLikelyModulePattern =

!fDeclaration && pnodeFnc && pnodeFnc->sxFnc.pnodeName == nullptr && fUnaryOrParen;

BOOL isDeferredFnc = IsDeferredFnc();

AnalysisAssert(isDeferredFnc || pnodeFnc);

isTopLevelDeferredFunc =

(!isDeferredFnc

&& DeferredParse(pnodeFnc->sxFnc.functionId)

&& (!pnodeFnc->sxFnc.IsNested() || CONFIG\_FLAG(DeferNested))

// Don't defer if this is a function expression not contained in a statement or other expression.

// Assume it will be called as part of this expression.

&& (!isLikelyModulePattern || !topLevelStmt || PHASE\_FORCE1(Js::DeferParsePhase))

&& !m\_InAsmMode

);

if (!fLambda &&

!isDeferredFnc &&

!isLikelyModulePattern &&

!this->IsBackgroundParser() &&

!this->m\_doingFastScan &&

!(pnodeFncParent && m\_currDeferredStub) &&

!(this->m\_parseType == ParseType\_Deferred && this->m\_functionBody && this->m\_functionBody->GetScopeInfo() && !isTopLevelDeferredFunc))

{

doParallel = DoParallelParse(pnodeFnc);

#if ENABLE\_BACKGROUND\_PARSING

if (doParallel)

{

BackgroundParser \*bgp = m\_scriptContext->GetBackgroundParser();

Assert(bgp);

if (bgp->HasFailedBackgroundParseItem())

{

Error(ERRsyntax);

}

doParallel = bgp->ParseBackgroundItem(this, pnodeFnc, isTopLevelDeferredFunc);

if (doParallel)

{

parallelJobStarted = true;

this->m\_hasParallelJob = true;

this->m\_doingFastScan = true;

doParallel = FastScanFormalsAndBody();

if (doParallel)

{

// Let the foreground thread take care of marking the limit on the function node,

// because in some cases this function's caller will want to change that limit,

// so we don't want the background thread to try and touch it.

pnodeFnc->ichLim = m\_pscan->IchLimTok();

pnodeFnc->sxFnc.cbLim = m\_pscan->IecpLimTok();

}

}

}

#endif

}

}

if (!doParallel)

{

// We don't want to, or couldn't, let the main thread scan past this function body, so parse

// it for real.

ParseNodePtr pnodeRealFnc = pnodeFnc;

if (parallelJobStarted)

{

// We have to deal with a failure to fast-scan the function (due to syntax error? "/"?) when

// a background thread may already have begun to work on the job. Both threads can't be allowed to

// operate on the same node.

pnodeFnc = CreateDummyFuncNode(fDeclaration);

}

ParseNodePtr pnodeBlock = nullptr;

if (buildAST || BindDeferredPidRefs())

{

AnalysisAssert(pnodeFnc);

pnodeBlock = StartParseBlock<buildAST>(PnodeBlockType::Parameter, ScopeType\_Parameter);

pnodeFnc->sxFnc.pnodeScopes = pnodeBlock;

m\_ppnodeVar = &pnodeFnc->sxFnc.pnodeArgs;

}

ParseNodePtr \*ppnodeScopeSave = nullptr;

ParseNodePtr \*ppnodeExprScopeSave = nullptr;

ppnodeScopeSave = m\_ppnodeScope;

if (pnodeBlock)

{

// This synthetic block scope will contain all the nested scopes.

m\_ppnodeScope = &pnodeBlock->sxBlock.pnodeScopes;

pnodeBlock->sxBlock.pnodeStmt = pnodeFnc;

}

// Keep nested function declarations and expressions in the same list at function scope.

// (Indicate this by nulling out the current function expressions list.)

ppnodeExprScopeSave = m\_ppnodeExprScope;

m\_ppnodeExprScope = nullptr;

this->ParseFncFormals<buildAST>(pnodeFnc, flags);

m\_fUseStrictMode = oldStrictMode;

// Create function body scope

ParseNodePtr pnodeInnerBlock = nullptr;

if (buildAST || BindDeferredPidRefs())

{

pnodeInnerBlock = StartParseBlock<buildAST>(PnodeBlockType::Function, ScopeType\_FunctionBody);

// Set the parameter block's child to the function body block.

\*m\_ppnodeScope = pnodeInnerBlock;

AnalysisAssert(pnodeFnc);

pnodeFnc->sxFnc.pnodeBodyScope = pnodeInnerBlock;

// This synthetic block scope will contain all the nested scopes.

m\_ppnodeScope = &pnodeInnerBlock->sxBlock.pnodeScopes;

pnodeInnerBlock->sxBlock.pnodeStmt = pnodeFnc;

}

// DEFER: Begin deferral here (after names are parsed and name nodes created).

// Create no more AST nodes until we're done.

// Try to defer this func if all these are true:

// 0. We are not already in deferred parsing (i.e. buildAST is true)

// 1. We are not re-parsing a deferred func which is being invoked.

// 2. Dynamic profile suggests this func can be deferred (and deferred parse is on).

// 3. This func is top level or defer nested func is on.

// 4. Optionally, the function is non-nested and not in eval, or the deferral decision was based on cached profile info,

// or the function is sufficiently long. (I.e., don't defer little nested functions unless we're

// confident they'll never be executed, because un-deferring nested functions is more expensive.)

// NOTE: I'm disabling #4 by default, because we've found other ways to reduce the cost of un-deferral,

// and we don't want to create function bodies aggressively for little functions.

// We will also temporarily defer all asm.js functions, except for the asm.js

// module itself, which we will never defer

bool strictModeTurnedOn = false;

if (isTopLevelDeferredFunc &&

!(this->m\_grfscr & fscrEvalCode) &&

pnodeFnc->sxFnc.IsNested() &&

#ifndef DISABLE\_DYNAMIC\_PROFILE\_DEFER\_PARSE

m\_sourceContextInfo->sourceDynamicProfileManager == nullptr &&

#endif

PHASE\_ON\_RAW(Js::ScanAheadPhase, m\_sourceContextInfo->sourceContextId, pnodeFnc->sxFnc.functionId) &&

(

!PHASE\_FORCE\_RAW(Js::DeferParsePhase, m\_sourceContextInfo->sourceContextId, pnodeFnc->sxFnc.functionId) ||

PHASE\_FORCE\_RAW(Js::ScanAheadPhase, m\_sourceContextInfo->sourceContextId, pnodeFnc->sxFnc.functionId)

))

{

// Try to scan ahead to the end of the function. If we get there before we've scanned a minimum

// number of tokens, don't bother deferring, because it's too small.

if (this->ScanAheadToFunctionEnd(CONFIG\_FLAG(MinDeferredFuncTokenCount)))

{

isTopLevelDeferredFunc = false;

}

}

if (fAsync)

{

if (!buildAST || isTopLevelDeferredFunc)

{

// We increment m\_nextFunctionId when there is an Async function to counterbalance the functionId because of the added generator to the AST with an async function that we use to keep deferred parsing in sync with non-deferred parsing

(\*m\_nextFunctionId)++;

}

// Same than before, we increment the nestedCount because we will have a Generator inside any async function.

pnodeFnc->sxFnc.nestedCount++;

}

if (isTopLevelDeferredFunc || (m\_InAsmMode && m\_deferAsmJs))

{

AssertMsg(!fLambda, "Deferring function parsing of a function does not handle lambda syntax");

fDeferred = true;

this->ParseTopLevelDeferredFunc(pnodeFnc, pnodeFncParent, pNameHint);

}

else

{

if (m\_token.tk == tkRParen) // This might be false due to error recovery or lambda.

{

m\_pscan->Scan();

}

if (fLambda)

{

BOOL hadNewLine = m\_pscan->FHadNewLine();

// it can be the case we do not have a fat arrow here if there is a valid expression on the left hand side

// of the fat arrow, but that expression does not parse as a parameter list. E.g.

// a.x => { }

// Therefore check for it and error if not found.

// LS Mode : since this is a lambda we supposed to get the fat arrow, if not we will skip till we get that fat arrow.

ChkCurTok(tkDArrow, ERRnoDArrow);

// Newline character between arrow parameters and fat arrow is a syntax error but we want to check for

// this after verifying there was a => token. Otherwise we would throw the wrong error.

if (hadNewLine)

{

Error(ERRsyntax);

}

}

if (buildAST || BindDeferredPidRefs())

{

AnalysisAssert(pnodeFnc);

// Shouldn't be any temps in the arg list.

Assert(\*m\_ppnodeVar == nullptr);

// Start the var list.

pnodeFnc->sxFnc.pnodeVars = nullptr;

m\_ppnodeVar = &pnodeFnc->sxFnc.pnodeVars;

}

// Keep nested function declarations and expressions in the same list at function scope.

// (Indicate this by nulling out the current function expressions list.)

m\_ppnodeExprScope = nullptr;

if (buildAST)

{

DeferredFunctionStub \*saveCurrentStub = m\_currDeferredStub;

if (pnodeFncParent && m\_currDeferredStub)

{

m\_currDeferredStub = (m\_currDeferredStub + (pnodeFncParent->sxFnc.nestedCount - 1))->deferredStubs;

}

if (m\_token.tk != tkLCurly && fLambda)

{

if (fAsync)

{

TransformAsyncFncDeclAST(&pnodeFnc, true);

}

else

{

ParseExpressionLambdaBody<true>(pnodeFnc);

}

\*pNeedScanRCurly = false;

}

else

{

this->FinishFncDecl(pnodeFnc, pNameHint, lastNodeRef);

}

m\_currDeferredStub = saveCurrentStub;

}

else

{

this->ParseNestedDeferredFunc(pnodeFnc, fLambda, pNeedScanRCurly, &strictModeTurnedOn);

}

}

if (pnodeInnerBlock)

{

FinishParseBlock(pnodeInnerBlock, \*pNeedScanRCurly);

}

if ((buildAST || BindDeferredPidRefs()) && !(m\_token.tk != tkLCurly && fLambda))

{

this->AddArgumentsNodeToVars(pnodeFnc);

}

// Restore the lists of scopes that contain function expressions.

Assert(m\_ppnodeExprScope == nullptr || \*m\_ppnodeExprScope == nullptr);

m\_ppnodeExprScope = ppnodeExprScopeSave;

AssertMem(m\_ppnodeScope);

Assert(nullptr == \*m\_ppnodeScope);

m\_ppnodeScope = ppnodeScopeSave;

if (pnodeBlock)

{

FinishParseBlock(pnodeBlock, \*pNeedScanRCurly);

}

if (IsStrictMode() || strictModeTurnedOn)

{

this->m\_fUseStrictMode = TRUE; // Now we know this function is in strict mode

if (!fLambda && !fWasAlreadyStrictMode)

{

// If this function turned on strict mode then we didn't check the formal

// parameters or function name hint for future reserved word usage. So do that now.

// Except for lambdas which always treat formal parameters as strict and do not have

// a name.

RestorePoint afterFnc;

m\_pscan->Capture(&afterFnc);

if (\*pHasName)

{

// Rewind to the function name hint and check if the token is a reserved word.

m\_pscan->SeekTo(beginNameHint);

m\_pscan->Scan();

if (pnodeFnc->sxFnc.IsGenerator())

{

Assert(m\_token.tk == tkStar);

Assert(m\_scriptContext->GetConfig()->IsES6GeneratorsEnabled());

Assert(!(flags & fFncClassMember));

m\_pscan->Scan();

}

if (m\_token.IsReservedWord())

{

IdentifierExpectedError(m\_token);

}

CheckStrictModeEvalArgumentsUsage(m\_token.GetIdentifier(m\_phtbl));

}

// Fast forward to formal parameter list, check for future reserved words,

// then restore scanner as it was.

m\_pscan->SeekTo(beginFormals);

CheckStrictFormalParameters();

m\_pscan->SeekTo(afterFnc);

}

if (buildAST)

{

if (pnodeFnc->sxFnc.pnodeName != nullptr && knopVarDecl == pnodeFnc->sxFnc.pnodeName->nop)

{

CheckStrictModeEvalArgumentsUsage(pnodeFnc->sxFnc.pnodeName->sxVar.pid, pnodeFnc->sxFnc.pnodeName);

}

}

this->m\_fUseStrictMode = oldStrictMode;

CHAKRATEL\_LANGSTATS\_INC\_LANGFEATURECOUNT(StrictModeFunctionCount, m\_scriptContext);

}

if (fDeferred)

{

AnalysisAssert(pnodeFnc);

pnodeFnc->sxFnc.pnodeVars = nullptr;

}

if (parallelJobStarted)

{

pnodeFnc = pnodeRealFnc;

m\_currentNodeFunc = pnodeRealFnc;

// Let the foreground thread take care of marking the limit on the function node,

// because in some cases this function's caller will want to change that limit,

// so we don't want the background thread to try and touch it.

pnodeFnc->ichLim = m\_pscan->IchLimTok();

pnodeFnc->sxFnc.cbLim = m\_pscan->IecpLimTok();

}

}

// after parsing asm.js module, we want to reset asm.js state before continuing

AnalysisAssert(pnodeFnc);

if (pnodeFnc->sxFnc.GetAsmjsMode())

{

m\_InAsmMode = false;

}

// Restore the statement stack.

Assert(nullptr == m\_pstmtCur);

SetCurrentStatement(pstmtSave);

if (pnodeFncExprScope)

{

FinishParseFncExprScope(pnodeFnc, pnodeFncExprScope);

}

if (!m\_stoppedDeferredParse)

{

m\_grfscr |= uDeferSave;

}

m\_pscan->SetYieldIsKeyword(fPreviousYieldIsKeyword);

m\_pscan->SetAwaitIsKeyword(fPreviousAwaitIsKeyword);

return true;

}

void Parser::ParseTopLevelDeferredFunc(ParseNodePtr pnodeFnc, ParseNodePtr pnodeFncParent, LPCOLESTR pNameHint)

{

// Parse a function body that is a transition point from building AST to doing fast syntax check.

pnodeFnc->sxFnc.pnodeVars = nullptr;

pnodeFnc->sxFnc.pnodeBody = nullptr;

this->m\_deferringAST = TRUE;

// Put the scanner into "no hashing" mode.

BYTE deferFlags = m\_pscan->SetDeferredParse(TRUE);

m\_pscan->Scan();

ChkCurTok(tkLCurly, ERRnoLcurly);

ParseNodePtr \*ppnodeVarSave = m\_ppnodeVar;

if (BindDeferredPidRefs())

{

m\_ppnodeVar = &pnodeFnc->sxFnc.pnodeVars;

}

if (pnodeFncParent != nullptr

&& m\_currDeferredStub != nullptr

// We don't create stubs for function bodies in parameter scope.

&& pnodeFnc->sxFnc.pnodeScopes->sxBlock.blockType != PnodeBlockType::Parameter)

{

// We've already parsed this function body for syntax errors on the initial parse of the script.

// We have information that allows us to skip it, so do so.

DeferredFunctionStub \*stub = m\_currDeferredStub + (pnodeFncParent->sxFnc.nestedCount - 1);

Assert(pnodeFnc->ichMin == stub->ichMin);

if (stub->fncFlags & kFunctionCallsEval)

{

this->MarkEvalCaller();

}

if (stub->fncFlags & kFunctionChildCallsEval)

{

pnodeFnc->sxFnc.SetChildCallsEval(true);

}

if (stub->fncFlags & kFunctionHasWithStmt)

{

pnodeFnc->sxFnc.SetHasWithStmt(true);

}

PHASE\_PRINT\_TRACE1(

Js::SkipNestedDeferredPhase,

L"Skipping nested deferred function %d. %s: %d...%d\n",

pnodeFnc->sxFnc.functionId, GetFunctionName(pnodeFnc, pNameHint), pnodeFnc->ichMin, stub->restorePoint.m\_ichMinTok);

m\_pscan->SeekTo(stub->restorePoint, m\_nextFunctionId);

pnodeFnc->sxFnc.nestedCount = stub->nestedCount;

pnodeFnc->sxFnc.deferredStub = stub->deferredStubs;

if (stub->fncFlags & kFunctionStrictMode)

{

pnodeFnc->sxFnc.SetStrictMode(true);

}

}

else

{

ParseStmtList<false>(nullptr, nullptr, SM\_DeferedParse, true /\* isSourceElementList \*/);

}

pnodeFnc->ichLim = m\_pscan->IchLimTok();

pnodeFnc->sxFnc.cbLim = m\_pscan->IecpLimTok();

if (BindDeferredPidRefs())

{

m\_ppnodeVar = ppnodeVarSave;

}

// Restore the scanner's default hashing mode.

// Do this before we consume the next token.

m\_pscan->SetDeferredParseFlags(deferFlags);

ChkCurTokNoScan(tkRCurly, ERRnoRcurly);

#if DBG

pnodeFnc->sxFnc.deferredParseNextFunctionId = \*this->m\_nextFunctionId;

#endif

this->m\_deferringAST = FALSE;

}

bool Parser::DoParallelParse(ParseNodePtr pnodeFnc) const

{

#if ENABLE\_BACKGROUND\_PARSING

if (!PHASE\_ON\_RAW(Js::ParallelParsePhase, m\_sourceContextInfo->sourceContextId, pnodeFnc->sxFnc.functionId))

{

return false;

}

BackgroundParser \*bgp = m\_scriptContext->GetBackgroundParser();

return bgp != nullptr;

#else

return false;

#endif

}

bool Parser::ScanAheadToFunctionEnd(uint count)

{

bool found = false;

uint curlyDepth = 0;

RestorePoint funcStart;

m\_pscan->Capture(&funcStart);

for (uint i = 0; i < count; i++)

{

switch (m\_token.tk)

{

case tkStrTmplBegin:

case tkStrTmplMid:

case tkStrTmplEnd:

case tkDiv:

case tkAsgDiv:

case tkScanError:

case tkEOF:

goto LEnd;

case tkLCurly:

UInt32Math::Inc(curlyDepth, Parser::OutOfMemory);

break;

case tkRCurly:

if (curlyDepth == 1)

{

found = true;

goto LEnd;

}

if (curlyDepth == 0)

{

goto LEnd;

}

curlyDepth--;

break;

}

m\_pscan->ScanAhead();

}

LEnd:

m\_pscan->SeekTo(funcStart);

return found;

}

bool Parser::FastScanFormalsAndBody()

{

// The scanner is currently pointing just past the name of a function.

// The idea here is to find the end of the function body as quickly as possible,

// by tokenizing and tracking {}'s if possible.

// String templates require some extra logic but can be handled.

// The real wrinkle is "/" and "/=", which may indicate either a RegExp literal or a division, depending

// on the context.

// To handle this with minimal work, keep track of the last ";" seen at each {} depth. If we see one of the

// difficult tokens, rewind to the last ";" at the current {} depth and parse statements until we pass the

// point where we had to rewind. This will process the "/" as required.

RestorePoint funcStart;

m\_pscan->Capture(&funcStart);

const int maxRestorePointDepth = 16;

struct FastScanRestorePoint

{

RestorePoint restorePoint;

uint parenDepth;

Js::LocalFunctionId functionId;

int blockId;

FastScanRestorePoint() : restorePoint(), parenDepth(0) {};

};

FastScanRestorePoint lastSColonAtCurlyDepth[maxRestorePointDepth];

charcount\_t ichStart = m\_pscan->IchMinTok();

uint blockIdSave = m\_nextBlockId;

uint functionIdSave = \*m\_nextFunctionId;

uint curlyDepth = 0;

uint strTmplDepth = 0;

for (;;)

{

switch (m\_token.tk)

{

case tkStrTmplBegin:

UInt32Math::Inc(strTmplDepth, Parser::OutOfMemory);

// Fall through

case tkStrTmplMid:

case tkLCurly:

UInt32Math::Inc(curlyDepth, Parser::OutOfMemory);

Int32Math::Inc(m\_nextBlockId, &m\_nextBlockId);

break;

case tkStrTmplEnd:

// We can assert here, because the scanner will only return this token if we've told it we're

// in a string template.

Assert(strTmplDepth > 0);

strTmplDepth--;

break;

case tkRCurly:

if (curlyDepth == 1)

{

Assert(strTmplDepth == 0);

if (PHASE\_TRACE1(Js::ParallelParsePhase))

{

Output::Print(L"Finished fast seek: %d. %s -- %d...%d\n",

m\_currentNodeFunc->sxFnc.functionId,

GetFunctionName(m\_currentNodeFunc, m\_currentNodeFunc->sxFnc.hint),

ichStart, m\_pscan->IchLimTok());

}

return true;

}

if (curlyDepth < maxRestorePointDepth)

{

lastSColonAtCurlyDepth[curlyDepth].restorePoint.m\_ichMinTok = (uint)-1;

}

curlyDepth--;

if (strTmplDepth > 0)

{

m\_pscan->SetScanState(Scanner\_t::ScanState::ScanStateStringTemplateMiddleOrEnd);

}

break;

case tkSColon:

// Track the location of the ";" (if it's outside parens, as we don't, for instance, want

// to track the ";"'s in a for-loop header. If we find it's important to rewind within a paren

// expression, we can do something more sophisticated.)

if (curlyDepth < maxRestorePointDepth && lastSColonAtCurlyDepth[curlyDepth].parenDepth == 0)

{

m\_pscan->Capture(&lastSColonAtCurlyDepth[curlyDepth].restorePoint);

lastSColonAtCurlyDepth[curlyDepth].functionId = \*this->m\_nextFunctionId;

lastSColonAtCurlyDepth[curlyDepth].blockId = m\_nextBlockId;

}

break;

case tkLParen:

if (curlyDepth < maxRestorePointDepth)

{

UInt32Math::Inc(lastSColonAtCurlyDepth[curlyDepth].parenDepth);

}

break;

case tkRParen:

if (curlyDepth < maxRestorePointDepth)

{

Assert(lastSColonAtCurlyDepth[curlyDepth].parenDepth != 0);

lastSColonAtCurlyDepth[curlyDepth].parenDepth--;

}

break;

case tkID:

{

charcount\_t tokLength = m\_pscan->IchLimTok() - m\_pscan->IchMinTok();

// Detect the function and class keywords so we can track function ID's.

// (In fast mode, the scanner doesn't distinguish keywords and doesn't point the token

// to a PID.)

// Detect try/catch/for to increment block count for them.

switch (tokLength)

{

case 3:

if (!memcmp(m\_pscan->PchMinTok(), "try", 3) || !memcmp(m\_pscan->PchMinTok(), "for", 3))

{

Int32Math::Inc(m\_nextBlockId, &m\_nextBlockId);

}

break;

case 5:

if (!memcmp(m\_pscan->PchMinTok(), "catch", 5))

{

Int32Math::Inc(m\_nextBlockId, &m\_nextBlockId);

}

else if (!memcmp(m\_pscan->PchMinTok(), "class", 5))

{

Int32Math::Inc(m\_nextBlockId, &m\_nextBlockId);

Int32Math::Inc(\*this->m\_nextFunctionId, (int\*)this->m\_nextFunctionId);

}

break;

case 8:

if (!memcmp(m\_pscan->PchMinTok(), "function", 8))

{

// Account for the possible func expr scope or dummy block for missing {}'s around a declaration

Int32Math::Inc(m\_nextBlockId, &m\_nextBlockId);

Int32Math::Inc(\*this->m\_nextFunctionId, (int\*)this->m\_nextFunctionId);

}

break;

}

break;

}

case tkDArrow:

Int32Math::Inc(m\_nextBlockId, &m\_nextBlockId);

Int32Math::Inc(\*this->m\_nextFunctionId, (int\*)this->m\_nextFunctionId);

break;

case tkDiv:

case tkAsgDiv:

{

int opl;

OpCode nop;

tokens tkPrev = m\_pscan->m\_tkPrevious;

if ((m\_pscan->m\_phtbl->TokIsBinop(tkPrev, &opl, &nop) && nop != knopNone) ||

(m\_pscan->m\_phtbl->TokIsUnop(tkPrev, &opl, &nop) &&

nop != knopNone &&

tkPrev != tkInc &&

tkPrev != tkDec) ||

tkPrev == tkColon ||

tkPrev == tkLParen ||

tkPrev == tkLBrack ||

tkPrev == tkRETURN)

{

// Previous token indicates that we're starting an expression here and can't have a

// binary operator now.

// Assume this is a RegExp.

ParseRegExp<false>();

break;

}

uint tempCurlyDepth = curlyDepth < maxRestorePointDepth ? curlyDepth : maxRestorePointDepth - 1;

for (; tempCurlyDepth != (uint)-1; tempCurlyDepth--)

{

// We don't know whether we've got a RegExp or a divide. Rewind to the last safe ";"

// if we can and parse statements until we pass this point.

if (lastSColonAtCurlyDepth[tempCurlyDepth].restorePoint.m\_ichMinTok != -1)

{

break;

}

}

if (tempCurlyDepth != (uint)-1)

{

ParseNodePtr pnodeFncSave = m\_currentNodeFunc;

long \*pastSizeSave = m\_pCurrentAstSize;

uint \*pnestedCountSave = m\_pnestedCount;

ParseNodePtr \*ppnodeScopeSave = m\_ppnodeScope;

ParseNodePtr \*ppnodeExprScopeSave = m\_ppnodeExprScope;

ParseNodePtr pnodeFnc = CreateDummyFuncNode(true);

m\_ppnodeScope = &pnodeFnc->sxFnc.pnodeScopes;

m\_ppnodeExprScope = nullptr;

charcount\_t ichStop = m\_pscan->IchLimTok();

curlyDepth = tempCurlyDepth;

m\_pscan->SeekTo(lastSColonAtCurlyDepth[tempCurlyDepth].restorePoint);

m\_nextBlockId = lastSColonAtCurlyDepth[tempCurlyDepth].blockId;

\*this->m\_nextFunctionId = lastSColonAtCurlyDepth[tempCurlyDepth].functionId;

ParseNodePtr pnodeBlock = StartParseBlock<true>(PnodeBlockType::Function, ScopeType\_FunctionBody);

m\_pscan->Scan();

do

{

ParseStatement<false>(true);

}

while(m\_pscan->IchMinTok() < ichStop);

FinishParseBlock(pnodeBlock);

m\_currentNodeFunc = pnodeFncSave;

m\_pCurrentAstSize = pastSizeSave;

m\_pnestedCount = pnestedCountSave;

m\_ppnodeScope = ppnodeScopeSave;

m\_ppnodeExprScope = ppnodeExprScopeSave;

// We've already consumed the first token of the next statement, so just continue

// without a further scan.

continue;

}

}

// fall through to rewind to function start

case tkScanError:

case tkEOF:

// Unexpected token.

if (PHASE\_TRACE1(Js::ParallelParsePhase))

{

Output::Print(L"Failed fast seek: %d. %s -- %d...%d\n",

m\_currentNodeFunc->sxFnc.functionId,

GetFunctionName(m\_currentNodeFunc, m\_currentNodeFunc->sxFnc.hint),

ichStart, m\_pscan->IchLimTok());

}

m\_nextBlockId = blockIdSave;

\*m\_nextFunctionId = functionIdSave;

m\_pscan->SeekTo(funcStart);

return false;

}

m\_pscan->ScanNoKeywords();

}

}

ParseNodePtr Parser::CreateDummyFuncNode(bool fDeclaration)

{

// Create a dummy node and make it look like the current function declaration.

// Do this in situations where we want to parse statements without impacting

// the state of the "real" AST.

ParseNodePtr pnodeFnc = CreateNode(knopFncDecl);

pnodeFnc->sxFnc.ClearFlags();

pnodeFnc->sxFnc.SetDeclaration(fDeclaration);

pnodeFnc->sxFnc.astSize = 0;

pnodeFnc->sxFnc.pnodeName = nullptr;

pnodeFnc->sxFnc.pnodeScopes = nullptr;

pnodeFnc->sxFnc.pnodeRest = nullptr;

pnodeFnc->sxFnc.pid = nullptr;

pnodeFnc->sxFnc.hint = nullptr;

pnodeFnc->sxFnc.hintOffset = 0;

pnodeFnc->sxFnc.hintLength = 0;

pnodeFnc->sxFnc.isNameIdentifierRef = true;

pnodeFnc->sxFnc.pnodeNext = nullptr;

pnodeFnc->sxFnc.pnodeArgs = nullptr;

pnodeFnc->sxFnc.pnodeVars = nullptr;

pnodeFnc->sxFnc.funcInfo = nullptr;

pnodeFnc->sxFnc.deferredStub = nullptr;

pnodeFnc->sxFnc.nestedCount = 0;

pnodeFnc->sxFnc.SetNested(m\_currentNodeFunc != nullptr); // If there is a current function, then we're a nested function.

pnodeFnc->sxFnc.SetStrictMode(IsStrictMode()); // Inherit current strict mode -- may be overridden by the function itself if it contains a strict mode directive.

pnodeFnc->sxFnc.firstDefaultArg = 0;

m\_pCurrentAstSize = &pnodeFnc->sxFnc.astSize;

m\_currentNodeFunc = pnodeFnc;

m\_pnestedCount = &pnodeFnc->sxFnc.nestedCount;

return pnodeFnc;

}

void Parser::ParseNestedDeferredFunc(ParseNodePtr pnodeFnc, bool fLambda, bool \*pNeedScanRCurly, bool \*pStrictModeTurnedOn)

{

// Parse a function nested inside another deferred function.

size\_t lengthBeforeBody = this->GetSourceLength();

if (m\_token.tk != tkLCurly && fLambda)

{

ParseExpressionLambdaBody<false>(pnodeFnc);

\*pNeedScanRCurly = false;

}

else

{

ChkCurTok(tkLCurly, ERRnoLcurly);

bool\* detectStrictModeOn = IsStrictMode() ? nullptr : pStrictModeTurnedOn;

if (BindDeferredPidRefs())

{

m\_ppnodeVar = &m\_currentNodeDeferredFunc->sxFnc.pnodeVars;

}

ParseStmtList<false>(nullptr, nullptr, SM\_DeferedParse, true /\* isSourceElementList \*/, detectStrictModeOn);

ChkCurTokNoScan(tkRCurly, ERRnoRcurly);

}

if (BindDeferredPidRefs())

{

pnodeFnc->ichLim = m\_pscan->IchLimTok();

pnodeFnc->sxFnc.cbLim = m\_pscan->IecpLimTok();

if (\*pStrictModeTurnedOn)

{

pnodeFnc->sxFnc.SetStrictMode(true);

}

if (!PHASE\_OFF1(Js::SkipNestedDeferredPhase))

{

// Record the end of the function and the function ID increment that happens inside the function.

// Byte code gen will use this to build stub information to allow us to skip this function when the

// enclosing function is fully parsed.

RestorePoint \*restorePoint = Anew(&m\_nodeAllocator, RestorePoint);

m\_pscan->Capture(restorePoint,

\*m\_nextFunctionId - pnodeFnc->sxFnc.functionId - 1,

lengthBeforeBody - this->GetSourceLength());

pnodeFnc->sxFnc.pRestorePoint = restorePoint;

}

}

}

template<bool buildAST>

bool Parser::ParseFncNames(ParseNodePtr pnodeFnc, ParseNodePtr pnodeFncParent, ushort flags, ParseNodePtr \*\*pLastNodeRef)

{

BOOL fDeclaration = flags & fFncDeclaration;

BOOL fIsAsync = flags & fFncAsync;

ParseNodePtr pnodeT;

charcount\_t ichMinNames, ichLimNames;

// Get the names to bind to.

/\*

\* KaushiS [5/15/08]:

\* ECMAScript defines a FunctionExpression as follows:

\*

\* "function" [Identifier] ( [FormalParameterList] ) { FunctionBody }

\*

\* The function name being optional is omitted by most real world

\* code that uses a FunctionExpression to define a function. This however

\* is problematic for tools because there isn't a function name that

\* the runtime can provide.

\*

\* To fix this (primarily for the profiler), I'm adding simple, static

\* name inferencing logic to the parser. When it encounters the following

\* productions

\*

\* "var" Identifier "=" FunctionExpression

\* "var" IdentifierA.IdentifierB...Identifier "=" FunctionExpression

\* Identifier = FunctionExpression

\* "{" Identifier: FunctionExpression "}"

\*

\* it associates Identifier with the function created by the

\* FunctionExpression. This identifier is \*not\* the function's name. It

\* is ignored by the runtime and is only an additional piece of information

\* about the function (function name hint) that tools could opt to

\* surface.

\*/

m\_pscan->Scan();

// If generators are enabled then we are in a recent enough version

// that deferred parsing will create a parse node for pnodeFnc and

// it is safe to assume it is not null.

if (flags & fFncGenerator)

{

Assert(m\_scriptContext->GetConfig()->IsES6GeneratorsEnabled());

pnodeFnc->sxFnc.SetIsGenerator();

}

else if (m\_scriptContext->GetConfig()->IsES6GeneratorsEnabled() &&

m\_token.tk == tkStar &&

!(flags & fFncClassMember))

{

if (!fDeclaration)

{

bool fPreviousYieldIsKeyword = m\_pscan->SetYieldIsKeyword(!fDeclaration);

m\_pscan->Scan();

m\_pscan->SetYieldIsKeyword(fPreviousYieldIsKeyword);

}

else

{

m\_pscan->Scan();

}

pnodeFnc->sxFnc.SetIsGenerator();

}

if (fIsAsync)

{

if (pnodeFnc->sxFnc.IsGenerator())

{

Error(ERRsyntax);

}

pnodeFnc->sxFnc.SetIsAsync();

}

if (pnodeFnc)

{

pnodeFnc->sxFnc.pnodeName = nullptr;

}

if ((m\_token.tk != tkID || flags & fFncNoName)

&& (IsStrictMode() || (pnodeFnc && pnodeFnc->sxFnc.IsGenerator()) || m\_token.tk != tkYIELD || fDeclaration)) // Function expressions can have the name yield even inside generator functions

{

if (fDeclaration ||

m\_token.IsReservedWord()) // For example: var x = (function break(){});

{

IdentifierExpectedError(m\_token);

}

return false;

}

ichMinNames = m\_pscan->IchMinTok();

Assert(m\_token.tk == tkID || (m\_token.tk == tkYIELD && !fDeclaration));

if (IsStrictMode())

{

CheckStrictModeEvalArgumentsUsage(m\_token.GetIdentifier(m\_phtbl));

}

Token tokenBase = m\_token;

charcount\_t ichMinBase = m\_pscan->IchMinTok();

charcount\_t ichLimBase = m\_pscan->IchLimTok();

m\_pscan->Scan();

if (buildAST || BindDeferredPidRefs())

{

IdentPtr pidBase = tokenBase.GetIdentifier(m\_phtbl);

pnodeT = CreateDeclNode(knopVarDecl, pidBase, STFunction);

pnodeT->ichMin = ichMinBase;

pnodeT->ichLim = ichLimBase;

if (fDeclaration &&

pnodeFncParent &&

pnodeFncParent->sxFnc.pnodeName &&

pnodeFncParent->sxFnc.pnodeName->nop == knopVarDecl &&

pnodeFncParent->sxFnc.pnodeName->sxVar.pid == pidBase)

{

pnodeFncParent->sxFnc.SetNameIsHidden();

}

}

if (buildAST)

{

AnalysisAssert(pnodeFnc);

ichLimNames = pnodeT->ichLim;

AddToNodeList(&pnodeFnc->sxFnc.pnodeName, pLastNodeRef, pnodeT);

pnodeFnc->sxFnc.pnodeName->ichMin = ichMinNames;

pnodeFnc->sxFnc.pnodeName->ichLim = ichLimNames;

if (knopVarDecl == pnodeFnc->sxFnc.pnodeName->nop)

{

// Only one name (the common case).

pnodeFnc->sxFnc.pid = pnodeFnc->sxFnc.pnodeName->sxVar.pid;

}

else

{

// Multiple names. Turn the source into an IdentPtr.

pnodeFnc->sxFnc.pid = m\_phtbl->PidHashNameLen(

m\_pscan->PchBase() + ichMinNames, ichLimNames - ichMinNames);

}

if(pnodeFnc->sxFnc.pid == wellKnownPropertyPids.arguments && fDeclaration && pnodeFncParent)

{

// This function declaration (or function expression in compat modes) overrides the built-in arguments object of the

// parent function

pnodeFncParent->grfpn |= PNodeFlags::fpnArguments\_overriddenByDecl;

}

}

return true;

}

void Parser::ValidateFormals()

{

ParseFncFormals<false>(NULL, fFncNoFlgs);

// Eat the tkRParen. The ParseFncDeclHelper caller expects to see it.

m\_pscan->Scan();

}

void Parser::ValidateSourceElementList()

{

ParseStmtList<false>(NULL, NULL, SM\_NotUsed, true);

}

void Parser::UpdateOrCheckForDuplicateInFormals(IdentPtr pid, SList<IdentPtr> \*formals)

{

bool isStrictMode = IsStrictMode();

if (isStrictMode)

{

CheckStrictModeEvalArgumentsUsage(pid);

}

if (formals->Has(pid))

{

if (isStrictMode)

{

Error(ERRES5ArgSame);

}

else

{

Error(ERRFormalSame);

}

}

else

{

formals->Prepend(pid);

}

}

template<bool buildAST>

void Parser::ParseFncFormals(ParseNodePtr pnodeFnc, ushort flags)

{

// In strict mode we need to detect duplicated formals so force PID creation (unless the function should take 0 or 1 arg).

BOOL forcePid = IsStrictMode() && ((flags & (fFncNoArg | fFncOneArg)) == 0);

AutoTempForcePid autoForcePid(m\_pscan, forcePid);

// Lambda's allow single formal specified by a single binding identifier without parentheses, special case it.

if (m\_token.tk == tkID && (flags & fFncLambda))

{

if (buildAST || BindDeferredPidRefs())

{

IdentPtr pid = m\_token.GetIdentifier(m\_phtbl);

CreateVarDeclNode(pid, STFormal, false, nullptr, false);

CheckPidIsValid(pid);

m\_pscan->Scan();

if (m\_token.tk != tkDArrow)

{

Error(ERRsyntax, m\_pscan->IchMinTok(), m\_pscan->IchLimTok());

}

return;

}

}

// Otherwise, must have a parameter list within parens.

ChkCurTok(tkLParen, ERRnoLparen);

// Now parse the list of arguments, if present

Assert((flags & (fFncNoArg | fFncOneArg)) != (fFncNoArg | fFncOneArg)); // fFncNoArg and fFncOneArg can never be at same time.

if (m\_token.tk == tkRParen)

{

if (flags & fFncOneArg)

{

Error(ERRSetterMustHaveOneArgument);

}

}

else

{

if (flags & fFncNoArg)

{

Error(ERRnoRparen); //enforce no arguments

// No recovery necessary since this is a semantic, not structural, error

}

SList<IdentPtr> formals(&m\_nodeAllocator);

ParseNodePtr pnodeT = nullptr;

bool seenRestParameter = false;

bool isNonSimpleParameterList = false;

for (Js::ArgSlot argPos = 0; ; ++argPos)

{

bool isBindingPattern = false;

if (m\_scriptContext->GetConfig()->IsES6RestEnabled() && m\_token.tk == tkEllipsis)

{

// Possible rest parameter

m\_pscan->Scan();

seenRestParameter = true;

}

if (m\_token.tk != tkID)

{

if (IsES6DestructuringEnabled() && IsPossiblePatternStart())

{

ParseNodePtr \*const ppnodeVarSave = m\_ppnodeVar;

m\_ppnodeVar = &pnodeFnc->sxFnc.pnodeVars;

ParseNodePtr paramPattern = nullptr;

ParseNodePtr pnodePattern = ParseDestructuredLiteral<buildAST>(tkLET, true /\*isDecl\*/, false /\*topLevel\*/);

if (buildAST)

{

// Instead of passing the STFormal all the way on many methods, it seems it is better to change the symbol type afterward.

Parser::MapBindIdentifier(pnodePattern, [&](ParseNodePtr item) {

Assert(item->IsVarLetOrConst());

UpdateOrCheckForDuplicateInFormals(item->sxVar.pid, &formals);

item->sxVar.sym->SetSymbolType(STFormal);

});

Assert(pnodePattern->IsPattern() || pnodePattern->nop == knopAsg);

}

m\_ppnodeVar = ppnodeVarSave;

if (buildAST)

{

paramPattern = CreateParamPatternNode(pnodePattern);

// Linking the current formal parameter (which is pattern parameter) with other formals.

\*m\_ppnodeVar = paramPattern;

paramPattern->sxParamPattern.pnodeNext = nullptr;

m\_ppnodeVar = &paramPattern->sxParamPattern.pnodeNext;

m\_currentNodeFunc->sxFnc.SetHasDestructuringPattern();

}

isBindingPattern = true;

isNonSimpleParameterList = true;

}

else

{

IdentifierExpectedError(m\_token);

}

}

if (!isBindingPattern)

{

if (seenRestParameter)

{

if (flags & fFncSetter)

{

// The parameter of a setter cannot be a rest parameter.

Error(ERRUnexpectedEllipsis);

}

if (buildAST || BindDeferredPidRefs())

{

pnodeT = CreateDeclNode(knopVarDecl, m\_token.GetIdentifier(m\_phtbl), STFormal, false);

pnodeT->sxVar.sym->SetIsNonSimpleParameter(true);

if (buildAST)

{

// When only validating formals, we won't have a function node.

pnodeFnc->sxFnc.pnodeRest = pnodeT;

}

if (!isNonSimpleParameterList)

{

// This is the first non-simple parameter we've seen. We need to go back

// and set the Symbols of all previous parameters.

MapFormalsWithoutRest(m\_currentNodeFunc, [&](ParseNodePtr pnodeArg) { pnodeArg->sxVar.sym->SetIsNonSimpleParameter(true); });

}

}

else

{

isNonSimpleParameterList = true;

}

}

else

{

if (buildAST || BindDeferredPidRefs())

{

pnodeT = CreateVarDeclNode(m\_token.GetIdentifier(m\_phtbl), STFormal, false, nullptr, false);

if (isNonSimpleParameterList)

{

pnodeT->sxVar.sym->SetIsNonSimpleParameter(true);

}

}

}

if (buildAST && m\_token.GetIdentifier(m\_phtbl) == wellKnownPropertyPids.arguments)

{

// This formal parameter overrides the built-in 'arguments' object

m\_currentNodeFunc->grfpn |= PNodeFlags::fpnArguments\_overriddenByDecl;

}

if (IsStrictMode() || isNonSimpleParameterList)

{

IdentPtr pid = m\_token.GetIdentifier(m\_phtbl);

UpdateOrCheckForDuplicateInFormals(pid, &formals);

}

m\_pscan->Scan();

if (seenRestParameter && m\_token.tk != tkRParen && m\_token.tk != tkAsg)

{

Error(ERRRestLastArg);

}

if (flags & fFncOneArg)

{

if (m\_token.tk != tkRParen)

{

Error(ERRSetterMustHaveOneArgument);

}

break; //enforce only one arg

}

if (m\_token.tk == tkAsg && m\_scriptContext->GetConfig()->IsES6DefaultArgsEnabled())

{

if (seenRestParameter && m\_scriptContext->GetConfig()->IsES6RestEnabled())

{

Error(ERRRestWithDefault);

}

m\_pscan->Scan();

ParseNodePtr pnodeInit = ParseExpr<buildAST>(koplCma);

if (buildAST || BindDeferredPidRefs())

{

AnalysisAssert(pnodeT);

pnodeT->sxVar.sym->SetIsNonSimpleParameter(true);

if (!isNonSimpleParameterList)

{

// This is the first non-simple parameter we've seen. We need to go back

// and set the Symbols of all previous parameters.

MapFormalsWithoutRest(m\_currentNodeFunc, [&](ParseNodePtr pnodeArg) { pnodeArg->sxVar.sym->SetIsNonSimpleParameter(true); });

// There may be previous parameters that need to be checked for duplicates.

isNonSimpleParameterList = true;

}

}

if (buildAST)

{

if (!m\_currentNodeFunc->sxFnc.HasDefaultArguments())

{

m\_currentNodeFunc->sxFnc.SetHasDefaultArguments();

m\_currentNodeFunc->sxFnc.firstDefaultArg = argPos;

CHAKRATEL\_LANGSTATS\_INC\_LANGFEATURECOUNT(DefaultArgFunctionCount, m\_scriptContext);

}

pnodeT->sxVar.pnodeInit = pnodeInit;

pnodeT->ichLim = m\_pscan->IchLimTok();

}

}

}

if (m\_token.tk != tkComma)

{

break;

}

m\_pscan->Scan();

if (m\_token.tk == tkRParen && m\_scriptContext->GetConfig()->IsES7TrailingCommaEnabled())

{

break;

}

}

if (seenRestParameter)

{

CHAKRATEL\_LANGSTATS\_INC\_LANGFEATURECOUNT(RestCount, m\_scriptContext);

}

if (m\_token.tk != tkRParen)

{

Error(ERRnoRparen);

}

}

Assert(m\_token.tk == tkRParen);

}

template<bool buildAST>

ParseNodePtr Parser::GenerateEmptyConstructor(bool extends)

{

ParseNodePtr pnodeFnc;

if (buildAST || BindDeferredPidRefs())

{

// Create the node.

pnodeFnc = CreateNode(knopFncDecl);

pnodeFnc->sxFnc.ClearFlags();

pnodeFnc->sxFnc.SetNested(NULL != m\_currentNodeFunc);

pnodeFnc->sxFnc.SetStrictMode();

pnodeFnc->sxFnc.SetDeclaration(TRUE);

pnodeFnc->sxFnc.SetIsMethod(TRUE);

pnodeFnc->sxFnc.SetIsClassMember(TRUE);

pnodeFnc->sxFnc.SetIsClassConstructor(TRUE);

pnodeFnc->sxFnc.SetIsBaseClassConstructor(!extends);

pnodeFnc->sxFnc.SetHasNonThisStmt(extends);

pnodeFnc->sxFnc.SetIsGeneratedDefault(TRUE);

pnodeFnc->ichLim = m\_pscan->IchLimTok();

pnodeFnc->ichMin = m\_pscan->IchMinTok();

pnodeFnc->sxFnc.cbLim = m\_pscan->IecpLimTok();

pnodeFnc->sxFnc.cbMin = m\_pscan->IecpMinTok();

pnodeFnc->sxFnc.astSize = 0;

pnodeFnc->sxFnc.lineNumber = m\_pscan->LineCur();

pnodeFnc->sxFnc.functionId = (\*m\_nextFunctionId);

pnodeFnc->sxFnc.pid = nullptr;

pnodeFnc->sxFnc.hint = nullptr;

pnodeFnc->sxFnc.hintOffset = 0;

pnodeFnc->sxFnc.hintLength = 0;

pnodeFnc->sxFnc.isNameIdentifierRef = true;

pnodeFnc->sxFnc.pnodeName = nullptr;

pnodeFnc->sxFnc.pnodeScopes = nullptr;

pnodeFnc->sxFnc.pnodeArgs = nullptr;

pnodeFnc->sxFnc.pnodeVars = nullptr;

pnodeFnc->sxFnc.pnodeBody = nullptr;

pnodeFnc->sxFnc.nestedCount = 0;

pnodeFnc->sxFnc.pnodeNext = nullptr;

pnodeFnc->sxFnc.pnodeRest = nullptr;

pnodeFnc->sxFnc.deferredStub = nullptr;

pnodeFnc->sxFnc.funcInfo = nullptr;

#ifdef DBG

pnodeFnc->sxFnc.deferredParseNextFunctionId = \*(this->m\_nextFunctionId);

#endif

AppendFunctionToScopeList(true, pnodeFnc);

}

if (m\_nextFunctionId)

{

(\*m\_nextFunctionId)++;

}

// Update the count of functions nested in the current parent.

if (m\_pnestedCount)

{

(\*m\_pnestedCount)++;

}

if (!buildAST)

{

return NULL;

}

if (m\_pscan->IchMinTok() >= m\_pscan->IchMinLine())

{

// In scenarios involving defer parse IchMinLine() can be incorrect for the first line after defer parse

pnodeFnc->sxFnc.columnNumber = m\_pscan->IchMinTok() - m\_pscan->IchMinLine();

}

else if (m\_currentNodeFunc)

{

// For the first line after defer parse, compute the column relative to the column number

// of the lexically parent function.

ULONG offsetFromCurrentFunction = m\_pscan->IchMinTok() - m\_currentNodeFunc->ichMin;

pnodeFnc->sxFnc.columnNumber = m\_currentNodeFunc->sxFnc.columnNumber + offsetFromCurrentFunction;

}

else

{

// if there is no current function, lets give a default of 0.

pnodeFnc->sxFnc.columnNumber = 0;

}

long \* pAstSizeSave = m\_pCurrentAstSize;

m\_pCurrentAstSize = &(pnodeFnc->sxFnc.astSize);

// Make this the current function.

ParseNodePtr pnodeFncSave = m\_currentNodeFunc;

m\_currentNodeFunc = pnodeFnc;

ParseNodePtr pnodeBlock = StartParseBlock<buildAST>(PnodeBlockType::Parameter, ScopeType\_Parameter);

ParseNodePtr pnodeInnerBlock = StartParseBlock<buildAST>(PnodeBlockType::Function, ScopeType\_FunctionBody);

pnodeBlock->sxBlock.pnodeScopes = pnodeInnerBlock;

pnodeFnc->sxFnc.pnodeBodyScope = pnodeInnerBlock;

pnodeFnc->sxFnc.pnodeScopes = pnodeBlock;

ParseNodePtr \*lastNodeRef = nullptr;

if (extends)

{

// constructor() { super(...arguments); } (equivalent to constructor(...args) { super(...args); } )

PidRefStack \*ref = this->PushPidRef(wellKnownPropertyPids.arguments);

ParseNodePtr argumentsId = CreateNameNode(wellKnownPropertyPids.arguments, pnodeFnc->ichMin, pnodeFnc->ichLim);

argumentsId->sxPid.symRef = ref->GetSymRef();

pnodeFnc->sxFnc.SetUsesArguments(true);

pnodeFnc->sxFnc.SetHasReferenceableBuiltInArguments(true);

ParseNodePtr \*const ppnodeVarSave = m\_ppnodeVar;

m\_ppnodeVar = &pnodeFnc->sxFnc.pnodeVars;

CreateVarDeclNode(wellKnownPropertyPids.arguments, STVariable, true, pnodeFnc)->grfpn |= PNodeFlags::fpnArguments;

m\_ppnodeVar = ppnodeVarSave;

ParseNodePtr spreadArg = CreateUniNode(knopEllipsis, argumentsId, pnodeFnc->ichMin, pnodeFnc->ichLim);

ParseNodePtr superRef = CreateNodeWithScanner<knopSuper>();

pnodeFnc->sxFnc.SetHasSuperReference(TRUE);

ParseNodePtr callNode = CreateCallNode(knopCall, superRef, spreadArg);

callNode->sxCall.spreadArgCount = 1;

AddToNodeList(&pnodeFnc->sxFnc.pnodeBody, &lastNodeRef, callNode);

}

AddToNodeList(&pnodeFnc->sxFnc.pnodeBody, &lastNodeRef, CreateNodeWithScanner<knopEndCode>());

FinishParseBlock(pnodeInnerBlock);

FinishParseBlock(pnodeBlock);

m\_currentNodeFunc = pnodeFncSave;

m\_pCurrentAstSize = pAstSizeSave;

return pnodeFnc;

}

template<bool buildAST>

void Parser::ParseExpressionLambdaBody(ParseNodePtr pnodeLambda)

{

ParseNodePtr \*lastNodeRef = nullptr;

// The lambda body is a single expression, the result of which is the return value.

ParseNodePtr pnodeRet = nullptr;

if (buildAST)

{

pnodeRet = CreateNodeWithScanner<knopReturn>();

pnodeRet->grfpn |= PNodeFlags::fpnSyntheticNode;

pnodeLambda->sxFnc.pnodeScopes->sxBlock.pnodeStmt = pnodeRet;

}

ParseNodePtr result = ParseExpr<buildAST>(koplAsg, nullptr, TRUE, FALSE, nullptr);

if (buildAST)

{

pnodeRet->sxReturn.pnodeExpr = result;

pnodeRet->ichMin = pnodeRet->sxReturn.pnodeExpr->ichMin;

pnodeRet->ichLim = pnodeRet->sxReturn.pnodeExpr->ichLim;

// Pushing a statement node with PushStmt<>() normally does this initialization

// but do it here manually since we know there is no outer statement node.

pnodeRet->sxStmt.grfnop = 0;

pnodeRet->sxStmt.pnodeOuter = nullptr;

pnodeLambda->ichLim = pnodeRet->ichLim;

pnodeLambda->sxFnc.cbLim = m\_pscan->IecpLimTokPrevious();

pnodeLambda->sxFnc.pnodeScopes->ichLim = pnodeRet->ichLim;

pnodeLambda->sxFnc.pnodeBody = nullptr;

AddToNodeList(&pnodeLambda->sxFnc.pnodeBody, &lastNodeRef, pnodeLambda->sxFnc.pnodeScopes);

// Append an EndCode node.

ParseNodePtr end = CreateNodeWithScanner<knopEndCode>(pnodeRet->ichLim);

end->ichLim = end->ichMin; // make end code zero width at the immediate end of lambda body

AddToNodeList(&pnodeLambda->sxFnc.pnodeBody, &lastNodeRef, end);

// Lambda's do not have arguments binding

pnodeLambda->sxFnc.SetHasReferenceableBuiltInArguments(false);

}

}

void Parser::CheckStrictFormalParameters()

{

Assert(m\_token.tk == tkLParen);

m\_pscan->ScanForcingPid();

if (m\_token.tk != tkRParen)

{

SList<IdentPtr> formals(&m\_nodeAllocator);

for (;;)

{

if (m\_token.tk != tkID)

{

IdentifierExpectedError(m\_token);

}

IdentPtr pid = m\_token.GetIdentifier(m\_phtbl);

CheckStrictModeEvalArgumentsUsage(pid);

if (formals.Has(pid))

{

Error(ERRES5ArgSame, m\_pscan->IchMinTok(), m\_pscan->IchLimTok());

}

else

{

formals.Prepend(pid);

}

m\_pscan->Scan();

if (m\_token.tk == tkAsg && m\_scriptContext->GetConfig()->IsES6DefaultArgsEnabled())

{

m\_pscan->Scan();

// We can avoid building the AST since we are just checking the default expression.

ParseNodePtr pnodeInit = ParseExpr<false>(koplCma);

Assert(pnodeInit == nullptr);

}

if (m\_token.tk != tkComma)

{

break;

}

m\_pscan->ScanForcingPid();

if (m\_token.tk == tkRParen && m\_scriptContext->GetConfig()->IsES7TrailingCommaEnabled())

{

break;

}

}

}

Assert(m\_token.tk == tkRParen);

}

void Parser::FinishFncNode(ParseNodePtr pnodeFnc)

{

AnalysisAssert(pnodeFnc);

// Finish the AST for a function that was deferred earlier, but which we decided

// to finish after the fact.

// We assume that the name(s) and arg(s) have already got parse nodes, so

// we just have to do the function body.

// Save the current next function Id, and resume from the old one.

Js::LocalFunctionId \* nextFunctionIdSave = m\_nextFunctionId;

Js::LocalFunctionId tempNextFunctionId = pnodeFnc->sxFnc.functionId + 1;

this->m\_nextFunctionId = &tempNextFunctionId;

ParseNodePtr pnodeFncSave = m\_currentNodeFunc;

uint \*pnestedCountSave = m\_pnestedCount;

long\* pAstSizeSave = m\_pCurrentAstSize;

m\_currentNodeFunc = pnodeFnc;

m\_pCurrentAstSize = & (pnodeFnc->sxFnc.astSize);

pnodeFnc->sxFnc.nestedCount = 0;

m\_pnestedCount = &pnodeFnc->sxFnc.nestedCount;

// Cue up the parser to the start of the function body.

if (pnodeFnc->sxFnc.pnodeName)

{

// Skip the name(s).

m\_pscan->SetCurrentCharacter(pnodeFnc->sxFnc.pnodeName->ichLim, pnodeFnc->sxFnc.lineNumber);

}

else

{

m\_pscan->SetCurrentCharacter(pnodeFnc->ichMin, pnodeFnc->sxFnc.lineNumber);

if (pnodeFnc->sxFnc.IsAccessor())

{

// Getter/setter. The node text starts with the name, so eat that.

m\_pscan->ScanNoKeywords();

}

else

{

// Anonymous function. Skip any leading "("'s and "function".

for (;;)

{

m\_pscan->Scan();

if (m\_token.tk == tkFUNCTION)

{

break;

}

Assert(m\_token.tk == tkLParen || m\_token.tk == tkStar);

}

}

}

// switch scanner to treat 'yield' as keyword in generator functions

// or as an identifier in non-generator functions

bool fPreviousYieldIsKeyword = m\_pscan->SetYieldIsKeyword(pnodeFnc && pnodeFnc->sxFnc.IsGenerator());

bool fPreviousAwaitIsKeyword = m\_pscan->SetAwaitIsKeyword(pnodeFnc && pnodeFnc->sxFnc.IsAsync());

// Skip the arg list.

m\_pscan->ScanNoKeywords();

if (m\_token.tk == tkStar)

{

Assert(pnodeFnc->sxFnc.IsGenerator());

m\_pscan->ScanNoKeywords();

}

Assert(m\_token.tk == tkLParen);

m\_pscan->ScanNoKeywords();

if (m\_token.tk != tkRParen)

{

for (;;)

{

if (m\_token.tk == tkEllipsis)

{

m\_pscan->ScanNoKeywords();

}

if (m\_token.tk == tkID)

{

m\_pscan->ScanNoKeywords();

if (m\_token.tk == tkAsg)

{

// Eat the default expression

m\_pscan->ScanNoKeywords();

ParseExpr<false>(koplCma);

}

}

else if (IsPossiblePatternStart())

{

ParseDestructuredLiteralWithScopeSave(tkLET, false/\*isDecl\*/, false /\*topLevel\*/);

}

else

{

AssertMsg(false, "Unexpected identifier prefix while fast-scanning formals");

}

if (m\_token.tk != tkComma)

{

break;

}

m\_pscan->ScanNoKeywords();

if (m\_token.tk == tkRParen && m\_scriptContext->GetConfig()->IsES7TrailingCommaEnabled())

{

break;

}

}

}

if (m\_token.tk == tkRParen) // This might be false due to a lambda => token.

{

m\_pscan->Scan();

}

// Finish the function body.

{

// Note that in IE8- modes, surrounding parentheses are considered part of function body. e.g. "( function x(){} )".

// We lose that context here since we start from middle of function body. So save and restore source range info.

ParseNodePtr\* lastNodeRef = NULL;

const charcount\_t ichLim = pnodeFnc->ichLim;

const size\_t cbLim = pnodeFnc->sxFnc.cbLim;

this->FinishFncDecl(pnodeFnc, NULL, lastNodeRef);

#if DBG

// The pnode extent may not match the original extent.

// We expect this to happen only when there are trailing ")"'s.

// Consume them and make sure that's all we've got.

if (pnodeFnc->ichLim != ichLim)

{

Assert(pnodeFnc->ichLim < ichLim);

m\_pscan->SetCurrentCharacter(pnodeFnc->ichLim);

while (m\_pscan->IchLimTok() != ichLim)

{

m\_pscan->ScanNoKeywords();

Assert(m\_token.tk == tkRParen);

}

}

#endif

pnodeFnc->ichLim = ichLim;

pnodeFnc->sxFnc.cbLim = cbLim;

}

m\_currentNodeFunc = pnodeFncSave;

m\_pCurrentAstSize = pAstSizeSave;

m\_pnestedCount = pnestedCountSave;

Assert(m\_pnestedCount);

Assert(tempNextFunctionId == pnodeFnc->sxFnc.deferredParseNextFunctionId);

this->m\_nextFunctionId = nextFunctionIdSave;

m\_pscan->SetYieldIsKeyword(fPreviousYieldIsKeyword);

m\_pscan->SetAwaitIsKeyword(fPreviousAwaitIsKeyword);

}

void Parser::FinishFncDecl(ParseNodePtr pnodeFnc, LPCOLESTR pNameHint, ParseNodePtr \*lastNodeRef)

{

LPCOLESTR name = NULL;

JS\_ETW(long startAstSize = \*m\_pCurrentAstSize);

if(IS\_JS\_ETW(EventEnabledJSCRIPT\_PARSE\_METHOD\_START()) || PHASE\_TRACE1(Js::DeferParsePhase))

{

name = GetFunctionName(pnodeFnc, pNameHint);

m\_functionBody = NULL; // for nested functions we do not want to get the name of the top deferred function return name;

JS\_ETW(EventWriteJSCRIPT\_PARSE\_METHOD\_START(m\_sourceContextInfo->dwHostSourceContext, GetScriptContext(), pnodeFnc->sxFnc.functionId, 0, m\_parseType, name));

OUTPUT\_TRACE(Js::DeferParsePhase, L"Parsing function (%s) : %s (%d)\n", GetParseType(), name, pnodeFnc->sxFnc.functionId);

}

JS\_ETW(EventWriteJSCRIPT\_PARSE\_FUNC(GetScriptContext(), pnodeFnc->sxFnc.functionId, /\*Undefer\*/FALSE));

// Do the work of creating an AST for a function body.

// This is common to the un-deferred case and the case in which we un-defer late in the game.

Assert(pnodeFnc->nop == knopFncDecl);

ChkCurTok(tkLCurly, ERRnoLcurly);

if (pnodeFnc->sxFnc.IsAsync())

{

TransformAsyncFncDeclAST(&pnodeFnc->sxFnc.pnodeBody, false);

}

else

{

ParseStmtList<true>(&pnodeFnc->sxFnc.pnodeBody, &lastNodeRef, SM\_OnFunctionCode, true /\* isSourceElementList \*/);

// Append an EndCode node.

AddToNodeList(&pnodeFnc->sxFnc.pnodeBody, &lastNodeRef, CreateNodeWithScanner<knopEndCode>());

}

ChkCurTokNoScan(tkRCurly, ERRnoRcurly);

pnodeFnc->ichLim = m\_pscan->IchLimTok();

pnodeFnc->sxFnc.cbLim = m\_pscan->IecpLimTok();

// Restore the lists of scopes that contain function expressions.

// Save the temps and restore the outer scope's list.

// NOTE: Eze makes no use of this.

//pnodeFnc->sxFnc.pnodeTmps = \*m\_ppnodeVar;

#ifdef ENABLE\_JS\_ETW

long astSize = \*m\_pCurrentAstSize - startAstSize;

EventWriteJSCRIPT\_PARSE\_METHOD\_STOP(m\_sourceContextInfo->dwHostSourceContext, GetScriptContext(), pnodeFnc->sxFnc.functionId, astSize, m\_parseType, name);

#endif

}

void Parser::AddArgumentsNodeToVars(ParseNodePtr pnodeFnc)

{

if((pnodeFnc->grfpn & PNodeFlags::fpnArguments\_overriddenByDecl) || pnodeFnc->sxFnc.IsLambda())

{

// In any of the following cases, there is no way to reference the built-in 'arguments' variable (in the order of checks

// above):

// - A function parameter is named 'arguments'

// - There is a nested function declaration (or named function expression in compat modes) named 'arguments'

// - In compat modes, the function is named arguments, does not have a var declaration named 'arguments', and does

// not call 'eval'

pnodeFnc->sxFnc.SetHasReferenceableBuiltInArguments(false);

}

else

{

if(m\_ppnodeVar == &pnodeFnc->sxFnc.pnodeVars)

{

// There were no var declarations in the function

CreateVarDeclNode(wellKnownPropertyPids.arguments, STVariable, true, pnodeFnc)->grfpn |= PNodeFlags::fpnArguments;

}

else

{

// There were var declarations in the function, so insert an 'arguments' local at the beginning of the var list.

// This is done because the built-in 'arguments' variable overrides an 'arguments' var declaration until the

// 'arguments' variable is assigned. By putting our built-in var declaration at the beginning, an 'arguments'

// identifier will resolve to this symbol, which has the fpnArguments flag set, and will be the built-in arguments

// object until it is replaced with something else.

ParseNodePtr \*const ppnodeVarSave = m\_ppnodeVar;

m\_ppnodeVar = &pnodeFnc->sxFnc.pnodeVars;

CreateVarDeclNode(wellKnownPropertyPids.arguments, STVariable, true, pnodeFnc)->grfpn |= PNodeFlags::fpnArguments;

m\_ppnodeVar = ppnodeVarSave;

}

pnodeFnc->sxFnc.SetHasReferenceableBuiltInArguments(true);

}

}

LPCOLESTR Parser::GetFunctionName(ParseNodePtr pnodeFnc, LPCOLESTR pNameHint)

{

LPCOLESTR name = nullptr;

if(pnodeFnc->sxFnc.pnodeName != nullptr && knopVarDecl == pnodeFnc->sxFnc.pnodeName->nop)

{

name = pnodeFnc->sxFnc.pnodeName->sxVar.pid->Psz();

}

if(name == nullptr && pNameHint != nullptr)

{

name = pNameHint;

}

if(name == nullptr && m\_functionBody != nullptr)

{

name = m\_functionBody->GetExternalDisplayName();

}

else if(name == nullptr)

{

name = Js::Constants::AnonymousFunction;

}

return name;

}

IdentPtr Parser::ParseClassPropertyName(IdentPtr \* pidHint)

{

if (m\_token.tk == tkID || m\_token.tk == tkStrCon || m\_token.IsReservedWord())

{

IdentPtr pid;

if (m\_token.tk == tkStrCon)

{

if (m\_pscan->IsOctOrLeadingZeroOnLastTKNumber())

{

Error(ERRES5NoOctal);

}

pid = m\_token.GetStr();

}

else

{

pid = m\_token.GetIdentifier(m\_phtbl);

}

\*pidHint = pid;

return pid;

}

else if (m\_token.tk == tkIntCon)

{

if (m\_pscan->IsOctOrLeadingZeroOnLastTKNumber())

{

Error(ERRES5NoOctal);

}

return m\_pscan->PidFromLong(m\_token.GetLong());

}

else if (m\_token.tk == tkFltCon)

{

if (m\_pscan->IsOctOrLeadingZeroOnLastTKNumber())

{

Error(ERRES5NoOctal);

}

return m\_pscan->PidFromDbl(m\_token.GetDouble());

}

Error(ERRnoMemberIdent);

}

LPCOLESTR Parser::ConstructFinalHintNode(IdentPtr pClassName, IdentPtr pMemberName, IdentPtr pGetSet, bool isStatic, ulong\* nameLength, ulong\* pShortNameOffset, bool isComputedName, LPCOLESTR pMemberNameHint)

{

if ((pMemberName == nullptr && !isComputedName) ||

(pMemberNameHint == nullptr && isComputedName) ||

!CONFIG\_FLAG(UseFullName))

{

return nullptr;

}

LPCOLESTR pFinalName = isComputedName? pMemberNameHint : pMemberName->Psz();

ulong fullNameHintLength = 0;

ulong shortNameOffset = 0;

if (!isStatic)

{

// Add prototype.

pFinalName = AppendNameHints(wellKnownPropertyPids.prototype, pFinalName, &fullNameHintLength, &shortNameOffset);

}

if (pClassName)

{

pFinalName = AppendNameHints(pClassName, pFinalName, &fullNameHintLength, &shortNameOffset);

}

if (pGetSet)

{

if (m\_scriptContext->GetConfig()->IsES6FunctionNameEnabled())

{

// displays as get/set prototype.funcname

ulong getSetOffset = 0;

pFinalName = AppendNameHints(pGetSet, pFinalName, &fullNameHintLength, &getSetOffset, true);

shortNameOffset += getSetOffset;

}

else

{

pFinalName = AppendNameHints(pFinalName, pGetSet, &fullNameHintLength, &shortNameOffset);

}

}

if (fullNameHintLength > \*nameLength)

{

\*nameLength = fullNameHintLength;

}

if (shortNameOffset > \*pShortNameOffset)

{

\*pShortNameOffset = shortNameOffset;

}

return pFinalName;

}

class AutoParsingSuperRestrictionStateRestorer

{

public:

AutoParsingSuperRestrictionStateRestorer(Parser\* parser) : m\_parser(parser)

{

AssertMsg(this->m\_parser != nullptr, "This just should not happen");

this->m\_originalParsingSuperRestrictionState = this->m\_parser->m\_parsingSuperRestrictionState;

}

~AutoParsingSuperRestrictionStateRestorer()

{

AssertMsg(this->m\_parser != nullptr, "This just should not happen");

this->m\_parser->m\_parsingSuperRestrictionState = m\_originalParsingSuperRestrictionState;

}

private:

Parser\* m\_parser;

int m\_originalParsingSuperRestrictionState;

};

template<bool buildAST>

ParseNodePtr Parser::ParseClassDecl(BOOL isDeclaration, LPCOLESTR pNameHint, ulong \*pHintLength, ulong \*pShortNameOffset)

{

bool hasConstructor = false;

bool hasExtends = false;

IdentPtr name = nullptr;

ParseNodePtr pnodeName = nullptr;

ParseNodePtr pnodeConstructor = nullptr;

ParseNodePtr pnodeExtends = nullptr;

ParseNodePtr pnodeMembers = nullptr;

ParseNodePtr \*lastMemberNodeRef = nullptr;

ParseNodePtr pnodeStaticMembers = nullptr;

ParseNodePtr \*lastStaticMemberNodeRef = nullptr;

ulong nameHintLength = pHintLength ? \*pHintLength : 0;

ulong nameHintOffset = pShortNameOffset ? \*pShortNameOffset : 0;

ArenaAllocator tempAllocator(L"ClassMemberNames", m\_nodeAllocator.GetPageAllocator(), Parser::OutOfMemory);

ParseNodePtr pnodeClass = nullptr;

if (buildAST)

{

pnodeClass = CreateNode(knopClassDecl);

CHAKRATEL\_LANGSTATS\_INC\_LANGFEATURECOUNT(ClassCount, m\_scriptContext);

}

m\_pscan->Scan();

if (m\_token.tk == tkID)

{

name = m\_token.GetIdentifier(m\_phtbl);

m\_pscan->Scan();

}

else if (isDeclaration)

{

IdentifierExpectedError(m\_token);

}

if (isDeclaration && name == wellKnownPropertyPids.arguments && GetCurrentBlockInfo()->pnodeBlock->sxBlock.blockType == Function)

{

GetCurrentFunctionNode()->grfpn |= PNodeFlags::fpnArguments\_overriddenByDecl;

}

BOOL strictSave = m\_fUseStrictMode;

m\_fUseStrictMode = TRUE;

if (m\_token.tk == tkEXTENDS)

{

m\_pscan->Scan();

pnodeExtends = ParseExpr<buildAST>();

hasExtends = true;

}

if (m\_token.tk != tkLCurly)

{

Error(ERRnoLcurly);

}

OUTPUT\_TRACE\_DEBUGONLY(Js::ES6VerboseFlag, L"Parsing class (%s) : %s\n", GetParseType(), name ? name->Psz() : L"anonymous class");

ParseNodePtr pnodeDeclName = nullptr;

if (isDeclaration)

{

pnodeDeclName = CreateBlockScopedDeclNode(name, knopLetDecl);

}

ParseNodePtr \*ppnodeScopeSave = nullptr;

ParseNodePtr \*ppnodeExprScopeSave = nullptr;

ParseNodePtr pnodeBlock = StartParseBlock<buildAST>(PnodeBlockType::Regular, ScopeType\_Block);

if (buildAST)

{

PushFuncBlockScope(pnodeBlock, &ppnodeScopeSave, &ppnodeExprScopeSave);

pnodeClass->sxClass.pnodeBlock = pnodeBlock;

}

if (name)

{

pnodeName = CreateBlockScopedDeclNode(name, knopConstDecl);

}

RestorePoint beginClass;

m\_pscan->Capture(&beginClass);

m\_pscan->ScanForcingPid();

IdentPtr pClassNamePid = pnodeName ? pnodeName->sxVar.pid : nullptr;

for (;;)

{

if (m\_token.tk == tkSColon)

{

m\_pscan->ScanForcingPid();

continue;

}

if (m\_token.tk == tkRCurly)

{

break;

}

bool isStatic = m\_token.tk == tkSTATIC;

if (isStatic)

{

m\_pscan->ScanForcingPid();

}

ushort fncDeclFlags = fFncNoName | fFncMethod | fFncClassMember;

charcount\_t ichMin = 0;

size\_t iecpMin = 0;

ParseNodePtr pnodeMemberName = nullptr;

IdentPtr pidHint = nullptr;

IdentPtr memberPid = nullptr;

LPCOLESTR pMemberNameHint = nullptr;

ulong memberNameHintLength = 0;

ulong memberNameOffset = 0;

bool isComputedName = false;

bool isAsyncMethod = false;

if (m\_token.tk == tkID && m\_token.GetIdentifier(m\_phtbl) == wellKnownPropertyPids.async && m\_scriptContext->GetConfig()->IsES7AsyncAndAwaitEnabled())

{

RestorePoint parsedAsync;

m\_pscan->Capture(&parsedAsync);

ichMin = m\_pscan->IchMinTok();

iecpMin = m\_pscan->IecpMinTok();

m\_pscan->Scan();

if (m\_token.tk == tkLParen)

{

m\_pscan->SeekTo(parsedAsync);

}

else

{

isAsyncMethod = true;

}

}

bool isGenerator = m\_scriptContext->GetConfig()->IsES6GeneratorsEnabled() &&

m\_token.tk == tkStar;

if (isGenerator)

{

fncDeclFlags |= fFncGenerator;

m\_pscan->ScanForcingPid();

}

if (m\_token.tk == tkLBrack && m\_scriptContext->GetConfig()->IsES6ObjectLiteralsEnabled())

{

// Computed member name: [expr] () { }

LPCOLESTR emptyHint = nullptr;

ParseComputedName<buildAST>(&pnodeMemberName, &emptyHint, &pMemberNameHint, &memberNameHintLength, &memberNameOffset);

isComputedName = true;

}

else // not computed name

{

memberPid = this->ParseClassPropertyName(&pidHint);

if (pidHint)

{

pMemberNameHint = pidHint->Psz();

memberNameHintLength = pidHint->Cch();

}

}

if (buildAST && memberPid)

{

pnodeMemberName = CreateStrNodeWithScanner(memberPid);

}

if (!isStatic && memberPid == wellKnownPropertyPids.constructor)

{

if (hasConstructor || isAsyncMethod)

{

Error(ERRsyntax);

}

hasConstructor = true;

LPCOLESTR pConstructorName = nullptr;

ulong constructorNameLength = 0;

ulong constructorShortNameHintOffset = 0;

if (pnodeName && pnodeName->sxVar.pid)

{

pConstructorName = pnodeName->sxVar.pid->Psz();

constructorNameLength = pnodeName->sxVar.pid->Cch();

}

else

{

pConstructorName = pNameHint;

constructorNameLength = nameHintLength;

constructorShortNameHintOffset = nameHintOffset;

}

{

AutoParsingSuperRestrictionStateRestorer restorer(this);

this->m\_parsingSuperRestrictionState = hasExtends ? ParsingSuperRestrictionState\_SuperCallAndPropertyAllowed : ParsingSuperRestrictionState\_SuperPropertyAllowed;

pnodeConstructor = ParseFncDecl<buildAST>(fncDeclFlags, pConstructorName, false, /\* needsPIDOnRCurlyScan \*/ true, /\* resetParsingSuperRestrictionState = \*/false);

}

if (pnodeConstructor->sxFnc.IsGenerator())

{

Error(ERRConstructorCannotBeGenerator);

}

Assert(constructorNameLength >= constructorShortNameHintOffset);

// The constructor function will get the same name as class.

pnodeConstructor->sxFnc.hint = pConstructorName;

pnodeConstructor->sxFnc.hintLength = constructorNameLength;

pnodeConstructor->sxFnc.hintOffset = constructorShortNameHintOffset;

pnodeConstructor->sxFnc.pid = pnodeName && pnodeName->sxVar.pid ? pnodeName->sxVar.pid : wellKnownPropertyPids.constructor;

pnodeConstructor->sxFnc.SetIsClassConstructor(TRUE);

pnodeConstructor->sxFnc.SetIsBaseClassConstructor(pnodeExtends == nullptr);

}

else

{

ParseNodePtr pnodeMember = nullptr;

bool isMemberNamedGetOrSet = false;

RestorePoint beginMethodName;

m\_pscan->Capture(&beginMethodName);

if (memberPid == wellKnownPropertyPids.getter || memberPid == wellKnownPropertyPids.setter)

{

m\_pscan->ScanForcingPid();

}

if (m\_token.tk == tkLParen)

{

m\_pscan->SeekTo(beginMethodName);

isMemberNamedGetOrSet = true;

}

if ((memberPid == wellKnownPropertyPids.getter || memberPid == wellKnownPropertyPids.setter) && !isMemberNamedGetOrSet)

{

bool isGetter = (memberPid == wellKnownPropertyPids.getter);

if (m\_token.tk == tkLBrack && m\_scriptContext->GetConfig()->IsES6ObjectLiteralsEnabled())

{

// Computed get/set member name: get|set [expr] () { }

LPCOLESTR emptyHint = nullptr;

ParseComputedName<buildAST>(&pnodeMemberName, &emptyHint, &pMemberNameHint, &memberNameHintLength, &memberNameOffset);

isComputedName = true;

}

else // not computed name

{

memberPid = this->ParseClassPropertyName(&pidHint);

}

if ((isStatic ? (memberPid == wellKnownPropertyPids.prototype) : (memberPid == wellKnownPropertyPids.constructor)) || isAsyncMethod)

{

Error(ERRsyntax);

}

if (buildAST && memberPid && !isComputedName)

{

pnodeMemberName = CreateStrNodeWithScanner(memberPid);

}

ParseNodePtr pnodeFnc = nullptr;

{

AutoParsingSuperRestrictionStateRestorer restorer(this);

this->m\_parsingSuperRestrictionState = ParsingSuperRestrictionState\_SuperPropertyAllowed;

pnodeFnc = ParseFncDecl<buildAST>((isGetter ? fFncNoArg : fFncSetter) | fncDeclFlags, pidHint ? pidHint->Psz() : nullptr, false, /\* needsPIDOnRCurlyScan \*/ true, /\* resetParsingSuperRestrictionState \*/false);

}

pnodeFnc->sxFnc.SetIsStaticMember(isStatic);

if (buildAST)

{

pnodeFnc->sxFnc.SetIsAccessor();

pnodeMember = CreateBinNode(isGetter ? knopGetMember : knopSetMember, pnodeMemberName, pnodeFnc);

pMemberNameHint = ConstructFinalHintNode(pClassNamePid, pidHint, isGetter ? wellKnownPropertyPids.getter : wellKnownPropertyPids.setter, isStatic, &memberNameHintLength, &memberNameOffset, isComputedName, pMemberNameHint);

}

}

else

{

if (isStatic && (memberPid == wellKnownPropertyPids.prototype))

{

Error(ERRsyntax);

}

ParseNodePtr pnodeFnc = nullptr;

{

AutoParsingSuperRestrictionStateRestorer restorer(this);

this->m\_parsingSuperRestrictionState = ParsingSuperRestrictionState\_SuperPropertyAllowed;

if (isAsyncMethod)

{

fncDeclFlags |= fFncAsync;

}

pnodeFnc = ParseFncDecl<buildAST>(fncDeclFlags, pidHint ? pidHint->Psz() : nullptr, false, /\* needsPIDOnRCurlyScan \*/ true, /\* resetParsingSuperRestrictionState \*/false);

if (isAsyncMethod)

{

pnodeFnc->sxFnc.cbMin = iecpMin;

pnodeFnc->ichMin = ichMin;

}

}

pnodeFnc->sxFnc.SetIsStaticMember(isStatic);

if (buildAST)

{

pnodeMember = CreateBinNode(knopMember, pnodeMemberName, pnodeFnc);

pMemberNameHint = ConstructFinalHintNode(pClassNamePid, pidHint, nullptr /\*pgetset\*/, isStatic, &memberNameHintLength, &memberNameOffset, isComputedName, pMemberNameHint);

}

}

if (buildAST)

{

Assert(memberNameHintLength >= memberNameOffset);

pnodeMember->sxBin.pnode2->sxFnc.hint = pMemberNameHint; // Fully qualified name

pnodeMember->sxBin.pnode2->sxFnc.hintLength = memberNameHintLength;

pnodeMember->sxBin.pnode2->sxFnc.hintOffset = memberNameOffset;

pnodeMember->sxBin.pnode2->sxFnc.pid = memberPid; // Short name

AddToNodeList(isStatic ? &pnodeStaticMembers : &pnodeMembers, isStatic ? &lastStaticMemberNodeRef : &lastMemberNodeRef, pnodeMember);

}

}

}

if (!hasConstructor)

{

OUTPUT\_TRACE\_DEBUGONLY(Js::ES6VerboseFlag, L"Generating constructor (%s) : %s\n", GetParseType(), name ? name->Psz() : L"anonymous class");

RestorePoint endClass;

m\_pscan->Capture(&endClass);

m\_pscan->SeekTo(beginClass);

pnodeConstructor = GenerateEmptyConstructor<buildAST>(pnodeExtends != nullptr);

if (buildAST)

{

if (pClassNamePid)

{

pnodeConstructor->sxFnc.hint = pClassNamePid->Psz();

pnodeConstructor->sxFnc.hintLength = pClassNamePid->Cch();

pnodeConstructor->sxFnc.hintOffset = 0;

}

else

{

Assert(nameHintLength >= nameHintOffset);

pnodeConstructor->sxFnc.hint = pNameHint;

pnodeConstructor->sxFnc.hintLength = nameHintLength;

pnodeConstructor->sxFnc.hintOffset = nameHintOffset;

}

pnodeConstructor->sxFnc.pid = pClassNamePid;

}

m\_pscan->SeekTo(endClass);

}

if (buildAST)

{

PopFuncBlockScope(ppnodeScopeSave, ppnodeExprScopeSave);

pnodeClass->sxClass.pnodeDeclName = pnodeDeclName;

pnodeClass->sxClass.pnodeName = pnodeName;

pnodeClass->sxClass.pnodeConstructor = pnodeConstructor;

pnodeClass->sxClass.pnodeExtends = pnodeExtends;

pnodeClass->sxClass.pnodeMembers = pnodeMembers;

pnodeClass->sxClass.pnodeStaticMembers = pnodeStaticMembers;

}

FinishParseBlock(pnodeBlock);

m\_fUseStrictMode = strictSave;

m\_pscan->Scan();

return pnodeClass;

}

template<bool buildAST>

ParseNodePtr Parser::ParseStringTemplateDecl(ParseNodePtr pnodeTagFnc)

{

ParseNodePtr pnodeStringLiterals = nullptr;

ParseNodePtr\* lastStringLiteralNodeRef = nullptr;

ParseNodePtr pnodeRawStringLiterals = nullptr;

ParseNodePtr\* lastRawStringLiteralNodeRef = nullptr;

ParseNodePtr pnodeSubstitutionExpressions = nullptr;

ParseNodePtr\* lastSubstitutionExpressionNodeRef = nullptr;

ParseNodePtr pnodeTagFncArgs = nullptr;

ParseNodePtr\* lastTagFncArgNodeRef = nullptr;

ParseNodePtr stringLiteral = nullptr;

ParseNodePtr stringLiteralRaw = nullptr;

ParseNodePtr pnodeStringTemplate = nullptr;

bool templateClosed = false;

const bool isTagged = pnodeTagFnc != nullptr;

uint16 stringConstantCount = 0;

charcount\_t ichMin = 0;

Assert(m\_token.tk == tkStrTmplBasic || m\_token.tk == tkStrTmplBegin);

if (buildAST)

{

pnodeStringTemplate = CreateNode(knopStrTemplate);

pnodeStringTemplate->sxStrTemplate.countStringLiterals = 0;

pnodeStringTemplate->sxStrTemplate.isTaggedTemplate = isTagged ? TRUE : FALSE;

// If this is a tagged string template, we need to start building the arg list for the call

if (isTagged)

{

ichMin = pnodeTagFnc->ichMin;

AddToNodeListEscapedUse(&pnodeTagFncArgs, &lastTagFncArgNodeRef, pnodeStringTemplate);

}

}

CHAKRATEL\_LANGSTATS\_INC\_LANGFEATURECOUNT(StringTemplatesCount, m\_scriptContext);

OUTPUT\_TRACE\_DEBUGONLY(

Js::StringTemplateParsePhase,

L"Starting to parse a string template (%s)...\n\tis tagged = %s\n",

GetParseType(),

isTagged ? L"true" : L"false (Raw and cooked strings will not differ!)");

// String template grammar

// `...` Simple string template

// `...${ String template beginning

// }...${ String template middle

// }...` String template end

while (!templateClosed)

{

// First, extract the string constant part - we always have one

if (IsStrictMode() && m\_pscan->IsOctOrLeadingZeroOnLastTKNumber())

{

Error(ERRES5NoOctal);

}

// We are not able to pass more than a ushort worth of arguments to the tag

// so use that as a logical limit on the number of string constant pieces.

if (stringConstantCount >= USHORT\_MAX)

{

Error(ERRnoMemory);

}

// Keep track of the string literal count (must be the same for raw strings)

// We use this in code gen so we don't need to count the string literals list

stringConstantCount++;

// If we are not creating parse nodes, there is no need to create strings

if (buildAST)

{

stringLiteral = CreateStrNodeWithScanner(m\_token.GetStr());

AddToNodeList(&pnodeStringLiterals, &lastStringLiteralNodeRef, stringLiteral);

// We only need to collect a raw string when we are going to pass the string template to a tag

if (isTagged)

{

// Make the scanner create a PID for the raw string constant for the preceding scan

IdentPtr pid = m\_pscan->GetSecondaryBufferAsPid();

stringLiteralRaw = CreateStrNodeWithScanner(pid);

// Should have gotten a raw string literal above

AddToNodeList(&pnodeRawStringLiterals, &lastRawStringLiteralNodeRef, stringLiteralRaw);

}

else

{

#if DBG

// Assign the raw string for debug tracing below

stringLiteralRaw = stringLiteral;

#endif

}

OUTPUT\_TRACE\_DEBUGONLY(

Js::StringTemplateParsePhase,

L"Parsed string constant: \n\tcooked = \"%s\" \n\traw = \"%s\" \n\tdiffer = %d\n",

stringLiteral->sxPid.pid->Psz(),

stringLiteralRaw->sxPid.pid->Psz(),

stringLiteral->sxPid.pid->Psz() == stringLiteralRaw->sxPid.pid->Psz() ? 0 : 1);

}

switch (m\_token.tk)

{

case tkStrTmplEnd:

case tkStrTmplBasic:

// We do not need to parse an expression for either the end or basic string template tokens

templateClosed = true;

break;

case tkStrTmplBegin:

case tkStrTmplMid:

{

// In the middle or begin string template token case, we need to parse an expression next

m\_pscan->Scan();

// Parse the contents of the curly braces as an expression

ParseNodePtr expression = ParseExpr<buildAST>(0);

// After parsing expression, scan should leave us with an RCurly token.

// Use the NoScan version so we do not automatically perform a scan - we need to

// set the scan state before next scan but we don't want to set that state if

// the token is not as expected since we'll error in that case.

ChkCurTokNoScan(tkRCurly, ERRnoRcurly);

// Notify the scanner that it should scan for a middle or end string template token

m\_pscan->SetScanState(Scanner\_t::ScanState::ScanStateStringTemplateMiddleOrEnd);

m\_pscan->Scan();

if (buildAST)

{

// If we are going to call the tag function, add this expression into the list of args

if (isTagged)

{

AddToNodeListEscapedUse(&pnodeTagFncArgs, &lastTagFncArgNodeRef, expression);

}

else

{

// Otherwise add it to the substitution expression list

// TODO: Store the arguments and substitution expressions in a single list?

AddToNodeList(&pnodeSubstitutionExpressions, &lastSubstitutionExpressionNodeRef, expression);

}

}

if (!(m\_token.tk == tkStrTmplMid || m\_token.tk == tkStrTmplEnd))

{

// Scan with ScanState ScanStateStringTemplateMiddleOrEnd should only return

// tkStrTmpMid/End unless it is EOF or tkScanError

Assert(m\_token.tk == tkEOF || m\_token.tk == tkScanError);

Error(ERRsyntax);

}

OUTPUT\_TRACE\_DEBUGONLY(Js::StringTemplateParsePhase, L"Parsed expression\n");

}

break;

default:

Assert(false);

break;

}

}

if (buildAST)

{

pnodeStringTemplate->sxStrTemplate.pnodeStringLiterals = pnodeStringLiterals;

pnodeStringTemplate->sxStrTemplate.pnodeStringRawLiterals = pnodeRawStringLiterals;

pnodeStringTemplate->sxStrTemplate.pnodeSubstitutionExpressions = pnodeSubstitutionExpressions;

pnodeStringTemplate->sxStrTemplate.countStringLiterals = stringConstantCount;

// We should still have the last string literal.

// Use the char offset of the end of that constant as the end of the string template.

pnodeStringTemplate->ichLim = stringLiteral->ichLim;

// If this is a tagged template, we now have the argument list and can construct a call node

if (isTagged)

{

// Return the call node here and let the byte code generator Emit the string template automagically

pnodeStringTemplate = CreateCallNode(knopCall, pnodeTagFnc, pnodeTagFncArgs, ichMin, pnodeStringTemplate->ichLim);

// We need to set the arg count explicitly

pnodeStringTemplate->sxCall.argCount = stringConstantCount;

}

}

m\_pscan->Scan();

return pnodeStringTemplate;

}

void Parser::TransformAsyncFncDeclAST(ParseNodePtr \*pnodeBody, bool fLambda)

{

StmtNest \*pstmtSave;

ParseNodePtr pnodeReturn;

ParseNodePtr pnodeAsyncSpawn;

ParseNodePtr pnodeFncGenerator = nullptr;

ParseNodePtr pnodeFncSave = nullptr;

ParseNodePtr pnodeDeferredFncSave = nullptr;

ParseNodePtr pnodeInnerBlock = nullptr;

ParseNodePtr pnodeBlock = nullptr;

ParseNodePtr \*lastNodeRef = nullptr;

ParseNodePtr \*ppnodeScopeSave = nullptr;

ParseNodePtr \*ppnodeExprScopeSave = nullptr;

AutoParsingSuperRestrictionStateRestorer restorer(this);

// Create the generator : function\*() {}

uint tryCatchOrFinallyDepthSave = this->m\_tryCatchOrFinallyDepth;

this->m\_tryCatchOrFinallyDepth = 0;

uint scopeCountNoAstSave = m\_scopeCountNoAst;

m\_scopeCountNoAst = 0;

long\* pAstSizeSave = m\_pCurrentAstSize;

pnodeFncSave = m\_currentNodeFunc;

pnodeDeferredFncSave = m\_currentNodeDeferredFunc;

pnodeFncGenerator = CreateAsyncSpawnGenerator();

m\_currentNodeDeferredFunc = pnodeFncGenerator;

m\_inDeferredNestedFunc = true;

pstmtSave = m\_pstmtCur;

SetCurrentStatement(nullptr);

bool fPreviousYieldIsKeyword = m\_pscan->SetYieldIsKeyword(FALSE);

BOOL oldStrictMode = this->m\_fUseStrictMode;

uint uDeferSave = m\_grfscr & fscrDeferFncParse;

pnodeBlock = StartParseBlock<true>(PnodeBlockType::Parameter, ScopeType\_Parameter);

pnodeFncGenerator->sxFnc.pnodeScopes = pnodeBlock;

m\_ppnodeVar = &pnodeFncGenerator->sxFnc.pnodeArgs;

ppnodeScopeSave = m\_ppnodeScope;

m\_ppnodeScope = &pnodeBlock->sxBlock.pnodeScopes;

pnodeBlock->sxBlock.pnodeStmt = pnodeFncGenerator;

ppnodeExprScopeSave = m\_ppnodeExprScope;

m\_ppnodeExprScope = nullptr;

m\_fUseStrictMode = oldStrictMode;

pnodeInnerBlock = StartParseBlock<true>(PnodeBlockType::Function, ScopeType\_FunctionBody);

\*m\_ppnodeScope = pnodeInnerBlock;

pnodeFncGenerator->sxFnc.pnodeBodyScope = pnodeInnerBlock;

m\_ppnodeScope = &pnodeInnerBlock->sxBlock.pnodeScopes;

pnodeInnerBlock->sxBlock.pnodeStmt = pnodeFncGenerator;

Assert(\*m\_ppnodeVar == nullptr);

pnodeFncGenerator->sxFnc.pnodeVars = nullptr;

m\_ppnodeVar = &pnodeFncGenerator->sxFnc.pnodeVars;

DeferredFunctionStub \*saveCurrentStub = m\_currDeferredStub;

if (pnodeFncSave && m\_currDeferredStub)

{

m\_currDeferredStub = (m\_currDeferredStub + (pnodeFncSave->sxFnc.nestedCount - 1))->deferredStubs;

}

pnodeFncGenerator->sxFnc.pnodeBody = nullptr;

if (fLambda)

{

// Parse and set the function body

ParseExpressionLambdaBody<true>(\*pnodeBody);

AddToNodeList(&pnodeFncGenerator->sxFnc.pnodeBody, &lastNodeRef, (\*pnodeBody)->sxFnc.pnodeScopes->sxBlock.pnodeStmt);

}

else

{

// Parse the function body

ParseStmtList<true>(&pnodeFncGenerator->sxFnc.pnodeBody, &lastNodeRef, SM\_OnFunctionCode, true);

ChkCurTokNoScan(tkRCurly, ERRnoRcurly);

}

AddToNodeList(&pnodeFncGenerator->sxFnc.pnodeBody, &lastNodeRef, CreateNodeWithScanner<knopEndCode>());

lastNodeRef = NULL;

pnodeFncGenerator->ichLim = m\_pscan->IchLimTok();

pnodeFncGenerator->sxFnc.cbLim = m\_pscan->IecpLimTok();

m\_currDeferredStub = saveCurrentStub;

FinishParseBlock(pnodeInnerBlock, true);

this->AddArgumentsNodeToVars(pnodeFncGenerator);

Assert(m\_ppnodeExprScope == nullptr || \*m\_ppnodeExprScope == nullptr);

m\_ppnodeExprScope = ppnodeExprScopeSave;

AssertMem(m\_ppnodeScope);

Assert(nullptr == \*m\_ppnodeScope);

m\_ppnodeScope = ppnodeScopeSave;

FinishParseBlock(pnodeBlock, true);

Assert(nullptr == m\_pstmtCur);

SetCurrentStatement(pstmtSave);

if (!m\_stoppedDeferredParse)

{

m\_grfscr |= uDeferSave;

}

m\_pscan->SetYieldIsKeyword(fPreviousYieldIsKeyword);

Assert(pnodeFncGenerator == m\_currentNodeFunc);

m\_currentNodeFunc = pnodeFncSave;

m\_currentNodeDeferredFunc = pnodeDeferredFncSave;

m\_pCurrentAstSize = pAstSizeSave;

m\_inDeferredNestedFunc = false;

m\_scopeCountNoAst = scopeCountNoAstSave;

this->m\_tryCatchOrFinallyDepth = tryCatchOrFinallyDepthSave;

// Create the call : spawn(function\*() {}, this)

pnodeAsyncSpawn = CreateBinNode(knopAsyncSpawn, pnodeFncGenerator, CreateNodeWithScanner<knopThis>());

// Create the return : return spawn(function\*() {}, this)

pnodeReturn = CreateNodeWithScanner<knopReturn>();

pnodeReturn->sxStmt.grfnop = 0;

pnodeReturn->sxStmt.pnodeOuter = nullptr;

pnodeReturn->sxReturn.pnodeExpr = pnodeAsyncSpawn;

if (fLambda)

{

(\*pnodeBody)->sxFnc.pnodeScopes->sxBlock.pnodeStmt = nullptr;

AddToNodeList(&(\*pnodeBody)->sxFnc.pnodeScopes->sxBlock.pnodeStmt, &lastNodeRef, pnodeReturn);

}

else

{

\*pnodeBody = nullptr;

AddToNodeList(pnodeBody, &lastNodeRef, pnodeReturn);

AddToNodeList(pnodeBody, &lastNodeRef, CreateNodeWithScanner<knopEndCode>());

}

lastNodeRef = NULL;

}

ParseNodePtr Parser::CreateAsyncSpawnGenerator()

{

ParseNodePtr pnodeFncGenerator = nullptr;

pnodeFncGenerator = CreateDummyFuncNode(false);

pnodeFncGenerator->sxFnc.functionId = (\*m\_nextFunctionId)++;

pnodeFncGenerator->sxFnc.cbMin = m\_pscan->IecpMinTok();

pnodeFncGenerator->sxFnc.cbLim = m\_pscan->IecpLimTok();

pnodeFncGenerator->sxFnc.lineNumber = m\_pscan->LineCur();

pnodeFncGenerator->sxFnc.columnNumber = CalculateFunctionColumnNumber();

pnodeFncGenerator->sxFnc.SetNested(m\_currentNodeFunc != nullptr);

pnodeFncGenerator->sxFnc.SetStrictMode(IsStrictMode());

pnodeFncGenerator->sxFnc.SetIsGenerator();

pnodeFncGenerator->sxFnc.scope = nullptr;

AppendFunctionToScopeList(false, pnodeFncGenerator);

return pnodeFncGenerator;

}

LPCOLESTR Parser::FormatPropertyString(LPCOLESTR propertyString, ParseNodePtr pNode, ulong \*fullNameHintLength, ulong \*pShortNameOffset)

{

// propertyString could be null, such as 'this.foo' =

// propertyString could be empty, found in pattern as in (-1)[""][(x = z)]

OpCode op = pNode->nop;

LPCOLESTR rightNode = nullptr;

if (propertyString == nullptr)

{

propertyString = L"";

}

if (op != knopInt && op != knopFlt && op != knopName && op != knopStr)

{

rightNode = L"";

}

else if (op == knopStr)

{

return AppendNameHints(propertyString, pNode->sxPid.pid, fullNameHintLength, pShortNameOffset, false, true/\*add brackets\*/);

}

else if(op == knopFlt)

{

rightNode = m\_pscan->StringFromDbl(pNode->sxFlt.dbl);

}

else

{

rightNode = op == knopInt ? m\_pscan->StringFromLong(pNode->sxInt.lw)

: pNode->sxPid.pid->Psz();

}

return AppendNameHints(propertyString, rightNode, fullNameHintLength, pShortNameOffset, false, true/\*add brackets\*/);

}

LPCOLESTR Parser::ConstructNameHint(ParseNodePtr pNode, ulong\* fullNameHintLength, ulong \*pShortNameOffset)

{

Assert(pNode != nullptr);

Assert(pNode->nop == knopDot || pNode->nop == knopIndex);

LPCOLESTR leftNode = nullptr;

if (pNode->sxBin.pnode1->nop == knopDot || pNode->sxBin.pnode1->nop == knopIndex)

{

leftNode = ConstructNameHint(pNode->sxBin.pnode1, fullNameHintLength, pShortNameOffset);

}

else if (pNode->sxBin.pnode1->nop == knopName)

{

leftNode = pNode->sxBin.pnode1->sxPid.pid->Psz();

\*fullNameHintLength = pNode->sxBin.pnode1->sxPid.pid->Cch();

\*pShortNameOffset = 0;

}

if (pNode->nop == knopIndex)

{

return FormatPropertyString(

leftNode ? leftNode : Js::Constants::AnonymousFunction, // e.g. f()[0] = function () {}

pNode->sxBin.pnode2, fullNameHintLength, pShortNameOffset);

}

Assert(pNode->sxBin.pnode2->nop == knopDot || pNode->sxBin.pnode2->nop == knopName);

LPCOLESTR rightNode = nullptr;

bool wrapWithBrackets = false;

if (pNode->sxBin.pnode2->nop == knopDot)

{

rightNode = ConstructNameHint(pNode->sxBin.pnode2, fullNameHintLength, pShortNameOffset);

}

else

{

rightNode = pNode->sxBin.pnode2->sxPid.pid->Psz();

wrapWithBrackets = PNodeFlags::fpnIndexOperator == (pNode->grfpn & PNodeFlags::fpnIndexOperator);

}

Assert(rightNode != nullptr);

return AppendNameHints(leftNode, rightNode, fullNameHintLength, pShortNameOffset, false, wrapWithBrackets);

}

LPCOLESTR Parser::AppendNameHints(LPCOLESTR leftStr, ulong leftLen, LPCOLESTR rightStr, ulong rightLen, ulong \*pNameLength, ulong \*pShortNameOffset, bool ignoreAddDotWithSpace, bool wrapInBrackets)

{

Assert(rightStr != nullptr);

Assert(leftLen != 0 || wrapInBrackets);

Assert(rightLen != 0 || wrapInBrackets);

bool ignoreDot = rightStr[0] == L'[' && !wrapInBrackets;//if we wrap in brackets it can be a string literal which can have brackets at the first char

ulong totalLength = leftLen + rightLen + ((ignoreDot) ? 1 : 2); // 1 (for dot or [) + 1 (for null termination)

if (wrapInBrackets)

{

totalLength++; //1 for ']';

}

WCHAR \* finalName = AllocateStringOfLength(totalLength);

if (leftStr != nullptr && leftLen != 0)

{

wcscpy\_s(finalName, leftLen + 1, leftStr);

}

if (ignoreAddDotWithSpace)

{

finalName[leftLen++] = (OLECHAR)L' ';

}

// mutually exclusive from ignoreAddDotWithSpace which is used for getters/setters

else if (wrapInBrackets)

{

finalName[leftLen++] = (OLECHAR)L'[';

finalName[totalLength-2] = (OLECHAR)L']';

}

else if (!ignoreDot)

{

finalName[leftLen++] = (OLECHAR)L'.';

}

//ignore case falls through

js\_wmemcpy\_s(finalName + leftLen, rightLen, rightStr, rightLen);

finalName[totalLength-1] = (OLECHAR)L'\0';

if (pNameLength != nullptr)

{

\*pNameLength = totalLength - 1;

}

if (pShortNameOffset != nullptr)

{

\*pShortNameOffset = leftLen;

}

return finalName;

}

WCHAR \* Parser::AllocateStringOfLength(ulong length)

{

Assert(length > 0);

ULONG totalBytes;

if (ULongMult(length, sizeof(OLECHAR), &totalBytes) != S\_OK)

{

Error(ERRnoMemory);

}

WCHAR\* finalName = (WCHAR\*)m\_phtbl->GetAllocator()->Alloc(totalBytes);

if (finalName == nullptr)

{

Error(ERRnoMemory);

}

return finalName;

}

LPCOLESTR Parser::AppendNameHints(IdentPtr left, IdentPtr right, ulong \*pNameLength, ulong \*pShortNameOffset, bool ignoreAddDotWithSpace, bool wrapInBrackets)

{

if (pShortNameOffset != nullptr)

{

\*pShortNameOffset = 0;

}

if (left == nullptr && !wrapInBrackets)

{

if (right)

{

\*pNameLength = right->Cch();

return right->Psz();

}

return nullptr;

}

ulong leftLen = 0;

LPCOLESTR leftStr = L"";

if (left != nullptr) // if wrapInBrackets is true

{

leftStr = left->Psz();

leftLen = left->Cch();

}

if (right == nullptr)

{

\*pNameLength = leftLen;

return left->Psz();

}

ulong rightLen = right->Cch();

return AppendNameHints(leftStr, leftLen, right->Psz(), rightLen, pNameLength, pShortNameOffset, ignoreAddDotWithSpace, wrapInBrackets);

}

LPCOLESTR Parser::AppendNameHints(IdentPtr left, LPCOLESTR right, ulong \*pNameLength, ulong \*pShortNameOffset, bool ignoreAddDotWithSpace, bool wrapInBrackets)

{

ulong rightLen = (right == nullptr) ? 0 : (ulong) wcslen(right);

if (pShortNameOffset != nullptr)

{

\*pShortNameOffset = 0;

}

Assert(rightLen <= ULONG\_MAX); // name hints should not exceed ULONG\_MAX characters

if (left == nullptr && !wrapInBrackets)

{

\*pNameLength = rightLen;

return right;

}

LPCOLESTR leftStr = L"";

ulong leftLen = 0;

if (left != nullptr) // if wrapInBrackets is true

{

leftStr = left->Psz();

leftLen = left->Cch();

}

if (rightLen == 0 && !wrapInBrackets)

{

\*pNameLength = leftLen;

return left->Psz();

}

return AppendNameHints(leftStr, leftLen, right, rightLen, pNameLength, pShortNameOffset, ignoreAddDotWithSpace, wrapInBrackets);

}

LPCOLESTR Parser::AppendNameHints(LPCOLESTR left, IdentPtr right, ulong \*pNameLength, ulong \*pShortNameOffset, bool ignoreAddDotWithSpace, bool wrapInBrackets)

{

ulong leftLen = (left == nullptr) ? 0 : (ulong) wcslen(left);

if (pShortNameOffset != nullptr)

{

\*pShortNameOffset = 0;

}

Assert(leftLen <= ULONG\_MAX); // name hints should not exceed ULONG\_MAX characters

if (left == nullptr || leftLen == 0 && !wrapInBrackets)

{

if (right != nullptr)

{

\*pNameLength = right->Cch();

return right->Psz();

}

return nullptr;

}

if (right == nullptr)

{

\*pNameLength = leftLen;

return left;

}

ulong rightLen = right->Cch();

return AppendNameHints(left, leftLen, right->Psz(), rightLen, pNameLength, pShortNameOffset, ignoreAddDotWithSpace, wrapInBrackets);

}

LPCOLESTR Parser::AppendNameHints(LPCOLESTR left, LPCOLESTR right, ulong \*pNameLength, ulong \*pShortNameOffset, bool ignoreAddDotWithSpace, bool wrapInBrackets)

{

ulong leftLen = (left == nullptr) ? 0 : (ulong) wcslen(left);

ulong rightLen = (right == nullptr) ? 0 : (ulong) wcslen(right);

if (pShortNameOffset != nullptr)

{

\*pShortNameOffset = 0;

}

Assert(rightLen <= ULONG\_MAX && leftLen <= ULONG\_MAX); // name hints should not exceed ULONG\_MAX characters

if (leftLen == 0 && !wrapInBrackets)

{

\*pNameLength = right ? rightLen : 0;

return right;

}

if (rightLen == 0 && !wrapInBrackets)

{

\*pNameLength = leftLen;

return left;

}

return AppendNameHints(left, leftLen, right, rightLen, pNameLength, pShortNameOffset, ignoreAddDotWithSpace, wrapInBrackets);

}

/\*\*

\* Emits a spread error if there is no ambiguity, or marks defers the error for

\* when we can determine if it is a rest error or a spread error.

\*

\* The ambiguity arises when we are parsing a lambda parameter list but we have

\* not seen the => token. At this point, we are either in a parenthesized

\* expression or a parameter list, and cannot issue an error until the matching

\* RParen has been scanned.

\*

\* The actual emission of the error happens in ParseExpr, when we first know if

\* the expression is a lambda parameter list or not.

\*

\*/

void Parser::DeferOrEmitPotentialSpreadError(ParseNodePtr pnodeT)

{

if (m\_parenDepth > 0)

{

if (m\_token.tk == tkRParen)

{

if (!m\_deferEllipsisError)

{

// Capture only the first error instance.

m\_pscan->Capture(&m\_EllipsisErrLoc);

m\_deferEllipsisError = true;

}

}

else

{

Error(ERRUnexpectedEllipsis);

}

}

else

{

Error(ERRInvalidSpreadUse);

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Parse an optional sub expression returning null if there was no expression.

Checks for no expression by looking for a token that can follow an

Expression grammar production.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

template<bool buildAST>

bool Parser::ParseOptionalExpr(ParseNodePtr\* pnode, bool fUnaryOrParen, int oplMin, BOOL \*pfCanAssign, BOOL fAllowIn, BOOL fAllowEllipsis, \_Inout\_opt\_ IdentToken\* pToken)

{

\*pnode = nullptr;

if (m\_token.tk == tkRCurly ||

m\_token.tk == tkRBrack ||

m\_token.tk == tkRParen ||

m\_token.tk == tkSColon ||

m\_token.tk == tkColon ||

m\_token.tk == tkComma ||

m\_token.tk == tkLimKwd ||

m\_pscan->FHadNewLine())

{

return false;

}

\*pnode = ParseExpr<buildAST>(oplMin, pfCanAssign, fAllowIn, fAllowEllipsis, nullptr /\*pNameHint\*/, nullptr /\*pHintLength\*/, nullptr /\*pShortNameOffset\*/, pToken, fUnaryOrParen);

return true;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Parse a sub expression.

'fAllowIn' indicates if the 'in' operator should be allowed in the initializing

expression ( it is not allowed in the context of the first expression in a 'for' loop).

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

template<bool buildAST>

ParseNodePtr Parser::ParseExpr(int oplMin,

BOOL \*pfCanAssign,

BOOL fAllowIn,

BOOL fAllowEllipsis,

LPCOLESTR pNameHint,

ulong \*pHintLength,

ulong \*pShortNameOffset,

\_Inout\_opt\_ IdentToken\* pToken,

bool fUnaryOrParen,

\_Inout\_opt\_ bool\* pfLikelyPattern)

{

Assert(pToken == nullptr || pToken->tk == tkNone); // Must be empty initially

int opl;

OpCode nop;

charcount\_t ichMin;

ParseNodePtr pnode = nullptr;

ParseNodePtr pnodeT = nullptr;

BOOL fCanAssign = TRUE;

bool assignmentStmt = false;

IdentToken term;

RestorePoint termStart;

ulong hintLength = 0;

ulong hintOffset = 0;

if (pHintLength != nullptr)

{

hintLength = \*pHintLength;

}

if (pShortNameOffset != nullptr)

{

hintOffset = \*pShortNameOffset;

}

EnsureStackAvailable();

m\_pscan->Capture(&termStart);

// Is the current token a unary operator?

if (m\_phtbl->TokIsUnop(m\_token.tk, &opl, &nop) && nop != knopNone)

{

IdentToken operandToken;

ichMin = m\_pscan->IchMinTok();

if (nop == knopYield)

{

if (!m\_pscan->YieldIsKeyword() || oplMin > opl)

{

// The case where 'yield' is scanned as a keyword (tkYIELD) but the scanner

// is not treating yield as a keyword (!m\_pscan->YieldIsKeyword()) happens

// in strict mode non-generator function contexts.

//

// That is, 'yield' is a keyword because of strict mode, but YieldExpression

// is not a grammar production outside of generator functions.

//

// Otherwise it is an error for a yield to appear in the context of a higher level

// binding operator, be it unary or binary.

Error(ERRsyntax);

}

if (m\_currentNodeFunc->sxFnc.IsGenerator()

&& m\_currentBlockInfo->pnodeBlock->sxBlock.blockType == PnodeBlockType::Parameter)

{

Error(ERRsyntax);

}

}

else if (nop == knopAwait)

{

if (!m\_pscan->AwaitIsKeyword() || oplMin > opl)

{

// As 'yield' keyword, the case where 'await' is scanned as a keyword (tkAWAIT) but the scanner

// is not treating await as a keyword (!m\_pscan->AwaitIsKeyword()) happens

// in strict mode non-generator function contexts.

//

// That is, 'await' is a keyword because of strict mode, but AwaitExpression

// is not a grammar production outside of generator functions.

//

// Otherwise it is an error for a yield to appear in the context of a higher level

// binding operator, be it unary or binary.

Error(ERRsyntax);

}

}

m\_pscan->Scan();

if (nop == knopYield && !m\_pscan->FHadNewLine() && m\_token.tk == tkStar)

{

m\_pscan->Scan();

nop = knopYieldStar;

}

if (nop == knopYield)

{

if (!ParseOptionalExpr<buildAST>(&pnodeT, false, opl, NULL, TRUE, fAllowEllipsis))

{

nop = knopYieldLeaf;

if (buildAST)

{

pnode = CreateNodeT<knopYieldLeaf>(ichMin, m\_pscan->IchLimTok());

}

}

}

else

{

// Disallow spread after a Ellipsis token. This prevents chaining, and ensures spread is the top level expression.

pnodeT = ParseExpr<buildAST>(opl, &fCanAssign, TRUE, nop != knopEllipsis && fAllowEllipsis, nullptr /\*hint\*/, nullptr /\*hintLength\*/, nullptr /\*hintOffset\*/, &operandToken, true);

}

if (nop != knopYieldLeaf)

{

if (nop == knopIncPre || nop == knopDecPre)

{

if (!fCanAssign && PHASE\_ON1(Js::EarlyReferenceErrorsPhase))

{

Error(JSERR\_CantAssignTo);

}

TrackAssignment<buildAST>(pnodeT, &operandToken, ichMin, m\_pscan->IchLimTok());

if (buildAST)

{

if (IsStrictMode() && pnodeT->nop == knopName)

{

CheckStrictModeEvalArgumentsUsage(pnodeT->sxPid.pid);

}

}

else

{

if (IsStrictMode() && operandToken.tk == tkID)

{

CheckStrictModeEvalArgumentsUsage(operandToken.pid);

}

}

}

else if (nop == knopEllipsis && !fAllowEllipsis)

{

DeferOrEmitPotentialSpreadError(pnodeT);

}

if (buildAST)

{

//Do not do the folding for Asm in case of KnopPos as we need this to determine the type

if (nop == knopPos && (pnodeT->nop == knopInt || pnodeT->nop == knopFlt) && !this->m\_InAsmMode)

{

// Fold away a unary '+' on a number.

pnode = pnodeT;

}

else if (nop == knopNeg &&

((pnodeT->nop == knopInt && pnodeT->sxInt.lw != 0) ||

(pnodeT->nop == knopFlt && (pnodeT->sxFlt.dbl != 0 || this->m\_InAsmMode))))

{

// Fold a unary '-' on a number into the value of the number itself.

pnode = pnodeT;

if (pnode->nop == knopInt)

{

pnode->sxInt.lw = -pnode->sxInt.lw;

}

else

{

pnode->sxFlt.dbl = -pnode->sxFlt.dbl;

}

}

else

{

pnode = CreateUniNode(nop, pnodeT);

this->CheckArguments(pnode->sxUni.pnode1);

}

pnode->ichMin = ichMin;

}

if (nop == knopDelete)

{

if (IsStrictMode())

{

if ((buildAST && pnode->sxUni.pnode1->nop == knopName) ||

(!buildAST && operandToken.tk == tkID))

{

Error(ERRInvalidDelete);

}

}

if (buildAST)

{

ParseNodePtr pnode1 = pnode->sxUni.pnode1;

if (m\_currentNodeFunc)

{

if (pnode1->nop == knopDot || pnode1->nop == knopIndex)

{

// If we delete an arguments property, use the conservative,

// heap-allocated arguments object.

this->CheckArguments(pnode1->sxBin.pnode1);

}

}

}

}

}

fCanAssign = FALSE;

}

else

{

ichMin = m\_pscan->IchMinTok();

BOOL fLikelyPattern = FALSE;

pnode = ParseTerm<buildAST>(TRUE, pNameHint, &hintLength, &hintOffset, &term, fUnaryOrParen, &fCanAssign, IsES6DestructuringEnabled() ? &fLikelyPattern : nullptr);

if (pfLikelyPattern != nullptr)

{

\*pfLikelyPattern = !!fLikelyPattern;

}

if (m\_token.tk == tkAsg && oplMin <= koplAsg && fLikelyPattern)

{

m\_pscan->SeekTo(termStart);

ParseDestructuredLiteralWithScopeSave(tkLCurly, false/\*isDecl\*/, false /\*topLevel\*/, DIC\_ShouldNotParseInitializer);

if (buildAST)

{

pnode = ConvertToPattern(pnode);

}

}

if (buildAST)

{

pNameHint = NULL;

if (pnode->nop == knopName)

{

pNameHint = pnode->sxPid.pid->Psz();

hintLength = pnode->sxPid.pid->Cch();

hintOffset = 0;

}

else if (pnode->nop == knopDot || pnode->nop == knopIndex)

{

if (CONFIG\_FLAG(UseFullName))

{

pNameHint = ConstructNameHint(pnode, &hintLength, &hintOffset);

}

else

{

ParseNodePtr pnodeName = pnode;

while (pnodeName->nop == knopDot)

{

pnodeName = pnodeName->sxBin.pnode2;

}

if (pnodeName->nop == knopName)

{

pNameHint = pnodeName->sxPid.pid->Psz();

hintLength = pnodeName->sxPid.pid->Cch();

hintOffset = 0;

}

}

}

}

// Check for postfix unary operators.

if (!m\_pscan->FHadNewLine() &&

(tkInc == m\_token.tk || tkDec == m\_token.tk))

{

if (!fCanAssign && PHASE\_ON1(Js::EarlyReferenceErrorsPhase))

{

Error(JSERR\_CantAssignTo);

}

TrackAssignment<buildAST>(pnode, &term, ichMin, m\_pscan->IchLimTok());

fCanAssign = FALSE;

if (buildAST)

{

if (IsStrictMode() && pnode->nop == knopName)

{

CheckStrictModeEvalArgumentsUsage(pnode->sxPid.pid);

}

this->CheckArguments(pnode);

pnode = CreateUniNode(tkInc == m\_token.tk ? knopIncPost : knopDecPost, pnode);

pnode->ichLim = m\_pscan->IchLimTok();

}

else

{

if (IsStrictMode() && term.tk == tkID)

{

CheckStrictModeEvalArgumentsUsage(term.pid);

}

// This expression is not an identifier

term.tk = tkNone;

}

m\_pscan->Scan();

}

}

// Process a sequence of operators and operands.

for (;;)

{

if (!m\_phtbl->TokIsBinop(m\_token.tk, &opl, &nop) || nop == knopNone)

{

break;

}

if ( ! fAllowIn && nop == knopIn )

{

break;

}

Assert(opl != koplNo);

if (opl == koplAsg)

{

if (m\_token.tk != tkDArrow)

{

// Assignment operator. These are the only right associative

// binary operators. We also need to special case the left

// operand - it should only be a LeftHandSideExpression.

Assert(ParseNode::Grfnop(nop) & fnopAsg || nop == knopFncDecl);

TrackAssignment<buildAST>(pnode, &term, ichMin, m\_pscan->IchLimTok());

if (buildAST)

{

if (IsStrictMode() && pnode->nop == knopName)

{

CheckStrictModeEvalArgumentsUsage(pnode->sxPid.pid);

}

// Assignment stmt of the form "this.<id> = <expr>"

if (nop == knopAsg && pnode->nop == knopDot && pnode->sxBin.pnode1->nop == knopThis && pnode->sxBin.pnode2->nop == knopName)

{

if (pnode->sxBin.pnode2->sxPid.pid != wellKnownPropertyPids.\_\_proto\_\_)

{

assignmentStmt = true;

}

}

}

else

{

if (IsStrictMode() && term.tk == tkID)

{

CheckStrictModeEvalArgumentsUsage(term.pid);

}

}

}

if (opl < oplMin)

{

break;

}

if (m\_token.tk != tkDArrow && !fCanAssign && PHASE\_ON1(Js::EarlyReferenceErrorsPhase))

{

Error(JSERR\_CantAssignTo);

// No recovery necessary since this is a semantic, not structural, error.

}

}

else if (opl == koplExpo)

{

// \*\* operator is right associative

if (opl < oplMin)

{

break;

}

}

else if (opl <= oplMin)

{

break;

}

// This expression is not an identifier

term.tk = tkNone;

// Precedence is high enough. Consume the operator token.

m\_pscan->Scan();

fCanAssign = FALSE;

// Special case the "?:" operator

if (nop == knopQmark)

{

pnodeT = ParseExpr<buildAST>(koplAsg, NULL, fAllowIn);

ChkCurTok(tkColon, ERRnoColon);

ParseNodePtr pnodeT2 = ParseExpr<buildAST>(koplAsg, NULL, fAllowIn);

if (buildAST)

{

pnode = CreateTriNode(nop, pnode, pnodeT, pnodeT2);

this->CheckArguments(pnode->sxTri.pnode2);

this->CheckArguments(pnode->sxTri.pnode3);

}

}

else if (nop == knopFncDecl)

{

ushort flags = fFncLambda;

size\_t iecpMin = 0;

bool isAsyncMethod = false;

m\_pscan->SeekTo(termStart);

if (m\_token.tk == tkID && m\_token.GetIdentifier(m\_phtbl) == wellKnownPropertyPids.async && m\_scriptContext->GetConfig()->IsES7AsyncAndAwaitEnabled())

{

ichMin = m\_pscan->IchMinTok();

iecpMin = m\_pscan->IecpMinTok();

m\_pscan->Scan();

if (m\_token.tk == tkID || m\_token.tk == tkLParen)

{

flags |= fFncAsync;

isAsyncMethod = true;

}

else

{

m\_pscan->SeekTo(termStart);

}

}

pnode = ParseFncDecl<buildAST>(flags, nullptr, /\* isSourceElement = \*/ false, /\* needsPIDOnRCurlyScan = \*/false, /\* resetParsingSuperRestrictionState = \*/false);

if (isAsyncMethod)

{

pnode->sxFnc.cbMin = iecpMin;

pnode->ichMin = ichMin;

}

}

else

{

// Parse the operand, make a new node, and look for more

pnodeT = ParseExpr<buildAST>(opl, NULL, fAllowIn, FALSE, pNameHint, &hintLength, &hintOffset, nullptr);

if (buildAST)

{

pnode = CreateBinNode(nop, pnode, pnodeT);

Assert(pnode->sxBin.pnode2 != NULL);

if (pnode->sxBin.pnode2->nop == knopFncDecl)

{

Assert(hintLength >= hintOffset);

pnode->sxBin.pnode2->sxFnc.hint = pNameHint;

pnode->sxBin.pnode2->sxFnc.hintLength = hintLength;

pnode->sxBin.pnode2->sxFnc.hintOffset = hintOffset;

if (pnode->sxBin.pnode1->nop == knopDot)

{

pnode->sxBin.pnode2->sxFnc.isNameIdentifierRef = false;

}

}

if (pnode->sxBin.pnode2->nop == knopClassDecl && pnode->sxBin.pnode1->nop == knopDot)

{

Assert(pnode->sxBin.pnode2->sxClass.pnodeConstructor);

pnode->sxBin.pnode2->sxClass.pnodeConstructor->sxFnc.isNameIdentifierRef = false;

}

}

pNameHint = NULL;

}

}

if (buildAST)

{

if (!assignmentStmt)

{

// Don't set the flag for following nodes

switch (pnode->nop)

{

case knopName:

case knopInt:

case knopFlt:

case knopStr:

case knopRegExp:

case knopNull:

case knopFalse:

case knopTrue:

break;

default:

if (m\_currentNodeFunc)

{

m\_currentNodeFunc->sxFnc.SetHasNonThisStmt();

}

else if (m\_currentNodeProg)

{

m\_currentNodeProg->sxFnc.SetHasNonThisStmt();

}

}

}

}

if (NULL != pfCanAssign)

{

\*pfCanAssign = fCanAssign;

}

// Pass back identifier if requested

if (pToken && term.tk == tkID)

{

\*pToken = term;

}

//Track "obj.a" assignment patterns here - Promote the Assignment state for the property's PID.

// This includes =, += etc.

if (pnode != NULL)

{

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

if (nodeType & fnopAsg)

{

if (nodeType & fnopBin)

{

ParseNodePtr lhs = pnode->sxBin.pnode1;

Assert(lhs);

if (lhs->nop == knopDot)

{

ParseNodePtr propertyNode = lhs->sxBin.pnode2;

if (propertyNode->nop == knopName)

{

propertyNode->sxPid.pid->PromoteAssignmentState();

}

}

}

else if (nodeType & fnopUni)

{

// cases like obj.a++, ++obj.a

ParseNodePtr lhs = pnode->sxUni.pnode1;

if (lhs->nop == knopDot)

{

ParseNodePtr propertyNode = lhs->sxBin.pnode2;

if (propertyNode->nop == knopName)

{

propertyNode->sxPid.pid->PromoteAssignmentState();

}

}

}

}

}

return pnode;

}

template<bool buildAST>

void Parser::TrackAssignment(ParseNodePtr pnodeT, IdentToken\* pToken, charcount\_t ichMin, charcount\_t ichLim)

{

if (buildAST)

{

Assert(pnodeT != NULL);

if (pnodeT->nop == knopName)

{

PidRefStack \*ref = pnodeT->sxPid.pid->GetTopRef();

Assert(ref);

ref->TrackAssignment(pnodeT->ichMin, pnodeT->ichLim);

}

}

else

{

Assert(pToken != NULL);

if (BindDeferredPidRefs() && pToken->tk == tkID)

{

PidRefStack \*ref = pToken->pid->GetTopRef();

Assert(ref);

ref->TrackAssignment(ichMin, ichLim);

}

}

}

void PidRefStack::TrackAssignment(charcount\_t ichMin, charcount\_t ichLim)

{

if (this->isAsg)

{

if (this->GetIchMin() <= ichMin)

{

return;

}

Assert(this->GetIchLim() >= ichLim);

}

this->isAsg = true;

this->span.Set(ichMin, ichLim);

}

void PnPid::SetSymRef(PidRefStack \*ref)

{

Assert(symRef == nullptr);

this->symRef = ref->GetSymRef();

}

Js::PropertyId PnPid::PropertyIdFromNameNode() const

{

Js::PropertyId propertyId;

Symbol \*sym = this->sym;

if (sym)

{

propertyId = sym->GetPosition();

}

else

{

propertyId = this->pid->GetPropertyId();

}

return propertyId;

}

PidRefStack\* Parser::PushPidRef(IdentPtr pid)

{

if (PHASE\_ON1(Js::ParallelParsePhase))

{

// NOTE: the phase check is here to protect perf. See OSG 1020424.

// In some LS AST-rewrite cases we lose a lot of perf searching the PID ref stack rather

// than just pushing on the top. This hasn't shown up as a perf issue in non-LS benchmarks.

return pid->FindOrAddPidRef(&m\_nodeAllocator, GetCurrentBlock()->sxBlock.blockId);

}

Assert(GetCurrentBlock() != nullptr);

AssertMsg(pid != nullptr, "PID should be created");

PidRefStack \*ref = pid->GetTopRef();

if (!ref || (ref->GetScopeId() < GetCurrentBlock()->sxBlock.blockId

// We could have the ref from the parameter scope. In that case we can skip creating a new one.

&& !(m\_currentBlockInfo->pBlockInfoOuter->pnodeBlock->sxBlock.blockType == PnodeBlockType::Parameter

&& m\_currentBlockInfo->pBlockInfoOuter->pnodeBlock->sxBlock.blockId == ref->GetScopeId())))

{

ref = Anew(&m\_nodeAllocator, PidRefStack);

if (ref == nullptr)

{

Error(ERRnoMemory);

}

pid->PushPidRef(GetCurrentBlock()->sxBlock.blockId, ref);

}

return ref;

}

PidRefStack\* Parser::FindOrAddPidRef(IdentPtr pid, int scopeId, int maxScopeId)

{

PidRefStack \*ref = pid->FindOrAddPidRef(&m\_nodeAllocator, scopeId, maxScopeId);

if (ref == NULL)

{

Error(ERRnoMemory);

}

return ref;

}

void Parser::RemovePrevPidRef(IdentPtr pid, PidRefStack \*ref)

{

PidRefStack \*prevRef = pid->RemovePrevPidRef(ref);

Assert(prevRef);

if (prevRef->GetSym() == nullptr)

{

AllocatorDelete(ArenaAllocator, &m\_nodeAllocator, prevRef);

}

}

void Parser::SetPidRefsInScopeDynamic(IdentPtr pid, int blockId)

{

PidRefStack \*ref = pid->GetTopRef();

while (ref && ref->GetScopeId() >= blockId)

{

ref->SetDynamicBinding();

ref = ref->prev;

}

}

ParseNode\* Parser::GetFunctionBlock()

{

Assert(m\_currentBlockInfo != nullptr);

return m\_currentBlockInfo->pBlockInfoFunction->pnodeBlock;

}

ParseNode\* Parser::GetCurrentBlock()

{

return m\_currentBlockInfo != nullptr ? m\_currentBlockInfo->pnodeBlock : nullptr;

}

BlockInfoStack\* Parser::GetCurrentBlockInfo()

{

return m\_currentBlockInfo;

}

BlockInfoStack\* Parser::GetCurrentFunctionBlockInfo()

{

return m\_currentBlockInfo->pBlockInfoFunction;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Parse a variable declaration.

'fAllowIn' indicates if the 'in' operator should be allowed in the initializing

expression ( it is not allowed in the context of the first expression in a 'for' loop).

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

template<bool buildAST>

ParseNodePtr Parser::ParseVariableDeclaration(

tokens declarationType, charcount\_t ichMin,

BOOL fAllowIn/\* = TRUE\*/,

BOOL\* pfForInOk/\* = nullptr\*/,

BOOL singleDefOnly/\* = FALSE\*/,

BOOL allowInit/\* = TRUE\*/,

BOOL isTopVarParse/\* = TRUE\*/,

BOOL isFor/\* = FALSE\*/)

{

ParseNodePtr pnodeThis = nullptr;

ParseNodePtr pnodeInit;

ParseNodePtr pnodeList = nullptr;

ParseNodePtr \*lastNodeRef = nullptr;

LPCOLESTR pNameHint = nullptr;

ulong nameHintLength = 0;

ulong nameHintOffset = 0;

Assert(declarationType == tkVAR || declarationType == tkCONST || declarationType == tkLET);

for (;;)

{

if (IsES6DestructuringEnabled() && IsPossiblePatternStart())

{

pnodeThis = ParseDestructuredLiteral<buildAST>(declarationType, true, !!isTopVarParse, DIC\_None, !!fAllowIn, pfForInOk);

if (pnodeThis != nullptr)

{

pnodeThis->ichMin = ichMin;

}

}

else

{

if (m\_token.tk != tkID)

{

IdentifierExpectedError(m\_token);

}

IdentPtr pid = m\_token.GetIdentifier(m\_phtbl);

Assert(pid);

pNameHint = pid->Psz();

nameHintLength = pid->Cch();

nameHintOffset = 0;

if (buildAST || BindDeferredPidRefs())

{

if (declarationType == tkVAR)

{

pnodeThis = CreateVarDeclNode(pid, STVariable);

}

else if (declarationType == tkCONST)

{

pnodeThis = CreateBlockScopedDeclNode(pid, knopConstDecl);

CHAKRATEL\_LANGSTATS\_INC\_LANGFEATURECOUNT(ConstCount, m\_scriptContext);

}

else

{

pnodeThis = CreateBlockScopedDeclNode(pid, knopLetDecl);

CHAKRATEL\_LANGSTATS\_INC\_LANGFEATURECOUNT(LetCount, m\_scriptContext);

}

}

else if (!buildAST)

{

CheckPidIsValid(pid);

}

if (pid == wellKnownPropertyPids.arguments && m\_currentNodeFunc)

{

// This var declaration may change the way an 'arguments' identifier in the function is resolved

if (declarationType == tkVAR)

{

m\_currentNodeFunc->grfpn |= PNodeFlags::fpnArguments\_varDeclaration;

}

else

{

if (GetCurrentBlockInfo()->pnodeBlock->sxBlock.blockType == Function)

{

// Only override arguments if we are at the function block level.

m\_currentNodeFunc->grfpn |= PNodeFlags::fpnArguments\_overriddenByDecl;

}

}

}

if (pnodeThis)

{

pnodeThis->ichMin = ichMin;

}

m\_pscan->Scan();

if (m\_token.tk == tkAsg)

{

if (!allowInit)

{

Error(ERRUnexpectedDefault);

}

if (pfForInOk && (declarationType == tkLET || declarationType == tkCONST))

{

\*pfForInOk = FALSE;

}

m\_pscan->Scan();

pnodeInit = ParseExpr<buildAST>(koplCma, nullptr, fAllowIn, FALSE, pNameHint, &nameHintLength, &nameHintOffset);

if (buildAST)

{

AnalysisAssert(pnodeThis);

pnodeThis->sxVar.pnodeInit = pnodeInit;

pnodeThis->ichLim = pnodeInit->ichLim;

if (pnodeInit->nop == knopFncDecl)

{

Assert(nameHintLength >= nameHintOffset);

pnodeInit->sxFnc.hint = pNameHint;

pnodeInit->sxFnc.hintLength = nameHintLength;

pnodeInit->sxFnc.hintOffset = nameHintOffset;

}

else

{

this->CheckArguments(pnodeInit);

}

pNameHint = nullptr;

}

//Track var a =, let a= , const a =

// This is for FixedFields Constant Heuristics

if (pnodeThis && pnodeThis->sxVar.pnodeInit != nullptr)

{

pnodeThis->sxVar.sym->PromoteAssignmentState();

}

}

else if (declarationType == tkCONST /\*pnodeThis->nop == knopConstDecl\*/

&& !singleDefOnly

&& !(isFor && TokIsForInOrForOf()))

{

Error(ERRUninitializedConst);

}

}

if (singleDefOnly)

{

return pnodeThis;

}

if (buildAST)

{

AddToNodeListEscapedUse(&pnodeList, &lastNodeRef, pnodeThis);

}

if (m\_token.tk != tkComma)

{

return pnodeList;

}

if (pfForInOk)

{

// don't allow "for (var a, b in c)"

\*pfForInOk = FALSE;

}

m\_pscan->Scan();

ichMin = m\_pscan->IchMinTok();

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Parse try-catch-finally statement

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Eze try-catch-finally tree nests the try-catch within a try-finally.

// This matches the new runtime implementation.

template<bool buildAST>

ParseNodePtr Parser::ParseTryCatchFinally()

{

this->m\_tryCatchOrFinallyDepth++;

ParseNodePtr pnodeT = ParseTry<buildAST>();

ParseNodePtr pnodeTC = nullptr;

StmtNest stmt;

bool hasCatch = false;

if (tkCATCH == m\_token.tk)

{

hasCatch = true;

if (buildAST)

{

pnodeTC = CreateNodeWithScanner<knopTryCatch>();

pnodeT->sxStmt.pnodeOuter = pnodeTC;

pnodeTC->sxTryCatch.pnodeTry = pnodeT;

}

PushStmt<buildAST>(&stmt, pnodeTC, knopTryCatch, nullptr, nullptr);

ParseNodePtr pnodeCatch = ParseCatch<buildAST>();

if (buildAST)

{

pnodeTC->sxTryCatch.pnodeCatch = pnodeCatch;

}

PopStmt(&stmt);

}

if (tkFINALLY != m\_token.tk)

{

if (!hasCatch)

{

Error(ERRnoCatch);

}

Assert(!buildAST || pnodeTC);

return pnodeTC;

}

ParseNodePtr pnodeTF = nullptr;

if (buildAST)

{

pnodeTF = CreateNode(knopTryFinally);

}

PushStmt<buildAST>(&stmt, pnodeTF, knopTryFinally, nullptr, nullptr);

ParseNodePtr pnodeFinally = ParseFinally<buildAST>();

if (buildAST)

{

if (!hasCatch)

{

pnodeTF->sxTryFinally.pnodeTry = pnodeT;

pnodeT->sxStmt.pnodeOuter = pnodeTF;

}

else

{

pnodeTF->sxTryFinally.pnodeTry = CreateNode(knopTry);

pnodeTF->sxTryFinally.pnodeTry->sxStmt.pnodeOuter = pnodeTF;

pnodeTF->sxTryFinally.pnodeTry->sxTry.pnodeBody = pnodeTC;

pnodeTC->sxStmt.pnodeOuter = pnodeTF->sxTryFinally.pnodeTry;

}

pnodeTF->sxTryFinally.pnodeFinally = pnodeFinally;

}

PopStmt(&stmt);

this->m\_tryCatchOrFinallyDepth--;

return pnodeTF;

}

template<bool buildAST>

ParseNodePtr Parser::ParseTry()

{

ParseNodePtr pnode = nullptr;

StmtNest stmt;

Assert(tkTRY == m\_token.tk);

if (buildAST)

{

pnode = CreateNode(knopTry);

}

m\_pscan->Scan();

if (tkLCurly != m\_token.tk)

{

Error(ERRnoLcurly);

}

PushStmt<buildAST>(&stmt, pnode, knopTry, nullptr, nullptr);

ParseNodePtr pnodeBody = ParseStatement<buildAST>();

if (buildAST)

{

pnode->sxTry.pnodeBody = pnodeBody;

if (pnode->sxTry.pnodeBody)

pnode->ichLim = pnode->sxTry.pnodeBody->ichLim;

}

PopStmt(&stmt);

return pnode;

}

template<bool buildAST>

ParseNodePtr Parser::ParseFinally()

{

ParseNodePtr pnode = nullptr;

StmtNest stmt;

Assert(tkFINALLY == m\_token.tk);

if (buildAST)

{

pnode = CreateNode(knopFinally);

}

m\_pscan->Scan();

if (tkLCurly != m\_token.tk)

{

Error(ERRnoLcurly);

}

PushStmt<buildAST>(&stmt, pnode, knopFinally, nullptr, nullptr);

ParseNodePtr pnodeBody = ParseStatement<buildAST>();

if (buildAST)

{

pnode->sxFinally.pnodeBody = pnodeBody;

if (!pnode->sxFinally.pnodeBody)

// Will only occur due to error correction.

pnode->sxFinally.pnodeBody = CreateNodeWithScanner<knopEmpty>();

else

pnode->ichLim = pnode->sxFinally.pnodeBody->ichLim;

}

PopStmt(&stmt);

return pnode;

}

template<bool buildAST>

ParseNodePtr Parser::ParseCatch()

{

ParseNodePtr rootNode = nullptr;

ParseNodePtr\* ppnode = &rootNode;

ParseNodePtr \*ppnodeExprScopeSave = nullptr;

ParseNodePtr pnode = nullptr;

ParseNodePtr pnodeCatchScope = nullptr;

StmtNest stmt;

IdentPtr pidCatch = nullptr;

//while (tkCATCH == m\_token.tk)

if (tkCATCH == m\_token.tk)

{

charcount\_t ichMin;

if (buildAST)

{

ichMin = m\_pscan->IchMinTok();

}

m\_pscan->Scan(); //catch

ChkCurTok(tkLParen, ERRnoLparen); //catch(

bool isPattern = false;

if (tkID != m\_token.tk)

{

isPattern = IsES6DestructuringEnabled() && IsPossiblePatternStart();

if (!isPattern)

{

IdentifierExpectedError(m\_token);

}

}

if (buildAST)

{

pnode = CreateNodeWithScanner<knopCatch>(ichMin);

PushStmt<buildAST>(&stmt, pnode, knopCatch, nullptr, nullptr);

\*ppnode = pnode;

ppnode = &pnode->sxCatch.pnodeNext;

\*ppnode = nullptr;

}

if (buildAST || BindDeferredPidRefs())

{

pnodeCatchScope = StartParseBlock<buildAST>(PnodeBlockType::Regular, isPattern ? ScopeType\_CatchParamPattern : ScopeType\_Catch);

}

if (isPattern)

{

ParseNodePtr pnodePattern = ParseDestructuredLiteral<buildAST>(tkLET, true /\*isDecl\*/, true /\*topLevel\*/, DIC\_ForceErrorOnInitializer);

if (buildAST)

{

pnode->sxCatch.pnodeParam = CreateParamPatternNode(pnodePattern);

Scope \*scope = pnodeCatchScope->sxBlock.scope;

pnode->sxCatch.scope = scope;

}

}

else

{

if (IsStrictMode())

{

IdentPtr pid = m\_token.GetIdentifier(m\_phtbl);

if (pid == wellKnownPropertyPids.eval)

{

Error(ERREvalUsage);

}

else if (pid == wellKnownPropertyPids.arguments)

{

Error(ERRArgsUsage);

}

}

if (buildAST)

{

pidCatch = m\_token.GetIdentifier(m\_phtbl);

PidRefStack \*ref = this->PushPidRef(pidCatch);

if (!m\_scriptContext->GetConfig()->IsBlockScopeEnabled())

{

// Strange case: the catch adds a scope for the catch object, but function declarations

// are hoisted out of the catch, so references within a function declaration to "x" do

// not bind to "catch(x)". Extra bookkeeping is required.

CatchPidRefList \*list = this->EnsureCatchPidRefList();

CatchPidRef \*catchPidRef = list->PrependNode(&m\_nodeAllocator);

catchPidRef->pid = pidCatch;

catchPidRef->ref = ref;

}

ParseNodePtr pnodeParam = CreateNameNode(pidCatch);

pnodeParam->sxPid.symRef = ref->GetSymRef();

pnode->sxCatch.pnodeParam = pnodeParam;

const wchar\_t \*name = reinterpret\_cast<const wchar\_t\*>(pidCatch->Psz());

int nameLength = pidCatch->Cch();

SymbolName const symName(name, nameLength);

Symbol \*sym = Anew(&m\_nodeAllocator, Symbol, symName, pnodeParam, STVariable);

sym->SetPid(pidCatch);

if (sym == nullptr)

{

Error(ERRnoMemory);

}

Assert(ref->GetSym() == nullptr);

ref->SetSym(sym);

Scope \*scope = pnodeCatchScope->sxBlock.scope;

scope->AddNewSymbol(sym);

pnode->sxCatch.scope = scope;

}

m\_pscan->Scan();

}

if (buildAST)

{

// Add this catch to the current scope list.

if (m\_ppnodeExprScope)

{

Assert(\*m\_ppnodeExprScope == nullptr);

\*m\_ppnodeExprScope = pnode;

m\_ppnodeExprScope = &pnode->sxCatch.pnodeNext;

}

else

{

Assert(m\_ppnodeScope);

Assert(\*m\_ppnodeScope == nullptr);

\*m\_ppnodeScope = pnode;

m\_ppnodeScope = &pnode->sxCatch.pnodeNext;

}

// Keep a list of function expressions (not declarations) at this scope.

ppnodeExprScopeSave = m\_ppnodeExprScope;

m\_ppnodeExprScope = &pnode->sxCatch.pnodeScopes;

pnode->sxCatch.pnodeScopes = nullptr;

}

charcount\_t ichLim;

if (buildAST)

{

ichLim = m\_pscan->IchLimTok();

}

ChkCurTok(tkRParen, ERRnoRparen); //catch(id[:expr])

if (tkLCurly != m\_token.tk)

{

Error(ERRnoLcurly);

}

ParseNodePtr pnodeBody = ParseStatement<buildAST>(); //catch(id[:expr]) {block}

if (buildAST)

{

pnode->sxCatch.pnodeBody = pnodeBody;

pnode->ichLim = ichLim;

}

if (pnodeCatchScope != nullptr)

{

FinishParseBlock(pnodeCatchScope);

}

if (buildAST)

{

PopStmt(&stmt);

// Restore the lists of function expression scopes.

AssertMem(m\_ppnodeExprScope);

Assert(\*m\_ppnodeExprScope == nullptr);

m\_ppnodeExprScope = ppnodeExprScopeSave;

if (!m\_scriptContext->GetConfig()->IsBlockScopeEnabled())

{

// Remove the catch object from the list.

CatchPidRefList \*list = this->GetCatchPidRefList();

Assert(list);

Assert(!list->Empty());

Assert(list->Head().pid == pidCatch);

list->RemoveHead(&m\_nodeAllocator);

}

}

}

return rootNode;

}

template<bool buildAST>

ParseNodePtr Parser::ParseCase(ParseNodePtr \*ppnodeBody)

{

ParseNodePtr pnodeT = nullptr;

charcount\_t ichMinT = m\_pscan->IchMinTok();

m\_pscan->Scan();

ParseNodePtr pnodeExpr = ParseExpr<buildAST>();

charcount\_t ichLim = m\_pscan->IchLimTok();

ChkCurTok(tkColon, ERRnoColon);

if (buildAST)

{

pnodeT = CreateNodeWithScanner<knopCase>(ichMinT);

pnodeT->sxCase.pnodeExpr = pnodeExpr;

pnodeT->ichLim = ichLim;

}

ParseStmtList<buildAST>(ppnodeBody);

return pnodeT;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Parse a single statement. Digest a trailing semicolon.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

template<bool buildAST>

ParseNodePtr Parser::ParseStatement(bool isSourceElement/\* = false\*/)

{

ParseNodePtr \*ppnodeT;

ParseNodePtr pnodeT;

ParseNodePtr pnode = nullptr;

LabelId\* pLabelIdList = nullptr;

charcount\_t ichMin = 0;

size\_t iecpMin = 0;

StmtNest stmt;

StmtNest \*pstmt;

BOOL fForInOrOfOkay;

BOOL fCanAssign;

IdentPtr pid;

uint fnop;

ParseNodePtr pnodeLabel = nullptr;

bool expressionStmt = false;

bool isAsyncMethod = false;

tokens tok;

#if EXCEPTION\_RECOVERY

ParseNodePtr pParentTryCatch = nullptr;

ParseNodePtr pTryBlock = nullptr;

ParseNodePtr pTry = nullptr;

ParseNodePtr pParentTryCatchBlock = nullptr;

StmtNest stmtTryCatchBlock;

StmtNest stmtTryCatch;

StmtNest stmtTry;

StmtNest stmtTryBlock;

#endif

if (buildAST)

{

#if EXCEPTION\_RECOVERY

if(Js::Configuration::Global.flags.SwallowExceptions)

{

// If we're swallowing exceptions, surround this statement with a try/catch block:

//

// Before: x.y = 3;

// After: try { x.y = 3; } catch(\_\_ehobj) { }

//

// This is done to force the runtime to recover from exceptions at the most granular

// possible point. Recovering from EH dramatically improves coverage of testing via

// fault injection.

// create and push the try-catch node

pParentTryCatchBlock = CreateBlockNode();

PushStmt<buildAST>(&stmtTryCatchBlock, pParentTryCatchBlock, knopBlock, nullptr, nullptr);

pParentTryCatch = CreateNodeWithScanner<knopTryCatch>();

PushStmt<buildAST>(&stmtTryCatch, pParentTryCatch, knopTryCatch, nullptr, nullptr);

// create and push a try node

pTry = CreateNodeWithScanner<knopTry>();

PushStmt<buildAST>(&stmtTry, pTry, knopTry, nullptr, nullptr);

pTryBlock = CreateBlockNode();

PushStmt<buildAST>(&stmtTryBlock, pTryBlock, knopBlock, nullptr, nullptr);

// these nodes will be closed after the statement is parsed.

}

#endif // EXCEPTION\_RECOVERY

}

EnsureStackAvailable();

LRestart:

tok = m\_token.tk;

switch (tok)

{

case tkEOF:

if (buildAST)

{

pnode = nullptr;

}

break;

case tkFUNCTION:

{

LFunctionStatement:

if (m\_grfscr & fscrDeferredFncExpression)

{

// The top-level deferred function body was defined by a function expression whose parsing was deferred. We are now

// parsing it, so unset the flag so that any nested functions are parsed normally. This flag is only applicable the

// first time we see it.

m\_grfscr &= ~fscrDeferredFncExpression;

pnode = ParseFncDecl<buildAST>(isAsyncMethod ? fFncAsync : fFncNoFlgs, nullptr, isSourceElement);

}

else

{

pnode = ParseFncDecl<buildAST>(fFncDeclaration | (isAsyncMethod ? fFncAsync : fFncNoFlgs), nullptr, isSourceElement);

}

if (isAsyncMethod)

{

pnode->sxFnc.cbMin = iecpMin;

pnode->ichMin = ichMin;

}

break;

}

case tkCLASS:

if (m\_scriptContext->GetConfig()->IsES6ClassAndExtendsEnabled())

{

pnode = ParseClassDecl<buildAST>(TRUE, nullptr, nullptr, nullptr);

}

else

{

goto LDefaultToken;

}

break;

case tkID:

if (m\_token.GetIdentifier(m\_phtbl) == wellKnownPropertyPids.let && m\_scriptContext->GetConfig()->IsLetAndConstEnabled())

{

// We see "let" at the start of a statement. This could either be a declaration or an identifier

// reference. The next token determines which.

RestorePoint parsedLet;

m\_pscan->Capture(&parsedLet);

ichMin = m\_pscan->IchMinTok();

m\_pscan->Scan();

if (this->NextTokenConfirmsLetDecl())

{

pnode = ParseVariableDeclaration<buildAST>(tkLET, ichMin);

goto LNeedTerminator;

}

m\_pscan->SeekTo(parsedLet);

}

else if (m\_token.GetIdentifier(m\_phtbl) == wellKnownPropertyPids.async && m\_scriptContext->GetConfig()->IsES7AsyncAndAwaitEnabled())

{

RestorePoint parsedAsync;

m\_pscan->Capture(&parsedAsync);

ichMin = m\_pscan->IchMinTok();

iecpMin = m\_pscan->IecpMinTok();

m\_pscan->Scan();

if (m\_token.tk == tkFUNCTION)

{

isAsyncMethod = true;

goto LFunctionStatement;

}

m\_pscan->SeekTo(parsedAsync);

}

goto LDefaultToken;

case tkCONST:

case tkLET:

if (m\_scriptContext->GetConfig()->IsLetAndConstEnabled())

{

ichMin = m\_pscan->IchMinTok();

m\_pscan->Scan();

pnode = ParseVariableDeclaration<buildAST>(tok, ichMin);

goto LNeedTerminator;

}

else

{

goto LDefaultToken;

}

case tkVAR:

ichMin = m\_pscan->IchMinTok();

m\_pscan->Scan();

pnode = ParseVariableDeclaration<buildAST>(tok, ichMin);

goto LNeedTerminator;

case tkFOR:

{

ParseNodePtr pnodeBlock = nullptr;

ParseNodePtr \*ppnodeScopeSave = nullptr;

ParseNodePtr \*ppnodeExprScopeSave = nullptr;

ichMin = m\_pscan->IchMinTok();

ChkNxtTok(tkLParen, ERRnoLparen);

if (buildAST || BindDeferredPidRefs())

{

pnodeBlock = StartParseBlock<buildAST>(PnodeBlockType::Regular, ScopeType\_Block);

if (buildAST)

{

PushFuncBlockScope(pnodeBlock, &ppnodeScopeSave, &ppnodeExprScopeSave);

}

}

RestorePoint startExprOrIdentifier;

fForInOrOfOkay = TRUE;

fCanAssign = TRUE;

tok = m\_token.tk;

switch (tok)

{

case tkID:

if (m\_token.GetIdentifier(m\_phtbl) == wellKnownPropertyPids.let && m\_scriptContext->GetConfig()->IsLetAndConstEnabled())

{

// We see "let" in the init part of a for loop. This could either be a declaration or an identifier

// reference. The next token determines which.

RestorePoint parsedLet;

m\_pscan->Capture(&parsedLet);

auto ichMin = m\_pscan->IchMinTok();

m\_pscan->Scan();

if (IsPossiblePatternStart())

{

m\_pscan->Capture(&startExprOrIdentifier);

}

if (this->NextTokenConfirmsLetDecl() && m\_token.tk != tkIN)

{

pnodeT = ParseVariableDeclaration<buildAST>(tkLET, ichMin

, /\*fAllowIn = \*/FALSE

, /\*pfForInOk = \*/&fForInOrOfOkay

, /\*singleDefOnly\*/FALSE

, /\*allowInit\*/TRUE

, /\*isTopVarParse\*/FALSE

, /\*isFor\*/TRUE);

break;

}

m\_pscan->SeekTo(parsedLet);

}

goto LDefaultTokenFor;

case tkLET:

case tkCONST:

if (!m\_scriptContext->GetConfig()->IsLetAndConstEnabled())

{

goto LDefaultTokenFor;

}

case tkVAR:

{

auto ichMin = m\_pscan->IchMinTok();

m\_pscan->Scan();

if (IsPossiblePatternStart())

{

m\_pscan->Capture(&startExprOrIdentifier);

}

pnodeT = ParseVariableDeclaration<buildAST>(tok, ichMin

, /\*fAllowIn = \*/FALSE

, /\*pfForInOk = \*/&fForInOrOfOkay

, /\*singleDefOnly\*/FALSE

, /\*allowInit\*/TRUE

, /\*isTopVarParse\*/FALSE

, /\*isFor\*/TRUE);

}

break;

case tkSColon:

pnodeT = nullptr;

fForInOrOfOkay = FALSE;

break;

default:

{

LDefaultTokenFor:

RestorePoint exprStart;

tokens beforeToken = tok;

m\_pscan->Capture(&exprStart);

if (IsPossiblePatternStart())

{

m\_pscan->Capture(&startExprOrIdentifier);

}

bool fLikelyPattern = false;

if (IsES6DestructuringEnabled() && (beforeToken == tkLBrack || beforeToken == tkLCurly))

{

pnodeT = ParseExpr<buildAST>(koplNo,

&fCanAssign,

/\*fAllowIn = \*/FALSE,

/\*fAllowEllipsis\*/FALSE,

/\*pHint\*/nullptr,

/\*pHintLength\*/nullptr,

/\*pShortNameOffset\*/nullptr,

/\*pToken\*/nullptr,

/\*\*fUnaryOrParen\*/false,

&fLikelyPattern);

}

else

{

pnodeT = ParseExpr<buildAST>(koplNo, &fCanAssign, /\*fAllowIn = \*/FALSE);

}

if (fLikelyPattern)

{

m\_pscan->SeekTo(exprStart);

ParseDestructuredLiteralWithScopeSave(tkNone, false/\*isDecl\*/, false /\*topLevel\*/, DIC\_None, false /\*allowIn\*/);

if (buildAST)

{

pnodeT = ConvertToPattern(pnodeT);

}

}

if (buildAST)

{

Assert(pnodeT);

pnodeT->isUsed = false;

}

}

break;

}

if (TokIsForInOrForOf())

{

bool isForOf = (m\_token.tk != tkIN);

Assert(!isForOf || (m\_token.tk == tkID && m\_token.GetIdentifier(m\_phtbl) == wellKnownPropertyPids.of));

if ((buildAST && nullptr == pnodeT) || !fForInOrOfOkay)

{

if (isForOf)

{

Error(ERRForOfNoInitAllowed);

}

else

{

Error(ERRForInNoInitAllowed);

}

}

if (!fCanAssign && PHASE\_ON1(Js::EarlyReferenceErrorsPhase))

{

Error(JSERR\_CantAssignTo);

}

m\_pscan->Scan();

ParseNodePtr pnodeObj = ParseExpr<buildAST>();

charcount\_t ichLim = m\_pscan->IchLimTok();

ChkCurTok(tkRParen, ERRnoRparen);

if (buildAST)

{

if (isForOf)

{

pnode = CreateNodeWithScanner<knopForOf>(ichMin);

}

else

{

pnode = CreateNodeWithScanner<knopForIn>(ichMin);

}

pnode->sxForInOrForOf.pnodeBlock = pnodeBlock;

pnode->sxForInOrForOf.pnodeLval = pnodeT;

pnode->sxForInOrForOf.pnodeObj = pnodeObj;

pnode->ichLim = ichLim;

}

PushStmt<buildAST>(&stmt, pnode, isForOf ? knopForOf : knopForIn, pnodeLabel, pLabelIdList);

ParseNodePtr pnodeBody = ParseStatement<buildAST>();

if (buildAST)

{

pnode->sxForInOrForOf.pnodeBody = pnodeBody;

}

PopStmt(&stmt);

}

else

{

ChkCurTok(tkSColon, ERRnoSemic);

ParseNodePtr pnodeCond = nullptr;

if (m\_token.tk != tkSColon)

{

pnodeCond = ParseExpr<buildAST>();

if (m\_token.tk != tkSColon)

{

Error(ERRnoSemic);

}

}

tokens tk;

tk = m\_pscan->Scan();

ParseNodePtr pnodeIncr = nullptr;

if (tk != tkRParen)

{

pnodeIncr = ParseExpr<buildAST>();

if(pnodeIncr)

{

pnodeIncr->isUsed = false;

}

}

charcount\_t ichLim = m\_pscan->IchLimTok();

ChkCurTok(tkRParen, ERRnoRparen);

if (buildAST)

{

pnode = CreateNodeWithScanner<knopFor>(ichMin);

pnode->sxFor.pnodeBlock = pnodeBlock;

pnode->sxFor.pnodeInverted= nullptr;

pnode->sxFor.pnodeInit = pnodeT;

pnode->sxFor.pnodeCond = pnodeCond;

pnode->sxFor.pnodeIncr = pnodeIncr;

pnode->ichLim = ichLim;

}

PushStmt<buildAST>(&stmt, pnode, knopFor, pnodeLabel, pLabelIdList);

ParseNodePtr pnodeBody = ParseStatement<buildAST>();

if (buildAST)

{

pnode->sxFor.pnodeBody = pnodeBody;

}

PopStmt(&stmt);

}

if (buildAST)

{

PopFuncBlockScope(ppnodeScopeSave, ppnodeExprScopeSave);

FinishParseBlock(pnodeBlock);

}

else if (BindDeferredPidRefs())

{

FinishParseBlock(pnodeBlock);

}

break;

}

case tkSWITCH:

{

BOOL fSeenDefault = FALSE;

StmtNest stmtBlock;

ParseNodePtr pnodeBlock = nullptr;

ParseNodePtr \*ppnodeScopeSave = nullptr;

ParseNodePtr \*ppnodeExprScopeSave = nullptr;

ichMin = m\_pscan->IchMinTok();

ChkNxtTok(tkLParen, ERRnoLparen);

ParseNodePtr pnodeVal = ParseExpr<buildAST>();

charcount\_t ichLim = m\_pscan->IchLimTok();

ChkCurTok(tkRParen, ERRnoRparen);

ChkCurTok(tkLCurly, ERRnoLcurly);

if (buildAST)

{

pnode = CreateNodeWithScanner<knopSwitch>(ichMin);

}

PushStmt<buildAST>(&stmt, pnode, knopSwitch, pnodeLabel, pLabelIdList);

if (buildAST || BindDeferredPidRefs())

{

pnodeBlock = StartParseBlock<buildAST>(PnodeBlockType::Regular, ScopeType\_Block, nullptr, pLabelIdList);

}

else

{

PushStmt<buildAST>(&stmtBlock, nullptr, knopBlock, nullptr, pLabelIdList);

}

if (buildAST)

{

pnode->sxSwitch.pnodeVal = pnodeVal;

pnode->sxSwitch.pnodeBlock = pnodeBlock;

pnode->ichLim = ichLim;

PushFuncBlockScope(pnode->sxSwitch.pnodeBlock, &ppnodeScopeSave, &ppnodeExprScopeSave);

pnode->sxSwitch.pnodeDefault = nullptr;

ppnodeT = &pnode->sxSwitch.pnodeCases;

}

for (;;)

{

ParseNodePtr pnodeBody = nullptr;

switch (m\_token.tk)

{

default:

goto LEndSwitch;

case tkCASE:

{

pnodeT = this->ParseCase<buildAST>(&pnodeBody);

break;

}

case tkDEFAULT:

if (fSeenDefault)

{

Error(ERRdupDefault);

// No recovery necessary since this is a semantic, not structural, error

}

fSeenDefault = TRUE;

charcount\_t ichMinT = m\_pscan->IchMinTok();

m\_pscan->Scan();

charcount\_t ichLim = m\_pscan->IchLimTok();

ChkCurTok(tkColon, ERRnoColon);

if (buildAST)

{

pnodeT = CreateNodeWithScanner<knopCase>(ichMinT);

pnode->sxSwitch.pnodeDefault = pnodeT;

pnodeT->ichLim = ichLim;

pnodeT->sxCase.pnodeExpr = nullptr;

}

ParseStmtList<buildAST>(&pnodeBody);

break;

}

if (buildAST)

{

if (pnodeBody)

{

// Create a block node to contain the statement list for this case.

// This helps us insert byte code to return the right value from

// global/eval code.

pnodeT->sxCase.pnodeBody = CreateBlockNode(pnodeT->ichMin, pnodeT->ichLim);

pnodeT->sxCase.pnodeBody->grfpn |= PNodeFlags::fpnSyntheticNode; // block is not a user specifier block

pnodeT->sxCase.pnodeBody->sxBlock.pnodeStmt = pnodeBody;

}

else

{

pnodeT->sxCase.pnodeBody = nullptr;

}

\*ppnodeT = pnodeT;

ppnodeT = &pnodeT->sxCase.pnodeNext;

}

}

LEndSwitch:

ChkCurTok(tkRCurly, ERRnoRcurly);

if (buildAST)

{

\*ppnodeT = nullptr;

PopFuncBlockScope(ppnodeScopeSave, ppnodeExprScopeSave);

FinishParseBlock(pnode->sxSwitch.pnodeBlock);

}

else

{

if (BindDeferredPidRefs())

{

FinishParseBlock(pnodeBlock);

}

else

{

PopStmt(&stmtBlock);

}

}

PopStmt(&stmt);

break;

}

case tkWHILE:

{

ichMin = m\_pscan->IchMinTok();

ChkNxtTok(tkLParen, ERRnoLparen);

ParseNodePtr pnodeCond = ParseExpr<buildAST>();

charcount\_t ichLim = m\_pscan->IchLimTok();

ChkCurTok(tkRParen, ERRnoRparen);

if (buildAST)

{

pnode = CreateNodeWithScanner<knopWhile>(ichMin);

pnode->sxWhile.pnodeCond = pnodeCond;

pnode->ichLim = ichLim;

}

PushStmt<buildAST>(&stmt, pnode, knopWhile, pnodeLabel, pLabelIdList);

ParseNodePtr pnodeBody = ParseStatement<buildAST>();

PopStmt(&stmt);

if (buildAST)

{

pnode->sxWhile.pnodeBody = pnodeBody;

}

break;

}

case tkDO:

{

if (buildAST)

{

pnode = CreateNodeWithScanner<knopDoWhile>();

}

PushStmt<buildAST>(&stmt, pnode, knopDoWhile, pnodeLabel, pLabelIdList);

m\_pscan->Scan();

ParseNodePtr pnodeBody = ParseStatement<buildAST>();

PopStmt(&stmt);

charcount\_t ichMinT = m\_pscan->IchMinTok();

ChkCurTok(tkWHILE, ERRnoWhile);

ChkCurTok(tkLParen, ERRnoLparen);

ParseNodePtr pnodeCond = ParseExpr<buildAST>();

charcount\_t ichLim = m\_pscan->IchLimTok();

ChkCurTok(tkRParen, ERRnoRparen);

if (buildAST)

{

pnode->sxWhile.pnodeBody = pnodeBody;

pnode->sxWhile.pnodeCond = pnodeCond;

pnode->ichLim = ichLim;

pnode->ichMin = ichMinT;

}

// REVIEW: Allow do...while statements to be embedded in other compound statements like if..else, or do..while?

// goto LNeedTerminator;

// For now just eat the trailing semicolon if present.

if (m\_token.tk == tkSColon)

{

if (pnode)

{

pnode->grfpn |= PNodeFlags::fpnExplicitSimicolon;

}

m\_pscan->Scan();

}

else if (pnode)

{

pnode->grfpn |= PNodeFlags::fpnAutomaticSimicolon;

}

break;

}

case tkIF:

{

ichMin = m\_pscan->IchMinTok();

ChkNxtTok(tkLParen, ERRnoLparen);

ParseNodePtr pnodeCond = ParseExpr<buildAST>();

if (buildAST)

{

pnode = CreateNodeWithScanner<knopIf>(ichMin);

pnode->ichLim = m\_pscan->IchLimTok();

pnode->sxIf.pnodeCond = pnodeCond;

}

ChkCurTok(tkRParen, ERRnoRparen);

PushStmt<buildAST>(&stmt, pnode, knopIf, pnodeLabel, pLabelIdList);

ParseNodePtr pnodeTrue = ParseStatement<buildAST>();

ParseNodePtr pnodeFalse = nullptr;

if (m\_token.tk == tkELSE)

{

m\_pscan->Scan();

pnodeFalse = ParseStatement<buildAST>();

}

if (buildAST)

{

pnode->sxIf.pnodeTrue = pnodeTrue;

pnode->sxIf.pnodeFalse = pnodeFalse;

}

PopStmt(&stmt);

break;

}

case tkTRY:

{

if (buildAST)

{

pnode = CreateBlockNode();

pnode->grfpn |= PNodeFlags::fpnSyntheticNode; // block is not a user specifier block

}

PushStmt<buildAST>(&stmt, pnode, knopBlock, pnodeLabel, pLabelIdList);

ParseNodePtr pnodeStmt = ParseTryCatchFinally<buildAST>();

if (buildAST)

{

pnode->sxBlock.pnodeStmt = pnodeStmt;

}

PopStmt(&stmt);

break;

}

case tkWITH:

{

if ( IsStrictMode() )

{

Error(ERRES5NoWith);

}

if (m\_currentNodeFunc)

{

GetCurrentFunctionNode()->sxFnc.SetHasWithStmt(); // Used by DeferNested

}

ichMin = m\_pscan->IchMinTok();

ChkNxtTok(tkLParen, ERRnoLparen);

ParseNodePtr pnodeObj = ParseExpr<buildAST>();

if (!buildAST)

{

m\_scopeCountNoAst++;

}

charcount\_t ichLim = m\_pscan->IchLimTok();

ChkCurTok(tkRParen, ERRnoRparen);

if (buildAST)

{

pnode = CreateNodeWithScanner<knopWith>(ichMin);

}

PushStmt<buildAST>(&stmt, pnode, knopWith, pnodeLabel, pLabelIdList);

ParseNodePtr \*ppnodeExprScopeSave = nullptr;

if (buildAST)

{

pnode->sxWith.pnodeObj = pnodeObj;

this->CheckArguments(pnode->sxWith.pnodeObj);

if (m\_ppnodeExprScope)

{

Assert(\*m\_ppnodeExprScope == nullptr);

\*m\_ppnodeExprScope = pnode;

m\_ppnodeExprScope = &pnode->sxWith.pnodeNext;

}

else

{

Assert(m\_ppnodeScope);

Assert(\*m\_ppnodeScope == nullptr);

\*m\_ppnodeScope = pnode;

m\_ppnodeScope = &pnode->sxWith.pnodeNext;

}

pnode->sxWith.pnodeNext = nullptr;

pnode->sxWith.scope = nullptr;

ppnodeExprScopeSave = m\_ppnodeExprScope;

m\_ppnodeExprScope = &pnode->sxWith.pnodeScopes;

pnode->sxWith.pnodeScopes = nullptr;

pnode->ichLim = ichLim;

}

if (buildAST || BindDeferredPidRefs())

{

PushBlockInfo(CreateBlockNode());

PushDynamicBlock();

}

ParseNodePtr pnodeBody = ParseStatement<buildAST>();

if (buildAST)

{

pnode->sxWith.pnodeBody = pnodeBody;

m\_ppnodeExprScope = ppnodeExprScopeSave;

}

else

{

m\_scopeCountNoAst--;

}

if (buildAST || BindDeferredPidRefs())

{

// The dynamic block is not stored in the actual parse tree and so will not

// be visited by the byte code generator. Grab the callsEval flag off it and

// pass on to outer block in case of:

// with (...) eval(...); // i.e. blockless form of with

bool callsEval = GetCurrentBlock()->sxBlock.GetCallsEval();

PopBlockInfo();

if (callsEval)

{

// be careful not to overwrite an existing true with false

GetCurrentBlock()->sxBlock.SetCallsEval(true);

}

}

PopStmt(&stmt);

break;

}

case tkLCurly:

pnode = ParseBlock<buildAST>(pnodeLabel, pLabelIdList);

break;

case tkSColon:

pnode = nullptr;

m\_pscan->Scan();

break;

case tkBREAK:

if (buildAST)

{

pnode = CreateNodeWithScanner<knopBreak>();

}

fnop = fnopBreak;

goto LGetJumpStatement;

case tkCONTINUE:

if (buildAST)

{

pnode = CreateNode(knopContinue);

}

fnop = fnopContinue;

LGetJumpStatement:

m\_pscan->ScanForcingPid();

if (tkID == m\_token.tk && !m\_pscan->FHadNewLine())

{

// Labeled break or continue.

pid = m\_token.GetIdentifier(m\_phtbl);

AssertMem(pid);

if (buildAST)

{

pnode->sxJump.hasExplicitTarget=true;

pnode->ichLim = m\_pscan->IchLimTok();

m\_pscan->Scan();

PushStmt<buildAST>(&stmt, pnode, pnode->nop, pnodeLabel, nullptr);

Assert(pnode->sxStmt.grfnop == 0);

for (pstmt = m\_pstmtCur; nullptr != pstmt; pstmt = pstmt->pstmtOuter)

{

AssertNodeMem(pstmt->pnodeStmt);

AssertNodeMemN(pstmt->pnodeLab);

for (pnodeT = pstmt->pnodeLab; nullptr != pnodeT;

pnodeT = pnodeT->sxLabel.pnodeNext)

{

Assert(knopLabel == pnodeT->nop);

if (pid == pnodeT->sxLabel.pid)

{

// Found the label. Make sure we can use it. We can

// break out of any statement, but we can only

// continue loops.

if (fnop == fnopContinue &&

!(pstmt->pnodeStmt->Grfnop() & fnop))

{

Error(ERRbadContinue);

}

else

{

pstmt->pnodeStmt->sxStmt.grfnop |= fnop;

pnode->sxJump.pnodeTarget = pstmt->pnodeStmt;

}

PopStmt(&stmt);

goto LNeedTerminator;

}

}

pnode->sxStmt.grfnop |=

(pstmt->pnodeStmt->Grfnop() & fnopCleanup);

}

}

else

{

m\_pscan->Scan();

for (pstmt = m\_pstmtCur; pstmt; pstmt = pstmt->pstmtOuter)

{

LabelId\* pLabelId;

for (pLabelId = pstmt->pLabelId; pLabelId; pLabelId = pLabelId->next)

{

if (pid == pLabelId->pid)

{

// Found the label. Make sure we can use it. We can

// break out of any statement, but we can only

// continue loops.

if (fnop == fnopContinue &&

!(ParseNode::Grfnop(pstmt->op) & fnop))

{

Error(ERRbadContinue);

}

goto LNeedTerminator;

}

}

}

}

Error(ERRnoLabel);

}

else

{

// If we're doing a fast scan, we're not tracking labels, so we can't accurately do this analysis.

// Let the thread that's doing the full parse detect the error, if there is one.

if (!this->m\_doingFastScan)

{

// Unlabeled break or continue.

if (buildAST)

{

pnode->sxJump.hasExplicitTarget=false;

PushStmt<buildAST>(&stmt, pnode, pnode->nop, pnodeLabel, nullptr);

Assert(pnode->sxStmt.grfnop == 0);

}

for (pstmt = m\_pstmtCur; nullptr != pstmt; pstmt = pstmt->pstmtOuter)

{

if (buildAST)

{

AnalysisAssert(pstmt->pnodeStmt);

if (pstmt->pnodeStmt->Grfnop() & fnop)

{

pstmt->pnodeStmt->sxStmt.grfnop |= fnop;

pnode->sxJump.pnodeTarget = pstmt->pnodeStmt;

PopStmt(&stmt);

goto LNeedTerminator;

}

pnode->sxStmt.grfnop |=

(pstmt->pnodeStmt->Grfnop() & fnopCleanup);

}

else

{

if (pstmt->isDeferred)

{

if (ParseNode::Grfnop(pstmt->op) & fnop)

{

goto LNeedTerminator;

}

}

else

{

AnalysisAssert(pstmt->pnodeStmt);

if (pstmt->pnodeStmt->Grfnop() & fnop)

{

pstmt->pnodeStmt->sxStmt.grfnop |= fnop;

goto LNeedTerminator;

}

}

}

}

Error(fnop == fnopBreak ? ERRbadBreak : ERRbadContinue);

}

goto LNeedTerminator;

}

case tkRETURN:

{

if (buildAST)

{

if (nullptr == m\_currentNodeFunc)

{

Error(ERRbadReturn);

}

pnode = CreateNodeWithScanner<knopReturn>();

}

m\_pscan->Scan();

ParseNodePtr pnodeExpr = nullptr;

ParseOptionalExpr<buildAST>(&pnodeExpr, true);

if (buildAST)

{

pnode->sxReturn.pnodeExpr = pnodeExpr;

if (pnodeExpr)

{

this->CheckArguments(pnode->sxReturn.pnodeExpr);

pnode->ichLim = pnode->sxReturn.pnodeExpr->ichLim;

}

// See if return should call finally

PushStmt<buildAST>(&stmt, pnode, knopReturn, pnodeLabel, nullptr);

Assert(pnode->sxStmt.grfnop == 0);

for (pstmt = m\_pstmtCur; nullptr != pstmt; pstmt = pstmt->pstmtOuter)

{

AssertNodeMem(pstmt->pnodeStmt);

AssertNodeMemN(pstmt->pnodeLab);

if (pstmt->pnodeStmt->Grfnop() & fnopCleanup)

{

pnode->sxStmt.grfnop |= fnopCleanup;

break;

}

}

PopStmt(&stmt);

}

goto LNeedTerminator;

}

case tkTHROW:

{

if (buildAST)

{

pnode = CreateUniNode(knopThrow, nullptr);

}

m\_pscan->Scan();

ParseNodePtr pnode1 = nullptr;

if (m\_token.tk != tkSColon &&

m\_token.tk != tkRCurly &&

!m\_pscan->FHadNewLine())

{

pnode1 = ParseExpr<buildAST>();

}

else

{

Error(ERRdanglingThrow);

}

if (buildAST)

{

pnode->sxUni.pnode1 = pnode1;

if (pnode1)

{

this->CheckArguments(pnode->sxUni.pnode1);

pnode->ichLim = pnode->sxUni.pnode1->ichLim;

}

}

goto LNeedTerminator;

}

case tkDEBUGGER:

if (buildAST)

{

pnode = CreateNodeWithScanner<knopDebugger>();

}

m\_pscan->Scan();

goto LNeedTerminator;

LDefaultToken:

default:

{

// An expression statement or a label.

IdentToken tok;

pnode = ParseExpr<buildAST>(koplNo, nullptr, TRUE, FALSE, nullptr, nullptr /\*hintLength\*/, nullptr /\*hintOffset\*/, &tok);

if (buildAST)

{

// Check for a label.

if (tkColon == m\_token.tk &&

nullptr != pnode && knopName == pnode->nop)

{

// We have a label. See if it is already defined.

if (nullptr != PnodeLabel(pnode->sxPid.pid, pnodeLabel))

{

Error(ERRbadLabel);

// No recovery is necessary since this is a semantic, not structural, error

}

pnodeT = CreateNodeWithScanner<knopLabel>();

pnodeT->sxLabel.pid = pnode->sxPid.pid;

pnodeT->sxLabel.pnodeNext = pnodeLabel;

pnodeLabel = pnodeT;

m\_pscan->Scan();

isSourceElement = false;

goto LRestart;

}

expressionStmt = true;

AnalysisAssert(pnode);

pnode->isUsed = false;

}

else

{

// Check for a label.

if (tkColon == m\_token.tk && tok.tk == tkID)

{

tok.pid = m\_pscan->PidAt(tok.ichMin, tok.ichLim);

if (PnodeLabelNoAST(&tok, pLabelIdList))

{

Error(ERRbadLabel);

}

LabelId\* pLabelId = CreateLabelId(&tok);

pLabelId->next = pLabelIdList;

pLabelIdList = pLabelId;

m\_pscan->Scan();

isSourceElement = false;

goto LRestart;

}

}

}

LNeedTerminator:

// Need a semicolon, new-line, } or end-of-file.

// We digest a semicolon if it's there.

switch (m\_token.tk)

{

case tkSColon:

m\_pscan->Scan();

if (pnode!= nullptr) pnode->grfpn |= PNodeFlags::fpnExplicitSimicolon;

break;

case tkEOF:

case tkRCurly:

if (pnode!= nullptr) pnode->grfpn |= PNodeFlags::fpnAutomaticSimicolon;

break;

default:

if (!m\_pscan->FHadNewLine())

{

Error(ERRnoSemic);

}

else

{

if (pnode!= nullptr) pnode->grfpn |= PNodeFlags::fpnAutomaticSimicolon;

}

break;

}

break;

}

if (buildAST)

{

// All non expression statements excluded from the "this.x" optimization

// Another check while parsing expressions

if (!expressionStmt)

{

if (m\_currentNodeFunc)

{

m\_currentNodeFunc->sxFnc.SetHasNonThisStmt();

}

else if (m\_currentNodeProg)

{

m\_currentNodeProg->sxFnc.SetHasNonThisStmt();

}

}

#if EXCEPTION\_RECOVERY

// close the try/catch block

if(Js::Configuration::Global.flags.SwallowExceptions)

{

// pop the try block and fill in the body

PopStmt(&stmtTryBlock);

pTryBlock->sxBlock.pnodeStmt = pnode;

PopStmt(&stmtTry);

if(pnode != nullptr)

{

pTry->ichLim = pnode->ichLim;

}

pTry->sxTry.pnodeBody = pTryBlock;

// create a catch block with an empty body

StmtNest stmtCatch;

ParseNodePtr pCatch;

pCatch = CreateNodeWithScanner<knopCatch>();

PushStmt<buildAST>(&stmtCatch, pCatch, knopCatch, nullptr, nullptr);

pCatch->sxCatch.pnodeBody = nullptr;

if(pnode != nullptr)

{

pCatch->ichLim = pnode->ichLim;

}

pCatch->sxCatch.grfnop = 0;

pCatch->sxCatch.pnodeNext = nullptr;

// create a fake name for the catch var.

WCHAR \*uniqueNameStr = L"\_\_ehobj";

IdentPtr uniqueName = m\_phtbl->PidHashNameLen(uniqueNameStr, static\_cast<long>(wcslen(uniqueNameStr)));

pCatch->sxCatch.pnodeParam = CreateNameNode(uniqueName);

// Add this catch to the current list. We don't bother adjusting the catch and function expression

// lists here because the catch is just an empty statement.

if (m\_ppnodeExprScope)

{

Assert(\*m\_ppnodeExprScope == nullptr);

\*m\_ppnodeExprScope = pCatch;

m\_ppnodeExprScope = &pCatch->sxCatch.pnodeNext;

}

else

{

Assert(m\_ppnodeScope);

Assert(\*m\_ppnodeScope == nullptr);

\*m\_ppnodeScope = pCatch;

m\_ppnodeScope = &pCatch->sxCatch.pnodeNext;

}

pCatch->sxCatch.pnodeScopes = nullptr;

PopStmt(&stmtCatch);

// fill in and pop the try-catch

pParentTryCatch->sxTryCatch.pnodeTry = pTry;

pParentTryCatch->sxTryCatch.pnodeCatch = pCatch;

PopStmt(&stmtTryCatch);

PopStmt(&stmtTryCatchBlock);

// replace the node that's being returned

pParentTryCatchBlock->sxBlock.pnodeStmt = pParentTryCatch;

pnode = pParentTryCatchBlock;

}

#endif // EXCEPTION\_RECOVERY

}

return pnode;

}

BOOL

Parser::TokIsForInOrForOf()

{

return m\_token.tk == tkIN ||

(m\_scriptContext->GetConfig()->IsES6IteratorsEnabled() &&

m\_token.tk == tkID &&

m\_token.GetIdentifier(m\_phtbl) == wellKnownPropertyPids.of);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Parse a sequence of statements.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

template<bool buildAST>

void Parser::ParseStmtList(ParseNodePtr \*ppnodeList, ParseNodePtr \*\*pppnodeLast, StrictModeEnvironment smEnvironment, const bool isSourceElementList, bool\* strictModeOn)

{

BOOL doneDirectives = !isSourceElementList; // directives may only exist in a SourceElementList, not a StatementList

BOOL seenDirectiveContainingOctal = false; // Have we seen an octal directive before a use strict directive?

BOOL old\_UseStrictMode = m\_fUseStrictMode;

ParseNodePtr pnodeStmt;

ParseNodePtr \*lastNodeRef = nullptr;

if (buildAST)

{

AssertMem(ppnodeList);

AssertMemN(pppnodeLast);

\*ppnodeList = nullptr;

}

if(CONFIG\_FLAG(ForceStrictMode))

{

m\_fUseStrictMode = TRUE;

}

for (;;)

{

switch (m\_token.tk)

{

case tkCASE:

case tkDEFAULT:

case tkRCurly:

case tkEOF:

if (buildAST && nullptr != pppnodeLast)

{

\*pppnodeLast = lastNodeRef;

}

if (!buildAST)

{

m\_fUseStrictMode = old\_UseStrictMode;

}

return;

}

if (doneDirectives == FALSE)

{

bool isOctalInString = false;

bool isUseStrictDirective = false;

bool isUseAsmDirective = false;

if (smEnvironment != SM\_NotUsed && CheckForDirective(&isUseStrictDirective, &isUseAsmDirective, &isOctalInString))

{

if (isUseStrictDirective)

{

if (seenDirectiveContainingOctal)

{

// Directives seen before a "use strict" cannot contain an octal.

Error(ERRES5NoOctal);

}

if (!buildAST)

{

// Turning on strict mode in deferred code.

m\_fUseStrictMode = TRUE;

if (!m\_inDeferredNestedFunc)

{

// Top-level deferred function, so there's a parse node

Assert(m\_currentNodeFunc != nullptr);

m\_currentNodeFunc->sxFnc.SetStrictMode();

}

else if (strictModeOn)

{

// This turns on strict mode in a deferred function, we need to go back

// and re-check duplicated formals.

\*strictModeOn = true;

}

}

else

{

if (smEnvironment == SM\_OnGlobalCode)

{

// Turning on strict mode at the top level

m\_fUseStrictMode = TRUE;

}

else

{

// i.e. smEnvironment == SM\_OnFunctionCode

Assert(m\_currentNodeFunc != nullptr);

m\_currentNodeFunc->sxFnc.SetStrictMode();

}

}

}

else if (isUseAsmDirective)

{

if (smEnvironment != SM\_OnGlobalCode) //Top level use asm doesn't mean anything.

{

// i.e. smEnvironment == SM\_OnFunctionCode

Assert(m\_currentNodeFunc != nullptr);

m\_currentNodeFunc->sxFnc.SetAsmjsMode();

m\_InAsmMode = true;

CHAKRATEL\_LANGSTATS\_INC\_LANGFEATURECOUNT(AsmJSFunctionCount, m\_scriptContext);

}

}

else if (isOctalInString)

{

seenDirectiveContainingOctal = TRUE;

}

}

else

{

// The first time we see anything other than a directive we can have no more directives.

doneDirectives = TRUE;

}

}

if (nullptr != (pnodeStmt = ParseStatement<buildAST>(isSourceElementList)))

{

Assert(buildAST || BindDeferredPidRefs());

if (buildAST)

{

AddToNodeList(ppnodeList, &lastNodeRef, pnodeStmt);

}

}

}

}

template <class Fn>

void Parser::VisitFunctionsInScope(ParseNodePtr pnodeScopeList, Fn fn)

{

ParseNodePtr pnodeScope;

for (pnodeScope = pnodeScopeList; pnodeScope;)

{

switch (pnodeScope->nop)

{

case knopBlock:

VisitFunctionsInScope(pnodeScope->sxBlock.pnodeScopes, fn);

pnodeScope = pnodeScope->sxBlock.pnodeNext;

break;

case knopFncDecl:

fn(pnodeScope);

pnodeScope = pnodeScope->sxFnc.pnodeNext;

break;

case knopCatch:

VisitFunctionsInScope(pnodeScope->sxCatch.pnodeScopes, fn);

pnodeScope = pnodeScope->sxCatch.pnodeNext;

break;

case knopWith:

VisitFunctionsInScope(pnodeScope->sxWith.pnodeScopes, fn);

pnodeScope = pnodeScope->sxWith.pnodeNext;

break;

default:

AssertMsg(false, "Unexpected node with scope list");

return;

}

}

}

// Scripts above this size (minus string literals and comments) will have parsing of

// function bodies deferred.

ULONG Parser::GetDeferralThreshold(bool isProfileLoaded)

{

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (CONFIG\_FLAG(ForceDeferParse) ||

PHASE\_FORCE1(Js::DeferParsePhase) ||

Js::Configuration::Global.flags.IsEnabled(Js::ForceUndoDeferFlag))

{

return 0;

}

else if (Js::Configuration::Global.flags.IsEnabled(Js::DeferParseFlag))

{

return Js::Configuration::Global.flags.DeferParse;

}

else

#endif

{

if (isProfileLoaded)

{

return DEFAULT\_CONFIG\_ProfileBasedDeferParseThreshold;

}

return DEFAULT\_CONFIG\_DeferParseThreshold;

}

}

void Parser::FinishDeferredFunction(ParseNodePtr pnodeScopeList)

{

VisitFunctionsInScope(pnodeScopeList,

[this](ParseNodePtr pnodeFnc)

{

Assert(pnodeFnc->nop == knopFncDecl);

if (pnodeFnc->sxFnc.pnodeBody == nullptr)

{

// Go back and generate an AST for this function.

JS\_ETW(EventWriteJSCRIPT\_PARSE\_FUNC(this->GetScriptContext(), pnodeFnc->sxFnc.functionId, /\*Undefer\*/TRUE));

ParseNodePtr pnodeFncSave = this->m\_currentNodeFunc;

this->m\_currentNodeFunc = pnodeFnc;

ParseNodePtr pnodeFncExprBlock = nullptr;

if (pnodeFnc->sxFnc.pnodeName &&

!pnodeFnc->sxFnc.IsDeclaration())

{

// Set up the named function expression symbol so references inside the function can be bound.

ParseNodePtr pnodeName = pnodeFnc->sxFnc.pnodeName;

Assert(pnodeName->nop == knopVarDecl);

Assert(pnodeName->sxVar.pnodeNext == nullptr);

pnodeFncExprBlock = this->StartParseBlock<true>(PnodeBlockType::Function, ScopeType\_FuncExpr);

PidRefStack \*ref = this->PushPidRef(pnodeName->sxVar.pid);

pnodeName->sxVar.symRef = ref->GetSymRef();

ref->SetSym(pnodeName->sxVar.sym);

Scope \*fncExprScope = pnodeFncExprBlock->sxBlock.scope;

fncExprScope->AddNewSymbol(pnodeName->sxVar.sym);

pnodeFnc->sxFnc.scope = fncExprScope;

}

ParseNodePtr pnodeBlock = this->StartParseBlock<true>(PnodeBlockType::Parameter, ScopeType\_Parameter);

pnodeFnc->sxFnc.pnodeScopes = pnodeBlock;

m\_ppnodeScope = &pnodeBlock->sxBlock.pnodeScopes;

pnodeBlock->sxBlock.pnodeStmt = pnodeFnc;

// Add the args to the scope, since we won't re-parse those.

Scope \*scope = pnodeBlock->sxBlock.scope;

auto addArgsToScope = [&](ParseNodePtr pnodeArg) {

if (pnodeArg->IsVarLetOrConst())

{

PidRefStack \*ref = this->PushPidRef(pnodeArg->sxVar.pid);

pnodeArg->sxVar.symRef = ref->GetSymRef();

if (ref->GetSym() != nullptr)

{

// Duplicate parameter in a configuration that allows them.

// The symbol is already in the scope, just point it to the right declaration.

Assert(ref->GetSym() == pnodeArg->sxVar.sym);

ref->GetSym()->SetDecl(pnodeArg);

}

else

{

ref->SetSym(pnodeArg->sxVar.sym);

scope->AddNewSymbol(pnodeArg->sxVar.sym);

}

}

};

MapFormals(pnodeFnc, addArgsToScope);

MapFormalsFromPattern(pnodeFnc, addArgsToScope);

ParseNodePtr pnodeInnerBlock = this->StartParseBlock<true>(PnodeBlockType::Function, ScopeType\_FunctionBody);

pnodeFnc->sxFnc.pnodeBodyScope = pnodeInnerBlock;

// Set the parameter block's child to the function body block.

\*m\_ppnodeScope = pnodeInnerBlock;

ParseNodePtr \*ppnodeScopeSave = nullptr;

ParseNodePtr \*ppnodeExprScopeSave = nullptr;

ppnodeScopeSave = m\_ppnodeScope;

// This synthetic block scope will contain all the nested scopes.

m\_ppnodeScope = &pnodeInnerBlock->sxBlock.pnodeScopes;

pnodeInnerBlock->sxBlock.pnodeStmt = pnodeFnc;

// Keep nested function declarations and expressions in the same list at function scope.

// (Indicate this by nulling out the current function expressions list.)

ppnodeExprScopeSave = m\_ppnodeExprScope;

m\_ppnodeExprScope = nullptr;

// Shouldn't be any temps in the arg list.

Assert(\*m\_ppnodeVar == nullptr);

// Start the var list.

pnodeFnc->sxFnc.pnodeVars = nullptr;

m\_ppnodeVar = &pnodeFnc->sxFnc.pnodeVars;

this->FinishFncNode(pnodeFnc);

m\_ppnodeExprScope = ppnodeExprScopeSave;

AssertMem(m\_ppnodeScope);

Assert(nullptr == \*m\_ppnodeScope);

m\_ppnodeScope = ppnodeScopeSave;

this->FinishParseBlock(pnodeInnerBlock);

this->AddArgumentsNodeToVars(pnodeFnc);

this->FinishParseBlock(pnodeBlock);

if (pnodeFncExprBlock)

{

this->FinishParseBlock(pnodeFncExprBlock);

}

this->m\_currentNodeFunc = pnodeFncSave;

}

});

}

void Parser::InitPids()

{

AssertMemN(m\_phtbl);

wellKnownPropertyPids.arguments = m\_phtbl->PidHashNameLen(g\_ssym\_arguments.sz, g\_ssym\_arguments.cch);

wellKnownPropertyPids.async = m\_phtbl->PidHashNameLen(g\_ssym\_async.sz, g\_ssym\_async.cch);

wellKnownPropertyPids.eval = m\_phtbl->PidHashNameLen(g\_ssym\_eval.sz, g\_ssym\_eval.cch);

wellKnownPropertyPids.getter = m\_phtbl->PidHashNameLen(g\_ssym\_get.sz, g\_ssym\_get.cch);

wellKnownPropertyPids.setter = m\_phtbl->PidHashNameLen(g\_ssym\_set.sz, g\_ssym\_set.cch);

wellKnownPropertyPids.let = m\_phtbl->PidHashNameLen(g\_ssym\_let.sz, g\_ssym\_let.cch);

wellKnownPropertyPids.constructor = m\_phtbl->PidHashNameLen(g\_ssym\_constructor.sz, g\_ssym\_constructor.cch);

wellKnownPropertyPids.prototype = m\_phtbl->PidHashNameLen(g\_ssym\_prototype.sz, g\_ssym\_prototype.cch);

wellKnownPropertyPids.\_\_proto\_\_ = m\_phtbl->PidHashNameLen(L"\_\_proto\_\_", sizeof("\_\_proto\_\_") - 1);

wellKnownPropertyPids.of = m\_phtbl->PidHashNameLen(L"of", sizeof("of") - 1);

wellKnownPropertyPids.target = m\_phtbl->PidHashNameLen(L"target", sizeof("target") - 1);

}

void Parser::RestoreScopeInfo(Js::FunctionBody\* functionBody)

{

if (!functionBody)

{

return;

}

Js::ScopeInfo\* scopeInfo = functionBody->GetScopeInfo();

if (!scopeInfo)

{

return;

}

if (this->IsBackgroundParser())

{

PROBE\_STACK\_NO\_DISPOSE(m\_scriptContext, Js::Constants::MinStackByteCodeVisitor);

}

else

{

PROBE\_STACK(m\_scriptContext, Js::Constants::MinStackByteCodeVisitor);

}

RestoreScopeInfo(scopeInfo->GetParent()); // Recursively restore outer func scope info

Js::ScopeInfo\* funcExprScopeInfo = scopeInfo->GetFuncExprScopeInfo();

if (funcExprScopeInfo)

{

funcExprScopeInfo->SetScopeId(m\_nextBlockId);

ParseNodePtr pnodeFncExprScope = StartParseBlockWithCapacity<true>(PnodeBlockType::Function, ScopeType\_FuncExpr, funcExprScopeInfo->GetSymbolCount());

Scope \*scope = pnodeFncExprScope->sxBlock.scope;

funcExprScopeInfo->GetScopeInfo(this, nullptr, nullptr, scope);

}

Js::ScopeInfo\* paramScopeInfo = scopeInfo->GetParamScopeInfo();

if (paramScopeInfo)

{

paramScopeInfo->SetScopeId(m\_nextBlockId);

ParseNodePtr pnodeFncExprScope = StartParseBlockWithCapacity<true>(PnodeBlockType::Parameter, ScopeType\_Parameter, paramScopeInfo->GetSymbolCount());

Scope \*scope = pnodeFncExprScope->sxBlock.scope;

paramScopeInfo->GetScopeInfo(this, nullptr, nullptr, scope);

}

scopeInfo->SetScopeId(m\_nextBlockId);

ParseNodePtr pnodeFncScope = nullptr;

if (scopeInfo->IsGlobalEval())

{

pnodeFncScope = StartParseBlockWithCapacity<true>(PnodeBlockType::Regular, ScopeType\_GlobalEvalBlock, scopeInfo->GetSymbolCount());

}

else

{

pnodeFncScope = StartParseBlockWithCapacity<true>(PnodeBlockType::Function, ScopeType\_FunctionBody, scopeInfo->GetSymbolCount());

}

Scope \*scope = pnodeFncScope->sxBlock.scope;

scopeInfo->GetScopeInfo(this, nullptr, nullptr, scope);

}

void Parser::FinishScopeInfo(Js::FunctionBody \*functionBody)

{

if (!functionBody)

{

return;

}

Js::ScopeInfo\* scopeInfo = functionBody->GetScopeInfo();

if (!scopeInfo)

{

return;

}

if (this->IsBackgroundParser())

{

PROBE\_STACK\_NO\_DISPOSE(m\_scriptContext, Js::Constants::MinStackByteCodeVisitor);

}

else

{

PROBE\_STACK(m\_scriptContext, Js::Constants::MinStackByteCodeVisitor);

}

int scopeId = scopeInfo->GetScopeId();

scopeInfo->GetScope()->ForEachSymbol([this, scopeId](Symbol \*sym)

{

this->BindPidRefsInScope(sym->GetPid(), sym, scopeId);

});

PopScope(scopeInfo->GetScope());

PopStmt(&m\_currentBlockInfo->pstmt);

PopBlockInfo();

Js::ScopeInfo \*paramScopeInfo = scopeInfo->GetParamScopeInfo();

if (paramScopeInfo)

{

scopeId = paramScopeInfo->GetScopeId();

paramScopeInfo->GetScope()->ForEachSymbol([this, scopeId](Symbol \*sym)

{

this->BindPidRefsInScope(sym->GetPid(), sym, scopeId);

});

PopScope(paramScopeInfo->GetScope());

PopStmt(&m\_currentBlockInfo->pstmt);

PopBlockInfo();

}

Js::ScopeInfo \*funcExprScopeInfo = scopeInfo->GetFuncExprScopeInfo();

if (funcExprScopeInfo)

{

scopeId = funcExprScopeInfo->GetScopeId();

funcExprScopeInfo->GetScope()->ForEachSymbol([this, scopeId](Symbol \*sym)

{

this->BindPidRefsInScope(sym->GetPid(), sym, scopeId);

});

PopScope(funcExprScopeInfo->GetScope());

PopStmt(&m\_currentBlockInfo->pstmt);

PopBlockInfo();

}

FinishScopeInfo(scopeInfo->GetParent());

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Parse the code.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

ParseNodePtr Parser::Parse(LPCUTF8 pszSrc, size\_t offset, size\_t length, charcount\_t charOffset, ULONG grfscr, ULONG lineNumber, Js::LocalFunctionId \* nextFunctionId, CompileScriptException \*pse)

{

ParseNodePtr pnodeProg;

ParseNodePtr \*lastNodeRef = nullptr;

m\_nextBlockId = 0;

// Scanner should run in Running mode and not syntax coloring mode

grfscr &= ~fscrSyntaxColor;

if (this->m\_scriptContext->IsInDebugMode() || PHASE\_OFF1(Js::Phase::DeferParsePhase)

#ifdef ENABLE\_PREJIT

|| Js::Configuration::Global.flags.Prejit

#endif

|| ((grfscr & fscrNoDeferParse) != 0)

)

{

// Don't do deferred parsing if debugger is attached or feature is disabled

// by command-line switch.

grfscr &= ~fscrDeferFncParse;

}

bool isDeferred = (grfscr & fscrDeferredFnc) != 0;

m\_grfscr = grfscr;

m\_length = length;

m\_originalLength = length;

m\_nextFunctionId = nextFunctionId;

if(m\_parseType != ParseType\_Deferred)

{

JS\_ETW(EventWriteJSCRIPT\_PARSE\_METHOD\_START(m\_sourceContextInfo->dwHostSourceContext, GetScriptContext(), \*m\_nextFunctionId, 0, m\_parseType, Js::Constants::GlobalFunction));

OUTPUT\_TRACE(Js::DeferParsePhase, L"Parsing function (%s) : %s (%d)\n", GetParseType(), Js::Constants::GlobalFunction, \*m\_nextFunctionId);

}

// Give the scanner the source and get the first token

m\_pscan->SetText(pszSrc, offset, length, charOffset, grfscr, lineNumber);

m\_pscan->Scan();

// Make the main 'knopProg' node

long initSize = 0;

m\_pCurrentAstSize = &initSize;

pnodeProg = CreateNodeWithScanner<knopProg>();

pnodeProg->grfpn = PNodeFlags::fpnNone;

pnodeProg->sxFnc.pid = nullptr;

pnodeProg->sxFnc.pnodeName = nullptr;

pnodeProg->sxFnc.pnodeRest = nullptr;

pnodeProg->sxFnc.ClearFlags();

pnodeProg->sxFnc.SetNested(FALSE);

pnodeProg->sxFnc.astSize = 0;

pnodeProg->sxFnc.cbMin = m\_pscan->IecpMinTok();

pnodeProg->sxFnc.lineNumber = lineNumber;

pnodeProg->sxFnc.columnNumber = 0;

if (!isDeferred || (isDeferred && grfscr & fscrGlobalCode))

{

// In the deferred case, if the global function is deferred parse (which is in no-refresh case),

// we will re-use the same function body, so start with the correct functionId.

pnodeProg->sxFnc.functionId = (\*m\_nextFunctionId)++;

}

else

{

pnodeProg->sxFnc.functionId = Js::Constants::NoFunctionId;

}

m\_pCurrentAstSize = & (pnodeProg->sxFnc.astSize);

pnodeProg->sxFnc.hint = nullptr;

pnodeProg->sxFnc.hintLength = 0;

pnodeProg->sxFnc.hintOffset = 0;

pnodeProg->sxFnc.isNameIdentifierRef = true;

// initialize parsing variables

pnodeProg->sxFnc.pnodeNext = nullptr;

m\_currentNodeFunc = nullptr;

m\_currentNodeDeferredFunc = nullptr;

m\_currentNodeProg = pnodeProg;

m\_cactIdentToNodeLookup = 1;

pnodeProg->sxFnc.nestedCount = 0;

m\_pnestedCount = &pnodeProg->sxFnc.nestedCount;

m\_inDeferredNestedFunc = false;

pnodeProg->sxFnc.pnodeArgs = nullptr;

pnodeProg->sxFnc.pnodeVars = nullptr;

pnodeProg->sxFnc.pnodeRest = nullptr;

m\_ppnodeVar = &pnodeProg->sxFnc.pnodeVars;

SetCurrentStatement(nullptr);

AssertMsg(m\_pstmtCur == nullptr, "Statement stack should be empty when we start parse global code");

// Create block for const's and let's

ParseNodePtr pnodeGlobalBlock = StartParseBlock<true>(PnodeBlockType::Global, ScopeType\_Global);

pnodeProg->sxProg.scope = pnodeGlobalBlock->sxBlock.scope;

ParseNodePtr pnodeGlobalEvalBlock = nullptr;

// Don't track function expressions separately from declarations at global scope.

m\_ppnodeExprScope = nullptr;

// This synthetic block scope will contain all the nested scopes.

pnodeProg->sxFnc.pnodeBodyScope = nullptr;

pnodeProg->sxFnc.pnodeScopes = pnodeGlobalBlock;

m\_ppnodeScope = &pnodeGlobalBlock->sxBlock.pnodeScopes;

if ((this->m\_grfscr & fscrEvalCode) &&

m\_scriptContext->GetConfig()->IsBlockScopeEnabled() &&

!(this->m\_functionBody && this->m\_functionBody->GetScopeInfo()))

{

pnodeGlobalEvalBlock = StartParseBlock<true>(PnodeBlockType::Regular, ScopeType\_GlobalEvalBlock);

pnodeProg->sxFnc.pnodeScopes = pnodeGlobalEvalBlock;

m\_ppnodeScope = &pnodeGlobalEvalBlock->sxBlock.pnodeScopes;

}

Js::ScopeInfo \*scopeInfo = nullptr;

if (m\_parseType == ParseType\_Deferred && m\_functionBody)

{

// this->m\_functionBody can be cleared during parsing, but we need access to the scope info later.

scopeInfo = m\_functionBody->GetScopeInfo();

if (scopeInfo)

{

this->RestoreScopeInfo(scopeInfo->GetParent());

}

}

// Process a sequence of statements/declarations

ParseStmtList<true>(

&pnodeProg->sxFnc.pnodeBody,

&lastNodeRef,

SM\_OnGlobalCode,

!(m\_grfscr & fscrDeferredFncExpression) /\* isSourceElementList \*/);

if (m\_parseType == ParseType\_Deferred)

{

if (scopeInfo)

{

this->FinishScopeInfo(scopeInfo->GetParent());

}

}

pnodeProg->sxProg.m\_UsesArgumentsAtGlobal = m\_UsesArgumentsAtGlobal;

if (IsStrictMode())

{

pnodeProg->sxFnc.SetStrictMode();

}

#if DEBUG

if(m\_grfscr & fscrEnforceJSON && !IsJSONValid(pnodeProg->sxFnc.pnodeBody))

{

Error(ERRsyntax);

}

#endif

if (tkEOF != m\_token.tk)

Error(ERRsyntax);

// Append an EndCode node.

AddToNodeList(&pnodeProg->sxFnc.pnodeBody, &lastNodeRef,

CreateNodeWithScanner<knopEndCode>());

AssertMem(lastNodeRef);

AssertNodeMem(\*lastNodeRef);

Assert((\*lastNodeRef)->nop == knopEndCode);

(\*lastNodeRef)->ichMin = 0;

(\*lastNodeRef)->ichLim = 0;

// Get the extent of the code.

pnodeProg->ichLim = m\_pscan->IchLimTok();

pnodeProg->sxFnc.cbLim = m\_pscan->IecpLimTok();

// save the temps and terminate the local list

// NOTE: Eze makes no use of this.

//pnodeProg->sxFnc.pnodeTmps = \*m\_ppnodeVar;

\*m\_ppnodeVar = nullptr;

Assert(nullptr == \*m\_ppnodeScope);

Assert(nullptr == pnodeProg->sxFnc.pnodeNext);

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Js::Configuration::Global.flags.IsEnabled(Js::ForceUndoDeferFlag))

{

m\_stoppedDeferredParse = true;

}

#endif

if (m\_stoppedDeferredParse)

{

if (this->m\_hasParallelJob)

{

#if ENABLE\_BACKGROUND\_PARSING

BackgroundParser \*bgp = static\_cast<BackgroundParser\*>(m\_scriptContext->GetBackgroundParser());

Assert(bgp);

this->WaitForBackgroundJobs(bgp, pse);

#endif

}

// Finally, see if there are any function bodies we now want to generate because we

// decided to stop deferring.

FinishDeferredFunction(pnodeProg->sxFnc.pnodeScopes);

}

if (pnodeGlobalEvalBlock)

{

FinishParseBlock(pnodeGlobalEvalBlock);

}

// Append block as body of pnodeProg

FinishParseBlock(pnodeGlobalBlock);

m\_scriptContext->AddSourceSize(m\_length);

if (m\_asgToConst)

{

Error(ERRAssignmentToConst, m\_asgToConst.GetIchMin(), m\_asgToConst.GetIchLim());

}

if(!m\_parseType != ParseType\_Deferred)

{

JS\_ETW(EventWriteJSCRIPT\_PARSE\_METHOD\_STOP(m\_sourceContextInfo->dwHostSourceContext, GetScriptContext(), pnodeProg->sxFnc.functionId, \*m\_pCurrentAstSize, false, Js::Constants::GlobalFunction));

}

return pnodeProg;

}

bool Parser::CheckForDirective(bool\* pIsUseStrict, bool \*pIsUseAsm, bool\* pIsOctalInString)

{

// A directive is a string constant followed by a statement terminating token

if (m\_token.tk != tkStrCon)

return false;

// Careful, need to check for octal before calling m\_pscan->Scan()

// because Scan() clears the "had octal" flag on the scanner and

// m\_pscan->Restore() does not restore this flag.

if (pIsOctalInString != nullptr)

{

\*pIsOctalInString = m\_pscan->IsOctOrLeadingZeroOnLastTKNumber();

}

Ident\* pidDirective = m\_token.GetStr();

RestorePoint start;

m\_pscan->Capture(&start);

m\_pscan->Scan();

bool isDirective = true;

switch (m\_token.tk)

{

case tkSColon:

case tkEOF:

case tkLCurly:

case tkRCurly:

break;

default:

if (!m\_pscan->FHadNewLine())

{

isDirective = false;

}

break;

}

if (isDirective)

{

if (pIsUseStrict != nullptr)

{

\*pIsUseStrict = CheckStrictModeStrPid(pidDirective);

}

if (pIsUseAsm != nullptr)

{

\*pIsUseAsm = CheckAsmjsModeStrPid(pidDirective);

}

}

m\_pscan->SeekTo(start);

return isDirective;

}

bool Parser::CheckStrictModeStrPid(IdentPtr pid)

{

// If we're already in strict mode, no need to check if the string would put us in strict mode. So, this function would only

// return true if it detects a transition from non-strict to strict, which is what matters for callers.

// This is a minor optimization to avoid redundant string comparisons of nested "use strict" directives.

if (IsStrictMode())

{

return false;

}

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Js::Configuration::Global.flags.NoStrictMode)

return false;

#endif

return pid != nullptr &&

pid->Cch() == 10 &&

!m\_pscan->IsEscapeOnLastTkStrCon() &&

wcsncmp(pid->Psz(), L"use strict", 10) == 0;

}

bool Parser::CheckAsmjsModeStrPid(IdentPtr pid)

{

#ifdef ASMJS\_PLAT

if (!CONFIG\_FLAG\_RELEASE(Asmjs))

{

return false;

}

bool isAsmCandidate = (pid != nullptr &&

AutoSystemInfo::Data.SSE2Available() &&

pid->Cch() == 7 &&

!m\_pscan->IsEscapeOnLastTkStrCon() &&

wcsncmp(pid->Psz(), L"use asm", 10) == 0);

if (isAsmCandidate && m\_scriptContext->IsInDebugMode())

{

// We would like to report this to debugger - they may choose to disable debugging.

// TODO : localization of the string?

m\_scriptContext->RaiseMessageToDebugger(DEIT\_ASMJS\_IN\_DEBUGGING, L"AsmJs initialization error - AsmJs disabled due to script debugger", !m\_sourceContextInfo->IsDynamic() ? m\_sourceContextInfo->url : nullptr);

return false;

}

return isAsmCandidate && !(m\_grfscr & fscrNoAsmJs);

#else

return false;

#endif

}

HRESULT Parser::ParseUtf8Source(\_\_out ParseNodePtr\* parseTree, LPCUTF8 pSrc, size\_t length, ULONG grfsrc, CompileScriptException \*pse,

Js::LocalFunctionId \* nextFunctionId, SourceContextInfo \* sourceContextInfo)

{

m\_functionBody = nullptr;

m\_parseType = ParseType\_Upfront;

return ParseSourceInternal( parseTree, pSrc, 0, length, 0, true, grfsrc, pse, nextFunctionId, 0, sourceContextInfo);

}

HRESULT Parser::ParseCesu8Source(\_\_out ParseNodePtr\* parseTree, LPCUTF8 pSrc, size\_t length, ULONG grfsrc, CompileScriptException \*pse,

Js::LocalFunctionId \* nextFunctionId, SourceContextInfo \* sourceContextInfo)

{

m\_functionBody = nullptr;

m\_parseType = ParseType\_Upfront;

return ParseSourceInternal( parseTree, pSrc, 0, length, 0, false, grfsrc, pse, nextFunctionId, 0, sourceContextInfo);

}

void Parser::PrepareScanner(bool fromExternal)

{

// NOTE: HashTbl and Scanner are currently allocated from the CRT heap. If we want to allocate them from the

// parser arena, then we also need to change the way the HashTbl allocates PID's from its underlying

// allocator (which also currently uses the CRT heap). This is not trivial, because we still need to support

// heap allocation for the colorizer interface.

// create the hash table and init PID members

if (nullptr == (m\_phtbl = HashTbl::Create(HASH\_TABLE\_SIZE, &m\_err)))

Error(ERRnoMemory);

InitPids();

// create the scanner

if (nullptr == (m\_pscan = Scanner\_t::Create(this, m\_phtbl, &m\_token, &m\_err, m\_scriptContext)))

Error(ERRnoMemory);

if (fromExternal)

m\_pscan->FromExternalSource();

}

#if ENABLE\_BACKGROUND\_PARSING

void Parser::PrepareForBackgroundParse()

{

m\_pscan->PrepareForBackgroundParse(m\_scriptContext);

}

void Parser::AddBackgroundParseItem(BackgroundParseItem \*const item)

{

if (currBackgroundParseItem == nullptr)

{

backgroundParseItems = item;

}

else

{

currBackgroundParseItem->SetNext(item);

}

currBackgroundParseItem = item;

}

#endif

void Parser::AddFastScannedRegExpNode(ParseNodePtr const pnode)

{

Assert(!IsBackgroundParser());

Assert(m\_doingFastScan);

if (fastScannedRegExpNodes == nullptr)

{

fastScannedRegExpNodes = Anew(&m\_nodeAllocator, NodeDList, &m\_nodeAllocator);

}

fastScannedRegExpNodes->Append(pnode);

}

#if ENABLE\_BACKGROUND\_PARSING

void Parser::AddBackgroundRegExpNode(ParseNodePtr const pnode)

{

Assert(IsBackgroundParser());

Assert(currBackgroundParseItem != nullptr);

currBackgroundParseItem->AddRegExpNode(pnode, &m\_nodeAllocator);

}

#endif

HRESULT Parser::ParseFunctionInBackground(ParseNodePtr pnodeFnc, ParseContext \*parseContext, bool topLevelDeferred, CompileScriptException \*pse)

{

m\_functionBody = nullptr;

m\_parseType = ParseType\_Upfront;

HRESULT hr = S\_OK;

SmartFPUControl smartFpuControl;

uint nextFunctionId = pnodeFnc->sxFnc.functionId + 1;

this->RestoreContext(parseContext);

DebugOnly( m\_err.fInited = TRUE; )

m\_nextFunctionId = &nextFunctionId;

m\_deferringAST = topLevelDeferred;

m\_inDeferredNestedFunc = false;

m\_scopeCountNoAst = 0;

SetCurrentStatement(nullptr);

pnodeFnc->sxFnc.pnodeVars = nullptr;

pnodeFnc->sxFnc.pnodeArgs = nullptr;

pnodeFnc->sxFnc.pnodeBody = nullptr;

pnodeFnc->sxFnc.nestedCount = 0;

m\_currentNodeFunc = pnodeFnc;

m\_currentNodeDeferredFunc = nullptr;

m\_ppnodeScope = nullptr;

m\_ppnodeExprScope = nullptr;

m\_pnestedCount = &pnodeFnc->sxFnc.nestedCount;

m\_pCurrentAstSize = &pnodeFnc->sxFnc.astSize;

ParseNodePtr pnodeBlock = StartParseBlock<true>(PnodeBlockType::Function, ScopeType\_FunctionBody);

pnodeFnc->sxFnc.pnodeScopes = pnodeBlock;

m\_ppnodeScope = &pnodeBlock->sxBlock.pnodeScopes;

uint uDeferSave = m\_grfscr & fscrDeferFncParse;

try

{

m\_pscan->Scan();

m\_ppnodeVar = &pnodeFnc->sxFnc.pnodeArgs;

this->ParseFncFormals<true>(pnodeFnc, fFncNoFlgs);

if (m\_token.tk == tkRParen)

{

m\_pscan->Scan();

}

ChkCurTok(tkLCurly, ERRnoLcurly);

m\_ppnodeVar = &pnodeFnc->sxFnc.pnodeVars;

// Put the scanner into "no hashing" mode.

BYTE deferFlags = m\_pscan->SetDeferredParse(topLevelDeferred);

// Process a sequence of statements/declarations

if (topLevelDeferred)

{

ParseStmtList<false>(nullptr, nullptr, SM\_DeferedParse, true);

}

else

{

ParseNodePtr \*lastNodeRef = nullptr;

ParseStmtList<true>(&pnodeFnc->sxFnc.pnodeBody, &lastNodeRef, SM\_OnFunctionCode, true);

AddArgumentsNodeToVars(pnodeFnc);

// Append an EndCode node.

AddToNodeList(&pnodeFnc->sxFnc.pnodeBody, &lastNodeRef, CreateNodeWithScanner<knopEndCode>());

}

// Restore the scanner's default hashing mode.

m\_pscan->SetDeferredParseFlags(deferFlags);

#if DBG

pnodeFnc->sxFnc.deferredParseNextFunctionId = \*this->m\_nextFunctionId;

#endif

this->m\_deferringAST = FALSE;

// Append block as body of pnodeProg

FinishParseBlock(pnodeBlock);

if (m\_asgToConst)

{

Error(ERRAssignmentToConst, m\_asgToConst.GetIchMin(), m\_asgToConst.GetIchLim());

}

}

catch(ParseExceptionObject& e)

{

m\_err.m\_hr = e.GetError();

hr = pse->ProcessError( m\_pscan, m\_err.m\_hr, nullptr);

}

if (IsStrictMode())

{

pnodeFnc->sxFnc.SetStrictMode();

}

if (topLevelDeferred)

{

pnodeFnc->sxFnc.pnodeVars = nullptr;

}

m\_grfscr |= uDeferSave;

Assert(nullptr == \*m\_ppnodeScope);

return hr;

}

HRESULT Parser::ParseSourceWithOffset(\_\_out ParseNodePtr\* parseTree, LPCUTF8 pSrc, size\_t offset, size\_t cbLength, charcount\_t cchOffset,

bool isCesu8, ULONG grfscr, CompileScriptException \*pse, Js::LocalFunctionId \* nextFunctionId, ULONG lineNumber, SourceContextInfo \* sourceContextInfo,

Js::ParseableFunctionInfo\* functionInfo, bool isReparse)

{

m\_functionBody = functionInfo;

if (m\_functionBody)

{

m\_currDeferredStub = m\_functionBody->GetDeferredStubs();

m\_InAsmMode = grfscr & fscrNoAsmJs ? false : m\_functionBody->GetIsAsmjsMode();

}

m\_deferAsmJs = !m\_InAsmMode;

m\_parseType = isReparse ? ParseType\_Reparse : ParseType\_Deferred;

return ParseSourceInternal( parseTree, pSrc, offset, cbLength, cchOffset, !isCesu8, grfscr, pse, nextFunctionId, lineNumber, sourceContextInfo);

}

bool Parser::IsStrictMode() const

{

return (m\_fUseStrictMode ||

(m\_currentNodeFunc != nullptr && m\_currentNodeFunc->sxFnc.GetStrictMode()));

}

BOOL Parser::ExpectingExternalSource()

{

return m\_fExpectExternalSource;

}

Symbol \*PnFnc::GetFuncSymbol()

{

if (pnodeName &&

pnodeName->nop == knopVarDecl)

{

return pnodeName->sxVar.sym;

}

return nullptr;

}

void PnFnc::SetFuncSymbol(Symbol \*sym)

{

Assert(pnodeName &&

pnodeName->nop == knopVarDecl);

pnodeName->sxVar.sym = sym;

}

ParseNodePtr PnFnc::GetParamScope() const

{

if (this->pnodeScopes == nullptr)

{

return nullptr;

}

Assert(this->pnodeScopes->nop == knopBlock &&

this->pnodeScopes->sxBlock.pnodeNext == nullptr);

return this->pnodeScopes->sxBlock.pnodeScopes;

}

ParseNodePtr \* PnFnc::GetParamScopeRef() const

{

if (this->pnodeScopes == nullptr)

{

return nullptr;

}

Assert(this->pnodeScopes->nop == knopBlock &&

this->pnodeScopes->sxBlock.pnodeNext == nullptr);

return &this->pnodeScopes->sxBlock.pnodeScopes;

}

ParseNodePtr PnFnc::GetBodyScope() const

{

if (this->pnodeBodyScope == nullptr)

{

return nullptr;

}

Assert(this->pnodeBodyScope->nop == knopBlock &&

this->pnodeBodyScope->sxBlock.pnodeNext == nullptr);

return this->pnodeBodyScope->sxBlock.pnodeScopes;

}

ParseNodePtr \* PnFnc::GetBodyScopeRef() const

{

if (this->pnodeBodyScope == nullptr)

{

return nullptr;

}

Assert(this->pnodeBodyScope->nop == knopBlock &&

this->pnodeBodyScope->sxBlock.pnodeNext == nullptr);

return &this->pnodeBodyScope->sxBlock.pnodeScopes;

}

// Create node versions with explicit token limits

ParseNodePtr Parser::CreateNode(OpCode nop, charcount\_t ichMin, charcount\_t ichLim)

{

Assert(!this->m\_deferringAST);

Assert(nop >= 0 && nop < knopLim);

ParseNodePtr pnode;

\_\_analysis\_assume(nop < knopLim);

int cb = nop >= 0 && nop < knopLim ? g\_mpnopcbNode[nop] : kcbPnNone;

pnode = (ParseNodePtr)m\_nodeAllocator.Alloc(cb);

Assert(pnode);

Assert(m\_pCurrentAstSize != NULL);

\*m\_pCurrentAstSize += cb;

InitNode(nop,pnode);

pnode->ichMin = ichMin;

pnode->ichLim = ichLim;

return pnode;

}

ParseNodePtr Parser::CreateNameNode(IdentPtr pid,charcount\_t ichMin,charcount\_t ichLim) {

ParseNodePtr pnode = CreateNodeT<knopName>(ichMin,ichLim);

pnode->sxPid.pid = pid;

pnode->sxPid.sym=NULL;

pnode->sxPid.symRef=NULL;

return pnode;

}

ParseNodePtr Parser::CreateUniNode(OpCode nop, ParseNodePtr pnode1, charcount\_t ichMin,charcount\_t ichLim)

{

Assert(!this->m\_deferringAST);

DebugOnly(VerifyNodeSize(nop, kcbPnUni));

ParseNodePtr pnode = (ParseNodePtr)m\_nodeAllocator.Alloc(kcbPnUni);

Assert(m\_pCurrentAstSize != NULL);

\*m\_pCurrentAstSize += kcbPnUni;

InitNode(nop, pnode);

pnode->sxUni.pnode1 = pnode1;

pnode->ichMin = ichMin;

pnode->ichLim = ichLim;

return pnode;

}

ParseNodePtr Parser::CreateBinNode(OpCode nop, ParseNodePtr pnode1,

ParseNodePtr pnode2,charcount\_t ichMin,charcount\_t ichLim)

{

Assert(!this->m\_deferringAST);

ParseNodePtr pnode = StaticCreateBinNode(nop, pnode1, pnode2, &m\_nodeAllocator);

Assert(m\_pCurrentAstSize != NULL);

\*m\_pCurrentAstSize += kcbPnBin;

pnode->ichMin = ichMin;

pnode->ichLim = ichLim;

return pnode;

}

ParseNodePtr Parser::CreateTriNode(OpCode nop, ParseNodePtr pnode1,

ParseNodePtr pnode2, ParseNodePtr pnode3,

charcount\_t ichMin,charcount\_t ichLim)

{

Assert(!this->m\_deferringAST);

DebugOnly(VerifyNodeSize(nop, kcbPnTri));

ParseNodePtr pnode = (ParseNodePtr)m\_nodeAllocator.Alloc(kcbPnTri);

Assert(m\_pCurrentAstSize != NULL);

\*m\_pCurrentAstSize += kcbPnTri;

InitNode(nop, pnode);

pnode->sxTri.pnodeNext = NULL;

pnode->sxTri.pnode1 = pnode1;

pnode->sxTri.pnode2 = pnode2;

pnode->sxTri.pnode3 = pnode3;

pnode->ichMin = ichMin;

pnode->ichLim = ichLim;

return pnode;

}

bool PnBlock::HasBlockScopedContent() const

{

// A block has its own content if a let, const, or function is declared there.

if (this->pnodeLexVars != nullptr || this->blockType == Parameter)

{

return true;

}

// The enclosing scopes can contain functions and other things, so walk the list

// looking specifically for functions.

for (ParseNodePtr pnode = this->pnodeScopes; pnode;)

{

switch (pnode->nop) {

case knopFncDecl:

return true;

case knopBlock:

pnode = pnode->sxBlock.pnodeNext;

break;

case knopCatch:

pnode = pnode->sxCatch.pnodeNext;

break;

case knopWith:

pnode = pnode->sxWith.pnodeNext;

break;

default:

Assert(UNREACHED);

return true;

}

}

return false;

}

class ByteCodeGenerator;

// Copy AST; this works mostly on expressions for now

ParseNode\* Parser::CopyPnode(ParseNode \*pnode) {

if (pnode==NULL)

return NULL;

switch (pnode->nop) {

//PTNODE(knopName , "name" ,None ,Pid ,fnopLeaf)

case knopName: {

ParseNode\* nameNode=CreateNameNode(pnode->sxPid.pid,pnode->ichMin,pnode->ichLim);

nameNode->sxPid.sym=pnode->sxPid.sym;

return nameNode;

}

//PTNODE(knopInt , "int const" ,None ,Int ,fnopLeaf|fnopConst)

case knopInt:

return pnode;

//PTNODE(knopFlt , "flt const" ,None ,Flt ,fnopLeaf|fnopConst)

case knopFlt:

return pnode;

//PTNODE(knopStr , "str const" ,None ,Pid ,fnopLeaf|fnopConst)

case knopStr:

return pnode;

//PTNODE(knopRegExp , "reg expr" ,None ,Pid ,fnopLeaf|fnopConst)

case knopRegExp:

return pnode;

break;

//PTNODE(knopThis , "this" ,None ,None ,fnopLeaf)

case knopThis:

return CreateNodeT<knopThis>(pnode->ichMin,pnode->ichLim);

//PTNODE(knopNull , "null" ,Null ,None ,fnopLeaf)

case knopNull:

return pnode;

//PTNODE(knopFalse , "false" ,False ,None ,fnopLeaf)

case knopFalse:

return CreateNodeT<knopFalse>(pnode->ichMin,pnode->ichLim);

break;

//PTNODE(knopTrue , "true" ,True ,None ,fnopLeaf)

case knopTrue:

return CreateNodeT<knopTrue>(pnode->ichMin,pnode->ichLim);

//PTNODE(knopEmpty , "empty" ,Empty ,None ,fnopLeaf)

case knopEmpty:

return CreateNodeT<knopEmpty>(pnode->ichMin,pnode->ichLim);

// Unary operators.

//PTNODE(knopNot , "~" ,BitNot ,Uni ,fnopUni)

//PTNODE(knopNeg , "unary -" ,Neg ,Uni ,fnopUni)

//PTNODE(knopPos , "unary +" ,Pos ,Uni ,fnopUni)

//PTNODE(knopLogNot , "!" ,LogNot ,Uni ,fnopUni)

//PTNODE(knopEllipsis , "..." ,Spread ,Uni , fnopUni)

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

//PTNODE(knopIncPre , "++ pre" ,Inc ,Uni ,fnopUni|fnopAsg)

//PTNODE(knopDecPre , "-- pre" ,Dec ,Uni ,fnopUni|fnopAsg)

//PTNODE(knopTypeof , "typeof" ,None ,Uni ,fnopUni)

//PTNODE(knopVoid , "void" ,Void ,Uni ,fnopUni)

//PTNODE(knopDelete , "delete" ,None ,Uni ,fnopUni)

case knopNot:

case knopNeg:

case knopPos:

case knopLogNot:

case knopEllipsis:

case knopIncPost:

case knopDecPost:

case knopIncPre:

case knopDecPre:

case knopTypeof:

case knopVoid:

case knopDelete:

return CreateUniNode(pnode->nop,CopyPnode(pnode->sxUni.pnode1),pnode->ichMin,pnode->ichLim);

//PTNODE(knopArray , "arr cnst" ,None ,Uni ,fnopUni)

//PTNODE(knopObject , "obj cnst" ,None ,Uni ,fnopUni)

case knopArray:

case knopObject:

// TODO: need to copy arr

Assert(false);

break;

// Binary operators

//PTNODE(knopAdd , "+" ,Add ,Bin ,fnopBin)

//PTNODE(knopSub , "-" ,Sub ,Bin ,fnopBin)

//PTNODE(knopMul , "\*" ,Mul ,Bin ,fnopBin)

//PTNODE(knopExpo , "\*\*" ,Expo ,Bin ,fnopBin)

//PTNODE(knopDiv , "/" ,Div ,Bin ,fnopBin)

//PTNODE(knopMod , "%" ,Mod ,Bin ,fnopBin)

//PTNODE(knopOr , "|" ,BitOr ,Bin ,fnopBin)

//PTNODE(knopXor , "^" ,BitXor ,Bin ,fnopBin)

//PTNODE(knopAnd , "&" ,BitAnd ,Bin ,fnopBin)

//PTNODE(knopEq , "==" ,EQ ,Bin ,fnopBin|fnopRel)

//PTNODE(knopNe , "!=" ,NE ,Bin ,fnopBin|fnopRel)

//PTNODE(knopLt , "<" ,LT ,Bin ,fnopBin|fnopRel)

//PTNODE(knopLe , "<=" ,LE ,Bin ,fnopBin|fnopRel)

//PTNODE(knopGe , ">=" ,GE ,Bin ,fnopBin|fnopRel)

//PTNODE(knopGt , ">" ,GT ,Bin ,fnopBin|fnopRel)

//PTNODE(knopEqv , "===" ,Eqv ,Bin ,fnopBin|fnopRel)

//PTNODE(knopIn , "in" ,In ,Bin ,fnopBin|fnopRel)

//PTNODE(knopInstOf , "instanceof",InstOf ,Bin ,fnopBin|fnopRel)

//PTNODE(knopNEqv , "!==" ,NEqv ,Bin ,fnopBin|fnopRel)

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

//PTNODE(knopLogOr , "||" ,None ,Bin ,fnopBin)

//PTNODE(knopLogAnd , "&&" ,None ,Bin ,fnopBin)

//PTNODE(knopLsh , "<<" ,Lsh ,Bin ,fnopBin)

//PTNODE(knopRsh , ">>" ,Rsh ,Bin ,fnopBin)

//PTNODE(knopRs2 , ">>>" ,Rs2 ,Bin ,fnopBin)

case knopAdd:

case knopSub:

case knopMul:

case knopExpo:

case knopDiv:

case knopMod:

case knopOr:

case knopXor:

case knopAnd:

case knopEq:

case knopNe:

case knopLt:

case knopLe:

case knopGe:

case knopGt:

case knopEqv:

case knopIn:

case knopInstOf:

case knopNEqv:

case knopComma:

case knopLogOr:

case knopLogAnd:

case knopLsh:

case knopRsh:

case knopRs2:

//PTNODE(knopAsg , "=" ,None ,Bin ,fnopBin|fnopAsg)

case knopAsg:

//PTNODE(knopDot , "." ,None ,Bin ,fnopBin)

case knopDot:

//PTNODE(knopAsgAdd , "+=" ,Add ,Bin ,fnopBin|fnopAsg)

case knopAsgAdd:

//PTNODE(knopAsgSub , "-=" ,Sub ,Bin ,fnopBin|fnopAsg)

case knopAsgSub:

//PTNODE(knopAsgMul , "\*=" ,Mul ,Bin ,fnopBin|fnopAsg)

case knopAsgMul:

//PTNODE(knopAsgDiv , "/=" ,Div ,Bin ,fnopBin|fnopAsg)

case knopAsgExpo:

//PTNODE(knopAsgExpo , "\*\*=" ,Expo ,Bin ,fnopBin|fnopAsg)

case knopAsgDiv:

//PTNODE(knopAsgMod , "%=" ,Mod ,Bin ,fnopBin|fnopAsg)

case knopAsgMod:

//PTNODE(knopAsgAnd , "&=" ,BitAnd ,Bin ,fnopBin|fnopAsg)

case knopAsgAnd:

//PTNODE(knopAsgXor , "^=" ,BitXor ,Bin ,fnopBin|fnopAsg)

case knopAsgXor:

//PTNODE(knopAsgOr , "|=" ,BitOr ,Bin ,fnopBin|fnopAsg)

case knopAsgOr:

//PTNODE(knopAsgLsh , "<<=" ,Lsh ,Bin ,fnopBin|fnopAsg)

case knopAsgLsh:

//PTNODE(knopAsgRsh , ">>=" ,Rsh ,Bin ,fnopBin|fnopAsg)

case knopAsgRsh:

//PTNODE(knopAsgRs2 , ">>>=" ,Rs2 ,Bin ,fnopBin|fnopAsg)

case knopAsgRs2:

//PTNODE(knopMember , ":" ,None ,Bin ,fnopBin)

case knopMember:

case knopMemberShort:

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

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

case knopIndex:

case knopList:

return CreateBinNode(pnode->nop,CopyPnode(pnode->sxBin.pnode1),

CopyPnode(pnode->sxBin.pnode2),pnode->ichMin,pnode->ichLim);

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

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

case knopNew:

case knopCall:

return CreateCallNode(pnode->nop,CopyPnode(pnode->sxBin.pnode1),

CopyPnode(pnode->sxBin.pnode2),pnode->ichMin,pnode->ichLim);

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

case knopQmark:

return CreateTriNode(pnode->nop,CopyPnode(pnode->sxTri.pnode1),

CopyPnode(pnode->sxTri.pnode2),CopyPnode(pnode->sxTri.pnode3),

pnode->ichMin,pnode->ichLim);

// General nodes.

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

case knopVarDecl: {

ParseNode\* copyNode=CreateNodeT<knopVarDecl>(pnode->ichMin,pnode->ichLim);

copyNode->sxVar.pnodeInit=CopyPnode(pnode->sxVar.pnodeInit);

copyNode->sxVar.sym=pnode->sxVar.sym;

// TODO: mult-decl

Assert(pnode->sxVar.pnodeNext==NULL);

copyNode->sxVar.pnodeNext=NULL;

return copyNode;

}

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

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

case knopFncDecl:

case knopProg:

Assert(false);

break;

//PTNODE(knopEndCode , "<endcode>" ,None ,None ,fnopNone)

case knopEndCode:

break;

//PTNODE(knopDebugger , "debugger" ,None ,None ,fnopNone)

case knopDebugger:

break;

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

case knopFor: {

ParseNode\* copyNode=CreateNodeT<knopFor>(pnode->ichMin,pnode->ichLim);

copyNode->sxFor.pnodeInverted=NULL;

copyNode->sxFor.pnodeInit=CopyPnode(pnode->sxFor.pnodeInit);

copyNode->sxFor.pnodeCond=CopyPnode(pnode->sxFor.pnodeCond);

copyNode->sxFor.pnodeIncr=CopyPnode(pnode->sxFor.pnodeIncr);

copyNode->sxFor.pnodeBody=CopyPnode(pnode->sxFor.pnodeBody);

return copyNode;

}

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

case knopIf:

Assert(false);

break;

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

case knopWhile:

Assert(false);

break;

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

case knopDoWhile:

Assert(false);

break;

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

case knopForIn:

Assert(false);

break;

case knopForOf:

Assert(false);

break;

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

case knopReturn: {

ParseNode\* copyNode=CreateNodeT<knopReturn>(pnode->ichMin,pnode->ichLim);

copyNode->sxReturn.pnodeExpr=CopyPnode(pnode->sxReturn.pnodeExpr);

return copyNode;

}

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

case knopBlock: {

ParseNode\* copyNode=CreateBlockNode(pnode->ichMin,pnode->ichLim,pnode->sxBlock.blockType);

if (pnode->grfpn & PNodeFlags::fpnSyntheticNode) {

// fpnSyntheticNode is sometimes set on PnodeBlockType::Regular blocks which

// CreateBlockNode() will not automatically set for us, so set it here if it's

// specified on the source node.

copyNode->grfpn |= PNodeFlags::fpnSyntheticNode;

}

copyNode->sxBlock.pnodeStmt=CopyPnode(pnode->sxBlock.pnodeStmt);

return copyNode;

}

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

case knopWith:

Assert(false);

break;

//PTNODE(knopBreak , "break" ,None ,Jump ,fnopNone)

case knopBreak:

Assert(false);

break;

//PTNODE(knopContinue , "continue" ,None ,Jump ,fnopNone)

case knopContinue:

Assert(false);

break;

//PTNODE(knopLabel , "label" ,None ,Label,fnopNone)

case knopLabel:

Assert(false);

break;

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

case knopSwitch:

Assert(false);

break;

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

case knopCase:

Assert(false);

break;

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

case knopTryFinally:

Assert(false);

break;

case knopFinally:

Assert(false);

break;

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

case knopCatch:

Assert(false);

break;

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

case knopTryCatch:

Assert(false);

break;

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

case knopTry:

Assert(false);

break;

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

case knopThrow:

Assert(false);

break;

default:

Assert(false);

break;

}

return NULL;

}

// Returns true when str is string for Nan, Infinity or -Infinity.

// Does not check for double number value being in NaN/Infinity range.

// static

template<bool CheckForNegativeInfinity>

inline bool Parser::IsNaNOrInfinityLiteral(LPCOLESTR str)

{

// Note: wcscmp crashes when one of the parameters is NULL.

return str &&

(wcscmp(L"NaN", str) == 0 ||

wcscmp(L"Infinity", str) == 0 ||

CheckForNegativeInfinity && wcscmp(L"-Infinity", str) == 0);

}

template <bool buildAST>

ParseNodePtr Parser::ParseSuper(ParseNodePtr pnode, bool fAllowCall)

{

ParseNodePtr currentNodeFunc = GetCurrentFunctionNode();

if (buildAST) {

pnode = CreateNodeWithScanner<knopSuper>();

}

m\_pscan->ScanForcingPid();

switch (m\_token.tk)

{

case tkDot: // super.prop

case tkLBrack: // super[foo]

case tkLParen: // super(args)

break;

default:

Error(ERRInvalidSuper);

break;

}

if (!fAllowCall && (m\_token.tk == tkLParen))

{

Error(ERRInvalidSuper); // new super() is not allowed

}

else if (this->m\_parsingSuperRestrictionState == ParsingSuperRestrictionState\_SuperCallAndPropertyAllowed)

{

// Any super access is good within a class constructor

}

else if (this->m\_parsingSuperRestrictionState == ParsingSuperRestrictionState\_SuperPropertyAllowed)

{

// Cannot call super within a class member

if (m\_token.tk == tkLParen)

{

Error(ERRInvalidSuper);

}

}

else

{

// Anything else is an error

Error(ERRInvalidSuper);

}

currentNodeFunc->sxFnc.SetHasSuperReference(TRUE);

CHAKRATEL\_LANGSTATS\_INC\_LANGFEATURECOUNT(SuperCount, m\_scriptContext);

return pnode;

}

void Parser::AppendToList(ParseNodePtr \*node, ParseNodePtr nodeToAppend)

{

Assert(nodeToAppend);

ParseNodePtr\* lastPtr = node;

while ((\*lastPtr) && (\*lastPtr)->nop == knopList)

{

lastPtr = &(\*lastPtr)->sxBin.pnode2;

}

auto last = (\*lastPtr);

if (last)

{

\*lastPtr = CreateBinNode(knopList, last, nodeToAppend, last->ichMin, nodeToAppend->ichLim);

}

else

{

\*lastPtr = nodeToAppend;

}

}

ParseNodePtr Parser::ConvertArrayToArrayPattern(ParseNodePtr pnode)

{

Assert(pnode->nop == knopArray);

pnode->nop = knopArrayPattern;

ForEachItemRefInList(&pnode->sxArrLit.pnode1, [&](ParseNodePtr \*itemRef) {

ParseNodePtr item = \*itemRef;

if (item->nop == knopEllipsis)

{

itemRef = &item->sxUni.pnode1;

item = \*itemRef;

if (!(item->nop == knopName

|| item->nop == knopDot

|| item->nop == knopIndex

|| item->nop == knopArray

|| item->nop == knopObject))

{

Error(ERRInvalidAssignmentTarget);

}

}

else if (item->nop == knopAsg)

{

itemRef = &item->sxBin.pnode1;

item = \*itemRef;

}

if (item->nop == knopArray)

{

ConvertArrayToArrayPattern(item);

}

else if (item->nop == knopObject)

{

\*itemRef = ConvertObjectToObjectPattern(item);

}

});

return pnode;

}

ParseNodePtr Parser::CreateParamPatternNode(ParseNodePtr pnode1)

{

ParseNodePtr paramPatternNode = CreateNode(knopParamPattern, pnode1->ichMin, pnode1->ichLim);

paramPatternNode->sxParamPattern.pnode1 = pnode1;

paramPatternNode->sxParamPattern.pnodeNext = nullptr;

paramPatternNode->sxParamPattern.location = Js::Constants::NoRegister;

return paramPatternNode;

}

ParseNodePtr Parser::ConvertObjectToObjectPattern(ParseNodePtr pnodeMemberList)

{

charcount\_t ichMin = m\_pscan->IchMinTok();

charcount\_t ichLim = m\_pscan->IchLimTok();

ParseNodePtr pnodeMemberNodeList = nullptr;

if (pnodeMemberList != nullptr && pnodeMemberList->nop == knopObject)

{

ichMin = pnodeMemberList->ichMin;

ichLim = pnodeMemberList->ichLim;

pnodeMemberList = pnodeMemberList->sxUni.pnode1;

}

ForEachItemInList(pnodeMemberList, [&](ParseNodePtr item) {

ParseNodePtr memberNode = ConvertMemberToMemberPattern(item);

AppendToList(&pnodeMemberNodeList, memberNode);

});

return CreateUniNode(knopObjectPattern, pnodeMemberNodeList, ichMin, ichLim);

}

ParseNodePtr Parser::GetRightSideNodeFromPattern(ParseNodePtr pnode)

{

Assert(pnode != nullptr);

ParseNodePtr rightNode = nullptr;

OpCode op = pnode->nop;

if (op == knopObject)

{

rightNode = ConvertObjectToObjectPattern(pnode);

}

else if (op == knopArray)

{

rightNode = ConvertArrayToArrayPattern(pnode);

}

else

{

// we should allow

// references (name/string/knopDots and knopIndex)

// Allow assignment operator for initializer

// rest is syntax error.

if (!(op == knopName || op == knopStr || op == knopDot || op == knopIndex || op == knopAsg))

{

if (m\_token.IsOperator())

{

Error(ERRDestructNoOper);

}

Error(ERRDestructIDRef);

}

rightNode = pnode;

}

return rightNode;

}

ParseNodePtr Parser::ConvertMemberToMemberPattern(ParseNodePtr pnodeMember)

{

Assert(pnodeMember->nop == knopMember || pnodeMember->nop == knopMemberShort);

ParseNodePtr rightNode = GetRightSideNodeFromPattern(pnodeMember->sxBin.pnode2);

ParseNodePtr resultNode = CreateBinNode(knopObjectPatternMember, pnodeMember->sxBin.pnode1, rightNode);

resultNode->ichMin = pnodeMember->ichMin;

resultNode->ichLim = pnodeMember->ichLim;

return resultNode;

}

ParseNodePtr Parser::ConvertToPattern(ParseNodePtr pnode)

{

if (pnode != nullptr)

{

if (pnode->nop == knopArray)

{

ConvertArrayToArrayPattern(pnode);

}

else if (pnode->nop == knopObject)

{

pnode = ConvertObjectToObjectPattern(pnode);

}

}

return pnode;

}

// This essentially be called for verifying the structure of the current tree with satisfying the destructuring grammar.

void Parser::ParseDestructuredLiteralWithScopeSave(tokens declarationType,

bool isDecl,

bool topLevel,

DestructuringInitializerContext initializerContext/\* = DIC\_None\*/,

bool allowIn /\*= true\*/)

{

// We are going to parse the text again to validate the current grammar as Destructuring. Saving some scopes and

// AST related information before the validation parsing and later they will be restored.

ParseNodePtr pnodeFncSave = m\_currentNodeFunc;

long \*pAstSizeSave = m\_pCurrentAstSize;

uint \*pNestedCountSave = m\_pnestedCount;

ParseNodePtr \*ppnodeScopeSave = m\_ppnodeScope;

ParseNodePtr \*ppnodeExprScopeSave = m\_ppnodeExprScope;

ParseNodePtr newTempScope = nullptr;

m\_ppnodeScope = &newTempScope;

long newTempAstSize = 0;

m\_pCurrentAstSize = &newTempAstSize;

uint newTempNestedCount = 0;

m\_pnestedCount = &newTempNestedCount;

m\_ppnodeExprScope = nullptr;

ParseDestructuredLiteral<false>(declarationType, isDecl, topLevel, initializerContext, allowIn);

m\_currentNodeFunc = pnodeFncSave;

m\_pCurrentAstSize = pAstSizeSave;

m\_pnestedCount = pNestedCountSave;

m\_ppnodeScope = ppnodeScopeSave;

m\_ppnodeExprScope = ppnodeExprScopeSave;

}

template <bool buildAST>

ParseNodePtr Parser::ParseDestructuredLiteral(tokens declarationType,

bool isDecl,

bool topLevel/\* = true\*/,

DestructuringInitializerContext initializerContext/\* = DIC\_None\*/,

bool allowIn/\* = true\*/,

BOOL \*forInOfOkay/\* = nullptr\*/)

{

ParseNodePtr pnode = nullptr;

Assert(IsPossiblePatternStart());

if (m\_token.tk == tkLCurly)

{

pnode = ParseDestructuredObjectLiteral<buildAST>(declarationType, isDecl, topLevel);

}

else

{

pnode = ParseDestructuredArrayLiteral<buildAST>(declarationType, isDecl, topLevel);

}

return ParseDestructuredInitializer<buildAST>(pnode, isDecl, topLevel, initializerContext, allowIn, forInOfOkay);

}

template <bool buildAST>

ParseNodePtr Parser::ParseDestructuredInitializer(ParseNodePtr lhsNode,

bool isDecl,

bool topLevel,

DestructuringInitializerContext initializerContext,

bool allowIn,

BOOL \*forInOfOkay)

{

m\_pscan->Scan();

if (topLevel)

{

if (initializerContext != DIC\_ForceErrorOnInitializer && m\_token.tk != tkAsg)

{

// e.g. var {x};

Error(ERRDestructInit);

}

else if (initializerContext == DIC\_ForceErrorOnInitializer && m\_token.tk == tkAsg)

{

// e.g. catch([x] = [0])

Error(ERRDestructNotInit);

}

}

if (m\_token.tk != tkAsg || initializerContext == DIC\_ShouldNotParseInitializer)

{

return lhsNode;

}

if (forInOfOkay)

{

\*forInOfOkay = FALSE;

}

m\_pscan->Scan();

ParseNodePtr pnodeDefault = ParseExpr<buildAST>(koplCma, nullptr, allowIn);

ParseNodePtr pnodeDestructAsg = nullptr;

if (buildAST)

{

Assert(lhsNode != nullptr);

pnodeDestructAsg = CreateNodeWithScanner<knopAsg>();

pnodeDestructAsg->sxBin.pnode1 = lhsNode;

pnodeDestructAsg->sxBin.pnode2 = pnodeDefault;

pnodeDestructAsg->ichMin = lhsNode->ichMin;

pnodeDestructAsg->ichLim = pnodeDefault->ichLim;

}

return pnodeDestructAsg;

}

template <bool buildAST>

ParseNodePtr Parser::ParseDestructuredObjectLiteral(tokens declarationType, bool isDecl, bool topLevel/\* = true\*/)

{

Assert(m\_token.tk == tkLCurly);

charcount\_t ichMin = m\_pscan->IchMinTok();

m\_pscan->Scan();

if (!isDecl)

{

declarationType = tkLCurly;

}

ParseNodePtr pnodeMemberList = ParseMemberList<buildAST>(nullptr/\*pNameHint\*/, nullptr/\*pHintLength\*/, declarationType);

Assert(m\_token.tk == tkRCurly);

ParseNodePtr objectPatternNode = nullptr;

if (buildAST)

{

charcount\_t ichLim = m\_pscan->IchLimTok();

objectPatternNode = CreateUniNode(knopObjectPattern, pnodeMemberList, ichMin, ichLim);

}

return objectPatternNode;

}

template <bool buildAST>

ParseNodePtr Parser::ParseDestructuredVarDecl(tokens declarationType, bool isDecl, bool \*hasSeenRest, bool topLevel/\* = true\*/)

{

ParseNodePtr pnodeElem = nullptr;

int parenCount = 0;

bool seenRest = false;

while (m\_token.tk == tkLParen)

{

m\_pscan->Scan();

++parenCount;

}

if (m\_token.tk == tkEllipsis)

{

// As per ES 2015 : Rest can have left-hand-side-expression when on assignment expression, but under declaration only binding identifier is allowed

// But spec is going to change for this one to allow LHS-expression both on expression and declaration - so making that happen early.

seenRest = true;

m\_pscan->Scan();

while (m\_token.tk == tkLParen)

{

m\_pscan->Scan();

++parenCount;

}

if (m\_token.tk != tkID && m\_token.tk != tkSUPER && m\_token.tk != tkLCurly && m\_token.tk != tkLBrack)

{

if (isDecl)

{

Error(ERRnoIdent);

}

else

{

Error(ERRInvalidAssignmentTarget);

}

}

}

if (IsPossiblePatternStart())

{

// Go recursively

pnodeElem = ParseDestructuredLiteral<buildAST>(declarationType, isDecl, false /\*topLevel\*/);

}

else if (m\_token.tk == tkSUPER || m\_token.tk == tkID)

{

if (isDecl)

{

charcount\_t ichMin = m\_pscan->IchMinTok();

pnodeElem = ParseVariableDeclaration<buildAST>(declarationType, ichMin

,/\* fAllowIn \*/false, /\* pfForInOk \*/nullptr, /\* singleDefOnly \*/true, /\* allowInit \*/!seenRest, false /\*topLevelParse\*/);

}

else

{

BOOL fCanAssign;

IdentToken token;

// We aren't declaring anything, so scan the ID reference manually.

pnodeElem = ParseTerm<buildAST>(/\* fAllowCall \*/ m\_token.tk != tkSUPER, nullptr /\*pNameHint\*/, nullptr /\*pHintLength\*/, nullptr /\*pShortNameOffset\*/, &token, false,

&fCanAssign);

if (!fCanAssign && PHASE\_ON1(Js::EarlyReferenceErrorsPhase))

{

Error(JSERR\_CantAssignTo);

}

if (buildAST)

{

if (IsStrictMode() && pnodeElem != nullptr && pnodeElem->nop == knopName)

{

CheckStrictModeEvalArgumentsUsage(pnodeElem->sxPid.pid);

}

}

else

{

if (IsStrictMode() && token.tk == tkID)

{

CheckStrictModeEvalArgumentsUsage(token.pid);

}

token.tk = tkNone;

}

}

}

else if (!(m\_token.tk == tkComma || m\_token.tk == tkRBrack || m\_token.tk == tkRCurly))

{

if (m\_token.IsOperator())

{

Error(ERRDestructNoOper);

}

Error(ERRDestructIDRef);

}

// Swallow RParens before a default expression, if any.

while (m\_token.tk == tkRParen)

{

m\_pscan->Scan();

--parenCount;

}

if (hasSeenRest != nullptr)

{

\*hasSeenRest = seenRest;

}

if (m\_token.tk == tkAsg)

{

// Parse the initializer.

if (seenRest)

{

Error(ERRRestWithDefault);

}

m\_pscan->Scan();

ParseNodePtr pnodeInit = ParseExpr<buildAST>(koplCma);

if (buildAST)

{

pnodeElem = CreateBinNode(knopAsg, pnodeElem, pnodeInit);

}

}

if (buildAST && seenRest)

{

ParseNodePtr pnodeRest = CreateNodeWithScanner<knopEllipsis>();

pnodeRest->sxUni.pnode1 = pnodeElem;

pnodeElem = pnodeRest;

}

while (m\_token.tk == tkRParen)

{

m\_pscan->Scan();

--parenCount;

}

if (!(m\_token.tk == tkComma || m\_token.tk == tkRBrack || m\_token.tk == tkRCurly))

{

if (m\_token.IsOperator())

{

Error(ERRDestructNoOper);

}

Error(ERRsyntax);

}

if (parenCount != 0)

{

Error(ERRnoRparen);

}

return pnodeElem;

}

template <bool buildAST>

ParseNodePtr Parser::ParseDestructuredArrayLiteral(tokens declarationType, bool isDecl, bool topLevel)

{

Assert(m\_token.tk == tkLBrack);

charcount\_t ichMin = m\_pscan->IchMinTok();

m\_pscan->Scan();

ParseNodePtr pnodeDestructArr = nullptr;

ParseNodePtr pnodeList = nullptr;

ParseNodePtr \*lastNodeRef = nullptr;

uint count = 0;

bool hasMissingValues = false;

bool seenRest = false;

while (true)

{

if (seenRest) // Rest must be in the last position.

{

Error(ERRDestructRestLast);

}

ParseNodePtr pnodeElem = ParseDestructuredVarDecl<buildAST>(declarationType, isDecl, &seenRest, topLevel);

if (buildAST)

{

if (pnodeElem == nullptr && buildAST)

{

pnodeElem = CreateNodeWithScanner<knopEmpty>();

hasMissingValues = true;

}

AddToNodeListEscapedUse(&pnodeList, &lastNodeRef, pnodeElem);

}

count++;

if (m\_token.tk == tkRBrack)

{

break;

}

if (m\_token.tk != tkComma)

{

Error(ERRDestructNoOper);

}

m\_pscan->Scan();

}

if (buildAST)

{

pnodeDestructArr = CreateNodeWithScanner<knopArrayPattern>();

pnodeDestructArr->sxArrLit.pnode1 = pnodeList;

pnodeDestructArr->sxArrLit.arrayOfTaggedInts = false;

pnodeDestructArr->sxArrLit.arrayOfInts = false;

pnodeDestructArr->sxArrLit.arrayOfNumbers = false;

pnodeDestructArr->sxArrLit.hasMissingValues = hasMissingValues;

pnodeDestructArr->sxArrLit.count = count;

pnodeDestructArr->sxArrLit.spreadCount = seenRest ? 1 : 0;

pnodeDestructArr->ichMin = ichMin;

pnodeDestructArr->ichLim = m\_pscan->IchLimTok();

if (pnodeDestructArr->sxArrLit.pnode1)

{

this->CheckArguments(pnodeDestructArr->sxArrLit.pnode1);

}

}

return pnodeDestructArr;

}

void Parser::CaptureContext(ParseContext \*parseContext) const

{

parseContext->pszSrc = m\_pscan->PchBase();

parseContext->length = this->m\_originalLength;

parseContext->characterOffset = m\_pscan->IchMinTok();

parseContext->offset = parseContext->characterOffset + m\_pscan->m\_cMultiUnits;

parseContext->grfscr = this->m\_grfscr;

parseContext->lineNumber = m\_pscan->LineCur();

parseContext->pnodeProg = this->m\_currentNodeProg;

parseContext->fromExternal = m\_pscan->IsFromExternalSource();

parseContext->strictMode = this->IsStrictMode();

parseContext->sourceContextInfo = this->m\_sourceContextInfo;

parseContext->currentBlockInfo = this->m\_currentBlockInfo;

parseContext->nextBlockId = this->m\_nextBlockId;

}

void Parser::RestoreContext(ParseContext \*const parseContext)

{

m\_sourceContextInfo = parseContext->sourceContextInfo;

m\_currentBlockInfo = parseContext->currentBlockInfo;

m\_nextBlockId = parseContext->nextBlockId;

m\_grfscr = parseContext->grfscr;

m\_length = parseContext->length;

m\_pscan->SetText(parseContext->pszSrc, parseContext->offset, parseContext->length, parseContext->characterOffset, parseContext->grfscr, parseContext->lineNumber);

m\_currentNodeProg = parseContext->pnodeProg;

m\_fUseStrictMode = parseContext->strictMode;

}

class ByteCodeGenerator;

#if DBG\_DUMP

#define INDENT\_SIZE 2

void PrintPnodeListWIndent(ParseNode \*pnode,int indentAmt);

void PrintFormalsWIndent(ParseNode \*pnode, int indentAmt);

void Indent(int indentAmt) {

for (int i=0;i<indentAmt;i++) {

Output::Print(L" ");

}

}

void PrintScopesWIndent(ParseNode \*pnode,int indentAmt) {

ParseNode \*scope = nullptr;

bool firstOnly = false;

switch(pnode->nop)

{

case knopProg:

case knopFncDecl: scope = pnode->sxFnc.pnodeScopes; break;

case knopBlock: scope = pnode->sxBlock.pnodeScopes; break;

case knopCatch: scope = pnode->sxCatch.pnodeScopes; break;

case knopWith: scope = pnode->sxWith.pnodeScopes; break;

case knopSwitch: scope = pnode->sxSwitch.pnodeBlock; firstOnly = true; break;

case knopFor: scope = pnode->sxFor.pnodeBlock; firstOnly = true; break;

case knopForIn: scope = pnode->sxForInOrForOf.pnodeBlock; firstOnly = true; break;

case knopForOf: scope = pnode->sxForInOrForOf.pnodeBlock; firstOnly = true; break;

}

if (scope) {

Indent(indentAmt);

Output::Print(L"Scopes: ");

ParseNode \*next = nullptr;

ParseNode \*syntheticBlock = nullptr;

while (scope) {

switch (scope->nop) {

case knopFncDecl: Output::Print(L"knopFncDecl"); next = scope->sxFnc.pnodeNext; break;

case knopBlock: Output::Print(L"knopBlock"); next = scope->sxBlock.pnodeNext; break;

case knopCatch: Output::Print(L"knopCatch"); next = scope->sxCatch.pnodeNext; break;

case knopWith: Output::Print(L"knopWith"); next = scope->sxWith.pnodeNext; break;

default: Output::Print(L"unknown"); break;

}

if (firstOnly) {

next = nullptr;

syntheticBlock = scope;

}

if (scope->grfpn & fpnSyntheticNode) {

Output::Print(L" synthetic");

if (scope->nop == knopBlock)

syntheticBlock = scope;

}

Output::Print(L" (%d-%d)", scope->ichMin, scope->ichLim);

if (next) Output::Print(L", ");

scope = next;

}

Output::Print(L"\n");

if (syntheticBlock || firstOnly) {

PrintScopesWIndent(syntheticBlock, indentAmt + INDENT\_SIZE);

}

}

}

void PrintPnodeWIndent(ParseNode \*pnode,int indentAmt) {

if (pnode==NULL)

return;

Output::Print(L"[%d, %d): ", pnode->ichMin, pnode->ichLim);

switch (pnode->nop) {

//PTNODE(knopName , "name" ,None ,Pid ,fnopLeaf)

case knopName:

Indent(indentAmt);

if (pnode->sxPid.pid!=NULL) {

Output::Print(L"id: %s\n",pnode->sxPid.pid->Psz());

}

else {

Output::Print(L"name node\n");

}

break;

//PTNODE(knopInt , "int const" ,None ,Int ,fnopLeaf|fnopConst)

case knopInt:

Indent(indentAmt);

Output::Print(L"%d\n",pnode->sxInt.lw);

break;

//PTNODE(knopFlt , "flt const" ,None ,Flt ,fnopLeaf|fnopConst)

case knopFlt:

Indent(indentAmt);

Output::Print(L"%lf\n",pnode->sxFlt.dbl);

break;

//PTNODE(knopStr , "str const" ,None ,Pid ,fnopLeaf|fnopConst)

case knopStr:

Indent(indentAmt);

Output::Print(L"\"%s\"\n",pnode->sxPid.pid->Psz());

break;

//PTNODE(knopRegExp , "reg expr" ,None ,Pid ,fnopLeaf|fnopConst)

case knopRegExp:

Indent(indentAmt);

Output::Print(L"/%x/\n",pnode->sxPid.regexPattern);

break;

//PTNODE(knopThis , "this" ,None ,None ,fnopLeaf)

case knopThis:

Indent(indentAmt);

Output::Print(L"this\n");

break;

//PTNODE(knopSuper , "super" ,None ,None ,fnopLeaf)

case knopSuper:

Indent(indentAmt);

Output::Print(L"super\n");

break;

//PTNODE(knopNewTarget , "new.target" ,None ,None ,fnopLeaf)

case knopNewTarget:

Indent(indentAmt);

Output::Print(L"new.target\n");

break;

//PTNODE(knopNull , "null" ,Null ,None ,fnopLeaf)

case knopNull:

Indent(indentAmt);

Output::Print(L"null\n");

break;

//PTNODE(knopFalse , "false" ,False ,None ,fnopLeaf)

case knopFalse:

Indent(indentAmt);

Output::Print(L"false\n");

break;

//PTNODE(knopTrue , "true" ,True ,None ,fnopLeaf)

case knopTrue:

Indent(indentAmt);

Output::Print(L"true\n");

break;

//PTNODE(knopEmpty , "empty" ,Empty ,None ,fnopLeaf)

case knopEmpty:

Indent(indentAmt);

Output::Print(L"empty\n");

break;

// Unary operators.

//PTNODE(knopNot , "~" ,BitNot ,Uni ,fnopUni)

case knopNot:

Indent(indentAmt);

Output::Print(L"~\n");

PrintPnodeWIndent(pnode->sxUni.pnode1,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopNeg , "unary -" ,Neg ,Uni ,fnopUni)

case knopNeg:

Indent(indentAmt);

Output::Print(L"U-\n");

PrintPnodeWIndent(pnode->sxUni.pnode1,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopPos , "unary +" ,Pos ,Uni ,fnopUni)

case knopPos:

Indent(indentAmt);

Output::Print(L"U+\n");

PrintPnodeWIndent(pnode->sxUni.pnode1,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopLogNot , "!" ,LogNot ,Uni ,fnopUni)

case knopLogNot:

Indent(indentAmt);

Output::Print(L"!\n");

PrintPnodeWIndent(pnode->sxUni.pnode1,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopEllipsis , "..." ,Spread ,Uni , fnopUni)

case knopEllipsis:

Indent(indentAmt);

Output::Print(L"...<expr>\n");

PrintPnodeWIndent(pnode->sxUni.pnode1,indentAmt+INDENT\_SIZE);

break;

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

case knopIncPost:

Indent(indentAmt);

Output::Print(L"<expr>++\n");

PrintPnodeWIndent(pnode->sxUni.pnode1,indentAmt+INDENT\_SIZE);

break;

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

case knopDecPost:

Indent(indentAmt);

Output::Print(L"<expr>--\n");

PrintPnodeWIndent(pnode->sxUni.pnode1,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopIncPre , "++ pre" ,Inc ,Uni ,fnopUni|fnopAsg)

case knopIncPre:

Indent(indentAmt);

Output::Print(L"++<expr>\n");

PrintPnodeWIndent(pnode->sxUni.pnode1,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopDecPre , "-- pre" ,Dec ,Uni ,fnopUni|fnopAsg)

case knopDecPre:

Indent(indentAmt);

Output::Print(L"--<expr>\n");

PrintPnodeWIndent(pnode->sxUni.pnode1,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopTypeof , "typeof" ,None ,Uni ,fnopUni)

case knopTypeof:

Indent(indentAmt);

Output::Print(L"typeof\n");

PrintPnodeWIndent(pnode->sxUni.pnode1,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopVoid , "void" ,Void ,Uni ,fnopUni)

case knopVoid:

Indent(indentAmt);

Output::Print(L"void\n");

PrintPnodeWIndent(pnode->sxUni.pnode1,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopDelete , "delete" ,None ,Uni ,fnopUni)

case knopDelete:

Indent(indentAmt);

Output::Print(L"delete\n");

PrintPnodeWIndent(pnode->sxUni.pnode1,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopArray , "arr cnst" ,None ,Uni ,fnopUni)

case knopArrayPattern:

Indent(indentAmt);

Output::Print(L"Array Pattern\n");

PrintPnodeListWIndent(pnode->sxUni.pnode1, indentAmt + INDENT\_SIZE);

break;

case knopObjectPattern:

Indent(indentAmt);

Output::Print(L"Object Pattern\n");

PrintPnodeListWIndent(pnode->sxUni.pnode1, indentAmt + INDENT\_SIZE);

break;

case knopArray:

Indent(indentAmt);

Output::Print(L"Array Literal\n");

PrintPnodeListWIndent(pnode->sxUni.pnode1,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopObject , "obj cnst" ,None ,Uni ,fnopUni)

case knopObject:

Indent(indentAmt);

Output::Print(L"Object Literal\n");

PrintPnodeListWIndent(pnode->sxUni.pnode1,indentAmt+INDENT\_SIZE);

break;

// Binary and Ternary Operators

//PTNODE(knopAdd , "+" ,Add ,Bin ,fnopBin)

case knopAdd:

Indent(indentAmt);

Output::Print(L"+\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopSub , "-" ,Sub ,Bin ,fnopBin)

case knopSub:

Indent(indentAmt);

Output::Print(L"-\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopMul , "\*" ,Mul ,Bin ,fnopBin)

case knopMul:

Indent(indentAmt);

Output::Print(L"\*\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopDiv , "/" ,Div ,Bin ,fnopBin)

case knopExpo:

Indent(indentAmt);

Output::Print(L"\*\*\n");

PrintPnodeWIndent(pnode->sxBin.pnode1, indentAmt + INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2, indentAmt + INDENT\_SIZE);

break;

//PTNODE(knopExpo , "\*\*" ,Expo ,Bin ,fnopBin)

case knopDiv:

Indent(indentAmt);

Output::Print(L"/\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopMod , "%" ,Mod ,Bin ,fnopBin)

case knopMod:

Indent(indentAmt);

Output::Print(L"%\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopOr , "|" ,BitOr ,Bin ,fnopBin)

case knopOr:

Indent(indentAmt);

Output::Print(L"|\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopXor , "^" ,BitXor ,Bin ,fnopBin)

case knopXor:

Indent(indentAmt);

Output::Print(L"^\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopAnd , "&" ,BitAnd ,Bin ,fnopBin)

case knopAnd:

Indent(indentAmt);

Output::Print(L"&\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopEq , "==" ,EQ ,Bin ,fnopBin|fnopRel)

case knopEq:

Indent(indentAmt);

Output::Print(L"==\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopNe , "!=" ,NE ,Bin ,fnopBin|fnopRel)

case knopNe:

Indent(indentAmt);

Output::Print(L"!=\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopLt , "<" ,LT ,Bin ,fnopBin|fnopRel)

case knopLt:

Indent(indentAmt);

Output::Print(L"<\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopLe , "<=" ,LE ,Bin ,fnopBin|fnopRel)

case knopLe:

Indent(indentAmt);

Output::Print(L"<=\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopGe , ">=" ,GE ,Bin ,fnopBin|fnopRel)

case knopGe:

Indent(indentAmt);

Output::Print(L">=\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopGt , ">" ,GT ,Bin ,fnopBin|fnopRel)

case knopGt:

Indent(indentAmt);

Output::Print(L">\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

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

case knopCall:

Indent(indentAmt);

Output::Print(L"Call\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeListWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopDot , "." ,None ,Bin ,fnopBin)

case knopDot:

Indent(indentAmt);

Output::Print(L".\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopAsg , "=" ,None ,Bin ,fnopBin|fnopAsg)

case knopAsg:

Indent(indentAmt);

Output::Print(L"=\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopInstOf , "instanceof",InstOf ,Bin ,fnopBin|fnopRel)

case knopInstOf:

Indent(indentAmt);

Output::Print(L"instanceof\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopIn , "in" ,In ,Bin ,fnopBin|fnopRel)

case knopIn:

Indent(indentAmt);

Output::Print(L"in\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopEqv , "===" ,Eqv ,Bin ,fnopBin|fnopRel)

case knopEqv:

Indent(indentAmt);

Output::Print(L"===\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopNEqv , "!==" ,NEqv ,Bin ,fnopBin|fnopRel)

case knopNEqv:

Indent(indentAmt);

Output::Print(L"!==\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

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

case knopComma:

Indent(indentAmt);

Output::Print(L",\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopLogOr , "||" ,None ,Bin ,fnopBin)

case knopLogOr:

Indent(indentAmt);

Output::Print(L"||\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopLogAnd , "&&" ,None ,Bin ,fnopBin)

case knopLogAnd:

Indent(indentAmt);

Output::Print(L"&&\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopLsh , "<<" ,Lsh ,Bin ,fnopBin)

case knopLsh:

Indent(indentAmt);

Output::Print(L"<<\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopRsh , ">>" ,Rsh ,Bin ,fnopBin)

case knopRsh:

Indent(indentAmt);

Output::Print(L">>\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopRs2 , ">>>" ,Rs2 ,Bin ,fnopBin)

case knopRs2:

Indent(indentAmt);

Output::Print(L">>>\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

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

case knopNew:

Indent(indentAmt);

Output::Print(L"new\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeListWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

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

case knopIndex:

Indent(indentAmt);

Output::Print(L"[]\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeListWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

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

case knopQmark:

Indent(indentAmt);

Output::Print(L"?:\n");

PrintPnodeWIndent(pnode->sxTri.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxTri.pnode2,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxTri.pnode3,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopAsgAdd , "+=" ,Add ,Bin ,fnopBin|fnopAsg)

case knopAsgAdd:

Indent(indentAmt);

Output::Print(L"+=\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopAsgSub , "-=" ,Sub ,Bin ,fnopBin|fnopAsg)

case knopAsgSub:

Indent(indentAmt);

Output::Print(L"-=\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopAsgMul , "\*=" ,Mul ,Bin ,fnopBin|fnopAsg)

case knopAsgMul:

Indent(indentAmt);

Output::Print(L"\*=\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopAsgDiv , "/=" ,Div ,Bin ,fnopBin|fnopAsg)

case knopAsgExpo:

Indent(indentAmt);

Output::Print(L"\*\*=\n");

PrintPnodeWIndent(pnode->sxBin.pnode1, indentAmt + INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2, indentAmt + INDENT\_SIZE);

break;

//PTNODE(knopAsgExpo , "\*\*=" ,Expo ,Bin ,fnopBin|fnopAsg)

case knopAsgDiv:

Indent(indentAmt);

Output::Print(L"/=\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopAsgMod , "%=" ,Mod ,Bin ,fnopBin|fnopAsg)

case knopAsgMod:

Indent(indentAmt);

Output::Print(L"%=\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopAsgAnd , "&=" ,BitAnd ,Bin ,fnopBin|fnopAsg)

case knopAsgAnd:

Indent(indentAmt);

Output::Print(L"&=\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopAsgXor , "^=" ,BitXor ,Bin ,fnopBin|fnopAsg)

case knopAsgXor:

Indent(indentAmt);

Output::Print(L"^=\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopAsgOr , "|=" ,BitOr ,Bin ,fnopBin|fnopAsg)

case knopAsgOr:

Indent(indentAmt);

Output::Print(L"|=\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopAsgLsh , "<<=" ,Lsh ,Bin ,fnopBin|fnopAsg)

case knopAsgLsh:

Indent(indentAmt);

Output::Print(L"<<=\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopAsgRsh , ">>=" ,Rsh ,Bin ,fnopBin|fnopAsg)

case knopAsgRsh:

Indent(indentAmt);

Output::Print(L">>=\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopAsgRs2 , ">>>=" ,Rs2 ,Bin ,fnopBin|fnopAsg)

case knopAsgRs2:

Indent(indentAmt);

Output::Print(L">>>=\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

case knopComputedName:

Indent(indentAmt);

Output::Print(L"ComputedProperty\n");

PrintPnodeWIndent(pnode->sxUni.pnode1, indentAmt + INDENT\_SIZE);

break;

//PTNODE(knopMember , ":" ,None ,Bin ,fnopBin)

case knopMember:

case knopMemberShort:

case knopObjectPatternMember:

Indent(indentAmt);

Output::Print(L":\n");

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxBin.pnode2,indentAmt+INDENT\_SIZE);

break;

// General nodes.

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

case knopList:

Indent(indentAmt);

Output::Print(L"List\n");

PrintPnodeListWIndent(pnode,indentAmt+INDENT\_SIZE);

break;

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

case knopVarDecl:

Indent(indentAmt);

Output::Print(L"var %s\n",pnode->sxVar.pid->Psz());

if (pnode->sxVar.pnodeInit!=NULL)

PrintPnodeWIndent(pnode->sxVar.pnodeInit,indentAmt+INDENT\_SIZE);

break;

case knopConstDecl:

Indent(indentAmt);

Output::Print(L"const %s\n",pnode->sxVar.pid->Psz());

if (pnode->sxVar.pnodeInit!=NULL)

PrintPnodeWIndent(pnode->sxVar.pnodeInit,indentAmt+INDENT\_SIZE);

break;

case knopLetDecl:

Indent(indentAmt);

Output::Print(L"let %s\n",pnode->sxVar.pid->Psz());

if (pnode->sxVar.pnodeInit!=NULL)

PrintPnodeWIndent(pnode->sxVar.pnodeInit,indentAmt+INDENT\_SIZE);

break;

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

case knopFncDecl:

Indent(indentAmt);

if (pnode->sxFnc.pid!=NULL)

{

Output::Print(L"fn decl %d nested %d name %s (%d-%d)\n",pnode->sxFnc.IsDeclaration(),pnode->sxFnc.IsNested(),

pnode->sxFnc.pid->Psz(), pnode->ichMin, pnode->ichLim);

}

else

{

Output::Print(L"fn decl %d nested %d anonymous (%d-%d)\n",pnode->sxFnc.IsDeclaration(),pnode->sxFnc.IsNested(),pnode->ichMin,pnode->ichLim);

}

PrintScopesWIndent(pnode, indentAmt+INDENT\_SIZE);

PrintFormalsWIndent(pnode->sxFnc.pnodeArgs, indentAmt + INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxFnc.pnodeRest, indentAmt + INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxFnc.pnodeBody, indentAmt + INDENT\_SIZE);

break;

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

case knopProg:

Indent(indentAmt);

Output::Print(L"program\n");

PrintScopesWIndent(pnode, indentAmt+INDENT\_SIZE);

PrintPnodeListWIndent(pnode->sxFnc.pnodeBody,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopEndCode , "<endcode>" ,None ,None ,fnopNone)

case knopEndCode:

Indent(indentAmt);

Output::Print(L"<endcode>\n");

break;

//PTNODE(knopDebugger , "debugger" ,None ,None ,fnopNone)

case knopDebugger:

Indent(indentAmt);

Output::Print(L"<debugger>\n");

break;

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

case knopFor:

Indent(indentAmt);

Output::Print(L"for\n");

PrintScopesWIndent(pnode, indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxFor.pnodeInit,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxFor.pnodeCond,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxFor.pnodeIncr,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxFor.pnodeBody,indentAmt+INDENT\_SIZE);

break;

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

case knopIf:

Indent(indentAmt);

Output::Print(L"if\n");

PrintPnodeWIndent(pnode->sxIf.pnodeCond,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxIf.pnodeTrue,indentAmt+INDENT\_SIZE);

if (pnode->sxIf.pnodeFalse!=NULL)

PrintPnodeWIndent(pnode->sxIf.pnodeFalse,indentAmt+INDENT\_SIZE);

break;

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

case knopWhile:

Indent(indentAmt);

Output::Print(L"while\n");

PrintPnodeWIndent(pnode->sxWhile.pnodeCond,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxWhile.pnodeBody,indentAmt+INDENT\_SIZE);

break;

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

case knopDoWhile:

Indent(indentAmt);

Output::Print(L"do\n");

PrintPnodeWIndent(pnode->sxWhile.pnodeCond,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxWhile.pnodeBody,indentAmt+INDENT\_SIZE);

break;

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

case knopForIn:

Indent(indentAmt);

Output::Print(L"forIn\n");

PrintScopesWIndent(pnode, indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxForInOrForOf.pnodeLval,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxForInOrForOf.pnodeObj,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxForInOrForOf.pnodeBody,indentAmt+INDENT\_SIZE);

break;

case knopForOf:

Indent(indentAmt);

Output::Print(L"forOf\n");

PrintScopesWIndent(pnode, indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxForInOrForOf.pnodeLval,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxForInOrForOf.pnodeObj,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxForInOrForOf.pnodeBody,indentAmt+INDENT\_SIZE);

break;

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

case knopReturn:

Indent(indentAmt);

Output::Print(L"return\n");

if (pnode->sxReturn.pnodeExpr!=NULL)

PrintPnodeWIndent(pnode->sxReturn.pnodeExpr,indentAmt+INDENT\_SIZE);

break;

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

case knopBlock:

Indent(indentAmt);

Output::Print(L"block ");

if (pnode->grfpn & fpnSyntheticNode)

Output::Print(L"synthetic ");

Output::Print(L"(%d-%d)\n",pnode->ichMin,pnode->ichLim);

PrintScopesWIndent(pnode, indentAmt+INDENT\_SIZE);

if (pnode->sxBlock.pnodeStmt!=NULL)

PrintPnodeWIndent(pnode->sxBlock.pnodeStmt,indentAmt+INDENT\_SIZE);

break;

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

case knopWith:

Indent(indentAmt);

Output::Print(L"with (%d-%d)\n", pnode->ichMin,pnode->ichLim);

PrintScopesWIndent(pnode, indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxWith.pnodeObj,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxWith.pnodeBody,indentAmt+INDENT\_SIZE);

break;

//PTNODE(knopBreak , "break" ,None ,Jump ,fnopNone)

case knopBreak:

Indent(indentAmt);

Output::Print(L"break\n");

// TODO: some representation of target

break;

//PTNODE(knopContinue , "continue" ,None ,Jump ,fnopNone)

case knopContinue:

Indent(indentAmt);

Output::Print(L"continue\n");

// TODO: some representation of target

break;

//PTNODE(knopLabel , "label" ,None ,Label,fnopNone)

case knopLabel:

Indent(indentAmt);

Output::Print(L"label %s",pnode->sxLabel.pid->Psz());

// TODO: print labeled statement

break;

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

case knopSwitch:

Indent(indentAmt);

Output::Print(L"switch\n");

PrintScopesWIndent(pnode, indentAmt+INDENT\_SIZE);

for (ParseNode \*pnodeT = pnode->sxSwitch.pnodeCases; NULL != pnodeT;pnodeT = pnodeT->sxCase.pnodeNext) {

PrintPnodeWIndent(pnodeT,indentAmt+2);

}

break;

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

case knopCase:

Indent(indentAmt);

Output::Print(L"case\n");

PrintPnodeWIndent(pnode->sxCase.pnodeExpr,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxCase.pnodeBody,indentAmt+INDENT\_SIZE);

break;

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

case knopTryFinally:

PrintPnodeWIndent(pnode->sxTryFinally.pnodeTry,indentAmt);

PrintPnodeWIndent(pnode->sxTryFinally.pnodeFinally,indentAmt);

break;

case knopFinally:

Indent(indentAmt);

Output::Print(L"finally\n");

PrintPnodeWIndent(pnode->sxFinally.pnodeBody,indentAmt+INDENT\_SIZE);

break;

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

case knopCatch:

Indent(indentAmt);

Output::Print(L"catch (%d-%d)\n", pnode->ichMin,pnode->ichLim);

PrintScopesWIndent(pnode, indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxCatch.pnodeParam,indentAmt+INDENT\_SIZE);

// if (pnode->sxCatch.pnodeGuard!=NULL)

// PrintPnodeWIndent(pnode->sxCatch.pnodeGuard,indentAmt+INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxCatch.pnodeBody,indentAmt+INDENT\_SIZE);

break;

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

case knopTryCatch:

PrintPnodeWIndent(pnode->sxTryCatch.pnodeTry,indentAmt);

PrintPnodeWIndent(pnode->sxTryCatch.pnodeCatch,indentAmt);

break;

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

case knopTry:

Indent(indentAmt);

Output::Print(L"try\n");

PrintPnodeWIndent(pnode->sxTry.pnodeBody,indentAmt+INDENT\_SIZE);

break;

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

case knopThrow:

Indent(indentAmt);

Output::Print(L"throw\n");

PrintPnodeWIndent(pnode->sxUni.pnode1,indentAmt+INDENT\_SIZE);

break;

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

case knopClassDecl:

Indent(indentAmt);

Output::Print(L"class %s", pnode->sxClass.pnodeName->sxVar.pid->Psz());

if (pnode->sxClass.pnodeExtends != nullptr)

{

Output::Print(L" extends ");

PrintPnodeWIndent(pnode->sxClass.pnodeExtends, 0);

}

else {

Output::Print(L"\n");

}

PrintPnodeWIndent(pnode->sxClass.pnodeConstructor, indentAmt + INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxClass.pnodeMembers, indentAmt + INDENT\_SIZE);

PrintPnodeWIndent(pnode->sxClass.pnodeStaticMembers, indentAmt + INDENT\_SIZE);

break;

case knopStrTemplate:

Indent(indentAmt);

Output::Print(L"string template\n");

PrintPnodeListWIndent(pnode->sxStrTemplate.pnodeSubstitutionExpressions, indentAmt + INDENT\_SIZE);

break;

case knopYieldStar:

Indent(indentAmt);

Output::Print(L"yield\*\n");

PrintPnodeListWIndent(pnode->sxUni.pnode1, indentAmt + INDENT\_SIZE);

break;

case knopYield:

case knopYieldLeaf:

Indent(indentAmt);

Output::Print(L"yield\n");

PrintPnodeListWIndent(pnode->sxUni.pnode1, indentAmt + INDENT\_SIZE);

break;

case knopAwait:

Indent(indentAmt);

Output::Print(L"await\n");

PrintPnodeListWIndent(pnode->sxUni.pnode1, indentAmt + INDENT\_SIZE);

break;

default:

Output::Print(L"unhandled pnode op %d\n",pnode->nop);

break;

}

}

void PrintPnodeListWIndent(ParseNode \*pnode,int indentAmt) {

if (pnode!=NULL) {

while(pnode->nop==knopList) {

PrintPnodeWIndent(pnode->sxBin.pnode1,indentAmt);

pnode = pnode->sxBin.pnode2;

}

PrintPnodeWIndent(pnode,indentAmt);

}

}

void PrintFormalsWIndent(ParseNode \*pnodeArgs, int indentAmt)

{

for (ParseNode \*pnode = pnodeArgs; pnode != nullptr; pnode = pnode->GetFormalNext())

{

PrintPnodeWIndent(pnode->nop == knopParamPattern ? pnode->sxParamPattern.pnode1 : pnode, indentAmt);

}

}

void PrintPnode(ParseNode \*pnode) {

PrintPnodeWIndent(pnode,0);

}

void ParseNode::Dump()

{

switch(nop)

{

case knopFncDecl:

case knopProg:

LPCOLESTR name = Js::Constants::AnonymousFunction;

if(this->sxFnc.pnodeName)

{

name = this->sxFnc.pnodeName->sxVar.pid->Psz();

}

Output::Print(L"%s (%d) [%d, %d]:\n", name, this->sxFnc.functionId, this->sxFnc.lineNumber, this->sxFnc.columnNumber);

Output::Print(L"hasArguments: %s callsEval:%s childCallsEval:%s HasReferenceableBuiltInArguments:%s ArgumentsObjectEscapes:%s HasWith:%s HasThis:%s HasOnlyThis:%s \n",

IsTrueOrFalse(this->sxFnc.HasHeapArguments()),

IsTrueOrFalse(this->sxFnc.CallsEval()),

IsTrueOrFalse(this->sxFnc.ChildCallsEval()),

IsTrueOrFalse(this->sxFnc.HasReferenceableBuiltInArguments()),

IsTrueOrFalse(this->sxFnc.GetArgumentsObjectEscapes()),

IsTrueOrFalse(this->sxFnc.HasWithStmt()),

IsTrueOrFalse(this->sxFnc.HasThisStmt()),

IsTrueOrFalse(this->sxFnc.HasOnlyThisStmts()));

if(this->sxFnc.funcInfo)

{

this->sxFnc.funcInfo->Dump();

}

break;

}

}

#endif

DeferredFunctionStub \* BuildDeferredStubTree(ParseNode \*pnodeFnc, Recycler \*recycler)

{

Assert(pnodeFnc->nop == knopFncDecl);

uint nestedCount = pnodeFnc->sxFnc.nestedCount;

if (nestedCount == 0)

{

return nullptr;

}

if (pnodeFnc->sxFnc.deferredStub)

{

return pnodeFnc->sxFnc.deferredStub;

}

DeferredFunctionStub \*deferredStubs = RecyclerNewArray(recycler, DeferredFunctionStub, nestedCount);

uint i = 0;

ParseNode \*pnodeBlock = pnodeFnc->sxFnc.pnodeBodyScope;

Assert(pnodeBlock != nullptr

&& pnodeBlock->nop == knopBlock

&& (pnodeBlock->sxBlock.blockType == PnodeBlockType::Function

|| pnodeBlock->sxBlock.blockType == PnodeBlockType::Parameter));

for (ParseNode \*pnodeChild = pnodeBlock->sxBlock.pnodeScopes; pnodeChild != nullptr;)

{

if (pnodeChild->nop != knopFncDecl)

{

// We only expect to find a function body block in a parameter scope block.

Assert(pnodeChild->nop == knopBlock

&& (pnodeBlock->sxBlock.blockType == PnodeBlockType::Parameter

|| pnodeChild->sxBlock.blockType == PnodeBlockType::Function));

pnodeChild = pnodeChild->sxBlock.pnodeNext;

continue;

}

Assert(i < nestedCount);

if (pnodeChild->sxFnc.IsGeneratedDefault())

{

++i;

pnodeChild = pnodeChild->sxFnc.pnodeNext;

continue;

}

\_\_analysis\_assume(i < nestedCount);

deferredStubs[i].fncFlags = pnodeChild->sxFnc.fncFlags;

deferredStubs[i].nestedCount = pnodeChild->sxFnc.nestedCount;

deferredStubs[i].restorePoint = \*pnodeChild->sxFnc.pRestorePoint;

deferredStubs[i].deferredStubs = BuildDeferredStubTree(pnodeChild, recycler);

#if DEBUG

deferredStubs[i].ichMin = pnodeChild->ichMin;

#endif

++i;

pnodeChild = pnodeChild->sxFnc.pnodeNext;

}

return deferredStubs;

}