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

// 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

namespace Js

{

class DelayLoadWindowsGlobalization;

}

#include "Windows.Globalization.h"

int CountNewlines(LPCOLESTR psz, int cch = -1);

class Parser;

struct ParseContext;

struct Token

{

private:

union

{

struct

{

IdentPtr pid;

const char \* pchMin;

long length;

};

long lw;

struct

{

double dbl;

// maybeInt will be true if the number did not contain 'e', 'E' , or '.'

// notably important in asm.js where the '.' has semantic importance

bool maybeInt;

};

UnifiedRegex::RegexPattern\* pattern;

struct

{

charcount\_t ichMin;

charcount\_t ichLim;

};

} u;

IdentPtr CreateIdentifier(HashTbl \* hashTbl);

public:

Token() : tk(tkLim) {}

tokens tk;

BOOL IsIdentifier() const

{

return tk == tkID;

}

IdentPtr GetStr() const

{

Assert(tk == tkStrCon || tk == tkStrTmplBasic || tk == tkStrTmplBegin || tk == tkStrTmplMid || tk == tkStrTmplEnd);

return u.pid;

}

IdentPtr GetIdentifier(HashTbl \* hashTbl)

{

Assert(IsIdentifier() || IsReservedWord());

if (u.pid)

{

return u.pid;

}

return CreateIdentifier(hashTbl);

}

long GetLong() const

{

Assert(tk == tkIntCon);

return u.lw;

}

double GetDouble() const

{

Assert(tk == tkFltCon);

return u.dbl;

}

bool GetDoubleMayBeInt() const

{

Assert(tk == tkFltCon);

return u.maybeInt;

}

UnifiedRegex::RegexPattern \* GetRegex()

{

Assert(tk == tkRegExp);

return u.pattern;

}

// NOTE: THESE ROUTINES DEPEND ON THE ORDER THAT OPERATORS

// ARE DECLARED IN kwd-xxx.h FILES.

BOOL IsReservedWord() const

{

// Keywords and future reserved words (does not include operators)

return tk < tkID;

}

BOOL IsKeyword() const;

BOOL IsFutureReservedWord(const BOOL isStrictMode) const

{

// Reserved words that are not keywords

return tk >= tkENUM && tk <= (isStrictMode ? tkSTATIC : tkENUM);

}

BOOL IsOperator() const

{

return tk >= tkComma && tk < tkLParen;

}

// UTF16 Scanner are only for syntax coloring. Only support

// defer pid creation for UTF8

void SetIdentifier(const char \* pchMin, long len)

{

this->u.pid = nullptr;

this->u.pchMin = pchMin;

this->u.length = len;

}

void SetIdentifier(IdentPtr pid)

{

this->u.pid = pid;

this->u.pchMin = nullptr;

}

void SetLong(long value)

{

this->u.lw = value;

}

void SetDouble(double dbl, bool maybeInt)

{

this->u.dbl = dbl;

this->u.maybeInt = maybeInt;

}

tokens SetRegex(UnifiedRegex::RegexPattern \*const pattern, Parser \*const parser);

};

typedef BYTE UTF8Char;

typedef UTF8Char\* UTF8CharPtr;

class NullTerminatedUnicodeEncodingPolicy

{

public:

typedef OLECHAR EncodedChar;

typedef const OLECHAR \*EncodedCharPtr;

protected:

static const bool MultiUnitEncoding = false;

static const size\_t m\_cMultiUnits = 0;

static BOOL IsMultiUnitChar(OLECHAR ch) { return FALSE; }

// See comment below regarding unused 'last' parameter

static OLECHAR ReadFirst(EncodedCharPtr &p, EncodedCharPtr last) { return \*p++; }

template <bool bScan>

static OLECHAR ReadRest(OLECHAR ch, EncodedCharPtr &p, EncodedCharPtr last) { return ch; }

template <bool bScan>

static OLECHAR ReadFull(EncodedCharPtr &p, EncodedCharPtr last) { return \*p++; }

static OLECHAR PeekFirst(EncodedCharPtr p, EncodedCharPtr last) { return \*p; }

static OLECHAR PeekFull(EncodedCharPtr p, EncodedCharPtr last) { return \*p; }

static void RestoreMultiUnits(size\_t multiUnits) { }

static size\_t CharacterOffsetToUnitOffset(EncodedCharPtr start, EncodedCharPtr current, EncodedCharPtr last, charcount\_t offset) { return offset; }

static void ConvertToUnicode(\_\_out\_ecount\_full(cch) LPOLESTR pch, charcount\_t cch, EncodedCharPtr pu)

{

js\_memcpy\_s(pch, cch \* sizeof(OLECHAR), pu, cch \* sizeof(OLECHAR));

}

public:

void FromExternalSource() { }

bool IsFromExternalSource() { return false; }

};

template <bool nullTerminated>

class UTF8EncodingPolicyBase

{

public:

typedef utf8char\_t EncodedChar;

typedef LPCUTF8 EncodedCharPtr;

protected:

static const bool MultiUnitEncoding = true;

size\_t m\_cMultiUnits;

utf8::DecodeOptions m\_decodeOptions;

UTF8EncodingPolicyBase(): m\_cMultiUnits(0), m\_decodeOptions(utf8::doAllowThreeByteSurrogates) { }

static BOOL IsMultiUnitChar(OLECHAR ch) { return ch > 0x7f; }

// Note when nullTerminated is false we still need to increment the character pointer because the scanner "puts back" this virtual null character by decrementing the pointer

static OLECHAR ReadFirst(EncodedCharPtr &p, EncodedCharPtr last) { return (nullTerminated || p < last) ? static\_cast< OLECHAR >(\*p++) : (p++, 0); }

// "bScan" indicates if this ReadFull is part of scanning. Pass true during scanning and ReadFull will update

// related Scanner state. The caller is supposed to sync result "p" to Scanner's current position. Pass false

// otherwise and this doesn't affect Scanner state.

template <bool bScan>

OLECHAR ReadFull(EncodedCharPtr &p, EncodedCharPtr last)

{

EncodedChar ch = (nullTerminated || p < last) ? \*p++ : (p++, 0);

return !IsMultiUnitChar(ch) ? static\_cast< OLECHAR >(ch) : ReadRest<bScan>(ch, p, last);

}

static OLECHAR PeekFirst(EncodedCharPtr p, EncodedCharPtr last) { return (nullTerminated || p < last) ? static\_cast< OLECHAR >(\*p) : 0; }

OLECHAR PeekFull(EncodedCharPtr p, EncodedCharPtr last)

{

OLECHAR result = PeekFirst(p, last);

if (IsMultiUnitChar(result))

{

result = ReadFull<false>(p, last);

}

return result;

}

// "bScan" indicates if this ReadRest is part of scanning. Pass true during scanning and ReadRest will update

// related Scanner state. The caller is supposed to sync result "p" to Scanner's current position. Pass false

// otherwise and this doesn't affect Scanner state.

template <bool bScan>

OLECHAR ReadRest(OLECHAR ch, EncodedCharPtr &p, EncodedCharPtr last)

{

EncodedCharPtr s;

if (bScan)

{

s = p;

}

OLECHAR result = utf8::DecodeTail(ch, p, last, m\_decodeOptions);

if (bScan)

{

// If we are scanning, update m\_cMultiUnits counter.

m\_cMultiUnits += p - s;

}

return result;

}

void RestoreMultiUnits(size\_t multiUnits) { m\_cMultiUnits = multiUnits; }

size\_t CharacterOffsetToUnitOffset(EncodedCharPtr start, EncodedCharPtr current, EncodedCharPtr last, charcount\_t offset)

{

// Note: current may be before or after last. If last is the null terminator, current should be within [start, last].

// But if we excluded HTMLCommentSuffix for the source, last is before "// -->\0". Scanner may stop at null

// terminator past last, then current is after last.

Assert(current >= start);

size\_t currentUnitOffset = current - start;

Assert(currentUnitOffset > m\_cMultiUnits);

Assert(currentUnitOffset - m\_cMultiUnits < LONG\_MAX);

charcount\_t currentCharacterOffset = charcount\_t(currentUnitOffset - m\_cMultiUnits);

// If the offset is the current character offset then just return the current unit offset.

if (currentCharacterOffset == offset) return currentUnitOffset;

// If we have not encountered any multi-unit characters and we are moving backward the

// character index and unit index are 1:1 so just return offset

if (m\_cMultiUnits == 0 && offset <= currentCharacterOffset) return offset;

// Use local decode options

utf8::DecodeOptions decodeOptions = IsFromExternalSource() ? utf8::doDefault : utf8::doAllowThreeByteSurrogates;

if (offset > currentCharacterOffset)

{

// If we are looking for an offset past current, current must be within [start, last]. We don't expect seeking

// scanner position past last.

Assert(current <= last);

// If offset > currentOffset we already know the current character offset. The unit offset is the

// unit index of offset - currentOffset characters from current.

charcount\_t charsLeft = offset - currentCharacterOffset;

return currentUnitOffset + utf8::CharacterIndexToByteIndex(current, last - current, charsLeft, decodeOptions);

}

// If all else fails calculate the index from the start of the buffer.

return utf8::CharacterIndexToByteIndex(start, currentUnitOffset, offset, decodeOptions);

}

void ConvertToUnicode(\_\_out\_ecount\_full(cch) LPOLESTR pch, charcount\_t cch, EncodedCharPtr pu)

{

m\_decodeOptions = (utf8::DecodeOptions)(m\_decodeOptions & ~utf8::doSecondSurrogatePair);

utf8::DecodeInto(pch, pu, cch, m\_decodeOptions);

}

public:

// If we get UTF8 source buffer, turn off doAllowThreeByteSurrogates but allow invalid WCHARs without replacing them with replacement 'g\_chUnknown'.

void FromExternalSource() { m\_decodeOptions = (utf8::DecodeOptions)(m\_decodeOptions & ~utf8::doAllowThreeByteSurrogates | utf8::doAllowInvalidWCHARs); }

bool IsFromExternalSource() { return (m\_decodeOptions & utf8::doAllowThreeByteSurrogates) == 0; }

};

typedef UTF8EncodingPolicyBase<true> NullTerminatedUTF8EncodingPolicy;

typedef UTF8EncodingPolicyBase<false> NotNullTerminatedUTF8EncodingPolicy;

interface IScanner

{

virtual void GetErrorLineInfo(\_\_out long& ichMin, \_\_out long& ichLim, \_\_out long& line, \_\_out long& ichMinLine) = 0;

virtual HRESULT SysAllocErrorLine(long ichMinLine, \_\_out BSTR\* pbstrLine) = 0;

};

// Flags that can be provided to the Scan functions.

// These can be bitwise OR'ed.

enum ScanFlag

{

ScanFlagNone = 0,

ScanFlagSuppressStrPid = 1, // Force strings to always have pid

ScanFlagSuppressIdPid = 2 // Force identifiers to always have pid

};

typedef HRESULT (\*CommentCallback)(void \*data, OLECHAR firstChar, OLECHAR secondChar, bool containTypeDef, charcount\_t min, charcount\_t lim, bool adjacent, bool multiline, charcount\_t startLine, charcount\_t endLine);

// Restore point defined using a relative offset rather than a pointer.

struct RestorePoint

{

charcount\_t m\_ichMinTok;

charcount\_t m\_ichMinLine;

size\_t m\_cMinTokMultiUnits;

size\_t m\_cMinLineMultiUnits;

charcount\_t m\_line;

uint functionIdIncrement;

size\_t lengthDecr;

BOOL m\_fHadEol;

#ifdef DEBUG

size\_t m\_cMultiUnits;

#endif

RestorePoint()

: m\_ichMinTok((charcount\_t)-1),

m\_ichMinLine((charcount\_t)-1),

m\_cMinTokMultiUnits((size\_t)-1),

m\_cMinLineMultiUnits((size\_t)-1),

m\_line((charcount\_t)-1),

functionIdIncrement(0),

lengthDecr(0),

m\_fHadEol(FALSE)

#ifdef DEBUG

, m\_cMultiUnits((size\_t)-1)

#endif

{

};

};

template <typename EncodingPolicy>

class Scanner : public IScanner, public EncodingPolicy

{

friend Parser;

typedef typename EncodingPolicy::EncodedChar EncodedChar;

typedef typename EncodingPolicy::EncodedCharPtr EncodedCharPtr;

public:

static Scanner \* Create(Parser\* parser, HashTbl \*phtbl, Token \*ptoken, ErrHandler \*perr, Js::ScriptContext \*scriptContext)

{

return HeapNewNoThrow(Scanner, parser, phtbl, ptoken, perr, scriptContext);

}

void Release(void)

{

delete this;

}

tokens Scan();

tokens ScanNoKeywords();

tokens ScanForcingPid();

void SetText(EncodedCharPtr psz, size\_t offset, size\_t length, charcount\_t characterOffset, ULONG grfscr, ULONG lineNumber = 0);

void PrepareForBackgroundParse(Js::ScriptContext \*scriptContext);

enum ScanState

{

ScanStateNormal = 0,

ScanStateMultiLineComment = 1,

ScanStateMultiLineSingleQuoteString = 2,

ScanStateMultiLineDoubleQuoteString = 3,

ScanStateStringTemplateMiddleOrEnd = 4,

};

ScanState GetScanState() { return m\_scanState; }

void SetScanState(ScanState state) { m\_scanState = state; }

bool SetYieldIsKeyword(bool fYieldIsKeyword)

{

bool fPrevYieldIsKeyword = m\_fYieldIsKeyword;

m\_fYieldIsKeyword = fYieldIsKeyword;

return fPrevYieldIsKeyword;

}

bool YieldIsKeyword()

{

return m\_fYieldIsKeyword;

}

bool SetAwaitIsKeyword(bool fAwaitIsKeyword)

{

bool fPrevAwaitIsKeyword = m\_fAwaitIsKeyword;

m\_fAwaitIsKeyword = fAwaitIsKeyword;

return fPrevAwaitIsKeyword;

}

bool AwaitIsKeyword()

{

return m\_fAwaitIsKeyword;

}

tokens TryRescanRegExp();

tokens RescanRegExp();

tokens RescanRegExpNoAST();

tokens RescanRegExpTokenizer();

BOOL FHadNewLine(void)

{

return m\_fHadEol;

}

IdentPtr PidFromLong(long lw);

IdentPtr PidFromDbl(double dbl);

LPCOLESTR StringFromLong(long lw);

LPCOLESTR StringFromDbl(double dbl);

IdentPtr GetSecondaryBufferAsPid();

bool BindDeferredPidRefs() const

{

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

}

BYTE SetDeferredParse(BOOL defer)

{

BYTE fOld = m\_DeferredParseFlags;

if (defer)

{

m\_DeferredParseFlags |= ScanFlagSuppressStrPid;

if (!this->BindDeferredPidRefs())

{

m\_DeferredParseFlags |= ScanFlagSuppressIdPid;

}

}

else

{

m\_DeferredParseFlags = ScanFlagNone;

}

return fOld;

}

void SetDeferredParseFlags(BYTE flags)

{

m\_DeferredParseFlags = flags;

}

// the functions IsDoubleQuoteOnLastTkStrCon() and IsHexOrOctOnLastTKNumber() works only with a scanner without lookahead

// Both functions are used to get more info on the last token for specific diffs necessary for JSON parsing.

//Single quotes are not legal in JSON strings. Make distinction between single quote string constant and single quote string

BOOL IsDoubleQuoteOnLastTkStrCon()

{

return m\_doubleQuoteOnLastTkStrCon;

}

// True if all chars of last string constant are ascii

BOOL IsEscapeOnLastTkStrCon()

{

return m\_EscapeOnLastTkStrCon;

}

bool IsOctOrLeadingZeroOnLastTKNumber()

{

return m\_OctOrLeadingZeroOnLastTKNumber;

}

// Returns the character offset of the first token. The character offset is the offset the first character of the token would

// have if the entire file was converted to Unicode (UTF16-LE).

charcount\_t IchMinTok(void) const

{

Assert(m\_pchMinTok - m\_pchBase >= 0);

Assert(m\_pchMinTok - m\_pchBase <= LONG\_MAX);

return static\_cast< charcount\_t >(m\_pchMinTok - m\_pchBase - m\_cMinTokMultiUnits);

}

// Returns the character offset of the character immediately following the token. The character offset is the offset the first

// character of the token would have if the entire file was converted to Unicode (UTF16-LE).

charcount\_t IchLimTok(void) const

{

Assert(m\_currentCharacter - m\_pchBase >= 0);

Assert(m\_currentCharacter - m\_pchBase <= LONG\_MAX);

return static\_cast< charcount\_t >(m\_currentCharacter - m\_pchBase - m\_cMultiUnits);

}

void SetErrorPosition(charcount\_t ichMinError, charcount\_t ichLimError)

{

Assert(ichLimError > 0 || ichMinError == 0);

m\_ichMinError = ichMinError;

m\_ichLimError = ichLimError;

}

charcount\_t IchMinError(void) const

{

return m\_ichLimError ? m\_ichMinError : IchMinTok();

}

charcount\_t IchLimError(void) const

{

return m\_ichLimError ? m\_ichLimError : IchLimTok();

}

// Returns the encoded unit offset of first character of the token. For example, in a UTF-8 encoding this is the offset into

// the UTF-8 buffer. In Unicode this is the same as IchMinTok().

size\_t IecpMinTok(void) const

{

return static\_cast< size\_t >(m\_pchMinTok - m\_pchBase);

}

// Returns the encoded unit offset of the character immediately following the token. For example, in a UTF-8 encoding this is

// the offset into the UTF-8 buffer. In Unicode this is the same as IchLimTok().

size\_t IecpLimTok(void) const

{

return static\_cast< size\_t >(m\_currentCharacter - m\_pchBase);

}

size\_t IecpLimTokPrevious() const

{

AssertMsg(m\_iecpLimTokPrevious != (size\_t)-1, "IecpLimTokPrevious() cannot be called before scanning a token");

return m\_iecpLimTokPrevious;

}

IdentPtr PidAt(size\_t iecpMin, size\_t iecpLim);

// Returns the character offset within the stream of the first character on the current line.

charcount\_t IchMinLine(void) const

{

Assert(m\_pchMinLine - m\_pchBase >= 0);

Assert(m\_pchMinLine - m\_pchBase <= LONG\_MAX);

return static\_cast<charcount\_t>(m\_pchMinLine - m\_pchBase - m\_cMinLineMultiUnits);

}

// Returns the current line number

charcount\_t LineCur(void) { return m\_line; }

tokens ErrorToken() { return m\_errorToken; }

void SetCurrentCharacter(charcount\_t offset, ULONG lineNumber = 0)

{

DebugOnly(m\_iecpLimTokPrevious = (size\_t)-1);

size\_t length = m\_pchLast - m\_pchBase;

if (offset > length) offset = static\_cast< charcount\_t >(length);

size\_t ibOffset = CharacterOffsetToUnitOffset(m\_pchBase, m\_currentCharacter, m\_pchLast, offset);

m\_currentCharacter = m\_pchBase + ibOffset;

Assert(ibOffset >= offset);

RestoreMultiUnits(ibOffset - offset);

m\_line = lineNumber;

}

// IScanner methods

virtual void GetErrorLineInfo(\_\_out long& ichMin, \_\_out long& ichLim, \_\_out long& line, \_\_out long& ichMinLine)

{

ichMin = this->IchMinError();

ichLim = this->IchLimError();

line = this->LineCur();

ichMinLine = this->IchMinLine();

if (m\_ichLimError && m\_ichMinError < (charcount\_t)ichMinLine)

{

line = m\_startLine;

ichMinLine = UpdateLine(line, m\_pchStartLine, m\_pchLast, 0, ichMin);

}

}

virtual HRESULT SysAllocErrorLine(long ichMinLine, \_\_out BSTR\* pbstrLine);

charcount\_t UpdateLine(long &line, EncodedCharPtr start, EncodedCharPtr last, charcount\_t ichStart, charcount\_t ichEnd);

class TemporaryBuffer

{

friend Scanner<EncodingPolicy>;

private:

// Keep a reference to the scanner.

// We will use it to signal an error if we fail to allocate the buffer.

Scanner<EncodingPolicy>\* m\_pscanner;

ulong m\_cchMax;

ulong m\_ichCur;

\_\_field\_ecount(m\_cchMax) OLECHAR \*m\_prgch;

byte m\_rgbInit[256];

public:

TemporaryBuffer()

{

m\_pscanner = nullptr;

m\_prgch = (OLECHAR\*)m\_rgbInit;

m\_cchMax = \_countof(m\_rgbInit) / sizeof(OLECHAR);

m\_ichCur = 0;

}

~TemporaryBuffer()

{

if (m\_prgch != (OLECHAR\*)m\_rgbInit)

{

free(m\_prgch);

}

}

void Init()

{

m\_ichCur = 0;

}

void AppendCh(uint ch)

{

return AppendCh<true>(ch);

}

template<bool performAppend> void AppendCh(uint ch)

{

if (performAppend)

{

if (m\_ichCur >= m\_cchMax)

{

Grow();

}

Assert(m\_ichCur < m\_cchMax);

\_\_analysis\_assume(m\_ichCur < m\_cchMax);

m\_prgch[m\_ichCur++] = static\_cast<OLECHAR>(ch);

}

}

void Grow()

{

Assert(m\_pscanner != nullptr);

byte \*prgbNew;

byte \*prgbOld = (byte \*)m\_prgch;

unsigned long cbNew;

if (FAILED(ULongMult(m\_cchMax, sizeof(OLECHAR) \* 2, &cbNew)))

{

m\_pscanner->Error(ERRnoMemory);

}

if (prgbOld == m\_rgbInit)

{

if (nullptr == (prgbNew = static\_cast<byte\*>(malloc(cbNew))))

m\_pscanner->Error(ERRnoMemory);

js\_memcpy\_s(prgbNew, cbNew, prgbOld, m\_ichCur \* sizeof(OLECHAR));

}

else if (nullptr == (prgbNew = static\_cast<byte\*>(realloc(prgbOld, cbNew))))

{

m\_pscanner->Error(ERRnoMemory);

}

m\_prgch = (OLECHAR\*)prgbNew;

m\_cchMax = cbNew / sizeof(OLECHAR);

}

};

void Capture(\_Out\_ RestorePoint\* restorePoint);

void SeekTo(const RestorePoint& restorePoint);

void Capture(\_Out\_ RestorePoint\* restorePoint, uint functionIdIncrement, size\_t lengthDecr);

void SeekTo(const RestorePoint& restorePoint, uint \*nextFunctionId);

void SetNextStringTemplateIsTagged(BOOL value)

{

this->m\_fNextStringTemplateIsTagged = value;

}

private:

Parser \*m\_parser;

HashTbl \*m\_phtbl;

Token \*m\_ptoken;

EncodedCharPtr m\_pchBase; // beginning of source

EncodedCharPtr m\_pchLast; // The end of source

EncodedCharPtr m\_pchMinLine; // beginning of current line

EncodedCharPtr m\_pchMinTok; // beginning of current token

EncodedCharPtr m\_currentCharacter; // current character

EncodedCharPtr m\_pchPrevLine; // beginning of previous line

size\_t m\_cMinTokMultiUnits; // number of multi-unit characters previous to m\_pchMinTok

size\_t m\_cMinLineMultiUnits; // number of multi-unit characters previous to m\_pchMinLine

ErrHandler \*m\_perr; // error handler to use

uint16 m\_fStringTemplateDepth; // we should treat } as string template middle starting character (depth instead of flag)

BOOL m\_fHadEol;

BOOL m\_fHtmlComments : 1;

BOOL m\_doubleQuoteOnLastTkStrCon :1;

bool m\_OctOrLeadingZeroOnLastTKNumber :1;

BOOL m\_fSyntaxColor : 1; // whether we're just syntax coloring

BOOL m\_EscapeOnLastTkStrCon:1;

BOOL m\_fNextStringTemplateIsTagged:1; // the next string template scanned has a tag (must create raw strings)

BYTE m\_DeferredParseFlags:2; // suppressStrPid and suppressIdPid

charcount\_t m\_ichCheck; // character at which completion is to be computed.

bool es6UnicodeMode; // True if ES6Unicode Extensions are enabled.

bool m\_fYieldIsKeyword; // Whether to treat 'yield' as an identifier or keyword

bool m\_fAwaitIsKeyword; // Whether to treat 'await' as an identifier or keyword

// Temporary buffer.

TemporaryBuffer m\_tempChBuf;

TemporaryBuffer m\_tempChBufSecondary;

charcount\_t m\_line;

ScanState m\_scanState;

tokens m\_errorToken;

charcount\_t m\_ichMinError;

charcount\_t m\_ichLimError;

charcount\_t m\_startLine;

EncodedCharPtr m\_pchStartLine;

Js::ScriptContext\* m\_scriptContext;

const Js::CharClassifier \*charClassifier;

tokens m\_tkPrevious;

size\_t m\_iecpLimTokPrevious;

Scanner(Parser\* parser, HashTbl \*phtbl, Token \*ptoken, ErrHandler \*perr, Js::ScriptContext \*scriptContext);

~Scanner(void);

tokens ScanCore(bool identifyKwds);

tokens ScanAhead();

tokens ScanError(EncodedCharPtr pchCur, tokens errorToken)

{

m\_currentCharacter = pchCur;

m\_errorToken = errorToken;

return m\_ptoken->tk = tkScanError;

}

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

{

Assert(FAILED(hr));

m\_pchMinTok = m\_currentCharacter;

m\_cMinTokMultiUnits = m\_cMultiUnits;

AssertMem(m\_perr);

m\_perr->Throw(hr);

}

const EncodedCharPtr PchBase(void)

{

return m\_pchBase;

}

const EncodedCharPtr PchMinTok(void)

{

return m\_pchMinTok;

}

template<bool stringTemplateMode, bool createRawString> tokens ScanStringConstant(OLECHAR delim, EncodedCharPtr \*pp);

tokens ScanStringConstant(OLECHAR delim, EncodedCharPtr \*pp);

tokens ScanStringTemplateBegin(EncodedCharPtr \*pp);

tokens ScanStringTemplateMiddleOrEnd(EncodedCharPtr \*pp);

void ScanNewLine(uint ch);

void NotifyScannedNewLine();

charcount\_t LineLength(EncodedCharPtr first, EncodedCharPtr last);

tokens ScanIdentifier(bool identifyKwds, EncodedCharPtr \*pp);

BOOL FastIdentifierContinue(EncodedCharPtr&p, EncodedCharPtr last);

tokens ScanIdentifierContinue(bool identifyKwds, bool fHasEscape, bool fHasMultiChar, EncodedCharPtr pchMin, EncodedCharPtr p, EncodedCharPtr \*pp);

tokens SkipComment(EncodedCharPtr \*pp, /\* out \*/ bool\* containTypeDef);

tokens ScanRegExpConstant(ArenaAllocator\* alloc);

tokens ScanRegExpConstantNoAST(ArenaAllocator\* alloc);

BOOL oFScanNumber(double \*pdbl, bool& likelyInt);

EncodedCharPtr FScanNumber(EncodedCharPtr p, double \*pdbl, bool& likelyInt);

IdentPtr PidOfIdentiferAt(EncodedCharPtr p, EncodedCharPtr last, bool fHadEscape, bool fHasMultiChar);

IdentPtr PidOfIdentiferAt(EncodedCharPtr p, EncodedCharPtr last);

ulong UnescapeToTempBuf(EncodedCharPtr p, EncodedCharPtr last);

void SaveSrcPos(void)

{

m\_pchMinTok = m\_currentCharacter;

}

OLECHAR PeekNextChar(void)

{

return PeekFull(m\_currentCharacter, m\_pchLast);

}

OLECHAR ReadNextChar(void)

{

return ReadFull<true>(m\_currentCharacter, m\_pchLast);

}

OLECHAR NextNonWhiteChar(EncodedCharPtr p, EncodedCharPtr last)

{

OLECHAR ch;

do

{

ch = ReadFull<false>(p, last);

}

while (this->charClassifier->IsWhiteSpace(ch));

return ch;

}

OLECHAR NextNonWhiteCharPlusOne(EncodedCharPtr p, EncodedCharPtr last)

{

OLECHAR ch;

do

{

ch = ReadFull<false>(p, last);

}

while (this->charClassifier->IsWhiteSpace(ch));

return ReadFull<false>(p, last);

}

EncodedCharPtr AdjustedLast() const

{

return m\_pchLast;

}

size\_t AdjustedLength() const

{

return AdjustedLast() - m\_pchBase;

}

bool IsStrictMode() const

{

return this->m\_parser != NULL && this->m\_parser->IsStrictMode();

}

// This function expects the first character to be a 'u'

// It will attempt to return a codepoint represented by a single escape point (either of the form \uXXXX or \u{any number of hex characters, s.t. value < 0x110000}

bool TryReadEscape(EncodedCharPtr &startingLocation, EncodedCharPtr endOfSource, codepoint\_t \*outChar = nullptr);

template <bool bScan>

bool TryReadCodePointRest(codepoint\_t lower, EncodedCharPtr &startingLocation, EncodedCharPtr endOfSource, codepoint\_t \*outChar, bool \*outContainsMultiUnitChar);

template <bool bScan>

\_\_inline bool TryReadCodePoint(EncodedCharPtr &startingLocation, EncodedCharPtr endOfSource, codepoint\_t \*outChar, bool \*hasEscape, bool \*outContainsMultiUnitChar);

\_\_inline BOOL IsIdContinueNext(EncodedCharPtr startingLocation, EncodedCharPtr endOfSource)

{

codepoint\_t nextCodepoint;

bool ignore;

if (TryReadCodePoint<false>(startingLocation, endOfSource, &nextCodepoint, &ignore, &ignore))

{

return charClassifier->IsIdContinue(nextCodepoint);

}

return false;

}

};

typedef Scanner<NullTerminatedUTF8EncodingPolicy> UTF8Scanner;

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

// 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"

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

\*

\* The following table speeds various tests of characters, such as whether

\* a given character can be part of an identifier, and so on.

\*/

int CountNewlines(LPCOLESTR psz, int cch)

{

int cln = 0;

while (0 != \*psz && 0 != cch--)

{

switch (\*psz++)

{

case OLESTR('\xD'):

if (\*psz == OLESTR('\xA'))

{

++psz;

if (0 == cch--)

break;

}

// fall-through

case OLESTR('\xA'):

cln++;

break;

}

}

return cln;

}

template< typename CharT >

struct AorW

{

};

// Specialization for UTF8Char

template<>

struct AorW< UTF8Char >

{

// Expressing the args as "arrays of size N" ensures that the both args

// are the same length. If not, we get a compile time error.

template< size\_t N >

static const UTF8Char\* Choose( const char (&a)[N], const wchar\_t (&w)[N] )

{

// The reinterpret\_cast is necessary to go from signed to unsigned char

return reinterpret\_cast< const UTF8Char\* >(a);

}

template< size\_t N >

static const bool Test(const char (&a)[N], const wchar\_t (&w)[N], LPCUTF8 value)

{

return 0 == memcmp(a, value, (N - 1) \* sizeof(utf8char\_t));

}

template< size\_t N >

static const bool Test(const char (&a)[N], const wchar\_t (&w)[N], LPCUTF8 start, LPCUTF8 end)

{

return (end - start == N - 1) && (0 == memcmp(a, start, (N - 1) \* sizeof(utf8char\_t)));

}

};

// Specialization for OLECHAR

template<>

struct AorW< OLECHAR >

{

template< size\_t N >

static const wchar\_t\* Choose( const char (&a)[N], const wchar\_t (&w)[N] )

{

return w;

}

template < size\_t N >

static bool Test(const char (&a)[N], const wchar\_t (&w)[N], const wchar\_t \*value)

{

return 0 == memcmp(w, value, (N - 1) \* sizeof(wchar\_t));

}

template < size\_t N >

static bool Test(const char (&a)[N], const wchar\_t (&w)[N], const wchar\_t \*start, const wchar\_t \*end)

{

return (end - start == N - 1) && (0 == memcmp(w, start, (N - 1) \* sizeof(wchar\_t)));

}

};

BOOL Token::IsKeyword() const

{

// keywords (but not future reserved words)

return (tk <= tkYIELD);

}

tokens Token::SetRegex(UnifiedRegex::RegexPattern \*const pattern, Parser \*const parser)

{

Assert(parser);

if(pattern)

parser->RegisterRegexPattern(pattern);

this->u.pattern = pattern;

return tk = tkRegExp;

}

IdentPtr Token::CreateIdentifier(HashTbl \* hashTbl)

{

Assert(this->u.pid == nullptr);

if (this->u.pchMin)

{

Assert(IsIdentifier());

IdentPtr pid = hashTbl->PidHashNameLen(this->u.pchMin, this->u.length);

this->u.pid = pid;

return pid;

}

Assert(IsReservedWord());

IdentPtr pid = hashTbl->PidFromTk(tk);

this->u.pid = pid;

return pid;

}

template <typename EncodingPolicy>

Scanner<EncodingPolicy>::Scanner(Parser\* parser, HashTbl \*phtbl, Token \*ptoken, ErrHandler \*perr, Js::ScriptContext\* scriptContext)

{

AssertMem(phtbl);

AssertMem(ptoken);

AssertMem(perr);

m\_parser = parser;

m\_phtbl = phtbl;

m\_ptoken = ptoken;

m\_cMinLineMultiUnits = 0;

m\_perr = perr;

m\_fHadEol = FALSE;

m\_doubleQuoteOnLastTkStrCon = FALSE;

m\_OctOrLeadingZeroOnLastTKNumber = false;

m\_fStringTemplateDepth = 0;

m\_scanState = ScanStateNormal;

m\_scriptContext = scriptContext;

m\_line = 0;

m\_startLine = 0;

m\_pchStartLine = NULL;

m\_ichMinError = 0;

m\_ichLimError = 0;

m\_tempChBuf.m\_pscanner = this;

m\_tempChBufSecondary.m\_pscanner = this;

m\_iecpLimTokPrevious = (size\_t)-1;

this->charClassifier = scriptContext->GetCharClassifier();

this->es6UnicodeMode = scriptContext->GetConfig()->IsES6UnicodeExtensionsEnabled();

m\_fYieldIsKeyword = false;

m\_fAwaitIsKeyword = false;

}

template <typename EncodingPolicy>

Scanner<EncodingPolicy>::~Scanner(void)

{

}

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

\*

\* Initializes the scanner to prepare to scan the given source text.

\*/

template <typename EncodingPolicy>

void Scanner<EncodingPolicy>::SetText(EncodedCharPtr pszSrc, size\_t offset, size\_t length, charcount\_t charOffset, ULONG grfscr, ULONG lineNumber)

{

// Save the start of the script and add the offset to get the point where we should start scanning.

m\_pchBase = pszSrc;

m\_pchLast = m\_pchBase + offset + length;

m\_pchPrevLine = m\_currentCharacter = m\_pchMinLine = m\_pchMinTok = pszSrc + offset;

RestoreMultiUnits(offset - charOffset);

// Absorb any byte order mark at the start

if(offset == 0)

{

switch( PeekFull(m\_currentCharacter, m\_pchLast) )

{

case 0xFFEE: // "Opposite" endian BOM

// We do not support big-endian encodings

// fall-through

case 0xFEFF: // "Correct" BOM

ReadFull<true>(m\_currentCharacter, m\_pchLast);

break;

}

}

m\_line = lineNumber;

m\_startLine = lineNumber;

m\_pchStartLine = m\_currentCharacter;

m\_ptoken->tk = tkNone;

m\_fHtmlComments = (grfscr & fscrHtmlComments) != 0;

m\_fHadEol = FALSE;

m\_fSyntaxColor = (grfscr & fscrSyntaxColor) != 0;

m\_DeferredParseFlags = ScanFlagNone;

}

template <typename EncodingPolicy>

void Scanner<EncodingPolicy>::PrepareForBackgroundParse(Js::ScriptContext \*scriptContext)

{

scriptContext->GetThreadContext()->GetStandardChars((EncodedChar\*)0);

scriptContext->GetThreadContext()->GetStandardChars((wchar\_t\*)0);

}

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

// Number of code points from 'first' up to, but not including the next

// newline character, embedded NUL, or 'last', depending on which comes first.

//

// This is used to determine a length of BSTR, which can't contain a NUL character.

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

template <typename EncodingPolicy>

charcount\_t Scanner<EncodingPolicy>::LineLength(EncodedCharPtr first, EncodedCharPtr last)

{

charcount\_t result = 0;

EncodedCharPtr p = first;

for (;;)

{

switch( ReadFull<false>(p, last) )

{

case kchNWL: // \_C\_NWL

case kchRET:

case kchLS:

case kchPS:

case kchNUL: // \_C\_NUL

return result;

}

result++;

}

}

template <typename EncodingPolicy>

charcount\_t Scanner<EncodingPolicy>::UpdateLine(long &line, EncodedCharPtr start, EncodedCharPtr last, charcount\_t ichStart, charcount\_t ichEnd)

{

EncodedCharPtr p = start;

charcount\_t ich = ichStart;

long current = line;

charcount\_t lastStart = ichStart;

while (ich < ichEnd)

{

ich++;

switch (ReadFull<false>(p, last))

{

case kchRET:

if (PeekFull(p, last) == kchNWL)

{

ich++;

ReadFull<false>(p, last);

}

// fall-through

case kchNWL:

case kchLS:

case kchPS:

current++;

lastStart = ich;

break;

case kchNUL:

goto done;

}

}

done:

line = current;

return lastStart;

}

template <typename EncodingPolicy>

bool Scanner<EncodingPolicy>::TryReadEscape(EncodedCharPtr& startingLocation, EncodedCharPtr endOfSource, codepoint\_t \*outChar)

{

Assert(outChar != nullptr);

Assert(startingLocation <= endOfSource);

EncodedCharPtr currentLocation = startingLocation;

codepoint\_t charToOutput = 0x0;

// '\' is Assumed as there is only one caller

// Read 'u' characters

if (currentLocation >= endOfSource || ReadFirst(currentLocation, endOfSource) != 'u')

{

return false;

}

bool expectCurly = false;

if (currentLocation < endOfSource && PeekFirst(currentLocation, endOfSource) == '{' && es6UnicodeMode)

{

expectCurly = true;

// Move past the character

ReadFirst(currentLocation, endOfSource);

}

uint i = 0;

OLECHAR ch = 0;

int hexValue = 0;

uint maxHexDigits = (expectCurly ? MAXUINT32 : 4u);

for(; i < maxHexDigits && currentLocation < endOfSource; i++)

{

if (!Js::NumberUtilities::FHexDigit(ch = ReadFirst(currentLocation, endOfSource), &hexValue))

{

break;

}

charToOutput = charToOutput \* 0x10 + hexValue;

if (charToOutput > 0x10FFFF)

{

return false;

}

}

//At least 4 characters have to be read

if (i == 0 || (i != 4 && !expectCurly))

{

return false;

}

Assert(expectCurly ? es6UnicodeMode : true);

if (expectCurly && ch != '}')

{

return false;

}

\*outChar = charToOutput;

startingLocation = currentLocation;

return true;

}

template <typename EncodingPolicy>

template <bool bScan>

bool Scanner<EncodingPolicy>::TryReadCodePointRest(codepoint\_t lower, EncodedCharPtr& startingLocation, EncodedCharPtr endOfSource, codepoint\_t \*outChar, bool \*outContainsMultiUnitChar)

{

Assert(outChar != nullptr);

Assert(outContainsMultiUnitChar != nullptr);

Assert(es6UnicodeMode);

Assert(Js::NumberUtilities::IsSurrogateLowerPart(lower));

EncodedCharPtr currentLocation = startingLocation;

\*outChar = lower;

if (currentLocation < endOfSource)

{

size\_t restorePoint = m\_cMultiUnits;

codepoint\_t upper = ReadFull<bScan>(currentLocation, endOfSource);

if (Js::NumberUtilities::IsSurrogateUpperPart(upper))

{

\*outChar = Js::NumberUtilities::SurrogatePairAsCodePoint(lower, upper);

if (IsMultiUnitChar(static\_cast<OLECHAR>(upper)))

{

\*outContainsMultiUnitChar = true;

}

startingLocation = currentLocation;

}

else

{

RestoreMultiUnits(restorePoint);

}

}

return true;

}

template <typename EncodingPolicy>

template <bool bScan>

\_\_inline bool Scanner<EncodingPolicy>::TryReadCodePoint(EncodedCharPtr &startingLocation, EncodedCharPtr endOfSource, codepoint\_t \*outChar, bool \*hasEscape, bool \*outContainsMultiUnitChar)

{

Assert(outChar != nullptr);

Assert(outContainsMultiUnitChar != nullptr);

if (startingLocation >= endOfSource)

{

return false;

}

codepoint\_t ch = ReadFull<bScan>(startingLocation, endOfSource);

if (FBigChar(ch))

{

if (IsMultiUnitChar(static\_cast<OLECHAR>(ch)))

{

\*outContainsMultiUnitChar = true;

}

if (es6UnicodeMode && Js::NumberUtilities::IsSurrogateLowerPart(ch))

{

return TryReadCodePointRest<bScan>(ch, startingLocation, endOfSource, outChar, outContainsMultiUnitChar);

}

}

else if (ch == '\\' && TryReadEscape(startingLocation, endOfSource, &ch))

{

\*hasEscape = true;

}

\*outChar = ch;

return true;

}

template <typename EncodingPolicy>

tokens Scanner<EncodingPolicy>::ScanIdentifier(bool identifyKwds, EncodedCharPtr \*pp)

{

EncodedCharPtr p = \*pp;

EncodedCharPtr pchMin = p;

// JS6 allows unicode characters in the form of \uxxxx escape sequences

// to be part of the identifier.

bool fHasEscape = false;

bool fHasMultiChar = false;

codepoint\_t codePoint = INVALID\_CODEPOINT;

size\_t multiUnitsBeforeLast = m\_cMultiUnits;

// Check if we started the id

if (!TryReadCodePoint<true>(p, m\_pchLast, &codePoint, &fHasEscape, &fHasMultiChar))

{

// If no chars. could be scanned as part of the identifier, return error.

return tkScanError;

}

Assert(codePoint < 0x110000u);

if (!charClassifier->IsIdStart(codePoint))

{

// Put back the last character

RestoreMultiUnits(multiUnitsBeforeLast);

// If no chars. could be scanned as part of the identifier, return error.

return tkScanError;

}

return ScanIdentifierContinue(identifyKwds, fHasEscape, fHasMultiChar, pchMin, p, pp);

}

template <typename EncodingPolicy>

BOOL Scanner<EncodingPolicy>::FastIdentifierContinue(EncodedCharPtr&p, EncodedCharPtr last)

{

if (MultiUnitEncoding)

{

while (p < last)

{

EncodedChar currentChar = \*p;

if (IsMultiUnitChar(currentChar))

{

// multi unit character, we may not have reach the end yet

return FALSE;

}

Assert(currentChar != '\\' || !charClassifier->IsIdContinueFast<false>(currentChar));

if (!charClassifier->IsIdContinueFast<false>(currentChar))

{

// only reach the end of the identifier if it is not the start of an escape sequence

return currentChar != '\\';

}

p++;

}

// We have reach the end of the identifier.

return TRUE;

}

// Not fast path for non multi unit encoding

return false;

}

template <typename EncodingPolicy>

tokens Scanner<EncodingPolicy>::ScanIdentifierContinue(bool identifyKwds, bool fHasEscape, bool fHasMultiChar,

EncodedCharPtr pchMin, EncodedCharPtr p, EncodedCharPtr \*pp)

{

EncodedCharPtr last = m\_pchLast;

while (true)

{

// Fast path for utf8, non-multi unit char and not escape

if (FastIdentifierContinue(p, last))

{

break;

}

// Slow path that has to deal with multi unit encoding

codepoint\_t codePoint = INVALID\_CODEPOINT;

EncodedCharPtr pchBeforeLast = p;

size\_t multiUnitsBeforeLast = m\_cMultiUnits;

if (TryReadCodePoint<true>(p, last, &codePoint, &fHasEscape, &fHasMultiChar))

{

Assert(codePoint < 0x110000u);

if (charClassifier->IsIdContinue(codePoint))

{

continue;

}

}

// Put back the last character

p = pchBeforeLast;

RestoreMultiUnits(multiUnitsBeforeLast);

break;

}

Assert(p - pchMin > 0 && p - pchMin <= LONG\_MAX);

\*pp = p;

if (!identifyKwds)

{

return tkID;

}

// During syntax coloring, scanner doesn't need to convert the escape sequence to get actual characters, it just needs the classification information

// So call up hashtables custom method to check if the string scanned is identifier or keyword.

// Do the same for deferred parsing, but use a custom method that only tokenizes JS keywords.

if ((m\_DeferredParseFlags & ScanFlagSuppressIdPid) != 0)

{

m\_ptoken->SetIdentifier(NULL);

if (!fHasEscape)

{

// If there are no escape, that the main scan loop would have found the keyword already

// So we can just assume it is an ID

DebugOnly(long cch = UnescapeToTempBuf(pchMin, p));

DebugOnly(tokens tk = m\_phtbl->TkFromNameLen(m\_tempChBuf.m\_prgch, cch, IsStrictMode()));

Assert(tk == tkID || (tk == tkYIELD && !m\_fYieldIsKeyword) || (tk == tkAWAIT && !m\_fAwaitIsKeyword));

return tkID;

}

long cch = UnescapeToTempBuf(pchMin, p);

tokens tk = m\_phtbl->TkFromNameLen(m\_tempChBuf.m\_prgch, cch, IsStrictMode());

return (!m\_fYieldIsKeyword && tk == tkYIELD) || (!m\_fAwaitIsKeyword && tk == tkAWAIT) ? tkID : tk;

}

else if (m\_fSyntaxColor)

{

m\_ptoken->SetIdentifier(NULL);

// We always need to check TkFromNameLenColor because

// the main Scan switch doesn't detect all non-keyword that needs coloring

// (e.g. int)

long cch = UnescapeToTempBuf(pchMin, p);

return m\_phtbl->TkFromNameLenColor(m\_tempChBuf.m\_prgch, cch);

}

// UTF16 Scanner are only for syntax coloring, so it shouldn't come here.

if (MultiUnitEncoding && !fHasMultiChar && !fHasEscape)

{

Assert(sizeof(EncodedChar) == 1);

// If there are no escape, that the main scan loop would have found the keyword already

// So we can just assume it is an ID

DebugOnly(long cch = UnescapeToTempBuf(pchMin, p));

DebugOnly(tokens tk = m\_phtbl->TkFromNameLen(m\_tempChBuf.m\_prgch, cch, IsStrictMode()));

Assert(tk == tkID || (tk == tkYIELD && !m\_fYieldIsKeyword) || (tk == tkAWAIT && !m\_fAwaitIsKeyword));

m\_ptoken->SetIdentifier(reinterpret\_cast<const char \*>(pchMin), (long)(p - pchMin));

return tkID;

}

IdentPtr pid = PidOfIdentiferAt(pchMin, p, fHasEscape, fHasMultiChar);

m\_ptoken->SetIdentifier(pid);

if (!fHasEscape)

{

// If it doesn't have escape, then Scan() should have taken care of keywords (except

// yield if m\_fYieldIsKeyword is false, in which case yield is treated as an identifier, and except

// await if m\_fAwaitIsKeyword is false, in which case await is treated as an identifier).

// We don't have to check if the name is reserved word and return it as an Identifier

Assert(pid->Tk(IsStrictMode()) == tkID

|| (pid->Tk(IsStrictMode()) == tkYIELD && !m\_fYieldIsKeyword)

|| (pid->Tk(IsStrictMode()) == tkAWAIT && !m\_fAwaitIsKeyword));

return tkID;

}

tokens tk = pid->Tk(IsStrictMode());

return tk == tkID || (tk == tkYIELD && !m\_fYieldIsKeyword) || (tk == tkAWAIT && !m\_fAwaitIsKeyword) ? tkID : tkNone;

}

template <typename EncodingPolicy>

IdentPtr Scanner<EncodingPolicy>::PidAt(size\_t iecpMin, size\_t iecpLim)

{

Assert(iecpMin < AdjustedLength() && iecpLim <= AdjustedLength() && iecpLim > iecpMin);

return PidOfIdentiferAt(m\_pchBase + iecpMin, m\_pchBase + iecpLim);

}

template <typename EncodingPolicy>

ulong Scanner<EncodingPolicy>::UnescapeToTempBuf(EncodedCharPtr p, EncodedCharPtr last)

{

m\_tempChBuf.Init();

while( p < last )

{

codepoint\_t codePoint;

bool hasEscape, isMultiChar;

bool gotCodePoint = TryReadCodePoint<false>(p, last, &codePoint, &hasEscape, &isMultiChar);

Assert(gotCodePoint);

Assert(codePoint < 0x110000);

if (codePoint < 0x10000)

{

m\_tempChBuf.AppendCh((OLECHAR)codePoint);

}

else

{

wchar\_t lower, upper;

Js::NumberUtilities::CodePointAsSurrogatePair(codePoint, &lower, &upper);

m\_tempChBuf.AppendCh(lower);

m\_tempChBuf.AppendCh(upper);

}

}

return m\_tempChBuf.m\_ichCur;

}

template <typename EncodingPolicy>

IdentPtr Scanner<EncodingPolicy>::PidOfIdentiferAt(EncodedCharPtr p, EncodedCharPtr last)

{

long cch = UnescapeToTempBuf(p, last);

return m\_phtbl->PidHashNameLen(m\_tempChBuf.m\_prgch, cch);

}

template <typename EncodingPolicy>

IdentPtr Scanner<EncodingPolicy>::PidOfIdentiferAt(EncodedCharPtr p, EncodedCharPtr last, bool fHadEscape, bool fHasMultiChar)

{

// If there is an escape sequence in the JS6 identifier or it is a UTF8

// source then we have to convert it to the equivalent char so we use a

// buffer for translation.

if ((MultiUnitEncoding && fHasMultiChar) || fHadEscape)

{

return PidOfIdentiferAt(p, last);

}

else if (MultiUnitEncoding)

{

Assert(sizeof(EncodedChar) == 1);

return m\_phtbl->PidHashNameLen(reinterpret\_cast<const char \*>(p), (long)(last - p));

}

else

{

Assert(sizeof(EncodedChar) == 2);

return m\_phtbl->PidHashNameLen(reinterpret\_cast< const wchar\_t \* >(p), (long)(last - p));

}

}

template <typename EncodingPolicy>

typename Scanner<EncodingPolicy>::EncodedCharPtr Scanner<EncodingPolicy>::FScanNumber(EncodedCharPtr p, double \*pdbl, bool& likelyInt)

{

EncodedCharPtr last = m\_pchLast;

EncodedCharPtr pchT;

likelyInt = true;

// Reset

m\_OctOrLeadingZeroOnLastTKNumber = false;

if ('0' == PeekFirst(p, last))

{

switch(PeekFirst(p + 1, last))

{

case '.':

case 'e':

case 'E':

likelyInt = false;

// Floating point

goto LFloat;

case 'x':

case 'X':

// Hex

\*pdbl = Js::NumberUtilities::DblFromHex(p + 2, &pchT);

if (pchT == p + 2)

{

// "Octal zero token "0" followed by an identifier token beginning with character 'x'/'X'

\*pdbl = 0;

return p + 1;

}

else

return pchT;

case 'o':

case 'O':

if (!this->m\_scriptContext->GetConfig()->IsES6NumericLiteralEnabled())

{

goto LDefaultFScanNumber;

}

// Octal

\*pdbl = Js::NumberUtilities::DblFromOctal(p + 2, &pchT);

if (pchT == p + 2)

{

// "Octal zero token "0" followed by an identifier token beginning with character 'o'/'O'

\*pdbl = 0;

return p + 1;

}

return pchT;

case 'b':

case 'B':

if (!this->m\_scriptContext->GetConfig()->IsES6NumericLiteralEnabled())

{

goto LDefaultFScanNumber;

}

// Binary

\*pdbl = Js::NumberUtilities::DblFromBinary(p + 2, &pchT);

if (pchT == p + 2)

{

// "Octal zero token "0" followed by an identifier token beginning with character 'b'/'B'

\*pdbl = 0;

return p + 1;

}

return pchT;

default:

LDefaultFScanNumber :

// Octal

\*pdbl = Js::NumberUtilities::DblFromOctal(p, &pchT);

Assert(pchT > p);

#if !SOURCERELEASE

// If an octal literal is malformed then it is in fact a decimal literal.

#endif // !SOURCERELEASE

if(\*pdbl != 0 || pchT > p + 1)

m\_OctOrLeadingZeroOnLastTKNumber = true; //report as an octal or hex for JSON when leading 0. Just '0' is ok

switch (\*pchT)

{

case '8':

case '9':

// case 'e':

// case 'E':

// case '.':

m\_OctOrLeadingZeroOnLastTKNumber = false; //08... or 09....

goto LFloat;

}

return pchT;

}

}

else

{

LFloat:

\*pdbl = Js::NumberUtilities::StrToDbl(p, &pchT, likelyInt);

Assert(pchT == p || !Js::NumberUtilities::IsNan(\*pdbl));

return pchT;

}

}

template <typename EncodingPolicy>

BOOL Scanner<EncodingPolicy>::oFScanNumber(double \*pdbl, bool& likelyInt)

{

EncodedCharPtr pchT;

m\_OctOrLeadingZeroOnLastTKNumber = false;

likelyInt = true;

if ('0' == \*m\_currentCharacter)

{

switch (m\_currentCharacter[1])

{

case '.':

case 'e':

case 'E':

likelyInt = false;

// Floating point.

goto LFloat;

case 'x':

case 'X':

// Hex.

\*pdbl = Js::NumberUtilities::DblFromHex<EncodedChar>(m\_currentCharacter + 2, &pchT);

if (pchT == m\_currentCharacter + 2)

{

// "Octal zero token "0" followed by an identifier token beginning with character 'x'/'X'

\*pdbl = 0;

m\_currentCharacter++;

}

else

m\_currentCharacter = pchT;

break;

case 'o':

case 'O':

if (!this->m\_scriptContext->GetConfig()->IsES6NumericLiteralEnabled())

{

goto LDefaultoFScanNumber;

}

\*pdbl = Js::NumberUtilities::DblFromOctal(m\_currentCharacter + 2, &pchT);

if (pchT == m\_currentCharacter + 2)

{

// "Octal zero token "0" followed by an identifier token beginning with character 'o'/'O'

\*pdbl = 0;

m\_currentCharacter++;

}

else

m\_currentCharacter = pchT;

break;

case 'b':

case 'B':

if (!this->m\_scriptContext->GetConfig()->IsES6NumericLiteralEnabled())

{

goto LDefaultoFScanNumber;

}

\*pdbl = Js::NumberUtilities::DblFromBinary(m\_currentCharacter + 2, &pchT);

if (pchT == m\_currentCharacter + 2)

{

// "Octal zero token "0" followed by an identifier token beginning with character 'b'/'B'

\*pdbl = 0;

m\_currentCharacter++;

}

else

m\_currentCharacter = pchT;

break;

default:

LDefaultoFScanNumber :

// Octal.

\*pdbl = Js::NumberUtilities::DblFromOctal(m\_currentCharacter, &pchT);

Assert(pchT > m\_currentCharacter);

#if !SOURCERELEASE

// If an octal literal is malformed then it is in fact a decimal literal.

#endif // !SOURCERELEASE

if(\*pdbl != 0 || pchT > m\_currentCharacter + 1)

m\_OctOrLeadingZeroOnLastTKNumber = true; //report as an octal or hex for JSON when leading 0. Just '0' is ok

switch (\*pchT)

{

case '8':

case '9':

// case 'e':

// case 'E':

// case '.':

m\_OctOrLeadingZeroOnLastTKNumber = false; //08... or 09....

goto LFloat;

}

m\_currentCharacter = pchT;

break;

}

}

else

{

LFloat:

// Let StrToDbl do all the work.

\*pdbl = Js::NumberUtilities::StrToDbl(m\_currentCharacter, &pchT, likelyInt);

if (pchT == m\_currentCharacter)

return FALSE;

m\_currentCharacter = pchT;

Assert(!Js::NumberUtilities::IsNan(\*pdbl));

}

return TRUE;

}

template <typename EncodingPolicy>

tokens Scanner<EncodingPolicy>::TryRescanRegExp()

{

EncodedCharPtr current = m\_currentCharacter;

tokens result = RescanRegExp();

if (result == tkScanError)

m\_currentCharacter = current;

return result;

}

template <typename EncodingPolicy>

tokens Scanner<EncodingPolicy>::RescanRegExp()

{

#if DEBUG

switch (m\_ptoken->tk)

{

case tkDiv:

Assert(m\_currentCharacter == m\_pchMinTok + 1);

break;

case tkAsgDiv:

Assert(m\_currentCharacter == m\_pchMinTok + 2);

break;

default:

AssertMsg(FALSE, "Who is calling RescanRegExp?");

break;

}

#endif //DEBUG

m\_currentCharacter = m\_pchMinTok;

if (\*m\_currentCharacter != '/')

Error(ERRnoSlash);

m\_currentCharacter++;

tokens tk = tkNone;

{

ArenaAllocator alloc(L"RescanRegExp", m\_parser->GetAllocator()->GetPageAllocator(), m\_parser->GetAllocator()->outOfMemoryFunc);

tk = ScanRegExpConstant(&alloc);

}

return tk;

}

template <typename EncodingPolicy>

tokens Scanner<EncodingPolicy>::RescanRegExpNoAST()

{

#if DEBUG

switch (m\_ptoken->tk)

{

case tkDiv:

Assert(m\_currentCharacter == m\_pchMinTok + 1);

break;

case tkAsgDiv:

Assert(m\_currentCharacter == m\_pchMinTok + 2);

break;

default:

AssertMsg(FALSE, "Who is calling RescanRegExpNoParseTree?");

break;

}

#endif //DEBUG

m\_currentCharacter = m\_pchMinTok;

if (\*m\_currentCharacter != '/')

Error(ERRnoSlash);

m\_currentCharacter++;

tokens tk = tkNone;

{

ArenaAllocator alloc(L"RescanRegExp", m\_parser->GetAllocator()->GetPageAllocator(), m\_parser->GetAllocator()->outOfMemoryFunc);

{

tk = ScanRegExpConstantNoAST(&alloc);

}

}

return tk;

}

template <typename EncodingPolicy>

tokens Scanner<EncodingPolicy>::RescanRegExpTokenizer()

{

#if DEBUG

switch (m\_ptoken->tk)

{

case tkDiv:

Assert(m\_currentCharacter == m\_pchMinTok + 1);

break;

case tkAsgDiv:

Assert(m\_currentCharacter == m\_pchMinTok + 2);

break;

default:

AssertMsg(FALSE, "Who is calling RescanRegExpNoParseTree?");

break;

}

#endif //DEBUG

m\_currentCharacter = m\_pchMinTok;

if (\*m\_currentCharacter != '/')

Error(ERRnoSlash);

m\_currentCharacter++;

tokens tk = tkNone;

ThreadContext \*threadContext = ThreadContext::GetContextForCurrentThread();

threadContext->EnsureRecycler();

Js::TempArenaAllocatorObject \*alloc = threadContext->GetTemporaryAllocator(L"RescanRegExp");

\_\_try

{

tk = ScanRegExpConstantNoAST(alloc->GetAllocator());

}

\_\_finally

{

threadContext->ReleaseTemporaryAllocator(alloc);

}

return tk;

}

template <typename EncodingPolicy>

tokens Scanner<EncodingPolicy>::ScanRegExpConstant(ArenaAllocator\* alloc)

{

if (m\_parser && m\_parser->IsBackgroundParser())

{

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

}

else

{

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

}

// SEE ALSO: RegexHelper::PrimCompileDynamic()

#ifdef PROFILE\_EXEC

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

#endif

ArenaAllocator\* ctAllocator = alloc;

UnifiedRegex::StandardChars<EncodedChar>\* standardEncodedChars = m\_scriptContext->GetThreadContext()->GetStandardChars((EncodedChar\*)0);

UnifiedRegex::StandardChars<wchar\_t>\* standardChars = m\_scriptContext->GetThreadContext()->GetStandardChars((wchar\_t\*)0);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

UnifiedRegex::DebugWriter \*w = 0;

if (REGEX\_CONFIG\_FLAG(RegexDebug))

w = m\_scriptContext->GetRegexDebugWriter();

if (REGEX\_CONFIG\_FLAG(RegexProfile))

m\_scriptContext->GetRegexStatsDatabase()->BeginProfile();

#endif

UnifiedRegex::Node\* root = 0;

charcount\_t totalLen = 0, bodyChars = 0, totalChars = 0, bodyLen = 0;

UnifiedRegex::RegexFlags flags = UnifiedRegex::NoRegexFlags;

UnifiedRegex::Parser<EncodingPolicy, true> parser

( m\_scriptContext

, ctAllocator

, standardEncodedChars

, standardChars

, IsFromExternalSource()

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, w

#endif

);

try

{

root = parser.ParseLiteral(m\_currentCharacter, m\_pchLast, bodyLen, totalLen, bodyChars, totalChars, flags);

}

catch (UnifiedRegex::ParseError e)

{

#ifdef PROFILE\_EXEC

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

#endif

if (m\_fSyntaxColor)

return ScanError(m\_currentCharacter + e.encodedPos, tkRegExp);

m\_currentCharacter += e.encodedPos;

Error(e.error);

}

UnifiedRegex::RegexPattern\* pattern;

if (m\_parser->IsBackgroundParser())

{

// Avoid allocating pattern from recycler on background thread. The main thread will create the pattern

// and hook it to this parse node.

pattern = parser.CompileProgram<false>(root, m\_currentCharacter, totalLen, bodyChars, totalChars, flags);

}

else

{

pattern = parser.CompileProgram<true>(root, m\_currentCharacter, totalLen, bodyChars, totalChars, flags);

}

RestoreMultiUnits(m\_cMultiUnits + parser.GetMultiUnits()); // m\_currentCharacter changed, sync MultiUnits

return m\_ptoken->SetRegex(pattern, m\_parser);

}

template<typename EncodingPolicy>

tokens Scanner<EncodingPolicy>::ScanRegExpConstantNoAST(ArenaAllocator\* alloc)

{

if (m\_parser && m\_parser->IsBackgroundParser())

{

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

}

else

{

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

}

ThreadContext \*threadContext = m\_fSyntaxColor ? ThreadContext::GetContextForCurrentThread() : m\_scriptContext->GetThreadContext();

UnifiedRegex::StandardChars<EncodedChar>\* standardEncodedChars = threadContext->GetStandardChars((EncodedChar\*)0);

UnifiedRegex::StandardChars<wchar\_t>\* standardChars = threadContext->GetStandardChars((wchar\_t\*)0);

charcount\_t totalLen = 0, bodyChars = 0, totalChars = 0, bodyLen = 0;

UnifiedRegex::Parser<EncodingPolicy, true> parser

( m\_scriptContext

, alloc

, standardEncodedChars

, standardChars

, IsFromExternalSource()

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, 0

#endif

);

try

{

parser.ParseLiteralNoAST(m\_currentCharacter, m\_pchLast, bodyLen, totalLen, bodyChars, totalChars);

}

catch (UnifiedRegex::ParseError e)

{

if (m\_fSyntaxColor)

return ScanError(m\_currentCharacter + e.encodedPos, tkRegExp);

m\_currentCharacter += e.encodedPos;

Error(e.error);

// never reached

}

UnifiedRegex::RegexPattern\* pattern = parser.CompileProgram<false>(nullptr, m\_currentCharacter, totalLen, bodyChars, totalChars, UnifiedRegex::NoRegexFlags);

Assert(pattern == nullptr); // BuildAST == false, CompileProgram should return nullptr

RestoreMultiUnits(m\_cMultiUnits + parser.GetMultiUnits()); // m\_currentCharacter changed, sync MultiUnits

return (m\_ptoken->tk = tkRegExp);

}

template<typename EncodingPolicy>

tokens Scanner<EncodingPolicy>::ScanStringTemplateBegin(EncodedCharPtr \*pp)

{

// String template must begin with a string constant followed by '`' or '${'

ScanStringConstant<true, true>('`', pp);

OLECHAR ch;

EncodedCharPtr last = m\_pchLast;

ch = ReadFirst(\*pp, last);

if (ch == '`')

{

// Simple string template - no substitutions

return tkStrTmplBasic;

}

else if (ch == '$')

{

ch = ReadFirst(\*pp, last);

if (ch == '{')

{

// Next token after expr should be tkStrTmplMid or tkStrTmplEnd.

// In string template scanning mode, we expect the next char to be '}'

// and will treat it as the beginning of tkStrTmplEnd or tkStrTmplMid

m\_fStringTemplateDepth++;

// Regular string template begin - next is first substitution

return tkStrTmplBegin;

}

}

// Error - make sure pointer stays at the last character of the error token instead of after it in the error case

(\*pp)--;

return ScanError(m\_currentCharacter, tkStrTmplBegin);

}

template<typename EncodingPolicy>

tokens Scanner<EncodingPolicy>::ScanStringTemplateMiddleOrEnd(EncodedCharPtr \*pp)

{

// String template middle and end tokens must begin with a string constant

ScanStringConstant<true, true>('`', pp);

OLECHAR ch;

EncodedCharPtr last = m\_pchLast;

ch = ReadFirst(\*pp, last);

if (ch == '`')

{

// No longer in string template scanning mode

m\_fStringTemplateDepth--;

// This is the last part of the template ...`

return tkStrTmplEnd;

}

else if (ch == '$')

{

ch = ReadFirst(\*pp, last);

if (ch == '{')

{

// This is just another middle part of the template }...${

return tkStrTmplMid;

}

}

// Error - make sure pointer stays at the last character of the error token instead of after it in the error case

(\*pp)--;

return ScanError(m\_currentCharacter, tkStrTmplEnd);

}

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

\*

\* Parses a string constant. Note that the string value is stored in

\* a volatile buffer (or allocated on the heap if too long), and thus

\* the string should be saved off before the next token is scanned.

\*/

template<typename EncodingPolicy>

template<bool stringTemplateMode, bool createRawString>

tokens Scanner<EncodingPolicy>::ScanStringConstant(OLECHAR delim, EncodedCharPtr \*pp)

{

static\_assert((stringTemplateMode && createRawString) || (!stringTemplateMode && !createRawString), "stringTemplateMode and createRawString must have the same value");

OLECHAR ch, c, rawch;

int wT;

EncodedCharPtr p = \*pp;

EncodedCharPtr last = m\_pchLast;

if (stringTemplateMode)

{

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

}

// Reset

m\_OctOrLeadingZeroOnLastTKNumber = false;

m\_EscapeOnLastTkStrCon = FALSE;

m\_tempChBuf.Init();

// Use template parameter to gate raw string creation.

// If createRawString is false, all these operations should be no-ops

if (createRawString)

{

m\_tempChBufSecondary.Init();

}

for (;;)

{

switch ((rawch = ch = ReadFirst(p, last)))

{

case kchRET:

if (stringTemplateMode)

{

if (PeekFirst(p, last) == kchNWL)

{

// Eat the <LF> char, ignore return

ReadFirst(p, last);

}

// Both <CR> and <CR><LF> are normalized to <LF> in template cooked and raw values

ch = rawch = kchNWL;

}

LEcmaLineBreak:

// Fall through

case kchNWL:

if (stringTemplateMode)

{

// Notify the scanner to update current line, number of lines etc

NotifyScannedNewLine();

break;

}

m\_currentCharacter = p - 1;

if (m\_fSyntaxColor)

{

\*pp = p - 1;

return ScanError(p - 1, tkStrCon);

}

Error(ERRnoStrEnd);

case '"':

case '\'':

if (ch == delim)

goto LBreak;

break;

case '`':

// In string template scan mode, don't consume the '`' - we need to differentiate

// between a closed string template and the expression open sequence - ${

if (stringTemplateMode)

{

p--;

goto LBreak;

}

// If we aren't scanning for a string template, do the default thing

goto LMainDefault;

case '$':

// If we are parsing a string literal part of a string template, ${ indicates we need to switch

// to parsing an expression.

if (stringTemplateMode && PeekFirst(p, last) == '{')

{

// Rewind to the $ and return

p--;

goto LBreak;

}

// If we aren't scanning for a string template, do the default thing

goto LMainDefault;

case kchNUL:

if (p >= last)

{

m\_currentCharacter = p - 1;

if (m\_fSyntaxColor)

{

\*pp = p - 1;

return ScanError(p - 1, tkStrCon);

}

Error(ERRnoStrEnd);

}

break;

default:

LMainDefault:

if (IsMultiUnitChar(ch))

{

if ((ch == kchLS || ch == kchPS))

{

goto LEcmaLineBreak;

}

rawch = ch = ReadRest<true>(ch, p, last);

switch (ch)

{

case kchLS: // 0x2028, classifies as new line

case kchPS: // 0x2029, classifies as new line

goto LEcmaLineBreak;

}

}

break;

case kchBSL:

// In raw mode '\\' is not an escape character, just add the char into the raw buffer.

m\_tempChBufSecondary.AppendCh<createRawString>(ch);

m\_EscapeOnLastTkStrCon=TRUE;

// In raw mode, we append the raw char itself and not the escaped value so save the char.

rawch = ch = ReadFirst(p, last);

codepoint\_t codePoint = 0;

uint errorType = (uint)ERRbadHexDigit;

switch (ch)

{

case 'b':

ch = 0x08;

break;

case 't':

ch = 0x09;

break;

case 'v':

ch = 0x0B; //Only in ES5 mode

break; //same as default

case 'n':

ch = 0x0A;

break;

case 'f':

ch = 0x0C;

break;

case 'r':

ch = 0x0D;

break;

case 'x':

// Insert the 'x' here before jumping to parse the hex digits.

m\_tempChBufSecondary.AppendCh<createRawString>(ch);

// 2 hex digits

ch = 0;

goto LTwoHex;

case 'u':

// Raw string just inserts a 'u' here.

m\_tempChBufSecondary.AppendCh<createRawString>(ch);

ch = 0;

if (Js::NumberUtilities::FHexDigit(c = ReadFirst(p, last), &wT))

goto LFourHex;

else if (c != '{' || !this->es6UnicodeMode)

goto ReturnScanError;

Assert(c == '{');

// c should definitely be a '{' which should be appended to the raw string.

m\_tempChBufSecondary.AppendCh<createRawString>(c);

//At least one digit is expected

if (!Js::NumberUtilities::FHexDigit(c = ReadFirst(p, last), &wT))

{

goto ReturnScanError;

}

m\_tempChBufSecondary.AppendCh<createRawString>(c);

codePoint = static\_cast<codepoint\_t>(wT);

while(Js::NumberUtilities::FHexDigit(c = ReadFirst(p, last), &wT))

{

m\_tempChBufSecondary.AppendCh<createRawString>(c);

codePoint <<= 4;

codePoint += static\_cast<codepoint\_t>(wT);

if (codePoint > 0x10FFFF)

{

errorType = (uint)ERRInvalidCodePoint;

goto ReturnScanError;

}

}

if (c != '}')

{

errorType = (uint)ERRMissingCurlyBrace;

goto ReturnScanError;

}

Assert(codePoint <= 0x10FFFF);

if (codePoint >= 0x10000)

{

OLECHAR lower = 0;

Js::NumberUtilities::CodePointAsSurrogatePair(codePoint, &lower, &ch);

m\_tempChBuf.AppendCh(lower);

}

else

{

ch = (wchar\_t)codePoint;

}

// In raw mode we want the last hex character or the closing curly. c should hold one or the other.

if (createRawString)

rawch = c;

break;

LFourHex:

codePoint = 0x0;

// Append first hex digit character to the raw string.

m\_tempChBufSecondary.AppendCh<createRawString>(c);

codePoint += static\_cast<codepoint\_t>(wT \* 0x1000);

if (!Js::NumberUtilities::FHexDigit(c = ReadFirst(p, last), &wT))

goto ReturnScanError;

// Append fourth (or second) hex digit character to the raw string.

m\_tempChBufSecondary.AppendCh<createRawString>(c);

codePoint += static\_cast<codepoint\_t>(wT \* 0x0100);

LTwoHex:

// This code path doesn't expect curly.

if (!Js::NumberUtilities::FHexDigit(c = ReadFirst(p, last), &wT))

goto ReturnScanError;

// Append first hex digit character to the raw string.

m\_tempChBufSecondary.AppendCh<createRawString>(c);

codePoint += static\_cast<codepoint\_t>(wT \* 0x0010);

if (!Js::NumberUtilities::FHexDigit(c = ReadFirst(p, last), &wT))

goto ReturnScanError;

codePoint += static\_cast<codepoint\_t>(wT);

// In raw mode we want the last hex character or the closing curly. c should hold one or the other.

if (createRawString)

rawch = c;

if (codePoint < 0x10000)

{

ch = static\_cast<OLECHAR>(codePoint);

}

else

{

goto ReturnScanError;

}

break;

case '0':

case '1':

case '2':

case '3':

// 1 to 3 octal digits

ch -= '0';

// Octal escape sequences are not allowed inside string template literals

if (stringTemplateMode)

{

c = PeekFirst(p, last);

if (ch != 0 || (c >= '0' && c <= '7'))

{

errorType = (uint)ERRES5NoOctal;

goto ReturnScanError;

}

break;

}

wT = (c = ReadFirst(p, last)) - '0';

if ((wchar\_t)wT > 7)

{

if (ch != 0 || ((wchar\_t)wT <= 9))

{

m\_OctOrLeadingZeroOnLastTKNumber = true;

}

p--;

break;

}

m\_OctOrLeadingZeroOnLastTKNumber = true;

ch = static\_cast< OLECHAR >(ch \* 8 + wT);

goto LOneOctal;

case '4':

case '5':

case '6':

case '7':

// 1 to 2 octal digits

// Octal escape sequences are not allowed inside string template literals

if (stringTemplateMode)

{

errorType = (uint)ERRES5NoOctal;

goto ReturnScanError;

}

ch -= '0';

m\_OctOrLeadingZeroOnLastTKNumber = true;

LOneOctal:

wT = (c = ReadFirst(p, last)) - '0';

if ((wchar\_t)wT > 7)

{

p--;

break;

}

ch = static\_cast< OLECHAR >(ch \* 8 + wT);

break;

case kchRET: // 0xD

if (stringTemplateMode)

{

// If this is \<CR><LF> we can eat the <LF> right now

if (PeekFirst(p, last) == kchNWL)

{

// Eat the <LF> char, ignore return

ReadFirst(p, last);

}

// Both \<CR> and \<CR><LF> are normalized to \<LF> in template raw string

rawch = kchNWL;

}

case kchLS: // 0x2028, classifies as new line

case kchPS: // 0x2029, classifies as new line

case kchNWL: // 0xA

LEcmaEscapeLineBreak:

if (stringTemplateMode)

{

// We're going to ignore the line continuation tokens for the cooked strings, but we need to append the token for raw strings

m\_tempChBufSecondary.AppendCh<createRawString>(rawch);

// Template literal strings ignore all escaped line continuation tokens

NotifyScannedNewLine();

continue;

}

m\_currentCharacter = p;

ScanNewLine(ch);

p = m\_currentCharacter;

if (m\_fSyntaxColor && \*p == 0)

{

// Special case for multi-line strings during colorization.

m\_scanState = delim == '"' ? ScanStateMultiLineDoubleQuoteString : ScanStateMultiLineSingleQuoteString;

\*pp = p;

return tkStrCon;

}

continue;

case 0:

if (p >= last)

{

errorType = (uint)ERRnoStrEnd;

ReturnScanError:

m\_currentCharacter = p - 1;

if (m\_fSyntaxColor)

{

\*pp = p - 1;

return ScanError(p - 1, tkStrCon);

}

Error(errorType);

}

else if (stringTemplateMode)

{

// Escaped null character is translated into 0x0030 for raw template literals

rawch = 0x0030;

}

break;

default:

if (IsMultiUnitChar(ch))

{

rawch = ch = ReadRest<true>(ch, p, last);

switch (ch)

{

case kchLS:

case kchPS:

goto LEcmaEscapeLineBreak;

}

}

break;

}

break;

}

m\_tempChBuf.AppendCh(ch);

m\_tempChBufSecondary.AppendCh<createRawString>(rawch);

}

LBreak:

bool createPid = true;

if (m\_fSyntaxColor || (m\_DeferredParseFlags & ScanFlagSuppressStrPid) != 0)

{

createPid = false;

if ((m\_tempChBuf.m\_ichCur == 10) && (0 == memcmp(L"use strict", m\_tempChBuf.m\_prgch, m\_tempChBuf.m\_ichCur \* sizeof(OLECHAR))))

{

createPid = true;

}

}

if (createPid)

{

m\_ptoken->SetIdentifier(m\_phtbl->PidHashNameLen(m\_tempChBuf.m\_prgch, m\_tempChBuf.m\_ichCur));

}

else

{

m\_ptoken->SetIdentifier(NULL);

}

m\_scanState = ScanStateNormal;

m\_doubleQuoteOnLastTkStrCon = '"' == delim;

\*pp = p;

return tkStrCon;

}

template<typename EncodingPolicy>

tokens Scanner<EncodingPolicy>::ScanStringConstant(OLECHAR delim, EncodedCharPtr \*pp)

{

return ScanStringConstant<false, false>(delim, pp);

}

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

\*

\* Consume a C-style comment.

\*/

template<typename EncodingPolicy>

tokens Scanner<EncodingPolicy>::SkipComment(EncodedCharPtr \*pp, /\* out \*/ bool\* containTypeDef)

{

Assert(containTypeDef != nullptr);

EncodedCharPtr p = \*pp;

\*containTypeDef = false;

EncodedCharPtr last = m\_pchLast;

OLECHAR ch;

for (;;)

{

switch((ch = ReadFirst(p, last)))

{

case '\*':

if (\*p == '/')

{

\*pp = p + 1;

if (m\_fSyntaxColor)

{

m\_scanState = ScanStateNormal;

return tkComment;

}

return tkNone;

}

break;

case kchLS: // 0x2028, classifies as new line

case kchPS: // 0x2029, classifies as new line

LEcmaLineBreak:

goto LLineBreak;

case kchRET:

case kchNWL:

LLineBreak:

m\_fHadEol = TRUE;

m\_currentCharacter = p;

ScanNewLine(ch);

p = m\_currentCharacter;

break;

case kchNUL:

if (p >= last)

{

m\_currentCharacter = p - 1;

\*pp = p - 1;

if (m\_fSyntaxColor)

{

m\_scanState = ScanStateMultiLineComment;

return tkComment;

}

Error(ERRnoCmtEnd);

}

break;

default:

if (IsMultiUnitChar(ch))

{

ch = ReadRest<true>(ch, p, last);

switch (ch)

{

case kchLS:

case kchPS:

goto LEcmaLineBreak;

}

}

break;

}

}

}

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

\*

\* We've encountered a newline - update various counters and things.

\*/

template<typename EncodingPolicy>

void Scanner<EncodingPolicy>::ScanNewLine(uint ch)

{

if (ch == '\r' && PeekNextChar() == '\n')

{

ReadNextChar();

}

NotifyScannedNewLine();

}

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

\*

\* We've encountered a newline - update various counters and things.

\*/

template<typename EncodingPolicy>

void Scanner<EncodingPolicy>::NotifyScannedNewLine()

{

// update in scanner: previous line, current line, number of lines.

m\_line++;

m\_pchPrevLine = m\_pchMinLine;

m\_pchMinLine = m\_currentCharacter;

m\_cMinLineMultiUnits = m\_cMultiUnits;

}

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

\*

\* Delivers a token stream.

\*/

template<typename EncodingPolicy>

tokens Scanner<EncodingPolicy>::ScanForcingPid()

{

if (m\_DeferredParseFlags != ScanFlagNone)

{

BYTE deferredParseFlagsSave = m\_DeferredParseFlags;

m\_DeferredParseFlags = ScanFlagNone;

tokens result = tkEOF;

\_\_try

{

result = Scan();

}

\_\_finally

{

m\_DeferredParseFlags = deferredParseFlagsSave;

}

return result;

}

return Scan();

}

template<typename EncodingPolicy>

tokens Scanner<EncodingPolicy>::Scan()

{

return ScanCore(true);

}

template<typename EncodingPolicy>

tokens Scanner<EncodingPolicy>::ScanNoKeywords()

{

return ScanCore(false);

}

template<typename EncodingPolicy>

tokens Scanner<EncodingPolicy>::ScanAhead()

{

return ScanNoKeywords();

}

template<typename EncodingPolicy>

tokens Scanner<EncodingPolicy>::ScanCore(bool identifyKwds)

{

codepoint\_t ch;

OLECHAR firstChar;

OLECHAR secondChar;

EncodedCharPtr pchT;

size\_t multiUnits = 0;

EncodedCharPtr p = m\_currentCharacter;

EncodedCharPtr last = m\_pchLast;

// store the last token

m\_tkPrevious = m\_ptoken->tk;

m\_iecpLimTokPrevious = IecpLimTok(); // Introduced for use by lambda parsing to find correct span of expression lambdas

if (p >= last)

{

m\_pchMinTok = p;

m\_cMinTokMultiUnits = m\_cMultiUnits;

goto LEof;

}

tokens token;

m\_fHadEol = FALSE;

CharTypes chType;

charcount\_t commentStartLine;

if (m\_scanState && \*p != 0)

{

if (m\_fSyntaxColor)

{

firstChar = 0;

secondChar = 0;

m\_pchMinTok = p;

m\_cMinTokMultiUnits = m\_cMultiUnits;

switch (m\_scanState)

{

case ScanStateMultiLineComment:

goto LMultiLineComment;

case ScanStateMultiLineSingleQuoteString:

ch = '\'';

m\_scanState = ScanStateNormal;

goto LScanStringConstant;

case ScanStateMultiLineDoubleQuoteString:

ch = '"';

m\_scanState = ScanStateNormal;

goto LScanStringConstant;

}

}

if (m\_scanState == ScanStateStringTemplateMiddleOrEnd)

{

AssertMsg(m\_fStringTemplateDepth > 0,

"Shouldn't be trying to parse a string template end or middle token if we aren't scanning a string template");

AssertMsg(m\_scriptContext->GetConfig()->IsES6StringTemplateEnabled(),

"Shouldn't be in string template parse mode if string templates are not enabled.");

m\_scanState = ScanStateNormal;

pchT = p;

token = ScanStringTemplateMiddleOrEnd(&pchT);

p = pchT;

goto LDone;

}

}

for (;;)

{

LLoop:

m\_pchMinTok = p;

m\_cMinTokMultiUnits = m\_cMultiUnits;

ch = ReadFirst(p, last);

#if DEBUG

chType = this->charClassifier->GetCharType((OLECHAR)ch);

#endif

switch (ch)

{

default:

LLoopDefault:

if (ch == kchLS ||

ch == kchPS )

{

goto LNewLine;

}

{

BOOL isMultiUnit = IsMultiUnitChar((OLECHAR)ch);

if (isMultiUnit)

{

ch = ReadRest<true>((OLECHAR)ch, p, last);

}

if (es6UnicodeMode && Js::NumberUtilities::IsSurrogateLowerPart(ch))

{

codepoint\_t upper = PeekFull(p, last);

if (Js::NumberUtilities::IsSurrogateUpperPart(upper))

{

ch = Js::NumberUtilities::SurrogatePairAsCodePoint(ch, upper);

ReadFull<true>(p, last);

}

}

if (this->charClassifier->IsIdStart(ch))

{

// We treat IDContinue as an error.

token = ScanIdentifierContinue(identifyKwds, false, !!isMultiUnit, m\_pchMinTok, p, &p);

break;

}

}

chType = this->charClassifier->GetCharType(ch);

switch (chType)

{

case \_C\_WSP: continue;

case \_C\_NWL: goto LNewLine;

// All other types (except errors) are handled by the outer switch.

}

Assert(chType == \_C\_LET || chType == \_C\_ERR || chType == \_C\_UNK || chType == \_C\_BKQ || chType == \_C\_SHP || chType == \_C\_AT || chType == \_C\_DIG);

if (m\_fSyntaxColor)

{

// No need to decrement the current position pointer as scanner will continue with scan next character onwards

return ScanError(p, tkID);

}

m\_currentCharacter = p - 1;

Error(ERRillegalChar);

continue;

case '\0':

// Put back the null in case we get called again.

p--;

LEof:

token = tkEOF;

if (p + 1 < last)

{

if (m\_fSyntaxColor)

{

return ScanError(p + 1, tkID);

}

// A \0 prior to the end of the text is an invalid character.

Error(ERRillegalChar);

}

break;

case 0x0009:

case 0x000B:

case 0x000C:

case 0x0020:

Assert(chType == \_C\_WSP);

continue;

case '.':

if (!Js::NumberUtilities::IsDigit(\*p))

{

// Not a double

if (m\_scriptContext->GetConfig()->IsES6SpreadEnabled() && PeekFirst(p, last) == '.' && PeekFirst(p + 1, last) == '.')

{

token = tkEllipsis;

p += 2;

}

else

{

token = tkDot;

}

break;

}

// May be a double, fall through

case '0': case '1': case '2': case '3': case '4':

case '5': case '6': case '7': case '8': case '9':

{

double dbl;

Assert(chType == \_C\_DIG || chType == \_C\_DOT);

p = m\_pchMinTok;

RestoreMultiUnits(m\_cMinTokMultiUnits);

bool likelyInt = true;

pchT = FScanNumber(p, &dbl, likelyInt);

if (p == pchT)

{

Assert(PeekFirst(p, last) != '.');

if (m\_fSyntaxColor)

{

return ScanError(m\_currentCharacter + 1, tkFltCon);

}

Error(ERRbadNumber);

}

Assert(!Js::NumberUtilities::IsNan(dbl));

p = pchT;

long value;

if (likelyInt && Js::NumberUtilities::FDblIsLong(dbl, &value))

{

m\_ptoken->SetLong(value);

token = tkIntCon;

}

else

{

token = tkFltCon;

m\_ptoken->SetDouble(dbl, likelyInt);

}

break;

}

case '(': Assert(chType == \_C\_LPR); token = tkLParen; break;

case ')': Assert(chType == \_C\_RPR); token = tkRParen; break;

case ',': Assert(chType == \_C\_CMA); token = tkComma; break;

case ';': Assert(chType == \_C\_SMC); token = tkSColon; break;

case '[': Assert(chType == \_C\_LBR); token = tkLBrack; break;

case ']': Assert(chType == \_C\_RBR); token = tkRBrack; break;

case '~': Assert(chType == \_C\_TIL); token = tkTilde; break;

case '?': Assert(chType == \_C\_QUE); token = tkQMark; break;

case '{': Assert(chType == \_C\_LC); token = tkLCurly; break;

case '\r':

case '\n':

LNewLine:

m\_currentCharacter = p;

ScanNewLine(ch);

p = m\_currentCharacter;

m\_fHadEol = TRUE;

continue;

LReserved:

{

// We will derive the PID from the token

Assert(token < tkID);

m\_ptoken->SetIdentifier(NULL);

goto LDone;

}

LEval:

{

token = tkID;

if (!this->m\_parser) goto LIdentifier;

m\_ptoken->SetIdentifier(this->m\_parser->GetEvalPid());

goto LDone;

}

LArguments:

{

token = tkID;

if (!this->m\_parser) goto LIdentifier;

m\_ptoken->SetIdentifier(this->m\_parser->GetArgumentsPid());

goto LDone;

}

LTarget:

{

token = tkID;

if (!this->m\_parser) goto LIdentifier;

m\_ptoken->SetIdentifier(this->m\_parser->GetTargetPid());

goto LDone;

}

#include "kwd-swtch.h"

case 'A': case 'B': case 'C': case 'D': case 'E':

case 'F': case 'G': case 'H': case 'I': case 'J':

case 'K': case 'L': case 'M': case 'N': case 'O':

case 'P': case 'Q': case 'R': case 'S': case 'T':

case 'U': case 'V': case 'W': case 'X': case 'Y':

case 'Z':

// Lower-case letters handled in kwd-swtch.h above during reserved word recognition.

case '$': case '\_':

LIdentifier:

Assert(this->charClassifier->IsIdStart(ch));

Assert(ch < 0x10000 && !IsMultiUnitChar((OLECHAR)ch));

token = ScanIdentifierContinue(identifyKwds, false, false, m\_pchMinTok, p, &p);

break;

case '`':

Assert(chType == \_C\_BKQ);

if (m\_scriptContext->GetConfig()->IsES6StringTemplateEnabled())

{

pchT = p;

token = ScanStringTemplateBegin(&pchT);

p = pchT;

}

else

{

goto LLoopDefault;

}

break;

case '}':

Assert(chType == \_C\_RC);

token = tkRCurly;

break;

case '\\':

pchT = p - 1;

token = ScanIdentifier(identifyKwds, &pchT);

if (tkScanError == token)

{

m\_currentCharacter = p;

if (m\_fSyntaxColor)

return ScanError(p, tkID);

Error(ERRillegalChar);

}

p = pchT;

break;

case ':':

token = tkColon;

break;

case '=':

token = tkAsg;

switch (PeekFirst(p, last))

{

case '=':

p++;

token = tkEQ;

if (PeekFirst(p, last) == '=')

{

p++;

token = tkEqv;

}

break;

case '>':

if (m\_scriptContext->GetConfig()->IsES6LambdaEnabled())

{

p++;

token = tkDArrow;

}

break;

}

break;

case '!':

token = tkBang;

if (PeekFirst(p, last) == '=')

{

p++;

token = tkNE;

if (PeekFirst(p, last) == '=')

{

p++;

token = tkNEqv;

}

}

break;

case '+':

token = tkAdd;

switch (PeekFirst(p, last))

{

case '=':

p++;

token = tkAsgAdd;

break;

case '+':

p++;

token = tkInc;

break;

}

break;

case '-':

token = tkSub;

switch (PeekFirst(p, last))

{

case '=':

p++;

token = tkAsgSub;

break;

case '-':

p++;

token = tkDec;

if (m\_fHtmlComments)

{

int i = 0;

while ('-' == PeekFirst(p + i, last)) //Have already seen --, skip any further - characters

i++;

if ('>' == PeekFirst(p + i++, last)) //This means we've got a --------------------------->.

{

//If that precedes an EOF or }NWL (disregarding whitespace), then it is a comment.

OLECHAR nextChar;

nextChar = NextNonWhiteChar(&p[i], last);

if (nextChar == 0)

{

//Treat the -----------------------------> EOF as if it were EOF

token = tkEOF;

++p;

}

else if (nextChar == '}')

{

CharTypes nextNextCharType = this->charClassifier->GetCharType(NextNonWhiteCharPlusOne(&p[i], last));

if (nextNextCharType == \_C\_NWL

// Corner case: If we have reached the end of the source, either we are at the end of the file or the end of

// a deferred function. We treat this case as NWL.

// TODO(tcare): Update to ES6 spec. Tracked in Bug 1164686

|| (last == m\_pchLast && nextNextCharType == \_C\_NUL))

{

//Treat the -----------------------------> }NWL as if it were }NWL

p += i;

continue;

}

}

}

}

break;

}

break;

case '\*':

token = tkStar;

switch(PeekFirst(p, last))

{

case '=' :

p++;

token = tkAsgMul;

break;

case '\*' :

if (!m\_scriptContext->GetConfig()->IsES7ExponentiationOperatorEnabled())

{

break;

}

p++;

token = tkExpo;

if (PeekFirst(p, last) == '=')

{

p++;

token = tkAsgExpo;

}

}

break;

case '/':

token = tkDiv;

switch(PeekFirst(p, last))

{

case '=':

p++;

token = tkAsgDiv;

break;

case '/':

if (p >= last)

{

AssertMsg(m\_fHtmlComments, "Do we have other line comment cases scanning pass last?");

// Effective source length may have excluded HTMLCommentSuffix "//... -->". If we are scanning

// those, we have passed "last" already. Move back and return EOF.

p = last;

goto LEof;

}

ch = \*++p;

firstChar = (OLECHAR)ch;

LSkipLineComment:

pchT = NULL;

for (;;)

{

switch ((ch = ReadFirst(p, last)))

{

case kchLS: // 0x2028, classifies as new line

case kchPS: // 0x2029, classifies as new line

LEcmaCommentLineBreak:

// kchPS and kchLS are more than one unit in UTF-8.

if (pchT)

{

// kchPS and kchLS are more than one unit in UTF-8.

p = pchT;

}

else

{

// But only a single code unit in UTF16

p--;

}

RestoreMultiUnits(multiUnits);

goto LCommentLineBreak;

case kchNWL:

case kchRET:

p--;

LCommentLineBreak:

if (m\_fSyntaxColor)

{

token = tkComment;

goto LDone;

}

// Subtract the comment length from the total char count for the purpose

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

m\_parser->ReduceDeferredScriptLength((ULONG)(p - m\_pchMinTok));

break;

case kchNUL:

if (p >= last)

{

p--;

goto LCommentLineBreak;

}

continue;

default:

if (IsMultiUnitChar((OLECHAR)ch))

{

pchT = p - 1;

multiUnits = m\_cMultiUnits;

switch (ch = ReadRest<true>((OLECHAR)ch, p, last))

{

case kchLS:

case kchPS:

goto LEcmaCommentLineBreak;

}

}

continue;

}

break;

}

continue;

case '\*':

ch = \*++p;

firstChar = (OLECHAR)ch;

if ((p + 1) < last)

{

secondChar = (OLECHAR)(\*(p + 1));

}

else

{

secondChar = '\0';

}

LMultiLineComment:

pchT = p;

commentStartLine = m\_line;

bool containTypeDef;

if (tkNone == (token = SkipComment(&pchT, &containTypeDef)))

{

// Subtract the comment length from the total char count for the purpose

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

m\_parser->ReduceDeferredScriptLength((ULONG)(pchT - m\_pchMinTok));

p = pchT;

goto LLoop;

}

p = pchT;

break;

}

break;

case '%':

Assert(chType == \_C\_PCT);

token = tkPct;

if (PeekFirst(p, last) == '=')

{

p++;

token = tkAsgMod;

}

break;

case '<':

Assert(chType == \_C\_LT);

token = tkLT;

switch (PeekFirst(p, last))

{

case '=':

p++;

token = tkLE;

break;

case '<':

p++;

token = tkLsh;

if (PeekFirst(p, last) == '=')

{

p++;

token = tkAsgLsh;

break;

}

break;

case '!':

if (m\_fHtmlComments && PeekFirst(p + 1, last) == '-' && PeekFirst(p + 2, last) == '-')

{

// This is a "<!--" comment - treat as //

if (p >= last)

{

// Effective source length may have excluded HTMLCommentSuffix "<!-- ... -->". If we are scanning

// those, we have passed "last" already. Move back and return EOF.

p = last;

goto LEof;

}

firstChar = '!';

goto LSkipLineComment;

}

break;

}

break;

case '>':

Assert(chType == \_C\_GT);

token = tkGT;

switch (PeekFirst(p, last))

{

case '=':

p++;

token = tkGE;

break;

case '>':

p++;

token = tkRsh;

switch (PeekFirst(p, last))

{

case '=':

p++;

token = tkAsgRsh;

break;

case '>':

p++;

token = tkRs2;

if (\*p == '=')

{

p++;

token = tkAsgRs2;

}

break;

}

break;

}

break;

case '^':

Assert(chType == \_C\_XOR);

token = tkXor;

if (PeekFirst(p, last) == '=')

{

p++;

token = tkAsgXor;

}

break;

case '|':

Assert(chType == \_C\_BAR);

token = tkOr;

switch (PeekFirst(p, last))

{

case '=':

p++;

token = tkAsgOr;

break;

case '|':

p++;

token = tkLogOr;

break;

}

break;

case '&':

Assert(chType == \_C\_AMP);

token = tkAnd;

switch (PeekFirst(p, last))

{

case '=':

p++;

token = tkAsgAnd;

break;

case '&':

p++;

token = tkLogAnd;

break;

}

break;

case '\'':

case '"':

Assert(chType == \_C\_QUO || chType == \_C\_APO);

LScanStringConstant:

pchT = p;

token = ScanStringConstant((OLECHAR)ch, &pchT);

p = pchT;

break;

}

break;

}

LDone:

m\_currentCharacter = p;

return (m\_ptoken->tk = token);

}

template <typename EncodingPolicy>

IdentPtr Scanner<EncodingPolicy>::GetSecondaryBufferAsPid()

{

bool createPid = true;

if (m\_fSyntaxColor || (m\_DeferredParseFlags & ScanFlagSuppressStrPid) != 0)

{

createPid = false;

}

if (createPid)

{

return m\_phtbl->PidHashNameLen(m\_tempChBufSecondary.m\_prgch, m\_tempChBufSecondary.m\_ichCur);

}

else

{

return nullptr;

}

}

template <typename EncodingPolicy>

LPCOLESTR Scanner<EncodingPolicy>::StringFromLong(long lw)

{

\_ltow\_s(lw, m\_tempChBuf.m\_prgch, m\_tempChBuf.m\_cchMax, 10);

return m\_tempChBuf.m\_prgch;

}

template <typename EncodingPolicy>

IdentPtr Scanner<EncodingPolicy>::PidFromLong(long lw)

{

return m\_phtbl->PidHashName(StringFromLong(lw));

}

template <typename EncodingPolicy>

LPCOLESTR Scanner<EncodingPolicy>::StringFromDbl(double dbl)

{

if (!Js::NumberUtilities::FDblToStr(dbl, m\_tempChBuf.m\_prgch, m\_tempChBuf.m\_cchMax))

{

Error(ERRnoMemory);

}

return m\_tempChBuf.m\_prgch;

}

template <typename EncodingPolicy>

IdentPtr Scanner<EncodingPolicy>::PidFromDbl(double dbl)

{

return m\_phtbl->PidHashName(StringFromDbl(dbl));

}

template <typename EncodingPolicy>

void Scanner<EncodingPolicy>::Capture(\_Out\_ RestorePoint\* restorePoint)

{

Capture(restorePoint, 0, 0);

}

template <typename EncodingPolicy>

void Scanner<EncodingPolicy>::Capture(\_Out\_ RestorePoint\* restorePoint, uint functionIdIncrement, size\_t lengthDecr)

{

restorePoint->m\_ichMinTok = this->IchMinTok();

restorePoint->m\_ichMinLine = this->IchMinLine();

restorePoint->m\_cMinTokMultiUnits = this->m\_cMinTokMultiUnits;

restorePoint->m\_cMinLineMultiUnits = this->m\_cMinLineMultiUnits;

restorePoint->m\_line = this->m\_line;

restorePoint->m\_fHadEol = this->m\_fHadEol;

restorePoint->functionIdIncrement = functionIdIncrement;

restorePoint->lengthDecr = lengthDecr;

#ifdef DEBUG

restorePoint->m\_cMultiUnits = this->m\_cMultiUnits;

#endif

}

template <typename EncodingPolicy>

void Scanner<EncodingPolicy>::SeekTo(const RestorePoint& restorePoint)

{

this->m\_currentCharacter = this->m\_pchBase + restorePoint.m\_ichMinTok + restorePoint.m\_cMinTokMultiUnits;

this->m\_pchMinLine = this->m\_pchBase + restorePoint.m\_ichMinLine + restorePoint.m\_cMinLineMultiUnits;

this->m\_cMinLineMultiUnits = restorePoint.m\_cMinLineMultiUnits;

this->RestoreMultiUnits(restorePoint.m\_cMinTokMultiUnits);

this->Scan();

this->m\_line = restorePoint.m\_line;

this->m\_fHadEol = restorePoint.m\_fHadEol;

this->m\_parser->ReduceDeferredScriptLength(restorePoint.lengthDecr);

Assert(this->m\_cMultiUnits == restorePoint.m\_cMultiUnits);

}

template <typename EncodingPolicy>

void Scanner<EncodingPolicy>::SeekTo(const RestorePoint& restorePoint, uint \*nextFunctionId)

{

SeekTo(restorePoint);

\*nextFunctionId += restorePoint.functionIdIncrement;

}

// Called by CompileScriptException::ProcessError to retrieve a BSTR for the line on which an error occurred.

template<typename EncodingPolicy>

HRESULT Scanner<EncodingPolicy>::SysAllocErrorLine(long ichMinLine, \_\_out BSTR\* pbstrLine)

{

if( !pbstrLine )

{

return E\_POINTER;

}

// If we overflow the string, we have a serious problem...

if (ichMinLine < 0 || static\_cast<size\_t>(ichMinLine) > AdjustedLength() )

{

return E\_UNEXPECTED;

}

EncodedCharPtr pStart = static\_cast<size\_t>(ichMinLine) == IchMinLine() ? m\_pchMinLine : m\_pchBase + CharacterOffsetToUnitOffset(m\_pchBase, m\_currentCharacter, m\_pchLast, ichMinLine);

EncodedCharPtr pEnd = AdjustedLast();

// Determine the length by scanning for the next newline

charcount\_t cch = LineLength(pStart, pEnd);

Assert(cch <= LONG\_MAX);

\*pbstrLine = SysAllocStringLen(NULL, cch);

if (!\*pbstrLine)

{

return E\_OUTOFMEMORY;

}

ConvertToUnicode(\*pbstrLine, cch, pStart);

return S\_OK;

}

template class Scanner<NullTerminatedUnicodeEncodingPolicy>;

template class Scanner<NullTerminatedUTF8EncodingPolicy>;

template class Scanner<NotNullTerminatedUTF8EncodingPolicy>;