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

// 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"

#if DEBUG

#define DEBUG\_TRASHMEM(pv, cb) memset(pv, 0xbc, cb)

#else

#define DEBUG\_TRASHMEM

#endif //DEBUG

#if \_WIN64

struct \_\_ALIGN\_FOO\_\_ {

int w1;

double dbl;

};

#define ALIGN\_FULL (offsetof(\_\_ALIGN\_FOO\_\_, dbl))

#else

// Force check for 4 byte alignment to support Win98/ME

#define ALIGN\_FULL 4

#endif // \_WIN64

#define AlignFull(VALUE) (~(~((VALUE) + (ALIGN\_FULL-1)) | (ALIGN\_FULL-1)))

NoReleaseAllocator::NoReleaseAllocator(long cbFirst, long cbMax)

: m\_pblkList(NULL)

, m\_ibCur(0)

, m\_ibMax(0)

, m\_cbMinBlock(cbFirst)

, m\_cbMaxBlock(cbMax)

#if DEBUG

, m\_cbTotRequested(0)

, m\_cbTotAlloced(0)

, m\_cblk(0)

, m\_cpvBig(0)

, m\_cpvSmall(0)

#endif

{

// require reasonable ranges

Assert((0 < cbFirst) && (cbFirst < SHRT\_MAX/2));

Assert((0 < cbMax ) && (cbMax < SHRT\_MAX));

}

void \* NoReleaseAllocator::Alloc(long cb)

{

Assert(cb > 0);

if (cb <= 0)

return NULL;

const long kcbHead = AlignFull(sizeof(NoReleaseAllocator::NraBlock));

void \* pv;

if (cb > m\_ibMax - m\_ibCur)

{

long cbBlock;

long cbAlloc;

NraBlock \* pblk;

if (cb >= m\_cbMaxBlock)

{

// check for integer overflow before allocating (See WindowsSE #88972)

cbAlloc = cb + kcbHead;

if (cbAlloc < cb)

{

Assert(FALSE); // too big!

return NULL;

}

// create a chunk just for this allocation

pblk = (NraBlock \*)malloc(cbAlloc);

if (NULL == pblk)

return NULL;

#if DEBUG

m\_cbTotAlloced += cbAlloc;

m\_cbTotRequested += cb;

m\_cpvBig++;

m\_cblk++;

#endif //DEBUG

if (m\_ibCur < m\_ibMax)

{

// There is still room in current block, so put the new block

// after the current block.

pblk->pblkNext = m\_pblkList->pblkNext;

m\_pblkList->pblkNext = pblk;

}

else

{

// Link into front of the list.

// Don't need to adjust m\_ibCur and m\_ibMax, because they

// already have the correct relationship for this full block

// (m\_ibCur >= m\_ibMax) and the actual values will not be

// used.

pblk->pblkNext = m\_pblkList;

m\_pblkList = pblk;

}

DEBUG\_TRASHMEM((byte \*)pblk + kcbHead, cb);

return (byte \*)pblk + kcbHead;

}

cbBlock = cb; // requested size

if (m\_ibMax > cbBlock) // at least current block size

cbBlock = m\_ibMax;

cbBlock += cbBlock; // \*2 (can overflow, but checked below)

if (m\_cbMinBlock > cbBlock) // at least minimum size

cbBlock = m\_cbMinBlock;

if (cbBlock > m\_cbMaxBlock) // no larger than the max

cbBlock = m\_cbMaxBlock;

if (cb > cbBlock) // guarantee it's big enough

{

Assert(("Request too large", FALSE));

return NULL;

}

// check for integer overflow before allocating (See WindowsSE #88972)

cbAlloc = cbBlock + kcbHead;

if ((cbAlloc < cbBlock) || (cbAlloc < cb))

{

Assert(FALSE); // too big!

return NULL ;

}

// allocate a new block

pblk = (NraBlock \*)malloc(cbAlloc);

#ifdef MEM\_TRACK

RegisterAlloc((char\*)pblk,cbAlloc);

#endif

if (NULL == pblk)

return NULL;

#if DEBUG

m\_cbTotAlloced += cbAlloc;

m\_cblk++;

#endif //DEBUG

pblk->pblkNext = m\_pblkList;

m\_pblkList = pblk;

m\_ibMax = cbBlock;

m\_ibCur = 0;

}

Assert(m\_ibCur + cb <= m\_ibMax);

#if DEBUG

m\_cbTotRequested += cb;

m\_cpvSmall++;

#endif //DEBUG

pv = (byte \*)m\_pblkList + kcbHead + m\_ibCur;

DEBUG\_TRASHMEM(pv, cb);

m\_ibCur += (long)AlignFull(cb);

Assert(m\_ibCur >= 0);

return pv;

}

void NoReleaseAllocator::FreeAll(void)

{

// Free all of the allocated blocks

while (NULL != m\_pblkList)

{

NraBlock \* pblk = m\_pblkList;

#pragma prefast(suppress:6001, "Not sure why it is complaining \*m\_plkList is uninitialized")

m\_pblkList = pblk->pblkNext;

free(pblk);

}

// prepare for next round of allocations

m\_ibCur = m\_ibMax = 0;

#if DEBUG

m\_cbTotRequested = 0;

m\_cbTotAlloced = 0;

m\_cblk = 0;

m\_cpvBig = 0;

m\_cpvSmall = 0;

#endif

}

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

// 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

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

NoReleaseAllocator - allocator that never releases until it is destroyed

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

class NoReleaseAllocator

{

public:

NoReleaseAllocator(long cbFirst = 256, long cbMax = 0x4000 /\*16K\*/);

~NoReleaseAllocator(void) { FreeAll(); }

void \*Alloc(long cb);

void FreeAll();

void Clear() { FreeAll(); }

private:

struct NraBlock

{

NraBlock \* pblkNext;

// ... DATA ...

};

NraBlock \* m\_pblkList;

long m\_ibCur;

long m\_ibMax;

long m\_cbMinBlock;

long m\_cbMaxBlock;

#if DEBUG

long m\_cbTotRequested; // total bytes requested

long m\_cbTotAlloced; // total bytes allocated including headers

long m\_cblk; // number of blocks including big blocks

long m\_cpvBig; // each generates its own big block

long m\_cpvSmall; // put in a common block

#endif //DEBUG

};

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

// 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"

#define ASSERT\_THREAD() AssertMsg(mainThreadId == GetCurrentThreadContextId(), \

"Cannot use this member of BackgroundParser from thread other than the creating context's current thread")

#if ENABLE\_NATIVE\_CODEGEN

BackgroundParser::BackgroundParser(Js::ScriptContext \*scriptContext)

: JsUtil::WaitableJobManager(scriptContext->GetThreadContext()->GetJobProcessor()),

scriptContext(scriptContext),

unprocessedItemsHead(nullptr),

unprocessedItemsTail(nullptr),

failedBackgroundParseItem(nullptr),

pendingBackgroundItems(0)

{

Processor()->AddManager(this);

#if DBG

this->mainThreadId = GetCurrentThreadContextId();

#endif

}

BackgroundParser::~BackgroundParser()

{

JsUtil::JobProcessor \*processor = Processor();

if (processor->ProcessesInBackground())

{

static\_cast<JsUtil::BackgroundJobProcessor\*>(processor)->IterateBackgroundThreads([&](JsUtil::ParallelThreadData \*threadData)->bool {

if (threadData->parser)

{

threadData->parser->Release();

threadData->parser = nullptr;

}

return false;

});

}

processor->RemoveManager(this);

}

BackgroundParser \* BackgroundParser::New(Js::ScriptContext \*scriptContext)

{

return HeapNew(BackgroundParser, scriptContext);

}

void BackgroundParser::Delete(BackgroundParser \*backgroundParser)

{

HeapDelete(backgroundParser);

}

bool BackgroundParser::Process(JsUtil::Job \*const job, JsUtil::ParallelThreadData \*threadData)

{

BackgroundParseItem \*backgroundItem = static\_cast<BackgroundParseItem\*>(job);

if (failedBackgroundParseItem)

{

if (backgroundItem->GetParseNode()->ichMin > failedBackgroundParseItem->GetParseNode()->ichMin)

{

return true;

}

}

if (threadData->parser == nullptr || threadData->canDecommit)

{

if (threadData->parser != nullptr)

{

// "canDecommit" means the previous parse finished.

// Don't leave a parser with stale state in the thread data, or we'll mess up the bindings.

threadData->backgroundPageAllocator.DecommitNow();

this->OnDecommit(threadData);

}

threadData->canDecommit = false;

// Lazily create a parser instance for this thread from the thread's page allocator.

// It will stay around until the main thread's current parser instance goes away, which will free

// the background thread to decommit its pages.

threadData->parser = Anew(threadData->threadArena, Parser, this->scriptContext, backgroundItem->IsStrictMode(), &threadData->backgroundPageAllocator, true);

threadData->pse = Anew(threadData->threadArena, CompileScriptException);

threadData->parser->PrepareScanner(backgroundItem->GetParseContext()->fromExternal);

}

Parser \*parser = threadData->parser;

return this->Process(backgroundItem, parser, threadData->pse);

}

bool BackgroundParser::Process(JsUtil::Job \*const job, Parser \*parser, CompileScriptException \*pse)

{

BackgroundParseItem \*backgroundItem = static\_cast<BackgroundParseItem\*>(job);

Assert(parser->GetCurrBackgroundParseItem() == nullptr);

parser->SetCurrBackgroundParseItem(backgroundItem);

backgroundItem->SetParser(parser);

HRESULT hr = parser->ParseFunctionInBackground(backgroundItem->GetParseNode(), backgroundItem->GetParseContext(), backgroundItem->IsDeferred(), pse);

backgroundItem->SetMaxBlockId(parser->GetLastBlockId());

backgroundItem->SetHR(hr);

if (FAILED(hr))

{

backgroundItem->SetPSE(pse);

}

backgroundItem->SetCompleted(true);

parser->SetCurrBackgroundParseItem(nullptr);

return hr == S\_OK;

}

void BackgroundParser::JobProcessed(JsUtil::Job \*const job, const bool succeeded)

{

// This is called from inside a lock, so we can mess with background parser attributes.

BackgroundParseItem \*backgroundItem = static\_cast<BackgroundParseItem\*>(job);

this->RemoveFromUnprocessedItems(backgroundItem);

--this->pendingBackgroundItems;

if (!succeeded)

{

Assert(FAILED(backgroundItem->GetHR()) || failedBackgroundParseItem);

if (FAILED(backgroundItem->GetHR()))

{

if (!failedBackgroundParseItem)

{

failedBackgroundParseItem = backgroundItem;

}

else

{

// If syntax errors are detected on multiple threads, the lexically earlier one should win.

CompileScriptException \*newPse = backgroundItem->GetPSE();

CompileScriptException \*oldPse = failedBackgroundParseItem->GetPSE();

if (newPse->line < oldPse->line ||

(newPse->line == oldPse->line && newPse->ichMinLine < oldPse->ichMinLine))

{

failedBackgroundParseItem = backgroundItem;

}

}

}

}

}

void BackgroundParser::OnDecommit(JsUtil::ParallelThreadData \*threadData)

{

if (threadData->parser)

{

threadData->parser->Release();

threadData->parser = nullptr;

}

}

BackgroundParseItem \* BackgroundParser::NewBackgroundParseItem(Parser \*parser, ParseNode \*parseNode, bool isDeferred)

{

BackgroundParseItem \*item = Anew(parser->GetAllocator(), BackgroundParseItem, this, parser, parseNode, isDeferred);

parser->AddBackgroundParseItem(item);

return item;

}

bool BackgroundParser::ParseBackgroundItem(Parser \*parser, ParseNode \*parseNode, bool isDeferred)

{

ASSERT\_THREAD();

AutoPtr<BackgroundParseItem> workItemAutoPtr(this->NewBackgroundParseItem(parser, parseNode, isDeferred));

if ((BackgroundParseItem\*) workItemAutoPtr == nullptr)

{

// OOM, just skip this work item and return.

// TODO: Raise an OOM parse-time exception.

return false;

}

parser->PrepareForBackgroundParse();

BackgroundParseItem \* backgroundItem = workItemAutoPtr.Detach();

this->AddToParseQueue(backgroundItem, false, this->Processor()->ProcessesInBackground());

return true;

}

BackgroundParseItem \*BackgroundParser::GetJob(BackgroundParseItem \*workitem) const

{

return workitem;

}

bool BackgroundParser::WasAddedToJobProcessor(JsUtil::Job \*const job) const

{

ASSERT\_THREAD();

Assert(job);

return static\_cast<BackgroundParseItem\*>(job)->IsInParseQueue();

}

void BackgroundParser::BeforeWaitForJob(BackgroundParseItem \*const item) const

{

}

void BackgroundParser::AfterWaitForJob(BackgroundParseItem \*const item) const

{

}

void BackgroundParser::AddToParseQueue(BackgroundParseItem \*const item, bool prioritize, bool lock)

{

AutoOptionalCriticalSection autoLock(lock ? Processor()->GetCriticalSection() : nullptr);

++this->pendingBackgroundItems;

Processor()->AddJob(item, prioritize); // This one can throw (really unlikely though), OOM specifically.

this->AddUnprocessedItem(item);

item->OnAddToParseQueue();

}

void BackgroundParser::AddUnprocessedItem(BackgroundParseItem \*const item)

{

if (this->unprocessedItemsTail == nullptr)

{

this->unprocessedItemsHead = item;

}

else

{

this->unprocessedItemsTail->SetNextUnprocessedItem(item);

}

item->SetPrevUnprocessedItem(this->unprocessedItemsTail);

this->unprocessedItemsTail = item;

}

void BackgroundParser::RemoveFromUnprocessedItems(BackgroundParseItem \*const item)

{

if (this->unprocessedItemsHead == item)

{

this->unprocessedItemsHead = item->GetNextUnprocessedItem();

}

else

{

item->GetPrevUnprocessedItem()->SetNextUnprocessedItem(item->GetNextUnprocessedItem());

}

if (this->unprocessedItemsTail == item)

{

this->unprocessedItemsTail = item->GetPrevUnprocessedItem();

}

else

{

item->GetNextUnprocessedItem()->SetPrevUnprocessedItem(item->GetPrevUnprocessedItem());

}

item->SetNextUnprocessedItem(nullptr);

item->SetPrevUnprocessedItem(nullptr);

}

BackgroundParseItem \*BackgroundParser::GetNextUnprocessedItem() const

{

BackgroundParseItem \*item;

bool background = this->Processor()->ProcessesInBackground();

for (item = this->unprocessedItemsHead; item; item = item->GetNextUnprocessedItem())

{

if (!background || !static\_cast<JsUtil::BackgroundJobProcessor\*>(Processor())->IsBeingProcessed(item))

{

return item;

}

}

return nullptr;

}

BackgroundParseItem::BackgroundParseItem(JsUtil::JobManager \*const manager, Parser \*const parser, ParseNode \*parseNode, bool defer)

: JsUtil::Job(manager),

maxBlockId((uint)-1),

strictMode(parser->IsStrictMode()),

parseNode(parseNode),

parser(nullptr),

nextItem(nullptr),

nextUnprocessedItem(nullptr),

prevUnprocessedItem(nullptr),

pse(nullptr),

regExpNodes(nullptr),

completed(false),

inParseQueue(false),

isDeferred(defer)

{

parser->CaptureContext(&parseContext);

}

void BackgroundParseItem::OnAddToParseQueue()

{

this->inParseQueue = true;

}

void BackgroundParseItem::OnRemoveFromParseQueue()

{

this->inParseQueue = false;

}

void BackgroundParseItem::AddRegExpNode(ParseNode \*const pnode, ArenaAllocator \*alloc)

{

if (regExpNodes == nullptr)

{

regExpNodes = Anew(alloc, NodeDList, alloc);

}

regExpNodes->Append(pnode);

}

#endif

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

// 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

#if ENABLE\_NATIVE\_CODEGEN

typedef DList<ParseNode\*, ArenaAllocator> NodeDList;

struct BackgroundParseItem sealed : public JsUtil::Job

{

BackgroundParseItem(JsUtil::JobManager \*const manager, Parser \*const parser, ParseNode \*parseNode, bool defer);

ParseContext \*GetParseContext() { return &parseContext; }

ParseNode \*GetParseNode() const { return parseNode; }

CompileScriptException \*GetPSE() const { return pse; }

HRESULT GetHR() const { return hr; }

bool IsStrictMode() const { return strictMode; }

bool Succeeded() const { return hr == S\_OK; }

bool IsInParseQueue() const { return inParseQueue; }

bool IsDeferred() const { return isDeferred;}

void SetHR(HRESULT hr) { this->hr = hr; }

void SetCompleted(bool has) { completed = has; }

void SetPSE(CompileScriptException \*pse) { this->pse = pse; }

uint GetMaxBlockId() const { return maxBlockId; }

void SetMaxBlockId(uint blockId) { maxBlockId = blockId; }

Parser \*GetParser() const { return parser; }

void SetParser(Parser \*p) { parser = p; }

BackgroundParseItem \*GetNext() const { return nextItem; }

void SetNext(BackgroundParseItem \*item) { nextItem = item; }

BackgroundParseItem \*GetNextUnprocessedItem() const { return nextUnprocessedItem; }

void SetNextUnprocessedItem(BackgroundParseItem \*item) { nextUnprocessedItem = item; }

BackgroundParseItem \*GetPrevUnprocessedItem() const { return prevUnprocessedItem; }

void SetPrevUnprocessedItem(BackgroundParseItem \*item) { prevUnprocessedItem = item; }

DList<ParseNode\*, ArenaAllocator>\* RegExpNodeList() { return regExpNodes; }

void OnAddToParseQueue();

void OnRemoveFromParseQueue();

void AddRegExpNode(ParseNode \*const pnode, ArenaAllocator \*alloc);

private:

ParseContext parseContext;

Parser \*parser;

BackgroundParseItem \*nextItem;

BackgroundParseItem \*nextUnprocessedItem;

BackgroundParseItem \*prevUnprocessedItem;

ParseNode \*parseNode;

CompileScriptException \*pse;

NodeDList\* regExpNodes;

HRESULT hr;

uint maxBlockId;

bool isDeferred;

bool strictMode;

bool inParseQueue;

bool completed;

};

class BackgroundParser sealed : public JsUtil::WaitableJobManager

{

public:

BackgroundParser(Js::ScriptContext \*scriptContext);

~BackgroundParser();

static BackgroundParser \* New(Js::ScriptContext \*scriptContext);

static void Delete(BackgroundParser \*backgroundParser);

volatile uint\* GetPendingBackgroundItemsPtr() const { return (volatile uint\*)&pendingBackgroundItems; }

virtual bool Process(JsUtil::Job \*const job, JsUtil::ParallelThreadData \*threadData) override;

virtual void JobProcessed(JsUtil::Job \*const job, const bool succeeded) override;

virtual void OnDecommit(JsUtil::ParallelThreadData \*threadData) override;

bool Process(JsUtil::Job \*const job, Parser \*parser, CompileScriptException \*pse);

bool ParseBackgroundItem(Parser \*parser, ParseNode \*parseNode, bool isDeferred);

BackgroundParseItem \* NewBackgroundParseItem(Parser \*parser, ParseNode \*parseNode, bool isDeferred);

BackgroundParseItem \*GetJob(BackgroundParseItem \*item) const;

bool WasAddedToJobProcessor(JsUtil::Job \*const job) const;

void BeforeWaitForJob(BackgroundParseItem \*const item) const;

void AfterWaitForJob(BackgroundParseItem \*const item) const;

BackgroundParseItem \*GetNextUnprocessedItem() const;

void AddUnprocessedItem(BackgroundParseItem \*const item);

void RemoveFromUnprocessedItems(BackgroundParseItem \*const item);

void SetFailedBackgroundParseItem(BackgroundParseItem \*item) { failedBackgroundParseItem = item; }

BackgroundParseItem \*GetFailedBackgroundParseItem() const { return failedBackgroundParseItem; }

bool HasFailedBackgroundParseItem() const { return failedBackgroundParseItem != nullptr; }

private:

void AddToParseQueue(BackgroundParseItem \*const item, bool prioritize, bool lock);

private:

Js::ScriptContext \*scriptContext;

uint pendingBackgroundItems;

BackgroundParseItem \*failedBackgroundParseItem;

BackgroundParseItem \*unprocessedItemsHead;

BackgroundParseItem \*unprocessedItemsTail;

#if DBG

ThreadContextId mainThreadId;

#endif

};

#endif

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

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

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

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

//

// Map Unicode characters to their equivalence classes induced by the modified ToUpper map.

// i.e.: c1 and c2 are in the same class if ToUpper(c1) == ToUpper(c2).

//

// The ToUpper map takes any character to its Unicode upper case equivalent, with the modification that

// a non-7-bit-ASCII character cannot be mapped to 7-bit-ASCII characters.

//

#include "ParserPch.h"

namespace UnifiedRegex

{

namespace CaseInsensitive

{

struct Transform

{

// This skipCount is to help define the range. Ex, given range [0 - 20]

// If skip count is 1, then all items between 0 and 20 are in the range.

// If skip count is 2, then every even item is in the range, so 0, 2, 4, 6, 8, etc.

byte skipCountOfRange;

MappingSource source;

// Range of chars this transform applies to

Chars<codepoint\_t>::UChar lo;

Chars<codepoint\_t>::UChar hi;

// Offsets to add to original character to get each equivalent character

int delta0;

int delta1;

int delta2;

int delta3;

template <typename Char>

inline void Apply(uint c, Char outEquiv[EquivClassSize]) const

{

Assert(c >= lo && c <= hi);

outEquiv[0] = Chars<Char>::UTC((lo + 1) % skipCountOfRange == c % skipCountOfRange ? (int)c + delta0 : c);

CompileAssert(CaseInsensitive::EquivClassSize == 4);

if (lo % skipCountOfRange == c % skipCountOfRange)

{

outEquiv[1] = Chars<Char>::ITC((int)c + delta1);

outEquiv[2] = Chars<Char>::ITC((int)c + delta2);

outEquiv[3] = Chars<Char>::ITC((int)c + delta3);

}

else

{

outEquiv[1] = outEquiv[2] = outEquiv[3] = Chars<Char>::UTC(c);

}

}

};

/\*

We first construct a total map from character codes to equivalence lists such that:

- if ToUpper(c1) == ToUpper(c2) then c1 has c2 in its equivalence list

- if c1 and c2 appear in the same equivalence list then c1 and c2 have equal equivalence lists

We then compress the above map such that:

- characters with singleton equivalence classes are elided

- consecutive characters with consecutive equivalence lists are represented as a range and delta

- the result is in strictly increasing range order

Using gawk the above is:

gawk -f equiv.gawk http://www.unicode.org/Public/UNIDATA/UnicodeData.txt | gawk -f table.gawk

Where equiv.gawk is:

----------------------------------------------------------------------

BEGIN {

FS = ";";

previncode = -1;

}

length($1) == 4 {

incode = strtonum("0x" $1);

for (i = previncode + 1; i < incode; i++)

map[i] = i;

if ($3 == "Ll" && $15 != "")

{

map[incode] = strtonum("0x" $15);

# non-7-bit-ASCII cannot map to 7-bit-ASCII

if (incode > 127 && map[incode] <= 127)

map[incode] = incode;

}

else

map[incode] = incode;

previncode = incode;

}

END {

for (i = previncode + 1; i <= 0xffff; i++)

map[i] = i;

for (i = 0x0000; i <= 0xffff; i++)

ninv[i] = 0;

for (i = 0x0000; i <= 0xffff; i++)

{

if (map[i] != i)

ninv[map[i]]++;

}

maxninv = 0;

for (i = 0x0000; i <= 0xffff; i++)

{

if (ninv[i] > maxninv)

maxninv = ninv[i];

}

if (maxninv > 2)

print "ERROR";

for (i = 0x0000; i <= 0xffff; i++)

inv[i] = "";

for (i = 0x0000; i <= 0xffff; i++)

{

if (map[i] != i)

inv[map[i]] = sprintf("%s;0x%04x", inv[map[i]], i);

}

for (i = 0x0000; i <= 0xffff; i++)

{

if (map[i] != i)

{

equiv[i] = sprintf("0x%04x%s", map[i], inv[map[i]]);

nequiv[i] = 1 + ninv[map[i]];

}

else if (inv[i] != "")

{

equiv[i] = sprintf("0x%04x%s", i, inv[i]);

nequiv[i] = 1 + ninv[i];

}

else

{

equiv[i] = sprintf("0x%04x", i);

nequiv[i] = 1;

}

}

nentries = 0

for (i = 0x0000; i <= 0xffff; i++)

{

if (nequiv[i] > 1)

{

printf("0x%04x;%s\n", i, equiv[i]);

nentries++;

}

}

#printf("nentries = %d\n", nentries);

}

----------------------------------------------------------------------

And table.gawk is:

----------------------------------------------------------------------

BEGIN {

FS = ";";

lastCode = -1;

currStart = -1;

for (i = 0; i < 3; i++)

currDeltas[i] = "";

}

{

if (NF > 4)

print "ERROR"

incode = strtonum($1);

for (i = 0; i < NF - 1; i++)

equivs[i] = strtonum($(i+2));

for (i = NF - 1; i < 3; i++)

equivs[i] = equivs[i - 1];

#printf("0x%04x, 0x%04x, 0x%04x, 0x%04x\n", incode, equivs[0], equivs[1], equivs[2]);

for (i = 0; i < 3; i++)

deltas[i] = equivs[i] - incode;

if (currStart < 0)

{

# start a new range

currStart = incode;

for (i = 0; i < 3; i++)

currDeltas[i] = deltas[i]

}

else if (incode == lastCode + 1 && deltas[0] == currDeltas[0] && deltas[1] == currDeltas[1] && deltas[2] == currDeltas[2])

{

# keep accumulating range

}

else

{

# dump current range and start a new one

printf(" 0x%04x, 0x%04x, %d, %d, %d,\n", currStart, lastCode, currDeltas[0], currDeltas[1], currDeltas[2]);

currStart = incode;

for (i = 0; i < 3; i++)

currDeltas[i] = deltas[i]

}

lastCode = incode;

}

END {

printf(" 0x%04x, 0x%04x, %d, %d, %d,\n", currStart, lastCode, currDeltas[0], currDeltas[1], currDeltas[2]);

}

----------------------------------------------------------------------

\*/

// For case-folding entries, version 8.0.0 of CaseFolding.txt located at [1] was used.

// [1] ftp://ftp.unicode.org/Public/UNIDATA/CaseFolding.txt

static const Transform transforms[] =

{

1, MappingSource::UnicodeData, 0x0041, 0x004a, 0, 32, 32, 32,

1, MappingSource::CaseFolding, 0x004b, 0x004b, 0, 32, 8415, 8415,

1, MappingSource::UnicodeData, 0x004b, 0x0052, 0, 32, 32, 32,

1, MappingSource::CaseFolding, 0x0053, 0x0053, 0, 32, 300, 300,

1, MappingSource::UnicodeData, 0x0053, 0x005a, 0, 32, 32, 32,

1, MappingSource::UnicodeData, 0x0061, 0x006a, -32, 0, 0, 0,

1, MappingSource::CaseFolding, 0x006b, 0x006b, -32, 0, 8383, 8383,

1, MappingSource::UnicodeData, 0x006b, 0x0072, -32, 0, 0, 0,

1, MappingSource::CaseFolding, 0x0073, 0x0073, -32, 0, 268, 268,

1, MappingSource::UnicodeData, 0x0073, 0x007a, -32, 0, 0, 0,

1, MappingSource::UnicodeData, 0x00b5, 0x00b5, 743, 0, 775, 775,

1, MappingSource::UnicodeData, 0x00c0, 0x00c4, 0, 32, 32, 32,

1, MappingSource::CaseFolding, 0x00c5, 0x00c5, 0, 32, 8294, 8294,

1, MappingSource::UnicodeData, 0x00c5, 0x00d6, 0, 32, 32, 32,

1, MappingSource::UnicodeData, 0x00d8, 0x00de, 0, 32, 32, 32,

1, MappingSource::UnicodeData, 0x00DF, 0x00DF, 0, 7615, 7615, 7615,

1, MappingSource::UnicodeData, 0x00e0, 0x00e4, -32, 0, 0, 0,

1, MappingSource::CaseFolding, 0x00e5, 0x00e5, -32, 0, 8262, 8262,

1, MappingSource::UnicodeData, 0x00e5, 0x00f6, -32, 0, 0, 0,

1, MappingSource::UnicodeData, 0x00f8, 0x00fe, -32, 0, 0, 0,

1, MappingSource::UnicodeData, 0x00ff, 0x00ff, 121, 0, 0, 0,

2, MappingSource::UnicodeData, 0x0100, 0x012f, -1, 1, 1, 1,

2, MappingSource::UnicodeData, 0x0132, 0x0137, -1, 1, 1, 1,

2, MappingSource::UnicodeData, 0x0139, 0x0148, -1, 1, 1, 1,

2, MappingSource::UnicodeData, 0x014a, 0x0177, -1, 1, 1, 1,

1, MappingSource::UnicodeData, 0x0178, 0x0178, 0, -121, -121, -121,

2, MappingSource::UnicodeData, 0x0179, 0x017e, -1, 1, 1, 1,

1, MappingSource::CaseFolding, 0x017f, 0x017f, -300, -268, 0, 0,

1, MappingSource::UnicodeData, 0x0180, 0x0180, 195, 0, 0, 0,

1, MappingSource::UnicodeData, 0x0181, 0x0181, 0, 210, 210, 210,

2, MappingSource::UnicodeData, 0x0182, 0x0185, -1, 1, 1, 1,

1, MappingSource::UnicodeData, 0x0186, 0x0186, 0, 206, 206, 206,

1, MappingSource::UnicodeData, 0x0187, 0x0187, 0, 1, 1, 1,

1, MappingSource::UnicodeData, 0x0188, 0x0188, -1, 0, 0, 0,

1, MappingSource::UnicodeData, 0x0189, 0x018a, 0, 205, 205, 205,

1, MappingSource::UnicodeData, 0x018b, 0x018b, 0, 1, 1, 1,

1, MappingSource::UnicodeData, 0x018c, 0x018c, -1, 0, 0, 0,

1, MappingSource::UnicodeData, 0x018e, 0x018e, 0, 79, 79, 79,

1, MappingSource::UnicodeData, 0x018f, 0x018f, 0, 202, 202, 202,

1, MappingSource::UnicodeData, 0x0190, 0x0190, 0, 203, 203, 203,

1, MappingSource::UnicodeData, 0x0191, 0x0191, 0, 1, 1, 1,

1, MappingSource::UnicodeData, 0x0192, 0x0192, -1, 0, 0, 0,

1, MappingSource::UnicodeData, 0x0193, 0x0193, 0, 205, 205, 205,

1, MappingSource::UnicodeData, 0x0194, 0x0194, 0, 207, 207, 207,

1, MappingSource::UnicodeData, 0x0195, 0x0195, 97, 0, 0, 0,

1, MappingSource::UnicodeData, 0x0196, 0x0196, 0, 211, 211, 211,

1, MappingSource::UnicodeData, 0x0197, 0x0197, 0, 209, 209, 209,

1, MappingSource::UnicodeData, 0x0198, 0x0198, 0, 1, 1, 1,

1, MappingSource::UnicodeData, 0x0199, 0x0199, -1, 0, 0, 0,

1, MappingSource::UnicodeData, 0x019a, 0x019a, 163, 0, 0, 0,

1, MappingSource::UnicodeData, 0x019c, 0x019c, 0, 211, 211, 211,

1, MappingSource::UnicodeData, 0x019d, 0x019d, 0, 213, 213, 213,

1, MappingSource::UnicodeData, 0x019e, 0x019e, 130, 0, 0, 0,

1, MappingSource::UnicodeData, 0x019f, 0x019f, 0, 214, 214, 214,

2, MappingSource::UnicodeData, 0x01a0, 0x01a5, -1, 1, 1, 1,

1, MappingSource::UnicodeData, 0x01a6, 0x01a6, 0, 218, 218, 218,

1, MappingSource::UnicodeData, 0x01a7, 0x01a7, 0, 1, 1, 1,

1, MappingSource::UnicodeData, 0x01a8, 0x01a8, -1, 0, 0, 0,

1, MappingSource::UnicodeData, 0x01a9, 0x01a9, 0, 218, 218, 218,

1, MappingSource::UnicodeData, 0x01ac, 0x01ac, 0, 1, 1, 1,

1, MappingSource::UnicodeData, 0x01ad, 0x01ad, -1, 0, 0, 0,

1, MappingSource::UnicodeData, 0x01ae, 0x01ae, 0, 218, 218, 218,

1, MappingSource::UnicodeData, 0x01af, 0x01af, 0, 1, 1, 1,

1, MappingSource::UnicodeData, 0x01b0, 0x01b0, -1, 0, 0, 0,

1, MappingSource::UnicodeData, 0x01b1, 0x01b2, 0, 217, 217, 217,

1, MappingSource::UnicodeData, 0x01b3, 0x01b3, 0, 1, 1, 1,

1, MappingSource::UnicodeData, 0x01b4, 0x01b4, -1, 0, 0, 0,

1, MappingSource::UnicodeData, 0x01b5, 0x01b5, 0, 1, 1, 1,

1, MappingSource::UnicodeData, 0x01b6, 0x01b6, -1, 0, 0, 0,

1, MappingSource::UnicodeData, 0x01b7, 0x01b7, 0, 219, 219, 219,

2, MappingSource::UnicodeData, 0x01b8, 0x01bd, -1, 1, 1, 1,

1, MappingSource::UnicodeData, 0x01bf, 0x01bf, 56, 0, 0, 0,

1, MappingSource::UnicodeData, 0x01c4, 0x01c4, 2, 0, 0, 0,

1, MappingSource::UnicodeData, 0x01c5, 0x01c5, 1, 0, 0, 0,

1, MappingSource::UnicodeData, 0x01c6, 0x01c6, 0, -2, -1, -1,

1, MappingSource::UnicodeData, 0x01c7, 0x01c7, 2, 0, 0, 0,

1, MappingSource::UnicodeData, 0x01c8, 0x01c8, 1, 0, 0, 0,

1, MappingSource::UnicodeData, 0x01c9, 0x01c9, 0, -2, -1, -1,

1, MappingSource::UnicodeData, 0x01CA, 0x01CA, 2, 0, 0, 0,

1, MappingSource::UnicodeData, 0x01CB, 0x01CB, 1, 0, 0, 0,

1, MappingSource::UnicodeData, 0x01CC, 0x01CC, 0, -2, -1, -1,

2, MappingSource::UnicodeData, 0x01cd, 0x01dc, -1, 1, 1, 1,

1, MappingSource::UnicodeData, 0x01dd, 0x01dd, -79, 0, 0, 0,

2, MappingSource::UnicodeData, 0x01de, 0x01f5, -1, 1, 1, 1,

1, MappingSource::UnicodeData, 0x01F1, 0x01F1, 2, 0, 0, 0,

1, MappingSource::UnicodeData, 0x01F2, 0x01F2, 1, 0, 0, 0,

1, MappingSource::UnicodeData, 0x01F3, 0x01F3, 0, -2, -1, -1,

2, MappingSource::UnicodeData, 0x01f4, 0x01f5, -1, 1, 1, 1,

1, MappingSource::UnicodeData, 0x01f6, 0x01f6, 0, -97, -97, -97,

1, MappingSource::UnicodeData, 0x01f7, 0x01f7, 0, -56, -56, -56,

2, MappingSource::UnicodeData, 0x01f8, 0x021f, -1, 1, 1, 1,

1, MappingSource::UnicodeData, 0x0220, 0x0220, 0, -130, -130, -130,

2, MappingSource::UnicodeData, 0x0222, 0x0233, -1, 1, 1, 1,

1, MappingSource::UnicodeData, 0x023a, 0x023a, 0, 10795, 10795, 10795,

1, MappingSource::UnicodeData, 0x023b, 0x023b, 0, 1, 1, 1,

1, MappingSource::UnicodeData, 0x023c, 0x023c, -1, 0, 0, 0,

1, MappingSource::UnicodeData, 0x023d, 0x023d, 0, -163, -163, -163,

1, MappingSource::UnicodeData, 0x023e, 0x023e, 0, 10792, 10792, 10792,

1, MappingSource::UnicodeData, 0x023f, 0x0240, 10815, 0, 0, 0,

1, MappingSource::UnicodeData, 0x0241, 0x0241, 0, 1, 1, 1,

1, MappingSource::UnicodeData, 0x0242, 0x0242, -1, 0, 0, 0,

1, MappingSource::UnicodeData, 0x0243, 0x0243, 0, -195, -195, -195,

1, MappingSource::UnicodeData, 0x0244, 0x0244, 0, 69, 69, 69,

1, MappingSource::UnicodeData, 0x0245, 0x0245, 0, 71, 71, 71,

2, MappingSource::UnicodeData, 0x0246, 0x024f, -1, 1, 1, 1,

1, MappingSource::UnicodeData, 0x0250, 0x0250, 10783, 0, 0, 0,

1, MappingSource::UnicodeData, 0x0251, 0x0251, 10780, 0, 0, 0,

1, MappingSource::UnicodeData, 0x0252, 0x0252, 10782, 0, 0, 0,

1, MappingSource::UnicodeData, 0x0253, 0x0253, -210, 0, 0, 0,

1, MappingSource::UnicodeData, 0x0254, 0x0254, -206, 0, 0, 0,

1, MappingSource::UnicodeData, 0x0256, 0x0257, -205, 0, 0, 0,

1, MappingSource::UnicodeData, 0x0259, 0x0259, -202, 0, 0, 0,

1, MappingSource::UnicodeData, 0x025b, 0x025b, -203, 0, 0, 0,

1, MappingSource::UnicodeData, 0x0260, 0x0260, -205, 0, 0, 0,

1, MappingSource::UnicodeData, 0x0263, 0x0263, -207, 0, 0, 0,

1, MappingSource::UnicodeData, 0x0265, 0x0265, 42280, 0, 0, 0,

1, MappingSource::UnicodeData, 0x0266, 0x0266, 42308, 0, 0, 0,

1, MappingSource::UnicodeData, 0x0268, 0x0268, -209, 0, 0, 0,

1, MappingSource::UnicodeData, 0x0269, 0x0269, -211, 0, 0, 0,

1, MappingSource::UnicodeData, 0x026b, 0x026b, 10743, 0, 0, 0,

1, MappingSource::UnicodeData, 0x026f, 0x026f, -211, 0, 0, 0,

1, MappingSource::UnicodeData, 0x0271, 0x0271, 10749, 0, 0, 0,

1, MappingSource::UnicodeData, 0x0272, 0x0272, -213, 0, 0, 0,

1, MappingSource::UnicodeData, 0x0275, 0x0275, -214, 0, 0, 0,

1, MappingSource::UnicodeData, 0x027d, 0x027d, 10727, 0, 0, 0,

1, MappingSource::UnicodeData, 0x0280, 0x0280, -218, 0, 0, 0,

1, MappingSource::UnicodeData, 0x0283, 0x0283, -218, 0, 0, 0,

1, MappingSource::UnicodeData, 0x0288, 0x0288, -218, 0, 0, 0,

1, MappingSource::UnicodeData, 0x0289, 0x0289, -69, 0, 0, 0,

1, MappingSource::UnicodeData, 0x028a, 0x028b, -217, 0, 0, 0,

1, MappingSource::UnicodeData, 0x028c, 0x028c, -71, 0, 0, 0,

1, MappingSource::UnicodeData, 0x0292, 0x0292, -219, 0, 0, 0,

1, MappingSource::CaseFolding, 0x0345, 0x0345, 0, 84, 116, 7289,

2, MappingSource::UnicodeData, 0x0370, 0x0373, -1, 1, 1, 1,

1, MappingSource::UnicodeData, 0x0376, 0x0376, 0, 1, 1, 1,

1, MappingSource::UnicodeData, 0x0377, 0x0377, -1, 0, 0, 0,

1, MappingSource::UnicodeData, 0x037b, 0x037d, 130, 0, 0, 0,

1, MappingSource::UnicodeData, 0x0386, 0x0386, 0, 38, 38, 38,

1, MappingSource::UnicodeData, 0x0388, 0x038a, 0, 37, 37, 37,

1, MappingSource::UnicodeData, 0x038c, 0x038c, 0, 64, 64, 64,

1, MappingSource::UnicodeData, 0x038e, 0x038f, 0, 63, 63, 63,

1, MappingSource::UnicodeData, 0x0391, 0x0391, 0, 32, 32, 32,

1, MappingSource::UnicodeData, 0x0392, 0x0392, 0, 32, 62, 62,

1, MappingSource::UnicodeData, 0x0393, 0x0394, 0, 32, 32, 32,

1, MappingSource::UnicodeData, 0x0395, 0x0395, 0, 32, 96, 96,

1, MappingSource::UnicodeData, 0x0396, 0x0397, 0, 32, 32, 32,

1, MappingSource::CaseFolding, 0x0398, 0x0398, 0, 32, 57, 92,

1, MappingSource::UnicodeData, 0x0398, 0x0398, 0, 32, 57, 57,

1, MappingSource::CaseFolding, 0x0399, 0x0399, -84, 0, 32, 7205,

1, MappingSource::UnicodeData, 0x0399, 0x0399, 0, 32, 7205, 7205,

1, MappingSource::UnicodeData, 0x039a, 0x039a, 0, 32, 86, 86,

1, MappingSource::UnicodeData, 0x039b, 0x039b, 0, 32, 32, 32,

1, MappingSource::UnicodeData, 0x039c, 0x039c, 0, -743, 32, 32,

1, MappingSource::UnicodeData, 0x039d, 0x039f, 0, 32, 32, 32,

1, MappingSource::UnicodeData, 0x03a0, 0x03a0, 0, 32, 54, 54,

1, MappingSource::UnicodeData, 0x03a1, 0x03a1, 0, 32, 80, 80,

1, MappingSource::UnicodeData, 0x03a3, 0x03a3, 0, 31, 32, 32,

1, MappingSource::UnicodeData, 0x03a4, 0x03a5, 0, 32, 32, 32,

1, MappingSource::UnicodeData, 0x03a6, 0x03a6, 0, 32, 47, 47,

1, MappingSource::UnicodeData, 0x03a7, 0x03a8, 0, 32, 32, 32,

1, MappingSource::CaseFolding, 0x03a9, 0x03a9, 0, 32, 7549, 7549,

1, MappingSource::UnicodeData, 0x03a9, 0x03ab, 0, 32, 32, 32,

1, MappingSource::UnicodeData, 0x03ac, 0x03ac, -38, 0, 0, 0,

1, MappingSource::UnicodeData, 0x03ad, 0x03af, -37, 0, 0, 0,

1, MappingSource::UnicodeData, 0x03b1, 0x03b1, -32, 0, 0, 0,

1, MappingSource::UnicodeData, 0x03b2, 0x03b2, -32, 0, 30, 30,

1, MappingSource::UnicodeData, 0x03b3, 0x03b4, -32, 0, 0, 0,

1, MappingSource::UnicodeData, 0x03b5, 0x03b5, -32, 0, 64, 64,

1, MappingSource::UnicodeData, 0x03b6, 0x03b7, -32, 0, 0, 0,

1, MappingSource::CaseFolding, 0x03b8, 0x03b8, -32, 0, 25, 60,

1, MappingSource::UnicodeData, 0x03b8, 0x03b8, -32, 0, 25, 25,

1, MappingSource::CaseFolding, 0x03b9, 0x03b9, -116, -32, 0, 7173,

1, MappingSource::UnicodeData, 0x03b9, 0x03b9, -32, 0, 7173, 7173,

1, MappingSource::UnicodeData, 0x03ba, 0x03ba, -32, 0, 54, 54,

1, MappingSource::UnicodeData, 0x03bb, 0x03bb, -32, 0, 0, 0,

1, MappingSource::UnicodeData, 0x03bc, 0x03bc, -32, -775, 0, 0,

1, MappingSource::UnicodeData, 0x03bd, 0x03bf, -32, 0, 0, 0,

1, MappingSource::UnicodeData, 0x03c0, 0x03c0, -32, 0, 22, 22,

1, MappingSource::UnicodeData, 0x03c1, 0x03c1, -32, 0, 48, 48,

1, MappingSource::UnicodeData, 0x03c2, 0x03c2, -31, 0, 1, 1,

1, MappingSource::UnicodeData, 0x03c3, 0x03c3, -32, -1, 0, 0,

1, MappingSource::UnicodeData, 0x03c4, 0x03c5, -32, 0, 0, 0,

1, MappingSource::UnicodeData, 0x03c6, 0x03c6, -32, 0, 15, 15,

1, MappingSource::UnicodeData, 0x03c7, 0x03c8, -32, 0, 0, 0,

1, MappingSource::CaseFolding, 0x03c9, 0x03c9, -32, 0, 7517, 7517,

1, MappingSource::UnicodeData, 0x03c9, 0x03cb, -32, 0, 0, 0,

1, MappingSource::UnicodeData, 0x03cc, 0x03cc, -64, 0, 0, 0,

1, MappingSource::UnicodeData, 0x03cd, 0x03ce, -63, 0, 0, 0,

1, MappingSource::UnicodeData, 0x03cf, 0x03cf, 0, 8, 8, 8,

1, MappingSource::UnicodeData, 0x03d0, 0x03d0, -62, -30, 0, 0,

1, MappingSource::CaseFolding, 0x03d1, 0x03d1, -57, -25, 0, 35,

1, MappingSource::UnicodeData, 0x03d1, 0x03d1, -57, -25, 0, 0,

1, MappingSource::UnicodeData, 0x03d5, 0x03d5, -47, -15, 0, 0,

1, MappingSource::UnicodeData, 0x03d6, 0x03d6, -54, -22, 0, 0,

1, MappingSource::UnicodeData, 0x03d7, 0x03d7, -8, 0, 0, 0,

2, MappingSource::UnicodeData, 0x03d8, 0x03ef, -1, 1, 1, 1,

1, MappingSource::UnicodeData, 0x03f0, 0x03f0, -86, -54, 0, 0,

1, MappingSource::UnicodeData, 0x03f1, 0x03f1, -80, -48, 0, 0,

1, MappingSource::UnicodeData, 0x03f2, 0x03f2, 7, 0, 0, 0,

1, MappingSource::CaseFolding, 0x03f4, 0x03f4, -92, -60, -35, 0,

1, MappingSource::UnicodeData, 0x03f5, 0x03f5, -96, -64, 0, 0,

1, MappingSource::UnicodeData, 0x03f7, 0x03f7, 0, 1, 1, 1,

1, MappingSource::UnicodeData, 0x03f8, 0x03f8, -1, 0, 0, 0,

1, MappingSource::UnicodeData, 0x03f9, 0x03f9, 0, -7, -7, -7,

1, MappingSource::UnicodeData, 0x03fa, 0x03fa, 0, 1, 1, 1,

1, MappingSource::UnicodeData, 0x03fb, 0x03fb, -1, 0, 0, 0,

1, MappingSource::UnicodeData, 0x03fd, 0x03ff, 0, -130, -130, -130,

1, MappingSource::UnicodeData, 0x0400, 0x040f, 0, 80, 80, 80,

1, MappingSource::UnicodeData, 0x0410, 0x042f, 0, 32, 32, 32,

1, MappingSource::UnicodeData, 0x0430, 0x044f, -32, 0, 0, 0,

1, MappingSource::UnicodeData, 0x0450, 0x045f, -80, 0, 0, 0,

2, MappingSource::UnicodeData, 0x0460, 0x0481, -1, 1, 1, 1,

2, MappingSource::UnicodeData, 0x048a, 0x04bf, -1, 1, 1, 1,

1, MappingSource::UnicodeData, 0x04c0, 0x04c0, 0, 15, 15, 15,

2, MappingSource::UnicodeData, 0x04c1, 0x04ce, -1, 1, 1, 1,

1, MappingSource::UnicodeData, 0x04cf, 0x04cf, -15, 0, 0, 0,

2, MappingSource::UnicodeData, 0x04d0, 0x0527, -1, 1, 1, 1,

1, MappingSource::UnicodeData, 0x0531, 0x0556, 0, 48, 48, 48,

1, MappingSource::UnicodeData, 0x0561, 0x0586, -48, 0, 0, 0,

1, MappingSource::UnicodeData, 0x10a0, 0x10c5, 0, 7264, 7264, 7264,

1, MappingSource::UnicodeData, 0x10C7, 0x10C7, 0, 7264, 7264, 7264,

1, MappingSource::UnicodeData, 0x10CD, 0x10CD, 0, 7264, 7264, 7264,

1, MappingSource::UnicodeData, 0x1d79, 0x1d79, 35332, 0, 0, 0,

1, MappingSource::UnicodeData, 0x1d7d, 0x1d7d, 3814, 0, 0, 0,

2, MappingSource::UnicodeData, 0x1e00, 0x1e5f, -1, 1, 1, 1,

1, MappingSource::UnicodeData, 0x1e60, 0x1e60, 0, 1, 59, 59,

1, MappingSource::UnicodeData, 0x1e61, 0x1e61, -1, 0, 58, 58,

2, MappingSource::UnicodeData, 0x1e62, 0x1e95, -1, 1, 1, 1,

1, MappingSource::UnicodeData, 0x1e9b, 0x1e9b, -59, -58, 0, 0,

1, MappingSource::UnicodeData, 0x1E9E, 0x1E9E, -7615, 0, 0, 0,

2, MappingSource::UnicodeData, 0x1ea0, 0x1eff, -1, 1, 1, 1,

1, MappingSource::UnicodeData, 0x1f00, 0x1f07, 8, 0, 0, 0,

1, MappingSource::UnicodeData, 0x1f08, 0x1f0f, 0, -8, -8, -8,

1, MappingSource::UnicodeData, 0x1f10, 0x1f15, 8, 0, 0, 0,

1, MappingSource::UnicodeData, 0x1f18, 0x1f1d, 0, -8, -8, -8,

1, MappingSource::UnicodeData, 0x1f20, 0x1f27, 8, 0, 0, 0,

1, MappingSource::UnicodeData, 0x1f28, 0x1f2f, 0, -8, -8, -8,

1, MappingSource::UnicodeData, 0x1f30, 0x1f37, 8, 0, 0, 0,

1, MappingSource::UnicodeData, 0x1f38, 0x1f3f, 0, -8, -8, -8,

1, MappingSource::UnicodeData, 0x1f40, 0x1f45, 8, 0, 0, 0,

1, MappingSource::UnicodeData, 0x1f48, 0x1f4d, 0, -8, -8, -8,

1, MappingSource::UnicodeData, 0x1f51, 0x1f51, 8, 0, 0, 0,

1, MappingSource::UnicodeData, 0x1f53, 0x1f53, 8, 0, 0, 0,

1, MappingSource::UnicodeData, 0x1f55, 0x1f55, 8, 0, 0, 0,

1, MappingSource::UnicodeData, 0x1f57, 0x1f57, 8, 0, 0, 0,

1, MappingSource::UnicodeData, 0x1f59, 0x1f59, 0, -8, -8, -8,

1, MappingSource::UnicodeData, 0x1f5b, 0x1f5b, 0, -8, -8, -8,

1, MappingSource::UnicodeData, 0x1f5d, 0x1f5d, 0, -8, -8, -8,

1, MappingSource::UnicodeData, 0x1f5f, 0x1f5f, 0, -8, -8, -8,

1, MappingSource::UnicodeData, 0x1f60, 0x1f67, 8, 0, 0, 0,

1, MappingSource::UnicodeData, 0x1f68, 0x1f6f, 0, -8, -8, -8,

1, MappingSource::UnicodeData, 0x1f70, 0x1f71, 74, 0, 0, 0,

1, MappingSource::UnicodeData, 0x1f72, 0x1f75, 86, 0, 0, 0,

1, MappingSource::UnicodeData, 0x1f76, 0x1f77, 100, 0, 0, 0,

1, MappingSource::UnicodeData, 0x1f78, 0x1f79, 128, 0, 0, 0,

1, MappingSource::UnicodeData, 0x1f7a, 0x1f7b, 112, 0, 0, 0,

1, MappingSource::UnicodeData, 0x1f7c, 0x1f7d, 126, 0, 0, 0,

1, MappingSource::UnicodeData, 0x1f80, 0x1f87, 8, 0, 0, 0,

1, MappingSource::UnicodeData, 0x1f88, 0x1f8f, 0, -8, -8, -8,

1, MappingSource::UnicodeData, 0x1f90, 0x1f97, 8, 0, 0, 0,

1, MappingSource::UnicodeData, 0x1f98, 0x1f9f, 0, -8, -8, -8,

1, MappingSource::UnicodeData, 0x1fa0, 0x1fa7, 8, 0, 0, 0,

1, MappingSource::UnicodeData, 0x1fa8, 0x1faf, 0, -8, -8, -8,

1, MappingSource::UnicodeData, 0x1fb0, 0x1fb1, 8, 0, 0, 0,

1, MappingSource::UnicodeData, 0x1fb3, 0x1fb3, 9, 0, 0, 0,

1, MappingSource::UnicodeData, 0x1fb8, 0x1fb9, 0, -8, -8, -8,

1, MappingSource::UnicodeData, 0x1fba, 0x1fbb, 0, -74, -74, -74,

1, MappingSource::UnicodeData, 0x1fbc, 0x1fbc, 0, -9, -9, -9,

1, MappingSource::CaseFolding, 0x1fbe, 0x1fbe, -7289, -7205, -7173, 0,

1, MappingSource::UnicodeData, 0x1fbe, 0x1fbe, -7205, -7173, 0, 0,

1, MappingSource::UnicodeData, 0x1fc3, 0x1fc3, 9, 0, 0, 0,

1, MappingSource::UnicodeData, 0x1fc8, 0x1fcb, 0, -86, -86, -86,

1, MappingSource::UnicodeData, 0x1fcc, 0x1fcc, 0, -9, -9, -9,

1, MappingSource::UnicodeData, 0x1fd0, 0x1fd1, 8, 0, 0, 0,

1, MappingSource::UnicodeData, 0x1fd8, 0x1fd9, 0, -8, -8, -8,

1, MappingSource::UnicodeData, 0x1fda, 0x1fdb, 0, -100, -100, -100,

1, MappingSource::UnicodeData, 0x1fe0, 0x1fe1, 8, 0, 0, 0,

1, MappingSource::UnicodeData, 0x1fe5, 0x1fe5, 7, 0, 0, 0,

1, MappingSource::UnicodeData, 0x1fe8, 0x1fe9, 0, -8, -8, -8,

1, MappingSource::UnicodeData, 0x1fea, 0x1feb, 0, -112, -112, -112,

1, MappingSource::UnicodeData, 0x1fec, 0x1fec, 0, -7, -7, -7,

1, MappingSource::UnicodeData, 0x1ff3, 0x1ff3, 9, 0, 0, 0,

1, MappingSource::UnicodeData, 0x1ff8, 0x1ff9, 0, -128, -128, -128,

1, MappingSource::UnicodeData, 0x1ffa, 0x1ffb, 0, -126, -126, -126,

1, MappingSource::UnicodeData, 0x1ffc, 0x1ffc, 0, -9, -9, -9,

1, MappingSource::CaseFolding, 0x2126, 0x2126, -7549, -7517, 0, 0,

1, MappingSource::CaseFolding, 0x212a, 0x212a, -8415, -8383, 0, 0,

1, MappingSource::CaseFolding, 0x212b, 0x212b, -8294, -8262, 0, 0,

1, MappingSource::UnicodeData, 0x2132, 0x2132, 0, 28, 28, 28,

1, MappingSource::UnicodeData, 0x214e, 0x214e, -28, 0, 0, 0,

1, MappingSource::UnicodeData, 0x2160, 0x216F, 0, 16, 16, 16,

1, MappingSource::UnicodeData, 0x2170, 0x217F, -16, 0, 0, 0,

1, MappingSource::UnicodeData, 0x2183, 0x2183, 0, 1, 1, 1,

1, MappingSource::UnicodeData, 0x2184, 0x2184, -1, 0, 0, 0,

1, MappingSource::UnicodeData, 0x24B6, 0x24CF, 0, 26, 26, 26,

1, MappingSource::UnicodeData, 0x24D0, 0x24E9, -26, 0, 0, 0,

1, MappingSource::UnicodeData, 0x2c00, 0x2c2e, 0, 48, 48, 48,

1, MappingSource::UnicodeData, 0x2c30, 0x2c5e, -48, 0, 0, 0,

1, MappingSource::UnicodeData, 0x2c60, 0x2c60, 0, 1, 1, 1,

1, MappingSource::UnicodeData, 0x2c61, 0x2c61, -1, 0, 0, 0,

1, MappingSource::UnicodeData, 0x2c62, 0x2c62, 0, -10743, -10743, -10743,

1, MappingSource::UnicodeData, 0x2c63, 0x2c63, 0, -3814, -3814, -3814,

1, MappingSource::UnicodeData, 0x2c64, 0x2c64, 0, -10727, -10727, -10727,

1, MappingSource::UnicodeData, 0x2c65, 0x2c65, -10795, 0, 0, 0,

1, MappingSource::UnicodeData, 0x2c66, 0x2c66, -10792, 0, 0, 0,

2, MappingSource::UnicodeData, 0x2c67, 0x2c6c, -1, 1, 1, 1,

1, MappingSource::UnicodeData, 0x2c6d, 0x2c6d, 0, -10780, -10780, -10780,

1, MappingSource::UnicodeData, 0x2c6e, 0x2c6e, 0, -10749, -10749, -10749,

1, MappingSource::UnicodeData, 0x2c6f, 0x2c6f, 0, -10783, -10783, -10783,

1, MappingSource::UnicodeData, 0x2c70, 0x2c70, 0, -10782, -10782, -10782,

1, MappingSource::UnicodeData, 0x2c72, 0x2c72, 0, 1, 1, 1,

1, MappingSource::UnicodeData, 0x2c73, 0x2c73, -1, 0, 0, 0,

1, MappingSource::UnicodeData, 0x2c75, 0x2c75, 0, 1, 1, 1,

1, MappingSource::UnicodeData, 0x2c76, 0x2c76, -1, 0, 0, 0,

1, MappingSource::UnicodeData, 0x2c7e, 0x2c7f, 0, -10815, -10815, -10815,

2, MappingSource::UnicodeData, 0x2c80, 0x2ce3, -1, 1, 1, 1,

2, MappingSource::UnicodeData, 0x2ceb, 0x2cee, -1, 1, 1, 1,

2, MappingSource::UnicodeData, 0x2CF2, 0x2CF3, -1, 1, 1, 1,

1, MappingSource::UnicodeData, 0x2d00, 0x2d25, -7264, 0, 0, 0,

1, MappingSource::UnicodeData, 0x2D27, 0x2D27, -7264, 0, 0, 0,

1, MappingSource::UnicodeData, 0x2D2D, 0x2D2D, -7264, 0, 0, 0,

2, MappingSource::UnicodeData, 0xa640, 0xa66d, -1, 1, 1, 1,

2, MappingSource::UnicodeData, 0xa680, 0xa697, -1, 1, 1, 1,

2, MappingSource::UnicodeData, 0xa722, 0xa72f, -1, 1, 1, 1,

2, MappingSource::UnicodeData, 0xa732, 0xa76f, -1, 1, 1, 1,

2, MappingSource::UnicodeData, 0xa779, 0xa77c, -1, 1, 1, 1,

1, MappingSource::UnicodeData, 0xa77d, 0xa77d, 0, -35332, -35332, -35332,

2, MappingSource::UnicodeData, 0xa77e, 0xa787, -1, 1, 1, 1,

1, MappingSource::UnicodeData, 0xa78b, 0xa78b, 0, 1, 1, 1,

1, MappingSource::UnicodeData, 0xa78c, 0xa78c, -1, 0, 0, 0,

1, MappingSource::UnicodeData, 0xa78d, 0xa78d, 0, -42280, -42280, -42280,

1, MappingSource::UnicodeData, 0xa790, 0xa790, 0, 1, 1, 1,

1, MappingSource::UnicodeData, 0xa791, 0xa791, -1, 0, 0, 0,

2, MappingSource::UnicodeData, 0xa792, 0xa793, -1, 1, 1, 1,

2, MappingSource::UnicodeData, 0xa7a0, 0xa7a9, -1, 1, 1, 1,

1, MappingSource::UnicodeData, 0xA7AA, 0XA7AA, 0, -42308, -42308, -42308,

1, MappingSource::UnicodeData, 0xff21, 0xff3a, 0, 32, 32, 32,

1, MappingSource::UnicodeData, 0xff41, 0xff5a, -32, 0, 0, 0,

1, MappingSource::CaseFolding, 0x10400, 0x10427, 0, 40, 40, 40,

1, MappingSource::CaseFolding, 0x10428, 0x1044f, -40, 0, 0, 0,

1, MappingSource::CaseFolding, 0x10c80, 0x10cb2, 0, 64, 64, 64,

1, MappingSource::CaseFolding, 0x10cc0, 0x10cf2, -64, 0, 0, 0,

1, MappingSource::CaseFolding, 0x118a0, 0x118bf, 0, 32, 32, 32,

1, MappingSource::CaseFolding, 0x118c0, 0x118df, -32, 0, 0, 0,

};

static const int numTransforms = sizeof(transforms) / sizeof(Transform);

static const Transform lastTransform = transforms[numTransforms - 1];

template <typename Char, typename Fn>

bool RangeToEquivClass(uint& tblidx, uint l, uint h, uint& acth, Char equivl[EquivClassSize], Fn acceptSource)

{

Assert(l <= h);

if (lastTransform.hi >= l)

{

// Skip transforms which come completely before l

while (tblidx < numTransforms && (transforms[tblidx].hi < l || !acceptSource(transforms[tblidx].source)))

{

tblidx++;

}

if (tblidx < numTransforms)

{

// Does current transform intersect the desired range?

uint interl = max(l, static\_cast<uint>(transforms[tblidx].lo));

uint interh = min(h, static\_cast<uint>(transforms[tblidx].skipCountOfRange == 1 ? transforms[tblidx].hi : interl));

if (interl <= interh)

{

if (l < interl)

{

// Part of input range comes before next table range, so that sub-range has trivial equivalence class

acth = interl - 1;

for (int i = 0; i < EquivClassSize; i++)

equivl[i] = Chars<Char>::UTC(l);

return false; // trivial

}

else

{

// Input range begins at a table range, so map the character range

acth = interh;

transforms[tblidx].Apply(interl, equivl);

return true; // non-trivial

}

}

// else fall-through: No intersection, so nothing in this range has non-trivial equivalence class

}

}

// else fall-through: No more transforms, so nothing in this range has a non-trivial equivalence class

acth = h;

for (int i = 0; i < EquivClassSize; i++)

{

equivl[i] = Chars<Char>::UTC(l);

}

return false; // trivial

}

bool RangeToEquivClass(uint & tblidx, uint l, uint h, uint & acth, \_\_out\_ecount(EquivClassSize) wchar\_t equivl[EquivClassSize])

{

return RangeToEquivClass(tblidx, l, h, acth, equivl, [](MappingSource source) {

return source == MappingSource::UnicodeData;

});

}

bool RangeToEquivClass(uint & tblidx, uint l, uint h, uint & acth, \_\_out\_ecount(EquivClassSize) codepoint\_t equivl[EquivClassSize])

{

return RangeToEquivClass(tblidx, l, h, acth, equivl, [](MappingSource source) {

return source == MappingSource::CaseFolding || source == MappingSource::UnicodeData;

});

}

bool RangeToEquivClassOnlyInSource(MappingSource mappingSource, uint& tblidx, uint l, uint h, uint& acth, \_\_out\_ecount(EquivClassSize) wchar\_t equivl[EquivClassSize])

{

return RangeToEquivClass(tblidx, l, h, acth, equivl, [&](MappingSource actualSource) {

return mappingSource == actualSource;

});

}

}

}

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

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

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

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

namespace UnifiedRegex

{

namespace CaseInsensitive

{

// It turns out there are many upper-case characters with three lower-case variants, thus

// the maximum size of an equivalence class is four.

static const int EquivClassSize = 4;

enum class MappingSource : uint8

{

UnicodeData,

CaseFolding

};

// Following two functions return equivalents from UnicodeData (for wchar\_t) and CaseFolding

// (for codepoint\_t) files. Their names don't have anything distinguishing them so that

// they can be called easily from template functions.

bool RangeToEquivClass(uint& tblidx, uint l, uint h, uint& acth, \_\_out\_ecount(EquivClassSize) wchar\_t equivl[EquivClassSize]);

bool RangeToEquivClass(uint& tblidx, uint l, uint h, uint& acth, \_\_out\_ecount(EquivClassSize) codepoint\_t equivl[EquivClassSize]);

// Returns equivalents only from the given source. Some case-folding mappings already exist in

// UnicodeData, so this function doesn't return them when CaseFolding is passed as the source.

bool RangeToEquivClassOnlyInSource(MappingSource mappingSource, uint& tblidx, uint l, uint h, uint& acth, \_\_out\_ecount(EquivClassSize) wchar\_t equivl[EquivClassSize]);

}

}

<?xml version="1.0" encoding="utf-8"?>

<Project DefaultTargets="Build" ToolsVersion="12.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

<Import Condition="'$(ChakraBuildPathImported)'!='true'" Project="$(SolutionDir)Chakra.Build.Paths.props" />

<Import Project="$(BuildConfigPropsPath)Chakra.Build.ProjectConfiguration.props" />

<PropertyGroup Label="Globals">

<TargetName>Chakra.Parser</TargetName>

<ProjectGuid>{F6FAD160-5A4B-476A-93AC-33E0B3A18C0C}</ProjectGuid>

<RootNamespace>JS</RootNamespace>

<Keyword>Win32Proj</Keyword>

</PropertyGroup>

<PropertyGroup Label="Configuration">

<ConfigurationType>StaticLibrary</ConfigurationType>

</PropertyGroup>

<Import Project="$(BuildConfigPropsPath)Chakra.Build.Default.props" />

<Import Project="$(VCTargetsPath)\Microsoft.Cpp.Default.props" />

<Import Project="$(VCTargetsPath)\Microsoft.Cpp.props" />

<Import Project="$(BuildConfigPropsPath)Chakra.Build.props" />

<PropertyGroup>

<\_ProjectFileVersion>10.0.30319.1</\_ProjectFileVersion>

</PropertyGroup>

<ItemDefinitionGroup>

<ClCompile>

<AdditionalIncludeDirectories>

$(MSBuildThisFileDirectory)..\Common;

$(MSBuildThisFileDirectory)..\Backend;

%(AdditionalIncludeDirectories)

</AdditionalIncludeDirectories>

<PrecompiledHeader>Use</PrecompiledHeader>

<PrecompiledHeaderFile>ParserPch.h</PrecompiledHeaderFile>

</ClCompile>

</ItemDefinitionGroup>

<ItemGroup>

<ClCompile Include="$(MSBuildThisFileDirectory)alloc.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)BackgroundParser.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)CaseInsensitive.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)CharClassifier.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)CharSet.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)CharTrie.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)cmperr.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)DebugWriter.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)errstr.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)globals.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)hash.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)OctoquadIdentifier.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)parse.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)RegexCompileTime.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)RegexParser.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)RegexPattern.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)RegexRunTime.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)RegexStats.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)rterror.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)scan.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)screrror.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)StandardChars.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)ParserPch.cpp">

<PrecompiledHeader>Create</PrecompiledHeader>

</ClCompile>

<ClCompile Include="$(MSBuildThisFileDirectory)TextbookBoyerMoore.cpp" />

<None Include="hashfunc.cpp" />

</ItemGroup>

<ItemGroup>

<ClInclude Include="alloc.h" />

<ClInclude Include="BackgroundParser.h" />

<ClInclude Include="CaseInsensitive.h" />

<ClInclude Include="CharClassifier.h" />

<ClInclude Include="CharMap.h" />

<ClInclude Include="Chars.h" />

<ClInclude Include="CharSet.h" />

<ClInclude Include="CharTrie.h" />

<ClInclude Include="cmperr.h" />

<ClInclude Include="DebugWriter.h" />

<ClInclude Include="errstr.h" />

<ClInclude Include="formalsutil.h" />

<ClInclude Include="globals.h" />

<ClInclude Include="hash.h" />

<ClInclude Include="idiom.h" />

<ClInclude Include="keywords.h" />

<ClInclude Include="kwd-lsc.h" />

<ClInclude Include="kwd-swtch.h" />

<ClInclude Include="kwds\_sw.h" />

<ClInclude Include="objnames.h" />

<ClInclude Include="OctoquadIdentifier.h" />

<ClInclude Include="parse.h" />

<ClInclude Include="ParseFlags.h" />

<ClInclude Include="Parser.h" />

<ClInclude Include="ParserCommon.h" />

<ClInclude Include="ParserPch.h" />

<ClInclude Include="ParseTreeComparer.h" />

<ClInclude Include="perrors.h" />

<ClInclude Include="pnodechange.h" />

<ClInclude Include="pnodediff.h" />

<ClInclude Include="pnodevisit.h" />

<ClInclude Include="pnodewalk.h" />

<ClInclude Include="popcode.h" />

<ClInclude Include="ptlist.h" />

<ClInclude Include="ptree.h" />

<ClInclude Include="regcodes.h" />

<ClInclude Include="RegexCommon.h" />

<ClInclude Include="RegexCompileTime.h" />

<ClInclude Include="RegexContcodes.h" />

<ClInclude Include="RegexFlags.h" />

<ClInclude Include="RegexOpcodes.h" />

<ClInclude Include="RegexParser.h" />

<ClInclude Include="RegexPattern.h" />

<ClInclude Include="RegexRunTime.h" />

<ClInclude Include="RegexStats.h" />

<ClInclude Include="rterror.h" />

<ClInclude Include="rterrors.h" />

<ClInclude Include="rterrors\_limits.h" />

<ClInclude Include="scan.h" />

<ClInclude Include="screrror.h" />

<ClInclude Include="StandardChars.h" />

<ClInclude Include="TextbookBoyerMoore.h" />

<ClInclude Include="tokens.h" />

</ItemGroup>

<ItemGroup>

<ClCompile Include="$(MSBuildThisFileDirectory)jserr.gen">

<PreprocessToFile>true</PreprocessToFile>

<PreprocessSuppressLineNumbers>true</PreprocessSuppressLineNumbers>

<ObjectFileName>$(IntDir)jserr.rc2</ObjectFileName>

<AdditionalIncludeDirectories>$(MSBuildThisFileDirectory);$(MSBuildThisFileDirectory)..\Common</AdditionalIncludeDirectories>

<LibCompiled>false</LibCompiled>

<ForcedIncludeFiles>

</ForcedIncludeFiles>

<PrecompiledHeader>NotUsing</PrecompiledHeader>

</ClCompile>

</ItemGroup>

<ItemGroup>

<None Include="jsscan.js" />

</ItemGroup>

<Import Project="$(BuildConfigPropsPath)Chakra.Build.targets" Condition="exists('$(BuildConfigPropsPath)Chakra.Build.targets')" />

<Import Project="$(VCTargetsPath)\Microsoft.Cpp.targets" />

</Project>

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

// 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 "..\Runtime\Base\WindowsGlobalizationAdapter.h"

using namespace Windows::Data::Text;

static const CharTypeFlags charFlags[128] =

{

UnknownChar, /\* 0x00 \*/

UnknownChar, /\* 0x01 \*/

UnknownChar, /\* 0x02 \*/

UnknownChar, /\* 0x03 \*/

UnknownChar, /\* 0x04 \*/

UnknownChar, /\* 0x05 \*/

UnknownChar, /\* 0x06 \*/

UnknownChar, /\* 0x07 \*/

UnknownChar, /\* 0x08 \*/

SpaceChar, /\* 0x09 \*/

LineCharGroup, /\* 0x0A \*/

SpaceChar, /\* 0x0B \*/

SpaceChar, /\* 0x0C \*/

LineCharGroup, /\* 0x0D \*/

UnknownChar, /\* 0x0E \*/

UnknownChar, /\* 0x0F \*/

UnknownChar, /\* 0x10 \*/

UnknownChar, /\* 0x11 \*/

UnknownChar, /\* 0x12 \*/

UnknownChar, /\* 0x13 \*/

UnknownChar, /\* 0x14 \*/

UnknownChar, /\* 0x15 \*/

UnknownChar, /\* 0x16 \*/

UnknownChar, /\* 0x17 \*/

UnknownChar, /\* 0x18 \*/

UnknownChar, /\* 0x19 \*/

UnknownChar, /\* 0x1A \*/

UnknownChar, /\* 0x1B \*/

UnknownChar, /\* 0x1C \*/

UnknownChar, /\* 0x1D \*/

UnknownChar, /\* 0x1E \*/

UnknownChar, /\* 0x1F \*/

SpaceChar, /\* 0x20 \*/

UnknownChar, /\* 0x21 ! \*/

UnknownChar, /\* 0x22 \*/

UnknownChar, /\* 0x23 # \*/

LetterCharGroup, /\* 0x24 $ \*/

UnknownChar, /\* 0x25 % \*/

UnknownChar, /\* 0x26 & \*/

UnknownChar, /\* 0x27 \*/

UnknownChar, /\* 0x28 \*/

UnknownChar, /\* 0x29 \*/

UnknownChar, /\* 0x2A \*/

UnknownChar, /\* 0x2B \*/

UnknownChar, /\* 0x2C \*/

UnknownChar, /\* 0x2D \*/

UnknownChar, /\* 0x2E \*/

UnknownChar, /\* 0x2F \*/

DecimalCharGroup, /\* 0x30 0 \*/

DecimalCharGroup, /\* 0x31 1 \*/

DecimalCharGroup, /\* 0x32 2 \*/

DecimalCharGroup, /\* 0x33 3 \*/

DecimalCharGroup, /\* 0x34 4 \*/

DecimalCharGroup, /\* 0x35 5 \*/

DecimalCharGroup, /\* 0x36 6 \*/

DecimalCharGroup, /\* 0x37 7 \*/

DecimalCharGroup, /\* 0x38 8 \*/

DecimalCharGroup, /\* 0x39 9 \*/

UnknownChar, /\* 0x3A \*/

UnknownChar, /\* 0x3B \*/

UnknownChar, /\* 0x3C < \*/

UnknownChar, /\* 0x3D = \*/

UnknownChar, /\* 0x3E > \*/

UnknownChar, /\* 0x3F \*/

UnknownChar, /\* 0x40 @ \*/

HexCharGroup, /\* 0x41 A \*/

HexCharGroup, /\* 0x42 B \*/

HexCharGroup, /\* 0x43 C \*/

HexCharGroup, /\* 0x44 D \*/

HexCharGroup, /\* 0x45 E \*/

HexCharGroup, /\* 0x46 F \*/

LetterCharGroup, /\* 0x47 G \*/

LetterCharGroup, /\* 0x48 H \*/

LetterCharGroup, /\* 0x49 I \*/

LetterCharGroup, /\* 0x4A J \*/

LetterCharGroup, /\* 0x4B K \*/

LetterCharGroup, /\* 0x4C L \*/

LetterCharGroup, /\* 0x4D M \*/

LetterCharGroup, /\* 0x4E N \*/

LetterCharGroup, /\* 0x4F O \*/

LetterCharGroup, /\* 0x50 P \*/

LetterCharGroup, /\* 0x51 Q \*/

LetterCharGroup, /\* 0x52 R \*/

LetterCharGroup, /\* 0x53 S \*/

LetterCharGroup, /\* 0x54 T \*/

LetterCharGroup, /\* 0x55 U \*/

LetterCharGroup, /\* 0x56 V \*/

LetterCharGroup, /\* 0x57 W \*/

LetterCharGroup, /\* 0x58 X \*/

LetterCharGroup, /\* 0x59 Y \*/

LetterCharGroup, /\* 0x5A Z \*/

UnknownChar, /\* 0x5B \*/

UnknownChar, /\* 0x5C \*/

UnknownChar, /\* 0x5D \*/

UnknownChar, /\* 0x5E \*/

LetterCharGroup, /\* 0x5F \_ \*/

UnknownChar, /\* 0x60 \*/

HexCharGroup, /\* 0x61 a \*/

HexCharGroup, /\* 0x62 b \*/

HexCharGroup, /\* 0x63 c \*/

HexCharGroup, /\* 0x64 d \*/

HexCharGroup, /\* 0x65 e \*/

HexCharGroup, /\* 0x66 f \*/

LetterCharGroup, /\* 0x67 g \*/

LetterCharGroup, /\* 0x68 h \*/

LetterCharGroup, /\* 0x69 i \*/

LetterCharGroup, /\* 0x6A j \*/

LetterCharGroup, /\* 0x6B k \*/

LetterCharGroup, /\* 0x6C l \*/

LetterCharGroup, /\* 0x6D m \*/

LetterCharGroup, /\* 0x6E n \*/

LetterCharGroup, /\* 0x6F o \*/

LetterCharGroup, /\* 0x70 p \*/

LetterCharGroup, /\* 0x71 q \*/

LetterCharGroup, /\* 0x72 r \*/

LetterCharGroup, /\* 0x73 s \*/

LetterCharGroup, /\* 0x74 t \*/

LetterCharGroup, /\* 0x75 u \*/

LetterCharGroup, /\* 0x76 v \*/

LetterCharGroup, /\* 0x77 w \*/

LetterCharGroup, /\* 0x78 x \*/

LetterCharGroup, /\* 0x79 y \*/

LetterCharGroup, /\* 0x7A z \*/

UnknownChar, /\* 0x7B \*/

UnknownChar, /\* 0x7C \*/

UnknownChar, /\* 0x7D \*/

UnknownChar, /\* 0x7E \*/

UnknownChar /\* 0x7F \*/

};

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

\*

\* The \_C\_xxx enum and charTypes[] table are used to map a character to

\* simple classification values and flags.

\*/

static const CharTypes charTypes[128] =

{

\_C\_NUL, \_C\_ERR, \_C\_ERR, \_C\_ERR, \_C\_ERR, \_C\_ERR, \_C\_ERR, \_C\_ERR, /\* 00-07 \*/

\_C\_ERR, \_C\_WSP, \_C\_NWL, \_C\_WSP, \_C\_WSP, \_C\_NWL, \_C\_ERR, \_C\_ERR, /\* 08-0F \*/

\_C\_ERR, \_C\_ERR, \_C\_ERR, \_C\_ERR, \_C\_ERR, \_C\_ERR, \_C\_ERR, \_C\_ERR, /\* 10-17 \*/

\_C\_ERR, \_C\_ERR, \_C\_ERR, \_C\_ERR, \_C\_ERR, \_C\_ERR, \_C\_ERR, \_C\_ERR, /\* 18-1F \*/

\_C\_WSP, \_C\_BNG, \_C\_QUO, \_C\_SHP, \_C\_DOL, \_C\_PCT, \_C\_AMP, \_C\_APO, /\* 20-27 \*/

\_C\_LPR, \_C\_RPR, \_C\_MUL, \_C\_PLS, \_C\_CMA, \_C\_MIN, \_C\_DOT, \_C\_SLH, /\* 28-2F \*/

\_C\_DIG, \_C\_DIG, \_C\_DIG, \_C\_DIG, \_C\_DIG, \_C\_DIG, \_C\_DIG, \_C\_DIG, /\* 30-37 \*/

\_C\_DIG, \_C\_DIG, \_C\_COL, \_C\_SMC, \_C\_LT , \_C\_EQ , \_C\_GT , \_C\_QUE, /\* 38-3F \*/

\_C\_AT , \_C\_LET, \_C\_LET, \_C\_LET, \_C\_LET, \_C\_LET, \_C\_LET, \_C\_LET, /\* 40-47 \*/

\_C\_LET, \_C\_LET, \_C\_LET, \_C\_LET, \_C\_LET, \_C\_LET, \_C\_LET, \_C\_LET, /\* 48-4F \*/

\_C\_LET, \_C\_LET, \_C\_LET, \_C\_LET, \_C\_LET, \_C\_LET, \_C\_LET, \_C\_LET, /\* 50-57 \*/

\_C\_LET, \_C\_LET, \_C\_LET, \_C\_LBR, \_C\_BSL, \_C\_RBR, \_C\_XOR, \_C\_USC, /\* 58-5F \*/

\_C\_BKQ, \_C\_LET, \_C\_LET, \_C\_LET, \_C\_LET, \_C\_LET, \_C\_LET, \_C\_LET, /\* 60-67 \*/

\_C\_LET, \_C\_LET, \_C\_LET, \_C\_LET, \_C\_LET, \_C\_LET, \_C\_LET, \_C\_LET, /\* 68-6F \*/

\_C\_LET, \_C\_LET, \_C\_LET, \_C\_LET, \_C\_LET, \_C\_LET, \_C\_LET, \_C\_LET, /\* 70-77 \*/

\_C\_LET, \_C\_LET, \_C\_LET, \_C\_LC , \_C\_BAR, \_C\_RC , \_C\_TIL, \_C\_ERR, /\* 78-7F \*/

};

typedef struct

{

OLECHAR chStart;

OLECHAR chFinish;

} oldCharTypesRangeStruct;

static const int cOldDigits = 156;

static const oldCharTypesRangeStruct oldDigits[] = {

{ 688, 734 }, { 736, 745 }, { 768, 837 }, { 864, 865 }, { 884, 885 },

{ 890, 890 }, { 900, 901 }, { 1154, 1158 }, { 1369, 1369 }, { 1425, 1441 },

{ 1443, 1465 }, { 1467, 1469 }, { 1471, 1471 }, { 1473, 1474 }, { 1476, 1476 },

{ 1600, 1600 }, { 1611, 1618 }, { 1648, 1648 }, { 1750, 1773 }, { 2305, 2307 },

{ 2364, 2381 }, { 2384, 2388 }, { 2402, 2403 }, { 2433, 2435 }, { 2492, 2492 },

{ 2494, 2500 }, { 2503, 2504 }, { 2507, 2509 }, { 2519, 2519 }, { 2530, 2531 },

{ 2546, 2554 }, { 2562, 2562 }, { 2620, 2620 }, { 2622, 2626 }, { 2631, 2632 },

{ 2635, 2637 }, { 2672, 2676 }, { 2689, 2691 }, { 2748, 2757 }, { 2759, 2761 },

{ 2763, 2765 }, { 2768, 2768 }, { 2817, 2819 }, { 2876, 2883 }, { 2887, 2888 },

{ 2891, 2893 }, { 2902, 2903 }, { 2928, 2928 }, { 2946, 2947 }, { 3006, 3010 },

{ 3014, 3016 }, { 3018, 3021 }, { 3031, 3031 }, { 3056, 3058 }, { 3073, 3075 },

{ 3134, 3140 }, { 3142, 3144 }, { 3146, 3149 }, { 3157, 3158 }, { 3202, 3203 },

{ 3262, 3268 }, { 3270, 3272 }, { 3274, 3277 }, { 3285, 3286 }, { 3330, 3331 },

{ 3390, 3395 }, { 3398, 3400 }, { 3402, 3405 }, { 3415, 3415 }, { 3647, 3647 },

{ 3759, 3769 }, { 3771, 3773 }, { 3776, 3780 }, { 3782, 3782 }, { 3784, 3789 },

{ 3840, 3843 }, { 3859, 3871 }, { 3882, 3897 }, { 3902, 3903 }, { 3953, 3972 },

{ 3974, 3979 }, { 8125, 8129 }, { 8141, 8143 }, { 8157, 8159 }, { 8173, 8175 },

{ 8189, 8190 }, { 8192, 8207 }, { 8232, 8238 }, { 8260, 8260 }, { 8298, 8304 },

{ 8308, 8316 }, { 8319, 8332 }, { 8352, 8364 }, { 8400, 8417 }, { 8448, 8504 },

{ 8531, 8578 }, { 8592, 8682 }, { 8704, 8945 }, { 8960, 8960 }, { 8962, 9000 },

{ 9003, 9082 }, { 9216, 9252 }, { 9280, 9290 }, { 9312, 9371 }, { 9450, 9450 },

{ 9472, 9621 }, { 9632, 9711 }, { 9728, 9747 }, { 9754, 9839 }, { 9985, 9988 },

{ 9990, 9993 }, { 9996, 10023 }, { 10025, 10059 }, { 10061, 10061 }, { 10063, 10066 },

{ 10070, 10070 }, { 10072, 10078 }, { 10081, 10087 }, { 10102, 10132 }, { 10136, 10159 },

{ 10161, 10174 }, { 12292, 12292 }, { 12294, 12294 }, { 12306, 12307 }, { 12320, 12335 },

{ 12337, 12343 }, { 12351, 12351 }, { 12441, 12442 }, { 12688, 12703 }, { 12800, 12828 },

{ 12832, 12867 }, { 12896, 12923 }, { 12927, 12976 }, { 12992, 13003 }, { 13008, 13054 },

{ 13056, 13174 }, { 13179, 13277 }, { 13280, 13310 }, { 64286, 64286 }, { 65056, 65059 },

{ 65122, 65122 }, { 65124, 65126 }, { 65129, 65129 }, { 65136, 65138 }, { 65140, 65140 },

{ 65142, 65151 }, { 65284, 65284 }, { 65291, 65291 }, { 65308, 65310 }, { 65342, 65342 },

{ 65344, 65344 }, { 65372, 65372 }, { 65374, 65374 }, { 65440, 65440 }, { 65504, 65510 },

{ 65512, 65518 }

};

static const int cOldAlphas = 11;

static const oldCharTypesRangeStruct oldAlphas[] = {

{ 402, 402 }, { 9372, 9449 }, { 12293, 12293 }, { 12295, 12295 }, { 12443, 12446 },

{ 12540, 12542 }, { 64297, 64297 }, { 65152, 65276 }, { 65392, 65392 }, { 65438, 65439 },

{ 65533, 65533 }

};

CharTypes GetBigCharType(codepoint\_t ch);

CharTypes GetBigCharTypeES6(codepoint\_t ch);

CharTypeFlags GetBigCharFlags(codepoint\_t ch, const Js::CharClassifier \*instance);

CharTypeFlags GetBigCharFlags5(codepoint\_t ch, const Js::CharClassifier \*instanceh);

CharTypeFlags GetBigCharFlagsES6(codepoint\_t ch, const Js::CharClassifier \*instance);

BOOL doBinSearch(OLECHAR ch, const oldCharTypesRangeStruct \*pRanges, int cSize)

{

int lo = 0;

int hi = cSize;

int mid;

while (lo != hi)

{

mid = lo + (hi - lo) / 2;

if (pRanges[mid].chStart <= ch && ch <= pRanges[mid].chFinish)

return true;

if (ch < pRanges[mid].chStart)

hi = mid;

else

lo = mid + 1;

}

return false;

}

WORD oFindOldCharType(OLECHAR ch)

{

if ((OLECHAR) 65279 == ch)

return C1\_SPACE;

if (doBinSearch(ch, oldAlphas, cOldAlphas))

return C1\_ALPHA;

if (doBinSearch(ch, oldDigits, cOldDigits))

return C1\_DIGIT;

return 0;

}

BOOL oGetCharType( DWORD dwInfoType, OLECHAR ch, LPWORD lpwCharType )

{

BOOL res = GetStringTypeW( dwInfoType, &ch, 1, lpwCharType );

// BOM ( 0xfeff) is recognized as GetStringTypeW as WS.

if ((0x03FF & \*lpwCharType) == 0x0200)

{

// Some of the char types changed for Whistler (Unicode 3.0).

// They will return 0x0200 on Whistler, indicating a defined char

// with no type attributes. We want to continue to support these

// characters, so we return the Win2K (Unicode 2.1) attributes.

// We only return the ones we care about - ALPHA for ALPHA, PUNCT

// for PUNCT or DIGIT, and SPACE for SPACE or BLANK.

WORD wOldCharType = oFindOldCharType(ch);

if (0 == wOldCharType)

return res;

\*lpwCharType = wOldCharType;

return TRUE;

}

return res;

}

CharTypes GetBigCharType(codepoint\_t ch, const Js::CharClassifier \*instance)

{

if(ch > 0xFFFF)

{

return CharTypes::\_C\_ERR;

}

OLECHAR oCh = (OLECHAR)ch;

WORD chType;

Assert( oCh >= 128 );

#if (\_WIN32 || \_WIN64) // We use the Win32 API function GetStringTypeW for Unicode char. classification

if( oCh == 0x2028 || oCh == 0x2029 )

{

return \_C\_NWL;

}

if( oGetCharType( CT\_CTYPE1, oCh, &chType) )

{

if( chType & C1\_ALPHA )

return \_C\_LET;

else if( chType & (C1\_SPACE|C1\_BLANK) )

return \_C\_WSP;

}

#else

#warning No Unicode character support on this platform

#endif

return \_C\_ERR;

}

CharTypeFlags GetBigCharFlags(codepoint\_t ch, const Js::CharClassifier \*instance)

{

WORD chType;

if(ch > 0xFFFF)

{

return CharTypeFlags::UnknownChar;

}

OLECHAR oCh = (OLECHAR)ch;

Assert( oCh >= 128 );

#if (\_WIN32 || \_WIN64) // We use the Win32 API function GetStringTypeW for Unicode char. classification

if( oCh == kchLS || oCh == kchPS )

{

return LineCharGroup;

}

if( oGetCharType( CT\_CTYPE1, oCh, &chType) )

{

if( chType & C1\_ALPHA )

return LetterCharGroup;

else if ( chType & (C1\_DIGIT|C1\_PUNCT) )

{

// non-ANSI digits can be used in identifiers but not in numeric constants - hence we

// return fChId instead of kgrfchDec

return IdChar;

}

else if( chType & (C1\_SPACE|C1\_BLANK) )

return SpaceChar;

}

#else

#warning No Unicode character support on this platform

#endif

return UnknownChar;

}

CharTypeFlags GetBigCharFlags5(codepoint\_t ch, const Js::CharClassifier \*instance)

{

//In ES5 the unicode <ZWNJ> and <ZWJ> could be identifier parts

if(ch == 0x200c || ch == 0x200d)

{

return IdChar;

}

return GetBigCharFlags(ch, instance);

}

/\*

\* CharClassifier implementation

\*/

UnicodeGeneralCategory Js::CharClassifier::GetUnicodeCategoryFor(codepoint\_t ch) const

{

UnicodeGeneralCategory category;

AssertMsg(this->winGlobCharApi != nullptr, "ES6 Mode 'GetUnicodeCategoryFor' must mean winGlobCharApi is initialized.");

if(FAILED(this->winGlobCharApi->GetGeneralCategory(ch, &category)))

{

AssertMsg(false, "Should not fail here!");

return UnicodeGeneralCategory::UnicodeGeneralCategory\_NotAssigned;

}

return category;

}

CharTypes Js::CharClassifier::GetBigCharTypeES6(codepoint\_t ch, const Js::CharClassifier \*instance)

{

Assert(ch > 0x7F);

UnicodeGeneralCategory category = instance->GetUnicodeCategoryFor(ch);

switch(category)

{

case UnicodeGeneralCategory::UnicodeGeneralCategory\_LowercaseLetter:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_UppercaseLetter:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_TitlecaseLetter:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_ModifierLetter:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_OtherLetter:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_LetterNumber:

return CharTypes::\_C\_LET;

case UnicodeGeneralCategory::UnicodeGeneralCategory\_LineSeparator:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_ParagraphSeparator:

return CharTypes::\_C\_NWL;

case UnicodeGeneralCategory::UnicodeGeneralCategory\_SpaceSeparator:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_SpacingCombiningMark:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_NonspacingMark:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_ConnectorPunctuation:

return CharTypes::\_C\_WSP;

case UnicodeGeneralCategory::UnicodeGeneralCategory\_DecimalDigitNumber:

return CharTypes::\_C\_DIG;

case UnicodeGeneralCategory::UnicodeGeneralCategory\_ClosePunctuation:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_EnclosingMark:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_Control:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_Format:

if (ch == 0xFEFF)

{

return CharTypes::\_C\_WSP;

}

// Fall through, otherwise

case UnicodeGeneralCategory::UnicodeGeneralCategory\_Surrogate:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_PrivateUse:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_DashPunctuation:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_OpenPunctuation:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_InitialQuotePunctuation:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_FinalQuotePunctuation:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_OtherPunctuation:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_MathSymbol:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_CurrencySymbol:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_ModifierSymbol:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_OtherSymbol:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_NotAssigned:

return CharTypes::\_C\_UNK;

}

return CharTypes::\_C\_UNK;

}

/\*

From Unicode 6.3 http://www.unicode.org/reports/tr31/tr31-19.html

ID\_Start:::

Characters having the Unicode General\_Category of uppercase letters (Lu), lowercase letters (Ll), titlecase letters (Lt), modifier letters (Lm), other letters (Lo), letter numbers (Nl), minus Pattern\_Syntax and Pattern\_White\_Space code points, plus stability extensions. Note that "other letters" includes ideographs.

In set notation, this is [[:L:][:Nl:]--[:Pattern\_Syntax:]--[:Pattern\_White\_Space:]] plus stability extensions.

ID\_Continue:::

All of the above, plus characters having the Unicode General\_Category of nonspacing marks (Mn), spacing combining marks (Mc), decimal number (Nd), connector punctuations (Pc), plus stability extensions, minus Pattern\_Syntax and Pattern\_White\_Space code points.

In set notation, this is [[:L:][:Nl:][:Mn:][:Mc:][:Nd:][:Pc:]--[:Pattern\_Syntax:]--[:Pattern\_White\_Space:]] plus stability extensions.

These are also known simply as Identifier Characters, because they are a superset of the ID\_Start characters.

\*/

CharTypeFlags Js::CharClassifier::GetBigCharFlagsES6(codepoint\_t ch, const Js::CharClassifier \*instance)

{

Assert(ch > 0x7F);

UnicodeGeneralCategory category = instance->GetUnicodeCategoryFor(ch);

switch(category)

{

case UnicodeGeneralCategory::UnicodeGeneralCategory\_LowercaseLetter:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_UppercaseLetter:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_TitlecaseLetter:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_ModifierLetter:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_OtherLetter:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_LetterNumber:

return BigCharIsIdStartES6(ch, instance) ? CharTypeFlags::LetterCharGroup : CharTypeFlags::UnknownChar;

case UnicodeGeneralCategory::UnicodeGeneralCategory\_SpacingCombiningMark:

return BigCharIsIdContinueES6(ch, instance) ? CharTypeFlags::IdChar : CharTypeFlags::SpaceChar;

case UnicodeGeneralCategory::UnicodeGeneralCategory\_NonspacingMark:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_ConnectorPunctuation:

return BigCharIsIdContinueES6(ch, instance) ? CharTypeFlags::IdChar : CharTypeFlags::UnknownChar;

case UnicodeGeneralCategory::UnicodeGeneralCategory\_DecimalDigitNumber:

return BigCharIsIdContinueES6(ch, instance) ? CharTypeFlags::DecimalCharGroup : CharTypeFlags::DecimalChar;

case UnicodeGeneralCategory::UnicodeGeneralCategory\_LineSeparator:

return CharTypeFlags::LineFeedChar;

case UnicodeGeneralCategory::UnicodeGeneralCategory\_ParagraphSeparator:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_SpaceSeparator:

return CharTypeFlags::SpaceChar;

case UnicodeGeneralCategory::UnicodeGeneralCategory\_ClosePunctuation:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_EnclosingMark:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_Control:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_Format:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_Surrogate:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_PrivateUse:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_DashPunctuation:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_OpenPunctuation:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_InitialQuotePunctuation:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_FinalQuotePunctuation:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_OtherPunctuation:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_MathSymbol:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_CurrencySymbol:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_ModifierSymbol:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_OtherSymbol:

case UnicodeGeneralCategory::UnicodeGeneralCategory\_NotAssigned:

return CharTypeFlags::UnknownChar;

}

return CharTypeFlags::UnknownChar;

}

BOOL Js::CharClassifier::BigCharIsWhitespaceES6(codepoint\_t ch, const CharClassifier \*instance)

{

Assert(ch > 0x7F);

if (ch == 0xFEFF)

{

return true;

}

boolean toReturn = false;

AssertMsg(instance->winGlobCharApi != nullptr, "ES6 Mode 'BigCharIsWhitespaceES6' must mean winGlobCharApi is initialized.");

if (FAILED(instance->winGlobCharApi->IsWhitespace(ch, &toReturn)))

{

AssertMsg(false, "Should not fail here!");

return toReturn;

}

return toReturn;

}

BOOL Js::CharClassifier::BigCharIsIdStartES6(codepoint\_t codePoint, const CharClassifier \*instance)

{

Assert(codePoint > 0x7F);

boolean toReturn = false;

AssertMsg(instance->winGlobCharApi != nullptr, "ES6 Mode 'BigCharIsIdStartES6' must mean winGlobCharApi is initialized.");

if (FAILED(instance->winGlobCharApi->IsIdStart(codePoint, &toReturn)))

{

AssertMsg(false, "Should not fail here!");

return toReturn;

}

return toReturn;

}

BOOL Js::CharClassifier::BigCharIsIdContinueES6(codepoint\_t codePoint, const CharClassifier \*instance)

{

Assert(codePoint > 0x7F);

if (codePoint == '$' || codePoint == '\_' || codePoint == 0x200C /\* Zero-width non-joiner \*/ || codePoint == 0x200D /\* Zero-width joiner \*/)

{

return true;

}

boolean toReturn = false;

AssertMsg(instance->winGlobCharApi != nullptr, "ES6 Mode 'BigCharIsIdContinueES6' must mean winGlobCharApi is initialized.");

if (FAILED(instance->winGlobCharApi->IsIdContinue(codePoint, &toReturn)))

{

AssertMsg(false, "Should not fail here!");

return toReturn;

}

return toReturn;

}

template <bool isBigChar>

BOOL Js::CharClassifier::IsWhiteSpaceFast(codepoint\_t ch) const

{

Assert(isBigChar ? ch > 0x7F : ch < 0x80);

return isBigChar ? this->bigCharIsWhitespaceFunc(ch, this) : (charFlags[ch] & CharTypeFlags::SpaceChar);

}

BOOL Js::CharClassifier::IsBiDirectionalChar(codepoint\_t ch) const

{

//From http://www.unicode.org/reports/tr9/#Directional\_Formatting\_Codes

switch (ch)

{

case 0x202A: //LEFT-TO-RIGHT EMBEDDING Treat the following text as embedded left-to-right

case 0x202B: //RIGHT-TO-LEFT EMBEDDING Treat the following text as embedded right-to-left.

case 0x202D: //LEFT-TO-RIGHT OVERRIDE Force following characters to be treated as strong left-to-right characters.

case 0x202E: //RIGHT-TO-LEFT OVERRIDE Force following characters to be treated as strong right-to-left characters.

case 0x202C: //POP DIRECTIONAL FORMATTING End the scope of the last LRE, RLE, RLO, or LRO.

case 0x2066: //LEFT-TO-RIGHT ISOLATE Treat the following text as isolated and left-to-right.

case 0x2067: //RIGHT-TO-LEFT ISOLATE Treat the following text as isolated and right-to-left.

case 0x2068: //FIRST STRONG ISOLATE Treat the following text as isolated and in the direction of its first strong directional character that is not inside a nested isolate.

case 0x2069: //POP DIRECTIONAL ISOLATE End the scope of the last LRI, RLI, or FSI.

case 0x200E: //LEFT-TO-RIGHT MARK Left-to-right zero-width character

case 0x200F: //RIGHT-TO-LEFT MARK Right-to-left zero-width non-Arabic character

case 0x061C: //ARABIC LETTER MARK Right-to-left zero-width Arabic character

return TRUE;

default:

return FALSE;

}

}

template<bool isBigChar>

BOOL Js::CharClassifier::IsIdStartFast(codepoint\_t ch) const

{

Assert(isBigChar ? ch > 0x7F : ch < 0x80);

return isBigChar ? this->bigCharIsIdStartFunc(ch, this) : (charFlags[ch] & CharTypeFlags::IdLeadChar);

}

template<bool isBigChar>

BOOL Js::CharClassifier::IsIdContinueFast(codepoint\_t ch) const

{

Assert(isBigChar ? ch > 0x7F : ch < 0x80);

return isBigChar ? this->bigCharIsIdContinueFunc(ch, this) : (charFlags[ch] & CharTypeFlags::IdChar);

}

Js::CharClassifier::CharClassifier(ScriptContext \* scriptContext)

{

CharClassifierModes overallMode = (CONFIG\_FLAG(ES6Unicode)) ? CharClassifierModes::ES6 : CharClassifierModes::ES5;

bool codePointSupport = overallMode == CharClassifierModes::ES6;

bool isES6UnicodeVerboseEnabled = scriptContext->GetConfig()->IsES6UnicodeVerboseEnabled();

initClassifier(scriptContext, overallMode, overallMode, overallMode, codePointSupport, isES6UnicodeVerboseEnabled, CharClassifierModes::ES6); // no fallback for chk

}

void Js::CharClassifier::initClassifier(ScriptContext \* scriptContext, CharClassifierModes identifierSupport,

CharClassifierModes whiteSpaceSupport, CharClassifierModes generalCharClassificationSupport, bool codePointSupport, bool isES6UnicodeVerboseEnabled, CharClassifierModes es6FallbackMode)

{

bool es6Supported = true;

bool es6ModeNeeded = identifierSupport == CharClassifierModes::ES6 || whiteSpaceSupport == CharClassifierModes::ES6 || generalCharClassificationSupport == CharClassifierModes::ES6;

#ifdef ENABLE\_ES6\_CHAR\_CLASSIFIER

ThreadContext\* threadContext = scriptContext->GetThreadContext();

Js::WindowsGlobalizationAdapter\* globalizationAdapter = threadContext->GetWindowsGlobalizationAdapter();

Js::DelayLoadWindowsGlobalization\* globLibrary = threadContext->GetWindowsGlobalizationLibrary();

if (es6ModeNeeded)

{

HRESULT hr = globalizationAdapter->EnsureDataTextObjectsInitialized(globLibrary);

if (FAILED(hr))

{

AssertMsg(false, "Failed to initialize COM interfaces, verify correct version of globalization dll is used.");

JavascriptError::MapAndThrowError(scriptContext, hr);

}

this->winGlobCharApi = globalizationAdapter->GetUnicodeStatics();

if (this->winGlobCharApi == nullptr)

{

// No fallback mode, then assert

if (es6FallbackMode == CharClassifierModes::ES6)

{

AssertMsg(false, "Windows::Data::Text::IUnicodeCharactersStatics not initialized");

//Fallback to ES5 just in case for fre builds.

es6FallbackMode = CharClassifierModes::ES5;

}

if (isES6UnicodeVerboseEnabled)

{

Output::Print(L"Windows::Data::Text::IUnicodeCharactersStatics not initialized\r\n");

}

//Default to non-es6

es6Supported = false;

}

}

#else

es6Supported = false;

es6FallbackMode = CharClassifierModes::ES5;

#endif

if (es6ModeNeeded && !es6Supported)

{

identifierSupport = identifierSupport == CharClassifierModes::ES6 ? es6FallbackMode : identifierSupport;

whiteSpaceSupport = whiteSpaceSupport == CharClassifierModes::ES6 ? es6FallbackMode : whiteSpaceSupport;

generalCharClassificationSupport = generalCharClassificationSupport == CharClassifierModes::ES6 ? es6FallbackMode : generalCharClassificationSupport;

}

bigCharIsIdStartFunc = identifierSupport == CharClassifierModes::ES6 ? &CharClassifier::BigCharIsIdStartES6 : &CharClassifier::BigCharIsIdStartDefault;

bigCharIsIdContinueFunc = identifierSupport == CharClassifierModes::ES6 ? &CharClassifier::BigCharIsIdContinueES6 : &CharClassifier::BigCharIsIdContinueDefault;

bigCharIsWhitespaceFunc = whiteSpaceSupport == CharClassifierModes::ES6 ? &CharClassifier::BigCharIsWhitespaceES6 : &CharClassifier::BigCharIsWhitespaceDefault;

skipWhiteSpaceFunc = codePointSupport ? &CharClassifier::SkipWhiteSpaceSurrogate : &CharClassifier::SkipWhiteSpaceNonSurrogate;

skipWhiteSpaceStartEndFunc = codePointSupport ? &CharClassifier::SkipWhiteSpaceSurrogateStartEnd : &CharClassifier::SkipWhiteSpaceNonSurrogateStartEnd;

skipIdentifierFunc = codePointSupport ? &CharClassifier::SkipIdentifierSurrogate : &CharClassifier::SkipIdentifierNonSurrogate;

skipIdentifierStartEndFunc = codePointSupport ? &CharClassifier::SkipIdentifierSurrogateStartEnd : &CharClassifier::SkipIdentifierNonSurrogateStartEnd;

if (generalCharClassificationSupport == CharClassifierModes::ES6)

{

getBigCharTypeFunc = &CharClassifier::GetBigCharTypeES6;

getBigCharFlagsFunc = &CharClassifier::GetBigCharFlagsES6;

}

else if (generalCharClassificationSupport == CharClassifierModes::ES5)

{

getBigCharTypeFunc = &GetBigCharType;

getBigCharFlagsFunc = &GetBigCharFlags5;

}

else

{

getBigCharTypeFunc = &GetBigCharType;

getBigCharFlagsFunc = &GetBigCharFlags;

}

}

const OLECHAR\* Js::CharClassifier::SkipWhiteSpaceNonSurrogate(LPCOLESTR psz, const CharClassifier \*instance)

{

for ( ; instance->IsWhiteSpace(\*psz); psz++)

{

}

return psz;

}

const OLECHAR\* Js::CharClassifier::SkipWhiteSpaceNonSurrogateStartEnd(\_In\_reads\_(pStrEnd - pStr) LPCOLESTR pStr, \_In\_ LPCOLESTR pStrEnd, const CharClassifier \*instance)

{

for ( ; instance->IsWhiteSpace(\*pStr) && pStr < pStrEnd; pStr++)

{

}

return pStr;

}

const OLECHAR\* Js::CharClassifier::SkipIdentifierNonSurrogate(LPCOLESTR psz, const CharClassifier \*instance)

{

if (!instance->IsIdStart(\*psz))

{

return psz;

}

for (psz++; instance->IsIdContinue(\*psz); psz++)

{

}

return psz;

}

const LPCUTF8 Js::CharClassifier::SkipIdentifierNonSurrogateStartEnd(LPCUTF8 psz, LPCUTF8 end, const CharClassifier \*instance)

{

utf8::DecodeOptions options = utf8::doAllowThreeByteSurrogates;

LPCUTF8 p = psz;

if (!instance->IsIdStart(utf8::Decode(p, end, options)))

{

return psz;

}

psz = p;

while (instance->IsIdContinue(utf8::Decode(p, end, options)))

{

psz = p;

}

return psz;

}

const OLECHAR\* Js::CharClassifier::SkipWhiteSpaceSurrogate(LPCOLESTR psz, const CharClassifier \*instance)

{

wchar\_t currentChar = 0x0;

// Slow path is to check for a surrogate each iteration.

// There is no new surrogate whitespaces as of yet, however, might be in the future, so surrogates still need to be checked

// So, based on that, best way is to hit the slow path if the current character is not a whitespace in [0, FFFF];

while((currentChar = \*psz) != '\0')

{

if (!instance->IsWhiteSpace(\*psz))

{

if (Js::NumberUtilities::IsSurrogateLowerPart(currentChar) && Js::NumberUtilities::IsSurrogateUpperPart(\*(psz + 1)))

{

if (instance->IsWhiteSpace(Js::NumberUtilities::SurrogatePairAsCodePoint(currentChar, \*(psz + 1))))

{

psz += 2;

continue;

}

}

// Above case failed, so we have reached the last whitespace

return psz;

}

psz++;

}

return psz;

}

const OLECHAR\* Js::CharClassifier::SkipWhiteSpaceSurrogateStartEnd(\_In\_reads\_(pStrEnd - pStr) LPCOLESTR pStr, \_In\_ LPCOLESTR pStrEnd, const CharClassifier \*instance)

{

wchar\_t currentChar = 0x0;

// Same reasoning as above

while(pStr < pStrEnd && (currentChar = \*pStr) != '\0')

{

if (!instance->IsWhiteSpace(currentChar))

{

if (Js::NumberUtilities::IsSurrogateLowerPart(currentChar) && (pStr + 1) < pStrEnd && Js::NumberUtilities::IsSurrogateUpperPart(\*(pStr + 1)))

{

if (instance->IsWhiteSpace(Js::NumberUtilities::SurrogatePairAsCodePoint(currentChar, \*(pStr + 1))))

{

pStr += 2;

continue;

}

}

// Above case failed, so we have reached the last whitespace

return pStr;

}

pStr++;

}

return pStr;

}

const OLECHAR\* Js::CharClassifier::SkipIdentifierSurrogate(LPCOLESTR psz, const CharClassifier \*instance)

{

// Similar reasoning to above, however we do have surrogate identifiers, but less likely to occur in code.

wchar\_t currentChar = \*psz;

if (!instance->IsIdStart(currentChar))

{

if (Js::NumberUtilities::IsSurrogateLowerPart(currentChar) && Js::NumberUtilities::IsSurrogateUpperPart(\*(psz + 1))

&& instance->IsIdStart(Js::NumberUtilities::SurrogatePairAsCodePoint(currentChar, \*(psz + 1))))

{

// For the extra surrogate char

psz ++;

}

else

{

return psz;

}

}

psz++;

while((currentChar = \*psz) != '\0')

{

if (!instance->IsIdContinue(\*psz))

{

if (Js::NumberUtilities::IsSurrogateLowerPart(currentChar) && Js::NumberUtilities::IsSurrogateUpperPart(\*(psz + 1)))

{

if (instance->IsIdContinue(Js::NumberUtilities::SurrogatePairAsCodePoint(currentChar, \*(psz + 1))))

{

psz += 2;

continue;

}

}

// Above case failed, so we have reached the last IDContinue

return psz;

}

psz++;

}

return psz;

}

const LPCUTF8 Js::CharClassifier::SkipIdentifierSurrogateStartEnd(LPCUTF8 psz, LPCUTF8 end, const CharClassifier \*instance)

{

LPCUTF8 currentPosition = psz;

utf8::DecodeOptions options = utf8::doAllowThreeByteSurrogates;

// Similar reasoning to above, however we do have surrogate identifiers, but less likely to occur in code.

codepoint\_t currentChar = utf8::Decode(currentPosition, end, options);

if (options & utf8::doSecondSurrogatePair)

{

currentChar = Js::NumberUtilities::SurrogatePairAsCodePoint(currentChar, utf8::Decode(currentPosition, end, options));

}

if (!instance->IsIdStart(currentChar))

{

return psz;

}

psz = currentPosition;

// Slow path is to check for a surrogate each iteration.

// There is no new surrogate whitespaces as of yet, however, might be in the future, so surrogates still need to be checked

// So, based on that, best way is to hit the slow path if the current character is not a whitespace in [0, FFFF];

while((currentChar = utf8::Decode(currentPosition, end, options)) != '\0')

{

if (options & utf8::doSecondSurrogatePair)

{

currentChar = Js::NumberUtilities::SurrogatePairAsCodePoint(currentChar, utf8::Decode(currentPosition, end, options));

}

if (!instance->IsIdContinue(currentChar))

{

return psz;

}

psz = currentPosition;

}

return psz;

}

CharTypes Js::CharClassifier::GetCharType(codepoint\_t ch) const

{

return FBigChar(ch) ? getBigCharTypeFunc(ch, this) : charTypes[ch];

}

CharTypeFlags Js::CharClassifier::GetCharFlags(codepoint\_t ch) const

{

return FBigChar(ch) ? getBigCharFlagsFunc(ch, this) : charFlags[ch];

}

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

// 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

#ifdef NTBUILD

#include "Windows.Globalization.h"

#else

#include "Windows.Data.Text.h"

using namespace ABI;

#endif

//Helpers

static inline BOOL FBigChar(codepoint\_t ch) { return ch >= 128u; }

static inline BOOL BoolFromDbl(double dbl) { return !Js::NumberUtilities::IsNan(dbl) && (0 != dbl); }

enum CharTypeFlags : uint

{

UnknownChar = 0x0,

IdChar = 0x01,

IdLeadChar = 0x02,

HexChar = 0x04,

DecimalChar = 0x08,

SpaceChar = 0x10,

LineFeedChar = 0x20,

LineCharGroup = SpaceChar | LineFeedChar,

LetterCharGroup = IdChar | IdLeadChar,

HexCharGroup = IdChar | IdLeadChar | HexChar,

DecimalCharGroup = IdChar | DecimalChar,

};

enum CharTypes

{

\_C\_UNK, // Unkown grouping

\_C\_ERR, // illegal character

\_C\_NUL, // NUL character

\_C\_LET, // letter (A-Z,a-z)

\_C\_DIG, // digit (0-9)

\_C\_WSP, // white space

\_C\_NWL, // new line

\_C\_DOL, // $

\_C\_BSL, // \ (backslash)

\_C\_BKQ, // `

\_C\_AT, // @

\_C\_SHP, // #

\_C\_BNG, // !

\_C\_QUO, // "

\_C\_APO, // '

\_C\_PCT, // %

\_C\_AMP, // &

\_C\_LPR, // (

\_C\_RPR, // )

\_C\_PLS, // +

\_C\_MIN, // -

\_C\_MUL, // \*

\_C\_SLH, // /

\_C\_XOR, // ^

\_C\_CMA, // ,

\_C\_DOT, // .

\_C\_LT, // <

\_C\_EQ, // =

\_C\_GT, // >

\_C\_QUE, // ?

\_C\_LBR, // [

\_C\_RBR, // ]

\_C\_USC, // \_

\_C\_LC, // {

\_C\_RC, // }

\_C\_BAR, // |

\_C\_TIL, // ~

\_C\_COL, // :

\_C\_SMC, // ;

};

enum

{

kchNUL = 0x00,

kchNWL = 0x0A,

kchRET = 0x0D,

kchBSL = '\\',

kchSHP = '#',

kchBNG = '!',

kchQUO = '"',

kchAPO = '\'',

kchPCT = '%',

kchAMP = '&',

kchLPR = '(',

kchRPR = ')',

kchPLS = '+',

kchMIN = '-',

kchMUL = '\*',

kchSLH = '/',

kchXOR = '^',

kchCMA = ',',

kchDOT = '.',

kchLT = '<',

kchEQ = '=',

kchGT = '>',

kchQUE = '?',

kchLBR = '[',

kchRBR = ']',

kchUSC = '\_',

kchLC = '{',

kchRC = '}',

kchBAR = '|',

kchTIL = '~',

kchCOL = ':',

kchSMC = ';',

kchLS = 0x2028, //classifies as new line

kchPS = 0x2029 //classifies as new line

};

namespace Js

{

class WindowsGlobalizationAdapter;

class DelayLoadWindowsGlobalization;

typedef

enum CharClassifierModes {

ES5 = 1,

ES6 = 2

} CharClassifierModes;

class CharClassifier

{

private:

Windows::Data::Text::IUnicodeCharactersStatics\* winGlobCharApi;

static BOOL BigCharIsWhitespaceDefault(codepoint\_t ch, const CharClassifier \*instance)

{

return (instance->getBigCharFlagsFunc(ch, instance) & CharTypeFlags::SpaceChar);

}

static BOOL BigCharIsIdStartDefault(codepoint\_t ch, const CharClassifier \*instance)

{

return (instance->getBigCharFlagsFunc(ch, instance) & CharTypeFlags::IdLeadChar);

}

static BOOL BigCharIsIdContinueDefault(codepoint\_t ch, const CharClassifier \*instance)

{

return (instance->getBigCharFlagsFunc(ch, instance) & CharTypeFlags::IdChar);

}

static BOOL BigCharIsWhitespaceES6(codepoint\_t ch, const CharClassifier \*instance);

static BOOL BigCharIsIdStartES6(codepoint\_t codePoint, const CharClassifier \*instance);

static BOOL BigCharIsIdContinueES6(codepoint\_t codePoint, const CharClassifier \*instance);

static CharTypes GetBigCharTypeES6(codepoint\_t ch, const CharClassifier \*instance);

static CharTypeFlags GetBigCharFlagsES6(codepoint\_t ch, const CharClassifier \*instance);

static const OLECHAR\* SkipWhiteSpaceSurrogate(LPCOLESTR psz, const CharClassifier \*instance);

static const OLECHAR\* SkipWhiteSpaceSurrogateStartEnd(\_In\_reads\_(pStrEnd - pStr) LPCOLESTR pStr, \_In\_ LPCOLESTR pStrEnd, const CharClassifier \*instance);

static const OLECHAR\* SkipIdentifierSurrogate(LPCOLESTR psz, const CharClassifier \*instance);

static const LPCUTF8 SkipIdentifierSurrogateStartEnd(LPCUTF8 psz, LPCUTF8 end, const CharClassifier \*instance);

static const OLECHAR\* SkipWhiteSpaceNonSurrogate(LPCOLESTR psz, const CharClassifier \*instance);

static const OLECHAR\* SkipWhiteSpaceNonSurrogateStartEnd(\_In\_reads\_(pStrEnd - pStr) LPCOLESTR pStr, \_In\_ LPCOLESTR pStrEnd, const CharClassifier \*instance);

static const OLECHAR\* SkipIdentifierNonSurrogate(LPCOLESTR psz, const CharClassifier \*instance);

static const LPCUTF8 SkipIdentifierNonSurrogateStartEnd(LPCUTF8 psz, LPCUTF8 end, const CharClassifier \*instance);

Windows::Data::Text::UnicodeGeneralCategory GetUnicodeCategoryFor(codepoint\_t ch) const;

CharTypes (\*getBigCharTypeFunc)(codepoint\_t ch, const CharClassifier \*instance);

CharTypeFlags (\*getBigCharFlagsFunc)(codepoint\_t ch, const CharClassifier \*instance);

BOOL (\*bigCharIsWhitespaceFunc)(codepoint\_t ch, const CharClassifier \*instance);

BOOL (\*bigCharIsIdStartFunc)(codepoint\_t ch, const CharClassifier \*instance);

BOOL (\*bigCharIsIdContinueFunc)(codepoint\_t ch, const CharClassifier \*instance);

const OLECHAR\* (\*skipWhiteSpaceFunc)(LPCOLESTR psz, const CharClassifier\* instance);

const OLECHAR\* (\*skipWhiteSpaceStartEndFunc)(LPCOLESTR pStr, LPCOLESTR pStrEnd, const CharClassifier\* instance);

const OLECHAR\* (\*skipIdentifierFunc)(LPCOLESTR pcz, const CharClassifier\* instance);

const LPCUTF8 (\*skipIdentifierStartEndFunc)(LPCUTF8 psz, LPCUTF8 end, const CharClassifier\* instance);

void initClassifier(ScriptContext\* scriptContext, CharClassifierModes identifierSupport,

CharClassifierModes whiteSpaceSupport, CharClassifierModes generalCharClassificationSupport, bool codePointSupport, bool isES6UnicodeVerboseEnabled, CharClassifierModes fallbackMode = CharClassifierModes::ES5);

public:

CharTypes GetCharType(codepoint\_t ch) const;

CharTypeFlags GetCharFlags(codepoint\_t ch) const;

template <bool isBigChar>

BOOL IsWhiteSpaceFast(codepoint\_t ch) const;

BOOL IsWhiteSpace(codepoint\_t ch) const

{

return FBigChar(ch) ? IsWhiteSpaceFast<true>(ch) : IsWhiteSpaceFast<false>(ch);

}

BOOL IsBiDirectionalChar(codepoint\_t ch) const;

template<bool isBigChar>

BOOL IsIdStartFast(codepoint\_t ch) const;

BOOL IsIdStart(codepoint\_t ch) const

{

return FBigChar(ch) ? IsIdStartFast<true>(ch) : IsIdStartFast<false>(ch);

}

template<bool isBigChar>

BOOL IsIdContinueFast(codepoint\_t ch) const;

BOOL IsIdContinue(codepoint\_t ch) const

{

return FBigChar(ch) ? IsIdContinueFast<true>(ch) : IsIdContinueFast<false>(ch);

}

const size\_t SkipBiDirectionalChars(\_In\_z\_bytecount\_(2 \* length) LPCOLESTR psz, \_In\_ size\_t startIndex, \_In\_ size\_t length) const

{

size\_t count = 0;

while (startIndex < length)

{

if (!IsBiDirectionalChar(psz[startIndex + count]))

{

return count;

}

count++;

}

return count;

}

const wchar\_t SkipBiDirectionalChars(\_In\_z\_ wchar\_t\* &pszRef) const

{

while (\*pszRef != '\0')

{

if (!IsBiDirectionalChar(\*pszRef))

{

return \*pszRef;

}

pszRef++;

}

return '\0';

}

const OLECHAR\* SkipWhiteSpace(LPCOLESTR psz) const

{

// Fast path for the case in which first character is not space

wchar\_t firstChar = \*psz;

if (firstChar == 0)

{

return psz;

}

if (!this->IsWhiteSpace(firstChar) &&

(skipWhiteSpaceFunc != &SkipWhiteSpaceSurrogate

|| !Js::NumberUtilities::IsSurrogateLowerPart(firstChar)))

{

return psz;

}

return skipWhiteSpaceFunc(psz, this);

}

const OLECHAR\* SkipWhiteSpace(\_In\_reads\_(pStrEnd - pStr) LPCOLESTR pStr, \_In\_ LPCOLESTR pStrEnd) const

{

// Fast path for the case in which first character is not space

if (pStr == pStrEnd)

{

return pStr;

}

wchar\_t firstChar = \*pStr;

if (!this->IsWhiteSpace(firstChar) &&

(skipWhiteSpaceStartEndFunc != &SkipWhiteSpaceSurrogateStartEnd

|| !Js::NumberUtilities::IsSurrogateLowerPart(firstChar)))

{

return pStr;

}

return skipWhiteSpaceStartEndFunc(pStr, pStrEnd, this);

}

const OLECHAR\* SkipIdentifier(LPCOLESTR psz) const

{

return skipIdentifierFunc(psz, this);

}

const LPCUTF8 SkipIdentifier(LPCUTF8 psz, LPCUTF8 end) const

{

return skipIdentifierStartEndFunc(psz, end, this);

}

const OLECHAR\* SkipIdentifier(LPCOLESTR psz, LPCOLESTR end) const

{

return SkipIdentifier(psz);

}

CharClassifier(Js::ScriptContext\* scriptContext);

};

};

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

// 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 UnifiedRegex

{

enum CharMapScheme

{

CharMapScheme\_Linear,

CharMapScheme\_Full

};

template <typename C, typename V, CharMapScheme scheme = CharMapScheme\_Full>

class CharMap {};

template <typename V, CharMapScheme scheme>

class CharMap<char, V, scheme> : private Chars<char>

{

private:

V map[NumChars];

public:

CharMap(V defv)

{

for (int i = 0; i < NumChars; i++)

map[i] = defv;

}

void FreeBody(ArenaAllocator\* allocator)

{

}

inline void Set(ArenaAllocator\* allocator, Char k, V v)

{

map[CTU(k)] = v;

}

inline V Get(Char k) const

{

return map[CTU(k)];

}

};

static const uint MaxCharMapLinearChars = 4;

template <typename V>

class CharMap<wchar\_t, V, CharMapScheme\_Linear> : private Chars<wchar\_t>

{

template <typename C>

friend class TextbookBoyerMooreWithLinearMap;

private:

V defv;

uint map[MaxCharMapLinearChars];

V lastOcc[MaxCharMapLinearChars];

public:

CharMap(V defv) : defv(defv)

{

for (uint i = 0; i < MaxCharMapLinearChars; i++)

{

map[i] = 0;

lastOcc[i] = defv;

}

}

inline void Set(uint numLinearChars, Char const \* map, V const \* lastOcc)

{

Assert(numLinearChars <= MaxCharMapLinearChars);

for (uint i = 0; i < numLinearChars; i++)

{

this->map[i] = CTU(map[i]);

this->lastOcc[i] = lastOcc[i];

}

}

uint GetChar(uint index) const

{

Assert(index < MaxCharMapLinearChars);

\_\_analysis\_assume(index < MaxCharMapLinearChars);

return map[index];

}

V GetLastOcc(uint index) const

{

Assert(index < MaxCharMapLinearChars);

\_\_analysis\_assume(index < MaxCharMapLinearChars);

return lastOcc[index];

}

inline V Get(uint inputChar) const

{

if (map[0] == inputChar)

return lastOcc[0];

if (map[1] == inputChar)

return lastOcc[1];

if (map[2] == inputChar)

return lastOcc[2];

if (map[3] == inputChar)

return lastOcc[3];

return defv;

}

inline V Get(Char k) const

{

return Get(CTU(k));

}

};

template <typename V, CharMapScheme scheme>

class CharMap<wchar\_t, V, scheme> : private Chars<wchar\_t>

{

private:

static const int directBits = Chars<char>::CharWidth;

static const int directSize = Chars<char>::NumChars;

static const int bitsPerLevel = 4;

static const int branchingPerLevel = 1 << bitsPerLevel;

static const uint mask = branchingPerLevel - 1;

static const int levels = CharWidth / bitsPerLevel;

inline static uint innerIdx(int level, uint v)

{

return (v >> (level \* bitsPerLevel)) & mask;

}

inline static uint leafIdx(uint v)

{

return v & mask;

}

struct Node

{

virtual void FreeSelf(ArenaAllocator\* allocator) = 0;

virtual void Set(ArenaAllocator\* allocator, V defv, int level, uint k, V v) = 0;

virtual V Get(V defv, int level, uint k) const = 0;

static inline Node\* For(ArenaAllocator\* allocator, int level, V defv)

{

if (level == 0)

return Anew(allocator, Leaf, defv);

else

return Anew(allocator, Inner);

}

};

struct Inner : Node

{

Node\* children[branchingPerLevel];

Inner()

{

for (int i = 0; i < branchingPerLevel; i++)

children[i] = 0;

}

void FreeSelf(ArenaAllocator\* allocator) override

{

for (int i = 0; i < branchingPerLevel; i++)

{

if (children[i] != 0)

{

children[i]->FreeSelf(allocator);

#if DBG

children[i] = 0;

#endif

}

}

Adelete(allocator, this);

}

void Set(ArenaAllocator\* allocator, V defv, int level, uint k, V v) override

{

Assert(level > 0);

uint i = innerIdx(level--, k);

if (children[i] == 0)

{

if (v == defv)

return;

children[i] = For(allocator, level, defv);

}

children[i]->Set(allocator, defv, level, k, v);

}

V Get(V defv, int level, uint k) const override

{

Assert(level > 0);

uint i = innerIdx(level--, k);

if (children[i] == 0)

return defv;

else

return children[i]->Get(defv, level, k);

}

};

struct Leaf : Node

{

V values[branchingPerLevel];

Leaf(V defv)

{

for (int i = 0; i < branchingPerLevel; i++)

values[i] = defv;

}

void FreeSelf(ArenaAllocator\* allocator) override

{

Adelete(allocator, this);

}

void Set(ArenaAllocator\* allocator, V defv, int level, uint k, V v) override

{

Assert(level == 0);

values[leafIdx(k)] = v;

}

V Get(V defv, int level, uint k) const override

{

Assert(level == 0);

return values[leafIdx(k)];

}

};

BVStatic<directSize> isInMap;

V defv;

V directMap[directSize];

Node\* root;

public:

CharMap(V defv)

: defv(defv)

, root(0)

{

for (int i = 0; i < directSize; i++)

directMap[i] = defv;

}

void FreeBody(ArenaAllocator\* allocator)

{

if (root != 0)

{

root->FreeSelf(allocator);

#if DBG

root = 0;

#endif

}

}

void Set(ArenaAllocator\* allocator, Char kc, V v)

{

uint k = CTU(kc);

if (k < directSize)

{

isInMap.Set(k);

directMap[k] = v;

}

else

{

if (root == 0)

{

if (v == defv)

return;

root = Anew(allocator, Inner);

}

root->Set(allocator, defv, levels - 1, k, v);

}

}

bool GetNonDirect(uint k, V& lastOcc) const

{

Assert(k >= directSize);

if (root == nullptr)

{

return false;

}

Node\* curr = root;

for (int level = levels - 1; level > 0; level--)

{

Inner\* inner = (Inner\*)curr;

uint i = innerIdx(level, k);

if (inner->children[i] == 0)

return false;

else

curr = inner->children[i];

}

Leaf\* leaf = (Leaf\*)curr;

lastOcc = leaf->values[leafIdx(k)];

return true;

}

uint GetDirectMapSize() const { return directSize; }

BOOL IsInDirectMap(uint c) const { Assert(c < directSize); return isInMap.Test(c); }

V GetDirectMap(uint c) const

{

Assert(c < directSize);

\_\_analysis\_assume(c < directSize);

return directMap[c];

}

\_\_inline V Get(Char kc) const

{

if (CTU(kc) < GetDirectMapSize())

{

if (!IsInDirectMap(CTU(kc)))

{

return defv;

}

return GetDirectMap(CTU(kc));

}

else

{

V lastOcc;

if (!GetNonDirect(CTU(kc), lastOcc))

{

return defv;

}

return lastOcc;

}

}

};

}

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

// 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 UnifiedRegex

{

template <typename C>

struct Chars

{

typedef C Char;

};

template <>

struct Chars<uint8>

{

typedef uint8 Char;

typedef uint8 UChar;

static const int CharWidth = sizeof(char) \* 8;

static const int NumChars = 1 << CharWidth;

static const uint MaxUChar = (uint8)-1;

static const uint MaxUCharAscii = (1 << 7) - 1;

static const Char MinChar = (Char)0;

static const Char MaxChar = (Char)MaxUChar;

// Char to unsigned int

static inline uint CTU(Char c)

{

return (uint)c;

}

// Unsigned int to Char

static inline Char UTC(uint u) {

Assert(u <= MaxUChar);

return (Char)u;

}

// int to Char

static inline Char ITC(int i) {

Assert(i >= 0 && i <= MaxUChar);

return (Char)i;

}

// Char to wchar\_t

static inline wchar\_t CTW(Char c)

{

return (wchar\_t)c;

}

// Offset, same buffer

static inline CharCount OSB(const Char\* ph, const Char\* pl)

{

Assert(ph >= pl && ph - pl <= MaxCharCount);

return (CharCount)(ph - pl);

}

static inline Char Shift(Char c, int n)

{

return UTC(CTU(c) + n);

}

};

template <>

struct Chars<char>

{

typedef char Char;

typedef uint8 UChar;

static const int CharWidth = sizeof(char) \* 8;

static const int NumChars = 1 << CharWidth;

static const uint MaxUChar = (uint8)-1;

static const uint MaxUCharAscii = (1 << 7) - 1;

static const Char MinChar = (Char)0;

static const Char MaxChar = (Char)MaxUChar;

// Char to unsigned int

static inline uint CTU(Char c)

{

return (uint8)c;

}

// Unsigned int to Char

static inline Char UTC(uint u) {

Assert(u <= MaxUChar);

return (Char)u;

}

// int to Char

static inline Char ITC(int i) {

Assert(i >= 0 && i <= MaxUChar);

return (Char)(uint8)i;

}

// Char to wchar\_t

static inline wchar\_t CTW(Char c)

{

return (wchar\_t)(uint8)c;

}

// Offset, same buffer

static inline CharCount OSB(const Char\* ph, const Char\* pl)

{

Assert(ph >= pl && ph - pl <= MaxCharCount);

return (CharCount)(ph - pl);

}

static inline Char Shift(Char c, int n)

{

return UTC(CTU(c) + n);

}

};

template <>

struct Chars<wchar\_t>

{

typedef wchar\_t Char;

typedef uint16 UChar;

static const int CharWidth = sizeof(wchar\_t) \* 8;

static const int NumChars = 1 << CharWidth;

static const uint MaxUChar = (uint16)-1;

static const uint MaxUCharAscii = (1 << 7) - 1;

static const Char MinChar = (Char)0;

static const Char MaxChar = (Char)MaxUChar;

// Char to unsigned int

static inline uint CTU(Char c)

{

return (uint16)c;

}

// Unsigned int to Char

static inline Char UTC(uint u)

{

Assert(u <= MaxUChar);

return (Char)u;

}

// int to Char

static inline Char ITC(int i) {

Assert(i >= 0 && i <= MaxUChar);

return (Char)(uint16)i;

}

// Char to wchar\_t

static inline wchar\_t CTW(Char c)

{

return c;

}

// Offset, same buffer

static inline CharCount OSB(const Char\* ph, const Char\* pl)

{

Assert(ph >= pl && ph - pl <= MaxCharCount);

return (CharCount)(ph - pl);

}

static inline Char Shift(Char c, int n)

{

return UTC(CTU(c) + n);

}

};

template <>

struct Chars<codepoint\_t>

{

typedef codepoint\_t Char;

typedef codepoint\_t UChar;

static const int CharWidth = sizeof(codepoint\_t) \* 8;

static const int NumChars = 0x110000;

static const uint MaxUChar = (NumChars) - 1;

static const uint MaxUCharAscii = (1 << 7) - 1;

static const Char MinChar = (Char)0;

static const Char MaxChar = (Char)MaxUChar;

// Char to unsigned int

static inline uint CTU(Char c)

{

Assert(c <= MaxChar);

return (codepoint\_t)c;

}

// Unsigned int to Char

static inline Char UTC(uint u)

{

Assert(u <= MaxUChar);

return (Char)u;

}

// int to Char

static inline Char ITC(int i) {

Assert(i >= 0 && i <= MaxUChar);

return (Char)(codepoint\_t)i;

}

// Char to wchar\_t

static inline wchar\_t CTW(Char c)

{

Assert(c < Chars<wchar\_t>::MaxUChar);

return (wchar\_t)c;

}

// Offset, same buffer

static inline CharCount OSB(const Char\* ph, const Char\* pl)

{

Assert(ph >= pl && ph - pl <= MaxCharCount);

return (CharCount)(ph - pl);

}

static inline Char Shift(Char c, int n)

{

return UTC(CTU(c) + n);

}

};

}

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

// 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"

namespace UnifiedRegex

{

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

// CharBitVec

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

inline uint32 popcnt(uint32 x)

{

// sum set bits in every bit pair

x -= (x >> 1) & 0x55555555u;

// sum pairs into quads

x = (x & 0x33333333u) + ((x >> 2) & 0x33333333u);

// sum quads into octets

x = (x + (x >> 4)) & 0x0f0f0f0fu;

// sum octets into topmost octet

x \*= 0x01010101u;

return x >> 24;

}

uint CharBitvec::Count() const

{

uint n = 0;

for (int w = 0; w < vecSize; w++)

{

n += popcnt(vec[w]);

}

return n;

}

int CharBitvec::NextSet(int k) const

{

if (k < 0 || k >= Size)

return -1;

uint w = k / wordSize;

uint o = k % wordSize;

uint32 v = vec[w] >> o;

do

{

if (v == 0)

{

k += wordSize - o;

break;

}

else if ((v & 0x1) != 0)

return k;

else

{

v >>= 1;

o++;

k++;

}

}

while (o < wordSize);

w++;

while (w < vecSize)

{

o = 0;

v = vec[w];

do

{

if (v == 0)

{

k += wordSize - o;

break;

}

else if ((v & 0x1) != 0)

return k;

else

{

v >>= 1;

o++;

k++;

}

}

while (o < wordSize);

w++;

}

return -1;

}

int CharBitvec::NextClear(int k) const

{

if (k < 0 || k >= Size)

return -1;

uint w = k / wordSize;

uint o = k % wordSize;

uint32 v = vec[w] >> o;

do

{

if (v == ones)

{

k += wordSize - o;

break;

}

else if ((v & 0x1) == 0)

return k;

else

{

v >>= 1;

o++;

k++;

}

}

while (o < wordSize);

w++;

while (w < vecSize)

{

o = 0;

v = vec[w];

do

{

if (v == ones)

{

k += wordSize - o;

break;

}

else if ((v & 0x1) == 0)

return k;

else

{

v >>= 1;

o++;

k++;

}

}

while (o < wordSize);

w++;

}

return -1;

}

template <typename C>

void CharBitvec::ToComplement(ArenaAllocator\* allocator, uint base, CharSet<C>& result) const

{

int hi = -1;

while (true)

{

// Find the next range of clear bits in vector

int li = NextClear(hi + 1);

if (li < 0)

return;

hi = NextSet(li + 1);

if (hi < 0)

hi = Size - 1;

else

{

Assert(hi > 0);

hi--;

}

// Add range as characters

result.SetRange(allocator, Chars<C>::ITC(base + li), Chars<C>::ITC(base + hi));

}

}

template <typename C>

void CharBitvec::ToEquivClass(ArenaAllocator\* allocator, uint base, uint& tblidx, CharSet<C>& result, codepoint\_t baseOffset) const

{

int hi = -1;

while (true)

{

// Find the next range of set bits in vector

int li = NextSet(hi + 1);

if (li < 0)

return;

hi = NextClear(li + 1);

if (hi < 0)

hi = Size - 1;

else

{

Assert(hi > 0);

hi--;

}

// Convert to character codes

uint l = base + li + baseOffset;

uint h = base + hi + baseOffset;

do

{

uint acth;

C equivl[CaseInsensitive::EquivClassSize];

CaseInsensitive::RangeToEquivClass(tblidx, l, h, acth, equivl);

uint n = acth - l;

for (int i = 0; i < CaseInsensitive::EquivClassSize; i++)

{

result.SetRange(allocator, equivl[i], Chars<C>::Shift(equivl[i], n));

}

// Go around again for rest of this range

l = acth + 1;

}

while (l <= h);

}

}

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

// CharSetNode

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

inline CharSetNode\* CharSetNode::For(ArenaAllocator\* allocator, int level)

{

if (level == 0)

return Anew(allocator, CharSetLeaf);

else

return Anew(allocator, CharSetInner);

}

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

// CharSetFull

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

CharSetFull CharSetFull::Instance;

CharSetFull\* const CharSetFull::TheFullNode = &CharSetFull::Instance;

CharSetFull::CharSetFull() {}

void CharSetFull::FreeSelf(ArenaAllocator\* allocator)

{

Assert(this == TheFullNode);

// Never allocated

}

CharSetNode\* CharSetFull::Clone(ArenaAllocator\* allocator) const

{

// Always shared

return (CharSetNode\*)this;

}

CharSetNode\* CharSetFull::Set(ArenaAllocator\* allocator, uint level, uint l, uint h)

{

return this;

}

CharSetNode\* CharSetFull::ClearRange(ArenaAllocator\* allocator, uint level, uint l, uint h)

{

AssertMsg(h <= lim(level), "The range for clearing provided is invalid for this level.");

AssertMsg(l <= h, "Can't clear where lover is bigger than the higher.");

if (l == 0 && h == lim(level))

{

return nullptr;

}

CharSetNode\* toReturn = For(allocator, level);

if (l > 0)

{

AssertVerify(toReturn->Set(allocator, level, 0, l - 1) == toReturn);

}

if (h < lim(level))

{

AssertVerify(toReturn->Set(allocator, level, h + 1, lim(level)) == toReturn);

}

return toReturn;

}

CharSetNode\* CharSetFull::UnionInPlace(ArenaAllocator\* allocator, uint level, const CharSetNode\* other)

{

return this;

}

bool CharSetFull::Get(uint level, uint k) const

{

return true;

}

void CharSetFull::ToComplement(ArenaAllocator\* allocator, uint level, uint base, CharSet<Char>& result) const

{

// Empty, so add nothing

}

void CharSetFull::ToEquivClassW(ArenaAllocator\* allocator, uint level, uint base, uint& tblidx, CharSet<wchar\_t>& result) const

{

this->ToEquivClass<wchar\_t>(allocator, level, base, tblidx, result);

}

void CharSetFull::ToEquivClassCP(ArenaAllocator\* allocator, uint level, uint base, uint& tblidx, CharSet<codepoint\_t>& result, codepoint\_t baseOffset) const

{

this->ToEquivClass<codepoint\_t>(allocator, level, base, tblidx, result, baseOffset);

}

template <typename C>

void CharSetFull::ToEquivClass(ArenaAllocator\* allocator, uint level, uint base, uint& tblidx, CharSet<C>& result, codepoint\_t baseOffset) const

{

uint l = base + (CharSetNode::levels - 1 == level ? 0xff : 0) + baseOffset;

uint h = base + lim(level) + baseOffset;

do

{

uint acth;

C equivl[CaseInsensitive::EquivClassSize];

CaseInsensitive::RangeToEquivClass(tblidx, l, h, acth, equivl);

uint n = acth - l;

for (int i = 0; i < CaseInsensitive::EquivClassSize; i++)

{

result.SetRange(allocator, equivl[i], Chars<C>::Shift(equivl[i], n));

}

// Go around again for rest of this range

l = acth + 1;

}

while (l <= h);

}

bool CharSetFull::IsSubsetOf(uint level, const CharSetNode\* other) const

{

Assert(other != nullptr);

return other == TheFullNode;

}

bool CharSetFull::IsEqualTo(uint level, const CharSetNode\* other) const

{

Assert(other != nullptr);

return other == TheFullNode;

}

uint CharSetFull::Count(uint level) const

{

return lim(level) + 1;

}

\_Success\_(return)

bool CharSetFull::GetNextRange(uint level, Char searchCharStart, \_Out\_ Char \*outLowerChar, \_Out\_ Char \*outHigherChar) const

{

Assert(searchCharStart < this->Count(level));

\*outLowerChar = searchCharStart;

\*outHigherChar = (Char)this->Count(level) - 1;

return true;

}

#if DBG

bool CharSetFull::IsLeaf() const

{

return false;

}

#endif

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

// CharSetInner

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

CharSetInner::CharSetInner()

{

for (uint i = 0; i < branchingPerInnerLevel; i++)

children[i] = 0;

}

void CharSetInner::FreeSelf(ArenaAllocator\* allocator)

{

for (uint i = 0; i < branchingPerInnerLevel; i++)

{

if (children[i] != 0)

{

children[i]->FreeSelf(allocator);

#if DBG

children[i] = 0;

#endif

}

}

Adelete(allocator, this);

}

CharSetNode\* CharSetInner::Clone(ArenaAllocator\* allocator) const

{

CharSetInner\* res = Anew(allocator, CharSetInner);

for (uint i = 0; i < branchingPerInnerLevel; i++)

{

if (children[i] != 0)

res->children[i] = children[i]->Clone(allocator);

}

return res;

}

CharSetNode\* CharSetInner::ClearRange(ArenaAllocator\* allocator, uint level, uint l, uint h)

{

Assert(level > 0);

AssertMsg(h <= lim(level), "The range for clearing provided is invalid for this level.");

AssertMsg(l <= h, "Can't clear where lover is bigger than the higher.");

if (l == 0 && h == lim(level))

{

return nullptr;

}

uint lowerIndex = innerIdx(level, l);

uint higherIndex = innerIdx(level--, h);

l = l & lim(level);

h = h & lim(level);

if (lowerIndex == higherIndex)

{

if (children[lowerIndex] != nullptr)

{

children[lowerIndex] = children[lowerIndex]->ClearRange(allocator, level, l, h);

}

}

else

{

if (children[lowerIndex] != nullptr)

{

children[lowerIndex] = children[lowerIndex]->ClearRange(allocator, level, l, lim(level));

}

for (uint i = lowerIndex + 1; i < higherIndex; i++)

{

if (children[i] != nullptr)

{

children[i]->FreeSelf(allocator);

}

children[i] = nullptr;

}

if (children[higherIndex] != nullptr)

{

children[higherIndex] = children[higherIndex]->ClearRange(allocator, level, 0, h);

}

}

for (int i = 0; i < branchingPerInnerLevel; i++)

{

if (children[i] != nullptr)

{

return this;

}

}

return nullptr;

}

CharSetNode\* CharSetInner::Set(ArenaAllocator\* allocator, uint level, uint l, uint h)

{

Assert(level > 0);

uint li = innerIdx(level, l);

uint hi = innerIdx(level--, h);

bool couldBeFull = true;

if (li == hi)

{

if (children[li] == nullptr)

{

if (remain(level, l) == 0 && remain(level, h + 1) == 0)

children[li] = CharSetFull::TheFullNode;

else

{

children[li] = For(allocator, level);

children[li] = children[li]->Set(allocator, level, l, h);

couldBeFull = false;

}

}

else

children[li] = children[li]->Set(allocator, level, l, h);

}

else

{

if (children[li] == nullptr)

{

if (remain(level, l) == 0)

children[li] = CharSetFull::TheFullNode;

else

{

children[li] = For(allocator, level);

children[li] = children[li]->Set(allocator, level, l, lim(level));

couldBeFull = false;

}

}

else

children[li] = children[li]->Set(allocator, level, l, lim(level));

for (uint i = li + 1; i < hi; i++)

{

if (children[i] != nullptr)

children[i]->FreeSelf(allocator);

children[i] = CharSetFull::TheFullNode;

}

if (children[hi] == nullptr)

{

if (remain(level, h + 1) == 0)

children[hi] = CharSetFull::TheFullNode;

else

{

children[hi] = For(allocator, level);

children[hi] = children[hi]->Set(allocator, level, 0, h);

couldBeFull = false;

}

}

else

children[hi] = children[hi]->Set(allocator, level, 0, h);

}

if (couldBeFull)

{

for (uint i = 0; i < branchingPerInnerLevel; i++)

{

if (children[i] != CharSetFull::TheFullNode)

return this;

}

FreeSelf(allocator);

return CharSetFull::TheFullNode;

}

else

return this;

}

CharSetNode\* CharSetInner::UnionInPlace(ArenaAllocator\* allocator, uint level, const CharSetNode\* other)

{

Assert(level > 0);

Assert(other != nullptr && other != CharSetFull::TheFullNode && !other->IsLeaf());

CharSetInner\* otherInner = (CharSetInner\*)other;

level--;

bool isFull = true;

for (uint i = 0; i < branchingPerInnerLevel; i++)

{

if (otherInner->children[i] != nullptr)

{

if (otherInner->children[i] == CharSetFull::TheFullNode)

{

if (children[i] != nullptr)

children[i]->FreeSelf(allocator);

children[i] = CharSetFull::TheFullNode;

}

else

{

if (children[i] == nullptr)

children[i] = For(allocator, level);

children[i] = children[i]->UnionInPlace(allocator, level, otherInner->children[i]);

if (children[i] != CharSetFull::TheFullNode)

isFull = false;

}

}

else if (children[i] != CharSetFull::TheFullNode)

isFull = false;

}

if (isFull)

{

FreeSelf(allocator);

return CharSetFull::TheFullNode;

}

else

return this;

}

bool CharSetInner::Get(uint level, uint k) const

{

Assert(level > 0);

uint i = innerIdx(level--, k);

if (children[i] == nullptr)

return false;

else

return children[i]->Get(level, k);

}

void CharSetInner::ToComplement(ArenaAllocator\* allocator, uint level, uint base, CharSet<Char>& result) const

{

Assert(level > 0);

level--;

uint delta = lim(level) + 1;

for (uint i = 0; i < branchingPerInnerLevel; i++)

{

if (children[i] == nullptr)

// Caution: Part of the range for this child may overlap with direct vector

result.SetRange(allocator, UTC(max(base, directSize)), UTC(base + delta - 1));

else

children[i]->ToComplement(allocator, level, base, result);

base += delta;

}

}

void CharSetInner::ToEquivClassW(ArenaAllocator\* allocator, uint level, uint base, uint& tblidx, CharSet<wchar\_t>& result) const

{

Assert(level > 0);

level--;

uint delta = lim(level) + 1;

for (uint i = 0; i < branchingPerInnerLevel; i++)

{

if (children[i] != nullptr)

{

children[i]->ToEquivClassW(allocator, level, base, tblidx, result);

}

base += delta;

}

}

void CharSetInner::ToEquivClassCP(ArenaAllocator\* allocator, uint level, uint base, uint& tblidx, CharSet<codepoint\_t>& result, codepoint\_t baseOffset) const

{

Assert(level > 0);

level--;

uint delta = lim(level) + 1;

for (uint i = 0; i < branchingPerInnerLevel; i++)

{

if (children[i] != nullptr)

{

children[i]->ToEquivClassCP(allocator, level, base, tblidx, result, baseOffset);

}

base += delta;

}

}

bool CharSetInner::IsSubsetOf(uint level, const CharSetNode\* other) const

{

Assert(level > 0);

Assert(other != nullptr && !other->IsLeaf());

if (other == CharSetFull::TheFullNode)

return true;

level--;

const CharSetInner\* otherInner = (CharSetInner\*)other;

for (uint i = 0; i < branchingPerInnerLevel; i++)

{

if (children[i] != nullptr)

{

if (otherInner->children[i] == nullptr)

return false;

if (children[i]->IsSubsetOf(level, otherInner->children[i]))

return false;

}

}

return true;

}

bool CharSetInner::IsEqualTo(uint level, const CharSetNode\* other) const

{

Assert(level > 0);

Assert(other != nullptr && !other->IsLeaf());

if (other == CharSetFull::TheFullNode)

return false;

level--;

const CharSetInner\* otherInner = (CharSetInner\*)other;

for (uint i = 0; i < branchingPerInnerLevel; i++)

{

if (children[i] != 0)

{

if (otherInner->children[i] == nullptr)

return false;

if (children[i]->IsSubsetOf(level, otherInner->children[i]))

return false;

}

}

return true;

}

uint CharSetInner::Count(uint level) const

{

uint n = 0;

Assert(level > 0);

level--;

for (uint i = 0; i < branchingPerInnerLevel; i++)

{

if (children[i] != nullptr)

n += children[i]->Count(level);

}

return n;

}

\_Success\_(return)

bool CharSetInner::GetNextRange(uint level, Char searchCharStart, \_Out\_ Char \*outLowerChar, \_Out\_ Char \*outHigherChar) const

{

Assert(searchCharStart < this->lim(level) + 1);

uint innerIndex = innerIdx(level--, searchCharStart);

Char currentLowChar = 0, currentHighChar = 0;

for (; innerIndex < branchingPerInnerLevel; innerIndex++)

{

if (children[innerIndex] != nullptr && children[innerIndex]->GetNextRange(level, (Char)remain(level, searchCharStart), &currentLowChar, &currentHighChar))

{

break;

}

if (innerIndex < branchingPerInnerLevel - 1)

{

searchCharStart = (Char)indexToValue(level + 1, innerIndex + 1, 0);

}

}

if (innerIndex == branchingPerInnerLevel)

{

return false;

}

currentLowChar = (Char)indexToValue(level + 1, innerIndex, currentLowChar);

currentHighChar = (Char)indexToValue(level + 1, innerIndex, currentHighChar);

innerIndex += 1;

for (; remain(level, currentHighChar) == lim(level) && innerIndex < branchingPerInnerLevel; innerIndex++)

{

Char tempLower, tempHigher;

if (children[innerIndex] == nullptr || !children[innerIndex]->GetNextRange(level, 0x0, &tempLower, &tempHigher) || remain(level, tempLower) != 0)

{

break;

}

currentHighChar = (Char)indexToValue(level + 1, innerIndex, tempHigher);

}

\*outLowerChar = currentLowChar;

\*outHigherChar = currentHighChar;

return true;

}

#if DBG

bool CharSetInner::IsLeaf() const

{

return false;

}

#endif

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

// CharSetLeaf

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

CharSetLeaf::CharSetLeaf()

{

vec.Clear();

}

void CharSetLeaf::FreeSelf(ArenaAllocator\* allocator)

{

Adelete(allocator, this);

}

CharSetNode\* CharSetLeaf::Clone(ArenaAllocator\* allocator) const

{

return Anew(allocator, CharSetLeaf, \*this);

}

CharSetNode\* CharSetLeaf::Set(ArenaAllocator\* allocator, uint level, uint l, uint h)

{

Assert(level == 0);

vec.SetRange(leafIdx(l), leafIdx(h));

if (vec.IsFull())

{

FreeSelf(allocator);

return CharSetFull::TheFullNode;

}

else

return this;

}

CharSetNode\* CharSetLeaf::ClearRange(ArenaAllocator\* allocator, uint level, uint l, uint h)

{

Assert(level == 0);

AssertMsg(h <= lim(level), "The range for clearing provided is invalid for this level.");

AssertMsg(l <= h, "Can't clear where lover is bigger than the higher.");

if (l == 0 && h == lim(level))

{

return nullptr;

}

vec.ClearRange(leafIdx(l), leafIdx(h));

if (vec.IsEmpty())

{

FreeSelf(allocator);

return nullptr;

}

return this;

}

CharSetNode\* CharSetLeaf::UnionInPlace(ArenaAllocator\* allocator, uint level, const CharSetNode\* other)

{

Assert(level == 0);

Assert(other != nullptr && other->IsLeaf());

CharSetLeaf\* otherLeaf = (CharSetLeaf\*)other;

if (vec.UnionInPlaceFullCheck(otherLeaf->vec))

{

FreeSelf(allocator);

return CharSetFull::TheFullNode;

}

else

return this;

}

bool CharSetLeaf::Get(uint level, uint k) const

{

Assert(level == 0);

return vec.Get(leafIdx(k));

}

void CharSetLeaf::ToComplement(ArenaAllocator\* allocator, uint level, uint base, CharSet<Char>& result) const

{

Assert(level == 0);

vec.ToComplement<wchar\_t>(allocator, base, result);

}

void CharSetLeaf::ToEquivClassW(ArenaAllocator\* allocator, uint level, uint base, uint& tblidx, CharSet<wchar\_t>& result) const

{

this->ToEquivClass<wchar\_t>(allocator, level, base, tblidx, result);

}

void CharSetLeaf::ToEquivClassCP(ArenaAllocator\* allocator, uint level, uint base, uint& tblidx, CharSet<codepoint\_t>& result, codepoint\_t baseOffset) const

{

this->ToEquivClass<codepoint\_t>(allocator, level, base, tblidx, result, baseOffset);

}

template <typename C>

void CharSetLeaf::ToEquivClass(ArenaAllocator\* allocator, uint level, uint base, uint& tblidx, CharSet<C>& result, codepoint\_t baseOffset) const

{

Assert(level == 0);

vec.ToEquivClass<C>(allocator, base, tblidx, result, baseOffset);

}

bool CharSetLeaf::IsSubsetOf(uint level, const CharSetNode\* other) const

{

Assert(level == 0);

Assert(other != nullptr);

if (other == CharSetFull::TheFullNode)

return true;

Assert(other->IsLeaf());

CharSetLeaf\* otherLeaf = (CharSetLeaf\*)other;

return vec.IsSubsetOf(otherLeaf->vec);

}

bool CharSetLeaf::IsEqualTo(uint level, const CharSetNode\* other) const

{

Assert(level == 0);

Assert(other != nullptr);

if (other == CharSetFull::TheFullNode)

return false;

Assert(other->IsLeaf());

CharSetLeaf\* otherLeaf = (CharSetLeaf\*)other;

return vec.IsSubsetOf(otherLeaf->vec);

}

uint CharSetLeaf::Count(uint level) const

{

Assert(level == 0);

return vec.Count();

}

\_Success\_(return)

bool CharSetLeaf::GetNextRange(uint level, Char searchCharStart, \_Out\_ Char \*outLowerChar, \_Out\_ Char \*outHigherChar) const

{

Assert(searchCharStart < lim(level) + 1);

int nextSet = vec.NextSet(searchCharStart);

if (nextSet == -1)

{

return false;

}

\*outLowerChar = (wchar\_t)nextSet;

int nextClear = vec.NextClear(nextSet);

\*outHigherChar = UTC(nextClear == -1 ? lim(level) : nextClear - 1);

return true;

}

#if DBG

bool CharSetLeaf::IsLeaf() const

{

return true;

}

#endif

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

// CharSet<wchar\_t>

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

void CharSet<wchar\_t>::SwitchRepresentations(ArenaAllocator\* allocator)

{

Assert(IsCompact());

uint existCount = this->GetCompactLength();

\_\_assume(existCount <= MaxCompact);

if (existCount <= MaxCompact)

{

Char existCs[MaxCompact];

for (uint i = 0; i < existCount; i++)

{

existCs[i] = GetCompactChar(i);

}

rep.full.root = nullptr;

rep.full.direct.Clear();

for (uint i = 0; i < existCount; i++)

Set(allocator, existCs[i]);

}

}

void CharSet<wchar\_t>::Sort()

{

Assert(IsCompact());

\_\_assume(this->GetCompactLength() <= MaxCompact);

for (uint i = 1; i < this->GetCompactLength(); i++)

{

uint curr = GetCompactCharU(i);

for (uint j = 0; j < i; j++)

{

if (GetCompactCharU(j) > curr)

{

for (int k = i; k > (int)j; k--)

{

this->ReplaceCompactCharU(k, this->GetCompactCharU(k - 1));

}

this->ReplaceCompactCharU(j, curr);

break;

}

}

}

}

CharSet<wchar\_t>::CharSet()

{

Assert(sizeof(Node\*) == sizeof(size\_t));

Assert(sizeof(CompactRep) == sizeof(FullRep));

rep.compact.countPlusOne = 1;

for (int i = 0; i < MaxCompact; i++)

rep.compact.cs[i] = emptySlot;

}

void CharSet<wchar\_t>::FreeBody(ArenaAllocator\* allocator)

{

if (!IsCompact() && rep.full.root != nullptr)

{

rep.full.root->FreeSelf(allocator);

#if DBG

rep.full.root = nullptr;

#endif

}

}

void CharSet<wchar\_t>::Clear(ArenaAllocator\* allocator)

{

if (!IsCompact() && rep.full.root != nullptr)

rep.full.root->FreeSelf(allocator);

rep.compact.countPlusOne = 1;

for (int i = 0; i < MaxCompact; i++)

rep.compact.cs[i] = emptySlot;

}

void CharSet<wchar\_t>::CloneFrom(ArenaAllocator\* allocator, const CharSet<Char>& other)

{

Clear(allocator);

Assert(IsCompact());

if (other.IsCompact())

{

this->SetCompactLength(other.GetCompactLength());

for (uint i = 0; i < other.GetCompactLength(); i++)

{

this->ReplaceCompactCharU(i, other.GetCompactCharU(i));

}

}

else

{

rep.full.root = other.rep.full.root == nullptr ? nullptr : other.rep.full.root->Clone(allocator);

rep.full.direct.CloneFrom(other.rep.full.direct);

}

}

void CharSet<wchar\_t>::CloneNonSurrogateCodeUnitsTo(ArenaAllocator\* allocator, CharSet<Char>& other)

{

if (this->IsCompact())

{

for (uint i = 0; i < this->GetCompactLength(); i++)

{

Char c = this->GetCompactChar(i);

uint uChar = CTU(c);

if (uChar < 0xD800 || uChar > 0xDFFF)

{

other.Set(allocator, c);

}

}

}

else

{

other.rep.full.direct.CloneFrom(rep.full.direct);

if (rep.full.root == nullptr)

{

other.rep.full.root = nullptr;

}

else

{

other.rep.full.root = rep.full.root->Clone(allocator);

other.rep.full.root->ClearRange(allocator, CharSetNode::levels - 1, 0xD800, 0XDFFF);

}

}

}

void CharSet<wchar\_t>::CloneSurrogateCodeUnitsTo(ArenaAllocator\* allocator, CharSet<Char>& other)

{

if (this->IsCompact())

{

for (uint i = 0; i < this->GetCompactLength(); i++)

{

Char c = this->GetCompactChar(i);

uint uChar = CTU(c);

if (0xD800 <= uChar && uChar <= 0xDFFF)

{

other.Set(allocator, c);

}

}

}

else

{

other.rep.full.direct.CloneFrom(rep.full.direct);

if (rep.full.root == nullptr)

{

other.rep.full.root = nullptr;

}

else

{

other.rep.full.root = rep.full.root->Clone(allocator);

other.rep.full.root->ClearRange(allocator, CharSetNode::levels - 1, 0, 0xD7FF);

}

}

}

void CharSet<wchar\_t>::SubtractRange(ArenaAllocator\* allocator, Char lowerChar, Char higherChar)

{

uint lowerValue = CTU(lowerChar);

uint higherValue = CTU(higherChar);

if (higherValue < lowerValue)

return;

if (IsCompact())

{

for (uint i = 0; i < this->GetCompactLength(); )

{

uint value = this->GetCompactCharU(i);

if (value >= lowerValue && value <= higherValue)

{

this->RemoveCompactChar(i);

}

else

{

i++;

}

}

}

else if(lowerValue == 0 && higherValue == MaxUChar)

{

this->Clear(allocator);

}

else

{

if (lowerValue < CharSetNode::directSize)

{

uint maxDirectValue = min(higherValue, CharSetNode::directSize - 1);

rep.full.direct.ClearRange(lowerValue, maxDirectValue);

}

if (rep.full.root != nullptr)

{

rep.full.root = rep.full.root->ClearRange(allocator, CharSetNode::levels - 1, lowerValue, higherValue);

}

}

}

void CharSet<wchar\_t>::SetRange(ArenaAllocator\* allocator, Char lc, Char hc)

{

uint l = CTU(lc);

uint h = CTU(hc);

if (h < l)

return;

if (IsCompact())

{

if (h - l < MaxCompact)

{

do

{

uint i;

for (i = 0; i < this->GetCompactLength(); i++)

{

\_\_assume(l <= MaxUChar);

if (l <= MaxUChar && i < MaxCompact)

{

if (this->GetCompactCharU(i) == l)

break;

}

}

if (i == this->GetCompactLength())

{

// Character not already in compact set

if (i < MaxCompact)

{

this->AddCompactCharU(l);

}

else

// Must switch representations

break;

}

l++;

}

while (l <= h);

if (h < l)

// All chars are now in compact set

return;

// else: fall-through to general case for remaining chars

}

// else: no use even trying

SwitchRepresentations(allocator);

}

Assert(!IsCompact());

if (l == 0 && h == MaxUChar)

{

rep.full.direct.SetRange(0, CharSetNode::directSize - 1);

if (rep.full.root != nullptr)

rep.full.root->FreeSelf(allocator);

rep.full.root = CharSetFull::TheFullNode;

}

else

{

if (l < CharSetNode::directSize)

{

if (h < CharSetNode::directSize)

{

rep.full.direct.SetRange(l, h);

return;

}

rep.full.direct.SetRange(l, CharSetNode::directSize - 1);

l = CharSetNode::directSize;

}

if (rep.full.root == nullptr)

rep.full.root = Anew(allocator, CharSetInner);

rep.full.root = rep.full.root->Set(allocator, CharSetNode::levels - 1, l, h);

}

}

void CharSet<wchar\_t>::SetRanges(ArenaAllocator\* allocator, int numSortedPairs, const Char\* sortedPairs)

{

for (int i = 0; i < numSortedPairs \* 2; i += 2)

{

Assert(i == 0 || sortedPairs[i-1] < sortedPairs[i]);

Assert(sortedPairs[i] <= sortedPairs[i+1]);

SetRange(allocator, sortedPairs[i], sortedPairs[i+1]);

}

}

void CharSet<wchar\_t>::SetNotRanges(ArenaAllocator\* allocator, int numSortedPairs, const Char\* sortedPairs)

{

if (numSortedPairs == 0)

SetRange(allocator, MinChar, MaxChar);

else

{

if (sortedPairs[0] != MinChar)

SetRange(allocator, MinChar, sortedPairs[0] - 1);

for (int i = 1; i < numSortedPairs \* 2 - 1; i += 2)

SetRange(allocator, sortedPairs[i] + 1, sortedPairs[i+1] - 1);

if (sortedPairs[numSortedPairs \* 2 - 1] != MaxChar)

SetRange(allocator, sortedPairs[numSortedPairs \* 2 - 1] + 1, MaxChar);

}

}

void CharSet<wchar\_t>::UnionInPlace(ArenaAllocator\* allocator, const CharSet<Char>& other)

{

if (other.IsCompact())

{

for (uint i = 0; i < other.GetCompactLength(); i++)

{

Set(allocator, other.GetCompactChar(i));

}

return;

}

if (IsCompact())

SwitchRepresentations(allocator);

Assert(!IsCompact() && !other.IsCompact());

rep.full.direct.UnionInPlace(other.rep.full.direct);

if (other.rep.full.root != nullptr)

{

if (other.rep.full.root == CharSetFull::TheFullNode)

{

if (rep.full.root != nullptr)

rep.full.root->FreeSelf(allocator);

rep.full.root = CharSetFull::TheFullNode;

}

else

{

if (rep.full.root == nullptr)

rep.full.root = Anew(allocator, CharSetInner);

rep.full.root = rep.full.root->UnionInPlace(allocator, CharSetNode::levels - 1, other.rep.full.root);

}

}

}

\_Success\_(return)

bool CharSet<wchar\_t>::GetNextRange(Char searchCharStart, \_Out\_ Char \*outLowerChar, \_Out\_ Char \*outHigherChar)

{

int count = this->Count();

if (count == 0)

{

return false;

}

else if (count == 1)

{

Char singleton = this->Singleton();

if (singleton < searchCharStart)

{

return false;

}

\*outLowerChar = \*outHigherChar = singleton;

return true;

}

if (IsCompact())

{

this->Sort();

uint i = 0;

size\_t compactLength = this->GetCompactLength();

for (; i < compactLength; i++)

{

Char nextChar = this->GetCompactChar(i);

if (nextChar >= searchCharStart)

{

\*outLowerChar = \*outHigherChar = nextChar;

break;

}

}

if (i == compactLength)

{

return false;

}

i++;

for (; i < compactLength; i++)

{

Char nextChar = this->GetCompactChar(i);

if (nextChar != \*outHigherChar + 1)

{

return true;

}

\*outHigherChar += 1;

}

return true;

}

else

{

bool found = false;

if (CTU(searchCharStart) < CharSetNode::directSize)

{

int nextSet = rep.full.direct.NextSet(searchCharStart);

if (nextSet != -1)

{

found = true;

\*outLowerChar = (wchar\_t)nextSet;

int nextClear = rep.full.direct.NextClear(nextSet);

if (nextClear != -1)

{

\*outHigherChar = UTC(nextClear - 1);

return true;

}

\*outHigherChar = CharSetNode::directSize - 1;

}

}

if (rep.full.root == nullptr)

{

return found;

}

Char tempLowChar = 0, tempHighChar = 0;

if (found)

{

searchCharStart = \*outHigherChar + 1;

}

else

{

searchCharStart = searchCharStart > CharSetNode::directSize ? searchCharStart : CharSetNode::directSize;

}

if (rep.full.root->GetNextRange(CharSetNode::levels - 1, searchCharStart, &tempLowChar, &tempHighChar) && (!found || tempLowChar == \*outHigherChar + 1))

{

if (!found)

{

\*outLowerChar = tempLowChar;

}

\*outHigherChar = tempHighChar;

return true;

}

return found;

}

}

bool CharSet<wchar\_t>::Get\_helper(uint k) const

{

Assert(!IsCompact());

CharSetNode\* curr = rep.full.root;

for (int level = CharSetNode::levels - 1; level > 0; level--)

{

if (curr == CharSetFull::TheFullNode)

return true;

CharSetInner\* inner = (CharSetInner\*)curr;

uint i = CharSetNode::innerIdx(level, k);

if (inner->children[i] == 0)

return false;

else

curr = inner->children[i];

}

if (curr == CharSetFull::TheFullNode)

return true;

CharSetLeaf\* leaf = (CharSetLeaf\*)curr;

return leaf->vec.Get(CharSetNode::leafIdx(k));

}

void CharSet<wchar\_t>::ToComplement(ArenaAllocator\* allocator, CharSet<Char>& result)

{

if (IsCompact())

{

Sort();

if (this->GetCompactLength() > 0)

{

if (this->GetCompactCharU(0) > 0)

result.SetRange(allocator, UTC(0), UTC(this->GetCompactCharU(0) - 1));

for (uint i = 0; i < this->GetCompactLength() - 1; i++)

{

result.SetRange(allocator, UTC(this->GetCompactCharU(i) + 1), UTC(this->GetCompactCharU(i + 1) - 1));

}

if (this->GetCompactCharU(this->GetCompactLength() - 1) < MaxUChar)

{

result.SetRange(allocator, UTC(this->GetCompactCharU(this->GetCompactLength() - 1) + 1), UTC(MaxUChar));

}

}

else if (this->GetCompactLength() == 0)

{

result.SetRange(allocator, UTC(0), UTC(MaxUChar));

}

}

else

{

rep.full.direct.ToComplement<wchar\_t>(allocator, 0, result);

if (rep.full.root == nullptr)

result.SetRange(allocator, UTC(CharSetNode::directSize), MaxChar);

else

rep.full.root->ToComplement(allocator, CharSetNode::levels - 1, 0, result);

}

}

void CharSet<wchar\_t>::ToEquivClass(ArenaAllocator\* allocator, CharSet<Char>& result)

{

uint tblidx = 0;

if (IsCompact())

{

Sort();

for (uint i = 0; i < this->GetCompactLength(); i++)

{

uint acth;

Char equivs[CaseInsensitive::EquivClassSize];

if (CaseInsensitive::RangeToEquivClass(tblidx, this->GetCompactCharU(i), this->GetCompactCharU(i), acth, equivs))

{

for (int j = 0; j < CaseInsensitive::EquivClassSize; j++)

{

result.Set(allocator, equivs[j]);

}

}

else

{

result.Set(allocator, this->GetCompactChar(i));

}

}

}

else

{

rep.full.direct.ToEquivClass<wchar\_t>(allocator, 0, tblidx, result);

if (rep.full.root != nullptr)

{

rep.full.root->ToEquivClassW(allocator, CharSetNode::levels - 1, 0, tblidx, result);

}

}

}

void CharSet<wchar\_t>::ToEquivClassCP(ArenaAllocator\* allocator, CharSet<codepoint\_t>& result, codepoint\_t baseOffset)

{

uint tblidx = 0;

if (IsCompact())

{

Sort();

for (uint i = 0; i < this->GetCompactLength(); i++)

{

uint acth;

codepoint\_t equivs[CaseInsensitive::EquivClassSize];

if (CaseInsensitive::RangeToEquivClass(tblidx, this->GetCompactCharU(i) + baseOffset, this->GetCompactCharU(i) + baseOffset, acth, equivs))

{

for (int j = 0; j < CaseInsensitive::EquivClassSize; j++)

{

result.Set(allocator, equivs[j]);

}

}

else

{

result.Set(allocator, this->GetCompactChar(i) + baseOffset);

}

}

}

else

{

rep.full.direct.ToEquivClass<codepoint\_t>(allocator, 0, tblidx, result, baseOffset);

if (rep.full.root != nullptr)

{

rep.full.root->ToEquivClassCP(allocator, CharSetNode::levels - 1, 0, tblidx, result, baseOffset);

}

}

}

int CharSet<wchar\_t>::GetCompactEntries(uint max, \_\_out\_ecount(max) Char\* entries) const

{

Assert(max <= MaxCompact);

if (!IsCompact())

return -1;

uint count = min(max, (uint)(this->GetCompactLength()));

\_\_analysis\_assume(count <= max);

for (uint i = 0; i < count; i++)

{

// Bug in oacr. it can't figure out count is less than or equal to max

#pragma warning(suppress: 22102)

entries[i] = this->GetCompactChar(i);

}

return static\_cast<int>(rep.compact.countPlusOne - 1);

}

bool CharSet<wchar\_t>::IsSubsetOf(const CharSet<Char>& other) const

{

if (IsCompact())

{

for (uint i = 0; i < this->GetCompactLength(); i++)

{

if (!other.Get(this->GetCompactChar(i)))

return false;

}

return true;

}

else

{

if (other.IsCompact())

return false;

if (!rep.full.direct.IsSubsetOf(other.rep.full.direct))

return false;

if (rep.full.root == nullptr)

return true;

if (other.rep.full.root == nullptr)

return false;

return rep.full.root->IsSubsetOf(CharSetNode::levels - 1, other.rep.full.root);

}

}

bool CharSet<wchar\_t>::IsEqualTo(const CharSet<Char>& other) const

{

if (IsCompact())

{

if (!other.IsCompact())

return false;

if (rep.compact.countPlusOne != other.rep.compact.countPlusOne)

return false;

for (uint i = 0; i < this->GetCompactLength(); i++)

{

if (!other.Get(this->GetCompactChar(i)))

return false;

}

return true;

}

else

{

if (other.IsCompact())

return false;

if (!rep.full.direct.IsEqualTo(other.rep.full.direct))

return false;

if ((rep.full.root == nullptr) != (other.rep.full.root == nullptr))

return false;

if (rep.full.root == nullptr)

return true;

return rep.full.root->IsEqualTo(CharSetNode::levels - 1, other.rep.full.root);

}

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

// CAUTION: This method is very slow.

void CharSet<wchar\_t>::Print(DebugWriter\* w) const

{

w->Print(L"[");

int start = -1;

for (uint i = 0; i < NumChars; i++)

{

if (Get(UTC(i)))

{

if (start < 0)

{

start = i;

w->PrintEscapedChar(UTC(i));

}

}

else

{

if (start >= 0)

{

if (i > (uint)(start + 1))

{

if (i > (uint)(start + 2))

w->Print(L"-");

w->PrintEscapedChar(UTC(i - 1));

}

start = -1;

}

}

}

if (start >= 0)

{

if ((uint)start < MaxUChar - 1)

w->Print(L"-");

w->PrintEscapedChar(MaxChar);

}

w->Print(L"]");

}

#endif

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

// CharSet<codepoint\_t>

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

CharSet<codepoint\_t>::CharSet()

{

#if DBG

for (int i = 0; i < NumberOfPlanes; i++)

{

this->characterPlanes[i].IsEmpty();

}

#endif

}

void CharSet<codepoint\_t>::FreeBody(ArenaAllocator\* allocator)

{

for (int i = 0; i < NumberOfPlanes; i++)

{

this->characterPlanes[i].FreeBody(allocator);

}

}

void CharSet<codepoint\_t>::Clear(ArenaAllocator\* allocator)

{

for (int i = 0; i < NumberOfPlanes; i++)

{

this->characterPlanes[i].Clear(allocator);

}

}

void CharSet<codepoint\_t>::CloneFrom(ArenaAllocator\* allocator, const CharSet<Char>& other)

{

for (int i = 0; i < NumberOfPlanes; i++)

{

this->characterPlanes[i].Clear(allocator);

this->characterPlanes[i].CloneFrom(allocator, other.characterPlanes[i]);

}

}

void CharSet<codepoint\_t>::CloneSimpleCharsTo(ArenaAllocator\* allocator, CharSet<wchar\_t>& other) const

{

other.CloneFrom(allocator, this->characterPlanes[0]);

}

void CharSet<codepoint\_t>::SetRange(ArenaAllocator\* allocator, Char lc, Char hc)

{

Assert(lc <= hc);

int lowerIndex = this->CharToIndex(lc);

int upperIndex = this->CharToIndex(hc);

if (lowerIndex == upperIndex)

{

this->characterPlanes[lowerIndex].SetRange(allocator, this->RemoveOffset(lc), this->RemoveOffset(hc));

}

else

{

// Do the partial ranges

wchar\_t partialLower = this->RemoveOffset(lc);

wchar\_t partialHigher = this->RemoveOffset(hc);

if (partialLower != 0)

{

this->characterPlanes[lowerIndex].SetRange(allocator, partialLower, Chars<wchar\_t>::MaxUChar);

lowerIndex++;

}

for(; lowerIndex < upperIndex; lowerIndex++)

{

this->characterPlanes[lowerIndex].SetRange(allocator, 0, Chars<wchar\_t>::MaxUChar);

}

this->characterPlanes[upperIndex].SetRange(allocator, 0, partialHigher);

}

}

void CharSet<codepoint\_t>::SetRanges(ArenaAllocator\* allocator, int numSortedPairs, const Char\* sortedPairs)

{

for (int i = 0; i < numSortedPairs \* 2; i += 2)

{

Assert(i == 0 || sortedPairs[i-1] < sortedPairs[i]);

Assert(sortedPairs[i] <= sortedPairs[i+1]);

SetRange(allocator, sortedPairs[i], sortedPairs[i+1]);

}

}

void CharSet<codepoint\_t>::SetNotRanges(ArenaAllocator\* allocator, int numSortedPairs, const Char\* sortedPairs)

{

if (numSortedPairs == 0)

{

for (int i = 0; i < NumberOfPlanes; i++)

{

this->characterPlanes[i].SetRange(allocator, 0, Chars<wchar\_t>::MaxUChar);

}

}

else

{

if (sortedPairs[0] != MinChar)

{

SetRange(allocator, MinChar, sortedPairs[0] - 1);

}

for (int i = 1; i < numSortedPairs \* 2 - 1; i += 2)

{

SetRange(allocator, sortedPairs[i] + 1, sortedPairs[i+1] - 1);

}

if (sortedPairs[numSortedPairs \* 2 - 1] != MaxChar)

{

SetRange(allocator, sortedPairs[numSortedPairs \* 2 - 1] + 1, MaxChar);

}

}

}

void CharSet<codepoint\_t>::UnionInPlace(ArenaAllocator\* allocator, const CharSet<Char>& other)

{

for (int i = 0; i < NumberOfPlanes; i++)

{

this->characterPlanes[i].UnionInPlace(allocator, other.characterPlanes[i]);

}

}

void CharSet<codepoint\_t>::UnionInPlace(ArenaAllocator\* allocator, const CharSet<wchar\_t>& other)

{

this->characterPlanes[0].UnionInPlace(allocator, other);

}

\_Success\_(return)

bool CharSet<codepoint\_t>::GetNextRange(Char searchCharStart, \_Out\_ Char \*outLowerChar, \_Out\_ Char \*outHigherChar)

{

Assert(outLowerChar != nullptr);

Assert(outHigherChar != nullptr);

if (searchCharStart >= 0x110000)

{

return false;

}

wchar\_t currentLowChar = 1, currentHighChar = 0;

int index = this->CharToIndex(searchCharStart);

wchar\_t offsetLessSearchCharStart = this->RemoveOffset(searchCharStart);

for (; index < NumberOfPlanes; index++)

{

if (this->characterPlanes[index].GetNextRange(offsetLessSearchCharStart, &currentLowChar, &currentHighChar))

{

break;

}

offsetLessSearchCharStart = 0x0;

}

if (index == NumberOfPlanes)

{

return false;

}

Assert(currentHighChar >= currentLowChar);

// else found range

\*outLowerChar = this->AddOffset(currentLowChar, index);

\*outHigherChar = this->AddOffset(currentHighChar, index);

// Check if range crosses plane boundaries

index ++;

for (; index < NumberOfPlanes; index++)

{

if (!this->characterPlanes[index].GetNextRange(0x0, &currentLowChar, &currentHighChar) || \*outHigherChar + 1 != this->AddOffset(currentLowChar, index))

{

break;

}

Assert(this->AddOffset(currentHighChar, index) > \*outHigherChar);

\*outHigherChar = this->AddOffset(currentHighChar, index);

}

return true;

}

void CharSet<codepoint\_t>::ToComplement(ArenaAllocator\* allocator, CharSet<Char>& result)

{

for (int i = 0; i < NumberOfPlanes; i++)

{

this->characterPlanes[i].ToComplement(allocator, result.characterPlanes[i]);

}

}

void CharSet<codepoint\_t>::ToSimpleComplement(ArenaAllocator\* allocator, CharSet<Char>& result)

{

this->characterPlanes[0].ToComplement(allocator, result.characterPlanes[0]);

}

void CharSet<codepoint\_t>::ToSimpleComplement(ArenaAllocator\* allocator, CharSet<wchar\_t>& result)

{

this->characterPlanes[0].ToComplement(allocator, result);

}

void CharSet<codepoint\_t>::ToEquivClass(ArenaAllocator\* allocator, CharSet<Char>& result)

{

for (int i = 0; i < NumberOfPlanes; i++)

{

this->characterPlanes[i].ToEquivClassCP(allocator, result, AddOffset(0, i));

}

}

void CharSet<codepoint\_t>::ToSurrogateEquivClass(ArenaAllocator\* allocator, CharSet<Char>& result)

{

this->CloneSimpleCharsTo(allocator, result.characterPlanes[0]);

for (int i = 1; i < NumberOfPlanes; i++)

{

this->characterPlanes[i].ToEquivClassCP(allocator, result, AddOffset(0, i));

}

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void CharSet<codepoint\_t>::Print(DebugWriter\* w) const

{

w->Print(L"Characters 0 - 65535");

for (int i = 0; i < NumberOfPlanes; i++)

{

int base = (i + 1) \* 0x10000;

w->Print(L"Characters %d - %d", base, base + 0xFFFF);

this->characterPlanes[i].Print(w);

}

}

#endif

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

// RuntimeCharSet<wchar\_t>

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

RuntimeCharSet<wchar\_t>::RuntimeCharSet()

{

root = nullptr;

direct.Clear();

}

void RuntimeCharSet<wchar\_t>::FreeBody(ArenaAllocator\* allocator)

{

if (root != nullptr)

{

root->FreeSelf(allocator);

#if DBG

root = nullptr;

#endif

}

}

void RuntimeCharSet<wchar\_t>::CloneFrom(ArenaAllocator\* allocator, const CharSet<Char>& other)

{

Assert(root == nullptr);

Assert(direct.Count() == 0);

if (other.IsCompact())

{

for (uint i = 0; i < other.GetCompactLength(); i++)

{

uint k = other.GetCompactCharU(i);

if (k < CharSetNode::directSize)

direct.Set(k);

else

{

if (root == nullptr)

root = Anew(allocator, CharSetInner);

#if DBG

CharSetNode\* newRoot =

#endif

root->Set(allocator, CharSetNode::levels - 1, k, k);

#if DBG

// NOTE: Since we can add at most MaxCompact characters, we can never fill a leaf or inner node,

// thus we will never need to reallocated nodes

Assert(newRoot == root);

#endif

}

}

}

else

{

root = other.rep.full.root == nullptr ? nullptr : other.rep.full.root->Clone(allocator);

direct.CloneFrom(other.rep.full.direct);

}

}

bool RuntimeCharSet<wchar\_t>::Get\_helper(uint k) const

{

CharSetNode\* curr = root;

for (int level = CharSetNode::levels - 1; level > 0; level--)

{

if (curr == CharSetFull::TheFullNode)

return true;

CharSetInner\* inner = (CharSetInner\*)curr;

uint i = CharSetNode::innerIdx(level, k);

if (inner->children[i] == 0)

return false;

else

curr = inner->children[i];

}

if (curr == CharSetFull::TheFullNode)

return true;

CharSetLeaf\* leaf = (CharSetLeaf\*)curr;

return leaf->vec.Get(CharSetNode::leafIdx(k));

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

// CAUTION: This method is very slow.

void RuntimeCharSet<wchar\_t>::Print(DebugWriter\* w) const

{

w->Print(L"[");

int start = -1;

for (uint i = 0; i < NumChars; i++)

{

if (Get(UTC(i)))

{

if (start < 0)

{

start = i;

w->PrintEscapedChar(UTC(i));

}

}

else

{

if (start >= 0)

{

if (i > (uint)(start + 1))

{

if (i > (uint)(start + 2))

w->Print(L"-");

w->PrintEscapedChar(UTC(i - 1));

}

start = -1;

}

}

}

if (start >= 0)

{

if ((uint)start < MaxUChar - 1)

w->Print(L"-");

w->PrintEscapedChar(MaxChar);

}

w->Print(L"]");

}

#endif

}

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

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

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

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

namespace UnifiedRegex

{

template <typename C>

class CharSet;

template <typename C>

class RuntimeCharSet;

class CharBitvec : private Chars<char>

{

public:

static const int Width = Chars<char>::CharWidth;

static const int Size = NumChars;

private:

static const int wordSize = sizeof(uint32) \* 8;

static const int vecSize = Size / wordSize;

static const uint32 ones = (uint32)-1;

static const uint8 oneBits[Size];

uint32 vec[vecSize];

inline static void setrng(uint32 &v, uint l, uint h)

{

uint w = h - l + 1;

if (w == wordSize)

v = ones;

else

v |= ((1U << w) - 1) << l;

}

inline static void clearrng(uint32 &v, uint l, uint h)

{

uint w = h - l + 1;

if (w == wordSize)

v = 0;

else

v &= ~(((1U << w) - 1) << l);

}

public:

inline void CloneFrom(const CharBitvec& other)

{

for (int w = 0; w < vecSize; w++)

vec[w] = other.vec[w];

}

inline void Clear()

{

for (int w = 0; w < vecSize; w++)

vec[w] = 0;

}

inline void SetAll()

{

for (int w = 0; w < vecSize; w++)

vec[w] = ones;

}

inline void Set(uint k)

{

Assert(k < Size);

\_\_assume(k < Size);

if (k < Size)

vec[k / wordSize] |= 1U << (k % wordSize);

}

inline void SetRange(uint l, uint h)

{

Assert(l < Size);

Assert(h < Size);

\_\_assume(l < Size);

\_\_assume(h < Size);

if (l < Size && h < Size)

{

if (l == h)

vec[l / wordSize] |= 1U << (l % wordSize);

else if (l < h)

{

int lw = l / wordSize;

int hw = h / wordSize;

int lo = l % wordSize;

int hio = h % wordSize;

if (lw == hw)

setrng(vec[lw], lo, hio);

else

{

setrng(vec[lw], lo, wordSize-1);

for (int w = lw + 1; w < hw; w++)

vec[w] = ones;

setrng(vec[hw], 0, hio);

}

}

}

}

inline void ClearRange(uint l, uint h)

{

Assert(l < Size);

Assert(h < Size);

\_\_assume(l < Size);

\_\_assume(h < Size);

if (l < Size && h < Size)

{

if (l == h)

{

vec[l / wordSize] &= ~(1U << (l % wordSize));

}

else if (l < h)

{

int lw = l / wordSize;

int hw = h / wordSize;

int lo = l % wordSize;

int hio = h % wordSize;

if (lw == hw)

{

clearrng(vec[lw], lo, hio);

}

else

{

clearrng(vec[lw], lo, wordSize-1);

for (int w = lw + 1; w < hw; w++)

vec[w] = 0;

clearrng(vec[hw], 0, hio);

}

}

}

}

inline bool IsEmpty()

{

for (int i = 0; i < vecSize; i++)

{

if(vec[i] != 0)

{

return false;

}

}

return true;

}

inline void UnionInPlace(const CharBitvec& other)

{

for (int w = 0; w < vecSize; w++)

vec[w] |= other.vec[w];

}

inline bool UnionInPlaceFullCheck(const CharBitvec& other)

{

bool isFull = true;

for (int w = 0; w < vecSize; w++)

{

vec[w] |= other.vec[w];

if (vec[w] != ones)

isFull = false;

}

return isFull;

}

inline bool Get(uint k) const

{

Assert(k < Size);

\_\_assume(k < Size);

return ((vec[k / wordSize] >> (k % wordSize)) & 1) != 0;

}

inline bool IsFull() const

{

for (int w = 0; w < vecSize; w++)

{

if (vec[w] != ones)

return false;

}

return true;

}

inline bool IsSubsetOf(const CharBitvec& other) const

{

for (int w = 0; w < vecSize; w++)

{

uint32 v = other.vec[w];

if (v != (vec[w] | v))

return false;

}

return true;

}

inline bool IsEqualTo(const CharBitvec& other) const

{

for (int w = 0; w < vecSize; w++)

{

if (vec[w] != other.vec[w])

return false;

}

return true;

}

uint Count() const;

int NextSet(int k) const;

int NextClear(int k) const;

template <typename C>

void ToComplement(ArenaAllocator\* allocator, uint base, CharSet<C>& result) const;

template <typename C>

void ToEquivClass(ArenaAllocator\* allocator, uint base, uint& tblidx, CharSet<C>& result, codepoint\_t baseOffset = 0x0) const;

};

template <typename C>

class CharSet {};

struct CharSetNode : protected Chars<wchar\_t>

{

static const int directBits = Chars<char>::CharWidth;

static const uint directSize = Chars<char>::NumChars;

static const uint bitsPerInnerLevel = 4;

static const uint branchingPerInnerLevel = 1 << bitsPerInnerLevel;

static const uint innerMask = branchingPerInnerLevel - 1;

static const int bitsPerLeafLevel = CharBitvec::Width;

static const int branchingPerLeafLevel = CharBitvec::Size;

static const uint leafMask = branchingPerLeafLevel - 1;

static const uint levels = 1 + (CharWidth - bitsPerLeafLevel) / bitsPerInnerLevel;

inline static uint innerIdx(uint level, uint v)

{

return (v >> ((level + 1) \* bitsPerInnerLevel)) & innerMask;

}

inline static uint indexToValue(uint level, uint index, uint offset)

{

Assert((index & innerMask) == index);

Assert((uint)(1 << ((level + 1) \* bitsPerInnerLevel)) > offset);

return (index << ((level + 1) \* bitsPerInnerLevel)) + offset;

}

inline static uint leafIdx(uint v)

{

return v & leafMask;

}

inline static uint lim(uint level)

{

return (1U << (bitsPerLeafLevel + level \* bitsPerInnerLevel)) - 1;

}

inline static uint remain(uint level, uint v)

{

return v & lim(level);

}

virtual void FreeSelf(ArenaAllocator\* allocator) = 0;

virtual CharSetNode\* Clone(ArenaAllocator\* allocator) const = 0;

virtual CharSetNode\* Set(ArenaAllocator\* allocator, uint level, uint l, uint h) = 0;

virtual CharSetNode\* ClearRange(ArenaAllocator\* allocator, uint level, uint l, uint h) = 0;

virtual CharSetNode\* UnionInPlace(ArenaAllocator\* allocator, uint level, const CharSetNode\* other) = 0;

virtual bool Get(uint level, uint k) const = 0;

virtual void ToComplement(ArenaAllocator\* allocator, uint level, uint base, CharSet<Char>& result) const = 0;

virtual void ToEquivClassW(ArenaAllocator\* allocator, uint level, uint base, uint& tblidx, CharSet<wchar\_t>& result) const = 0;

virtual void ToEquivClassCP(ArenaAllocator\* allocator, uint level, uint base, uint& tblidx, CharSet<codepoint\_t>& result, codepoint\_t baseOffset) const = 0;

virtual bool IsSubsetOf(uint level, const CharSetNode\* other) const = 0;

virtual bool IsEqualTo(uint level, const CharSetNode\* other) const = 0;

virtual uint Count(uint level) const = 0;

\_Success\_(return) virtual bool GetNextRange(uint level, Char searchCharStart, \_Out\_ Char \*outLowerChar, \_Out\_ Char \*outHigherChar) const = 0;

#if DBG

virtual bool IsLeaf() const = 0;

#endif

static CharSetNode\* For(ArenaAllocator\* allocator, int level);

};

struct CharSetFull : CharSetNode

{

private:

template <typename C>

void ToEquivClass(ArenaAllocator\* allocator, uint level, uint base, uint& tblidx, CharSet<C>& result, codepoint\_t baseOffset = 0x0) const;

public:

static CharSetFull Instance;

static CharSetFull\* const TheFullNode;

CharSetFull();

void FreeSelf(ArenaAllocator\* allocator) override;

CharSetNode\* Clone(ArenaAllocator\* allocator) const override;

CharSetNode\* Set(ArenaAllocator\* allocator, uint level, uint l, uint h) override;

CharSetNode\* ClearRange(ArenaAllocator\* allocator, uint level, uint l, uint h) override;

CharSetNode\* UnionInPlace(ArenaAllocator\* allocator, uint level, const CharSetNode\* other) override;

bool Get(uint level, uint k) const override;

void ToComplement(ArenaAllocator\* allocator, uint level, uint base, CharSet<Char>& result) const override;

void ToEquivClassW(ArenaAllocator\* allocator, uint level, uint base, uint& tblidx, CharSet<wchar\_t>& result) const override;

void ToEquivClassCP(ArenaAllocator\* allocator, uint level, uint base, uint& tblidx, CharSet<codepoint\_t>& result, codepoint\_t baseOffset) const override;

bool IsSubsetOf(uint level, const CharSetNode\* other) const override;

bool IsEqualTo(uint level, const CharSetNode\* other) const override;

uint Count(uint level) const override;

\_Success\_(return) bool GetNextRange(uint level, Char searchCharStart, \_Out\_ Char \*outLowerChar, \_Out\_ Char \*outHigherChar) const override;

#if DBG

bool IsLeaf() const override;

#endif

};

struct CharSetInner sealed : CharSetNode

{

private:

template <typename C>

void ToEquivClass(ArenaAllocator\* allocator, uint level, uint base, uint& tblidx, CharSet<C>& result) const;

public:

CharSetNode\* children[branchingPerInnerLevel];

CharSetInner();

void FreeSelf(ArenaAllocator\* allocator) override;

CharSetNode\* Clone(ArenaAllocator\* allocator) const override;

CharSetNode\* Set(ArenaAllocator\* allocator, uint level, uint l, uint h) override;

CharSetNode\* ClearRange(ArenaAllocator\* allocator, uint level, uint l, uint h) override;

CharSetNode\* UnionInPlace(ArenaAllocator\* allocator, uint level, const CharSetNode\* other) override;

bool Get(uint level, uint k) const override;

void ToComplement(ArenaAllocator\* allocator, uint level, uint base, CharSet<Char>& result) const override;\

void ToEquivClassW(ArenaAllocator\* allocator, uint level, uint base, uint& tblidx, CharSet<wchar\_t>& result) const override;

void ToEquivClassCP(ArenaAllocator\* allocator, uint level, uint base, uint& tblidx, CharSet<codepoint\_t>& result, codepoint\_t baseOffset) const override;

bool IsSubsetOf(uint level, const CharSetNode\* other) const override;

bool IsEqualTo(uint level, const CharSetNode\* other) const override;

uint Count(uint level) const override;

\_Success\_(return) bool GetNextRange(uint level, Char searchCharStart, \_Out\_ Char \*outLowerChar, \_Out\_ Char \*outHigherChar) const override;

#if DBG

bool IsLeaf() const override;

#endif

};

struct CharSetLeaf sealed: CharSetNode

{

private:

template <typename C>

void ToEquivClass(ArenaAllocator\* allocator, uint level, uint base, uint& tblidx, CharSet<C>& result, codepoint\_t baseOffset = 0x0) const;

public:

CharBitvec vec;

CharSetLeaf();

void FreeSelf(ArenaAllocator\* allocator) override;

CharSetNode\* Clone(ArenaAllocator\* allocator) const override;

CharSetNode\* Set(ArenaAllocator\* allocator, uint level, uint l, uint h) override;

CharSetNode\* ClearRange(ArenaAllocator\* allocator, uint level, uint l, uint h) override;

CharSetNode\* UnionInPlace(ArenaAllocator\* allocator, uint level, const CharSetNode\* other) override;

bool Get(uint level, uint k) const override;

void ToComplement(ArenaAllocator\* allocator, uint level, uint base, CharSet<Char>& result) const override;

void ToEquivClassW(ArenaAllocator\* allocator, uint level, uint base, uint& tblidx, CharSet<wchar\_t>& result) const override;

void ToEquivClassCP(ArenaAllocator\* allocator, uint level, uint base, uint& tblidx, CharSet<codepoint\_t>& result, codepoint\_t baseOffset) const override;

bool IsSubsetOf(uint level, const CharSetNode\* other) const override;

bool IsEqualTo(uint level, const CharSetNode\* other) const override;

uint Count(uint level) const override;

\_Success\_(return) bool GetNextRange(uint level, Char searchCharStart, \_Out\_ Char \*outLowerChar, \_Out\_ Char \*outHigherChar) const override;

#if DBG

bool IsLeaf() const override;

#endif

};

template <>

class CharSet<wchar\_t> : private Chars<wchar\_t>

{

public:

static const uint MaxCompact = 4;

static const uint emptySlot = (uint)-1;

struct CompactRep

{

// 1 + number of distinct characters, 1..MaxCompact+1

size\_t countPlusOne;

// Characters, in no particular order, or (uint)-1 for tail empty slots

uint cs[MaxCompact];

uint8 padding[sizeof(CharBitvec) - sizeof(uint) \* MaxCompact];

};

struct FullRep

{

// Trie for remaining characters. Pointer value will be 0 or >> MaxCompact.

CharSetNode\* root;

// Entries for first 256 characters

CharBitvec direct;

};

union Rep

{

struct CompactRep compact;

struct FullRep full;

} rep;

static const int compactSize = sizeof(CompactRep);

static const int fullSize = sizeof(FullRep);

inline bool IsCompact() const { return rep.compact.countPlusOne - 1 <= MaxCompact; }

void SwitchRepresentations(ArenaAllocator\* allocator);

void Sort();

public:

CharSet();

void FreeBody(ArenaAllocator\* allocator);

void Clear(ArenaAllocator\* allocator);

void CloneFrom(ArenaAllocator\* allocator, const CharSet<Char>& other);

void CloneNonSurrogateCodeUnitsTo(ArenaAllocator\* allocator, CharSet<Char>& other);

void CloneSurrogateCodeUnitsTo(ArenaAllocator\* allocator, CharSet<Char>& other);

inline void Set(ArenaAllocator\* allocator, Char kc) { SetRange(allocator, kc, kc); }

void SetRange(ArenaAllocator\* allocator, Char lc, Char hc);

void SubtractRange(ArenaAllocator\* allocator, Char lc, Char hc);

void SetRanges(ArenaAllocator\* allocator, int numSortedPairs, const Char\* sortedPairs);

void SetNotRanges(ArenaAllocator\* allocator, int numSortedPairs, const Char\* sortedPairs);

void UnionInPlace(ArenaAllocator\* allocator, const CharSet<Char>& other);

\_Success\_(return) bool GetNextRange(Char searchCharStart, \_Out\_ Char \*outLowerChar, \_Out\_ Char \*outHigherChar);

bool Get\_helper(uint k) const;

\_\_inline bool Get(Char kc) const

{

if (IsCompact())

{

Assert(MaxCompact == 4);

return rep.compact.cs[0] == CTU(kc) ||

rep.compact.cs[1] == CTU(kc) ||

rep.compact.cs[2] == CTU(kc) ||

rep.compact.cs[3] == CTU(kc);

}

else

{

if (CTU(kc) < CharSetNode::directSize)

return rep.full.direct.Get(CTU(kc));

else if (rep.full.root == 0)

return false;

else

return Get\_helper(CTU(kc));

}

}

inline bool IsEmpty() const

{

return rep.compact.countPlusOne == 1;

}

inline bool IsSingleton() const

{

return rep.compact.countPlusOne == 2;

}

// Helpers to clean up the code

inline uint GetCompactLength() const

{

Assert(IsCompact());

return (uint)(rep.compact.countPlusOne - 1u);

}

inline void SetCompactLength(size\_t length)

{

rep.compact.countPlusOne = length + 1;

}

inline uint GetCompactCharU(uint index) const

{

Assert(index < this->GetCompactLength());

Assert(IsCompact());

Assert(rep.compact.cs[index] <= MaxUChar);

return rep.compact.cs[index];

}

inline Char GetCompactChar(uint index) const

{

return (Char)(GetCompactCharU(index));

}

//Replaces an existing character with a new value

inline void ReplaceCompactChar(uint index, Char value)

{

ReplaceCompactChar(index, (Char)(value));

}

//Replaces an existing character with a new value

inline void ReplaceCompactCharU(uint index, uint value)

{

Assert(index < this->GetCompactLength());

Assert(IsCompact());

Assert(value <= MaxUChar);

rep.compact.cs[index] = value;

}

inline void ClearCompactChar(uint index)

{

Assert(index < this->GetCompactLength());

Assert(IsCompact());

rep.compact.cs[index] = emptySlot;

}

// Adds the character to the end, assuming there is enough space. (Assert in place)

// Increments count.

inline void AddCompactCharU(uint value)

{

Assert(this->GetCompactLength() < MaxCompact);

Assert(IsCompact());

rep.compact.cs[this->GetCompactLength()] = value;

rep.compact.countPlusOne += 1;

}

// Adds the character to the end, assuming there is enough space. (Assert in place)

// Increments count.

inline void AddCompactChar(Char value)

{

AddCompactCharU((Char)(value));

}

// This performs a check to see if the index is the last char, if so sets it to emptySlot

// If not, replaces it with last index.

inline void RemoveCompactChar(uint index)

{

Assert(index < this->GetCompactLength());

Assert(IsCompact());

if (index == this->GetCompactLength() - 1)

{

this->ClearCompactChar(index);

}

else

{

this->ReplaceCompactCharU(index, this->GetCompactCharU((uint)this->GetCompactLength() - 1));

}

rep.compact.countPlusOne -= 1;

}

inline wchar\_t Singleton() const

{

Assert(IsSingleton());

Assert(rep.compact.cs[0] <= MaxUChar);

return UTC(rep.compact.cs[0]);

}

int GetCompactEntries(uint max, \_\_out\_ecount(max) Char\* entries) const;

bool IsSubsetOf(const CharSet<Char>& other) const;

bool IsEqualTo(const CharSet<Char>& other) const;

inline uint Count() const

{

if (IsCompact())

return (uint)rep.compact.countPlusOne - 1;

else if (rep.full.root == 0)

return rep.full.direct.Count();

else

{

//The bit vector

Assert(rep.full.root == CharSetFull::TheFullNode || rep.full.root->Count(CharSetNode::levels - 1) <= 0xFF00);

return rep.full.direct.Count() + (rep.full.root == CharSetFull::TheFullNode ? 0xFF00 : rep.full.root->Count(CharSetNode::levels - 1));

}

}

// NOTE: These are not 'const' methods since they may sort the compact representation internally

void ToComplement(ArenaAllocator\* allocator, CharSet<Char>& result);

void ToEquivClass(ArenaAllocator\* allocator, CharSet<Char>& result);

void ToEquivClassCP(ArenaAllocator\* allocator, CharSet<codepoint\_t>& result, codepoint\_t baseOffset);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w) const;

#endif

};

template <>

class CharSet<codepoint\_t> : private Chars<codepoint\_t>

{

static const int NumberOfPlanes = 17;

private:

// Character planes are composed of 65536 characters each.

// First plane is the Basic Multilingual Plane (characters 0 - 65535)

// Every subsequent plane also stores characters in the form [0 - 65535]; to get the actual value, add 'index \* 0x10000' to it

CharSet<wchar\_t> characterPlanes [NumberOfPlanes];

// Takes a character, and returns the index of the CharSet<wchar\_t> that holds it.

inline int CharToIndex(Char c) const

{

Assert(c <= Chars<codepoint\_t>::MaxUChar);

return (int)(CTU(c) / (Chars<wchar\_t>::MaxUChar + 1));

}

// Takes a character, and removes the offset to make it < 0x10000

inline wchar\_t RemoveOffset(Char c) const

{

Assert(c <= Chars<codepoint\_t>::MaxUChar);

return (wchar\_t)(CTU(c) % 0x10000);

}

// Takes a character, and removes the offset to make it < 0x10000

inline Char AddOffset(wchar\_t c, int index) const

{

Assert(c <= Chars<wchar\_t>::MaxUChar);

Assert(index >= 0);

Assert(index < NumberOfPlanes);

return (Char)(c) + 0x10000 \* index;

}

public:

CharSet();

void FreeBody(ArenaAllocator\* allocator);

void Clear(ArenaAllocator\* allocator);

void CloneFrom(ArenaAllocator\* allocator, const CharSet<Char>& other);

void CloneSimpleCharsTo(ArenaAllocator\* allocator, CharSet<wchar\_t>& other) const;

inline void CloneNonSurrogateCodeUnitsTo(ArenaAllocator\* allocator, CharSet<wchar\_t>& other)

{

Assert(this->SimpleCharCount() > 0);

AssertMsg(this->ContainSurrogateCodeUnits(), "This doesn't contain surrogate code units, a simple clone is faster.");

this->characterPlanes[0].CloneNonSurrogateCodeUnitsTo(allocator, other);

}

inline void CloneSurrogateCodeUnitsTo(ArenaAllocator\* allocator, CharSet<wchar\_t>& other)

{

Assert(this->SimpleCharCount() > 0);

AssertMsg(this->ContainSurrogateCodeUnits(), "This doesn't contain surrogate code units, will not produce any result.");

this->characterPlanes[0].CloneSurrogateCodeUnitsTo(allocator, other);

}

inline void Set(ArenaAllocator\* allocator, Char kc) { SetRange(allocator, kc, kc); }

inline bool ContainSurrogateCodeUnits()

{

wchar\_t outLower = 0xFFFF, ignore = 0x0;

return this->characterPlanes[0].GetNextRange(0xD800, &outLower, &ignore) ? outLower <= 0xDFFF : false;

}

void SetRange(ArenaAllocator\* allocator, Char lc, Char hc);

void SetRanges(ArenaAllocator\* allocator, int numSortedPairs, const Char\* sortedPairs);

void SetNotRanges(ArenaAllocator\* allocator, int numSortedPairs, const Char\* sortedPairs);

void UnionInPlace(ArenaAllocator\* allocator, const CharSet<Char>& other);

void UnionInPlace(ArenaAllocator\* allocator, const CharSet<wchar\_t>& other);

\_Success\_(return) bool GetNextRange(Char searchCharStart, \_Out\_ Char \*outLowerChar, \_Out\_ Char \*outHigherChar);

inline bool Get(Char kc) const

{

return this->characterPlanes[CharToIndex(kc)].Get(RemoveOffset(kc));

}

inline bool IsEmpty() const

{

for (int i = 0; i < NumberOfPlanes; i++)

{

if (!this->characterPlanes[i].IsEmpty())

{

return false;

}

}

return true;

}

inline bool IsSimpleCharASingleton() const

{

return this->characterPlanes[0].IsSingleton();

}

inline wchar\_t SimpleCharSingleton() const

{

return this->characterPlanes[0].Singleton();

}

inline bool IsSingleton() const

{

return this->Count() == 1;

}

inline codepoint\_t Singleton() const

{

Assert(IsSingleton());

for (int i = 0; i < NumberOfPlanes; i++)

{

if (this->characterPlanes[i].IsSingleton())

{

return AddOffset(this->characterPlanes[i].Singleton(), i);

}

}

AssertMsg(false, "Should not reach here, first Assert verifies we are a singleton.");

return INVALID\_CODEPOINT;

}

bool IsSubsetOf(const CharSet<Char>& other) const;

bool IsEqualTo(const CharSet<Char>& other) const;

inline uint Count() const

{

uint totalCount = 0;

for (int i = 0; i < NumberOfPlanes; i++)

{

totalCount += this->characterPlanes[i].Count();

}

return totalCount;

}

inline uint SimpleCharCount() const

{

return this->characterPlanes[0].Count();

}

// NOTE: These are not 'const' methods since they may sort the compact representation internally

void ToComplement(ArenaAllocator\* allocator, CharSet<Char>& result);

void ToSimpleComplement(ArenaAllocator\* allocator, CharSet<codepoint\_t>& result);

void ToSimpleComplement(ArenaAllocator\* allocator, CharSet<wchar\_t>& result);

void ToEquivClass(ArenaAllocator\* allocator, CharSet<Char>& result);

void ToSurrogateEquivClass(ArenaAllocator\* allocator, CharSet<Char>& result);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w) const;

#endif

};

template <>

class RuntimeCharSet<wchar\_t> : private Chars<wchar\_t>

{

private:

// Trie for remaining characters. Pointer value will be 0 or >> MaxCompact.

CharSetNode\* root;

// Entries for first 256 characters

CharBitvec direct;

public:

RuntimeCharSet();

void FreeBody(ArenaAllocator\* allocator);

void CloneFrom(ArenaAllocator\* allocator, const CharSet<Char>& other);

bool Get\_helper(uint k) const;

\_\_inline bool Get(Char kc) const

{

if (CTU(kc) < CharSetNode::directSize)

return direct.Get(CTU(kc));

else if (root == 0)

return false;

else

return Get\_helper(CTU(kc));

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w) const;

#endif

};

typedef CharSet<wchar\_t> UnicodeCharSet;

typedef RuntimeCharSet<wchar\_t> UnicodeRuntimeCharSet;

}

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

// 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"

namespace UnifiedRegex

{

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

// CharTrie

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

\_\_inline bool CharTrie::Find(Char c, int& outi)

{

if (count == 0)

{

outi = 0;

return false;

}

int l = 0;

int h = count - 1;

while (true)

{

int m = (l + h) / 2;

if (children[m].c == c)

{

outi = m;

return true;

}

else if (CTU(children[m].c) < CTU(c))

{

l = m + 1;

if (l > h)

{

outi = l;

return false;

}

}

else

{

h = m - 1;

if (h < l)

{

outi = m;

return false;

}

}

}

return false;

}

void CharTrie::FreeBody(ArenaAllocator\* allocator)

{

for (int i = 0; i < count; i++)

children[i].node.FreeBody(allocator);

if (capacity > 0)

AdeleteArray(allocator, capacity, children);

#if DBG

count = 0;

capacity = 0;

children = 0;

#endif

}

CharTrie\* CharTrie::Add(ArenaAllocator\* allocator, Char c)

{

int i;

if (!Find(c, i))

{

if (capacity <= count)

{

int newCapacity = max(capacity \* 2, initCapacity);

children = (CharTrieEntry\*)allocator->Realloc(children, capacity \* sizeof(CharTrieEntry), newCapacity \* sizeof(CharTrieEntry));

capacity = newCapacity;

}

for (int j = count; j > i; j--)

{

children[j].c = children[j - 1].c;

children[j].node = children[j - 1].node;

}

children[i].c = c;

children[i].node.Reset();

count++;

}

return &children[i].node;

}

bool CharTrie::IsDepthZero() const

{

return isAccepting && count == 0;

}

bool CharTrie::IsDepthOne() const

{

if (isAccepting)

return 0;

for (int i = 0; i < count; i++)

{

if (!children[i].node.IsDepthZero())

return false;

}

return true;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void CharTrie::Print(DebugWriter\* w) const

{

w->Indent();

if (isAccepting)

w->PrintEOL(L"<accept>");

for (int i = 0; i < count; i++)

{

w->PrintQuotedChar(children[i].c);

w->EOL();

children[i].node.Print(w);

}

w->Unindent();

}

#endif

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

// RuntimeCharTrie

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

bool RuntimeCharTrie::Match

(const Char\* const input

, const CharCount inputLength

, CharCount& inputOffset

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, RegexStats\* stats

#endif

) const

{

const RuntimeCharTrie\* curr = this;

while (true)

{

if (curr->count == 0)

return true;

if (inputOffset >= inputLength)

return false;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

if (stats != 0)

stats->numCompares++;

#endif

#if 0

int l = 0;

int h = curr->count - 1;

while (true)

{

if (l > h)

return false;

int m = (l + h) / 2;

if (curr->children[m].c == input[inputOffset])

{

inputOffset++;

curr = &curr->children[m].node;

break;

}

else if (CTU(curr->children[m].c) < CTU(input[inputOffset]))

l = m + 1;

else

h = m - 1;

}

#else

int i = 0;

while (true)

{

if (curr->children[i].c == input[inputOffset])

{

inputOffset++;

curr = &curr->children[i].node;

break;

}

else if (curr->children[i].c > input[inputOffset])

return false;

else if (++i >= curr->count)

return false;

}

#endif

}

}

void RuntimeCharTrie::FreeBody(ArenaAllocator\* allocator)

{

for (int i = 0; i < count; i++)

children[i].node.FreeBody(allocator);

if (count > 0)

AdeleteArray(allocator, count, children);

#if DBG

count = 0;

children = 0;

#endif

}

void RuntimeCharTrie::CloneFrom(ArenaAllocator\* allocator, const CharTrie& other)

{

count = other.count;

if (count > 0)

{

children = AnewArray(allocator, RuntimeCharTrieEntry, count);

for (int i = 0; i < count; i++)

{

children[i].c = other.children[i].c;

children[i].node.CloneFrom(allocator, other.children[i].node);

}

}

else

children = 0;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void RuntimeCharTrie::Print(DebugWriter\* w) const

{

w->Indent();

for (int i = 0; i < count; i++)

{

w->PrintQuotedChar(children[i].c);

w->EOL();

children[i].node.Print(w);

}

w->Unindent();

}

#endif

}

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

// 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 UnifiedRegex

{

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

// CharTrie

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

// FORWARD

struct CharTrieEntry;

class CharTrie : private Chars<wchar\_t>

{

friend class RuntimeCharTrie;

static const int initCapacity = 4;

CharTrieEntry\* children;

bool isAccepting;

int capacity;

int count;

// Array of capacity entries, first count are used, in increasing character order

\_\_inline bool Find(Char c, int& outi);

public:

inline CharTrie() : isAccepting(false), capacity(0), count(0), children(0) {}

inline void Reset() { isAccepting = false; capacity = 0; count = 0; children = 0; }

void FreeBody(ArenaAllocator\* allocator);

inline int Count() const { return count; }

inline bool IsAccepting() const { return isAccepting; }

inline void SetAccepting() { isAccepting = true; }

CharTrie\* Add(ArenaAllocator\* allocator, Char c);

bool IsDepthZero() const;

bool IsDepthOne() const;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w) const;

#endif

};

struct CharTrieEntry : private Chars<wchar\_t>

{

Char c;

CharTrie node;

};

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

// RuntimeCharTrie

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

// FORWARD

struct RuntimeCharTrieEntry;

class RuntimeCharTrie : private Chars<wchar\_t>

{

int count;

// Array of count entries, in increasing character order

RuntimeCharTrieEntry\* children;

public:

inline RuntimeCharTrie() : count(0), children(0) {}

void FreeBody(ArenaAllocator\* allocator);

void CloneFrom(ArenaAllocator\* allocator, const CharTrie& other);

bool Match

( const Char\* const input

, const CharCount inputLength

, CharCount &inputOffset

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, RegexStats\* stats

#endif

) const;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w) const;

#endif

};

struct RuntimeCharTrieEntry : private Chars<wchar\_t>

{

Char c;

RuntimeCharTrie node;

};

}

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

// 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"

#if DEBUG

#include <stdarg.h>

#endif //DEBUG

void ErrHandler::Throw(HRESULT hr)

{

Assert(fInited);

Assert(FAILED(hr));

m\_hr = hr;

throw ParseExceptionObject(hr);

}

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

// 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

enum

{

#undef LSC\_ERROR\_MSG

#define LSC\_ERROR\_MSG(err, name, str) name = MAKE\_HR(err),

#include "perrors.h"

#undef LSC\_ERROR\_MSG

MWUNUSED\_ENUM

};

class ParseExceptionObject

{

public:

ParseExceptionObject(HRESULT hr) : m\_hr(hr) {}

HRESULT GetError() { return m\_hr; }

private:

HRESULT m\_hr;

};

typedef void (\*ErrorCallback)(void \*data, HRESULT hr);

class ErrHandler

{

public:

HRESULT m\_hr;

void \*m\_data;

ErrorCallback m\_callback;

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

#if DEBUG

BOOL fInited;

ErrHandler()

{ fInited = FALSE; }

#endif //DEBUG

};

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

// 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"

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

namespace UnifiedRegex

{

const wchar\_t\* const DebugWriter::hex = L"0123456789abcdef";

DebugWriter::DebugWriter() : indent(0), nlPending(false)

{

}

void \_\_cdecl DebugWriter::Print(const Char \*form, ...)

{

va\_list argptr;

va\_start(argptr, form);

int len = \_vsnwprintf\_s(buf, bufLen, \_TRUNCATE, form, argptr);

if (len < 0)

Output::Print(L"<not enough buffer space to format>");

else

{

if (len > 0)

CheckForNewline();

Output::Print(L"%s", buf);

}

}

void \_\_cdecl DebugWriter::PrintEOL(const Char \*form, ...)

{

va\_list argptr;

va\_start(argptr, form);

int len = \_vsnwprintf\_s(buf, bufLen, \_TRUNCATE, form, argptr);

Assert(len >= 0 && len < bufLen - 1);

if (len > 0)

CheckForNewline();

Output::Print(L"%s", buf);

EOL();

}

void DebugWriter::PrintEscapedString(const Char\* str, CharCount len)

{

Assert(str != 0);

CheckForNewline();

const Char\* pl = str + len;

for (const Char\* p = str; p < pl; p++)

{

if (\*p == '"')

Output::Print(L"\\\"");

else

PrintEscapedChar(\*p);

}

}

void DebugWriter::PrintQuotedString(const Char\* str, CharCount len)

{

CheckForNewline();

if (str == 0)

Output::Print(L"null");

else

{

Output::Print(L"\"");

PrintEscapedString(str, len);

Output::Print(L"\"");

}

}

void DebugWriter::PrintEscapedChar(const Char c)

{

CheckForNewline();

if (c > 0xff)

Output::Print(L"\\u%lc%lc%lc%lc", hex[c >> 12], hex[(c >> 8) & 0xf], hex[(c >> 4) & 0xf], hex[c & 0xf]);

else if (c < ' ' || c > '~')

Output::Print(L"\\x%lc%lc", hex[c >> 4], hex[c & 0xf]);

else

Output::Print(L"%lc", c);

}

void DebugWriter::PrintQuotedChar(const Char c)

{

CheckForNewline();

Output::Print(L"'");

if (c == '\'')

Output::Print(L"\\'");

else

PrintEscapedChar(c);

Output::Print(L"'");

}

void DebugWriter::EOL()

{

CheckForNewline();

nlPending = true;

}

void DebugWriter::Indent()

{

indent++;

}

void DebugWriter::Unindent()

{

indent--;

}

void DebugWriter::Flush()

{

Output::Print(L"\n");

Output::Flush();

nlPending = false;

}

void DebugWriter::BeginLine()

{

Output::Print(L"\n%\*s", indent \* 4, L"");

}

}

#endif

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

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

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

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

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

#pragma once

namespace UnifiedRegex

{

class DebugWriter : private Chars<wchar\_t>

{

private:

static const Char\* const hex;

static const int bufLen = 2048;

Char buf[bufLen];

int indent;

bool nlPending;

public:

DebugWriter();

void \_\_cdecl Print(const Char \*form, ...);

void \_\_cdecl PrintEOL(const Char \*form, ...);

void PrintEscapedString(const Char \*str, CharCount len);

void PrintQuotedString(const Char \*str, CharCount len);

void PrintEscapedChar(Char c);

void PrintQuotedChar(Char c);

void EOL();

void Indent();

void Unindent();

void Flush();

private:

inline void CheckForNewline()

{

if (nlPending)

{

BeginLine();

nlPending = false;

}

}

void BeginLine();

};

}

#endif

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

// 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 "errstr.h"

// scaffolding - get a g\_hInstance from scrbgase.cpp

HANDLE g\_hInstance;

// Used as a prefix to generate the resource dll name.

const wchar\_t g\_wszPrefix[] = L"js";

static BOOL FGetStringFromLibrary(HMODULE hlib, int istring, \_\_out\_ecount(cchMax) WCHAR \* psz, int cchMax)

{

// NOTE - istring is expected to be HRESULT

Assert(0 < cchMax);

AssertArrMem(psz, cchMax);

HGLOBAL hgl = NULL;

WCHAR \* pchRes = NULL;

HRSRC hrsrc;

WCHAR \* pchCur;

int cch;

int cstring;

DWORD cbRes;

int itable = ((WORD)istring >> 4) + 1;

istring &= 0x0F;

BOOL fRet = FALSE;

psz[0] = '\0';

if (NULL == hlib)

goto LError;

hrsrc = FindResourceEx((HMODULE)hlib, RT\_STRING, MAKEINTRESOURCE(itable), 0);

if (NULL == hrsrc)

goto LError;

hgl = LoadResource((HMODULE)hlib, hrsrc);

if (NULL == hgl)

goto LError;

pchRes = (WCHAR \*)LockResource(hgl);

if (NULL == pchRes)

goto LError;

cbRes = SizeofResource((HMODULE)hlib, hrsrc);

if (cbRes < sizeof(WORD))

goto LError;

pchCur = pchRes;

for (cstring = istring; cstring-- > 0;)

{

if (cbRes - sizeof(WORD) < sizeof(WCHAR) \* (pchCur - pchRes))

goto LError;

cch = (\*(WORD \*) pchCur) + 1;

if (cch <= 0)

goto LError;

if (cbRes < sizeof(WCHAR) \* cch)

goto LError;

if (cbRes - sizeof(WCHAR) \* cch < sizeof(WCHAR) \* (pchCur - pchRes))

goto LError;

pchCur += cch;

}

if (cbRes - sizeof(WORD) < sizeof(WCHAR) \* (pchCur - pchRes))

goto LError;

cch = \* (WORD \*) pchCur;

if (cch <= 0)

goto LError;

if (cbRes < sizeof(WCHAR) \* (cch + 1))

goto LError;

if (cbRes - sizeof(WCHAR) \* (cch + 1) < sizeof(WCHAR) \* (pchCur - pchRes))

goto LError;

if (cch > cchMax - 1)

cch = cchMax - 1;

js\_memcpy\_s(psz, cchMax \* sizeof(WCHAR), pchCur + 1, cch \* sizeof(WCHAR));

psz[cch] = '\0';

fRet = TRUE;

LError:

#if !\_WIN32 && !\_WIN64

//

// Unlock/FreeResource non-essential on win32/64.

//

if (NULL != hgl)

{

if (NULL != pchRes)

UnlockResource(hgl);

FreeResource(hgl);

}

#endif

return fRet;

}

BOOL FGetResourceString(long isz, \_\_out\_ecount(cchMax) OLECHAR \*psz, int cchMax)

{

return FGetStringFromLibrary((HINSTANCE)g\_hInstance, isz, psz, cchMax);

}

// Get a bstr version of the error string

\_\_declspec(noinline) // Don't inline. This function needs 2KB stack.

BSTR BstrGetResourceString(long isz)

{

// NOTE - isz is expected to be HRESULT

OLECHAR szT[1024];

if (!FGetResourceString(isz, szT,

sizeof(szT) / sizeof(szT[0]) - 1))

{

return NULL;

}

return SysAllocString(szT);

}

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

// 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

BOOL FGetResourceString(long isz, \_\_out\_ecount(cchMax) OLECHAR \*psz, int cchMax);

BSTR BstrGetResourceString(long isz);

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

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

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

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

#pragma once

template <class Fn, bool mapRest>

void MapFormalsImpl(ParseNode \*pnodeFunc, Fn fn)

{

for (ParseNode \*pnode = pnodeFunc->sxFnc.pnodeArgs; pnode != nullptr; pnode = pnode->GetFormalNext())

{

fn(pnode);

}

if (mapRest && pnodeFunc->sxFnc.pnodeRest != nullptr)

{

fn(pnodeFunc->sxFnc.pnodeRest);

}

}

template <class Fn>

void MapFormalsWithoutRest(ParseNode \*pnodeFunc, Fn fn)

{

return MapFormalsImpl<Fn, false>(pnodeFunc, fn);

}

template <class Fn>

void MapFormals(ParseNode \*pnodeFunc, Fn fn)

{

return MapFormalsImpl<Fn, true>(pnodeFunc, fn);

}

template <class Fn>

void MapFormalsFromPattern(ParseNode \*pnodeFunc, Fn fn)

{

for (ParseNode \*pnode = pnodeFunc->sxFnc.pnodeArgs; pnode != nullptr; pnode = pnode->GetFormalNext())

{

if (pnode->nop == knopParamPattern)

{

Parser::MapBindIdentifier(pnode->sxParamPattern.pnode1, fn);

}

}

}

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

// 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"

// strings for builtin names

#define HASH\_NAME(name, hashCS, hashCI) \

const StaticSym g\_ssym\_##name = \

{ \

hashCS, \

sizeof(#name) - 1, \

OLESTR(#name) \

};

#include "objnames.h"

#undef HASH\_NAME

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

// 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

struct StaticSym

{

ulong luHash;

ulong cch;

OLECHAR sz[];

};

// Builtin symbols.

#define HASH\_NAME(name, hashCS, hashCI) extern const StaticSym g\_ssym\_##name;

#include "objnames.h"

#undef HASH\_NAME

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

// 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"

#if PROFILE\_DICTIONARY

#include "DictionaryStats.h"

#endif

const HashTbl::KWD HashTbl::g\_mptkkwd[tkLimKwd] =

{

{ knopNone,0,knopNone,0 },

#define KEYWORD(tk,f,prec2,nop2,prec1,nop1,name) \

{ nop2,kopl##prec2,nop1,kopl##prec1 },

#define TOK\_DCL(tk,prec2,nop2,prec1,nop1) \

{ nop2,kopl##prec2,nop1,kopl##prec1 },

#include "keywords.h"

};

const HashTbl::ReservedWordInfo HashTbl::s\_reservedWordInfo[tkID] =

{

{ nullptr, fidNil },

#define KEYWORD(tk,f,prec2,nop2,prec1,nop1,name) \

{ &g\_ssym\_##name, f },

#include "keywords.h"

};

HashTbl \* HashTbl::Create(uint cidHash, ErrHandler \* perr)

{

HashTbl \* phtbl;

if (nullptr == (phtbl = HeapNewNoThrow(HashTbl,perr)))

return nullptr;

if (!phtbl->Init(cidHash))

{

delete phtbl;

return nullptr;

}

return phtbl;

}

BOOL HashTbl::Init(uint cidHash)

{

// cidHash must be a power of two

Assert(cidHash > 0 && 0 == (cidHash & (cidHash - 1)));

long cb;

/\* Allocate and clear the hash bucket table \*/

m\_luMask = cidHash - 1;

m\_luCount = 0;

// (Bug 1117873 - Windows OS Bugs)

// Prefast: Verify that cidHash \* sizeof(Ident \*) does not cause an integer overflow

// NoReleaseAllocator( ) takes long - so check for LONG\_MAX

// Win8 730594 - Use intsafe function to check for overflow.

uint cbTemp;

if (FAILED(UIntMult(cidHash, sizeof(Ident \*), &cbTemp)) || cbTemp > LONG\_MAX)

return FALSE;

cb = cbTemp;

if (nullptr == (m\_prgpidName = (Ident \*\*)m\_noReleaseAllocator.Alloc(cb)))

return FALSE;

memset(m\_prgpidName, 0, cb);

#if PROFILE\_DICTIONARY

stats = DictionaryStats::Create(typeid(this).name(), cidHash);

#endif

return TRUE;

}

void HashTbl::Grow()

{

// Grow the bucket size by grow factor

// Has the side-effect of inverting the order the pids appear in their respective buckets.

uint cidHash = m\_luMask + 1;

uint n\_cidHash = cidHash \* GrowFactor;

Assert(n\_cidHash > 0 && 0 == (n\_cidHash & (n\_cidHash - 1)));

// Win8 730594 - Use intsafe function to check for overflow.

uint cbTemp;

if (FAILED(UIntMult(n\_cidHash, sizeof(Ident \*), &cbTemp)) || cbTemp > LONG\_MAX)

// It is fine to exit early here, we will just have a potentially densely populated hash table

return;

long cb = cbTemp;

uint n\_luMask = n\_cidHash - 1;

IdentPtr \*n\_prgpidName = (IdentPtr \*)m\_noReleaseAllocator.Alloc(cb);

if (n\_prgpidName == nullptr)

// It is fine to exit early here, we will just have a potentially densely populated hash table

return;

// Clear the array

memset(n\_prgpidName, 0, cb);

// Place each entry its new bucket.

for (uint i = 0; i < cidHash; i++)

{

for (IdentPtr pid = m\_prgpidName[i], next = pid ? pid->m\_pidNext : nullptr; pid; pid = next, next = pid ? pid->m\_pidNext : nullptr)

{

ulong luHash = pid->m\_luHash;

ulong luIndex = luHash & n\_luMask;

pid->m\_pidNext = n\_prgpidName[luIndex];

n\_prgpidName[luIndex] = pid;

}

}

Assert(CountAndVerifyItems(n\_prgpidName, n\_cidHash, n\_luMask) == m\_luCount);

// Update the table fields.

m\_prgpidName = n\_prgpidName;

m\_luMask= n\_luMask;

#if PROFILE\_DICTIONARY

if(stats)

{

int emptyBuckets = 0;

for (uint i = 0; i < n\_cidHash; i++)

{

if(m\_prgpidName[i] == nullptr)

{

emptyBuckets++;

}

}

stats->Resize(n\_cidHash, emptyBuckets);

}

#endif

}

#if DEBUG

uint HashTbl::CountAndVerifyItems(IdentPtr \*buckets, uint bucketCount, uint mask)

{

uint count = 0;

for (uint i = 0; i < bucketCount; i++)

for (IdentPtr pid = buckets[i]; pid; pid = pid->m\_pidNext)

{

Assert((pid->m\_luHash & mask) == i);

count++;

}

return count;

}

#endif

#pragma warning(push)

#pragma warning(disable:4740) // flow in or out of inline asm code suppresses global optimization

tokens Ident::Tk(bool isStrictMode)

{

const tokens token = (tokens)m\_tk;

if (token == tkLim)

{

m\_tk = tkNone;

const ulong luHash = this->m\_luHash;

const LPCOLESTR prgch = Psz();

const ulong cch = Cch();

#include "kwds\_sw.h"

#define KEYWORD(tk,f,prec2,nop2,prec1,nop1,name) \

LEqual\_##name: \

if (cch == g\_ssym\_##name.cch && \

0 == memcmp(g\_ssym\_##name.sz, prgch, cch \* sizeof(OLECHAR))) \

{ \

if (f) \

this->m\_grfid |= f; \

this->m\_tk = tk; \

return ((f & fidKwdRsvd) || (isStrictMode && (f & fidKwdFutRsvd))) ? tk : tkID; \

} \

goto LDefault;

#include "keywords.h"

LDefault:

return tkID;

}

else if (token == tkNone || !(m\_grfid & fidKwdRsvd))

{

if ( !isStrictMode || !(m\_grfid & fidKwdFutRsvd))

{

return tkID;

}

}

return token;

}

#pragma warning(pop)

void Ident::SetTk(tokens token, ushort grfid)

{

Assert(token != tkNone && token < tkID);

if (m\_tk == tkLim)

{

m\_tk = (ushort)token;

m\_grfid |= grfid;

}

else

{

Assert(m\_tk == token);

Assert((m\_grfid & grfid) == grfid);

}

}

IdentPtr HashTbl::PidFromTk(tokens token)

{

Assert(token > tkNone && token < tkID);

\_\_analysis\_assume(token > tkNone && token < tkID);

// Create a pid so we can create a name node

IdentPtr rpid = m\_rpid[token];

if (nullptr == rpid)

{

StaticSym const \* sym = s\_reservedWordInfo[token].sym;

Assert(sym != nullptr);

rpid = this->PidHashNameLenWithHash(sym->sz, sym->cch, sym->luHash);

rpid->SetTk(token, s\_reservedWordInfo[token].grfid);

m\_rpid[token] = rpid;

}

return rpid;

}

template <typename CharType>

IdentPtr HashTbl::PidHashNameLen(CharType const \* prgch, ulong cch)

{

// NOTE: We use case sensitive hash during compilation, but the runtime

// uses case insensitive hashing so it can do case insensitive lookups.

ulong luHash = CaseSensitiveComputeHashCch(prgch, cch);

return PidHashNameLenWithHash(prgch, cch, luHash);

};

template IdentPtr HashTbl::PidHashNameLen<utf8char\_t>(utf8char\_t const \* prgch, ulong cch);

template IdentPtr HashTbl::PidHashNameLen<char>(char const \* prgch, ulong cch);

template IdentPtr HashTbl::PidHashNameLen<wchar\_t>(wchar\_t const \* prgch, ulong cch);

template <typename CharType>

IdentPtr HashTbl::PidHashNameLenWithHash(\_In\_reads\_(cch) CharType const \* prgch, long cch, ulong luHash)

{

Assert(cch >= 0);

AssertArrMemR(prgch, cch);

Assert(luHash == CaseSensitiveComputeHashCch(prgch, cch));

IdentPtr \* ppid;

IdentPtr pid;

long cb;

long bucketCount;

#if PROFILE\_DICTIONARY

int depth = 0;

#endif

pid = this->FindExistingPid(prgch, cch, luHash, &ppid, &bucketCount

#if PROFILE\_DICTIONARY

, depth

#endif

);

if (pid)

{

return pid;

}

if (bucketCount > BucketLengthLimit && m\_luCount > m\_luMask)

{

Grow();

// ppid is now invalid because the Grow() moves the entries around.

// Find the correct ppid by repeating the find of the end of the bucket

// the new item will be placed in.

// Note this is similar to the main find loop but does not count nor does it

// look at the entries because we already proved above the entry is not in the

// table, we just want to find the end of the bucket.

ppid = &m\_prgpidName[luHash & m\_luMask];

while (\*ppid)

ppid = &(\*ppid)->m\_pidNext;

}

#if PROFILE\_DICTIONARY

++depth;

if (stats)

stats->Insert(depth);

#endif

//Windows OS Bug 1795286 : CENTRAL PREFAST RUN: inetcore\scriptengines\src\src\core\hash.cpp :

// 'sizeof((\*pid))+((cch+1))\*sizeof(OLECHAR)' may be smaller than

// '((cch+1))\*sizeof(OLECHAR)'. This can be caused by integer overflows

// or underflows. This could yield an incorrect buffer all

/\* Allocate space for the identifier \*/

ULONG Len;

if (FAILED(ULongAdd(cch, 1, &Len)) ||

FAILED(ULongMult(Len, sizeof(OLECHAR), &Len)) ||

FAILED(ULongAdd(Len, sizeof(\*pid), &Len)) ||

FAILED(ULongToLong(Len, &cb)))

{

cb = 0;

m\_perr->Throw(ERRnoMemory);

}

if (nullptr == (pid = (IdentPtr)m\_noReleaseAllocator.Alloc(cb)))

m\_perr->Throw(ERRnoMemory);

/\* Insert the identifier into the hash list \*/

\*ppid = pid;

// Increment the number of entries in the table.

m\_luCount++;

/\* Fill in the identifier record \*/

pid->m\_pidNext = nullptr;

pid->m\_tk = tkLim;

pid->m\_grfid = fidNil;

pid->m\_luHash = luHash;

pid->m\_cch = cch;

pid->m\_pidRefStack = nullptr;

pid->m\_propertyId = Js::Constants::NoProperty;

pid->assignmentState = NotAssigned;

HashTbl::CopyString(pid->m\_sz, prgch, cch);

return pid;

}

template <typename CharType>

IdentPtr HashTbl::FindExistingPid(

CharType const \* prgch,

long cch,

ulong luHash,

IdentPtr \*\*pppInsert,

long \*pBucketCount

#if PROFILE\_DICTIONARY

, int& depth

#endif

)

{

long bucketCount;

IdentPtr pid;

IdentPtr \*ppid = &m\_prgpidName[luHash & m\_luMask];

/\* Search the hash table for an existing match \*/

ppid = &m\_prgpidName[luHash & m\_luMask];

for (bucketCount = 0; nullptr != (pid = \*ppid); ppid = &pid->m\_pidNext, bucketCount++)

{

if (pid->m\_luHash == luHash && (int)pid->m\_cch == cch &&

HashTbl::CharsAreEqual(pid->m\_sz, prgch, cch))

{

return pid;

}

#if PROFILE\_DICTIONARY

++depth;

#endif

}

if (pBucketCount)

{

\*pBucketCount = bucketCount;

}

if (pppInsert)

{

\*pppInsert = ppid;

}

return nullptr;

}

template IdentPtr HashTbl::FindExistingPid<utf8char\_t>(

utf8char\_t const \* prgch, long cch, ulong luHash, IdentPtr \*\*pppInsert, long \*pBucketCount

#if PROFILE\_DICTIONARY

, int& depth

#endif

);

template IdentPtr HashTbl::FindExistingPid<char>(

char const \* prgch, long cch, ulong luHash, IdentPtr \*\*pppInsert, long \*pBucketCount

#if PROFILE\_DICTIONARY

, int& depth

#endif

);

template IdentPtr HashTbl::FindExistingPid<wchar\_t>(

wchar\_t const \* prgch, long cch, ulong luHash, IdentPtr \*\*pppInsert, long \*pBucketCount

#if PROFILE\_DICTIONARY

, int& depth

#endif

);

bool HashTbl::Contains(\_In\_reads\_(cch) LPCOLESTR prgch, long cch)

{

ulong luHash = CaseSensitiveComputeHashCch(prgch, cch);

for (auto pid = m\_prgpidName[luHash & m\_luMask]; pid; pid = pid->m\_pidNext)

{

if (pid->m\_luHash == luHash && (int)pid->m\_cch == cch &&

HashTbl::CharsAreEqual(pid->m\_sz, prgch, cch))

{

return true;

}

}

return false;

}

#include "hashfunc.cpp"

#pragma warning(push)

#pragma warning(disable:4740) // flow in or out of inline asm code suppresses global optimization

// Decide if token is keyword by string matching -

// This method is used during colorizing when scanner isn't interested in storing the actual id and does not care about conversion of escape sequences

tokens HashTbl::TkFromNameLenColor(\_In\_reads\_(cch) LPCOLESTR prgch, ulong cch)

{

ulong luHash = CaseSensitiveComputeHashCch(prgch, cch);

// look for a keyword

#include "kwds\_sw.h"

#define KEYWORD(tk,f,prec2,nop2,prec1,nop1,name) \

LEqual\_##name: \

if (cch == g\_ssym\_##name.cch && \

0 == memcmp(g\_ssym\_##name.sz, prgch, cch \* sizeof(OLECHAR))) \

{ \

return tk; \

} \

goto LDefault;

#include "keywords.h"

LDefault:

return tkID;

}

#pragma warning(pop)

#pragma warning(push)

#pragma warning(disable:4740) // flow in or out of inline asm code suppresses global optimization

// Decide if token is keyword by string matching -

// This method is used during colorizing when scanner isn't interested in storing the actual id and does not care about conversion of escape sequences

tokens HashTbl::TkFromNameLen(\_In\_reads\_(cch) LPCOLESTR prgch, ulong cch, bool isStrictMode)

{

ulong luHash = CaseSensitiveComputeHashCch(prgch, cch);

// look for a keyword

#include "kwds\_sw.h"

#define KEYWORD(tk,f,prec2,nop2,prec1,nop1,name) \

LEqual\_##name: \

if (cch == g\_ssym\_##name.cch && \

0 == memcmp(g\_ssym\_##name.sz, prgch, cch \* sizeof(OLECHAR))) \

{ \

return ((f & fidKwdRsvd) || (isStrictMode && (f & fidKwdFutRsvd))) ? tk : tkID; \

} \

goto LDefault;

#include "keywords.h"

LDefault:

return tkID;

}

#pragma warning(pop)

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

// 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

struct StaticSym;

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

Hashing functions. Definitions in core\hashfunc.cpp.

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

ULONG CaseSensitiveComputeHashCch(LPCOLESTR prgch, long cch);

ULONG CaseSensitiveComputeHashCch(LPCUTF8 prgch, long cch);

ULONG CaseInsensitiveComputeHash(LPCOLESTR posz);

enum

{

fidNil = 0x0000,

fidKwdRsvd = 0x0001, // the keyword is a reserved word

fidKwdFutRsvd = 0x0002, // a future reserved word, but only in strict mode

// Flags to identify tracked aliases of "eval"

fidEval = 0x0008,

// Flags to identify tracked aliases of "let"

fidLetOrConst = 0x0010, // ID has previously been used in a block-scoped declaration

// This flag is used by the Parser CountDcls and FillDcls methods.

// CountDcls sets the bit as it walks through the var decls so that

// it can skip duplicates. FillDcls clears the bit as it walks through

// again to skip duplicates.

fidGlobalDcl = 0x2000,

fidUsed = 0x4000 // name referenced by source code

};

struct BlockIdsStack

{

int id;

BlockIdsStack \*prev;

};

class Span

{

charcount\_t m\_ichMin;

charcount\_t m\_ichLim;

public:

Span(): m\_ichMin((charcount\_t)-1), m\_ichLim((charcount\_t)-1) { }

Span(charcount\_t ichMin, charcount\_t ichLim): m\_ichMin(ichMin), m\_ichLim(ichLim) { }

charcount\_t GetIchMin() { return m\_ichMin; }

charcount\_t GetIchLim() { Assert(m\_ichMin != (charcount\_t)-1); return m\_ichLim; }

void Set(charcount\_t ichMin, charcount\_t ichLim)

{

m\_ichMin = ichMin;

m\_ichLim = ichLim;

}

operator bool() { return m\_ichMin != -1; }

};

struct PidRefStack

{

PidRefStack() : isAsg(false), isDynamic(false), id(0), span(), sym(nullptr), prev(nullptr) {}

PidRefStack(int id) : isAsg(false), isDynamic(false), id(id), span(), sym(nullptr), prev(nullptr) {}

charcount\_t GetIchMin() { return span.GetIchMin(); }

charcount\_t GetIchLim() { return span.GetIchLim(); }

int GetScopeId() const { return id; }

Symbol \*GetSym() const { return sym; }

void SetSym(Symbol \*sym) { this->sym = sym; }

bool IsAssignment() const { return isAsg; }

bool IsDynamicBinding() const { return isDynamic; }

void SetDynamicBinding() { isDynamic = true; }

void TrackAssignment(charcount\_t ichMin, charcount\_t ichLim);

Symbol \*\*GetSymRef()

{

return &sym;

}

bool isAsg;

bool isDynamic;

int id;

Span span;

Symbol \*sym;

PidRefStack \*prev;

};

enum AssignmentState : byte {

NotAssigned,

AssignedOnce,

AssignedMultipleTimes

};

struct Ident

{

friend class HashTbl;

private:

Ident \* m\_pidNext; // next identifier in this hash bucket

PidRefStack \*m\_pidRefStack;

ushort m\_tk; // token# if identifier is a keyword

ushort m\_grfid; // see fidXXX above

ulong m\_luHash; // hash value

ulong m\_cch; // length of the identifier spelling

Js::PropertyId m\_propertyId;

AssignmentState assignmentState;

OLECHAR m\_sz[]; // the spelling follows (null terminated)

void SetTk(tokens tk, ushort grfid);

public:

LPCOLESTR Psz(void)

{ return m\_sz; }

ulong Cch(void)

{ return m\_cch; }

tokens Tk(bool isStrictMode);

ulong Hash(void)

{ return m\_luHash; }

PidRefStack \*GetTopRef() const

{

return m\_pidRefStack;

}

void SetTopRef(PidRefStack \*ref)

{

m\_pidRefStack = ref;

}

void PromoteAssignmentState()

{

if (assignmentState == NotAssigned)

{

assignmentState = AssignedOnce;

}

else if (assignmentState == AssignedOnce)

{

assignmentState = AssignedMultipleTimes;

}

}

bool IsSingleAssignment()

{

return assignmentState == AssignedOnce;

}

PidRefStack \*GetPidRefForScopeId(int scopeId)

{

PidRefStack \*ref;

for (ref = m\_pidRefStack; ref; ref = ref->prev)

{

int refId = ref->GetScopeId();

if (refId == scopeId)

{

return ref;

}

if (refId < scopeId)

{

break;

}

}

return nullptr;

}

charcount\_t GetTopIchMin() const

{

Assert(m\_pidRefStack);

return m\_pidRefStack->GetIchMin();

}

charcount\_t GetTopIchLim() const

{

Assert(m\_pidRefStack);

return m\_pidRefStack->GetIchLim();

}

void PushPidRef(int blockId, PidRefStack \*newRef)

{

AssertMsg(blockId >= 0, "Block Id's should be greater than 0");

newRef->id = blockId;

newRef->prev = m\_pidRefStack;

m\_pidRefStack = newRef;

}

PidRefStack \* RemovePrevPidRef(PidRefStack \*ref)

{

PidRefStack \*prevRef;

if (ref == nullptr)

{

prevRef = m\_pidRefStack;

Assert(prevRef);

m\_pidRefStack = prevRef->prev;

}

else

{

prevRef = ref->prev;

Assert(prevRef);

ref->prev = prevRef->prev;

}

return prevRef;

}

PidRefStack \* FindOrAddPidRef(ArenaAllocator \*alloc, int scopeId, int maxScopeId = -1)

{

// If we were supplied with a maxScopeId, then we potentially need to look one more

// scope level out. This can happen if we have a declaration in function scope shadowing

// a parameter scope declaration. In this case we'd need to look beyond the body scope (scopeId)

// to the outer parameterScope (maxScopeId).

if (maxScopeId == -1)

{

maxScopeId = scopeId;

}

// If the stack is empty, or we are pushing to the innermost scope already,

// we can go ahead and push a new PidRef on the stack.

if (m\_pidRefStack == nullptr || m\_pidRefStack->id < maxScopeId)

{

PidRefStack \*newRef = Anew(alloc, PidRefStack, scopeId);

if (newRef == nullptr)

{

return nullptr;

}

newRef->prev = m\_pidRefStack;

m\_pidRefStack = newRef;

return newRef;

}

// Search for the corresponding PidRef, or the position to insert the new PidRef.

PidRefStack \*ref = m\_pidRefStack;

while (1)

{

// We may already have a ref for this scopeId.

if (ref->id == scopeId)

{

return ref;

}

if (ref->id == maxScopeId

// If we match the different maxScopeId, then this match is sufficent if it is a decl.

// This is because the parameter scope decl would have been created before this point.

&& ref->sym != nullptr)

{

return ref;

}

if (ref->prev == nullptr || ref->prev->id < maxScopeId)

{

// No existing PidRef for this scopeId, so create and insert one at this position.

PidRefStack \*newRef = Anew(alloc, PidRefStack, scopeId);

if (newRef == nullptr)

{

return nullptr;

}

if (ref->id < scopeId)

{

// Without parameter scope, we would have just pushed the ref instead of inserting.

// We effectively had a false positive match (a parameter scope ref with no sym)

// so we need to push the Pid rather than inserting.

newRef->prev = m\_pidRefStack;

m\_pidRefStack = newRef;

}

else

{

newRef->prev = ref->prev;

ref->prev = newRef;

}

return newRef;

}

Assert(ref->prev->id <= ref->id);

ref = ref->prev;

}

}

Js::PropertyId GetPropertyId() const { return m\_propertyId; }

void SetPropertyId(Js::PropertyId id) { m\_propertyId = id; }

void SetIsEval() { m\_grfid |= fidEval; }

BOOL GetIsEval() const { return m\_grfid & fidEval; }

void SetIsLetOrConst() { m\_grfid |= fidLetOrConst; }

BOOL GetIsLetOrConst() const { return m\_grfid & fidLetOrConst; }

};

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

class HashTbl

{

public:

static HashTbl \* Create(uint cidHash, ErrHandler \* perr);

void Release(void)

{

delete this;

}

BOOL TokIsBinop(tokens tk, int \*popl, OpCode \*pnop)

{

const KWD \*pkwd = KwdOfTok(tk);

if (nullptr == pkwd)

return FALSE;

\*popl = pkwd->prec2;

\*pnop = pkwd->nop2;

return TRUE;

}

BOOL TokIsUnop(tokens tk, int \*popl, OpCode \*pnop)

{

const KWD \*pkwd = KwdOfTok(tk);

if (nullptr == pkwd)

return FALSE;

\*popl = pkwd->prec1;

\*pnop = pkwd->nop1;

return TRUE;

}

IdentPtr PidFromTk(tokens tk);

IdentPtr PidHashName(LPCOLESTR psz)

{

size\_t csz = wcslen(psz);

Assert(csz <= ULONG\_MAX);

return PidHashNameLen(psz, static\_cast<ulong>(csz));

}

template <typename CharType>

IdentPtr PidHashNameLen(CharType const \* psz, ulong cch);

template <typename CharType>

IdentPtr PidHashNameLenWithHash(\_In\_reads\_(cch) CharType const \* psz, long cch, ulong luHash);

template <typename CharType>

\_\_inline IdentPtr FindExistingPid(

CharType const \* prgch,

long cch,

ulong luHash,

IdentPtr \*\*pppInsert,

long \*pBucketCount

#if PROFILE\_DICTIONARY

, int& depth

#endif

);

tokens TkFromNameLen(\_In\_reads\_(cch) LPCOLESTR prgch, ulong cch, bool isStrictMode);

tokens TkFromNameLenColor(\_In\_reads\_(cch) LPCOLESTR prgch, ulong cch);

NoReleaseAllocator\* GetAllocator() {return &m\_noReleaseAllocator;}

bool Contains(\_In\_reads\_(cch) LPCOLESTR prgch, long cch);

private:

NoReleaseAllocator m\_noReleaseAllocator; // to allocate identifiers

Ident \*\* m\_prgpidName; // hash table for names

ulong m\_luMask; // hash mask

ulong m\_luCount; // count of the number of entires in the hash table

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

IdentPtr m\_rpid[tkLimKwd];

HashTbl(ErrHandler \* perr)

{

m\_prgpidName = nullptr;

m\_perr = perr;

memset(&m\_rpid, 0, sizeof(m\_rpid));

}

~HashTbl(void) {}

// Called to grow the number of buckets in the table to reduce the table density.

void Grow();

// Automatically grow the table if a bucket's length grows beyond BucketLengthLimit and the table is densely populated.

static const uint BucketLengthLimit = 5;

// When growing the bucket size we'll grow by GrowFactor. GrowFactor MUST be a power of 2.

static const uint GrowFactor = 4;

#if DEBUG

uint CountAndVerifyItems(IdentPtr \*buckets, uint bucketCount, uint mask);

#endif

static bool CharsAreEqual(\_\_in\_z LPCOLESTR psz1, \_\_in\_ecount(cch2) LPCOLESTR psz2, long cch2)

{

return memcmp(psz1, psz2, cch2 \* sizeof(OLECHAR)) == 0;

}

static bool CharsAreEqual(\_\_in\_z LPCOLESTR psz1, LPCUTF8 psz2, long cch2)

{

return utf8::CharsAreEqual(psz1, psz2, cch2, utf8::doAllowThreeByteSurrogates);

}

static bool CharsAreEqual(\_\_in\_z LPCOLESTR psz1, \_\_in\_ecount(cch2) char const \* psz2, long cch2)

{

while (cch2-- > 0)

{

if (\*psz1++ != \*psz2++)

return false;

}

return true;

}

static void CopyString(\_\_in\_ecount(cch + 1) LPOLESTR psz1, \_\_in\_ecount(cch) LPCOLESTR psz2, long cch)

{

js\_memcpy\_s(psz1, cch \* sizeof(OLECHAR), psz2, cch \* sizeof(OLECHAR));

psz1[cch] = 0;

}

static void CopyString(\_\_in\_ecount(cch + 1) LPOLESTR psz1, LPCUTF8 psz2, long cch)

{

utf8::DecodeIntoAndNullTerminate(psz1, psz2, cch);

}

static void CopyString(\_\_in\_ecount(cch + 1) LPOLESTR psz1, \_\_in\_ecount(cch) char const \* psz2, long cch)

{

while (cch-- > 0)

\*(psz1++) = \*psz2++;

\*psz1 = 0;

}

// note: on failure this may throw or return FALSE, depending on

// where the failure occurred.

BOOL Init(uint cidHash);

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

/\* The following members are related to the keyword descriptor tables \*/

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

struct KWD

{

OpCode nop2;

byte prec2;

OpCode nop1;

byte prec1;

};

struct ReservedWordInfo

{

StaticSym const \* sym;

ushort grfid;

};

static const ReservedWordInfo s\_reservedWordInfo[tkID];

static const KWD g\_mptkkwd[tkLimKwd];

static const KWD \* KwdOfTok(tokens tk)

{ return (unsigned int)tk < tkLimKwd ? g\_mptkkwd + tk : nullptr; }

#if PROFILE\_DICTIONARY

DictionaryStats \*stats;

#endif

};

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

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

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

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

/\*

\* IMPORTANT:

\* This file does not compile stand alone. It was required so that

\* the same code could be built into a utility program comphash.exe as well

\* as the scripting dll's. This file is included in core\comphash.cpp

\* to be used by comphash.exe. It is included in core\scrutil.cpp where to

\* be used by jscript.dll and vbscript.dll.

\*

\* comphash.exe is a utility used in the build to generate a source code file

\* containing a table of hash values associated with strings needed by

\* jscript and vbscript. It is highly desirable to have a single definition

\* of the hash function so things don't go out of sync.

\*/

// scaffolding - define ULONG

typedef unsigned long ULONG;

ULONG CaseSensitiveComputeHashCch(LPCOLESTR prgch, long cch)

{

ULONG luHash = 0;

while (cch-- > 0)

luHash = 17 \* luHash + \*(wchar\_t \*)prgch++;

return luHash;

}

ULONG CaseSensitiveComputeHashCch(LPCUTF8 prgch, long cch)

{

utf8::DecodeOptions options = utf8::doAllowThreeByteSurrogates;

ULONG luHash = 0;

while (cch-- > 0)

luHash = 17 \* luHash + utf8::Decode(prgch, prgch + 4, options); // WARNING: Assume cch correct, suppress end-of-buffer checking

return luHash;

}

ULONG CaseSensitiveComputeHashCch(char const \* prgch, long cch)

{

ULONG luHash = 0;

while (cch-- > 0)

{

Assert(utf8::IsStartByte(\*prgch) && !utf8::IsLeadByte(\*prgch));

luHash = 17 \* luHash + \*prgch++;

}

return luHash;

}

ULONG CaseInsensitiveComputeHash(LPCOLESTR posz)

{

ULONG luHash = 0;

wchar\_t ch;

while (0 != (ch = \*(wchar\_t \*)posz++))

{

if (ch <= 'Z' && ch >= 'A')

ch += 'a' - 'A';

luHash = 17 \* luHash + ch;

}

return luHash;

}

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

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

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

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

#ifndef \_\_IDIOM\_H\_\_

#define \_\_IDIOM\_H\_\_

// cleanup if needed, and set to (null)

#ifndef DELETEARR

#define DELETEARR(arr) do {if (arr){ delete [] (arr); (arr) = NULL; }} while (0)

#endif

#ifndef DELETEPTR

#define DELETEPTR(p) do {if (p){ delete (p); (p) = NULL; }} while (0)

#endif

#ifndef FREEPTR

#define FREEPTR(p) do {if (p){ free(p); (p) = NULL; }} while (0)

#endif

#ifndef SYSFREE

#define SYSFREE(p) do {if (p){ ::SysFreeString(p); (p) = NULL; }} while (0)

#endif

#ifndef RELEASEPTR

#define RELEASEPTR(p) do {if (p){ (p)->Release(); (p) = NULL; }} while (0)

#endif

#ifndef UNADVISERELEASE

#define UNADVISERELEASE(p, dwCookie) do {if (p){ (p)->Unadvise(dwCookie); (p)->Release(); (p) = NULL; }} while (0)

#endif

#ifndef RELEASETYPEINFOATTR

#define RELEASETYPEINFOATTR(pinfo, pattr) do { if (NULL != (pinfo)) { if (NULL != (pattr)) { (pinfo)->ReleaseTypeAttr(pattr); (pattr) = NULL; } (pinfo)->Release(); (pinfo) = NULL; } } while (0)

#endif

#ifndef REGCLOSE

#define REGCLOSE(hkey) do {if (NULL != (hkey)){ RegCloseKey(hkey); (hkey) = NULL; }} while (0)

#endif

#ifndef CLOSEPTR

#define CLOSEPTR(p) do {if (NULL != (p)) { (p)->Close(); (p) = 0; }} while (0)

#endif

// check result, cleanup if failed

#ifndef IFNULLMEMGOLABEL

#define IFNULLMEMGOLABEL(p, label) do {if (NULL == (p)){ hr = E\_OUTOFMEMORY; goto label; }} while (0)

#endif

#ifndef IFNULLMEMGO

#define IFNULLMEMGO(p) IFNULLMEMGOLABEL(p, LReturn)

#endif

#ifndef IFNULLMEMRET

#define IFNULLMEMRET(p) do {if (!(p)) return E\_OUTOFMEMORY; } while (0)

#endif

#ifndef IFFAILGOLABEL

#define IFFAILGOLABEL(expr, label) do {if (FAILED(hr = (expr))) goto label; } while (0)

#endif

#ifndef IFFAILGO

#define IFFAILGO(expr) IFFAILGOLABEL(expr, LReturn)

#endif

// If (expr) failed, go to LReturn with (code)

#ifndef IFFAILGORET

#define IFFAILGORET(expr, code) do {if (FAILED(hr = (expr))) { hr = (code); goto LReturn; }} while (0)

#endif

#ifndef FAILGO

#define FAILGO(hresult) do { hr = (hresult); goto LReturn; } while (0)

#endif

#ifndef IFFAILWINERRGO

#define IFFAILWINERRGO(expr) do { if (FAILED(hr = HRESULT\_FROM\_WIN32(expr))) goto LReturn; } while (0)

#endif

#ifndef FAILWINERRGO

#define FAILWINERRGO(expr) do { hr = HRESULT\_FROM\_WIN32(expr); goto LReturn; } while (0)

#endif

#ifndef IFFAILRET

#define IFFAILRET(expr) do {if (FAILED(hr = (expr))) return hr; } while (0)

#endif

#ifndef IFFAILLEAVE

#define IFFAILLEAVE(expr) do {if (FAILED(hr = (expr))) \_\_leave; } while (0)

#endif

#ifndef FAILLEAVE

#define FAILLEAVE(expr) do { hr = (expr); \_\_leave; } while (0)

#endif

// set optional return value

#ifndef SETRETVAL

#define SETRETVAL(ptr, val) do { if (ptr) \*(ptr) = (val); } while (0)

#endif

#ifndef CHECK\_POINTER

#define CHECK\_POINTER(p) do { if (NULL == (p)) return E\_POINTER; } while (0)

#endif

#ifndef EXPECT\_POINTER

#define EXPECT\_POINTER(p) do { if (NULL == (p)) return E\_UNEXPECTED; } while (0)

#endif

#ifndef ARG\_POINTER

#define ARG\_POINTER(p) do { if (NULL == (p)) return E\_INVALIDARG; } while (0)

#endif

#endif \_\_IDIOM\_H\_\_

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

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

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

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

/////////////////////////////////////////////////////////////////////////////

//

// String Table

//

STRINGTABLE DISCARDABLE

BEGIN

#include <rterrors\_limits.h>

#define RT\_ERROR\_MSG(name, errnum, str1, str2, jst, errorNumSource) errnum str2

#define RT\_PUBLICERROR\_MSG(name, errnum, str1, str2, jst, errorNumSource) errnum+RTERROR\_PUBLIC\_RESOURCEOFFSET str2

#include <rterrors.h>

#undef RT\_PUBLICERROR\_MSG

#undef RT\_ERROR\_MSG

#define RT\_ERROR\_MSG(name, errnum, str1, str2, jst, errorNumSource) errnum+RTERROR\_STRINGFORMAT\_OFFSET str1

#define RT\_PUBLICERROR\_MSG(name, errnum, str1, str2, jst, errorNumSource) errnum+RTERROR\_STRINGFORMAT\_OFFSET+RTERROR\_PUBLIC\_RESOURCEOFFSET str1

#include <rterrors.h>

#undef RT\_PUBLICERROR\_MSG

#undef RT\_ERROR\_MSG

#define LSC\_ERROR\_MSG(errnum, name, str) errnum str

#include <perrors.h>

#undef LSC\_ERROR\_MSG

IDS\_COMPILATION\_ERROR\_SOURCE "JavaScript compilation error"

IDS\_RUNTIME\_ERROR\_SOURCE "JavaScript runtime error"

IDS\_UNKNOWN\_RUNTIME\_ERROR "Unknown runtime error"

IDS\_INFINITY "Infinity"

IDS\_MINUSINFINITY "-Infinity"

END

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

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

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

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

// WARNING: See Bug 1335253

// This file will not generate Trie branches when symbols share prefixes under certain circumstances (thus skipping cases).

objArgs = WScript.Arguments;

function emitToken(token, d, indent) {

r = "";

indent += " ";

if (d)

r += indent + "p += " + d + ";\r\n";

if (token.res == 1) {

if (token.tk === "tkYIELD") {

r += indent + "if (this->m\_fYieldIsKeyword || !this->m\_parser || this->m\_parser->IsStrictMode()) {" + "\r\n";

r += indent + " token = " + token.tk + ";\r\n";

r += indent + " goto LReserved;\r\n";

r += indent + "}\r\n";

r += indent + "goto LIdentifier;\r\n";

} else if (token.tk === "tkAWAIT") {

r += indent + "if (this->m\_fAwaitIsKeyword || !this->m\_parser || this->m\_parser->IsStrictMode()) {" + "\r\n";

r += indent + " token = " + token.tk + ";\r\n";

r += indent + " goto LReserved;\r\n";

r += indent + "}\r\n";

r += indent + "goto LIdentifier;\r\n";

} else {

r += indent + "token = " + token.tk + ";\r\n";

r += indent + "goto LReserved;\r\n";

}

} else if (token.res == 2) {

r += indent + "if (!this->m\_parser || this->m\_parser->IsStrictMode()) {" + "\r\n";

r += indent + " " + "token = " + token.tk + ";\r\n";

r += indent + " " + "goto LReserved;\r\n";

r += indent + "}\r\n";

r += indent + "goto LIdentifier;\r\n";

} else if (token.res == 3) {

// These are special case of identifiers that we have a well known PID for and always want to be filled

// whether or not we have suppressed generated pids. (e.g. eval and arguments)

r += indent + "goto " + token.tk + ";\r\n";

} else {

WScript.Echo("Error: Unsupported Keyword type");

}

return r;

}

function noMoreBranches(token) {

for (var c = token; c.length; c = c[0]) {

if (c.length > 1) return false;

}

return true;

}

function emit(token, d, indent) {

var r = "";

if (token.length > 1) {

r += indent + "switch (";

if (d < 0) r += "ch";

else r += "p[" + d + "]";

r += ") {\r\n";

for (var i = 0; i < token.length; i++) {

var tk = token[i];

r += indent + "case '" + tk.char + "':\r\n";

r += emit(tk, d + 1, indent + " ");

if (tk.tk && tk.length) {

r += indent + " if (!IsIdContinueNext(p+" +(d + 1) + ",last)) {\r\n" + emitToken(tk, d + 1, indent + " ") + indent + " }\r\n";

}

r += indent + " break;\r\n";

}

r += indent + "}\r\n";

}

else if (noMoreBranches(token)) {

r += indent + "if (";

for (var c = token; c.length; c = c[0]) {

r += "p[" + d++ + "] == '" + c[0].char + "' && ";

}

r += "!IsIdContinueNext(p+" + d + ", last)) {\r\n";

r += emitToken(c, d, indent);

r += indent + "}\r\n";

}

else {

r += indent + "if (p[" + d + "] == '" + token[0].char + "') {\r\n";

r += emit(token[0], d + 1, indent + " ");

r += indent + "}\r\n";

}

return r;

}

if (objArgs.length != 1 && objArgs.length != 2) {

WScript.Echo("Supply the header file name and optional output file");

}

else {

var fso = new ActiveXObject("Scripting.FileSystemObject");

var file = fso.OpenTextFile(objArgs(0), 1);

var text = file.ReadAll();

file.Close();

var reg = /KEYWORD\((tk[A-Z]+)\s\*,([1-2]),.\*,\s\*([a-z]+)\)/g;

var s\_reg = /S\_KEYWORD\((L[A-Z][a-z]+)\s\*,(3),\s\*([a-z]+)\)/g;

var t = [];

var s = text.replace(reg, function (a, p1, p2, p3, offset) {

t.push({ tk: p1, res: p2, word: p3 });

});

var s\_s = text.replace(s\_reg, function (a, p1, p2, p3, offset) {

t.push({ tk: p1, res: p2, word: p3 });

});

var tokens = [];

var counter = 0;

for (var i = 0; i < t.length; i++) {

var token = t[i];

var current = tokens;

for (var j = 0; j < token.word.length; j++) {

l = token.word.substring(j, j + 1);

var n = current[l];

if (n)

current = n;

else {

var nt = [];

nt.char = l;

current[l] = nt;

current.push(nt);

current = nt;

}

counter++;

}

current.tk = token.tk;

current.res = token.res;

}

var indent = " ";

var r = "";

r += "//-------------------------------------------------------------------------------------------------------\r\n";

r += "// Copyright (C) Microsoft. All rights reserved.\r\n";

r += "// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.\r\n";

r += "//-------------------------------------------------------------------------------------------------------\r\n";

r += "// GENERATED FILE, DO NOT HAND-MODIFY!\r\n";

r += "// Generated with the following command line: wscript jsscan.js " + objArgs(0) + " " + objArgs(1) + "\r\n";

r += "// This should be regenerated whenever the keywords change.\r\n";

r += "\r\n";

// Generate the reserved word recognizer

for (var i = 0; i < tokens.length; i++) {

var tk = tokens[i];

r += indent + "case '" + tk.char + "':\r\n";

r += indent + " if (identifyKwds)\r\n";

r += indent + " {\r\n";

var simple = tk.length == 1 && noMoreBranches(tk);

r += emit(tk, 0, indent + " ");

r += indent + " }\r\n";

r += indent + " goto LIdentifier;\r\n";

}

r += "\r\n";

// Generate lower case letters that are not part of the recognizer

r += indent + "// characters not in a reserved word\r\n";

var c = 0;

var chars = "abcdefghijklmnopqrstuvwxyz";

for (var i = 0; i < chars.length; i++) {

if (c == 0) r += indent;

var ch = chars.substring(i, i + 1);

if (!tokens[ch])

r += "case '" + ch + "': ";

else

r += " ";

if (++c == 5) {

c = 0;

r += "\r\n";

}

}

r += "\r\n";

r += indent + " goto LIdentifier;\r\n";

if (objArgs.length == 2) {

var outfile = fso.CreateTextFile(objArgs(1), true);

outfile.Write(r);

outfile.Close();

} else {

WScript.Echo(r);

}

}

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

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

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

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

#include "kwd-lsc.h"

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

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

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

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

#ifndef KEYWORD

#define KEYWORD(tk,f,prec2,nop2,prec1,nop1,name)

#endif //!KEYWORD

#ifndef S\_KEYWORD

#define S\_KEYWORD(name,f,lab)

#endif //!S\_KEYWORD

// token reserved word? (see fidXXX values in enum, in hash.h)

// binary operator precedence

// binary operator

// unary operator precedence

// unary operator

// name

KEYWORD(tkABSTRACT ,0, No, knopNone , No, knopNone , abstract)

KEYWORD(tkASSERT ,0, No, knopNone , No, knopNone , assert)

KEYWORD(tkAWAIT ,1, No, knopNone ,Uni, knopAwait , await)

KEYWORD(tkBOOLEAN ,0, No, knopNone , No, knopNone , boolean)

KEYWORD(tkBREAK ,1, No, knopNone , No, knopNone , break)

KEYWORD(tkBYTE ,0, No, knopNone , No, knopNone , byte)

KEYWORD(tkCASE ,1, No, knopNone , No, knopNone , case)

KEYWORD(tkCATCH ,1, No, knopNone , No, knopNone , catch)

KEYWORD(tkCHAR ,0, No, knopNone , No, knopNone , char)

KEYWORD(tkCONTINUE ,1, No, knopNone , No, knopNone , continue)

KEYWORD(tkDEBUGGER ,1, No, knopNone , No, knopNone , debugger)

KEYWORD(tkDECIMAL ,0, No, knopNone , No, knopNone , decimal)

KEYWORD(tkDEFAULT ,1, No, knopNone , No, knopNone , default)

KEYWORD(tkDELETE ,1, No, knopNone ,Uni, knopDelete , delete)

KEYWORD(tkDO ,1, No, knopNone , No, knopNone , do)

KEYWORD(tkDOUBLE ,0, No, knopNone , No, knopNone , double)

KEYWORD(tkELSE ,1, No, knopNone , No, knopNone , else)

KEYWORD(tkENSURE ,0, No, knopNone , No, knopNone , ensure)

KEYWORD(tkEVENT ,0, No, knopNone , No, knopNone , event)

KEYWORD(tkFALSE ,1, No, knopNone , No, knopNone , false)

KEYWORD(tkFINAL ,0, No, knopNone , No, knopNone , final)

KEYWORD(tkFINALLY ,1, No, knopNone , No, knopNone , finally)

KEYWORD(tkFLOAT ,0, No, knopNone , No, knopNone , float)

KEYWORD(tkFOR ,1, No, knopNone , No, knopNone , for)

KEYWORD(tkFUNCTION ,1, No, knopNone , No, knopNone , function)

KEYWORD(tkGET ,0, No, knopNone , No, knopNone , get)

KEYWORD(tkGOTO ,0, No, knopNone , No, knopNone , goto)

KEYWORD(tkIF ,1, No, knopNone , No, knopNone , if)

KEYWORD(tkIN ,1, Cmp, knopIn , No, knopNone , in)

KEYWORD(tkINSTANCEOF ,1, Cmp,knopInstOf , No, knopNone , instanceof)

KEYWORD(tkINT ,0, No, knopNone , No, knopNone , int)

KEYWORD(tkINTERNAL ,0, No, knopNone , No, knopNone , internal)

KEYWORD(tkINVARIANT ,0, No, knopNone , No, knopNone , invariant)

KEYWORD(tkLONG ,0, No, knopNone , No, knopNone , long)

KEYWORD(tkNAMESPACE ,0, No, knopNone , No, knopNone , namespace)

KEYWORD(tkNATIVE ,0, No, knopNone , No, knopNone , native)

KEYWORD(tkNEW ,1, No, knopNone , No, knopNone , new)

KEYWORD(tkNULL ,1, No, knopNone , No, knopNone , null)

KEYWORD(tkREQUIRE ,0, No, knopNone , No, knopNone , require)

KEYWORD(tkRETURN ,1, No, knopNone , No, knopNone , return)

KEYWORD(tkSBYTE ,0, No, knopNone , No, knopNone , sbyte)

KEYWORD(tkSET ,0, No, knopNone , No, knopNone , set)

KEYWORD(tkSHORT ,0, No, knopNone , No, knopNone , short)

KEYWORD(tkSWITCH ,1, No, knopNone , No, knopNone , switch)

KEYWORD(tkSYNCHRONIZED,0, No, knopNone , No, knopNone , synchronized)

KEYWORD(tkTHIS ,1, No, knopNone , No, knopNone , this)

KEYWORD(tkTHROW ,1, No, knopNone , No, knopNone , throw)

KEYWORD(tkTHROWS ,0, No, knopNone , No, knopNone , throws)

KEYWORD(tkTRANSIENT ,0, No, knopNone , No, knopNone , transient)

KEYWORD(tkTRUE ,1, No, knopNone , No, knopNone , true)

KEYWORD(tkTRY ,1, No, knopNone , No, knopNone , try)

KEYWORD(tkTYPEOF ,1, No, knopNone ,Uni, knopTypeof , typeof)

KEYWORD(tkUINT ,0, No, knopNone , No, knopNone , uint)

KEYWORD(tkULONG ,0, No, knopNone , No, knopNone , ulong)

KEYWORD(tkUSE ,0, No, knopNone , No, knopNone , use)

KEYWORD(tkUSHORT ,0, No, knopNone , No, knopNone , ushort)

KEYWORD(tkVAR ,1, No, knopNone , No, knopNone , var)

KEYWORD(tkVOID ,1, No, knopNone ,Uni, knopVoid , void)

KEYWORD(tkVOLATILE ,0, No, knopNone , No, knopNone , volatile)

KEYWORD(tkWHILE ,1, No, knopNone , No, knopNone , while)

KEYWORD(tkWITH ,1, No, knopNone , No, knopNone , with)

// Future reserved words that become keywords in ES6

KEYWORD(tkCLASS ,1, No, knopNone , No, knopNone , class)

KEYWORD(tkCONST ,1, No, knopNone , No, knopNone , const)

KEYWORD(tkEXPORT ,1, No, knopNone , No, knopNone , export)

KEYWORD(tkEXTENDS ,1, No, knopNone , No, knopNone , extends)

KEYWORD(tkIMPORT ,1, No, knopNone , No, knopNone , import)

KEYWORD(tkSUPER ,1, No, knopNone , No, knopNone , super)

// Note: yield is still treated as an identifier in non-strict, non-generator functions

// and is special cased in jsscan.js when generating kwd-swtch.h

// Note: yield is a weird operator in that it has assignment expression level precedence

// but looks like a unary operator

KEYWORD(tkYIELD ,1, No, knopNone ,Asg, knopYield , yield)

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

KEYWORD(tkENUM ,1, No, knopNone , No, knopNone , enum)

// Additional future reserved words in strict mode

KEYWORD(tkIMPLEMENTS ,2, No, knopNone , No, knopNone , implements)

KEYWORD(tkINTERFACE ,2, No, knopNone , No, knopNone , interface)

KEYWORD(tkLET ,2, No, knopNone , No, knopNone , let)

KEYWORD(tkPACKAGE ,2, No, knopNone , No, knopNone , package)

KEYWORD(tkPRIVATE ,2, No, knopNone , No, knopNone , private)

KEYWORD(tkPROTECTED ,2, No, knopNone , No, knopNone , protected)

KEYWORD(tkPUBLIC ,2, No, knopNone , No, knopNone , public)

KEYWORD(tkSTATIC ,2, No, knopNone , No, knopNone , static)

S\_KEYWORD(LEval ,3, eval)

S\_KEYWORD(LArguments ,3, arguments)

S\_KEYWORD(LTarget ,3, target)

#undef KEYWORD

#ifndef TOK\_DCL

#define TOK\_DCL(tk,prec2,nop2,prec1,nop1)

#endif //!TOK\_DCL

// The identifier token must follow the last identifier keyword

TOK\_DCL(tkID , No, knopNone , No, knopNone)

// Non-operator non-identifier tokens

TOK\_DCL(tkSColon , No, knopNone , No, knopNone ) // ;

TOK\_DCL(tkRParen , No, knopNone , No, knopNone ) // )

TOK\_DCL(tkRBrack , No, knopNone , No, knopNone ) // ]

TOK\_DCL(tkLCurly , No, knopNone , No, knopNone ) // {

TOK\_DCL(tkRCurly , No, knopNone , No, knopNone ) // }

// Operator non-identifier tokens

TOK\_DCL(tkComma ,Cma, knopComma , No, knopNone ) // ,

TOK\_DCL(tkDArrow ,Asg, knopFncDecl, No, knopNone ) // =>

TOK\_DCL(tkAsg ,Asg, knopAsg , No, knopNone ) // =

TOK\_DCL(tkAsgAdd ,Asg, knopAsgAdd , No, knopNone ) // +=

TOK\_DCL(tkAsgSub ,Asg, knopAsgSub , No, knopNone ) // -=

TOK\_DCL(tkAsgMul ,Asg, knopAsgMul , No, knopNone ) // \*=

TOK\_DCL(tkAsgDiv ,Asg, knopAsgDiv , No, knopNone ) // /=

TOK\_DCL(tkAsgExpo ,Asg, knopAsgExpo, No, knopNone ) // \*\*=

TOK\_DCL(tkAsgMod ,Asg, knopAsgMod , No, knopNone ) // %=

TOK\_DCL(tkAsgAnd ,Asg, knopAsgAnd , No, knopNone ) // &=

TOK\_DCL(tkAsgXor ,Asg, knopAsgXor , No, knopNone ) // ^=

TOK\_DCL(tkAsgOr ,Asg, knopAsgOr , No, knopNone ) // |=

TOK\_DCL(tkAsgLsh ,Asg, knopAsgLsh , No, knopNone ) // <<=

TOK\_DCL(tkAsgRsh ,Asg, knopAsgRsh , No, knopNone ) // >>=

TOK\_DCL(tkAsgRs2 ,Asg, knopAsgRs2 , No, knopNone ) // >>>=

TOK\_DCL(tkQMark ,Que, knopQmark , No, knopNone ) // ?

TOK\_DCL(tkColon , No, knopNone , No, knopNone ) // :

TOK\_DCL(tkLogOr ,Lor, knopLogOr , No, knopNone ) // ||

TOK\_DCL(tkLogAnd ,Lan, knopLogAnd , No, knopNone ) // &&

TOK\_DCL(tkOr ,Bor, knopOr , No, knopNone ) // |

TOK\_DCL(tkXor ,Xor, knopXor , No, knopNone ) // ^

TOK\_DCL(tkAnd ,Ban, knopAnd , No, knopNone ) // &

TOK\_DCL(tkEQ ,Equ, knopEq , No, knopNone ) // ==

TOK\_DCL(tkNE ,Equ, knopNe , No, knopNone ) // !=

TOK\_DCL(tkEqv ,Equ, knopEqv , No, knopNone ) // ===

TOK\_DCL(tkNEqv ,Equ, knopNEqv , No, knopNone ) // !==

TOK\_DCL(tkLT ,Cmp, knopLt , No, knopNone ) // <

TOK\_DCL(tkLE ,Cmp, knopLe , No, knopNone ) // <=

TOK\_DCL(tkGT ,Cmp, knopGt , No, knopNone ) // >

TOK\_DCL(tkGE ,Cmp, knopGe , No, knopNone ) // >=

TOK\_DCL(tkLsh ,Shf, knopLsh , No, knopNone ) // <<

TOK\_DCL(tkRsh ,Shf, knopRsh , No, knopNone ) // >>

TOK\_DCL(tkRs2 ,Shf, knopRs2 , No, knopNone ) // >>>

TOK\_DCL(tkAdd ,Add, knopAdd ,Uni, knopPos ) // +

TOK\_DCL(tkSub ,Add, knopSub ,Uni, knopNeg ) // -

TOK\_DCL(tkExpo ,Expo, knopExpo , No, knopNone ) // \*\*

TOK\_DCL(tkStar ,Mul, knopMul , No, knopNone ) // \*

TOK\_DCL(tkDiv ,Mul, knopDiv , No, knopNone ) // /

TOK\_DCL(tkPct ,Mul, knopMod , No, knopNone ) // %

TOK\_DCL(tkTilde , No, knopNone ,Uni, knopNot ) // ~

TOK\_DCL(tkBang , No, knopNone ,Uni, knopLogNot ) // !

TOK\_DCL(tkInc , No, knopNone ,Uni, knopIncPre ) // ++

TOK\_DCL(tkDec , No, knopNone ,Uni, knopDecPre ) // --

TOK\_DCL(tkEllipsis , No, knopNone ,Spr, knopEllipsis ) // ...

TOK\_DCL(tkLParen , No, knopNone , No, knopNone ) // (

TOK\_DCL(tkLBrack , No, knopNone , No, knopNone ) // [

TOK\_DCL(tkDot , No, knopNone , No, knopNone ) // .

// String template tokens

TOK\_DCL(tkStrTmplBasic , No, knopNone , No, knopNone ) // `...`

TOK\_DCL(tkStrTmplBegin , No, knopNone , No, knopNone ) // `...${

TOK\_DCL(tkStrTmplMid , No, knopNone , No, knopNone ) // }...${ Note: tkStrTmplMid and tkStrTmplEnd tokens do not actually contain the opening '}' character.

TOK\_DCL(tkStrTmplEnd , No, knopNone , No, knopNone ) // }...` Since the scanner can't disambiguate a tkRCurly which is part of the expression, literal, or string template syntax

// we check to make sure the token after parsing the expression is a tkRCurly and put the scanner into a string template

// scanning mode which will scan the string literal and search for the closing '${' or '`'.

TOK\_DCL(tkComment , No, knopNone, No, knopNone ) // Comment for syntax coloring

TOK\_DCL(tkScanError , No, knopNone, No, knopNone ) // Error in syntax coloring

#undef TOK\_DCL

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

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

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

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

// GENERATED FILE, DO NOT HAND-MODIFY!

// Generated with the following command line: wscript jsscan.js kwd-lsc.h kwd-swtch.h

// This should be regenerated whenever the keywords change.

case 'a':

if (identifyKwds)

{

switch (p[0]) {

case 'w':

if (p[1] == 'a' && p[2] == 'i' && p[3] == 't' && !IsIdContinueNext(p+4, last)) {

p += 4;

if (this->m\_fAwaitIsKeyword || !this->m\_parser || this->m\_parser->IsStrictMode()) {

token = tkAWAIT;

goto LReserved;

}

goto LIdentifier;

}

break;

case 'r':

if (p[1] == 'g' && p[2] == 'u' && p[3] == 'm' && p[4] == 'e' && p[5] == 'n' && p[6] == 't' && p[7] == 's' && !IsIdContinueNext(p+8, last)) {

p += 8;

goto LArguments;

}

break;

}

}

goto LIdentifier;

case 'b':

if (identifyKwds)

{

if (p[0] == 'r' && p[1] == 'e' && p[2] == 'a' && p[3] == 'k' && !IsIdContinueNext(p+4, last)) {

p += 4;

token = tkBREAK;

goto LReserved;

}

}

goto LIdentifier;

case 'c':

if (identifyKwds)

{

switch (p[0]) {

case 'a':

switch (p[1]) {

case 's':

if (p[2] == 'e' && !IsIdContinueNext(p+3, last)) {

p += 3;

token = tkCASE;

goto LReserved;

}

break;

case 't':

if (p[2] == 'c' && p[3] == 'h' && !IsIdContinueNext(p+4, last)) {

p += 4;

token = tkCATCH;

goto LReserved;

}

break;

}

break;

case 'o':

if (p[1] == 'n') {

switch (p[2]) {

case 't':

if (p[3] == 'i' && p[4] == 'n' && p[5] == 'u' && p[6] == 'e' && !IsIdContinueNext(p+7, last)) {

p += 7;

token = tkCONTINUE;

goto LReserved;

}

break;

case 's':

if (p[3] == 't' && !IsIdContinueNext(p+4, last)) {

p += 4;

token = tkCONST;

goto LReserved;

}

break;

}

}

break;

case 'l':

if (p[1] == 'a' && p[2] == 's' && p[3] == 's' && !IsIdContinueNext(p+4, last)) {

p += 4;

token = tkCLASS;

goto LReserved;

}

break;

}

}

goto LIdentifier;

case 'd':

if (identifyKwds)

{

switch (p[0]) {

case 'e':

switch (p[1]) {

case 'b':

if (p[2] == 'u' && p[3] == 'g' && p[4] == 'g' && p[5] == 'e' && p[6] == 'r' && !IsIdContinueNext(p+7, last)) {

p += 7;

token = tkDEBUGGER;

goto LReserved;

}

break;

case 'f':

if (p[2] == 'a' && p[3] == 'u' && p[4] == 'l' && p[5] == 't' && !IsIdContinueNext(p+6, last)) {

p += 6;

token = tkDEFAULT;

goto LReserved;

}

break;

case 'l':

if (p[2] == 'e' && p[3] == 't' && p[4] == 'e' && !IsIdContinueNext(p+5, last)) {

p += 5;

token = tkDELETE;

goto LReserved;

}

break;

}

break;

case 'o':

if (!IsIdContinueNext(p+1, last)) {

p += 1;

token = tkDO;

goto LReserved;

}

break;

}

}

goto LIdentifier;

case 'e':

if (identifyKwds)

{

switch (p[0]) {

case 'l':

if (p[1] == 's' && p[2] == 'e' && !IsIdContinueNext(p+3, last)) {

p += 3;

token = tkELSE;

goto LReserved;

}

break;

case 'x':

switch (p[1]) {

case 'p':

if (p[2] == 'o' && p[3] == 'r' && p[4] == 't' && !IsIdContinueNext(p+5, last)) {

p += 5;

token = tkEXPORT;

goto LReserved;

}

break;

case 't':

if (p[2] == 'e' && p[3] == 'n' && p[4] == 'd' && p[5] == 's' && !IsIdContinueNext(p+6, last)) {

p += 6;

token = tkEXTENDS;

goto LReserved;

}

break;

}

break;

case 'n':

if (p[1] == 'u' && p[2] == 'm' && !IsIdContinueNext(p+3, last)) {

p += 3;

token = tkENUM;

goto LReserved;

}

break;

case 'v':

if (p[1] == 'a' && p[2] == 'l' && !IsIdContinueNext(p+3, last)) {

p += 3;

goto LEval;

}

break;

}

}

goto LIdentifier;

case 'f':

if (identifyKwds)

{

switch (p[0]) {

case 'a':

if (p[1] == 'l' && p[2] == 's' && p[3] == 'e' && !IsIdContinueNext(p+4, last)) {

p += 4;

token = tkFALSE;

goto LReserved;

}

break;

case 'i':

if (p[1] == 'n' && p[2] == 'a' && p[3] == 'l' && p[4] == 'l' && p[5] == 'y' && !IsIdContinueNext(p+6, last)) {

p += 6;

token = tkFINALLY;

goto LReserved;

}

break;

case 'o':

if (p[1] == 'r' && !IsIdContinueNext(p+2, last)) {

p += 2;

token = tkFOR;

goto LReserved;

}

break;

case 'u':

if (p[1] == 'n' && p[2] == 'c' && p[3] == 't' && p[4] == 'i' && p[5] == 'o' && p[6] == 'n' && !IsIdContinueNext(p+7, last)) {

p += 7;

token = tkFUNCTION;

goto LReserved;

}

break;

}

}

goto LIdentifier;

case 'i':

if (identifyKwds)

{

switch (p[0]) {

case 'f':

if (!IsIdContinueNext(p+1, last)) {

p += 1;

token = tkIF;

goto LReserved;

}

break;

case 'n':

switch (p[1]) {

case 's':

if (p[2] == 't' && p[3] == 'a' && p[4] == 'n' && p[5] == 'c' && p[6] == 'e' && p[7] == 'o' && p[8] == 'f' && !IsIdContinueNext(p+9, last)) {

p += 9;

token = tkINSTANCEOF;

goto LReserved;

}

break;

case 't':

if (p[2] == 'e' && p[3] == 'r' && p[4] == 'f' && p[5] == 'a' && p[6] == 'c' && p[7] == 'e' && !IsIdContinueNext(p+8, last)) {

p += 8;

if (!this->m\_parser || this->m\_parser->IsStrictMode()) {

token = tkINTERFACE;

goto LReserved;

}

goto LIdentifier;

}

break;

}

if (!IsIdContinueNext(p+1,last)) {

p += 1;

token = tkIN;

goto LReserved;

}

break;

case 'm':

if (p[1] == 'p') {

switch (p[2]) {

case 'o':

if (p[3] == 'r' && p[4] == 't' && !IsIdContinueNext(p+5, last)) {

p += 5;

token = tkIMPORT;

goto LReserved;

}

break;

case 'l':

if (p[3] == 'e' && p[4] == 'm' && p[5] == 'e' && p[6] == 'n' && p[7] == 't' && p[8] == 's' && !IsIdContinueNext(p+9, last)) {

p += 9;

if (!this->m\_parser || this->m\_parser->IsStrictMode()) {

token = tkIMPLEMENTS;

goto LReserved;

}

goto LIdentifier;

}

break;

}

}

break;

}

}

goto LIdentifier;

case 'n':

if (identifyKwds)

{

switch (p[0]) {

case 'e':

if (p[1] == 'w' && !IsIdContinueNext(p+2, last)) {

p += 2;

token = tkNEW;

goto LReserved;

}

break;

case 'u':

if (p[1] == 'l' && p[2] == 'l' && !IsIdContinueNext(p+3, last)) {

p += 3;

token = tkNULL;

goto LReserved;

}

break;

}

}

goto LIdentifier;

case 'r':

if (identifyKwds)

{

if (p[0] == 'e' && p[1] == 't' && p[2] == 'u' && p[3] == 'r' && p[4] == 'n' && !IsIdContinueNext(p+5, last)) {

p += 5;

token = tkRETURN;

goto LReserved;

}

}

goto LIdentifier;

case 's':

if (identifyKwds)

{

switch (p[0]) {

case 'w':

if (p[1] == 'i' && p[2] == 't' && p[3] == 'c' && p[4] == 'h' && !IsIdContinueNext(p+5, last)) {

p += 5;

token = tkSWITCH;

goto LReserved;

}

break;

case 'u':

if (p[1] == 'p' && p[2] == 'e' && p[3] == 'r' && !IsIdContinueNext(p+4, last)) {

p += 4;

token = tkSUPER;

goto LReserved;

}

break;

case 't':

if (p[1] == 'a' && p[2] == 't' && p[3] == 'i' && p[4] == 'c' && !IsIdContinueNext(p+5, last)) {

p += 5;

if (!this->m\_parser || this->m\_parser->IsStrictMode()) {

token = tkSTATIC;

goto LReserved;

}

goto LIdentifier;

}

break;

}

}

goto LIdentifier;

case 't':

if (identifyKwds)

{

switch (p[0]) {

case 'h':

switch (p[1]) {

case 'i':

if (p[2] == 's' && !IsIdContinueNext(p+3, last)) {

p += 3;

token = tkTHIS;

goto LReserved;

}

break;

case 'r':

if (p[2] == 'o' && p[3] == 'w' && !IsIdContinueNext(p+4, last)) {

p += 4;

token = tkTHROW;

goto LReserved;

}

break;

}

break;

case 'r':

switch (p[1]) {

case 'u':

if (p[2] == 'e' && !IsIdContinueNext(p+3, last)) {

p += 3;

token = tkTRUE;

goto LReserved;

}

break;

case 'y':

if (!IsIdContinueNext(p+2, last)) {

p += 2;

token = tkTRY;

goto LReserved;

}

break;

}

break;

case 'y':

if (p[1] == 'p' && p[2] == 'e' && p[3] == 'o' && p[4] == 'f' && !IsIdContinueNext(p+5, last)) {

p += 5;

token = tkTYPEOF;

goto LReserved;

}

break;

case 'a':

if (p[1] == 'r' && p[2] == 'g' && p[3] == 'e' && p[4] == 't' && !IsIdContinueNext(p+5, last)) {

p += 5;

goto LTarget;

}

break;

}

}

goto LIdentifier;

case 'v':

if (identifyKwds)

{

switch (p[0]) {

case 'a':

if (p[1] == 'r' && !IsIdContinueNext(p+2, last)) {

p += 2;

token = tkVAR;

goto LReserved;

}

break;

case 'o':

if (p[1] == 'i' && p[2] == 'd' && !IsIdContinueNext(p+3, last)) {

p += 3;

token = tkVOID;

goto LReserved;

}

break;

}

}

goto LIdentifier;

case 'w':

if (identifyKwds)

{

switch (p[0]) {

case 'h':

if (p[1] == 'i' && p[2] == 'l' && p[3] == 'e' && !IsIdContinueNext(p+4, last)) {

p += 4;

token = tkWHILE;

goto LReserved;

}

break;

case 'i':

if (p[1] == 't' && p[2] == 'h' && !IsIdContinueNext(p+3, last)) {

p += 3;

token = tkWITH;

goto LReserved;

}

break;

}

}

goto LIdentifier;

case 'y':

if (identifyKwds)

{

if (p[0] == 'i' && p[1] == 'e' && p[2] == 'l' && p[3] == 'd' && !IsIdContinueNext(p+4, last)) {

p += 4;

if (this->m\_fYieldIsKeyword || !this->m\_parser || this->m\_parser->IsStrictMode()) {

token = tkYIELD;

goto LReserved;

}

goto LIdentifier;

}

}

goto LIdentifier;

case 'l':

if (identifyKwds)

{

if (p[0] == 'e' && p[1] == 't' && !IsIdContinueNext(p+2, last)) {

p += 2;

if (!this->m\_parser || this->m\_parser->IsStrictMode()) {

token = tkLET;

goto LReserved;

}

goto LIdentifier;

}

}

goto LIdentifier;

case 'p':

if (identifyKwds)

{

switch (p[0]) {

case 'a':

if (p[1] == 'c' && p[2] == 'k' && p[3] == 'a' && p[4] == 'g' && p[5] == 'e' && !IsIdContinueNext(p+6, last)) {

p += 6;

if (!this->m\_parser || this->m\_parser->IsStrictMode()) {

token = tkPACKAGE;

goto LReserved;

}

goto LIdentifier;

}

break;

case 'r':

switch (p[1]) {

case 'i':

if (p[2] == 'v' && p[3] == 'a' && p[4] == 't' && p[5] == 'e' && !IsIdContinueNext(p+6, last)) {

p += 6;

if (!this->m\_parser || this->m\_parser->IsStrictMode()) {

token = tkPRIVATE;

goto LReserved;

}

goto LIdentifier;

}

break;

case 'o':

if (p[2] == 't' && p[3] == 'e' && p[4] == 'c' && p[5] == 't' && p[6] == 'e' && p[7] == 'd' && !IsIdContinueNext(p+8, last)) {

p += 8;

if (!this->m\_parser || this->m\_parser->IsStrictMode()) {

token = tkPROTECTED;

goto LReserved;

}

goto LIdentifier;

}

break;

}

break;

case 'u':

if (p[1] == 'b' && p[2] == 'l' && p[3] == 'i' && p[4] == 'c' && !IsIdContinueNext(p+5, last)) {

p += 5;

if (!this->m\_parser || this->m\_parser->IsStrictMode()) {

token = tkPUBLIC;

goto LReserved;

}

goto LIdentifier;

}

break;

}

}

goto LIdentifier;

// characters not in a reserved word

case 'g': case 'h': case 'j':

case 'k': case 'm': case 'o':

case 'q':

case 'u': case 'x':

case 'z':

goto LIdentifier;

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

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

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

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

// Generated by switch.exe on Wed Jan 28 10:56:16 2009

// Hand-edited to exclude identifiers that are not JScript keywords.

#if \_WIN32 && \_M\_IX86

\_\_asm

{

mov eax,luHash

// 0001 while

cmp eax, 0x009FF239

jb L0003

je LEqual\_while

// 0002 protected

cmp eax, 0x7679AA2A

jb L0005

// 0004 package

cmp eax, 0xA9DEE87C

ja LDefault

// 0009 decimal

cmp eax, 0x98F1BFFF

jb L0013

// 0012 extends

cmp eax, 0x9C13124B

jb L0025

je LEqual\_extends

// 0024 finally

cmp eax, 0x9C3672EF

je LEqual\_finally

jmp LDefault

L0025:

// 0025 default

cmp eax, 0x98F50305

je LEqual\_default

jmp LDefault

L0013:

cmp eax, 0x79C146FA

je LEqual\_instanceof

jmp LDefault

L0005:

// 0005 static

cmp eax, 0x0A5727B8

jb L000B

// 000A continue

cmp eax, 0x1EF7AD75

jb L0015

je LEqual\_continue

// 0014 synchronized

cmp eax, 0x3189C9E0

jb L0029

// 0028 function

cmp eax, 0x70E47CD6

je LEqual\_function

jmp LDefault

L0029:

// 0029 debugger

cmp eax, 0x2806F445

je LEqual\_debugger

jmp LDefault

L0015:

// 0015 typeof

cmp eax, 0x0A7440A7

jb L002B

je LEqual\_typeof

L002B:

// 002B throws

// 0057 switch

cmp eax, 0x0A5B93A2

je LEqual\_switch

jmp LDefault

L000B:

// 000B export

cmp eax, 0x092E0B32

jb L0017

je LEqual\_export

// 0016 public

cmp eax, 0x0A17792F

jb L002D

// 002C return

cmp eax, 0x0A2FCE00

je LEqual\_return

jmp LDefault

L002D:

// 002D native

// 005B import

cmp eax, 0x0976AFAB

je LEqual\_import

jmp LDefault

L0017:

// 0017 double

cmp eax, 0x090D396B

jb L002F

jmp LDefault

L002F:

// 002F delete

cmp eax, 0x08FFD213

je LEqual\_delete

jmp LDefault

L0003:

// 0003 null

cmp eax, 0x0008CABB

jb L0007

je LEqual\_null

// 0006 false

cmp eax, 0x0089C56B

jb L000D

je LEqual\_false

// 000C short

cmp eax, 0x009AE070

jb LDefault

// 0018 throw

cmp eax, 0x009C29E4

jb L0031

je LEqual\_throw

L0031:

// 0031 super

cmp eax, 0x009BDA2F

je LEqual\_super

jmp LDefault

L000D:

// 000D with

cmp eax, 0x00096A6C

jb L001B

je LEqual\_with

// 001A class

cmp eax, 0x0086B966

jb L0035

je LEqual\_class

// 0034 const

cmp eax, 0x008701A7

je LEqual\_const

jmp LDefault

L0035:

// 0035 catch

cmp eax, 0x0085FAA3

je LEqual\_catch

// 006B break

cmp eax, 0x0085E995

je LEqual\_break

jmp LDefault

L001B:

// 001B uint

cmp eax, 0x000943B0

jb L0037

// 0036 void

cmp eax, 0x00095D42

je LEqual\_void

jmp LDefault

L0037:

// 0037 true

cmp eax, 0x00093B10

je LEqual\_true

// 006F this

cmp eax, 0x00092F08

je LEqual\_this

jmp LDefault

L0007:

// 0007 var

cmp eax, 0x00008C19

jb L000F

je LEqual\_var

// 000E else

cmp eax, 0x00081449

jb L003B

je LEqual\_else

// 001C goto

cmp eax, 0x00083E29

jb L0039

jmp LDefault

L0039:

// 0039 enum

cmp eax, 0x000816B5

je LEqual\_enum

jmp LDefault

L003B:

// 003B case

cmp eax, 0x0007E17C

je LEqual\_case

jmp LDefault

L000F:

// 000F get

cmp eax, 0x00007B70

jb L001F

// 001E set

cmp eax, 0x000088FC

jb L003D

// 003C try

cmp eax, 0x00008AFF

je LEqual\_try

jmp LDefault

L003D:

// 003D new

cmp eax, 0x0000835A

je LEqual\_new

jmp LDefault

L001F:

// 001F in

cmp eax, 0x00000767

jb L003F

je LEqual\_in

// 003E for

cmp eax, 0x00007AF7

je LEqual\_for

jmp LDefault

L003F:

// 003F if

cmp eax, 0x0000075F

je LEqual\_if

// 007F do

cmp eax, 0x00000713

je LEqual\_do

jmp LDefault

}

#else

// 0001 while

if (luHash < 0x009FF239) goto L0003;

if (luHash == 0x009FF239) goto LEqual\_while;

// 0002 protected

if (luHash < 0x7679AA2A) goto L0005;

// 0004 package

if (luHash < 0xA9DEE87C) goto L0009;

goto LDefault;

L0009:

// 0009 decimal

if (luHash < 0x98F1BFFF) goto L0013;

// 0012 extends

if (luHash < 0x9C13124B) goto L0025;

if (luHash == 0x9C13124B) goto LEqual\_extends;

// 0024 finally

if (luHash == 0x9C3672EF) goto LEqual\_finally;

goto LDefault;

L0025:

// 0025 default

if (luHash == 0x98F50305) goto LEqual\_default;

goto LDefault;

L0013:

// 004F instanceof

if (luHash == 0x79C146FA) goto LEqual\_instanceof;

goto LDefault;

L0005:

// 0005 static

if (luHash < 0x0A5727B8) goto L000B;

// 000A continue

if (luHash < 0x1EF7AD75) goto L0015;

if (luHash == 0x1EF7AD75) goto LEqual\_continue;

// 0014 synchronized

if (luHash < 0x3189C9E0) goto L0029;

// 0028 function

if (luHash == 0x70E47CD6) goto LEqual\_function;

goto LDefault;

L0029:

// 0029 debugger

if (luHash == 0x2806F445) goto LEqual\_debugger;

goto LDefault;

L0015:

// 0015 typeof

if (luHash < 0x0A7440A7) goto L002B;

if (luHash == 0x0A7440A7) goto LEqual\_typeof;

// 002A ushort

goto LDefault;

L002B:

// 0057 switch

if (luHash == 0x0A5B93A2) goto LEqual\_switch;

goto LDefault;

L000B:

// 000B export

if (luHash < 0x092E0B32) goto L0017;

if (luHash == 0x092E0B32) goto LEqual\_export;

// 0016 public

if (luHash < 0x0A17792F) goto L002D;

// 002C return

if (luHash == 0x0A2FCE00) goto LEqual\_return;

goto LDefault;

L002D:

// 005B import

if (luHash == 0x0976AFAB) goto LEqual\_import;

goto LDefault;

L0017:

// 0017 double

if (luHash < 0x090D396B) goto L002F;

goto LDefault;

L002F:

// 002F delete

if (luHash == 0x08FFD213) goto LEqual\_delete;

goto LDefault;

L0003:

// 0003 null

if (luHash < 0x0008CABB) goto L0007;

if (luHash == 0x0008CABB) goto LEqual\_null;

// 0006 false

if (luHash < 0x0089C56B) goto L000D;

if (luHash == 0x0089C56B) goto LEqual\_false;

// 000C short

if (luHash < 0x009AE070) goto LDefault;

// 0018 throw

if (luHash < 0x009C29E4) goto L0031;

if (luHash == 0x009C29E4) goto LEqual\_throw;

// 0030 ulong

goto LDefault;

L0031:

// 0031 super

if (luHash == 0x009BDA2F) goto LEqual\_super;

goto LDefault;

L000D:

// 000D with

if (luHash < 0x00096A6C) goto L001B;

if (luHash == 0x00096A6C) goto LEqual\_with;

// 001A class

if (luHash < 0x0086B966) goto L0035;

if (luHash == 0x0086B966) goto LEqual\_class;

// 0034 const

if (luHash == 0x008701A7) goto LEqual\_const;

goto LDefault;

L0035:

// 0035 catch

if (luHash == 0x0085FAA3) goto LEqual\_catch;

// 006B break

if (luHash == 0x0085E995) goto LEqual\_break;

goto LDefault;

L001B:

// 001B uint

if (luHash < 0x000943B0) goto L0037;

// 0036 void

if (luHash == 0x00095D42) goto LEqual\_void;

goto LDefault;

L0037:

// 0037 true

if (luHash == 0x00093B10) goto LEqual\_true;

// 006F this

if (luHash == 0x00092F08) goto LEqual\_this;

goto LDefault;

L0007:

// 0007 var

if (luHash < 0x00008C19) goto L000F;

if (luHash == 0x00008C19) goto LEqual\_var;

// 000E else

if (luHash < 0x00081449) goto L001D;

if (luHash == 0x00081449) goto LEqual\_else;

// 001C goto

if (luHash < 0x00083E29) goto L0039;

goto LDefault;

L0039:

// 0039 enum

if (luHash == 0x000816B5) goto LEqual\_enum;

goto LDefault;

L001D:

// 001D char

if (luHash < 0x0007E83E) goto L003B;

goto LDefault;

L003B:

// 003B case

if (luHash == 0x0007E17C) goto LEqual\_case;

goto LDefault;

L000F:

// 000F get

if (luHash < 0x00007B70) goto L001F;

// 001E set

if (luHash < 0x000088FC) goto L003D;

// 003C try

if (luHash == 0x00008AFF) goto LEqual\_try;

goto LDefault;

L003D:

// 003D new

if (luHash == 0x0000835A) goto LEqual\_new;

// 007B int

goto LDefault;

L001F:

// 001F in

if (luHash < 0x00000767) goto L003F;

if (luHash == 0x00000767) goto LEqual\_in;

// 003E for

if (luHash == 0x00007AF7) goto LEqual\_for;

goto LDefault;

L003F:

// 003F if

if (luHash == 0x0000075F) goto LEqual\_if;

// 007F do

if (luHash == 0x00000713) goto LEqual\_do;

goto LDefault;

#endif // \_WIN32 && \_M\_IX86

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

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

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

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

// Generated by switch.exe on Wed Mar 30 14:58:29 2011

#if \_WIN32 && \_M\_IX86

\_\_asm

{

mov eax,luHash

// 0001 while

cmp eax, 0x009FF239

jb L0003

je LEqual\_while

// 0002 function

cmp eax, 0x70E47CD6

jb L0005

je LEqual\_function

// 0004 package

cmp eax, 0xA9DEE87C

jb L0009

je LEqual\_package

// 0008 invariant

cmp eax, 0xBEA8C42C

jb L0011

je LEqual\_invariant

// 0010 transient

cmp eax, 0xE10C0168

jb L0021

je LEqual\_transient

// 0020 volatile

cmp eax, 0xEF691640

je LEqual\_volatile

jmp LDefault

L0021:

// 0021 abstract

cmp eax, 0xDBC60B24

je LEqual\_abstract

jmp LDefault

L0011:

// 0011 internal

cmp eax, 0xB0B68A3D

jb L0023

je LEqual\_internal

// 0022 interface

cmp eax, 0xBC1E9451

je LEqual\_interface

jmp LDefault

L0023:

// 0023 require

cmp eax, 0xAD28D1BD

je LEqual\_require

// 0047 private

cmp eax, 0xAB57B15B

je LEqual\_private

jmp LDefault

L0009:

// 0009 boolean

cmp eax, 0x96F94400

jb L0013

je LEqual\_boolean

// 0012 extends

cmp eax, 0x9C13124B

jb L0025

je LEqual\_extends

// 0024 finally

cmp eax, 0x9C3672EF

je LEqual\_finally

jmp LDefault

L0025:

// 0025 default

cmp eax, 0x98F50305

je LEqual\_default

// 004B decimal

cmp eax, 0x98F1BFFF

je LEqual\_decimal

jmp LDefault

L0013:

// 0013 implements

cmp eax, 0x856AF30E

jb L0027

je LEqual\_implements

// 0026 namespace

cmp eax, 0x9309F69D

je LEqual\_namespace

jmp LDefault

L0027:

// 0027 instanceof

cmp eax, 0x79C146FA

je LEqual\_instanceof

// 004F protected

cmp eax, 0x7679AA2A

je LEqual\_protected

jmp LDefault

L0005:

// 0005 return

cmp eax, 0x0A2FCE00

jb L000B

je LEqual\_return

// 000A ushort

cmp eax, 0x0A81B775

jb L0015

je LEqual\_ushort

// 0014 debugger

cmp eax, 0x2806F445

jb L0029

je LEqual\_debugger

// 0028 synchronized

cmp eax, 0x3189C9E0

je LEqual\_synchronized

jmp LDefault

L0029:

// 0029 continue

cmp eax, 0x1EF7AD75

je LEqual\_continue

jmp LDefault

L0015:

// 0015 throws

cmp eax, 0x0A5EC897

jb L002B

je LEqual\_throws

// 002A typeof

cmp eax, 0x0A7440A7

je LEqual\_typeof

jmp LDefault

L002B:

// 002B switch

cmp eax, 0x0A5B93A2

je LEqual\_switch

// 0057 static

cmp eax, 0x0A5727B8

je LEqual\_static

jmp LDefault

L000B:

// 000B ensure

cmp eax, 0x09218CF2

jb L0017

je LEqual\_ensure

// 0016 native

cmp eax, 0x09D3FE67

jb L002D

je LEqual\_native

// 002C public

cmp eax, 0x0A17792F

je LEqual\_public

jmp LDefault

L002D:

// 002D import

cmp eax, 0x0976AFAB

je LEqual\_import

// 005B export

cmp eax, 0x092E0B32

je LEqual\_export

jmp LDefault

L0017:

// 0017 delete

cmp eax, 0x08FFD213

jb L002F

je LEqual\_delete

// 002E double

cmp eax, 0x090D396B

je LEqual\_double

jmp LDefault

L002F:

// 002F assert

cmp eax, 0x08D130F2

je LEqual\_assert

// 005F yield

cmp eax, 0x00A28D67

je LEqual\_yield

jmp LDefault

L0003:

// 0003 null

cmp eax, 0x0008CABB

jb L0007

je LEqual\_null

// 0006 false

cmp eax, 0x0089C56B

jb L000D

je LEqual\_false

// 000C short

cmp eax, 0x009AE070

jb L0019

je LEqual\_short

// 0018 throw

cmp eax, 0x009C29E4

jb L0031

je LEqual\_throw

// 0030 ulong

cmp eax, 0x009DB965

je LEqual\_ulong

jmp LDefault

L0031:

// 0031 super

cmp eax, 0x009BDA2F

je LEqual\_super

jmp LDefault

L0019:

// 0019 float

cmp eax, 0x008A9AC6

jb L0033

je LEqual\_float

// 0032 sbyte

cmp eax, 0x009A78A7

je LEqual\_sbyte

jmp LDefault

L0033:

// 0033 final

cmp eax, 0x008A600A

je LEqual\_final

// 0067 event

cmp eax, 0x008A0A02

je LEqual\_event

jmp LDefault

L000D:

// 000D with

cmp eax, 0x00096A6C

jb L001B

je LEqual\_with

// 001A class

cmp eax, 0x0086B966

jb L0035

je LEqual\_class

// 0034 const

cmp eax, 0x008701A7

je LEqual\_const

jmp LDefault

L0035:

// 0035 catch

cmp eax, 0x0085FAA3

je LEqual\_catch

// 006B break

cmp eax, 0x0085E995

je LEqual\_break

// await

cmp eax, 0x0084FF56

je LEqual\_await

jmp LDefault

L001B:

// 001B uint

cmp eax, 0x000943B0

jb L0037

je LEqual\_uint

// 0036 void

cmp eax, 0x00095D42

je LEqual\_void

jmp LDefault

L0037:

// 0037 true

cmp eax, 0x00093B10

je LEqual\_true

// 006F this

cmp eax, 0x00092F08

je LEqual\_this

jmp LDefault

L0007:

// 0007 try

cmp eax, 0x00008AFF

jb L000F

je LEqual\_try

// 000E byte

cmp eax, 0x0007E974

jb L001D

je LEqual\_byte

// 001C goto

cmp eax, 0x00083E29

jb L0039

je LEqual\_goto

// 0038 long

cmp eax, 0x00089DB0

je LEqual\_long

jmp LDefault

L0039:

// 0039 enum

cmp eax, 0x000816B5

je LEqual\_enum

// 0073 else

cmp eax, 0x00081449

je LEqual\_else

jmp LDefault

L001D:

// 001D case

cmp eax, 0x0007E17C

jb L003B

je LEqual\_case

// 003A char

cmp eax, 0x0007E83E

je LEqual\_char

jmp LDefault

L003B:

// 003B use

cmp eax, 0x00008C1D

je LEqual\_use

// 0077 var

cmp eax, 0x00008C19

je LEqual\_var

jmp LDefault

L000F:

// 000F get

cmp eax, 0x00007B70

jb L001F

je LEqual\_get

// 001E new

cmp eax, 0x0000835A

jb L003D

je LEqual\_new

// 003C set

cmp eax, 0x000088FC

je LEqual\_set

jmp LDefault

L003D:

// 003D let

cmp eax, 0x00008115

je LEqual\_let

// 007B int

cmp eax, 0x00007E4B

je LEqual\_int

jmp LDefault

L001F:

// 001F in

cmp eax, 0x00000767

jb L003F

je LEqual\_in

// 003E for

cmp eax, 0x00007AF7

je LEqual\_for

jmp LDefault

L003F:

// 003F if

cmp eax, 0x0000075F

je LEqual\_if

// 007F do

cmp eax, 0x00000713

je LEqual\_do

jmp LDefault

}

#else

// 0001 while

if (luHash < 0x009FF239) goto L0003;

if (luHash == 0x009FF239) goto LEqual\_while;

// 0002 function

if (luHash < 0x70E47CD6) goto L0005;

if (luHash == 0x70E47CD6) goto LEqual\_function;

// 0004 package

if (luHash < 0xA9DEE87C) goto L0009;

if (luHash == 0xA9DEE87C) goto LEqual\_package;

// 0008 invariant

if (luHash < 0xBEA8C42C) goto L0011;

if (luHash == 0xBEA8C42C) goto LEqual\_invariant;

// 0010 transient

if (luHash < 0xE10C0168) goto L0021;

if (luHash == 0xE10C0168) goto LEqual\_transient;

// 0020 volatile

if (luHash == 0xEF691640) goto LEqual\_volatile;

goto LDefault;

L0021:

// 0021 abstract

if (luHash == 0xDBC60B24) goto LEqual\_abstract;

goto LDefault;

L0011:

// 0011 internal

if (luHash < 0xB0B68A3D) goto L0023;

if (luHash == 0xB0B68A3D) goto LEqual\_internal;

// 0022 interface

if (luHash == 0xBC1E9451) goto LEqual\_interface;

goto LDefault;

L0023:

// 0023 require

if (luHash == 0xAD28D1BD) goto LEqual\_require;

// 0047 private

if (luHash == 0xAB57B15B) goto LEqual\_private;

goto LDefault;

L0009:

// 0009 boolean

if (luHash < 0x96F94400) goto L0013;

if (luHash == 0x96F94400) goto LEqual\_boolean;

// 0012 extends

if (luHash < 0x9C13124B) goto L0025;

if (luHash == 0x9C13124B) goto LEqual\_extends;

// 0024 finally

if (luHash == 0x9C3672EF) goto LEqual\_finally;

goto LDefault;

L0025:

// 0025 default

if (luHash == 0x98F50305) goto LEqual\_default;

// 004B decimal

if (luHash == 0x98F1BFFF) goto LEqual\_decimal;

goto LDefault;

L0013:

// 0013 implements

if (luHash < 0x856AF30E) goto L0027;

if (luHash == 0x856AF30E) goto LEqual\_implements;

// 0026 namespace

if (luHash == 0x9309F69D) goto LEqual\_namespace;

goto LDefault;

L0027:

// 0027 instanceof

if (luHash == 0x79C146FA) goto LEqual\_instanceof;

// 004F protected

if (luHash == 0x7679AA2A) goto LEqual\_protected;

goto LDefault;

L0005:

// 0005 return

if (luHash < 0x0A2FCE00) goto L000B;

if (luHash == 0x0A2FCE00) goto LEqual\_return;

// 000A ushort

if (luHash < 0x0A81B775) goto L0015;

if (luHash == 0x0A81B775) goto LEqual\_ushort;

// 0014 debugger

if (luHash < 0x2806F445) goto L0029;

if (luHash == 0x2806F445) goto LEqual\_debugger;

// 0028 synchronized

if (luHash == 0x3189C9E0) goto LEqual\_synchronized;

goto LDefault;

L0029:

// 0029 continue

if (luHash == 0x1EF7AD75) goto LEqual\_continue;

goto LDefault;

L0015:

// 0015 throws

if (luHash < 0x0A5EC897) goto L002B;

if (luHash == 0x0A5EC897) goto LEqual\_throws;

// 002A typeof

if (luHash == 0x0A7440A7) goto LEqual\_typeof;

goto LDefault;

L002B:

// 002B switch

if (luHash == 0x0A5B93A2) goto LEqual\_switch;

// 0057 static

if (luHash == 0x0A5727B8) goto LEqual\_static;

goto LDefault;

L000B:

// 000B ensure

if (luHash < 0x09218CF2) goto L0017;

if (luHash == 0x09218CF2) goto LEqual\_ensure;

// 0016 native

if (luHash < 0x09D3FE67) goto L002D;

if (luHash == 0x09D3FE67) goto LEqual\_native;

// 002C public

if (luHash == 0x0A17792F) goto LEqual\_public;

goto LDefault;

L002D:

// 002D import

if (luHash == 0x0976AFAB) goto LEqual\_import;

// 005B export

if (luHash == 0x092E0B32) goto LEqual\_export;

goto LDefault;

L0017:

// 0017 delete

if (luHash < 0x08FFD213) goto L002F;

if (luHash == 0x08FFD213) goto LEqual\_delete;

// 002E double

if (luHash == 0x090D396B) goto LEqual\_double;

goto LDefault;

L002F:

// 002F assert

if (luHash == 0x08D130F2) goto LEqual\_assert;

// 005F yield

if (luHash == 0x00A28D67) goto LEqual\_yield;

goto LDefault;

L0003:

// 0003 null

if (luHash < 0x0008CABB) goto L0007;

if (luHash == 0x0008CABB) goto LEqual\_null;

// 0006 false

if (luHash < 0x0089C56B) goto L000D;

if (luHash == 0x0089C56B) goto LEqual\_false;

// 000C short

if (luHash < 0x009AE070) goto L0019;

if (luHash == 0x009AE070) goto LEqual\_short;

// 0018 throw

if (luHash < 0x009C29E4) goto L0031;

if (luHash == 0x009C29E4) goto LEqual\_throw;

// 0030 ulong

if (luHash == 0x009DB965) goto LEqual\_ulong;

goto LDefault;

L0031:

// 0031 super

if (luHash == 0x009BDA2F) goto LEqual\_super;

goto LDefault;

L0019:

// 0019 float

if (luHash < 0x008A9AC6) goto L0033;

if (luHash == 0x008A9AC6) goto LEqual\_float;

// 0032 sbyte

if (luHash == 0x009A78A7) goto LEqual\_sbyte;

goto LDefault;

L0033:

// 0033 final

if (luHash == 0x008A600A) goto LEqual\_final;

// 0067 event

if (luHash == 0x008A0A02) goto LEqual\_event;

goto LDefault;

L000D:

// 000D with

if (luHash < 0x00096A6C) goto L001B;

if (luHash == 0x00096A6C) goto LEqual\_with;

// 001A class

if (luHash < 0x0086B966) goto L0035;

if (luHash == 0x0086B966) goto LEqual\_class;

// 0034 const

if (luHash == 0x008701A7) goto LEqual\_const;

goto LDefault;

L0035:

// 0035 catch

if (luHash == 0x0085FAA3) goto LEqual\_catch;

// 006B break

if (luHash == 0x0085E995) goto LEqual\_break;

// await

if (luHash == 0x0084FF56) goto LEqual\_await;

goto LDefault;

L001B:

// 001B uint

if (luHash < 0x000943B0) goto L0037;

if (luHash == 0x000943B0) goto LEqual\_uint;

// 0036 void

if (luHash == 0x00095D42) goto LEqual\_void;

goto LDefault;

L0037:

// 0037 true

if (luHash == 0x00093B10) goto LEqual\_true;

// 006F this

if (luHash == 0x00092F08) goto LEqual\_this;

goto LDefault;

L0007:

// 0007 try

if (luHash < 0x00008AFF) goto L000F;

if (luHash == 0x00008AFF) goto LEqual\_try;

// 000E byte

if (luHash < 0x0007E974) goto L001D;

if (luHash == 0x0007E974) goto LEqual\_byte;

// 001C goto

if (luHash < 0x00083E29) goto L0039;

if (luHash == 0x00083E29) goto LEqual\_goto;

// 0038 long

if (luHash == 0x00089DB0) goto LEqual\_long;

goto LDefault;

L0039:

// 0039 enum

if (luHash == 0x000816B5) goto LEqual\_enum;

// 0073 else

if (luHash == 0x00081449) goto LEqual\_else;

goto LDefault;

L001D:

// 001D case

if (luHash < 0x0007E17C) goto L003B;

if (luHash == 0x0007E17C) goto LEqual\_case;

// 003A char

if (luHash == 0x0007E83E) goto LEqual\_char;

goto LDefault;

L003B:

// 003B use

if (luHash == 0x00008C1D) goto LEqual\_use;

// 0077 var

if (luHash == 0x00008C19) goto LEqual\_var;

goto LDefault;

L000F:

// 000F get

if (luHash < 0x00007B70) goto L001F;

if (luHash == 0x00007B70) goto LEqual\_get;

// 001E new

if (luHash < 0x0000835A) goto L003D;

if (luHash == 0x0000835A) goto LEqual\_new;

// 003C set

if (luHash == 0x000088FC) goto LEqual\_set;

goto LDefault;

L003D:

// 003D let

if (luHash == 0x00008115) goto LEqual\_let;

// 007B int

if (luHash == 0x00007E4B) goto LEqual\_int;

goto LDefault;

L001F:

// 001F in

if (luHash < 0x00000767) goto L003F;

if (luHash == 0x00000767) goto LEqual\_in;

// 003E for

if (luHash == 0x00007AF7) goto LEqual\_for;

goto LDefault;

L003F:

// 003F if

if (luHash == 0x0000075F) goto LEqual\_if;

// 007F do

if (luHash == 0x00000713) goto LEqual\_do;

goto LDefault;

#endif // \_WIN32 && \_M\_IX86

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

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

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

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

// Generated by comphash.exe version 5.6.0.6202 on Tue Jan 27 21:03:14 2009

HASH\_NAME(eval, 0x00081E68, 0x00081E68)

HASH\_NAME(arguments, 0x0E470096, 0x0E470096)

HASH\_NAME(constructor, 0x46FD13C6, 0x46FD13C6)

HASH\_NAME(prototype, 0x7687B556, 0x7687B556)

HASH\_NAME(abstract, 0xDBC60B24, 0xDBC60B24)

HASH\_NAME(assert, 0x08D130F2, 0x08D130F2)

HASH\_NAME(async, 0x0084CDEE, 0x0084CDEE)

HASH\_NAME(await, 0x0084FF56, 0x0084FF56)

HASH\_NAME(boolean, 0x96F94400, 0x96F94400)

HASH\_NAME(break, 0x0085E995, 0x0085E995)

HASH\_NAME(byte, 0x0007E974, 0x0007E974)

HASH\_NAME(case, 0x0007E17C, 0x0007E17C)

HASH\_NAME(catch, 0x0085FAA3, 0x0085FAA3)

HASH\_NAME(char, 0x0007E83E, 0x0007E83E)

HASH\_NAME(class, 0x0086B966, 0x0086B966)

HASH\_NAME(const, 0x008701A7, 0x008701A7)

HASH\_NAME(continue, 0x1EF7AD75, 0x1EF7AD75)

HASH\_NAME(debugger, 0x2806F445, 0x2806F445)

HASH\_NAME(decimal, 0x98F1BFFF, 0x98F1BFFF)

HASH\_NAME(default, 0x98F50305, 0x98F50305)

HASH\_NAME(delete, 0x08FFD213, 0x08FFD213)

HASH\_NAME(do, 0x00000713, 0x00000713)

HASH\_NAME(double, 0x090D396B, 0x090D396B)

HASH\_NAME(else, 0x00081449, 0x00081449)

HASH\_NAME(enum, 0x000816B5, 0x000816B5)

HASH\_NAME(ensure, 0x09218CF2, 0x09218CF2)

HASH\_NAME(event, 0x008A0A02, 0x008A0A02)

HASH\_NAME(export, 0x092E0B32, 0x092E0B32)

HASH\_NAME(extends, 0x9C13124B, 0x9C13124B)

HASH\_NAME(false, 0x0089C56B, 0x0089C56B)

HASH\_NAME(final, 0x008A600A, 0x008A600A)

HASH\_NAME(finally, 0x9C3672EF, 0x9C3672EF)

HASH\_NAME(float, 0x008A9AC6, 0x008A9AC6)

HASH\_NAME(for, 0x00007AF7, 0x00007AF7)

HASH\_NAME(function, 0x70E47CD6, 0x70E47CD6)

HASH\_NAME(get, 0x00007B70, 0x00007B70)

HASH\_NAME(goto, 0x00083E29, 0x00083E29)

HASH\_NAME(if, 0x0000075F, 0x0000075F)

HASH\_NAME(implements, 0x856AF30E, 0x856AF30E)

HASH\_NAME(import, 0x0976AFAB, 0x0976AFAB)

HASH\_NAME(in, 0x00000767, 0x00000767)

HASH\_NAME(instanceof, 0x79C146FA, 0x79C146FA)

HASH\_NAME(int, 0x00007E4B, 0x00007E4B)

HASH\_NAME(interface, 0xBC1E9451, 0xBC1E9451)

HASH\_NAME(internal, 0xB0B68A3D, 0xB0B68A3D)

HASH\_NAME(invariant, 0xBEA8C42C, 0xBEA8C42C)

HASH\_NAME(long, 0x00089DB0, 0x00089DB0)

HASH\_NAME(namespace, 0x9309F69D, 0x9309F69D)

HASH\_NAME(native, 0x09D3FE67, 0x09D3FE67)

HASH\_NAME(new, 0x0000835A, 0x0000835A)

HASH\_NAME(null, 0x0008CABB, 0x0008CABB)

HASH\_NAME(package, 0xA9DEE87C, 0xA9DEE87C)

HASH\_NAME(private, 0xAB57B15B, 0xAB57B15B)

HASH\_NAME(protected, 0x7679AA2A, 0x7679AA2A)

HASH\_NAME(public, 0x0A17792F, 0x0A17792F)

HASH\_NAME(require, 0xAD28D1BD, 0xAD28D1BD)

HASH\_NAME(return, 0x0A2FCE00, 0x0A2FCE00)

HASH\_NAME(sbyte, 0x009A78A7, 0x009A78A7)

HASH\_NAME(set, 0x000088FC, 0x000088FC)

HASH\_NAME(short, 0x009AE070, 0x009AE070)

HASH\_NAME(static, 0x0A5727B8, 0x0A5727B8)

HASH\_NAME(super, 0x009BDA2F, 0x009BDA2F)

HASH\_NAME(switch, 0x0A5B93A2, 0x0A5B93A2)

HASH\_NAME(synchronized, 0x3189C9E0, 0x3189C9E0)

HASH\_NAME(this, 0x00092F08, 0x00092F08)

HASH\_NAME(throw, 0x009C29E4, 0x009C29E4)

HASH\_NAME(throws, 0x0A5EC897, 0x0A5EC897)

HASH\_NAME(transient, 0xE10C0168, 0xE10C0168)

HASH\_NAME(true, 0x00093B10, 0x00093B10)

HASH\_NAME(try, 0x00008AFF, 0x00008AFF)

HASH\_NAME(typeof, 0x0A7440A7, 0x0A7440A7)

HASH\_NAME(uint, 0x000943B0, 0x000943B0)

HASH\_NAME(ulong, 0x009DB965, 0x009DB965)

HASH\_NAME(use, 0x00008C1D, 0x00008C1D)

HASH\_NAME(ushort, 0x0A81B775, 0x0A81B775)

HASH\_NAME(var, 0x00008C19, 0x00008C19)

HASH\_NAME(void, 0x00095D42, 0x00095D42)

HASH\_NAME(volatile, 0xEF691640, 0xEF691640)

HASH\_NAME(while, 0x009FF239, 0x009FF239)

HASH\_NAME(with, 0x00096A6C, 0x00096A6C)

HASH\_NAME(let, 0x00008115, 0x00008115)

HASH\_NAME(yield, 0x00A28D67, 0x00A28D67)

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

// 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"

namespace UnifiedRegex

{

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

// Trigrams

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

TrigramInfo::TrigramInfo(\_\_in\_ecount(PatternLength) char\* pat1,\_\_in\_ecount(PatternLength) char\* pat2, Recycler\* recycler)

{

isTrigramPattern=true;

hasCachedResultString = false;

int k;

triPat1=0;

triPat2=0;

resultCount=0;

for (k=3;k<PatternLength;k++) {

triPat1=(triPat1<<4)+pat1[k];

triPat2=(triPat2<<4)+pat2[k];

}

}

void TrigramAlphabet::InitTrigramMap() {

input=NULL;

// set up mapping from 9 bits to trigram

for (int i=0;i<TrigramMapSize;i++) {

int t1=i>>6;

int t2=(i>>3)&0x7;

int t3=i&0x7;

if ((t1>=AlphaCount)||(t2>=AlphaCount)||(t3>=AlphaCount)) {

trigramMap[i]=TrigramNotInPattern;

}

else {

// number of trigram

trigramMap[i]=(char)((t1<<4)+(t2<<2)+t3);

}

}

for (int j=0;j<TrigramCount;j++) {

trigramStarts[j].count=0;

}

}

bool TrigramAlphabet::AddStarts(\_\_in\_xcount(TrigramInfo::PatternLength) char\* pat1,\_\_in\_xcount(TrigramInfo::PatternLength) char\* pat2, RegexPattern\* pattern)

{

for (int k=0;k<TrigramCount;k++) {

char t1=1<<(k>>4);

char t2=1<<((k>>2)&0x3);

char t3=1<<(k&0x3);

if ((t1&pat1[0])&&(t2&pat1[1])&&(t3&pat1[2])) {

if ((t1&pat2[0])&&(t2&pat2[1])&&(t3&pat2[2])) {

return false;

}

else {

TrigramStart\* trigramStart=(&trigramStarts[k]);

if (trigramStart->count>=TrigramStart::MaxPatPerStart) {

return false;

}

else {

PatternTri\* tri= &(trigramStart->patterns[trigramStart->count++]);

tri->pattern=pattern;

tri->encodedPattern=pattern->rep.unified.trigramInfo->triPat1;

}

}

}

else if ((t1&pat2[0])&&(t2&pat2[1])&&(t3&pat2[2])) {

TrigramStart\* trigramStart=(&trigramStarts[k]);

if (trigramStart->count>=TrigramStart::MaxPatPerStart) {

return false;

}

else {

PatternTri\* tri= &(trigramStart->patterns[trigramStart->count++]);

tri->pattern=pattern;

tri->encodedPattern=pattern->rep.unified.trigramInfo->triPat2;

}

}

}

return true;

}

void TrigramAlphabet::MegaMatch(\_\_in\_ecount(inputLen) const wchar\_t\* input,int inputLen) {

this->input=input;

this->inputLen=inputLen;

if (inputLen<TrigramInfo::PatternLength) {

return;

}

// prime the pump

unsigned char c1=alphaBits[input[0]&UpperCaseMask];

unsigned char c2=alphaBits[input[1]&UpperCaseMask];

unsigned char c3=alphaBits[input[2]&UpperCaseMask];

// pump

for (int k=3;k<inputLen-5;k++) {

int index=(c1<<6)+(c2<<3)+c3;

if (index<TrigramMapSize) {

int t=trigramMap[index];

if (t!=TrigramNotInPattern) {

int count=trigramStarts[t].count;

if (count>0) {

int inputMask=0;

bool validInput=true;

for (int j=0;j<5;j++) {

// ascii check

if (input[k+j]<128) {

int bits=alphaBits[input[k+j]&UpperCaseMask];

if (bits==BitsNotInAlpha) {

validInput=false;

break;

}

inputMask=(inputMask<<AlphaCount)+(1<<bits);

}

else {

validInput=false;

break;

}

}

if (validInput) {

for (int j=0;j<count;j++) {

PatternTri\* tri= &(trigramStarts[t].patterns[j]);

if ((inputMask&(tri->encodedPattern))==inputMask) {

if (tri->pattern->rep.unified.trigramInfo->resultCount<TrigramInfo::MaxResults) {

tri->pattern->rep.unified.trigramInfo->offsets[tri->pattern->rep.unified.trigramInfo->resultCount++]=k-3;

}

else {

tri->pattern->rep.unified.trigramInfo->isTrigramPattern=false;

}

}

}

}

}

}

}

c1=c2;

c2=c3;

c3=alphaBits[input[k]&UpperCaseMask];

}

}

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

// OctoquadIdentifier

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

bool OctoquadIdentifier::Qualifies(const Program \*const program)

{

return (program->flags & (GlobalRegexFlag | IgnoreCaseRegexFlag)) == (GlobalRegexFlag | IgnoreCaseRegexFlag);

}

OctoquadIdentifier::OctoquadIdentifier(

const int numCodes,

char (&codeToChar)[TrigramAlphabet::AlphaCount],

char (&charToCode)[TrigramAlphabet::AsciiTableSize])

: numCodes(numCodes),

codeToChar(codeToChar),

charToCode(charToCode),

currPatternLength(0),

currPatternNum(-1)

{

// 'patternBits' will be initialized as necessary

}

int OctoquadIdentifier::GetOrAddCharCode(const Char c)

{

if (c >= static\_cast<Char>('A') && c <= static\_cast<Char>('Z'))

{

for (int i = 0; i < numCodes; i++)

{

if (codeToChar[i] == static\_cast<char>(c))

return i;

}

if (numCodes == TrigramAlphabet::AlphaCount)

return -1;

codeToChar[numCodes] = static\_cast<char>(c);

charToCode[c] = static\_cast<char>(numCodes);

return numCodes++;

}

else

return -1;

}

bool OctoquadIdentifier::BeginConcat()

{

if (currPatternNum >= 0 && currPatternLength != TrigramInfo::PatternLength)

return false;

if (currPatternNum >= NumPatterns)

return false;

currPatternNum++;

currPatternLength = 0;

return true;

}

bool OctoquadIdentifier::CouldAppend(const CharCount n) const

{

return n <= static\_cast<CharCount>(TrigramInfo::PatternLength - currPatternLength);

}

bool OctoquadIdentifier::AppendChar(Char c)

{

if (currPatternLength >= TrigramInfo::PatternLength || currPatternNum < 0 || currPatternNum >= NumPatterns)

return false;

int code = GetOrAddCharCode(c);

if (code < 0)

return false;

patternBits[currPatternNum][currPatternLength++] = 1 << code;

return true;

}

bool OctoquadIdentifier::BeginUnions()

{

if(currPatternLength >= TrigramInfo::PatternLength || currPatternNum < 0 || currPatternNum >= NumPatterns)

return false;

patternBits[currPatternNum][currPatternLength] = 0;

return true;

}

bool OctoquadIdentifier::UnionChar(Char c)

{

if (currPatternLength >= TrigramInfo::PatternLength || currPatternNum < 0 || currPatternNum >= NumPatterns)

return false;

int code = GetOrAddCharCode(c);

if (code < 0)

return false;

patternBits[currPatternNum][currPatternLength] |= 1 << code;

return true;

}

void OctoquadIdentifier::EndUnions()

{

Assert(currPatternLength < TrigramInfo::PatternLength);

++currPatternLength;

}

bool OctoquadIdentifier::IsOctoquad()

{

return

numCodes == TrigramAlphabet::AlphaCount &&

currPatternLength == TrigramInfo::PatternLength &&

currPatternNum == NumPatterns - 1;

}

void OctoquadIdentifier::SetTrigramAlphabet(Js::ScriptContext \* scriptContext,

\_\_in\_xcount(regex::TrigramAlphabet::AlphaCount) char\* alpha

, \_\_in\_xcount(regex::TrigramAlphabet::AsciiTableSize) char\* alphaBits)

{

ArenaAllocator\* alloc = scriptContext->RegexAllocator();

TrigramAlphabet \* trigramAlphabet = AnewStruct(alloc, UnifiedRegex::TrigramAlphabet);

for (uint i = 0; i < UnifiedRegex::TrigramAlphabet::AsciiTableSize; i++) {

trigramAlphabet->alphaBits[i] = UnifiedRegex::TrigramAlphabet::BitsNotInAlpha;

}

for (uint i = 0; i < UnifiedRegex::TrigramAlphabet::AlphaCount; i++) {

trigramAlphabet->alpha[i] = alpha[i];

trigramAlphabet->alphaBits[alpha[i]] = alphaBits[alpha[i]];

}

trigramAlphabet->InitTrigramMap();

scriptContext->SetTrigramAlphabet(trigramAlphabet);

}

void OctoquadIdentifier::InitializeTrigramInfo(Js::ScriptContext\* scriptContext, RegexPattern\* const pattern)

{

if(!scriptContext->GetTrigramAlphabet())

{

this->SetTrigramAlphabet(scriptContext, codeToChar, charToCode);

}

const auto recycler = scriptContext->GetRecycler();

pattern->rep.unified.trigramInfo = RecyclerNew(recycler, TrigramInfo, patternBits[0], patternBits[1], recycler);

pattern->rep.unified.trigramInfo->isTrigramPattern =

scriptContext->GetTrigramAlphabet()->AddStarts(patternBits[0], patternBits[1], pattern);

}

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

// OctoquadMatcher

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

OctoquadMatcher::OctoquadMatcher(const StandardChars<Char>\* standardChars, CaseInsensitive::MappingSource mappingSource, OctoquadIdentifier\* identifier)

{

for (int i = 0; i < TrigramAlphabet::AlphaCount; i++)

codeToChar[i] = (Char)identifier->codeToChar[i];

for (int i = 0; i < TrigramAlphabet::AsciiTableSize; i++)

charToBits[i] = 0;

for (int i = 0; i < TrigramAlphabet::AlphaCount; i++)

{

Char equivs[CaseInsensitive::EquivClassSize];

standardChars->ToEquivs(mappingSource, codeToChar[i], equivs);

for (int j = 0; j < CaseInsensitive::EquivClassSize; j++)

{

if (CTU(equivs[j]) < TrigramAlphabet::AsciiTableSize)

charToBits[CTU(equivs[j])] = 1 << i;

}

}

for (int i = 0; i < OctoquadIdentifier::NumPatterns; i++)

{

patterns[i] = 0;

for (int j = 0; j < TrigramInfo::PatternLength; j++)

{

patterns[i] <<= 4;

patterns[i] |= (uint32)identifier->patternBits[i][j];

}

}

}

OctoquadMatcher \*OctoquadMatcher::New(

Recycler\* recycler,

const StandardChars<Char>\* standardChars,

CaseInsensitive::MappingSource mappingSource,

OctoquadIdentifier\* identifier)

{

return RecyclerNewLeaf(recycler, OctoquadMatcher, standardChars, mappingSource, identifier);

}

// It exploits the fact that each quad of bits has at most only one bit set.

\_\_inline bool oneBitSetInEveryQuad(uint32 x)

{

x -= 0x11111111;

return (x & 0x88888888u) == 0;

}

bool OctoquadMatcher::Match

( const Char\* const input

, const CharCount inputLength

, CharCount& offset

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, RegexStats\* stats

#endif

)

{

Assert(TrigramInfo::PatternLength == 8);

Assert(OctoquadIdentifier::NumPatterns == 2);

if (inputLength < TrigramInfo::PatternLength)

return false;

if (offset > inputLength - TrigramInfo::PatternLength)

return false;

uint32 v = 0;

for (int i = 0; i < TrigramInfo::PatternLength; i++)

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

if (stats != 0)

stats->numCompares++;

#endif

v <<= 4;

if (CTU(input[offset + i]) < TrigramAlphabet::AsciiTableSize)

v |= charToBits[CTU(input[offset + i])];

}

const uint32 lp = patterns[0];

const uint32 rp = patterns[1];

CharCount next = offset + TrigramInfo::PatternLength;

while (true)

{

if (oneBitSetInEveryQuad(v & lp) || oneBitSetInEveryQuad(v & rp))

{

offset = next - TrigramInfo::PatternLength;

return true;

}

if (next >= inputLength)

return false;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

if (stats != 0)

stats->numCompares++;

#endif

v <<= 4;

if (CTU(input[next]) < TrigramAlphabet::AsciiTableSize)

v |= charToBits[CTU(input[next])];

next++;

}

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void OctoquadMatcher::Print(DebugWriter\* w) const

{

for (int i = 0; i < OctoquadIdentifier::NumPatterns; i++)

{

if (i > 0)

w->Print(L"|");

for (int j = 0; j < TrigramInfo::PatternLength; j++)

{

uint8 v = (patterns[i] >> ((TrigramInfo::PatternLength - j - 1) \* TrigramAlphabet::AlphaCount)) & 0xf;

int n = 0;

uint8 x = v;

while (x > 0)

{

x &= x-1;

n++;

}

if (n != 1)

w->Print(L"[");

for (int k = 0; k < TrigramAlphabet::AlphaCount; k++)

{

if ((v & 1) == 1)

w->PrintEscapedChar(codeToChar[k]);

v >>= 1;

}

if (n != 1)

w->Print(L"]");

}

}

}

#endif

}

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

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

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

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

//

// Matchers for pattern of form:

// pattern ::= atom{8} '|' atom{8}

// atom ::= A | [...charset drawn from A's...]

// where:

// - A is a set of exactly four ASCII characters

// - The pattern ignores case

// - The pattern includes the global flag

// An example pattern would be "ABCdABCd|aDcAbBcD".

#pragma once

namespace UnifiedRegex

{

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

// Trigrams

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

struct TrigramInfo {

static const int PatternLength=8;

static const int MaxResults=32;

bool isTrigramPattern;

bool hasCachedResultString;

int triPat1;

int triPat2;

int resultCount;

int offsets[MaxResults];

Js::JavascriptString \* cachedResult[MaxResults];

TrigramInfo(\_\_in\_ecount(PatternLength) char\* pat1,\_\_in\_ecount(PatternLength) char\* pat2, Recycler\* recycler);

};

struct PatternTri {

RegexPattern\* pattern;

int encodedPattern;

};

struct TrigramStart {

static const int MaxPatPerStart=12;

int count;

PatternTri patterns[MaxPatPerStart];

};

struct TrigramAlphabet {

static const int AlphaCount=4;

static const int AsciiTableSize=128;

static const int BitsNotInAlpha=4;

static const int TrigramMapSize=221;

static const int TrigramNotInPattern=65;

static const char LowerCaseBit=0x20;

static const char UpperCaseMask=0x5f;

static const int TrigramCount=64;

static const int MaxCachedStarts=48;

TrigramStart trigramStarts[TrigramCount];

char alpha[AlphaCount];

char alphaBits[AsciiTableSize];

char trigramMap[TrigramMapSize];

const wchar\_t\* input;

int inputLen;

void InitTrigramMap();

bool AddStarts(\_\_in\_xcount(TrigramInfo::PatternLength) char\* pat1,\_\_in\_xcount(TrigramInfo::PatternLength) char\* pat2, RegexPattern\* pattern);

void MegaMatch(\_\_in\_ecount(inputLen) const wchar\_t\* input,int inputLen);

};

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

// OctoquadIdentifier

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

class OctoquadIdentifier : private Chars<wchar\_t>

{

friend class OctoquadMatcher;

public:

static const int NumPatterns = 2;

private:

// Number of characters in the alphabet encountered so far

int numCodes;

// Maps a character code to the character

char (&codeToChar)[TrigramAlphabet::AlphaCount];

// Maps a character to its code 0-3. This array is passed into the constructor and only indexes for characters in the

// alphabet are updated.

char (&charToCode)[TrigramAlphabet::AsciiTableSize];

// For each octoquad pattern, each byte contains a 4-bit pattern. One character will be represented as 0x1, 0x2, 0x4, or

// 0x8 since it's a quad alphabet. A character class in the pattern can cause the bit pattern to be a combination of the

// character bits.

char patternBits[NumPatterns][TrigramInfo::PatternLength];

int currPatternLength;

int currPatternNum;

void SetTrigramAlphabet(Js::ScriptContext \* scriptContext,

\_\_in\_xcount(regex::TrigramAlphabet::AlphaCount) char\* alpha,

\_\_in\_xcount(regex::TrigramAlphabet::AsciiTableSize) char\* alphaBits);

public:

static bool Qualifies(const Program \*const program);

OctoquadIdentifier(

const int numCodes,

char (&codeToChar)[TrigramAlphabet::AlphaCount],

char (&charToCode)[TrigramAlphabet::AsciiTableSize]);

// Returns -1 if character not in quad alphabet and the alphabet is full

int GetOrAddCharCode(const Char c);

bool BeginConcat();

bool CouldAppend(const CharCount n) const;

bool AppendChar(Char c);

bool BeginUnions();

bool UnionChar(Char c);

void EndUnions();

bool IsOctoquad();

void InitializeTrigramInfo(Js::ScriptContext\* scriptContext, RegexPattern\* const pattern);

};

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

// OctoquadMatcher

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

class OctoquadMatcher : private Chars<wchar\_t>

{

private:

OctoquadMatcher(const StandardChars<Char>\* standardChars, CaseInsensitive::MappingSource mappingSource, OctoquadIdentifier\* identifier);

Char codeToChar[TrigramAlphabet::AlphaCount];

// Maps characters (0..AsciTableSize-1) to 0 if not in alphabet, or 0x1, 0x2, 0x4 or 0x8.

// Allocated and filled only if invoke Match below.

uint8 charToBits[TrigramAlphabet::AsciiTableSize];

uint32 patterns[OctoquadIdentifier::NumPatterns];

public:

static OctoquadMatcher \*New(Recycler\* recycler, const StandardChars<Char>\* standardChars, CaseInsensitive::MappingSource mappingSource, OctoquadIdentifier\* identifier);

bool Match

( const Char\* const input

, const CharCount inputLength

, CharCount& offset

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, RegexStats\* stats

#endif

);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w) const;

#endif

};

}

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

// 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;

}

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

// 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.

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

#pragma once

// Parse flags

enum

{

fscrNil = 0,

fscrHtmlComments = 1 << 0, // throw away html style comments

fscrReturnExpression = 1 << 1, // call should return the last expression

fscrImplicitThis = 1 << 2, // 'this.' is optional (for Call)

fscrImplicitParents = 1 << 3, // the parents of 'this' are implicit

fscrMapQuote = 1 << 4, // map single quote to double quote

fscrDynamicCode = 1 << 5, // The code is being generated dynamically (eval, new Function, etc.)

fscrSyntaxColor = 1 << 6, // used by the scanner for syntax coloring

fscrNoImplicitHandlers = 1 << 7, // same as Opt NoConnect at start of block

// prevents the need to make a copy to strip off trailing html comments

// - modifies the behavior of fscrHtmlComments

fscrDoNotHandleTrailingHtmlComments = 1 << 8,

#if DEBUG

fscrEnforceJSON = 1 << 9, // used together with fscrReturnExpression

// enforces JSON semantics in the parsing.

#endif

fscrEval = 1 << 10, // this expression has eval semantics (i.e., run in caller's context

fscrEvalCode = 1 << 11, // this is an eval expression

fscrGlobalCode = 1 << 12, // this is a global script

fscrDeferFncParse = 1 << 13, // parser: defer creation of AST's for non-global code

fscrDeferredFncExpression = 1 << 14, // the function decl node we deferred is an expression,

// i.e., not a declaration statement

fscrDeferredFnc = 1 << 15, // the function we are parsing is deferred

fscrNoPreJit = 1 << 16, // ignore prejit global flag

fscrAllowFunctionProxy = 1 << 17, // Allow creation of function proxies instead of function bodies

fscrIsLibraryCode = 1 << 18, // Current code is engine library code written in Javascript

fscrNoDeferParse = 1 << 19, // Do not defer parsing

// Unused = 1 << 20,

#ifdef IR\_VIEWER

fscrIrDumpEnable = 1 << 21, // Allow parseIR to generate an IR dump

#endif /\* IRVIEWER \*/

// Throw a ReferenceError when the global 'this' is used (possibly in a lambda),

// for debugger when broken in a lambda that doesn't capture 'this'

fscrDebuggerErrorOnGlobalThis = 1 << 22,

fscrDeferredClassMemberFnc = 1 << 23,

fscrConsoleScopeEval = 1 << 24, // The eval string is console eval or debugEval, used to have top level

// let/const in global scope instead of eval scope so that they can be preserved across console inputs

fscrNoAsmJs = 1 << 25, // Disable generation of asm.js code

fscrAll = (1 << 26) - 1

};

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

// 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

#define REGEX\_TRIGRAMS 1

#include "Common.h"

// FORWARD

namespace Js

{

class ScriptContext;

class JavascriptString;

}

namespace UnifiedRegex {

struct RegexPattern;

struct Program;

template <typename T> class StandardChars;

typedef StandardChars<uint8> UTF8StandardChars;

typedef StandardChars<wchar\_t> UnicodeStandardChars;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

class DebugWriter;

struct RegexStats;

class RegexStatsDatabase;

#endif

}

#include "ParserCommon.h"

#include "alloc.h"

#include "cmperr.h"

#include "idiom.h"

#include "popcode.h"

#include "ptree.h"

#include "tokens.h"

#include "hash.h"

#include "CharClassifier.h"

#include "scan.h"

#include "screrror.h"

#include "rterror.h"

#include "parse.h"

#include "BackgroundParser.h"

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

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

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

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

// Common definitions used outside parser so that we don't have to include the whole Parser.h.

#pragma once

namespace Js

{

typedef int32 ByteCodeLabel; // Size of this match the offset size in layouts

typedef uint32 RegSlot;

typedef uint8 RegSlot\_OneByte;

typedef int8 RegSlot\_OneSByte;

typedef int16 RegSlot\_TwoSByte;

typedef uint16 RegSlot\_TwoByte;

}

enum ErrorTypeEnum

{

kjstError,

kjstEvalError,

kjstRangeError,

kjstReferenceError,

kjstSyntaxError,

kjstTypeError,

kjstURIError,

kjstCustomError,

#ifdef ENABLE\_PROJECTION

kjstWinRTError,

#endif

};

struct ParseNode;

typedef ParseNode \*ParseNodePtr;

//

// Below was moved from scrutil.h to share with chakradiag.

//

#define HR(sc) ((HRESULT)(sc))

#define MAKE\_HR(vbserr) (MAKE\_HRESULT(SEVERITY\_ERROR, FACILITY\_CONTROL, vbserr))

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

// 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"

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

// 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 <intsafe.h>

// Parser Includes

#include "Parser.h"

#include "keywords.h"

#include "globals.h"

#include "RegexCommon.h"

#include "DebugWriter.h"

#include "RegexStats.h"

#include "StandardChars.h"

#include "OctoquadIdentifier.h"

#include "RegexCompileTime.h"

#include "RegexParser.h"

#include "RegexPattern.h"

// Runtime includes

#include "..\Runtime\runtime.h"

#include "..\Runtime\ByteCode\Symbol.h"

#include "..\Runtime\ByteCode\Scope.h"

#include "..\Runtime\ByteCode\FuncInfo.h"

#include "..\Runtime\ByteCode\ScopeInfo.h"

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

// 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

#ifdef EDIT\_AND\_CONTINUE

namespace Js

{

class SyntaxEquivalenceBase;

template <class Allocator> class SyntaxEquivalence;

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

// TreeComparer for ParseNode TreeMatch.

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

template <class SubClass, class Allocator>

class ParseTreeComparer : public TreeComparerBase<SubClass, ParseNode>

{

private:

static const int TOKENLIST\_MAXDIFF\_SHIFT = 3; // Used to detect lists of significantly different lengths

SyntaxEquivalence<Allocator> syntaxEquivalence;

// 2 lists used in GetDistance. (Can mark isLeaf because they don't own the nodes.)

typedef JsUtil::List<PNode, Allocator, /\*isLeaf\*/true> NodeList;

NodeList leftList, rightList;

public:

ParseTreeComparer(Allocator\* alloc) :

syntaxEquivalence(alloc), leftList(alloc), rightList(alloc)

{}

ParseTreeComparer(const ParseTreeComparer& other) :

syntaxEquivalence(other.GetAllocator()), leftList(other.GetAllocator()), rightList(other.GetAllocator())

{}

Allocator\* GetAllocator() const

{

return leftList.GetAllocator();

}

int LabelCount() const

{

return ::OpCode::knopLim;

}

int GetLabel(PNode x) const

{

return x->nop;

}

PNode GetParent(PNode x) const

{

return x->parent;

}

template <class Func>

void MapChildren(PNode x, const Func& func) const

{

Js::MapChildren(x, func);

}

// Map (sub)tree nodes to compute distance. Child class can re-implement to control which nodes participate in

// distance computing.

template <class Func>

void MapTreeToComputeDistance(PNode x, const Func& func) const

{

pThis()->MapTree(x, func);

}

double GetDistance(PNode left, PNode right)

{

Assert(pThis()->GetLabel(left) == pThis()->GetLabel(right)); // Only called for nodes of same label

return ComputeValueDistance(left, right);

}

bool ValuesEqual(PNode oldNode, PNode newNode)

{

// This determines if we emit Update edit for matched nodes. If ValuesEqual, don't need update edit.

return !(syntaxEquivalence.IsToken(oldNode) || syntaxEquivalence.HasToken(oldNode))

|| syntaxEquivalence.AreEquivalent(oldNode, newNode);

}

private:

double ComputeValueDistance(PNode left, PNode right)

{

// If 2 nodes are equivalent trees, consider them exact match.

if (syntaxEquivalence.AreEquivalent(left, right))

{

return ExactMatchDistance;

}

double distance = ComputeDistance(left, right);

// We don't want to return an exact match, because there

// must be something different, since we got here

return (distance == ExactMatchDistance) ? EpsilonDistance : distance;

}

//

// Computer distance the same as Roslyn:

// \* For token nodes, use their string LCS distance.

// \* Otherwise, flatten the tree to get all tokens, use token list LCS distance.

//

// However, our parser are significantly different to Roslyn. Roslyn uses "full fidelity" parser,

// keeping every token scanned from source. e.g., "var a = 1" -> "var","a","=","1". Our parser keeps

// much less tokens. Thus our LCS distance will be quite different, which may affect diff accuracy.

//

double ComputeDistance(PNode left, PNode right)

{

// For token nodes, use their string LCS distance

if (syntaxEquivalence.IsToken(left))

{

return ComputeTokenDistance(left, right);

}

// Otherwise, flatten the tree to get all tokens, use token list LCS distance

Flatten(left, leftList);

Flatten(right, rightList);

// If token list lengths are significantly different, consider they are quite different.

{

int leftLen = leftList.Count();

int rightLen = rightList.Count();

int minLen = min(leftLen, rightLen);

int maxLen = max(leftLen, rightLen);

if (minLen < (maxLen >> TOKENLIST\_MAXDIFF\_SHIFT))

{

// Assuming minLen are all matched, distance > 0.875 (7/8). These two nodes shouldn't be a match.

return 1.0 - (double)minLen / (double)maxLen;

}

}

return ComputeLongestCommonSubsequenceDistance(GetAllocator(), leftList.Count(), rightList.Count(), [this](int indexA, int indexB)

{

return AreNodesTokenEquivalent(leftList.Item(indexA), rightList.Item(indexB));

});

}

// Flatten IsToken/HasToken nodes in the (sub)tree into given list to compute distance.

void Flatten(PNode root, NodeList& list)

{

list.Clear();

pThis()->MapTreeToComputeDistance(root, [&](PNode child)

{

if (syntaxEquivalence.IsToken(child) || syntaxEquivalence.HasToken(child))

{

list.Add(child);

}

});

}

// Check if IsToken/HasToken nodes are equivalent

bool AreNodesTokenEquivalent(PNode left, PNode right)

{

if (left->nop == right->nop)

{

return syntaxEquivalence.IsToken(left) ?

syntaxEquivalence.AreTokensEquivalent(left, right) : syntaxEquivalence.HaveEquivalentTokens(left, right);

}

return false;

}

double ComputeTokenDistance(PNode left, PNode right) const

{

Assert(syntaxEquivalence.IsToken(left));

switch (left->nop)

{

case knopName:

case knopStr:

return ComputeDistance(left->sxPid.pid, right->sxPid.pid);

case knopInt:

return left->sxInt.lw == right->sxInt.lw ? ExactMatchDistance : 1.0;

case knopFlt:

return left->sxFlt.dbl == right->sxFlt.dbl ? ExactMatchDistance : 1.0;

case knopRegExp: //TODO: sxPid.regexPattern

break;

}

// Other token nodes with fixed strings, e.g. "true", "null", always match exactly

return ExactMatchDistance;

}

// Compute distance of 2 PIDs as their string LCS distance

double ComputeDistance(IdentPtr left, IdentPtr right) const

{

Allocator\* alloc = leftList.GetAllocator();

return ComputeLongestCommonSubsequenceDistance(alloc, left->Cch(), right->Cch(), [=](int indexA, int indexB)

{

return left->Psz()[indexA] == right->Psz()[indexB];

});

}

};

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

// Function TreeComparer for TreeMatch at function level. View the parse tree as a hierarchy of functions.

// Ignore statement details.

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

template <class Allocator>

class FunctionTreeComparer : public ParseTreeComparer<FunctionTreeComparer<Allocator>, Allocator>

{

public:

FunctionTreeComparer(Allocator\* alloc) : ParseTreeComparer(alloc) {}

FunctionTreeComparer(const FunctionTreeComparer& other) : ParseTreeComparer(other) {}

// We only have 1 kind of node in this view -- FuncDecl

int LabelCount() const { return 1; }

int GetLabel(PNode x) const { return 0; }

PNode GetParent(PNode x) const

{

while (true)

{

x = \_\_super::GetParent(x);

if (!x || x->nop == knopFncDecl || x->nop == knopProg)

{

break;

}

}

return x;

}

template <class Func>

void MapChildren(PNode x, const Func& func) const

{

\_\_super::MapChildren(x, [&](PNode child)

{

if (child->nop == knopFncDecl)

{

func(child);

}

else

{

pThis()->MapChildren(child, func);

}

});

}

// To compute function node distance, only use their direct child nodes. Do not include descendant nodes

// under nested child functions.

template <class Func>

void MapTreeToComputeDistance(PNode x, const Func& func) const

{

func(x);

\_\_super::MapChildren(x, [&](PNode child)

{

if (child->nop == knopFncDecl)

{

func(child); // For child func, output the node itself but don't map its descendants

}

else

{

pThis()->MapTreeToComputeDistance(child, func); // recursive into other nodes

}

});

}

};

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

// Full TreeComparer for TreeMatch full parse tree. Used for test only.

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

template <class Allocator>

class FullTreeComparer : public ParseTreeComparer<FullTreeComparer<Allocator>, Allocator>

{

public:

FullTreeComparer(Allocator\* alloc) : ParseTreeComparer(alloc) {}

FullTreeComparer(const FullTreeComparer& other) : ParseTreeComparer(other) {}

};

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

// Visit every node of a parse (sub)tree in preorder. Delegates to Preorder/Postorder of PreorderContext.

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

template <class PreorderContext>

void ParseTreePreorder(ParseNode\* root, PreorderContext\* context)

{

class ParseTreePreorderVisitorPolicy : public VisitorPolicyBase<PreorderContext\*>

{

protected:

bool Preorder(ParseNode\* pnode, Context context) { context->Preorder(pnode); return true; }

void Postorder(ParseNode\* pnode, Context context) { context->Postorder(pnode); }

};

ParseNodeVisitor<ParseTreePreorderVisitorPolicy> visitor;

visitor.Visit(root, context);

}

template <class Func>

void ParseTreePreorder(ParseNode\* root, const Func& func)

{

class PreorderContext

{

private:

const Func& func;

public:

PreorderContext(const Func& func) : func(func) {}

void Preorder(ParseNode\* pnode) { func(pnode); }

void Postorder(ParseNode\* pnode) {}

};

PreorderContext context(func);

ParseTreePreorder(root, &context);

}

// TEMP: Consider setting parent at parse time. Temporarily traverse the whole tree to fix parent links.

template <class Allocator>

void FixParentLinks(ParseNodePtr root, Allocator\* alloc)

{

class FixAstParentVisitorContext

{

private:

JsUtil::Stack<ParseNodePtr, Allocator, /\*isLeaf\*/true> stack;

public:

FixAstParentVisitorContext(Allocator\* alloc) : stack(alloc) {};

void Preorder(ParseNode\* pnode)

{

pnode->parent = !stack.Empty() ? stack.Top() : nullptr;

stack.Push(pnode);

}

void Postorder(ParseNode\* pnode)

{

Assert(pnode == stack.Peek());

stack.Pop();

}

};

FixAstParentVisitorContext fixAstParentVisitorContext(alloc);

ParseTreePreorder(root, &fixAstParentVisitorContext);

}

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

// Map child nodes of a parse node.

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

template <class Func>

void MapChildren(ParseNode\* pnode, const Func& func)

{

struct ChildrenWalkerPolicy : public WalkerPolicyBase<bool, const Func&>

{

ResultType WalkChildChecked(ParseNode \*pnode, Context context)

{

// Some of Walker code calls with null ParseNode. e.g., a for loop with null init child.

if (pnode)

{

context(pnode);

}

return true;

}

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

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

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

};

ParseNodeWalker<ChildrenWalkerPolicy> walker;

walker.Walk(pnode, func);

}

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

// Helpers for testing ParseNode equivalence

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

class SyntaxEquivalenceBase

{

public:

//

// Check if a node is a token node (leaf only, can never have child nodes). e.g., "123" (number literal).

//

static bool IsToken(ParseNode\* pnode)

{

// TODO: We may use a new flag fnopToken

return (ParseNode::Grfnop(pnode->nop) & fnopLeaf)

&& pnode->nop != knopFncDecl

&& pnode->nop != knopClassDecl;

}

//

// Check if a node has token (node type ownning an implicit token, e.g. "var x" (var declaration)).

//

static bool HasToken(ParseNode\* pnode)

{

// TODO: We may use a new flag fnopHasToken

return pnode->nop == knopVarDecl

|| pnode->nop == knopFncDecl; // TODO: other nodes with data

}

//

// Check if 2 IsToken nodes (of the same type) are equivalent.

//

static bool AreTokensEquivalent(ParseNodePtr left, ParseNodePtr right)

{

Assert(IsToken(left) && left->nop == right->nop);

switch (left->nop)

{

case knopName:

case knopStr:

return AreEquivalent(left->sxPid.pid, right->sxPid.pid);

case knopInt:

return left->sxInt.lw == right->sxInt.lw;

case knopFlt:

return left->sxFlt.dbl == right->sxFlt.dbl;

case knopRegExp:

//TODO: sxPid.regexPattern

break;

}

// Other tokens have fixed strings and are always equivalent, e.g. "true", "null"

return true;

}

//

// Check if 2 HasToken nodes (of the same type) have equivalent tokens.

//

static bool HaveEquivalentTokens(ParseNodePtr left, ParseNodePtr right)

{

Assert(HasToken(left) && left->nop == right->nop);

switch (left->nop)

{

case knopVarDecl:

return AreEquivalent(left->sxVar.pid, right->sxVar.pid);

case knopFncDecl:

return AreEquivalent(left->sxFnc.pid, right->sxFnc.pid);

//TODO: other nodes with data

}

Assert(false);

return false;

}

private:

// Test if 2 PIDs refer to the same text.

static bool AreEquivalent(IdentPtr pid1, IdentPtr pid2)

{

if (pid1 && pid2)

{

// Optimize: If we can have both trees (scanner/parser) share Ident dictionary, this can become pid1 == pid2.

return pid1->Hash() == pid2->Hash()

&& pid1->Cch() == pid2->Cch()

&& wcsncmp(pid1->Psz(), pid2->Psz(), pid1->Cch()) == 0;

}

// PIDs may be null, e.g. anonymous function declarations

return pid1 == pid2;

}

};

template <class Allocator>

class SyntaxEquivalence : public SyntaxEquivalenceBase

{

private:

// 2 stacks used during equivalence test. (Can mark isLeaf because they don't own the nodes.)

JsUtil::Stack<ParseNode\*, Allocator, /\*isLeaf\*/true> leftStack, rightStack;

public:

SyntaxEquivalence(Allocator\* alloc) : leftStack(alloc), rightStack(alloc)

{}

//

// Tests if 2 parse (sub)trees are equivalent.

//

bool AreEquivalent(ParseNode\* left, ParseNode\* right)

{

bool result;

if (TryTestEquivalenceFast(left, right, &result))

{

return result;

}

Reset(); // Clear possible remaining nodes in leftStack/rightStack

PushChildren(left, right);

while (!leftStack.Empty() && leftStack.Count() == rightStack.Count())

{

left = leftStack.Pop();

right = rightStack.Pop();

if (TryTestEquivalenceFast(left, right, &result))

{

if (!result)

{

return false;

}

}

else

{

PushChildren(left, right); // Sub-pair is ok, but need to compare children

}

}

return leftStack.Empty() && rightStack.Empty();

}

private:

void Reset()

{

leftStack.Clear();

rightStack.Clear();

}

void PushChildren(ParseNode\* left, ParseNode\* right)

{

Assert(leftStack.Count() == rightStack.Count());

MapChildren(left, [&](ParseNode\* child) { leftStack.Push(child); });

MapChildren(right, [&](ParseNode\* child) { rightStack.Push(child); });

}

//

// Try to test 2 nodes for equivalence. Return true if we can determine the pair equivalence.

// Otherwise return false, which means the pair test is ok but we need further child nodes comparison.

//

static bool TryTestEquivalenceFast(ParseNode\* left, ParseNode\* right, \_Out\_ bool\* result)

{

Assert(left && right);

if (left == right)

{

\*result = true; // Same node

return true;

}

if (left->nop != right->nop)

{

\*result = false; // Different node type

return true;

}

if (IsToken(left))

{

\*result = AreTokensEquivalent(left, right); // Token comparison suffices

return true;

}

if (HasToken(left) && !HaveEquivalentTokens(left, right))

{

\*result = false; // Different implicit tokens, e.g. "var x" vs "var y"

return true;

}

return false; // This pair is ok, but not sure about children

}

};

} // namespace Js

#endif

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

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

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

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

// parser error messages

// NOTE: the error numbers should not change from version to version.

// Error numbers MUST be sorted.

LSC\_ERROR\_MSG( 1001, ERRnoMemory , "Out of memory")

LSC\_ERROR\_MSG( 1002, ERRsyntax , "Syntax error")

LSC\_ERROR\_MSG( 1003, ERRnoColon , "Expected ':'")

LSC\_ERROR\_MSG( 1004, ERRnoSemic , "Expected ';'")

LSC\_ERROR\_MSG( 1005, ERRnoLparen , "Expected '('")

LSC\_ERROR\_MSG( 1006, ERRnoRparen , "Expected ')'")

LSC\_ERROR\_MSG( 1007, ERRnoRbrack , "Expected ']'")

LSC\_ERROR\_MSG( 1008, ERRnoLcurly , "Expected '{'")

LSC\_ERROR\_MSG( 1009, ERRnoRcurly , "Expected '}'")

LSC\_ERROR\_MSG( 1010, ERRnoIdent , "Expected identifier")

LSC\_ERROR\_MSG( 1011, ERRnoEq , "Expected '='")

LSC\_ERROR\_MSG( 1012, ERRnoSlash , "Expected '/'")

LSC\_ERROR\_MSG( 1013, ERRbadNumber , "Invalid number")

LSC\_ERROR\_MSG( 1014, ERRillegalChar , "Invalid character")

LSC\_ERROR\_MSG( 1015, ERRnoStrEnd , "Unterminated string constant")

LSC\_ERROR\_MSG( 1016, ERRnoCmtEnd , "Unterminated comment")

LSC\_ERROR\_MSG( 1018, ERRbadReturn , "'return' statement outside of function")

LSC\_ERROR\_MSG( 1019, ERRbadBreak , "Can't have 'break' outside of loop")

LSC\_ERROR\_MSG( 1020, ERRbadContinue , "Can't have 'continue' outside of loop")

LSC\_ERROR\_MSG( 1023, ERRbadHexDigit , "Expected hexadecimal digit")

LSC\_ERROR\_MSG( 1024, ERRnoWhile , "Expected 'while'")

LSC\_ERROR\_MSG( 1025, ERRbadLabel , "Label redefined")

LSC\_ERROR\_MSG( 1026, ERRnoLabel , "Label not found")

LSC\_ERROR\_MSG( 1027, ERRdupDefault , "'default' can only appear once in a 'switch' statement")

LSC\_ERROR\_MSG( 1028, ERRnoMemberIdent , "Expected identifier, string or number")

// RETIRED Cc no longer supported ;; LSC\_ERROR\_MSG( 1029, ERRnoCcEnd , "Expected '@end'")

// RETIRED Cc no longer supported ;; LSC\_ERROR\_MSG( 1030, ERRccOff , "Conditional compilation is turned off")

LSC\_ERROR\_MSG( 1031, ERRnotConst , "Expected constant")

// RETIRED Cc no longer supported ;; LSC\_ERROR\_MSG( 1032, ERRnoAt , "Expected '@'")

LSC\_ERROR\_MSG( 1033, ERRnoCatch , "Expected 'catch'")

LSC\_ERROR\_MSG( 1034, ERRnoVar , "Expected 'var'")

LSC\_ERROR\_MSG( 1035, ERRdanglingThrow , "'throw' must be followed by an expression on the same source line")

// RETIRED ECMACP removed ;; LSC\_ERROR\_MSG( 1036, ERRWithNotInCP , "'with' not available in the ECMA 327 Compact Profile")

LSC\_ERROR\_MSG( 1037, ERRES5NoWith , "'with' statements are not allowed in strict mode") // string 8

LSC\_ERROR\_MSG( 1038, ERRES5ArgSame , "Duplicate formal parameter names not allowed in strict mode") // string 9

LSC\_ERROR\_MSG( 1039, ERRES5NoOctal , "Octal numeric literals and escape characters not allowed in strict mode") // string 1

LSC\_ERROR\_MSG( 1041, ERREvalUsage , "Invalid usage of 'eval' in strict mode") // string 3

LSC\_ERROR\_MSG( 1042, ERRArgsUsage , "Invalid usage of 'arguments' in strict mode") // string 3

LSC\_ERROR\_MSG( 1045, ERRInvalidDelete , "Calling delete on expression not allowed in strict mode") //string 4

LSC\_ERROR\_MSG( 1046, ERRDupeObjLit , "Multiple definitions of a property not allowed in strict mode") //string 7

LSC\_ERROR\_MSG( 1047, ERRFncDeclNotSourceElement, "In strict mode, function declarations cannot be nested inside a statement or block. They may only appear at the top level or directly inside a function body.")

LSC\_ERROR\_MSG( 1048, ERRKeywordNotId , "The use of a keyword for an identifier is invalid")

LSC\_ERROR\_MSG( 1049, ERRFutureReservedWordNotId, "The use of a future reserved word for an identifier is invalid")

LSC\_ERROR\_MSG( 1050, ERRFutureReservedWordInStrictModeNotId, "The use of a future reserved word for an identifier is invalid. The identifier name is reserved in strict mode.")

LSC\_ERROR\_MSG( 1051, ERRSetterMustHaveOneArgument , "Setter functions must have one argument")

LSC\_ERROR\_MSG( 1052, ERRRedeclaration , "Let/Const redeclaration") // "var x; let x;" is also a redeclaration

LSC\_ERROR\_MSG( 1053, ERRUninitializedConst , "Const must be initialized")

LSC\_ERROR\_MSG( 1054, ERRDeclOutOfStmt , "Declaration outside statement context")

LSC\_ERROR\_MSG( 1055, ERRAssignmentToConst , "Assignment to const")

LSC\_ERROR\_MSG( 1056, ERRUnicodeOutOfRange , "Unicode escape sequence value is higher than 0x10FFFF")

LSC\_ERROR\_MSG( 1057, ERRInvalidSpreadUse , "Invalid use of the ... operator. Spread can only be used in call arguments or an array literal.")

LSC\_ERROR\_MSG( 1058, ERRInvalidSuper , "Invalid use of the 'super' keyword")

LSC\_ERROR\_MSG( 1059, ERRInvalidSuperScope , "The 'super' keyword cannot be used at global scope")

LSC\_ERROR\_MSG( 1060, ERRSuperInIndirectEval , "The 'super' keyword cannot be used in an indirect eval() call")

LSC\_ERROR\_MSG( 1061, ERRSuperInGlobalEval , "The 'super' keyword cannot be used in a globally scoped eval() call")

LSC\_ERROR\_MSG( 1062, ERRnoDArrow , "Expected '=>'")

LSC\_ERROR\_MSG( 1063, ERRInvalidCodePoint , "Invalid codepoint value in the escape sequence.")

LSC\_ERROR\_MSG( 1064, ERRMissingCurlyBrace , "Closing curly brace ('}') expected.")

LSC\_ERROR\_MSG( 1065, ERRRestLastArg, "The rest parameter must be the last parameter in a formals list.")

LSC\_ERROR\_MSG( 1066, ERRRestWithDefault, "The rest parameter cannot have a default intializer.")

LSC\_ERROR\_MSG( 1067, ERRUnexpectedEllipsis, "Unexpected ... operator")

LSC\_ERROR\_MSG( 1068, ERRDestructInit, "Destructuring declarations must have an initializer")

LSC\_ERROR\_MSG( 1069, ERRDestructRestLast, "Destructuring rest variables must be in the last position of the expression")

LSC\_ERROR\_MSG( 1070, ERRUnexpectedDefault, "Unexpected default initializer")

LSC\_ERROR\_MSG( 1071, ERRDestructNoOper, "Unexpected operator in destructuring expression")

LSC\_ERROR\_MSG( 1072, ERRDestructIDRef, "Destructuring expressions can only have identifier references")

LSC\_ERROR\_MSG( 1073, ERRYieldInTryCatchOrFinally, "'yield' expressions are not allowed in 'try', 'catch', or 'finally' blocks")

LSC\_ERROR\_MSG( 1074, ERRConstructorCannotBeGenerator, "Class constructor may not be a generator")

LSC\_ERROR\_MSG( 1075, ERRInvalidAssignmentTarget, "Invalid destructuring assignment target")

LSC\_ERROR\_MSG( 1076, ERRFormalSame, "Duplicate formal parameter names not allowed in this context")

LSC\_ERROR\_MSG( 1077, ERRDestructNotInit, "Destructuring declarations cannot have an initializer")

// 1078 -- removed

LSC\_ERROR\_MSG(1079, ERRInvalidNewTarget, "Invalid use of the 'new.target' keyword")

LSC\_ERROR\_MSG(1080, ERRForInNoInitAllowed, "for-in loop head declarations cannot have an initializer")

LSC\_ERROR\_MSG(1081, ERRForOfNoInitAllowed, "for-of loop head declarations cannot have an initializer")

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

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

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

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

#pragma once

template <class Context>

class ParseNodeMutatorBase

{

public:

typedef Context Context;

};

template <typename Mutator, typename TContext = typename Mutator::Context>

struct ParseNodeMutatingVisitor : public Mutator

{

public:

typedef TContext Context;

typedef typename Mutator::Context MutatorContext;

class MutatorWalkerPolicy

{

public:

typedef bool ResultType;

typedef struct

{

MutatorContext mutatorContext;

ParseNodeMutatingVisitor<Mutator, Context> \*mutator;

} Context;

inline bool DefaultResult() { return true; }

inline bool ContinueWalk(bool value) { return value; }

inline bool WalkNode(ParseNode \*node, Context context) { return true; }

inline bool WalkListNode(ParseNode \*node, Context context) { return true; }

inline bool WalkFirstChild(ParseNode \*&node, Context context) { return context.mutator->Mutate(node, context.mutatorContext); }

inline bool WalkSecondChild(ParseNode \*&node, Context context) { return context.mutator->Mutate(node, context.mutatorContext); }

inline bool WalkNthChild(ParseNode\* pnodeParent, ParseNode \*&node, Context context) { return context.mutator->Mutate(node, context.mutatorContext); }

};

// Warning: This contains an unsafe cast if TContext != Mutator::Context.

// If you use a non-default type parameter for TContext you must override this method with the safe version.

// This cast is in place because if TContext != Muator::Context this will not compile even thought it will

// not be used if it is overridden.

virtual MutatorContext GetMutatorContext(Context context) { return (MutatorContext)context; }

inline bool Preorder(ParseNode \*node, Context context)

{

MutatorWalkerPolicy::Context mutatorWalkerContext;

mutatorWalkerContext.mutatorContext = GetMutatorContext(context);

mutatorWalkerContext.mutator = this;

ParseNodeWalker<MutatorWalkerPolicy> walker;

return walker.Walk(node, mutatorWalkerContext);

}

inline void Inorder(ParseNode \*node, Context context) { }

inline void Midorder(ParseNode \*node, Context context) { }

inline void Postorder(ParseNode \*node, Context context) { }

inline void InList(ParseNode \*pnode, Context context) { }

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

};

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

// 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

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

enum EditKind

{

// No change.

None = 0,

// Node value was updated.

Update,

// Node was inserted.

Insert,

// Node was deleted.

Delete,

// Node changed parent.

Move,

// Node changed position within its parent. The parent nodes of the old node and the new node are matching.

Reorder,

};

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

// Calculates Longest Common Subsequence.

// This uses the basic version in

//

// EUGENE W. MYERS: An O(ND) Difference Algorithm and Its Variations

//

// The idea is that LCS is a dual problem of shortest path in an edit graph. The edit graph is a grid of lengthA

// columns and lengthB rows. A path starts from (0,0) and moves toward (lenghA, lengthB).

// - A horizontal move (i,j) -> (i+1,j) represents deleting A[i].

// - A vertical move (i,j) -> (i,j+1) represents inserting B[j].

// - A diagonal move (i,j) -> (i+1,j+1) represents a match, A[i] == B[j].

// Each diagonal move represents a match. We want more diagnonal moves. Let diagonal move cost 0, horizontal or

// vertical move each costs 1. The basic algorithm is a greedy algorthm to find a shortest path from (0,0) to

// (lengthA, lengthB).

//

// Terms:

// diagonal k: The diagnonal where x-y==k.

// d-path: A path starting from (0,0) with d number of horizontal or vertical moves. Or, its length is d (note

// that each horizontal/vertical move costs 1, diagnonal move costs 0).

//

// 0-path can only move along and end on diagonal 0.

// 1-path can only end on diagonal -1 or 1.

// d-path can end on diagonal [-d, -d+2, ..., d-2, d].

//

// The basic algorithm tries to find the smallest d, where there is a d-path reaches (lengthA, lengthB).

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

template <class Allocator>

class LongestCommonSubsequence

{

private:

// Stores d-path furthest reaching endpoints. They can be on diagonal [-d, -d+2, ..., d-2, d].

class EndPoints

{

private:

int d;

int x[]; // Stores x for endpoints on the (d+1) diagonals. y == x - k.

EndPoints(int d) : d(d)

{

}

public:

int getd() const

{

return d;

}

// Get x of furthest reaching endpoint on diagonal k.

int operator[](int k) const

{

Assert(k >= -d && k <= d && (d - k) % 2 == 0); // k must be in [-d, -d+2, ..., d-2, d]

int i = (k + d) / 2;

return x[i];

}

// Get x reference of furthest reaching endpoint on diagonal k.

int& operator[](int k)

{

Assert(k >= -d && k <= d && (d - k) % 2 == 0); // k must be in [-d, -d+2, ..., d-2, d]

int i = (k + d) / 2;

return x[i];

}

static EndPoints\* New(Allocator\* alloc, int d)

{

Assert(d >= 0);

return AllocatorNewPlusLeaf(Allocator, alloc, sizeof(int) \* (d + 1), EndPoints, d);

}

void Destroy(Allocator\* alloc)

{

AllocatorDeletePlusLeaf(Allocator, alloc, sizeof(int) \* (d + 1), this);

}

};

// Represents an EditGraph for finding LCS

class EditGraph

{

private:

typedef JsUtil::List<EndPoints\*, Allocator> EndPointsList;

EndPointsList m\_endPoints; // Stores endPoints for paths: -1, 0, 1, ..., d

int m\_diagonal; // The final diagonal found on d-path that reaches destination

// Add EndPoints storage for d-path

EndPoints\* AddEndPoints(int d)

{

int i = m\_endPoints.Add(nullptr);

EndPoints\* e = EndPoints::New(m\_endPoints.GetAllocator(), d);

m\_endPoints.Item(i, e);

return e;

}

public:

EditGraph(Allocator\* alloc) : m\_endPoints(alloc) {}

~EditGraph()

{

Allocator\* alloc = m\_endPoints.GetAllocator();

m\_endPoints.Map([=](int, EndPoints\* e)

{

if (e)

{

e->Destroy(alloc);

}

});

}

//

// This is the basic algorithm to find a shortest path in the edit graph from (0,0) to (lengthA, lengthB).

// We iterate through d=0,1,2,... to find smallest d, where one d-path reaches (lengthA, lengthB).

// - d-path must end on diagonal [-d, -d+2, ..., d-2, d]

// - A furthest reaching d-path on diagonal k is composed of a furthest reaching d-1 path on diagonal k-1 or k+1,

// followed by a vertical or horizontal move, followed by moving along diagonal k.

//

template <class ItemEquals>

void FindPath(int lengthA, int lengthB, const ItemEquals& equals)

{

Assert(m\_endPoints.Empty()); // Only support one FindPath

int maxD;

if (Int32Math::Add(lengthA, lengthB, &maxD) || maxD > INT\_MAX / 2) // Limits maxD to simplify overflow handling

{

Math::DefaultOverflowPolicy();

}

// Pre-add virtual path -1

{

EndPoints& pre = \*AddEndPoints(1);

pre[1] = 0;

}

bool found = false;

for (int d = 0; d <= maxD && !found; d++)

{

const EndPoints& v = \*m\_endPoints.Item(d); // d-1 path

EndPoints& cur = \*AddEndPoints(d); // d path

for (int k = -d; k <= d; k += 2)

{

const bool verticalMove = (k == -d || (k != d && v[k - 1] < v[k + 1]));

int x = verticalMove ? v[k + 1] : v[k - 1] + 1;

int y = x - k;

while (x < lengthA && y < lengthB && equals(x, y))

{

x++;

y++;

}

cur[k] = x; // furthest reaching end point

if (x == lengthA && y == lengthB)

{

m\_diagonal = k;

found = true;

break;

}

}

}

Assert(found);

}

template <class Func>

void MapEdits(const Func& map) const

{

// m\_endPoints contains endPoints for paths: -1, 0, 1, ..., d

int d = m\_endPoints.Count() - 2;

int k = m\_diagonal;

for (; d >= 0; d--)

{

const EndPoints& v = \*m\_endPoints.Item(d); // d-1 path

const EndPoints& cur = \*m\_endPoints.Item(d + 1); // d path

Assert(cur.getd() == d);

const bool verticalMove = (k == -d || (k != d && v[k - 1] < v[k + 1]));

int x0 = verticalMove ? v[k + 1] : v[k - 1] + 1;

int x = cur[k];

int y = x - k;

while (x > x0)

{

map(EditKind::Update, --x, --y);

}

if (verticalMove)

{

if (d > 0) // Don't emit virtual initial move from path -1 to 0

{

map(EditKind::Insert, -1, --y);

}

k++;

}

else

{

map(EditKind::Delete, --x, -1);

k--;

}

}

}

};

struct Edit

{

EditKind kind;

int indexA;

int indexB;

Edit() {}

Edit(EditKind kind, int indexA, int indexB) :

kind(kind), indexA(indexA), indexB(indexB)

{

Assert((kind == EditKind::Insert && indexA == -1 && indexB >= 0)

|| (kind == EditKind::Delete && indexA >= 0 && indexB == -1)

|| (kind == EditKind::Update && indexA >= 0 && indexB >= 0));

}

};

typedef JsUtil::List<Edit, Allocator, /\*isLeaf\*/true> EditList;

EditList m\_edits;

public:

template <class ItemEquals>

LongestCommonSubsequence(Allocator\* alloc, int lengthA, int lengthB, const ItemEquals& equals) :

m\_edits(alloc)

{

EditGraph graph(alloc);

graph.FindPath(lengthA, lengthB, equals);

graph.MapEdits([this](EditKind kind, int indexA, int indexB)

{

m\_edits.Add(Edit(kind, indexA, indexB));

});

}

template <class Func>

void MapEdits(const Func& map) const

{

for (int i = m\_edits.Count() - 1; i >= 0; i--)

{

const Edit& e = m\_edits.Item(i);

map(e.kind, e.indexA, e.indexB);

}

}

template <class Func>

void MapMatches(const Func& map) const

{

MapEdits([&](EditKind kind, int indexA, int indexB)

{

if (kind == EditKind::Update)

{

map(indexA, indexB);

}

});

}

};

//

// Returns a distance [0..1] of the specified sequences. The smaller distance the more of their elements match.

//

template <class Allocator, class ItemEquals>

double ComputeLongestCommonSubsequenceDistance(Allocator\* alloc, int lengthA, int lengthB, const ItemEquals& equals)

{

Assert(lengthA >= 0 && lengthB >= 0);

if (lengthA == 0 || lengthB == 0)

{

return (lengthA == lengthB) ? 0.0 : 1.0;

}

int lcsLength = 0;

LongestCommonSubsequence<Allocator> lcs(alloc, lengthA, lengthB, equals);

lcs.MapMatches([&](int, int)

{

++lcsLength;

});

return 1.0 - (double)lcsLength / (double)max(lengthA, lengthB);

}

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

// Base class for TreeComparers, used with TreeMatch. TreeComparers specify parse node details.

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

template <class SubClass, class Node>

struct TreeComparerBase

{

typedef Node Node;

typedef Node\* PNode;

static const double ExactMatchDistance;

static const double EpsilonDistance;

const SubClass\* pThis() const { return static\_cast<const SubClass\*>(this); }

SubClass\* pThis() { return static\_cast<SubClass\*>(this); }

// The number of distinct labels used in the tree.

int LabelCount() const { return 0; }

// Returns an integer label corresponding to the given node.

// Returned value must be within [0, LabelCount).

int GetLabel(PNode x) const { return 0; }

// Returns N > 0 if the node with specified label can't change its N-th ancestor node, zero otherwise.

// 1st ancestor is the node's parent node.

// 2nd ancestor is the node's grandparent node.

// etc.

int TiedToAncestor(int label) { return 0; }

// Calculates the distance [0..1] of two nodes.

// The more similar the nodes the smaller the distance.

//

// Used to determine whether two nodes of the same label match.

// Even if 0 is returned the nodes might be slightly different.

double GetDistance(PNode x, PNode y) const { return 0; }

// Returns true if the specified nodes have equal values.

// Called with matching nodes (oldNode, newNode).

// Return true if the values of the nodes are the same, or their difference is not important.

bool ValuesEqual(PNode oldNode, PNode newNode) const { return true; }

PNode GetParent(PNode x) const { return nullptr; }

bool TryGetParent(PNode x, \_Out\_ PNode\* p) const

{

\*p = pThis()->GetParent(x);

return \*p != nullptr;

}

PNode GetAncestor(PNode node, int level) const

{

while (level > 0)

{

node = pThis()->GetParent(node);

level--;

}

return node;

}

// Map children nodes of x

template <class Func>

void MapChildren(PNode x, const Func& func) const {}

// Map all descendant nodes of x (not including x itself)

template <class Func>

void MapDescendants(PNode x, const Func& func) const

{

pThis()->MapChildren(x, [&](PNode child)

{

func(child);

MapDescendants(child, func);

});

}

// Map every node in the (sub)tree x.

template <class Func>

void MapTree(PNode x, const Func& func) const

{

func(x);

pThis()->MapDescendants(x, func);

}

// Return true if specified nodes belong to the same tree. For debug only.

bool TreesEqual(PNode left, PNode right) const { return true; }

};

template <class SubClass, class Node> const double TreeComparerBase<SubClass, Node>::ExactMatchDistance = 0.0;

template <class SubClass, class Node> const double TreeComparerBase<SubClass, Node>::EpsilonDistance = 0.00001;

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

// Tree match algorithm, based on general algorithm described in

// Change Detection in Hierarchically Structured Information

// by Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom

//

// Derived from Roslyn implementation.

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

template <class TreeComparer, class Allocator>

class TreeMatch

{

public:

// ParseNodes are owned by Parser arena. Considered leaf here.

typedef typename TreeComparer::PNode PNode;

typedef JsUtil::List<PNode, Allocator, /\*isLeaf\*/true> NodeList;

typedef JsUtil::BaseDictionary<PNode, PNode, typename ForceLeafAllocator<Allocator>::AllocatorType> NodeMap;

private:

static const double ExactMatchDistance;

static const double EpsilonDistance;

static const double MatchingDistance1;

static const double MatchingDistance2;

static const double MatchingDistance3;

static const double MaxDistance;

Allocator\* alloc;

const PNode root1;

const PNode root2;

TreeComparer comparer;

NodeMap\* oneToTwo;

NodeMap\* twoToOne;

public:

TreeMatch(Allocator\* alloc, PNode root1, PNode root2, const TreeComparer& comparer = TreeComparer()) :

alloc(alloc), root1(root1), root2(root2), comparer(comparer)

{

const int labelCount = comparer.LabelCount();

// calculate chains (not including root node)

AutoAllocatorObjectArrayPtr<NodeList, Allocator> nodes1(AllocatorNewArrayZ(Allocator, alloc, NodeList\*, labelCount), labelCount, alloc);

AutoAllocatorObjectArrayPtr<NodeList, Allocator> nodes2(AllocatorNewArrayZ(Allocator, alloc, NodeList\*, labelCount), labelCount, alloc);

int count1 = CategorizeNodesByLabels(root1, labelCount, nodes1);

int count2 = CategorizeNodesByLabels(root2, labelCount, nodes2);

AutoAllocatorObjectPtr<NodeMap, Allocator> map1(AllocatorNew(Allocator, alloc, NodeMap, alloc, count1), alloc);

AutoAllocatorObjectPtr<NodeMap, Allocator> map2(AllocatorNew(Allocator, alloc, NodeMap, alloc, count2), alloc);

this->oneToTwo = map1;

this->twoToOne = map2;

ComputeMatch(nodes1, nodes2, labelCount);

// Succeeded. Detach local objects that are now owned by this instance.

map1.Detach();

map2.Detach();

}

~TreeMatch()

{

DeleteObject<Allocator>(alloc, oneToTwo);

DeleteObject<Allocator>(alloc, twoToOne);

}

const TreeComparer& Comparer() const { return comparer; }

PNode OldRoot() const { return root1; }

PNode NewRoot() const { return root2; }

bool HasPartnerInTree1(PNode node2) const

{

Assert(comparer.TreesEqual(node2, root2));

return twoToOne->ContainsKey(node2);

}

bool HasPartnerInTree2(PNode node1) const

{

Assert(comparer.TreesEqual(node1, root1));

return oneToTwo->ContainsKey(node1);

}

bool TryGetPartnerInTree1(PNode node2, PNode\* partner1) const

{

Assert(comparer.TreesEqual(node2, root2));

return twoToOne->TryGetValue(node2, partner1);

}

bool TryGetPartnerInTree2(PNode node1, PNode\* partner2) const

{

Assert(comparer.TreesEqual(node1, root1));

return oneToTwo->TryGetValue(node1, partner2);

}

bool Contains(PNode node1, PNode node2) const

{

Assert(comparer.TreesEqual(node2, root2));

PNode partner2;

return TryGetPartnerInTree2(node1, &partner2) && node2 == partner2;

}

private:

int CategorizeNodesByLabels(PNode root, int labelCount, \_Out\_writes\_(labelCount) NodeList\* nodes[])

{

int count = 0;

comparer.MapDescendants(root, [&](PNode node)

{

int label = comparer.GetLabel(node);

Assert(label >= 0 && label < labelCount);

NodeList\* list = nodes[label];

if (!list)

{

list = NodeList::New(alloc);

nodes[label] = list;

}

list->Add(node);

count++;

});

return count;

}

void ComputeMatch(\_In\_reads\_(labelCount) NodeList\* nodes1[], \_In\_reads\_(labelCount) NodeList\* nodes2[], int labelCount)

{

// Root nodes always match but they might have been added as knownMatches

if (!HasPartnerInTree2(root1))

{

Add(root1, root2);

}

// --- The original FastMatch algorithm ---

//

// For each leaf label l, and then for each internal node label l do:

// a) S1 := chain T1(l)

// b) S2 := chain T2(l)

// c) lcs := LCS(S1, S2, Equal)

// d) For each pair of nodes (x,y) in lcs add (x,y) to M.

// e) Pair unmatched nodes with label l as in Algorithm Match, adding matches to M:

// For each unmatched node x in T1, if there is an unmatched node y in T2 such that equal(x,y)

// then add (x,y) to M.

//

// equal(x,y) is defined as follows:

// x, y are leafs => equal(x,y) := label(x) == label(y) && compare(value(x), value(y)) <= f

// x, y are nodes => equal(x,y) := label(x) == label(y) && |common(x,y)| / max(|x|, |y|) > t

// where f, t are constants.

//

// --- Actual implementation ---

//

// We also categorize nodes by their labels, but then we proceed differently:

//

// 1) A label may be marked "tied to parent". Let x, y have both label l and l is "tied to parent".

// Then (x,y) can be in M only if (parent(x), parent(y)) in M.

// Thus we require labels of children tied to a parent to be preceeded by all their possible parent labels.

//

// 2) Rather than defining function equal in terms of constants f and t, which are hard to get right,

// we try to match multiple times with different threashold for node distance.

// The comparer defines the distance [0..1] between two nodes and it can do so by analyzing

// the node structure and value. The comparer can tune the distance specifically for each node kind.

// We first try to match nodes of the same labels to the exactly matching or almost matching counterpars.

// The we keep increasing the threashold and keep adding matches.

for (int label = 0; label < labelCount; label++)

{

if (nodes1[label] && nodes2[label])

{

ComputeMatchForLabel(label, \*nodes1[label], \*nodes2[label]);

}

}

}

void ComputeMatchForLabel(int label, NodeList& s1, NodeList& s2)

{

int tiedToAncestor = comparer.TiedToAncestor(label);

ComputeMatchForLabel(s1, s2, tiedToAncestor, EpsilonDistance); // almost exact match

ComputeMatchForLabel(s1, s2, tiedToAncestor, MatchingDistance1); // ok match

ComputeMatchForLabel(s1, s2, tiedToAncestor, MatchingDistance2); // ok match

ComputeMatchForLabel(s1, s2, tiedToAncestor, MatchingDistance3); // ok match

ComputeMatchForLabel(s1, s2, tiedToAncestor, MaxDistance); // any match

}

void ComputeMatchForLabel(NodeList& s1, NodeList& s2, int tiedToAncestor, double maxAcceptableDistance)

{

// Obviously, the algorithm below is O(n^2). However, in the common case, the 2 lists will

// be sequences that exactly match. The purpose of "firstNonMatch2" is to reduce the complexity

// to O(n) in this case. Basically, the pointer is the 1st non-matched node in the list of nodes of tree2

// with the given label.

// Whenever we match to firstNonMatch2 we set firstNonMatch2 to the subsequent node.

// So in the case of totally matching sequences, we process them in O(n) -

// both node1 and firstNonMatch2 will be advanced simultaneously.

UnmatchedIterator i1(s1);

for (;;)

{

PNode node1 = i1.GetNextUnmatched();

if (!node1) break;

Assert(!HasPartnerInTree2(node1));

// Find node2 that matches node1 the best, i.e. has minimal distance.

double bestDistance = MaxDistance;

PNode bestMatch = nullptr;

int bestMatchIndex = -1; // node1's best match index in list2

bool matched = false;

UnmatchedIterator i2(s2);

for (;;)

{

PNode node2 = i2.GetNextUnmatched();

if (!node2) break;

Assert(!HasPartnerInTree1(node2));

// this requires parents to be processed before their children:

if (tiedToAncestor > 0)

{

// TODO: For nodes tied to their parents,

// consider avoding matching them to all other nodes of the same label.

// Rather we should only match them with their siblings that share the same parent.

PNode ancestor1 = comparer.GetAncestor(node1, tiedToAncestor);

PNode ancestor2 = comparer.GetAncestor(node2, tiedToAncestor);

Assert(comparer.GetLabel(ancestor1) < comparer.GetLabel(node1));

if (!Contains(ancestor1, ancestor2))

{

continue;

}

}

// We know that

// 1. (node1, node2) not in M

// 2. Both of their parents are matched to the same parent (or are not matched)

//

// Now, we have no other choice than comparing the node "values"

// and looking for the one with the smaller distance.

//

double distance = comparer.GetDistance(node1, node2);

if (distance < bestDistance)

{

matched = true;

bestMatch = node2;

bestMatchIndex = i2.CurIndex();

bestDistance = distance;

// We only stop if we've got an exact match. This is to resolve the problem

// of entities with identical names(name is often used as the "value" of a

// node) but with different "sub-values" (e.g. two locals may have the same name

// but different types. Since the type is not part of the value, we don't want

// to stop looking for the best match if we don't have an exact match).

if (distance == ExactMatchDistance)

{

break;

}

}

}

if (matched && bestDistance <= maxAcceptableDistance)

{

Add(node1, bestMatch);

i1.MarkCurrentMatched(); // i1's match is current node1

i2.MarkMatched(bestMatchIndex); // i2's match is one of the nodes examined in the above for(;;) pass

}

}

}

void Add(PNode node1, PNode node2)

{

Assert(comparer.TreesEqual(node1, root1));

Assert(comparer.TreesEqual(node2, root2));

oneToTwo->Add(node1, node2);

twoToOne->Add(node2, node1);

}

// The customized Match algorithm iterates over the 2 node lists, compares every unmatched node pair to match nodes.

// To find the next unmatched node, original algorithm iterates over every node in each list, use a dictionary lookup

// to test if the node has been matched or not, until it sees next unmatched node. This could be very expensive if the

// lists are huge. E.g., assume the only diff is inserting a new node at the beginning of list2. Then for each node in

// list1, it checks every node starting from the beginning new node in list2 for next unmatched node. This results in

// O(N^2) dictionary lookups. And we do 5 passes of these.

//

// To improve on this, we can try to record every match span and directly jump to next unmatched position. Note that

// in both lists once a node is matched, the list entry is no longer used. We can reuse that space to record extra info.

// \* Original PNode pointer value must be at even address. The list item must have 0 at bit0 (lowest bit).

// \* Once a node is matched, mark 1 at bit0. With this we can get rid of dictionary lookup.

// \* Next, for each matched entry, use the upper bits to record "next" unmatched index. Try to maintain match span,

// so that from a matched node we can directly jump to next unmatched index.

//

// This class is for above purpose. Expected call pattern:

// \* GetNextUnmatched, [MarkCurrentMatched], GetNextUnmatched, [MarkCurrentMatched], ...

// -- (A) With first MarkCurrentMatched we know the start of a match span.

// -- (B) Subsequent MarkCurrentMatched indicates continuous match span.

// -- (C) When MarkCurrentMatched is not called for an entry, we know the end of a match span. Record the whole

// span (A)->(C). If walked again we would directly jump from (A) to (C).

// \* Random MarkMatched(i)

// -- We don't know the exact match span. Just mark this entry "i" as matched, but set its "next" (upper bits) to 0.

// -- During next pass, we can merge all adjacent match spans and individual matched entries to bigger match spans.

// This would help next pass (we have 5).

//

class UnmatchedIterator

{

private:

NodeList& list;

int lastMatched; // last matched node index. -1 means no known last matched index.

int index; // current examining index. Only moved by GetNextUnmatched().

public:

UnmatchedIterator(NodeList& list) :

list(list),

lastMatched(-1),

index(-1)

{

VerifySize(list);

}

~UnmatchedIterator()

{

// If we have lastMatched, we could have one of following:

// \* index is matched by MarkCurrentMatched(). Link lastMatched -> index (== lastMatched). GetNextUnmatched() can handle it.

// \* index remains unmatched (ends a matched sequence). Link lastMatched -> index.

// \* index is out of range. That means [lastMatched, ...end) are all matched. Link lastMatched -> index (out of range).

//

if (lastMatched >= 0)

{

SetNext(lastMatched, index);

}

}

PNode GetNextUnmatched()

{

// If current ends a matched sequence, make a link [lastMatched -> current).

if (lastMatched >= 0 && !IsMatched(index))

{

SetNext(lastMatched, index);

lastMatched = -1;

}

++index;

if (index < list.Count())

{

if (IsMatched(index))

{

if (lastMatched < 0) // Check if current starts a matched sequence

{

lastMatched = index;

}

// Jumps all matched span, until sees an unmatched entry or the end.

int next;

while (index < list.Count() && IsNext(list.Item(index), &next))

{

index = max(next, index + 1); // Ensure moves forward (next could be 0, from individual MarkMatched() call).

}

}

if (index < list.Count())

{

return list.Item(index);

}

}

return nullptr;

}

int CurIndex() const { return index; }

void MarkMatched(int i)

{

if (i == index)

{

MarkCurrentMatched();

}

else

{

SetMatched(i);

}

}

void MarkCurrentMatched()

{

Assert(!IsMatched(index));

SetMatched(index);

if (lastMatched < 0) // If current starts a matched sequence

{

lastMatched = index;

}

}

private:

static void VerifySize(const NodeList& list)

{

if (list.Count() > INT\_MAX / 2) // Limit max size as we used bit0

{

Math::DefaultOverflowPolicy();

}

}

static void SetMatched(PNode& node)

{

SetNext(node, 0);

}

static bool IsMatched(PNode node)

{

return !!(reinterpret\_cast<UINT\_PTR>(node) & 1);

}

static void SetNext(PNode& node, int next)

{

UINT\_PTR value = (static\_cast<UINT\_PTR>(next) << 1) | 1;

node = reinterpret\_cast<PNode>(value);

}

static bool IsNext(PNode node, \_Out\_ int\* next)

{

UINT\_PTR value = reinterpret\_cast<UINT\_PTR>(node);

if (value & 1)

{

\*next = static\_cast<int>(value >> 1);

return true;

}

return false;

}

void SetMatched(int i) { SetMatched(list.Item(i)); }

bool IsMatched(int i) const { return IsMatched(list.Item(i)); }

void SetNext(int i, int next) { SetNext(list.Item(i), next); }

bool IsNext(int i, \_Out\_ int\* next) const { return IsNext(list.Item(i), next); }

};

};

template <class TreeComparer, class Allocator> const double TreeMatch<TreeComparer, Allocator>::ExactMatchDistance = TreeComparer::ExactMatchDistance;

template <class TreeComparer, class Allocator> const double TreeMatch<TreeComparer, Allocator>::EpsilonDistance = TreeComparer::EpsilonDistance;

template <class TreeComparer, class Allocator> const double TreeMatch<TreeComparer, Allocator>::MatchingDistance1 = 0.5;

template <class TreeComparer, class Allocator> const double TreeMatch<TreeComparer, Allocator>::MatchingDistance2 = 1.0;

template <class TreeComparer, class Allocator> const double TreeMatch<TreeComparer, Allocator>::MatchingDistance3 = 1.5;

template <class TreeComparer, class Allocator> const double TreeMatch<TreeComparer, Allocator>::MaxDistance = 2.0;

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

// Represents an edit operation on a tree or a sequence of nodes.

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

template <class PNode>

class Edit

{

private:

EditKind kind;

PNode node1;

PNode node2;

public:

Edit() {}

//

// Insert nullptr NewNode

// Delete OldNode nullptr

// Move/Update OldNode NewNode

//

Edit(EditKind kind, PNode node1, PNode node2) :

kind(kind), node1(node1), node2(node2)

{

Assert((node1 == nullptr) == (kind == EditKind::Insert));

Assert((node2 == nullptr) == (kind == EditKind::Delete));

}

EditKind Kind() const { return kind; }

PNode OldNode() const { return node1; }

PNode NewNode() const { return node2; }

};

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

// Represents a sequence of tree edits.

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

template <class TreeComparer, class Allocator>

class EditScript

{

public:

typedef TreeMatch<TreeComparer, Allocator> TreeMatch;

typedef typename TreeMatch::PNode PNode;

typedef typename TreeMatch::NodeList NodeList;

typedef typename TreeMatch::NodeMap NodeMap;

typedef JsUtil::List<Edit<PNode>, Allocator, /\*isLeaf\*/true> EditList;

private:

const TreeMatch& match;

TreeComparer comparer;

EditList edits;

public:

EditScript(Allocator\* alloc, const TreeMatch& match) :

match(match), comparer(match.Comparer()), edits(alloc)

{

AddUpdatesInsertsMoves();

AddDeletes();

}

const EditList& Edits() const { return edits; }

private:

PNode Root1() const { return match.OldRoot(); }

PNode Root2() const { return match.NewRoot(); }

void AddUpdatesInsertsMoves()

{

// Breadth-first traversal.

ProcessNode(Root2());

JsUtil::Queue<PNode, Allocator> queue(edits.GetAllocator());

queue.Enqueue(Root2());

while (!queue.Empty())

{

PNode head = queue.Dequeue();

comparer.MapChildren(head, [&](PNode child)

{

ProcessNode(child);

queue.Enqueue(child);

});

}

}

void ProcessNode(PNode x)

{

Assert(comparer.TreesEqual(x, Root2()));

// NOTE:

// Our implementation differs from the algorithm described in the paper in following:

// - We don't update M' and T1 since we don't need the final matching and the transformed tree.

// - Insert and Move edits don't need to store the offset of the nodes relative to their parents,

// so we don't calculate those. Thus we don't need to implement FindPos.

// - We don't mark nodes "in order" since the marks are only needed by FindPos.

// a)

// Let x be the current node in the breadth-first search of T2.

// Let y = parent(x).

// Let z be the partner of parent(x) in M'. (note: we don't need z for insert)

//

// NOTE:

// If we needed z then we would need to be updating M' as we encounter insertions.

PNode w;

bool hasPartner = match.TryGetPartnerInTree1(x, &w);

PNode y;

bool hasParent = comparer.TryGetParent(x, &y);

if (!hasPartner)

{

// b) If x has no partner in M'.

// i. k := FindPos(x)

// ii. Append INS((w, a, value(x)), z, k) to E for a new identifier w.

// iii. Add (w, x) to M' and apply INS((w, a, value(x)), z, k) to T1.

edits.Add(Edit<PNode>(EditKind::Insert, /\*node1\*/nullptr, /\*node2\*/x));

// NOTE:

// We don't update M' here.

}

else if (hasParent)

{

// c) else if x is not a root

// i. Let w be the partner of x in M', and let v = parent(w) in T1.

PNode v = comparer.GetParent(w);

// ii. if value(w) != value(x)

// A. Append UPD(w, value(x)) to E

// B. Apply UPD(w, value(x) to T1

// Let the Comparer decide whether an update should be added to the edit list.

// The Comparer defines what changes in node values it cares about.

if (!comparer.ValuesEqual(w, x))

{

edits.Add(Edit<PNode>(EditKind::Update, /\*node1\*/w, /\*node2\*/x));

}

// If parents of w and x don't match, it's a move.

// iii. if not (v, y) in M'

// NOTE: The paper says (y, v) but that seems wrong since M': T1 -> T2 and w,v in T1 and x,y in T2.

if (!match.Contains(v, y))

{

// A. Let z be the partner of y in M'. (NOTE: z not needed)

// B. k := FindPos(x)

// C. Append MOV(w, z, k)

// D. Apply MOV(w, z, k) to T1

edits.Add(Edit<PNode>(EditKind::Move, /\*node1\*/w, /\*node2\*/x));

}

}

// d) AlignChildren(w, x)

// NOTE: If we just applied an INS((w, a, value(x)), z, k) operation on tree T1

// the newly created node w would have no children. So there is nothing to align.

if (hasPartner)

{

AlignChildren(w, x);

}

}

void AddDeletes()

{

// 3. Do a post-order traversal of T1.

// a) Let w be the current node in the post-order traversal of T1.

// b) If w has no partner in M' then append DEL(w) to E and apply DEL(w) to T1.

//

// NOTE: The fact that we haven't updated M' during the Insert phase

// doesn't affect Delete phase. The original algorithm inserted new node n1 into T1

// when an insertion INS(n1, n2) was detected. It also added (n1, n2) to M'.

// Then in Delete phase n1 is visited but nothing is done since it has a partner n2 in M'.

// Since we don't add n1 into T1, not adding (n1, n2) to M' doesn't affect the Delete phase.

comparer.MapDescendants(Root1(), [&](PNode w)

{

if (!match.HasPartnerInTree2(w))

{

edits.Add(Edit<PNode>(EditKind::Delete, /\*node1\*/w, /\*node2\*/nullptr));

}

});

}

void AlignChildren(PNode w, PNode x)

{

Assert(comparer.TreesEqual(w, Root1()));

Assert(comparer.TreesEqual(x, Root2()));

Allocator\* alloc = edits.GetAllocator();

// Step 1

// Make all children of w and and all children x "out of order"

// NOTE: We don't need to mark nodes "in order".

// Step 2

// Let S1 be the sequence of children of w whose partner are children

// of x and let S2 be the sequence of children of x whose partner are

// children of w.

NodeList s1(alloc), s2(alloc);

if (!TryGetMatchedChildren(s1, w, x, [&](PNode e, PNode\* partner) { return match.TryGetPartnerInTree2(e, partner); }) ||

!TryGetMatchedChildren(s2, x, w, [&](PNode e, PNode\* partner) { return match.TryGetPartnerInTree1(e, partner); }))

{

return;

}

// Step 3, 4

// Define the function Equal(a,b) to be true if and only if (a,b) in M'

// Let S <- LCS(S1, S2, Equal)

NodeMap s(alloc);

{

LongestCommonSubsequence<Allocator> lcs(alloc, s1.Count(), s2.Count(), [&](int indexA, int indexB)

{

return match.Contains(s1.Item(indexA), s2.Item(indexB));

});

lcs.MapMatches([&](int indexA, int indexB)

{

s.AddNew(s1.Item(indexA), s2.Item(indexB));

});

}

// Step 5

// For each (a,b) in S, mark nodes a and b "in order"

// NOTE: We don't need to mark nodes "in order".

// Step 6

// For each a in S1, b in S2 such that (a,b) in M but (a,b) not in S

// (a) k <- FindPos(b)

// (b) Append MOV(a,w,k) to E and apply MOV(a,w,k) to T1

// (c) Mark a and b "in order"

// NOTE: We don't mark nodes "in order".

s1.Map([&](int index, PNode a)

{

PNode b;

if (match.TryGetPartnerInTree2(a, &b) // (a,b) in M

&& comparer.GetParent(b) == x // => b in S2 since S2 == { b | parent(b) == x && parent(partner(b)) == w }

&& !ContainsPair(s, a, b)) // (a,b) not in S

{

Assert(comparer.TreesEqual(a, Root1()));

Assert(comparer.TreesEqual(b, Root2()));

edits.Add(Edit<PNode>(EditKind::Reorder, /\*node1\*/a, /\*node2\*/b));

}

});

}

// Helper: Get the sequence of children of x whose partner are children of y.

template <class TryGetPartnerFunc>

bool TryGetMatchedChildren(NodeList& nodes, PNode x, PNode y, const TryGetPartnerFunc& tryGetPartner)

{

Assert(nodes.Empty());

comparer.MapChildren(x, [&](PNode e)

{

PNode partner;

if (tryGetPartner(e, &partner) && comparer.GetParent(partner) == y)

{

nodes.Add(e);

}

});

return !nodes.Empty();

}

static bool ContainsPair(const NodeMap& dict, PNode a, PNode b)

{

PNode value;

return dict.TryGetValue(a, &value) && value == b;

}

};

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

// 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

// Minimum amout of memory on the stack required by a vistor call.

// Use this value to control when to stop the visitor recursion before a SOE occurs.

#define PNODEVISIRORSIZE 256

template <class Context>

struct VisitorPolicyBase

{

typedef Context Context;

protected:

inline bool Preorder(ParseNode \*pnode, Context context) { return true; }

inline void Inorder(ParseNode \*pnode, Context context) { }

inline void Midorder(ParseNode \*pnode, Context context) { }

inline void Postorder(ParseNode \*pnode, Context context) { }

inline void InList(ParseNode \*pnode, Context context) { }

// This will be useful when you want the reference of your current node.

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

};

template <class Visitor, class VisitorPolicy>

struct VisitorWalkerPolicy : public VisitorPolicy

{

public:

typedef ParseNode \*ResultType;

typedef struct WalkerContext

{

typename VisitorPolicy::Context visitorContext;

Visitor \*visitor;

WalkerContext(typename VisitorPolicy::Context context, Visitor \*visitor): visitorContext(context), visitor(visitor) { }

} \*Context;

inline ParseNode \*DefaultResult() { return NULL; }

inline bool ContinueWalk(ParseNode \*result) { return true; }

inline ParseNode \*WalkNode(ParseNode \*pnode, Context context) { Inorder(pnode, context->visitorContext); return pnode; }

inline ParseNode \*WalkListNode(ParseNode \*pnode, Context context) { InList(pnode, context->visitorContext); return NULL; }

inline ParseNode \*WalkFirstChild(ParseNode \*pnode, Context context) { context->visitor->VisitNode(pnode, context); return pnode; }

inline ParseNode \*WalkSecondChild(ParseNode \*pnode, Context context) { context->visitor->VisitNode(pnode, context); return pnode; }

inline ParseNode \*WalkNthChild(ParseNode \*pparentnode, ParseNode \*pnode, Context context) { Midorder(pparentnode, context->visitorContext); context->visitor->VisitNode(pnode, context); return pnode; }

inline void WalkReference(ParseNode \*\*ppnode, Context context) { context->visitor->PassReferenceNode(ppnode, context); }

};

template <class VisitorPolicy>

class ParseNodeVisitor : public ParseNodeWalker<VisitorWalkerPolicy<ParseNodeVisitor<VisitorPolicy>, VisitorPolicy> >

{

typedef VisitorWalkerPolicy<ParseNodeVisitor<VisitorPolicy>, VisitorPolicy> WalkerPolicy;

typedef typename WalkerPolicy::WalkerContext WalkerContext;

public:

typedef typename VisitorPolicy::Context VisitorContext;

void Visit(ParseNode \*pnode, VisitorContext context = VisitorContext())

{

WalkerContext walkerContext(context, this);

VisitNode(pnode, &walkerContext);

}

void VisitNode(ParseNode \*pnode, Context context)

{

if (!ThreadContext::IsCurrentStackAvailable(PNODEVISIRORSIZE))

return;

if (!pnode) return;

if (!Preorder(pnode, context->visitorContext))

return;

Walk(pnode, context);

Postorder(pnode, context->visitorContext);

}

void PassReferenceNode(ParseNode \*\*ppnode, Context context)

{

PassReference(ppnode, context->visitorContext);

}

};

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

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

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

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

#pragma once

template <class ResultType, class Context>

struct WalkerPolicyBase

{

typedef ResultType ResultType;

typedef Context Context;

inline bool ContinueWalk(ResultType) { return true; }

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

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

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

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

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

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

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

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

};

template <class Context>

struct WalkerPolicyBase<bool, Context>

{

typedef bool ResultType;

typedef Context Context;

inline bool ContinueWalk(ResultType) { return true; }

inline bool DefaultResult() { return true; }

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

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

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

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

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

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

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

};

template <typename WalkerPolicy>

class ParseNodeWalker : public WalkerPolicy

{

public:

typedef typename WalkerPolicy::Context Context;

protected:

typedef typename WalkerPolicy::ResultType ResultType;

private:

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

{

ResultType result = DefaultResult();

bool first = true;

if (pnode)

{

result = WalkListNode(pnode, context);

if (!ContinueWalk(result)) return result;

ParseNodePtr current = pnode;

ParseNodePtr \*ppnode = &pnode;

// Skip list nodes and nested VarDeclList nodes

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

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

{

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

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

first = false;

if (!ContinueWalk(result)) return result;

ppnode = &current->sxBin.pnode2;

current = \*ppnode;

}

WalkReference(ppnode, context);

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

}

// Reset the reference back.

WalkReference(nullptr, context);

return result;

}

ResultType WalkLeaf(ParseNode \*pnode, Context context)

{

return WalkNode(pnode, context);

}

ResultType WalkPreUnary(ParseNode \*pnode, Context context)

{

ResultType result = WalkNode(pnode, context);

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

return result;

}

ResultType WalkPostUnary(ParseNode \*pnode, Context context)

{

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

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

return result;

}

ResultType WalkBinary(ParseNode \*pnode, Context context)

{

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

if (ContinueWalk(result))

{

result = WalkNode(pnode, context);

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

}

return result;

}

ResultType WalkTiernary(ParseNode \*pnode, Context context)

{

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

if (ContinueWalk(result))

{

result = WalkNode(pnode, context);

if (ContinueWalk(result))

{

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

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

}

}

return result;

}

ResultType WalkCall(ParseNode \*pnode, Context context)

{

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

if (ContinueWalk(result))

{

result = WalkNode(pnode, context);

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

}

return result;

}

ResultType WalkStringTemplate(ParseNode \*pnode, Context context)

{

ResultType result;

if (!pnode->sxStrTemplate.isTaggedTemplate)

{

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

{

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

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

}

else

{

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

if (ContinueWalk(result))

{

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

}

}

}

else

{

// Tagged template nodes are call nodes

result = WalkCall(pnode, context);

}

return result;

}

ResultType WalkVar(ParseNode \*pnode, Context context)

{

ResultType result = WalkNode(pnode, context);

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

return result;

}

ResultType WalkFnc(ParseNode \*pnode, Context context)

{

ResultType result;

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

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

{

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

if (!ContinueWalk(result)) return result;

}

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

{

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

if (!ContinueWalk(result)) return result;

}

result = WalkNode(pnode, context);

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

return result;

}

ResultType WalkProg(ParseNode \*pnode, Context context)

{

ResultType result = WalkNode(pnode, context);

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

return result;

}

ResultType WalkFor(ParseNode \*pnode, Context context)

{

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

if (ContinueWalk(result))

{

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

if (ContinueWalk(result))

{

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

if (ContinueWalk(result))

{

result = WalkNode(pnode, context);

if (ContinueWalk(result))

{

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

}

}

}

}

return result;

}

ResultType WalkIf(ParseNode \*pnode, Context context)

{

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

if (ContinueWalk(result))

{

result = WalkNode(pnode, context);

if (ContinueWalk(result))

{

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

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

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

}

}

return result;

}

ResultType WalkWhile(ParseNode \*pnode, Context context)

{

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

if (ContinueWalk(result))

{

result = WalkNode(pnode, context);

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

}

return result;

}

ResultType WalkDoWhile(ParseNode \*pnode, Context context)

{

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

if (ContinueWalk(result))

{

result = WalkNode(pnode, context);

if (ContinueWalk(result))

{

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

}

}

return result;

}

ResultType WalkForInOrForOf(ParseNode \*pnode, Context context)

{

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

if (ContinueWalk(result))

{

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

if (ContinueWalk(result))

{

result = WalkNode(pnode, context);

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

}

}

return result;

}

ResultType WalkReturn(ParseNode \*pnode, Context context)

{

ResultType result = WalkNode(pnode, context);

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

return result;

}

ResultType WalkBlock(ParseNode \*pnode, Context context)

{

ResultType result = WalkNode(pnode, context);

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

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

return result;

}

ResultType WalkWith(ParseNode \*pnode, Context context)

{

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

if (ContinueWalk(result))

{

result = WalkNode(pnode, context);

if (ContinueWalk(result))

{

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

}

}

return result;

}

ResultType WalkSwitch(ParseNode \*pnode, Context context)

{

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

if (ContinueWalk(result))

{

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

{

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

if (!ContinueWalk(result)) return result;

}

result = WalkNode(pnode, context);

}

return result;

}

ResultType WalkCase(ParseNode \*pnode, Context context)

{

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

if (ContinueWalk(result))

{

result = WalkNode(pnode, context);

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

}

return result;

}

ResultType WalkTryFinally(ParseNode \*pnode, Context context)

{

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

if (ContinueWalk(result))

{

result = WalkNode(pnode, context);

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

}

return result;

}

ResultType WalkFinally(ParseNode \*pnode, Context context)

{

ResultType result = WalkNode(pnode, context);

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

return result;

}

ResultType WalkCatch(ParseNode \*pnode, Context context)

{

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

if (ContinueWalk(result))

{

result = WalkNode(pnode, context);

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

}

return result;

}

ResultType WalkTryCatch(ParseNode \*pnode, Context context)

{

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

if (ContinueWalk(result))

{

result = WalkNode(pnode, context);

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

}

return result;

}

ResultType WalkTry(ParseNode \*pnode, Context context)

{

ResultType result = WalkNode(pnode, context);

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

return result;

}

ResultType WalkClass(ParseNode \*pnode, Context context)

{

// First walk the class node itself

ResultType result = WalkNode(pnode, context);

if (!ContinueWalk(result)) return result;

// Walk extends expr

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

if (!ContinueWalk(result)) return result;

// Walk the constructor

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

if (!ContinueWalk(result)) return result;

// Walk all non-static members

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

if (!ContinueWalk(result)) return result;

// Walk all static members

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

return result;

}

public:

ResultType Walk(ParseNode \*pnode, Context context)

{

if (!pnode) return DefaultResult();

switch (pnode->nop) {

// Handle all special cases first.

// Post-fix unary operators.

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

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

case knopIncPost:

case knopDecPost:

return WalkPostUnary(pnode, context);

// Call and call like

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

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

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

case knopCall:

case knopNew:

case knopIndex:

return WalkCall(pnode, context);

// Tierinary operator

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

case knopQmark:

return WalkTiernary(pnode, context);

// General nodes.

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

case knopList:

return WalkList(NULL, pnode, context);

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

case knopVarDecl:

case knopConstDecl:

case knopLetDecl:

case knopTemp:

return WalkVar(pnode, context);

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

case knopFncDecl:

return WalkFnc(pnode, context);

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

case knopProg:

return WalkProg(pnode, context);

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

case knopFor:

return WalkFor(pnode, context);

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

case knopIf:

return WalkIf(pnode, context);

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

case knopWhile:

return WalkWhile(pnode, context);

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

case knopDoWhile:

return WalkDoWhile(pnode, context);

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

case knopForIn:

return WalkForInOrForOf(pnode, context);

case knopForOf:

return WalkForInOrForOf(pnode, context);

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

case knopReturn:

return WalkReturn(pnode, context);

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

case knopBlock:

return WalkBlock(pnode, context);

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

case knopWith:

return WalkWith(pnode, context);

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

case knopSwitch:

return WalkSwitch(pnode, context);

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

case knopCase:

return WalkCase(pnode, context);

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

case knopTryFinally:

return WalkTryFinally(pnode, context);

case knopFinally:

return WalkFinally(pnode, context);

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

case knopCatch:

return WalkCatch(pnode, context);

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

case knopTryCatch:

return WalkTryCatch(pnode, context);

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

case knopTry:

return WalkTry(pnode, context);

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

case knopThrow:

return WalkPostUnary(pnode, context);

case knopStrTemplate:

return WalkStringTemplate(pnode, context);

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

case knopClassDecl:

return WalkClass(pnode, context);

default:

{

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

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

{

return WalkLeaf(pnode, context);

}

else if (fnop & fnopBin)

{

return WalkBinary(pnode, context);

}

else if (fnop & fnopUni)

{

// Prefix unary operators.

return WalkPreUnary(pnode, context);

}

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

if (fnop & fnopNotExprStmt)

{

return WalkLeaf(pnode, context);

}

Assert(false);

\_\_assume(false);

}

}

}

};

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

// 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

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

Node operators (indicates semantics of the parse node)

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

enum OpCode : byte

{

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

#include "ptlist.h"

knopLim

};

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

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

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

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

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

#ifndef PTNODE

#error Define PTNODE before including this file.

#endif

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

//

// Node oper

// , "Node name"

// , pcode

// , parse node kind

// , flags

// , JSON Name

//

PTNODE(knopNone , "<none>" , Nop , None , fnopNone , "" )

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

Leaf nodes.

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

PTNODE(knopName , "name" , Nop , Pid , fnopLeaf , "NameExpr" )

PTNODE(knopInt , "int const" , Nop , Int , fnopLeaf|fnopConst , "NumberLit" )

PTNODE(knopFlt , "flt const" , Nop , Flt , fnopLeaf|fnopConst , "NumberLit" )

PTNODE(knopStr , "str const" , Nop , Pid , fnopLeaf|fnopConst , "StringLit" )

PTNODE(knopRegExp , "reg expr" , Nop , Pid , fnopLeaf|fnopConst , "RegExprLit" )

PTNODE(knopThis , "this" , Nop , None , fnopLeaf , "ThisExpr" )

PTNODE(knopSuper , "super" , Nop , None , fnopLeaf , "SuperExpr" )

PTNODE(knopNewTarget , "new.target" , Nop , None , fnopLeaf , "NewTargetExpr" )

PTNODE(knopNull , "null" , Nop , None , fnopLeaf , "NullLit" )

PTNODE(knopFalse , "false" , Nop , None , fnopLeaf , "FalseLit" )

PTNODE(knopTrue , "true" , Nop , None , fnopLeaf , "TrueLit" )

PTNODE(knopEmpty , "empty" , Nop , None , fnopLeaf , "EmptStmt" )

PTNODE(knopYieldLeaf , "yield leaf" , Nop , None , fnopLeaf , "YieldLeafExpr" )

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

Unary operators.

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

PTNODE(knopNot , "~" , Nop , Uni , fnopUni , "BitNotOper" )

PTNODE(knopNeg , "unary -" , Nop , Uni , fnopUni , "NegOper" )

PTNODE(knopPos , "unary +" , Nop , Uni , fnopUni , "PosOper" )

PTNODE(knopLogNot , "!" , Nop , Uni , fnopUni , "LogNotOper" )

PTNODE(knopEllipsis , "..." , Nop , Uni , fnopUni , "Spread" )

// \_\_\_compact range : do not add or remove in this range.

// Gen code of OP\_LclIncPost,.. depends on parallel tables with this range

PTNODE(knopIncPost , "++ post" , Nop , Uni , fnopUni|fnopAsg , "PostIncExpr" )

PTNODE(knopDecPost , "-- post" , Nop , Uni , fnopUni|fnopAsg , "PostDecExpr" )

PTNODE(knopIncPre , "++ pre" , Nop , Uni , fnopUni|fnopAsg , "PreIncExpr" )

PTNODE(knopDecPre , "-- pre" , Nop , Uni , fnopUni|fnopAsg , "PreDecExpr" )

//\_\_\_end range

PTNODE(knopTypeof , "typeof" , Nop , Uni , fnopUni , "TypeOfExpr" )

PTNODE(knopVoid , "void" , Nop , Uni , fnopUni , "VoidExpr" )

PTNODE(knopDelete , "delete" , Nop , Uni , fnopUni , "DeleteStmt" )

PTNODE(knopArray , "arr cnst" , Nop , ArrLit , fnopUni , "ArrayExpr" )

PTNODE(knopObject , "obj cnst" , Nop , Uni , fnopUni , "ObjectExpr" )

PTNODE(knopTempRef , "temp ref" , Nop , Uni , fnopUni , "TempRef" )

PTNODE(knopComputedName,"[name]" , Nop , Uni , fnopUni , "ComputedNameExpr" )

PTNODE(knopYield , "yield" , Nop , Uni , fnopUni|fnopAsg , "YieldExpr" )

PTNODE(knopYieldStar , "yield \*" , Nop , Uni , fnopUni|fnopAsg , "YieldStarExpr" )

PTNODE(knopAwait , "await" , Nop , Uni , fnopUni , "AwaitExpr" )

PTNODE(knopAsyncSpawn , "asyncspawn" , Nop , Bin , fnopBin , "AsyncSpawnExpr" )

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

Binary and ternary operators.

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

PTNODE(knopAdd , "+" , Add\_A , Bin , fnopBin , "AddOper" )

PTNODE(knopSub , "-" , Sub\_A , Bin , fnopBin , "SubOper" )

PTNODE(knopMul , "\*" , Mul\_A , Bin , fnopBin , "MulOper" )

PTNODE(knopDiv , "/" , Div\_A , Bin , fnopBin , "DivOper" )

PTNODE(knopExpo , "\*\*" , Expo\_A , Bin , fnopBin , "ExpoOper" )

PTNODE(knopMod , "%" , Rem\_A , Bin , fnopBin , "ModOper" )

PTNODE(knopOr , "|" , Or\_A , Bin , fnopBin , "BitOrOper" )

PTNODE(knopXor , "^" , Xor\_A , Bin , fnopBin , "BitXorOper" )

PTNODE(knopAnd , "&" , And\_A , Bin , fnopBin , "BitAndOper" )

PTNODE(knopEq , "==" , OP(Eq) , Bin , fnopBin|fnopRel , "EqualOper" )

PTNODE(knopNe , "!=" , OP(Neq) , Bin , fnopBin|fnopRel , "NotEqualOper" )

PTNODE(knopLt , "<" , OP(Lt) , Bin , fnopBin|fnopRel , "LessThanOper" )

PTNODE(knopLe , "<=" , OP(Le) , Bin , fnopBin|fnopRel , "LessThanEqualOper" )

PTNODE(knopGe , ">=" , OP(Ge) , Bin , fnopBin|fnopRel , "GreaterThanEqualOper" )

PTNODE(knopGt , ">" , OP(Gt) , Bin , fnopBin|fnopRel , "GreaterThanOper" )

PTNODE(knopCall , "()" , Nop , Call , fnopBin , "CallExpr" )

PTNODE(knopDot , "." , Nop , Bin , fnopBin , "DotOper" )

PTNODE(knopAsg , "=" , Nop , Bin , fnopBin|fnopAsg , "AssignmentOper" )

PTNODE(knopInstOf , "instanceof" , IsInst , Bin , fnopBin|fnopRel , "InstanceOfExpr" )

PTNODE(knopIn , "in" , IsIn , Bin , fnopBin|fnopRel , "InOper" )

PTNODE(knopEqv , "===" , OP(SrEq) , Bin , fnopBin|fnopRel , "StrictEqualOper" )

PTNODE(knopNEqv , "!==" , OP(SrNeq), Bin , fnopBin|fnopRel , "NotStrictEqualOper" )

PTNODE(knopComma , "," , Nop , Bin , fnopBin , "CommaOper" )

PTNODE(knopLogOr , "||" , Nop , Bin , fnopBin , "LogOrOper" )

PTNODE(knopLogAnd , "&&" , Nop , Bin , fnopBin , "LogAndOper" )

PTNODE(knopLsh , "<<" , Shl\_A , Bin , fnopBin , "LeftShiftOper" )

PTNODE(knopRsh , ">>" , Shr\_A , Bin , fnopBin , "RightShiftOper" )

PTNODE(knopRs2 , ">>>" , ShrU\_A , Bin , fnopBin , "UnsignedRightShiftOper" )

PTNODE(knopNew , "new" , Nop , Call , fnopBin , "NewExpr" )

PTNODE(knopIndex , "[]" , Nop , Bin , fnopBin , "IndexOper" )

PTNODE(knopQmark , "?" , Nop , Tri , fnopBin , "IfExpr" )

// \_\_\_compact range : do not add or remove in this range.

// Gen code of OP\_LclAsg\*,.. depends on parallel tables with this range

PTNODE(knopAsgAdd , "+=" , Add\_A , Bin , fnopBin|fnopAsg , "AddAssignExpr" )

PTNODE(knopAsgSub , "-=" , Sub\_A , Bin , fnopBin|fnopAsg , "SubAssignExpr" )

PTNODE(knopAsgMul , "\*=" , Mul\_A , Bin , fnopBin|fnopAsg , "MulAssignExpr" )

PTNODE(knopAsgDiv , "/=" , Div\_A , Bin , fnopBin|fnopAsg , "DivAssignExpr" )

PTNODE(knopAsgExpo , "\*\*=" , Expo\_A , Bin , fnopBin|fnopAsg , "ExpoAssignExpr" )

PTNODE(knopAsgMod , "%=" , Rem\_A , Bin , fnopBin|fnopAsg , "ModAssignExpr" )

PTNODE(knopAsgAnd , "&=" , And\_A , Bin , fnopBin|fnopAsg , "BitAndAssignExpr" )

PTNODE(knopAsgXor , "^=" , Xor\_A , Bin , fnopBin|fnopAsg , "BitXorAssignExpr" )

PTNODE(knopAsgOr , "|=" , Or\_A , Bin , fnopBin|fnopAsg , "BitOrAssignExpr" )

PTNODE(knopAsgLsh , "<<=" , Shl\_A , Bin , fnopBin|fnopAsg , "LeftShiftAssignExpr" )

PTNODE(knopAsgRsh , ">>=" , Shr\_A , Bin , fnopBin|fnopAsg , "RightShiftAssignExpr" )

PTNODE(knopAsgRs2 , ">>>=" , ShrU\_A , Bin , fnopBin|fnopAsg , "UnsignedRightShiftAssignExpr" )

//\_\_\_end range

PTNODE(knopMember , ":" , Nop , Bin , fnopNotExprStmt|fnopBin, "MemberOper" )

PTNODE(knopMemberShort, "membShort" , Nop , Bin , fnopNotExprStmt|fnopBin, "ShorthandMember" )

PTNODE(knopSetMember , "set" , Nop , Bin , fnopBin , "SetDecl" )

PTNODE(knopGetMember , "get" , Nop , Bin , fnopBin , "GetDecl" )

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

General nodes.

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

PTNODE(knopList , "<list>" , Nop , Bin , fnopBinList|fnopNotExprStmt, "" )

PTNODE(knopVarDecl , "varDcl" , Nop , Var , fnopNotExprStmt , "VarDecl" )

PTNODE(knopConstDecl , "constDcl" , Nop , Var , fnopNotExprStmt , "ConstDecl" )

PTNODE(knopLetDecl , "letDcl" , Nop , Var , fnopNotExprStmt , "LetDecl" )

PTNODE(knopTemp , "temp" , Nop , Var , fnopNone , "Temp" )

PTNODE(knopFncDecl , "fncDcl" , Nop , Fnc , fnopLeaf , "FuncDecl" )

PTNODE(knopClassDecl , "classDecl" , Nop , Class , fnopLeaf , "ClassDecl" )

PTNODE(knopProg , "program" , Nop , Prog , fnopNotExprStmt , "Unit" )

PTNODE(knopEndCode , "<endcode>" , Nop , None , fnopNotExprStmt , "" )

PTNODE(knopDebugger , "debugger" , Nop , None , fnopNotExprStmt , "DebuggerStmt" )

PTNODE(knopFor , "for" , Nop , For , fnopNotExprStmt|fnopCleanup|fnopBreak|fnopContinue , "ForStmtm" )

PTNODE(knopIf , "if" , Nop , If , fnopNotExprStmt , "IfStmt" )

PTNODE(knopWhile , "while" , Nop , While , fnopNotExprStmt|fnopCleanup|fnopBreak|fnopContinue , "WhileStmt" )

PTNODE(knopDoWhile , "do-while" , Nop , While , fnopNotExprStmt|fnopCleanup|fnopBreak|fnopContinue , "DoWhileStmt" )

PTNODE(knopForIn , "for in" , Nop , ForIn , fnopNotExprStmt|fnopCleanup|fnopBreak|fnopContinue , "ForInStmt" )

PTNODE(knopForOf , "for of" , Nop , ForOf , fnopNotExprStmt|fnopCleanup|fnopBreak|fnopContinue , "ForOfStmt" )

PTNODE(knopBlock , "{}" , Nop , Block , fnopNotExprStmt , "Block" )

PTNODE(knopStrTemplate, "``" , Nop , StrTemplate , fnopNone , "StringTemplateDecl" )

PTNODE(knopWith , "with" , Nop , With , fnopNotExprStmt , "WithStmt" )

PTNODE(knopBreak , "break" , Nop , Jump , fnopNotExprStmt , "BreakStmt" )

PTNODE(knopContinue , "continue" , Nop , Jump , fnopNotExprStmt , "ContinueStmt" )

PTNODE(knopLabel , "label" , Nop , Label , fnopNotExprStmt , "LabelDecl" )

PTNODE(knopSwitch , "switch" , Nop , Switch , fnopNotExprStmt|fnopBreak, "SwitchStmt" )

PTNODE(knopCase , "case" , Nop , Case , fnopNotExprStmt , "CaseStmt" )

PTNODE(knopTryCatch , "try-catch" , Nop , TryCatch , fnopNotExprStmt , "TryCatchStmt" )

PTNODE(knopCatch , "catch" , Nop , Catch , fnopNotExprStmt|fnopCleanup, "CatchClause" )

PTNODE(knopReturn , "return" , Nop , Return , fnopNotExprStmt , "ReturnStmt" )

PTNODE(knopTry , "try" , Nop , Try , fnopNotExprStmt|fnopCleanup, "TryStmt" )

PTNODE(knopThrow , "throw" , Nop , Uni , fnopNotExprStmt , "ThrowStmt" )

PTNODE(knopFinally , "finally" , Nop , Finally , fnopNotExprStmt|fnopCleanup, "FinallyStmt" )

PTNODE(knopTryFinally , "try-finally" , Nop , TryFinally , fnopNotExprStmt , "TryFinallyStmt" )

PTNODE(knopObjectPattern, "{} = " , Nop , Uni , fnopUni , "ObjectAssignmentPattern" )

PTNODE(knopObjectPatternMember, "{:} = " , Nop , Bin , fnopBin , "ObjectAssignmentPatternMember" )

PTNODE(knopArrayPattern, "[] = " , Nop , ArrLit , fnopUni , "ArrayAssignmentPattern" )

PTNODE(knopParamPattern, "({[]})" , Nop , ParamPattern, fnopUni , "DestructurePattern" )

#undef PTNODE

#undef OP

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

// 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

struct Ident;

typedef Ident \*IdentPtr;

class Scope;

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

Flags for classifying node operators.

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

const uint fnopNone = 0x0000;

const uint fnopConst = 0x0001; // constant

const uint fnopLeaf = 0x0002; // leaf

const uint fnopUni = 0x0004; // unary

const uint fnopBin = 0x0008; // binary

const uint fnopRel = 0x0010; // relational

const uint fnopAsg = 0x0020; // assignment

const uint fnopBreak = 0x0040; // break can be used within this statement

const uint fnopContinue = 0x0080; // continue can be used within this statement

const uint fnopCleanup = 0x0100; // requires cleanup (eg, with or for-in).

const uint fnopJump = 0x0200;

const uint fnopNotExprStmt = 0x0400;

const uint fnopBinList = 0x0800;

const uint fnopExprMask = (fnopLeaf|fnopUni|fnopBin);

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

Flags for classifying parse nodes.

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

enum PNodeFlags : ushort

{

fpnNone = 0x0000,

// knopFncDecl nodes.

fpnArguments\_overriddenByDecl = 0x0001, // function has a parameter, let/const decl, class or nested function named 'arguments', which overrides the built-in arguments object

fpnArguments\_varDeclaration = 0x0002, // function has a var declaration named 'arguments', which may change the way an 'arguments' identifier is resolved

// knopVarDecl nodes.

fpnArguments = 0x0004,

fpnHidden = 0x0008,

// Statment nodes.

fpnExplicitSimicolon = 0x0010, // statment terminated by an explicit semicolon

fpnAutomaticSimicolon = 0x0020, // statment terminated by an automatic semicolon

fpnMissingSimicolon = 0x0040, // statment missing terminating semicolon, and is not applicable for automatic semicolon insersion

fpnDclList = 0x0080, // statment is a declaration list

fpnSyntheticNode = 0x0100, // node is added by the parser or does it represent user code

fpnIndexOperator = 0x0200, // dot operator is an optimization of an index operator

fpnJumbStatement = 0x0400, // break or continue that was removed by error recovery

// Unary/Binary nodes

fpnCanFlattenConcatExpr = 0x0800, // the result of the binary operation can particpate in concat N

// Potentially overlapping transitor flags

// These flags are set and cleared during a single node traversal and their values can be used in other node traversals.

fpnMemberReference = 0x1000, // The node is a member reference symbol

fpnCapturesSyms = 0x2000, // The node is a statement (or contains a sub-statement)

// that captures symbols.

};

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

Data structs for ParseNodes. ParseNode includes a union of these.

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

struct PnUni

{

ParseNodePtr pnode1;

};

struct PnBin

{

ParseNodePtr pnodeNext;

ParseNodePtr pnode1;

ParseNodePtr pnode2;

};

struct PnTri

{

ParseNodePtr pnodeNext;

ParseNodePtr pnode1;

ParseNodePtr pnode2;

ParseNodePtr pnode3;

};

struct PnSlot

{

uint slotIndex;

};

struct PnUniSlot : PnUni

{

uint slotIndex;

uint staticFuncId;

};

struct PnInt

{

long lw;

};

struct PnFlt

{

double dbl;

bool maybeInt : 1;

};

class Symbol;

struct PidRefStack;

struct PnPid

{

IdentPtr pid;

Symbol \*\*symRef;

Symbol \*sym;

UnifiedRegex::RegexPattern\* regexPattern;

uint regexPatternIndex;

void SetSymRef(PidRefStack \*ref);

Symbol \*\*GetSymRef() const { return symRef; }

Js::PropertyId PropertyIdFromNameNode() const;

};

struct PnVar

{

ParseNodePtr pnodeNext;

IdentPtr pid;

Symbol \*sym;

Symbol \*\*symRef;

ParseNodePtr pnodeInit;

BOOLEAN isSwitchStmtDecl;

BOOLEAN isBlockScopeFncDeclVar;

void InitDeclNode(IdentPtr name, ParseNodePtr initExpr)

{

this->pid = name;

this->pnodeInit = initExpr;

this->pnodeNext = nullptr;

this->sym = nullptr;

this->symRef = nullptr;

this->isSwitchStmtDecl = false;

this->isBlockScopeFncDeclVar = false;

}

};

struct PnLabel

{

IdentPtr pid;

ParseNodePtr pnodeNext;

};

struct PnArrLit : PnUni

{

uint count;

uint spreadCount;

BYTE arrayOfTaggedInts:1; // indicates that array initialzer nodes are all tagged ints

BYTE arrayOfInts:1; // indicates that array initialzer nodes are all ints

BYTE arrayOfNumbers:1; // indicates that array initialzer nodes are all numbers

BYTE hasMissingValues:1;

};

class FuncInfo;

enum PnodeBlockType : unsigned

{

Global,

Function,

Regular,

Parameter

};

enum FncFlags

{

kFunctionNone = 0,

kFunctionNested = 1 << 0, // True if function is nested in another.

kFunctionDeclaration = 1 << 1, // is this a declaration or an expression?

kFunctionCallsEval = 1 << 2, // function uses eval

kFunctionUsesArguments = 1 << 3, // function uses arguments

kFunctionHasHeapArguments = 1 << 4, // function's "arguments" escape the scope

kFunctionHasReferencableBuiltInArguments = 1 << 5, // the built-in 'arguments' object is referenceable in the function

kFunctionIsAccessor = 1 << 6, // function is a property getter or setter

kFunctionHasNonThisStmt = 1 << 7,

kFunctionStrictMode = 1 << 8,

kFunctionDoesNotEscape = 1 << 9, // function is known not to escape its declaring scope

kFunctionSubsumed = 1 << 10, // function expression is a parameter in a call that has no closing paren and should be treated as a global declaration (only occurs during error correction)

kFunctionHasThisStmt = 1 << 11, // function has at least one this.assignment and might be a constructor

kFunctionHasWithStmt = 1 << 12, // function (or child) uses with

kFunctionIsLambda = 1 << 13,

kFunctionChildCallsEval = 1 << 14,

kFunctionHasDestructuringPattern = 1 << 15,

kFunctionHasSuperReference = 1 << 16,

kFunctionIsMethod = 1 << 17,

kFunctionIsClassConstructor = 1 << 18, // function is a class constructor

kFunctionIsBaseClassConstructor = 1 << 19, // function is a base class constructor

kFunctionIsClassMember = 1 << 20, // function is a class member

kFunctionNameIsHidden = 1 << 21, // True if a named function expression has its name hidden from nested functions

kFunctionIsGeneratedDefault = 1 << 22, // Is the function generated by us as a default (e.g. default class constructor)

kFunctionHasDefaultArguments = 1 << 23, // Function has one or more ES6 default arguments

kFunctionIsStaticMember = 1 << 24,

kFunctionIsGenerator = 1 << 25, // Function is an ES6 generator function

kFunctionAsmjsMode = 1 << 26,

kFunctionHasNewTargetReference = 1 << 27, // function has a reference to new.target

kFunctionIsAsync = 1 << 28, // function is async

kFunctionHasDirectSuper = 1 << 29, // super()

};

struct RestorePoint;

struct DeferredFunctionStub;

struct PnFnc

{

ParseNodePtr pnodeNext;

ParseNodePtr pnodeName;

IdentPtr pid;

LPCOLESTR hint;

ulong hintLength;

ulong hintOffset;

bool isNameIdentifierRef;

ParseNodePtr pnodeScopes;

ParseNodePtr pnodeBodyScope;

ParseNodePtr pnodeArgs;

ParseNodePtr pnodeVars;

ParseNodePtr pnodeBody;

ParseNodePtr pnodeRest;

FuncInfo \*funcInfo; // function information gathered during byte code generation

Scope \*scope;

uint nestedCount; // Nested function count (valid until children have been processed)

uint nestedIndex; // Index within the parent function

uint16 firstDefaultArg; // Position of the first default argument, if any

unsigned int fncFlags;

long astSize;

size\_t cbMin; // Min an Lim UTF8 offsets.

size\_t cbLim;

ULONG lineNumber; // Line number relative to the current source buffer of the function declaration.

ULONG columnNumber; // Column number of the declaration.

Js::LocalFunctionId functionId;

#if DBG

Js::LocalFunctionId deferredParseNextFunctionId;

#endif

RestorePoint \*pRestorePoint;

DeferredFunctionStub \*deferredStub;

static const long MaxStackClosureAST = 800000;

private:

void SetFlags(uint flags, bool set)

{

if (set)

{

fncFlags |= flags;

}

else

{

fncFlags &= ~flags;

}

}

bool HasFlags(uint flags) const

{

return (fncFlags & flags) == flags;

}

public:

void ClearFlags()

{

fncFlags = kFunctionNone;

}

void SetAsmjsMode(bool set = true) { SetFlags(kFunctionAsmjsMode, set); }

void SetCallsEval(bool set = true) { SetFlags(kFunctionCallsEval, set); }

void SetChildCallsEval(bool set = true) { SetFlags(kFunctionChildCallsEval, set); }

void SetDeclaration(bool set = true) { SetFlags(kFunctionDeclaration, set); }

void SetDoesNotEscape(bool set = true) { SetFlags(kFunctionDoesNotEscape, set); }

void SetHasDefaultArguments(bool set = true) { SetFlags(kFunctionHasDefaultArguments, set); }

void SetHasDestructuringPattern(bool set = true) { SetFlags(kFunctionHasDestructuringPattern, set); }

void SetHasHeapArguments(bool set = true) { SetFlags(kFunctionHasHeapArguments, set); }

void SetHasNonThisStmt(bool set = true) { SetFlags(kFunctionHasNonThisStmt, set); }

void SetHasReferenceableBuiltInArguments(bool set = true) { SetFlags(kFunctionHasReferencableBuiltInArguments, set); }

void SetHasSuperReference(bool set = true) { SetFlags(kFunctionHasSuperReference, set); }

void SetHasDirectSuper(bool set = true) { SetFlags(kFunctionHasDirectSuper, set); }

void SetHasNewTargetReferene(bool set = true) { SetFlags(kFunctionHasNewTargetReference, set); }

void SetHasThisStmt(bool set = true) { SetFlags(kFunctionHasThisStmt, set); }

void SetHasWithStmt(bool set = true) { SetFlags(kFunctionHasWithStmt, set); }

void SetIsAccessor(bool set = true) { SetFlags(kFunctionIsAccessor, set); }

void SetIsAsync(bool set = true) { SetFlags(kFunctionIsAsync, set); }

void SetIsClassConstructor(bool set = true) { SetFlags(kFunctionIsClassConstructor, set); }

void SetIsBaseClassConstructor(bool set = true) { SetFlags(kFunctionIsBaseClassConstructor, set); }

void SetIsClassMember(bool set = true) { SetFlags(kFunctionIsClassMember, set); }

void SetIsGeneratedDefault(bool set = true) { SetFlags(kFunctionIsGeneratedDefault, set); }

void SetIsGenerator(bool set = true) { SetFlags(kFunctionIsGenerator, set); }

void SetIsLambda(bool set = true) { SetFlags(kFunctionIsLambda, set); }

void SetIsMethod(bool set = true) { SetFlags(kFunctionIsMethod, set); }

void SetIsStaticMember(bool set = true) { SetFlags(kFunctionIsStaticMember, set); }

void SetNameIsHidden(bool set = true) { SetFlags(kFunctionNameIsHidden, set); }

void SetNested(bool set = true) { SetFlags(kFunctionNested, set); }

void SetStrictMode(bool set = true) { SetFlags(kFunctionStrictMode, set); }

void SetSubsumed(bool set = true) { SetFlags(kFunctionSubsumed, set); }

void SetUsesArguments(bool set = true) { SetFlags(kFunctionUsesArguments, set); }

bool CallsEval() const { return HasFlags(kFunctionCallsEval); }

bool ChildCallsEval() const { return HasFlags(kFunctionChildCallsEval); }

bool DoesNotEscape() const { return HasFlags(kFunctionDoesNotEscape); }

bool GetArgumentsObjectEscapes() const { return HasFlags(kFunctionHasHeapArguments); }

bool GetAsmjsMode() const { return HasFlags(kFunctionAsmjsMode); }

bool GetStrictMode() const { return HasFlags(kFunctionStrictMode); }

bool HasDefaultArguments() const { return HasFlags(kFunctionHasDefaultArguments); }

bool HasDestructuringPattern() const { return HasFlags(kFunctionHasDestructuringPattern); }

bool HasHeapArguments() const { return true; /\* HasFlags(kFunctionHasHeapArguments); Disabling stack arguments. Always return HeapArguments as True \*/ }

bool HasOnlyThisStmts() const { return !HasFlags(kFunctionHasNonThisStmt); }

bool HasReferenceableBuiltInArguments() const { return HasFlags(kFunctionHasReferencableBuiltInArguments); }

bool HasSuperReference() const { return HasFlags(kFunctionHasSuperReference); }

bool HasDirectSuper() const { return HasFlags(kFunctionHasDirectSuper); }

bool HasNewTargetReference() const { return HasFlags(kFunctionHasNewTargetReference); }

bool HasThisStmt() const { return HasFlags(kFunctionHasThisStmt); }

bool HasWithStmt() const { return HasFlags(kFunctionHasWithStmt); }

bool IsAccessor() const { return HasFlags(kFunctionIsAccessor); }

bool IsAsync() const { return HasFlags(kFunctionIsAsync); }

bool IsClassConstructor() const { return HasFlags(kFunctionIsClassConstructor); }

bool IsBaseClassConstructor() const { return HasFlags(kFunctionIsBaseClassConstructor); }

bool IsClassMember() const { return HasFlags(kFunctionIsClassMember); }

bool IsDeclaration() const { return HasFlags(kFunctionDeclaration); }

bool IsGeneratedDefault() const { return HasFlags(kFunctionIsGeneratedDefault); }

bool IsGenerator() const { return HasFlags(kFunctionIsGenerator); }

bool IsLambda() const { return HasFlags(kFunctionIsLambda); }

bool IsMethod() const { return HasFlags(kFunctionIsMethod); }

bool IsNested() const { return HasFlags(kFunctionNested); }

bool IsStaticMember() const { return HasFlags(kFunctionIsStaticMember); }

bool IsSubsumed() const { return HasFlags(kFunctionSubsumed); }

bool NameIsHidden() const { return HasFlags(kFunctionNameIsHidden); }

bool UsesArguments() const { return HasFlags(kFunctionUsesArguments); }

bool IsSimpleParameterList() const { return !HasDefaultArguments() && !HasDestructuringPattern() && pnodeRest == nullptr; }

size\_t LengthInBytes()

{

return cbLim - cbMin;

}

Symbol \*GetFuncSymbol();

void SetFuncSymbol(Symbol \*sym);

ParseNodePtr GetParamScope() const;

ParseNodePtr \*GetParamScopeRef() const;

ParseNodePtr GetBodyScope() const;

ParseNodePtr \*GetBodyScopeRef() const;

ParseNodePtr GetTopLevelScope() const

{

// Top level scope will be the same for knopProg and knopFncDecl.

return GetParamScope();

}

template<typename Fn>

void MapContainerScopes(Fn fn)

{

fn(this->pnodeScopes->sxBlock.pnodeScopes);

if (this->pnodeBodyScope != nullptr)

{

fn(this->pnodeBodyScope->sxBlock.pnodeScopes);

}

}

};

struct PnClass

{

ParseNodePtr pnodeName;

ParseNodePtr pnodeDeclName;

ParseNodePtr pnodeBlock;

ParseNodePtr pnodeConstructor;

ParseNodePtr pnodeMembers;

ParseNodePtr pnodeStaticMembers;

ParseNodePtr pnodeExtends;

};

struct PnStrTemplate

{

ParseNodePtr pnodeStringLiterals;

ParseNodePtr pnodeStringRawLiterals;

ParseNodePtr pnodeSubstitutionExpressions;

uint16 countStringLiterals;

BYTE isTaggedTemplate:1;

};

struct PnProg : PnFnc

{

ParseNodePtr pnodeLastValStmt;

bool m\_UsesArgumentsAtGlobal;

};

struct PnCall

{

ParseNodePtr pnodeNext;

ParseNodePtr pnodeTarget;

ParseNodePtr pnodeArgs;

uint16 argCount;

uint16 spreadArgCount;

BYTE callOfConstants : 1;

BYTE isApplyCall : 1;

BYTE isEvalCall : 1;

};

struct PnStmt

{

ParseNodePtr pnodeOuter;

// Set by parsing code, used by code gen.

uint grfnop;

// Needed for byte code gen.

Js::ByteCodeLabel breakLabel;

Js::ByteCodeLabel continueLabel;

};

struct PnBlock : PnStmt

{

ParseNodePtr pnodeStmt;

ParseNodePtr pnodeLastValStmt;

ParseNodePtr pnodeLexVars;

ParseNodePtr pnodeScopes;

ParseNodePtr pnodeNext;

Scope \*scope;

ParseNodePtr enclosingBlock;

int blockId;

PnodeBlockType blockType:2;

BYTE callsEval:1;

BYTE childCallsEval:1;

void SetCallsEval(bool does) { callsEval = does; }

bool GetCallsEval() const { return callsEval; }

void SetChildCallsEval(bool does) { childCallsEval = does; }

bool GetChildCallsEval() const { return childCallsEval; }

void SetEnclosingBlock(ParseNodePtr pnode) { enclosingBlock = pnode; }

ParseNodePtr GetEnclosingBlock() const { return enclosingBlock; }

bool HasBlockScopedContent() const;

};

struct PnJump : PnStmt

{

ParseNodePtr pnodeTarget;

bool hasExplicitTarget;

};

struct PnLoop : PnStmt

{

// Needed for byte code gen

uint loopId;

};

struct PnWhile : PnLoop

{

ParseNodePtr pnodeCond;

ParseNodePtr pnodeBody;

};

struct PnWith : PnStmt

{

ParseNodePtr pnodeObj;

ParseNodePtr pnodeBody;

ParseNodePtr pnodeScopes;

ParseNodePtr pnodeNext;

Scope \*scope;

};

struct PnParamPattern

{

ParseNodePtr pnodeNext;

Js::RegSlot location;

ParseNodePtr pnode1;

};

struct PnIf : PnStmt

{

ParseNodePtr pnodeCond;

ParseNodePtr pnodeTrue;

ParseNodePtr pnodeFalse;

};

struct PnHelperCall2 {

ParseNodePtr pnodeArg1;

ParseNodePtr pnodeArg2;

int helperId;

};

struct PnForInOrForOf : PnLoop

{

ParseNodePtr pnodeObj;

ParseNodePtr pnodeBody;

ParseNodePtr pnodeLval;

ParseNodePtr pnodeBlock;

Js::RegSlot itemLocation;

};

struct PnFor : PnLoop

{

ParseNodePtr pnodeCond;

ParseNodePtr pnodeBody;

ParseNodePtr pnodeInit;

ParseNodePtr pnodeIncr;

ParseNodePtr pnodeBlock;

ParseNodePtr pnodeInverted;

};

struct PnSwitch : PnStmt

{

ParseNodePtr pnodeVal;

ParseNodePtr pnodeCases;

ParseNodePtr pnodeDefault;

ParseNodePtr pnodeBlock;

};

struct PnCase : PnStmt

{

ParseNodePtr pnodeNext;

ParseNodePtr pnodeExpr; // nullptr for default

ParseNodePtr pnodeBody;

Js::ByteCodeLabel labelCase;

};

struct PnReturn : PnStmt

{

ParseNodePtr pnodeExpr;

};

struct PnTryFinally : PnStmt

{

ParseNodePtr pnodeTry;

ParseNodePtr pnodeFinally;

};

struct PnTryCatch : PnStmt

{

ParseNodePtr pnodeTry;

ParseNodePtr pnodeCatch;

};

struct PnTry : PnStmt

{

ParseNodePtr pnodeBody;

};

struct PnCatch : PnStmt

{

ParseNodePtr pnodeNext;

ParseNodePtr pnodeParam;

ParseNodePtr pnodeBody;

ParseNodePtr pnodeScopes;

Scope \*scope;

};

struct PnFinally : PnStmt

{

ParseNodePtr pnodeBody;

};

struct ParseNode

{

OpCode nop;

ushort grfpn;

charcount\_t ichMin; // start offset into the original source buffer

charcount\_t ichLim; // end offset into the original source buffer

Js::RegSlot location;

bool isUsed; // indicates whether an expression such as x++ is used

bool emitLabels;

bool notEscapedUse; // Use by byte code generator. Currently, only used by child of knopComma

bool isInList;

bool isCallApplyTargetLoad;

#ifdef EDIT\_AND\_CONTINUE

ParseNodePtr parent;

#endif

union

{

PnArrLit sxArrLit; // Array literal

PnBin sxBin; // binary operators

PnBlock sxBlock; // block { }

PnCall sxCall; // function call

PnCase sxCase; // switch case

PnCatch sxCatch; // { catch(e : expr) {body} }

PnClass sxClass; // class declaration

PnFinally sxFinally; // finally

PnFlt sxFlt; // double constant

PnFnc sxFnc; // function declaration

PnFor sxFor; // for loop

PnForInOrForOf sxForInOrForOf; // for-in loop

PnHelperCall2 sxHelperCall2; // call to helper

PnIf sxIf; // if

PnInt sxInt; // integer constant

PnJump sxJump; // break and continue

PnLabel sxLabel; // label nodes

PnLoop sxLoop; // base for loop nodes

PnPid sxPid; // identifier or string

PnProg sxProg; // global program

PnReturn sxReturn; // return [expr]

PnStmt sxStmt; // base for statement nodes

PnStrTemplate sxStrTemplate; // string template declaration

PnSwitch sxSwitch; // switch

PnTri sxTri; // ternary operator

PnTry sxTry; // try-catch

PnTryCatch sxTryCatch; // try-catch

PnTryFinally sxTryFinally; // try-catch-finally

PnUni sxUni; // unary operators

PnVar sxVar; // variable declaration

PnWhile sxWhile; // while and do-while loops

PnWith sxWith; // with

PnParamPattern sxParamPattern; // Destructure pattern for function/catch parameter

};

IdentPtr name()

{

if (this->nop == knopName || this->nop == knopStr)

{

return this->sxPid.pid;

}

else if (this->nop == knopVarDecl)

{

return this->sxVar.pid;

}

else if (this->nop == knopConstDecl)

{

return this->sxVar.pid;

}

return nullptr;

}

static const uint mpnopgrfnop[knopLim];

static uint Grfnop(int nop)

{

Assert(nop < knopLim);

return nop < knopLim ? mpnopgrfnop[nop] : fnopNone;

}

BOOL IsStatement()

{

return (nop >= knopList && nop != knopLabel) || ((Grfnop(nop) & fnopAsg) != 0);

}

uint Grfnop(void)

{

Assert(nop < knopLim);

return nop < knopLim ? mpnopgrfnop[nop] : fnopNone;

}

charcount\_t LengthInCodepoints() const

{

return (this->ichLim - this->ichMin);

}

// This node is a function decl node and function has a var declaration named 'arguments',

bool HasVarArguments() const

{

return ((nop == knopFncDecl) && (grfpn & PNodeFlags::fpnArguments\_varDeclaration));

}

bool CapturesSyms() const

{

return (grfpn & PNodeFlags::fpnCapturesSyms) != 0;

}

void SetCapturesSyms()

{

grfpn |= PNodeFlags::fpnCapturesSyms;

}

bool IsInList() const { return this->isInList; }

void SetIsInList() { this->isInList = true; }

bool IsNotEscapedUse() const { return this->notEscapedUse; }

void SetNotEscapedUse() { this->notEscapedUse = true; }

bool CanFlattenConcatExpr() const { return !!(this->grfpn & PNodeFlags::fpnCanFlattenConcatExpr); }

bool IsCallApplyTargetLoad() { return isCallApplyTargetLoad; }

void SetIsCallApplyTargetLoad() { isCallApplyTargetLoad = true; }

bool IsVarLetOrConst() const

{

return this->nop == knopVarDecl || this->nop == knopLetDecl || this->nop == knopConstDecl;

}

ParseNodePtr GetFormalNext()

{

ParseNodePtr pnodeNext = nullptr;

if (nop == knopParamPattern)

{

pnodeNext = this->sxParamPattern.pnodeNext;

}

else

{

Assert(IsVarLetOrConst());

pnodeNext = this->sxVar.pnodeNext;

}

return pnodeNext;

}

bool IsPattern() const

{

return nop == knopObjectPattern || nop == knopArrayPattern;

}

#if DBG\_DUMP

void Dump();

#endif

};

const int kcbPnNone = offsetof(ParseNode, sxUni);

const int kcbPnArrLit = kcbPnNone + sizeof(PnArrLit);

const int kcbPnBin = kcbPnNone + sizeof(PnBin);

const int kcbPnBlock = kcbPnNone + sizeof(PnBlock);

const int kcbPnCall = kcbPnNone + sizeof(PnCall);

const int kcbPnCase = kcbPnNone + sizeof(PnCase);

const int kcbPnCatch = kcbPnNone + sizeof(PnCatch);

const int kcbPnClass = kcbPnNone + sizeof(PnClass);

const int kcbPnFinally = kcbPnNone + sizeof(PnFinally);

const int kcbPnFlt = kcbPnNone + sizeof(PnFlt);

const int kcbPnFnc = kcbPnNone + sizeof(PnFnc);

const int kcbPnFor = kcbPnNone + sizeof(PnFor);

const int kcbPnForIn = kcbPnNone + sizeof(PnForInOrForOf);

const int kcbPnForOf = kcbPnNone + sizeof(PnForInOrForOf);

const int kcbPnHelperCall3 = kcbPnNone + sizeof(PnHelperCall2);

const int kcbPnIf = kcbPnNone + sizeof(PnIf);

const int kcbPnInt = kcbPnNone + sizeof(PnInt);

const int kcbPnJump = kcbPnNone + sizeof(PnJump);

const int kcbPnLabel = kcbPnNone + sizeof(PnLabel);

const int kcbPnPid = kcbPnNone + sizeof(PnPid);

const int kcbPnProg = kcbPnNone + sizeof(PnProg);

const int kcbPnReturn = kcbPnNone + sizeof(PnReturn);

const int kcbPnSlot = kcbPnNone + sizeof(PnSlot);

const int kcbPnStrTemplate = kcbPnNone + sizeof(PnStrTemplate);

const int kcbPnSwitch = kcbPnNone + sizeof(PnSwitch);

const int kcbPnTri = kcbPnNone + sizeof(PnTri);

const int kcbPnTry = kcbPnNone + sizeof(PnTry);

const int kcbPnTryCatch = kcbPnNone + sizeof(PnTryCatch);

const int kcbPnTryFinally = kcbPnNone + sizeof(PnTryFinally);

const int kcbPnUni = kcbPnNone + sizeof(PnUni);

const int kcbPnUniSlot = kcbPnNone + sizeof(PnUniSlot);

const int kcbPnVar = kcbPnNone + sizeof(PnVar);

const int kcbPnWhile = kcbPnNone + sizeof(PnWhile);

const int kcbPnWith = kcbPnNone + sizeof(PnWith);

const int kcbPnParamPattern = kcbPnNone + sizeof(PnParamPattern);

#define AssertNodeMem(pnode) AssertPvCb(pnode, kcbPnNone)

#define AssertNodeMemN(pnode) AssertPvCbN(pnode, kcbPnNone)

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

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

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

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

// These byte codes \_cannot\_ appear in the regexOpCode field of a Node.

RegOp(GreedyStar ,Rev,Lng, 0 , 0 ) // 00/00 \* label, cchMinRem

RegOp(RevBranch ,Rev,Lng,Lng, 0 ) // 01/01 label, cchMinRem, pnode->cchMinTot

RegOp(GreedyLoop ,Rev,Lng,Lng,Lng) // 02/02 label, min, max, cchMinRem

RegOp(LoopInit ,Fwd,Int,Lng, 0 ) // 03/03 label, #, min

RegOp(GreedyLoopTest ,Rev,Int,Lng,Lng) // 04/04 label, #, max-min, cchMinRem

RegOp(NonGreedyStar ,Rev,Lng, 0 , 0 ) // 05 \* label, cchMinRem

RegOp(NonGreedyLoop ,Rev,Lng,Lng,Lng) // 06 label, min, max, cchMinRem

RegOp(NonGreedyLoopTest,Rev,Int,Lng,Lng) // 07 label, #, max-min, cchMinRem

RegOp(Open ,Int, 0 , 0 , 0 ) // 05/08 ( #

RegOp(Close ,Int, 0 , 0 , 0 ) // 06/09 ) #

RegOp(MatchOne ,Chr, 0 , 0 , 0 ) // 07/0A character

RegOp(Need ,Lng, 0 , 0 , 0 ) // 08/0B cchMinRem

RegOp(Fail , 0 , 0 , 0 , 0 ) // 09/0C no way to match this

RegOp(Jump ,Fwd, 0 , 0 , 0 ) // 0A/0D label

RegOp(PosLookahead ,Fwd, 0 , 0 , 0 ) // 0E label

RegOp(NegLookahead ,Fwd, 0 , 0 , 0 ) // 0F label

RegOp(LookaheadEnd , 0 , 0 , 0 , 0 ) // 10

RegOp(End , 0 , 0 , 0 , 0 ) // 0B/11

// These byte codes \_can\_ appear in the regexOpCode field of a Node.

RegOp(Branch ,Fwd, 0 , 0 , 0 ) // 0C/12 label

RegOp(Match ,Cch, 0 , 0 , 0 ) // 0D/13 cch, characters

RegOp(MatchGroup ,Int, 0 , 0 , 0 ) // 0E/14 \1 #

RegOp(Head , 0 , 0 , 0 , 0 ) // 0F/15 ^

RegOp(Tail , 0 , 0 , 0 , 0 ) // 10/16 $

RegOp(WordBound , 0 , 0 , 0 , 0 ) // 11/17 \b

RegOp(NotWordBound , 0 , 0 , 0 , 0 ) // 12/18 \B

RegOp(Any , 0 , 0 , 0 , 0 ) // 13/19 No longer used

RegOp(AnyOf ,Cch, 0 , 0 , 0 ) // 14/1A [] cch, transition characters

RegOp(AnyBut ,Cch, 0 , 0 , 0 ) // 15/1B [^] cch, transition characters

RegOp(Digit , 0 , 0 , 0 , 0 ) // 16/1C \d

RegOp(NotDigit , 0 , 0 , 0 , 0 ) // 17/1D \D

RegOp(Space , 0 , 0 , 0 , 0 ) // 18/1E \s

RegOp(NotSpace , 0 , 0 , 0 , 0 ) // 19/1F \S

RegOp(Letter , 0 , 0 , 0 , 0 ) // 1A/20 \w

RegOp(NotLetter , 0 , 0 , 0 , 0 ) // 1B/21 \W

RegOp(NotLF , 0 , 0 , 0 , 0 ) // 1C/22 .

RegOp(Dummy , 0 , 0 , 0 , 0 ) // 1D/23 Dummy. Default opcode to check for absence of opcode

#undef RegOp

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

// 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 "RegexFlags.h"

#include "Chars.h"

#include "CaseInsensitive.h"

#include "CharSet.h"

#include "CharMap.h"

#include "CharTrie.h"

#include "TextbookBoyerMoore.h"

#include "RegexRunTime.h"

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

// 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"

namespace UnifiedRegex

{

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

// Compiler (inlines etc)

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

uint8\* Compiler::Emit(size\_t size)

{

Assert(size <= UINT32\_MAX);

if (instLen - instNext < size)

{

CharCount newLen = max(instLen, initInstBufSize);

CharCount instLenPlus = (CharCount)(instLen + size - 1);

// check for overflow

if (instLenPlus < instLen || instLenPlus \* 2 < instLenPlus)

{

Js::Throw::OutOfMemory();

}

while (newLen <= instLenPlus)

{

newLen \*= 2;

}

instBuf = (uint8\*)ctAllocator->Realloc(instBuf, instLen, newLen);

instLen = newLen;

}

uint8\* inst = instBuf + instNext;

instNext += (CharCount)size;

return inst;

}

template <typename T>

T\* Compiler::Emit()

{

return new(Emit(sizeof(T))) T;

}

#define EMIT(compiler, T, ...) (new (compiler.Emit(sizeof(T))) T(\_\_VA\_ARGS\_\_))

#define L2I(O, label) LabelToInstPointer<O##Inst>(Inst::O, label)

// Remember: The machine address of an instruction is no longer valid after a subsequent emit,

// so all label fixups must be done using Compiler::GetFixup / Compiler::DoFixup

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

// Node

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

void Node::AppendLiteral(CharCount& litbufNext, CharCount litbufLen, \_\_inout\_ecount(litbufLen) Char\* litbuf) const

{

Assert(false);

}

CharCount Node::EmitScanFirstSet(Compiler& compiler)

{

Assert(prevConsumes.IsExact(0));

if (thisConsumes.CouldMatchEmpty())

// Can't be sure of consuming something in FIRST

return 0;

if (firstSet->Count() > maxSyncToSetSize)

// HEURISTIC: If FIRST is large we'll get too many false positives

return 0;

//

// Compilation scheme:

//

// SyncTo(Char|Char2|Set)And(Consume|Continue)

//

Char entries[CharSet<Char>::MaxCompact];

int count = firstSet->GetCompactEntries(2, entries);

if (SupportsPrefixSkipping(compiler))

{

if (count == 1)

EMIT(compiler, SyncToCharAndConsumeInst, entries[0]);

else if (count == 2)

EMIT(compiler, SyncToChar2SetAndConsumeInst, entries[0], entries[1]);

else

EMIT(compiler, SyncToSetAndConsumeInst<false>)->set.CloneFrom(compiler.rtAllocator, \*firstSet);

return 1;

}

else

{

if (count == 1)

EMIT(compiler, SyncToCharAndContinueInst, entries[0]);

else if (count == 2)

EMIT(compiler, SyncToChar2SetAndContinueInst, entries[0], entries[1]);

else

EMIT(compiler, SyncToSetAndContinueInst<false>)->set.CloneFrom(compiler.rtAllocator, \*firstSet);

return 0;

}

}

bool Node::IsBetterSyncronizingNode(Compiler& compiler, Node\* curr, Node\* proposed)

{

int proposedNumLiterals = 0;

CharCount proposedLength = proposed->MinSyncronizingLiteralLength(compiler, proposedNumLiterals);

if (proposedLength == 0 || proposedNumLiterals > maxNumSyncLiterals)

// Not a synchronizable node or too many literals.

return false;

if (curr == nullptr)

// We'll take whatever we can get

return true;

int currNumLiterals = 0;

CharCount currLength = curr->MinSyncronizingLiteralLength(compiler, currNumLiterals);

// Lexicographic ordering based on

// - whether literal length is above a threshold (above is better)

// - number of literals (smaller is better)

// - upper bound on backup (finite is better)

// - minimum literal length (longer is better)

// - actual backup upper bound (shorter is better)

if (proposedLength >= preferredMinSyncToLiteralLength

&& currLength < preferredMinSyncToLiteralLength)

{

return true;

}

if (proposedLength < preferredMinSyncToLiteralLength

&& currLength >= preferredMinSyncToLiteralLength)

{

return false;

}

if (proposedNumLiterals < currNumLiterals)

return true;

if (proposedNumLiterals > currNumLiterals)

return false;

if (!proposed->prevConsumes.IsUnbounded() && curr->prevConsumes.IsUnbounded())

return true;

if (proposed->prevConsumes.IsUnbounded() && !curr->prevConsumes.IsUnbounded())

return false;

if (proposedLength > currLength)

return true;

if (proposedLength < currLength)

return false;

return proposed->prevConsumes.upper < curr->prevConsumes.upper;

}

bool Node::IsSingleChar(Compiler& compiler, Char& outChar) const

{

if (tag != Node::MatchChar)

return false;

const MatchCharNode\* node = (const MatchCharNode\*)this;

if (node->isEquivClass)

return false;

outChar = node->cs[0];

return true;

}

bool Node::IsBoundedWord(Compiler& compiler) const

{

if (tag != Node::Concat)

return false;

const ConcatNode\* concatNode = (const ConcatNode \*)this;

if (concatNode->head->tag != Node::WordBoundary ||

concatNode->tail == 0 ||

concatNode->tail->head->tag != Node::Loop ||

concatNode->tail->tail == 0 ||

concatNode->tail->tail->head->tag != Node::WordBoundary ||

concatNode->tail->tail->tail != 0)

return false;

const WordBoundaryNode\* enter = (const WordBoundaryNode\*)concatNode->head;

const LoopNode\* loop = (const LoopNode\*)concatNode->tail->head;

const WordBoundaryNode\* leave = (const WordBoundaryNode\*)concatNode->tail->tail->head;

if (enter->isNegation ||

!loop->isGreedy ||

loop->repeats.lower != 1 ||

loop->repeats.upper != CharCountFlag ||

loop->body->tag != Node::MatchSet ||

leave->isNegation)

return false;

const MatchSetNode\* wordSet = (const MatchSetNode\*)loop->body;

if (wordSet->isNegation)

return false;

return wordSet->set.IsEqualTo(\*compiler.standardChars->GetWordSet());

}

bool Node::IsBOILiteral2(Compiler& compiler) const

{

if (tag != Node::Concat)

return false;

const ConcatNode\* concatNode = (const ConcatNode \*)this;

if ((compiler.program->flags & (IgnoreCaseRegexFlag | MultilineRegexFlag)) != 0 ||

concatNode->head->tag != Node::BOL ||

concatNode->tail == nullptr ||

concatNode->tail->head->tag != Node::MatchLiteral ||

concatNode->tail->tail != nullptr ||

((MatchLiteralNode \*)concatNode->tail->head)->isEquivClass ||

((MatchLiteralNode \*)concatNode->tail->head)->length != 2)

{

return false;

}

return true;

}

bool Node::IsLeadingTrailingSpaces(Compiler& compiler, CharCount& leftMinMatch, CharCount& rightMinMatch) const

{

if (tag != Node::Alt)

return false;

if (compiler.program->flags & MultilineRegexFlag)

return false;

const AltNode\* altNode = (const AltNode\*)this;

if (altNode->head->tag != Node::Concat ||

altNode->tail == 0 ||

altNode->tail->head->tag != Node::Concat ||

altNode->tail->tail != 0)

return false;

const ConcatNode\* left = (const ConcatNode\*)altNode->head;

const ConcatNode\* right = (const ConcatNode\*)altNode->tail->head;

if (left->head->tag != Node::BOL ||

left->tail == 0 ||

left->tail->head->tag != Node::Loop ||

left->tail->tail != 0)

return false;

if (right->head->tag != Node::Loop ||

right->tail == 0 ||

right->tail->head->tag != Node::EOL ||

right->tail->tail != 0)

return false;

const LoopNode\* leftLoop = (const LoopNode\*)left->tail->head;

const LoopNode\* rightLoop = (const LoopNode\*)right->head;

if (!leftLoop->isGreedy ||

leftLoop->repeats.upper != CharCountFlag ||

leftLoop->body->tag != Node::MatchSet ||

!rightLoop->isGreedy ||

rightLoop->repeats.upper != CharCountFlag ||

rightLoop->body->tag != Node::MatchSet)

return false;

const MatchSetNode\* leftSet = (const MatchSetNode\*)leftLoop->body;

const MatchSetNode\* rightSet = (const MatchSetNode\*)rightLoop->body;

if (leftSet->isNegation ||

rightSet->isNegation)

return false;

leftMinMatch = leftLoop->repeats.lower;

rightMinMatch = rightLoop->repeats.lower;

return

leftSet->set.IsEqualTo(\*compiler.standardChars->GetWhitespaceSet()) &&

rightSet->set.IsEqualTo(\*compiler.standardChars->GetWhitespaceSet());

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Node::PrintAnnotations(DebugWriter\* w) const

{

if (firstSet != 0)

{

w->PrintEOL(L"<");

w->Indent();

w->Print(L"features: {");

bool first = true;

for (uint i = Empty; i <= Assertion; i++)

{

if ((features & (1 << i)) != 0)

{

if (first)

first = false;

else

w->Print(L",");

switch (i)

{

case Empty: w->Print(L"Empty"); break;

case BOL: w->Print(L"BOL"); break;

case EOL: w->Print(L"EOL"); break;

case WordBoundary: w->Print(L"WordBoundary"); break;

case MatchLiteral: w->Print(L"MatchLiteral"); break;

case MatchChar: w->Print(L"MatchChar"); break;

case Concat: w->Print(L"Concat"); break;

case Alt: w->Print(L"Alt"); break;

case DefineGroup: w->Print(L"DefineGroup"); break;

case MatchGroup: w->Print(L"MatchGroup"); break;

case Loop: w->Print(L"Loop"); break;

case MatchSet: w->Print(L"MatchSet"); break;

case Assertion: w->Print(L"Assertion"); break;

}

}

}

w->PrintEOL(L"}");

w->Print(L"firstSet: ");

firstSet->Print(w);

if (isFirstExact)

w->Print(L" (exact)");

w->EOL();

w->Print(L"followSet: ");

followSet->Print(w);

w->EOL();

w->Print(L"prevConsumes: ");

prevConsumes.Print(w);

w->EOL();

w->Print(L"thisConsumes: ");

thisConsumes.Print(w);

w->EOL();

w->Print(L"followConsumes: ");

followConsumes.Print(w);

w->EOL();

w->PrintEOL(L"isThisIrrefutable: %s", isThisIrrefutable ? L"true" : L"false");

w->PrintEOL(L"isFollowIrrefutable: %s", isFollowIrrefutable ? L"true" : L"false");

w->PrintEOL(L"isWord: %s", isWord ? L"true" : L"false");

w->PrintEOL(L"isThisWillNotProgress: %s", isThisWillNotProgress ? L"true" : L"false");

w->PrintEOL(L"isThisWillNotRegress: %s", isThisWillNotRegress ? L"true" : L"false");

w->PrintEOL(L"isPrevWillNotProgress: %s", isPrevWillNotProgress ? L"true" : L"false");

w->PrintEOL(L"isPrevWillNotRegress: %s", isPrevWillNotRegress ? L"true" : L"false");

w->PrintEOL(L"isDeterministic: %s", isDeterministic ? L"true" : L"false");

w->PrintEOL(L"isNotInLoop: %s", isNotInLoop ? L"true" : L"false");

w->PrintEOL(L"isNotNegated: %s", isNotNegated ? L"true" : L"false");

w->PrintEOL(L"isAtLeastOnce: %s", isAtLeastOnce ? L"true" : L"false");

w->PrintEOL(L"hasInitialHardFailBOI: %s", hasInitialHardFailBOI ? L"true" : L"false");

w->Unindent();

w->PrintEOL(L">");

}

}

#endif

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

// SimpleNode

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

CharCount SimpleNode::LiteralLength() const

{

return 0;

}

bool SimpleNode::IsCharOrPositiveSet() const

{

return false;

}

CharCount SimpleNode::TransferPass0(Compiler& compiler, const Char\* litbuf)

{

return 0;

}

void SimpleNode::TransferPass1(Compiler& compiler, const Char\* litbuf)

{

}

bool SimpleNode::IsRefiningAssertion(Compiler& compiler)

{

return tag == EOL && (compiler.program->flags & MultilineRegexFlag) != 0;

}

void SimpleNode::AnnotatePass0(Compiler& compiler)

{

isWord = false;

}

void SimpleNode::AnnotatePass1(Compiler& compiler, bool parentNotInLoop, bool parentAtLeastOnce, bool parentNotSpeculative, bool parentNotNegated)

{

isFirstExact = false;

thisConsumes.Exact(0);

isThisWillNotProgress = true;

isThisWillNotRegress = true;

isNotInLoop = parentNotInLoop;

isAtLeastOnce = parentAtLeastOnce;

isNotSpeculative = parentNotSpeculative;

isNotNegated = parentNotNegated;

switch (tag)

{

case Empty:

features = HasEmpty;

firstSet = compiler.standardChars->GetEmptySet();

isThisIrrefutable = true;

break;

case BOL:

features = HasBOL;

firstSet = compiler.standardChars->GetFullSet();

isThisIrrefutable = false;

break;

case EOL:

features = HasEOL;

if ((compiler.program->flags & MultilineRegexFlag) != 0)

firstSet = compiler.standardChars->GetNewlineSet();

else

firstSet = compiler.standardChars->GetEmptySet();

isThisIrrefutable = false;

break;

default:

Assert(false);

}

}

void SimpleNode::AnnotatePass2(Compiler& compiler, CountDomain accumConsumes, bool accumPrevWillNotProgress, bool accumPrevWillNotRegress)

{

prevConsumes = accumConsumes;

isPrevWillNotProgress = accumPrevWillNotProgress;

isPrevWillNotRegress = accumPrevWillNotRegress;

}

void SimpleNode::AnnotatePass3(Compiler& compiler, CountDomain accumConsumes, CharSet<Char>\* accumFollow, bool accumFollowIrrefutable, bool accumFollowEOL)

{

followConsumes = accumConsumes;

followSet = accumFollow;

isFollowIrrefutable = accumFollowIrrefutable;

isFollowEOL = accumFollowEOL;

hasInitialHardFailBOI = ((tag == BOL) &&

prevConsumes.IsExact(0) &&

(compiler.program->flags & MultilineRegexFlag) == 0 &&

isAtLeastOnce &&

isNotNegated &&

isPrevWillNotRegress);

}

void SimpleNode::AnnotatePass4(Compiler& compiler)

{

isDeterministic = true;

}

bool SimpleNode::SupportsPrefixSkipping(Compiler& compiler) const

{

return false;

}

Node\* SimpleNode::HeadSyncronizingNode(Compiler& compiler)

{

return 0;

}

CharCount SimpleNode::MinSyncronizingLiteralLength(Compiler& compiler, int& numLiterals) const

{

return 0;

}

void SimpleNode::CollectSyncronizingLiterals(Compiler& compiler, ScannersMixin& scanners) const

{

Assert(false);

}

void SimpleNode::BestSyncronizingNode(Compiler& compiler, Node\*& bestNode)

{

}

void SimpleNode::AccumDefineGroups(Js::ScriptContext\* scriptContext, int& minGroup, int& maxGroup)

{

}

void SimpleNode::Emit(Compiler& compiler, CharCount& skipped)

{

Assert(skipped == 0);

switch (tag)

{

case Empty:

// Nothing

break;

case BOL:

{

if ((compiler.program->flags & MultilineRegexFlag) != 0)

{

//

// Compilation scheme:

//

// BOLTest

//

EMIT(compiler, BOLTestInst);

}

else

{

if (compiler.CurrentLabel() == 0)

{

// The first instruction is BOI, change the tag and only execute it once

// without looping every start position

compiler.SetBOIInstructionsProgramTag();

}

else

{

//

// Compilation scheme:

//

// BOITest

//

// Obviously starting later in the string won't help, so can hard fail if:

// - this pattern must always be matched

// - not in an negative assertion

// - backtracking could never rewind the input pointer

//

EMIT(compiler, BOITestInst, isAtLeastOnce && isNotNegated && isPrevWillNotRegress);

}

}

break;

}

case EOL:

{

if ((compiler.program->flags & MultilineRegexFlag) != 0)

//

// Compilation scheme:

//

// EOLTest

//

EMIT(compiler, EOLTestInst);

else

//

// Compilation scheme:

//

// EOITest

//

// Can hard fail if

// - this pattern must always be matched

// - not in an negative assertion

// - backtracking could never advance the input pointer

//

EMIT(compiler, EOITestInst, isAtLeastOnce && isNotNegated && isPrevWillNotProgress);

break;

}

default:

Assert(false);

}

}

CharCount SimpleNode::EmitScan(Compiler& compiler, bool isHeadSyncronizingNode)

{

Assert(false);

return 0;

}

bool SimpleNode::IsOctoquad(Compiler& compiler, OctoquadIdentifier\* oi)

{

return false;

}

bool SimpleNode::IsCharTrieArm(Compiler& compiler, uint& accNumAlts) const

{

return tag == Empty;

}

bool SimpleNode::BuildCharTrie(Compiler& compiler, CharTrie\* trie, Node\* cont, bool isAcceptFirst) const

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

Assert(tag == Empty);

if (cont == 0)

{

if (trie->Count() > 0)

// This literal is a proper prefix of an earlier literal

return false;

trie->SetAccepting();

return true;

}

return cont->BuildCharTrie(compiler, trie, 0, isAcceptFirst);

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void SimpleNode::Print(DebugWriter\* w, const Char\* litbuf) const

{

switch (tag)

{

case Empty:

w->Print(L"Empty"); break;

case BOL:

w->Print(L"BOL"); break;

case EOL:

w->Print(L"EOL"); break;

default:

Assert(false);

}

w->PrintEOL(L"()");

PrintAnnotations(w);

}

#endif

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

// WordBoundaryNode

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

CharCount WordBoundaryNode::LiteralLength() const

{

return 0;

}

bool WordBoundaryNode::IsCharOrPositiveSet() const

{

return false;

}

CharCount WordBoundaryNode::TransferPass0(Compiler& compiler, const Char\* litbuf)

{

return 0;

}

void WordBoundaryNode::TransferPass1(Compiler& compiler, const Char\* litbuf)

{

// WordChars and NonWordChars sets are already case invariant

}

bool WordBoundaryNode::IsRefiningAssertion(Compiler& compiler)

{

return mustIncludeEntering != mustIncludeLeaving;

}

void WordBoundaryNode::AnnotatePass0(Compiler& compiler)

{

isWord = false;

}

void WordBoundaryNode::AnnotatePass1(Compiler& compiler, bool parentNotInLoop, bool parentAtLeastOnce, bool parentNotSpeculative, bool parentNotNegated)

{

features = HasWordBoundary;

thisConsumes.Exact(0);

isFirstExact = false;

isThisIrrefutable = false;

isThisWillNotProgress = true;

isThisWillNotRegress = true;

isNotInLoop = parentNotInLoop;

isAtLeastOnce = parentAtLeastOnce;

isNotSpeculative = parentNotSpeculative;

isNotNegated = parentNotNegated;

if (isNegation)

firstSet = compiler.standardChars->GetFullSet();

else

{

if (mustIncludeEntering && !mustIncludeLeaving)

firstSet = compiler.standardChars->GetWordSet();

else if (mustIncludeLeaving && !mustIncludeEntering)

firstSet = compiler.standardChars->GetNonWordSet();

else

firstSet = compiler.standardChars->GetFullSet();

}

}

void WordBoundaryNode::AnnotatePass2(Compiler& compiler, CountDomain accumConsumes, bool accumPrevWillNotProgress, bool accumPrevWillNotRegress)

{

prevConsumes = accumConsumes;

isPrevWillNotProgress = accumPrevWillNotProgress;

isPrevWillNotRegress = accumPrevWillNotRegress;

}

void WordBoundaryNode::AnnotatePass3(Compiler& compiler, CountDomain accumConsumes, CharSet<Char>\* accumFollow, bool accumFollowIrrefutable, bool accumFollowEOL)

{

followConsumes = accumConsumes;

followSet = accumFollow;

isFollowIrrefutable = accumFollowIrrefutable;

isFollowEOL = accumFollowEOL;

}

void WordBoundaryNode::AnnotatePass4(Compiler& compiler)

{

isDeterministic = true;

}

bool WordBoundaryNode::SupportsPrefixSkipping(Compiler& compiler) const

{

return false;

}

Node\* WordBoundaryNode::HeadSyncronizingNode(Compiler& compiler)

{

return 0;

}

CharCount WordBoundaryNode::MinSyncronizingLiteralLength(Compiler& compiler, int& numLiterals) const

{

return 0;

}

void WordBoundaryNode::CollectSyncronizingLiterals(Compiler& compiler, ScannersMixin& scanners) const

{

Assert(false);

}

void WordBoundaryNode::BestSyncronizingNode(Compiler& compiler, Node\*& bestNode)

{

}

void WordBoundaryNode::AccumDefineGroups(Js::ScriptContext\* scriptContext, int& minGroup, int& maxGroup)

{

}

void WordBoundaryNode::Emit(Compiler& compiler, CharCount& skipped)

{

Assert(skipped == 0);

//

// Compilation scheme:

//

// WordBoundaryTest

//

EMIT(compiler, WordBoundaryTestInst, isNegation);

}

CharCount WordBoundaryNode::EmitScan(Compiler& compiler, bool isHeadSyncronizingNode)

{

Assert(false);

return 0;

}

bool WordBoundaryNode::IsOctoquad(Compiler& compiler, OctoquadIdentifier\* oi)

{

return false;

}

bool WordBoundaryNode::IsCharTrieArm(Compiler& compiler, uint& accNumAlts) const

{

return false;

}

bool WordBoundaryNode::BuildCharTrie(Compiler& compiler, CharTrie\* trie, Node\* cont, bool isAcceptFirst) const

{

Assert(false);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void WordBoundaryNode::Print(DebugWriter\* w, const Char\* litbuf) const

{

w->PrintEOL(L"WordBoundary(%s, %s, %s)", isNegation ? L"negative" : L"positive", mustIncludeEntering ? L"entering" : L"-", mustIncludeLeaving ? L"leaving" : L"-");

PrintAnnotations(w);

}

#endif

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

// MatchLiteralNode

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

CharCount MatchLiteralNode::LiteralLength() const

{

return length;

}

void MatchLiteralNode::AppendLiteral(CharCount& litbufNext, CharCount litbufLen, \_\_inout\_ecount(litbufLen) Char\* litbuf) const

{

// Called during parsing only, so literal always in original form

Assert(!isEquivClass);

Assert(litbufNext + length <= litbufLen && offset + length <= litbufLen);

#pragma prefast(suppress:26000, "The error said that offset + length >= litbufLen + 1, which is incorrect due to if statement below.")

if (litbufNext + length <= litbufLen && offset + length <= litbufLen) // for prefast

{

js\_wmemcpy\_s(litbuf + litbufNext, litbufLen - litbufNext, litbuf + offset, length);

}

litbufNext += length;

}

bool MatchLiteralNode::IsCharOrPositiveSet() const

{

return false;

}

CharCount MatchLiteralNode::TransferPass0(Compiler& compiler, const Char\* litbuf)

{

Assert(length > 1);

if ((compiler.program->flags & IgnoreCaseRegexFlag) != 0

&& !compiler.standardChars->IsTrivialString(compiler.program->GetCaseMappingSource(), litbuf + offset, length))

{

// We'll need to expand each character of literal into its equivalence class

isEquivClass = true;

return length \* CaseInsensitive::EquivClassSize;

}

else

return length;

}

void MatchLiteralNode::TransferPass1(Compiler& compiler, const Char\* litbuf)

{

CharCount nextLit = compiler.program->rep.insts.litbufLen;

if (isEquivClass)

{

Assert((compiler.program->flags & IgnoreCaseRegexFlag) != 0);

// Expand literal according to character equivalence classes

for (CharCount i = 0; i < length; i++)

{

compiler.standardChars->ToEquivs(

compiler.program->GetCaseMappingSource(),

litbuf[offset + i],

compiler.program->rep.insts.litbuf + nextLit + i \* CaseInsensitive::EquivClassSize);

}

compiler.program->rep.insts.litbufLen += length \* CaseInsensitive::EquivClassSize;

}

else

{

for (CharCount i = 0; i < length; i++)

compiler.program->rep.insts.litbuf[nextLit + i] = litbuf[offset + i];

compiler.program->rep.insts.litbufLen += length;

}

offset = nextLit;

}

void MatchLiteralNode::AnnotatePass0(Compiler& compiler)

{

const Char\* litbuf = compiler.program->rep.insts.litbuf;

for (CharCount i = offset; i < offset + length; i++)

{

if (!compiler.standardChars->IsWord(litbuf[i]))

{

isWord = false;

return;

}

}

isWord = true;

}

bool MatchLiteralNode::IsRefiningAssertion(Compiler& compiler)

{

return false;

}

void MatchLiteralNode::AnnotatePass1(Compiler& compiler, bool parentNotInLoop, bool parentAtLeastOnce, bool parentNotSpeculative, bool parentNotNegated)

{

features = HasMatchLiteral;

thisConsumes.Exact(length);

firstSet = Anew(compiler.ctAllocator, UnicodeCharSet);

for (int i = 0; i < (isEquivClass ? CaseInsensitive::EquivClassSize : 1); i++)

firstSet->Set(compiler.ctAllocator, compiler.program->rep.insts.litbuf[offset + i]);

isFirstExact = true;

isThisIrrefutable = false;

isThisWillNotProgress = true;

isThisWillNotRegress = true;

isNotInLoop = parentNotInLoop;

isAtLeastOnce = parentAtLeastOnce;

isNotSpeculative = parentNotSpeculative;

isNotNegated = parentNotNegated;

}

void MatchLiteralNode::AnnotatePass2(Compiler& compiler, CountDomain accumConsumes, bool accumPrevWillNotProgress, bool accumPrevWillNotRegress)

{

prevConsumes = accumConsumes;

isPrevWillNotProgress = accumPrevWillNotProgress;

isPrevWillNotRegress = accumPrevWillNotRegress;

}

void MatchLiteralNode::AnnotatePass3(Compiler& compiler, CountDomain accumConsumes, CharSet<Char>\* accumFollow, bool accumFollowIrrefutable, bool accumFollowEOL)

{

followConsumes = accumConsumes;

followSet = accumFollow;

isFollowIrrefutable = accumFollowIrrefutable;

isFollowEOL = accumFollowEOL;

}

void MatchLiteralNode::AnnotatePass4(Compiler& compiler)

{

isDeterministic = true;

}

bool MatchLiteralNode::SupportsPrefixSkipping(Compiler& compiler) const

{

return true;

}

Node\* MatchLiteralNode::HeadSyncronizingNode(Compiler& compiler)

{

return this;

}

CharCount MatchLiteralNode::MinSyncronizingLiteralLength(Compiler& compiler, int& numLiterals) const

{

numLiterals++;

return length;

}

void MatchLiteralNode::CollectSyncronizingLiterals(Compiler& compiler, ScannersMixin& scanners) const

{

ScannerMixin\* scanner =

scanners.Add(compiler.GetScriptContext()->GetRecycler(), compiler.GetProgram(), offset, length, isEquivClass);

scanner->scanner.Setup(compiler.rtAllocator, compiler.program->rep.insts.litbuf + offset, length, isEquivClass ? CaseInsensitive::EquivClassSize : 1);

}

void MatchLiteralNode::BestSyncronizingNode(Compiler& compiler, Node\*& bestNode)

{

if (IsBetterSyncronizingNode(compiler, bestNode, this))

bestNode = this;

}

void MatchLiteralNode::AccumDefineGroups(Js::ScriptContext\* scriptContext, int& minGroup, int& maxGroup)

{

}

void MatchLiteralNode::Emit(Compiler& compiler, CharCount& skipped)

{

if (skipped >= length)

{

// Asking to skip entire literal

skipped -= length;

return;

}

//

// Compilation scheme:

//

// Match(Char|Char4|Literal|LiteralEquiv)Inst

//

CharCount effectiveOffset = offset + skipped \* (isEquivClass ? CaseInsensitive::EquivClassSize : 1);

CharCount effectiveLength = length - skipped;

skipped -= min(skipped, length);

if (effectiveLength == 1)

{

Char\* cs = compiler.program->rep.insts.litbuf + effectiveOffset;

MatchCharNode::Emit(compiler, cs, isEquivClass);

}

else

{

if (isEquivClass)

EMIT(compiler, MatchLiteralEquivInst, effectiveOffset, effectiveLength);

else

EMIT(compiler, MatchLiteralInst, effectiveOffset, effectiveLength);

}

}

CompileAssert(CaseInsensitive::EquivClassSize == 4);

CharCount MatchLiteralNode::EmitScan(Compiler& compiler, bool isHeadSyncronizingNode)

{

//

// Compilation scheme:

//

// SyncTo(Literal|LiteralEquiv|LinearLiteral)And(Continue|Consume|Backup)

//

Char \* litptr = compiler.program->rep.insts.litbuf + offset;

if (isHeadSyncronizingNode)

{

// For a head literal there's no need to back up after finding the literal, so use a faster instruction

Assert(prevConsumes.IsExact(0)); // there should not be any consumes before this node

if (isEquivClass)

{

const uint lastPatCharIndex = length - 1;

if (litptr[lastPatCharIndex \* CaseInsensitive::EquivClassSize] == litptr[lastPatCharIndex \* CaseInsensitive::EquivClassSize + 1]

&& litptr[lastPatCharIndex \* CaseInsensitive::EquivClassSize] == litptr[lastPatCharIndex \* CaseInsensitive::EquivClassSize + 2]

&& litptr[lastPatCharIndex \* CaseInsensitive::EquivClassSize] == litptr[lastPatCharIndex \* CaseInsensitive::EquivClassSize + 3])

{

EMIT(compiler, SyncToLiteralEquivTrivialLastPatCharAndConsumeInst, offset, length)->scanner.Setup(compiler.rtAllocator, litptr, length, CaseInsensitive::EquivClassSize);

}

else

{

EMIT(compiler, SyncToLiteralEquivAndConsumeInst, offset, length)->scanner.Setup(compiler.rtAllocator, litptr, length, CaseInsensitive::EquivClassSize);

}

}

else if (length == 1)

EMIT(compiler, SyncToCharAndConsumeInst, litptr[0]);

else if (length == 2)

EMIT(compiler, SyncToChar2LiteralAndConsumeInst, litptr[0], litptr[1]);

else

{

TextbookBoyerMooreSetup<wchar\_t> setup(litptr, length);

switch (setup.GetScheme())

{

case TextbookBoyerMooreSetup<wchar\_t>::LinearScheme:

EMIT(compiler, SyncToLinearLiteralAndConsumeInst, offset, length)->scanner.Setup(compiler.rtAllocator, setup);

break;

case TextbookBoyerMooreSetup<wchar\_t>::DefaultScheme:

EMIT(compiler, SyncToLiteralAndConsumeInst, offset, length)->scanner.Setup(compiler.rtAllocator, setup);

break;

};

}

return length;

}

else

{

// We're synchronizing on a non-head literal so we may need to back up. Or if we're syncing to the first literal

// inside a group for instance, then we won't need to back up but we cannot consume the literal.

if (prevConsumes.IsExact(0))

{

if (isEquivClass)

{

const uint lastPatCharIndex = length - 1;

if (litptr[lastPatCharIndex \* CaseInsensitive::EquivClassSize] == litptr[lastPatCharIndex \* CaseInsensitive::EquivClassSize + 1]

&& litptr[lastPatCharIndex \* CaseInsensitive::EquivClassSize] == litptr[lastPatCharIndex \* CaseInsensitive::EquivClassSize + 2]

&& litptr[lastPatCharIndex \* CaseInsensitive::EquivClassSize] == litptr[lastPatCharIndex \* CaseInsensitive::EquivClassSize + 3])

{

EMIT(compiler, SyncToLiteralEquivTrivialLastPatCharAndContinueInst, offset, length)->scanner.Setup(compiler.rtAllocator, litptr, length, CaseInsensitive::EquivClassSize);

}

else

{

EMIT(compiler, SyncToLiteralEquivAndContinueInst, offset, length)->scanner.Setup(compiler.rtAllocator, litptr, length, CaseInsensitive::EquivClassSize);

}

}

else if (length == 1)

EMIT(compiler, SyncToCharAndContinueInst, litptr[0]);

else if (length == 2)

EMIT(compiler, SyncToChar2LiteralAndContinueInst, litptr[0], litptr[1]);

else

{

TextbookBoyerMooreSetup<wchar\_t> setup(litptr, length);

switch (setup.GetScheme())

{

case TextbookBoyerMooreSetup<wchar\_t>::LinearScheme:

EMIT(compiler, SyncToLinearLiteralAndContinueInst, offset, length)->scanner.Setup(compiler.rtAllocator, setup);

break;

case TextbookBoyerMooreSetup<wchar\_t>::DefaultScheme:

EMIT(compiler, SyncToLiteralAndContinueInst, offset, length)->scanner.Setup(compiler.rtAllocator, setup);

break;

};

}

}

else

{

if (isEquivClass)

{

const uint lastPatCharIndex = length - 1;

if (litptr[lastPatCharIndex \* CaseInsensitive::EquivClassSize] == litptr[lastPatCharIndex \* CaseInsensitive::EquivClassSize + 1]

&& litptr[lastPatCharIndex \* CaseInsensitive::EquivClassSize] == litptr[lastPatCharIndex \* CaseInsensitive::EquivClassSize + 2]

&& litptr[lastPatCharIndex \* CaseInsensitive::EquivClassSize] == litptr[lastPatCharIndex \* CaseInsensitive::EquivClassSize + 3])

{

EMIT(compiler, SyncToLiteralEquivTrivialLastPatCharAndBackupInst, offset, length, prevConsumes)->scanner.Setup(compiler.rtAllocator, litptr, length, CaseInsensitive::EquivClassSize);

}

else

{

EMIT(compiler, SyncToLiteralEquivAndBackupInst, offset, length, prevConsumes)->scanner.Setup(compiler.rtAllocator, litptr, length, CaseInsensitive::EquivClassSize);

}

}

else if (length == 1)

EMIT(compiler, SyncToCharAndBackupInst, litptr[0], prevConsumes);

else if (length == 2)

EMIT(compiler, SyncToChar2LiteralAndBackupInst, litptr[0], litptr[1], prevConsumes);

else

{

TextbookBoyerMooreSetup<wchar\_t> setup(litptr, length);

switch (setup.GetScheme())

{

case TextbookBoyerMooreSetup<wchar\_t>::LinearScheme:

EMIT(compiler, SyncToLinearLiteralAndBackupInst, offset, length, prevConsumes)->scanner.Setup(compiler.rtAllocator, setup);

break;

case TextbookBoyerMooreSetup<wchar\_t>::DefaultScheme:

EMIT(compiler, SyncToLiteralAndBackupInst, offset, length, prevConsumes)->scanner.Setup(compiler.rtAllocator, setup);

break;

};

}

}

return 0;

}

}

bool MatchLiteralNode::IsOctoquad(Compiler& compiler, OctoquadIdentifier\* oi)

{

// We look for octoquad patterns before converting for case-insensitivity

Assert(!isEquivClass);

if (!oi->CouldAppend(length))

return false;

for (CharCount i = 0; i < length; i++)

{

if (!oi->AppendChar(compiler.standardChars->ToCanonical(compiler.program->GetCaseMappingSource(), compiler.program->rep.insts.litbuf[offset + i])))

return false;

}

return true;

}

bool MatchLiteralNode::IsCharTrieArm(Compiler& compiler, uint& accNumAlts) const

{

if (isEquivClass)

// The literal would expand into length^3 alternatives

return false;

return true;

}

bool MatchLiteralNode::BuildCharTrie(Compiler& compiler, CharTrie\* trie, Node\* cont, bool isAcceptFirst) const

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

Assert(!isEquivClass);

CharTrie\* tail = trie;

for (CharCount i = 0; i < length; i++)

{

if (tail->IsAccepting())

{

// An earlier literal is a prefix of this literal

// If isAcceptFirst, can ignore suffix of already recognized literal.

// Otherwise, must fail.

return isAcceptFirst;

}

CharTrie\* newTail = tail->Add(compiler.ctAllocator, compiler.program->rep.insts.litbuf[offset + i]);

if (tail->Count() > maxTrieArmExpansion)

return false;

tail = newTail;

}

if (cont == 0)

{

if (tail->Count() > 0)

// This literal is a proper prefix of an earlier literal

return false;

tail->SetAccepting();

}

else

{

if (!cont->BuildCharTrie(compiler, tail, 0, isAcceptFirst))

return false;

}

return true;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void MatchLiteralNode::Print(DebugWriter\* w, const Char\* litbuf) const

{

w->Print(L"MatchLiteral(");

int skip = isEquivClass ? CaseInsensitive::EquivClassSize : 1;

for (int i = 0; i < skip; i++)

{

if (i > 0)

w->Print(L", ");

w->Print(L"\"");

for (CharCount j = 0; j < length; j++)

w->PrintEscapedChar(litbuf[offset + j \* skip + i]);

w->Print(L"\"");

}

w->PrintEOL(L")");

PrintAnnotations(w);

}

#endif

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

// MatchCharNode

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

CharCount MatchCharNode::LiteralLength() const

{

return 1;

}

void MatchCharNode::AppendLiteral(CharCount& litbufNext, CharCount litbufLen, \_\_inout\_ecount(litbufLen) Char\* litbuf) const

{

Assert(!isEquivClass);

Assert(litbufNext + 1 <= litbufLen);

if (litbufNext + 1 <= litbufLen) // for prefast

litbuf[litbufNext++] = cs[0];

}

bool MatchCharNode::IsCharOrPositiveSet() const

{

return true;

}

CharCount MatchCharNode::TransferPass0(Compiler& compiler, const Char\* litbuf)

{

if ((compiler.program->flags & IgnoreCaseRegexFlag) != 0)

{

Char equivs[CaseInsensitive::EquivClassSize];

bool isNonTrivial = compiler.standardChars->ToEquivs(compiler.program->GetCaseMappingSource(), cs[0], equivs);

if (isNonTrivial)

{

isEquivClass = true;

for (int i = 0; i < CaseInsensitive::EquivClassSize; i++)

{

cs[i] = equivs[i];

}

}

}

return 0;

}

void MatchCharNode::TransferPass1(Compiler& compiler, const Char\* litbuf)

{

}

bool MatchCharNode::IsRefiningAssertion(Compiler& compiler)

{

return false;

}

void MatchCharNode::AnnotatePass0(Compiler& compiler)

{

// If c is a word char then all characters equivalent to c are word chars

isWord = compiler.standardChars->IsWord(cs[0]);

}

void MatchCharNode::AnnotatePass1(Compiler& compiler, bool parentNotInLoop, bool parentAtLeastOnce, bool parentNotSpeculative, bool parentNotNegated)

{

features = HasMatchChar;

thisConsumes.Exact(1);

firstSet = Anew(compiler.ctAllocator, UnicodeCharSet);

for (int i = 0; i < (isEquivClass ? CaseInsensitive::EquivClassSize : 1); i++)

firstSet->Set(compiler.ctAllocator, cs[i]);

isFirstExact = true;

isThisIrrefutable = false;

isThisWillNotProgress = true;

isThisWillNotRegress = true;

isNotInLoop = parentNotInLoop;

isAtLeastOnce = parentAtLeastOnce;

isNotSpeculative = parentNotSpeculative;

isNotNegated = parentNotNegated;

}

void MatchCharNode::AnnotatePass2(Compiler& compiler, CountDomain accumConsumes, bool accumPrevWillNotProgress, bool accumPrevWillNotRegress)

{

prevConsumes = accumConsumes;

isPrevWillNotProgress = accumPrevWillNotProgress;

isPrevWillNotRegress = accumPrevWillNotRegress;

}

void MatchCharNode::AnnotatePass3(Compiler& compiler, CountDomain accumConsumes, CharSet<Char>\* accumFollow, bool accumFollowIrrefutable, bool accumFollowEOL)

{

followConsumes = accumConsumes;

followSet = accumFollow;

isFollowIrrefutable = accumFollowIrrefutable;

isFollowEOL = accumFollowEOL;

}

void MatchCharNode::AnnotatePass4(Compiler& compiler)

{

isDeterministic = true;

}

bool MatchCharNode::SupportsPrefixSkipping(Compiler& compiler) const

{

return true;

}

Node\* MatchCharNode::HeadSyncronizingNode(Compiler& compiler)

{

return this;

}

CharCount MatchCharNode::MinSyncronizingLiteralLength(Compiler& compiler, int& numLiterals) const

{

numLiterals++;

return 1;

}

void MatchCharNode::CollectSyncronizingLiterals(Compiler& compiler, ScannersMixin& scanners) const

{

Assert(false);

}

void MatchCharNode::BestSyncronizingNode(Compiler& compiler, Node\*& bestNode)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

if (IsBetterSyncronizingNode(compiler, bestNode, this))

{

bestNode = this;

}

}

void MatchCharNode::AccumDefineGroups(Js::ScriptContext\* scriptContext, int& minGroup, int& maxGroup)

{

}

CompileAssert(CaseInsensitive::EquivClassSize == 4);

void MatchCharNode::Emit(Compiler& compiler, \_\_in\_ecount(4) Char \* cs, bool isEquivClass)

{

if (isEquivClass)

{

Char uniqueEquivs[CaseInsensitive::EquivClassSize];

CharCount uniqueEquivCount = FindUniqueEquivs(cs, uniqueEquivs);

switch (uniqueEquivCount)

{

case 2:

EMIT(compiler, MatchChar2Inst, uniqueEquivs[0], uniqueEquivs[1]);

break;

case 3:

EMIT(compiler, MatchChar3Inst, uniqueEquivs[0], uniqueEquivs[1], uniqueEquivs[2]);

break;

default:

EMIT(compiler, MatchChar4Inst, uniqueEquivs[0], uniqueEquivs[1], uniqueEquivs[2], uniqueEquivs[3]);

break;

}

}

else

EMIT(compiler, MatchCharInst, cs[0]);

}

CharCount MatchCharNode::FindUniqueEquivs(const Char equivs[CaseInsensitive::EquivClassSize], \_\_out\_ecount(4) Char uniqueEquivs[CaseInsensitive::EquivClassSize])

{

uniqueEquivs[0] = equivs[0];

CharCount uniqueCount = 1;

for (CharCount equivIndex = 1; equivIndex < CaseInsensitive::EquivClassSize; ++equivIndex)

{

bool alreadyHave = false;

for (CharCount uniqueIndex = 0; uniqueIndex < uniqueCount; ++uniqueIndex)

{

if (uniqueEquivs[uniqueIndex] == equivs[equivIndex])

{

alreadyHave = true;

break;

}

}

if (!alreadyHave)

{

uniqueEquivs[uniqueCount] = equivs[equivIndex];

uniqueCount += 1;

}

}

return uniqueCount;

}

void MatchCharNode::Emit(Compiler& compiler, CharCount& skipped)

{

if (skipped >= 1)

{

// Asking to skip entire char

skipped--;

return;

}

//

// Compilation scheme:

//

// MatchChar(2|3|4)?

//

skipped -= min(skipped, static\_cast<CharCount>(1));

Emit(compiler, cs, isEquivClass);

}

CharCount MatchCharNode::EmitScan(Compiler& compiler, bool isHeadSyncronizingNode)

{

//

// Compilation scheme:

//

// SyncTo(Char|Char2Set|Set)And(Consume|Continue|Backup)

//

if (isHeadSyncronizingNode)

{

// For a head literal there's no need to back up after finding the literal, so use a faster instruction

Assert(prevConsumes.IsExact(0)); // there should not be any consumes before this node

if (firstSet->IsSingleton())

EMIT(compiler, SyncToCharAndConsumeInst, firstSet->Singleton());

else

EMIT(compiler, SyncToSetAndConsumeInst<false>)->set.CloneFrom(compiler.rtAllocator, \*firstSet);

return 1;

}

else

{

// We're synchronizing on a non-head literal so we may need to back up. Or if we're syncing to the first literal

// inside a group for instance, then we won't need to back up but we cannot consume the literal. If we don't need to

// back up, we can use SyncToCharAndContinue instead.

if (prevConsumes.IsExact(0))

{

Char entries[CharSet<Char>::MaxCompact];

int count = firstSet->GetCompactEntries(2, entries);

if (count == 1)

EMIT(compiler, SyncToCharAndContinueInst, entries[0]);

else if (count == 2)

EMIT(compiler, SyncToChar2SetAndContinueInst, entries[0], entries[1]);

else

EMIT(compiler, SyncToSetAndContinueInst<false>)->set.CloneFrom(compiler.rtAllocator, \*firstSet);

}

else

{

if (firstSet->IsSingleton())

EMIT(compiler, SyncToCharAndBackupInst, firstSet->Singleton(), prevConsumes);

else

EMIT(compiler, SyncToSetAndBackupInst<false>, prevConsumes)->set.CloneFrom(compiler.rtAllocator, \*firstSet);

}

return 0;

}

}

bool MatchCharNode::IsOctoquad(Compiler& compiler, OctoquadIdentifier\* oi)

{

// We look for octoquad patterns before converting for case-insensitivity

Assert(!isEquivClass);

return oi->AppendChar(compiler.standardChars->ToCanonical(compiler.program->GetCaseMappingSource(), cs[0]));

}

bool MatchCharNode::IsCharTrieArm(Compiler& compiler, uint& accNumAlts) const

{

if (isEquivClass)

{

accNumAlts \*= CaseInsensitive::EquivClassSize;

if (accNumAlts > maxTrieArmExpansion)

return false;

}

return true;

}

bool MatchCharNode::BuildCharTrie(Compiler& compiler, CharTrie\* trie, Node\* cont, bool isAcceptFirst) const

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

for (int i = 0; i < (isEquivClass ? CaseInsensitive::EquivClassSize : 1); i++)

{

if (trie->IsAccepting())

{

// An earlier literal is a prefix of this literal

// If isAcceptFirst, can ignore suffix of already recognized literal.

// Otherwise, must fail.

return isAcceptFirst;

}

CharTrie\* tail = trie->Add(compiler.ctAllocator, cs[i]);

if (trie->Count() > maxTrieArmExpansion)

return false;

if (cont == 0)

{

if (tail->Count() > 0)

// This literal is a proper prefix of an earlier literal

return false;

tail->SetAccepting();

}

else

{

if (!cont->BuildCharTrie(compiler, tail, 0, isAcceptFirst))

return false;

}

}

return true;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void MatchCharNode::Print(DebugWriter\* w, const Char\* litbuf) const

{

w->Print(L"MatchChar(");

for (int i = 0; i < (isEquivClass ? CaseInsensitive::EquivClassSize : 1); i++)

{

if (i > 0)

w->Print(L", ");

w->PrintQuotedChar(cs[i]);

}

w->PrintEOL(L")");

PrintAnnotations(w);

}

#endif

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

// MatchSetNode

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

CharCount MatchSetNode::LiteralLength() const

{

return 0;

}

bool MatchSetNode::IsCharOrPositiveSet() const

{

return !isNegation;

}

CharCount MatchSetNode::TransferPass0(Compiler& compiler, const Char\* litbuf)

{

return 0;

}

void MatchSetNode::TransferPass1(Compiler& compiler, const Char\* litbuf)

{

if ((compiler.program->flags & IgnoreCaseRegexFlag) != 0 && this->needsEquivClass)

{

// If the set is negated, then at this point we could:

// (1) take each char in the set to its equivalence class and complement it

// (2) complement the set and take each char to its equivalence class

// Thankfully the spec demands (1), so we don't need to actually calculate any complement, just

// retain the isNegated flag

CharSet<Char> newSet;

// First include all the existing characters in the result set so that large ranges such as \x80-\uffff

// don't temporarily turn into a large number of small ranges corresponding to the various equivalent

// characters

newSet.UnionInPlace(compiler.rtAllocator, set);

set.ToEquivClass(compiler.rtAllocator, newSet);

set = newSet;

}

}

bool MatchSetNode::IsRefiningAssertion(Compiler& compiler)

{

return false;

}

void MatchSetNode::AnnotatePass0(Compiler& compiler)

{

if (isNegation || set.IsEmpty())

isWord = false;

else

isWord = set.IsSubsetOf(\*compiler.standardChars->GetWordSet());

}

void MatchSetNode::AnnotatePass1(Compiler& compiler, bool parentNotInLoop, bool parentAtLeastOnce, bool parentNotSpeculative, bool parentNotNegated)

{

features = HasMatchSet;

if (isNegation)

{

firstSet = Anew(compiler.ctAllocator, UnicodeCharSet);

set.ToComplement(compiler.ctAllocator, \*firstSet);

}

else

{

// CAUTION:

// Be careful not to use firstSet after deleting the node.

firstSet = &set;

}

isFirstExact = true;

// If firstSet is empty then pattern will always fail

thisConsumes.Exact(firstSet->IsEmpty() ? 0 : 1);

isThisIrrefutable = false;

isThisWillNotProgress = true;

isThisWillNotRegress = true;

isNotInLoop = parentNotInLoop;

isAtLeastOnce = parentAtLeastOnce;

isNotSpeculative = parentNotSpeculative;

isNotNegated = parentNotNegated;

}

void MatchSetNode::AnnotatePass2(Compiler& compiler, CountDomain accumConsumes, bool accumPrevWillNotProgress, bool accumPrevWillNotRegress)

{

prevConsumes = accumConsumes;

isPrevWillNotProgress = accumPrevWillNotProgress;

isPrevWillNotRegress = accumPrevWillNotRegress;

}

void MatchSetNode::AnnotatePass3(Compiler& compiler, CountDomain accumConsumes, CharSet<Char>\* accumFollow, bool accumFollowIrrefutable, bool accumFollowEOL)

{

followConsumes = accumConsumes;

followSet = accumFollow;

isFollowIrrefutable = accumFollowIrrefutable;

isFollowEOL = accumFollowEOL;

}

void MatchSetNode::AnnotatePass4(Compiler& compiler)

{

isDeterministic = true;

}

bool MatchSetNode::SupportsPrefixSkipping(Compiler& compiler) const

{

return true;

}

Node\* MatchSetNode::HeadSyncronizingNode(Compiler& compiler)

{

return this;

}

CharCount MatchSetNode::MinSyncronizingLiteralLength(Compiler& compiler, int& numLiterals) const

{

return 0;

}

void MatchSetNode::CollectSyncronizingLiterals(Compiler& compiler, ScannersMixin& scanners) const

{

Assert(false);

}

void MatchSetNode::BestSyncronizingNode(Compiler& compiler, Node\*& bestNode)

{

}

void MatchSetNode::AccumDefineGroups(Js::ScriptContext\* scriptContext, int& minGroup, int& maxGroup)

{

}

void MatchSetNode::Emit(Compiler& compiler, CharCount& skipped)

{

if (skipped >= 1)

{

// Asking to skip entire set

skipped--;

return;

}

//

// Compilation scheme:

//

// MatchSet

//

skipped -= min(skipped, static\_cast<CharCount>(1));

RuntimeCharSet<Char> \*runtimeSet;

if(isNegation)

runtimeSet = &EMIT(compiler, MatchSetInst<true>)->set;

else

runtimeSet = &EMIT(compiler, MatchSetInst<false>)->set;

runtimeSet->CloneFrom(compiler.rtAllocator, set);

}

CharCount MatchSetNode::EmitScan(Compiler& compiler, bool isHeadSyncronizingNode)

{

//

// Compilation scheme:

//

// SyncToSetAnd(Consume|Continue|Backup)

//

RuntimeCharSet<Char> \*runtimeSet;

CharCount consumedChars;

if (isHeadSyncronizingNode)

{

// For a head literal there's no need to back up after finding the literal, so use a faster instruction

Assert(prevConsumes.IsExact(0)); // there should not be any consumes before this node

if(isNegation)

runtimeSet = &EMIT(compiler, SyncToSetAndConsumeInst<true>)->set;

else

runtimeSet = &EMIT(compiler, SyncToSetAndConsumeInst<false>)->set;

consumedChars = 1;

}

else

{

// We're synchronizing on a non-head literal so we may need to back up. Or if we're syncing to the first literal

// inside a group for instance, then we won't need to back up but we cannot consume the literal. If we don't need to

// back up, we still cannot use SyncToSetAndContinue like in MatchCharNode::EmitScan, since SyncToSetAndContinue does not support negation

// sets.

if(prevConsumes.IsExact(0))

{

if(isNegation)

runtimeSet = &EMIT(compiler, SyncToSetAndContinueInst<true>)->set;

else

runtimeSet = &EMIT(compiler, SyncToSetAndContinueInst<false>)->set;

}

else if(isNegation)

runtimeSet = &EMIT(compiler, SyncToSetAndBackupInst<true>, prevConsumes)->set;

else

runtimeSet = &EMIT(compiler, SyncToSetAndBackupInst<false>, prevConsumes)->set;

consumedChars = 0;

}

runtimeSet->CloneFrom(compiler.rtAllocator, set);

return consumedChars;

}

bool MatchSetNode::IsOctoquad(Compiler& compiler, OctoquadIdentifier\* oi)

{

if (isNegation || set.IsEmpty() || !oi->BeginUnions())

return false;

Assert(CharSet<Char>::MaxCompact >= TrigramAlphabet::AlphaCount);

Char entries[CharSet<Char>::MaxCompact];

int count = set.GetCompactEntries(TrigramAlphabet::AlphaCount, entries);

if (count < 0)

// Too many unique characters

return false;

for (int i = 0; i < count; i++)

{

if (!oi->UnionChar(compiler.standardChars->ToCanonical(compiler.program->GetCaseMappingSource(), entries[i])))

// Too many unique characters

return false;

}

oi->EndUnions(); // this doesn't need to be called if we return false earlier since the OctoquadPattern won't be used

return true;

}

bool MatchSetNode::IsCharTrieArm(Compiler& compiler, uint& accNumAlts) const

{

if (isNegation || !set.IsCompact())

return false;

const uint count = set.Count();

if(count == 0)

return false; // empty set always fails and consumes nothing, and therefore is not a char-trie arm

accNumAlts \*= count;

if (accNumAlts > maxTrieArmExpansion)

return false;

return true;

}

bool MatchSetNode::BuildCharTrie(Compiler& compiler, CharTrie\* trie, Node\* cont, bool isAcceptFirst) const

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

Assert(!isNegation && set.IsCompact());

Char entries[CharSet<Char>::MaxCompact];

int count = set.GetCompactEntries(CharSet<Char>::MaxCompact, entries);

Assert(count > 0);

for (int i = 0; i < count; i++)

{

if (trie->IsAccepting())

{

// An earlier literal is a prefix of this literal

// If isAcceptFirst, can ignore suffix of already recognized literal.

// Otherwise, must fail.

return isAcceptFirst;

}

CharTrie\* tail = trie->Add(compiler.ctAllocator, entries[i]);

if (trie->Count() > maxTrieArmExpansion)

return false;

if (cont == 0)

{

if (tail->Count() > 0)

// This literal is a proper prefix of an earlier literal

return false;

tail->SetAccepting();

}

else

{

if (!cont->BuildCharTrie(compiler, tail, 0, isAcceptFirst))

return false;

}

}

return true;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void MatchSetNode::Print(DebugWriter\* w, const Char\* litbuf) const

{

w->Print(L"MatchSet(%s, ", isNegation ? L"negative" : L"positive");

set.Print(w);

w->PrintEOL(L")");

PrintAnnotations(w);

}

#endif

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

// ConcatNode

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

CharCount ConcatNode::LiteralLength() const

{

return 0;

}

bool ConcatNode::IsCharOrPositiveSet() const

{

return false;

}

CharCount ConcatNode::TransferPass0(Compiler& compiler, const Char\* litbuf)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

Assert(tail != 0);

CharCount n = 0;

#if DBG

ConcatNode\* prev = 0;

#endif

for (ConcatNode\* curr = this; curr != 0; curr = curr->tail)

{

Assert(curr->head->tag != Concat);

Assert(prev == 0 || !(prev->head->LiteralLength() > 0 && curr->head->LiteralLength() > 0));

n += curr->head->TransferPass0(compiler, litbuf);

#if DBG

prev = curr;

#endif

}

return n;

}

void ConcatNode::TransferPass1(Compiler& compiler, const Char\* litbuf)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

for (ConcatNode \*curr = this; curr != 0; curr = curr->tail)

curr->head->TransferPass1(compiler, litbuf);

}

bool ConcatNode::IsRefiningAssertion(Compiler& compiler)

{

return false;

}

void ConcatNode::AnnotatePass0(Compiler& compiler)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

Node\* prev = 0;

for (ConcatNode\* curr = this; curr != 0; curr = curr->tail)

{

curr->head->AnnotatePass0(compiler);

if (prev != 0)

{

if (curr->head->tag == WordBoundary && prev->isWord)

{

WordBoundaryNode\* wb = (WordBoundaryNode\*)curr->head;

wb->mustIncludeLeaving = true;

}

else if (prev->tag == WordBoundary && curr->head->isWord)

{

WordBoundaryNode\* wb = (WordBoundaryNode\*)prev;

wb->mustIncludeEntering = true;

}

}

prev = curr->head;

}

}

void ConcatNode::AnnotatePass1(Compiler& compiler, bool parentNotInLoop, bool parentAtLeastOnce, bool parentNotSpeculative, bool parentNotNegated)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

features = HasConcat;

isNotInLoop = parentNotInLoop;

isAtLeastOnce = parentAtLeastOnce;

isNotSpeculative = parentNotSpeculative;

isNotNegated = parentNotNegated;

// Pass 1: Count items

int n = 0;

for (ConcatNode \*curr = this; curr != 0; curr = curr->tail)

n++;

// Pass 2: Annotate each item, accumulate features, consumes, find longest prefix of possibly-empty-accepting items,

// check if all items are irrefutable

int emptyPrefix = 0;

thisConsumes.Exact(0);

isThisIrrefutable = true;

isThisWillNotProgress = true;

isThisWillNotRegress = true;

int item = 0;

for (ConcatNode\* curr = this; curr != 0; curr = curr->tail, item++)

{

curr->head->AnnotatePass1(compiler, parentNotInLoop, parentAtLeastOnce, parentNotSpeculative, isNotNegated);

features |= curr->head->features;

thisConsumes.Add(curr->head->thisConsumes);

if (!curr->head->isThisIrrefutable)

isThisIrrefutable = false;

if (!curr->head->isThisWillNotProgress)

isThisWillNotProgress = false;

if (!curr->head->isThisWillNotRegress)

isThisWillNotRegress = false;

if (emptyPrefix == item && curr->head->thisConsumes.CouldMatchEmpty())

emptyPrefix++;

}

if (emptyPrefix == 0)

{

firstSet = head->firstSet;

isFirstExact = head->isFirstExact;

}

else

{

// Pass 3: Overall first set is union of first's of emptyPrefx

firstSet = Anew(compiler.ctAllocator, UnicodeCharSet);

isFirstExact = true;

item = 0;

for (ConcatNode\* curr = this; item <= min(emptyPrefix, n - 1); curr = curr->tail, item++)

{

AnalysisAssert(curr);

firstSet->UnionInPlace(compiler.ctAllocator, \*curr->head->firstSet);

if (!curr->head->isFirstExact)

isFirstExact = false;

}

}

}

void ConcatNode::AnnotatePass2(Compiler& compiler, CountDomain accumConsumes, bool accumPrevWillNotProgress, bool accumPrevWillNotRegress)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

prevConsumes = accumConsumes;

isPrevWillNotProgress = accumPrevWillNotProgress;

isPrevWillNotRegress = accumPrevWillNotRegress;

for (ConcatNode\* curr = this; curr != 0; curr = curr->tail)

{

curr->head->AnnotatePass2(compiler, accumConsumes, accumPrevWillNotProgress, accumPrevWillNotRegress);

accumConsumes.Add(curr->head->thisConsumes);

if (!curr->head->isThisWillNotProgress)

accumPrevWillNotProgress = false;

if (!curr->head->isThisWillNotRegress)

accumPrevWillNotRegress = false;

}

}

void ConcatNode::AnnotatePass3(Compiler& compiler, CountDomain accumConsumes, CharSet<Char>\* accumFollow, bool accumFollowIrrefutable, bool accumFollowEOL)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

followConsumes = accumConsumes;

followSet = accumFollow;

isFollowIrrefutable = accumFollowIrrefutable;

isFollowEOL = accumFollowEOL;

// Pass 1: Count items

int n = 0;

for (ConcatNode\* curr = this; curr != 0; curr = curr->tail)

n++;

// Pass 2: Collect items so we can enumerate backwards

Node\*\* nodes = AnewArray(compiler.ctAllocator, Node \*, n);

int item = 0;

for (ConcatNode\* curr = this; curr != 0; curr = curr->tail, item++)

nodes[item] = curr->head;

// Pass 3: Work backwards propagating follow set, irrefutability and FollowEndLineOrPattern, and adding consumes

CharSet<Char>\* innerFollow = accumFollow;

for (item = n - 1; item >= 0; item--)

{

nodes[item]->AnnotatePass3(compiler, accumConsumes, innerFollow, accumFollowIrrefutable, accumFollowEOL);

if (!nodes[item]->isThisIrrefutable)

accumFollowIrrefutable = false;

if (!nodes[item]->IsEmptyOnly() && (compiler.program->flags & MultilineRegexFlag) == 0)

accumFollowEOL = nodes[item]->tag == EOL;

// ConcatNode has hardfail BOI test if any child has hardfail BOI

hasInitialHardFailBOI = hasInitialHardFailBOI || nodes[item]->hasInitialHardFailBOI;

if (item > 0)

{

CharSet<Char>\* nextInnerFollow = Anew(compiler.ctAllocator, UnicodeCharSet);

if (nodes[item]->thisConsumes.CouldMatchEmpty() && !nodes[item]->IsRefiningAssertion(compiler))

{

// Later follows can shine through this item to the previous item

nextInnerFollow->UnionInPlace(compiler.ctAllocator, \*innerFollow);

}

nextInnerFollow->UnionInPlace(compiler.ctAllocator, \*nodes[item]->firstSet);

innerFollow = nextInnerFollow;

accumConsumes.Add(nodes[item]->thisConsumes);

}

}

}

void ConcatNode::AnnotatePass4(Compiler& compiler)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

isDeterministic = true;

for (ConcatNode\* curr = this; curr != 0; curr = curr->tail)

{

curr->head->AnnotatePass4(compiler);

if (!curr->head->isDeterministic)

isDeterministic = false;

}

}

bool ConcatNode::SupportsPrefixSkipping(Compiler& compiler) const

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

int prefix = 0;

for (const ConcatNode\* curr = this; curr != 0; curr = curr->tail)

{

if (curr->head->SupportsPrefixSkipping(compiler))

prefix++;

else

break;

}

return prefix > 0;

}

Node\* ConcatNode::HeadSyncronizingNode(Compiler& compiler)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

return head->HeadSyncronizingNode(compiler);

}

void ConcatNode::AccumDefineGroups(Js::ScriptContext\* scriptContext, int& minGroup, int& maxGroup)

{

PROBE\_STACK(scriptContext, Js::Constants::MinStackRegex);

for (ConcatNode \*curr = this; curr != 0; curr = curr->tail)

curr->head->AccumDefineGroups(scriptContext, minGroup, maxGroup);

}

CharCount ConcatNode::MinSyncronizingLiteralLength(Compiler& compiler, int& numLiterals) const

{

return 0;

}

void ConcatNode::CollectSyncronizingLiterals(Compiler& compiler, ScannersMixin& scanners) const

{

Assert(false);

}

void ConcatNode::BestSyncronizingNode(Compiler& compiler, Node\*& bestNode)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

for (ConcatNode\* curr = this; curr != 0; curr = curr->tail)

curr->head->BestSyncronizingNode(compiler, bestNode);

}

void ConcatNode::Emit(Compiler& compiler, CharCount& skipped)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

//

// Compilation scheme:

//

// <item 1>

// ...

// <item n>

//

// :-)

for (ConcatNode \*curr = this; curr != 0; curr = curr->tail)

curr->head->Emit(compiler, skipped);

}

CharCount ConcatNode::EmitScan(Compiler& compiler, bool isHeadSyncronizingNode)

{

Assert(false);

return 0;

}

bool ConcatNode::IsOctoquad(Compiler& compiler, OctoquadIdentifier\* oi)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

for (ConcatNode\* curr = this; curr != 0; curr = curr->tail)

{

if (!curr->head->IsOctoquad(compiler, oi))

return false;

}

return true;

}

bool ConcatNode::IsCharTrieArm(Compiler& compiler, uint& accNumAlts) const

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

for (const ConcatNode\* curr = this; curr != 0; curr = curr->tail)

{

if (!curr->head->IsCharTrieArm(compiler, accNumAlts))

return false;

}

return true;

}

bool ConcatNode::BuildCharTrie(Compiler& compiler, CharTrie\* trie, Node\* cont, bool isAcceptFirst) const

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

if (cont != 0)

// We don't want to manage a stack of continuations

return false;

// NOTE: This is the only place we use an internal node of a concat sequence as a sub-concat sequence

return head->BuildCharTrie(compiler, trie, tail, isAcceptFirst);

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void ConcatNode::Print(DebugWriter\* w, const Char\* litbuf) const

{

w->PrintEOL(L"Concat()");

PrintAnnotations(w);

w->PrintEOL(L"{");

w->Indent();

for (const ConcatNode \*curr = this; curr != 0; curr = curr->tail)

curr->head->Print(w, litbuf);

w->Unindent();

w->PrintEOL(L"}");

}

#endif

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

// AltNode

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

CharCount AltNode::LiteralLength() const

{

return 0;

}

bool AltNode::IsCharOrPositiveSet() const

{

return false;

}

CharCount AltNode::TransferPass0(Compiler& compiler, const Char\* litbuf)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

Assert(tail != 0);

CharCount n = 0;

#if DBG

AltNode\* prev = 0;

#endif

for (AltNode\* curr = this; curr != 0; curr = curr->tail)

{

Assert(curr->head->tag != Alt);

Assert(prev == 0 || !(prev->head->IsCharOrPositiveSet() && curr->head->IsCharOrPositiveSet()));

n += curr->head->TransferPass0(compiler, litbuf);

#if DBG

prev = curr;

#endif

}

return n;

}

void AltNode::TransferPass1(Compiler& compiler, const Char\* litbuf)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

for (AltNode \*curr = this; curr != 0; curr = curr->tail)

curr->head->TransferPass1(compiler, litbuf);

}

bool AltNode::IsRefiningAssertion(Compiler& compiler)

{

return false;

}

void AltNode::AnnotatePass0(Compiler& compiler)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

isWord = true;

for (AltNode\* curr = this; curr != 0; curr = curr->tail)

{

curr->head->AnnotatePass0(compiler);

if (!curr->head->isWord)

isWord = false;

}

}

void AltNode::AnnotatePass1(Compiler& compiler, bool parentNotInLoop, bool parentAtLeastOnce, bool parentNotSpeculative, bool parentNotNegated)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

features = HasAlt;

isNotInLoop = parentNotInLoop;

isAtLeastOnce = parentAtLeastOnce;

isNotSpeculative = parentNotSpeculative;

isNotNegated = parentNotNegated;

// Overall alternative:

// - is irrefutable if at least one item is irrefutable

// - will not progress(regress) if each item will not progress(regress) and has strictly decreasing(increasing) consumes

firstSet = Anew(compiler.ctAllocator, UnicodeCharSet);

isThisIrrefutable = false;

isThisWillNotProgress = true;

isThisWillNotRegress = true;

isFirstExact = true;

CountDomain prevConsumes;

int item = 0;

for (AltNode \*curr = this; curr != 0; curr = curr->tail, item++)

{

curr->head->AnnotatePass1(compiler, parentNotInLoop, false, parentNotSpeculative, isNotNegated);

features |= curr->head->features;

if (!curr->head->isThisWillNotProgress)

isThisWillNotProgress = false;

if (!curr->head->isThisWillNotRegress)

isThisWillNotRegress = false;

if (item == 0)

prevConsumes = thisConsumes = curr->head->thisConsumes;

else

{

thisConsumes.Lub(curr->head->thisConsumes);

if (!curr->head->thisConsumes.IsLessThan(prevConsumes))

isThisWillNotProgress = false;

if (!curr->head->thisConsumes.IsGreaterThan(prevConsumes))

isThisWillNotRegress = false;

prevConsumes = curr->head->thisConsumes;

}

firstSet->UnionInPlace(compiler.ctAllocator, \*curr->head->firstSet);

if (!curr->head->isFirstExact || curr->head->isThisIrrefutable)

// If any item is irrefutable then later items may never be taken, so first set cannot be exact

isFirstExact = false;

if (curr->head->isThisIrrefutable)

isThisIrrefutable = true;

}

}

void AltNode::AnnotatePass2(Compiler& compiler, CountDomain accumConsumes, bool accumPrevWillNotProgress, bool accumPrevWillNotRegress)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

prevConsumes = accumConsumes;

isPrevWillNotProgress = accumPrevWillNotProgress;

isPrevWillNotRegress = accumPrevWillNotRegress;

for (AltNode\* curr = this; curr != 0; curr = curr->tail)

curr->head->AnnotatePass2(compiler, accumConsumes, accumPrevWillNotProgress, accumPrevWillNotRegress);

}

void AltNode::AnnotatePass3(Compiler& compiler, CountDomain accumConsumes, CharSet<Char>\* accumFollow, bool accumFollowIrrefutable, bool accumFollowEOL)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

followConsumes = accumConsumes;

followSet = accumFollow;

isFollowIrrefutable = accumFollowIrrefutable;

isFollowEOL = accumFollowEOL;

for (AltNode\* curr = this; curr != 0; curr = curr->tail)

{

curr->head->AnnotatePass3(compiler, accumConsumes, accumFollow, accumFollowIrrefutable, accumFollowEOL);

// AltNode has hardfail BOI test if all child Nodes have hardfail BOI tests

hasInitialHardFailBOI = curr->head->hasInitialHardFailBOI && (hasInitialHardFailBOI || (curr == this));

}

}

void AltNode::AnnotatePass4(Compiler& compiler)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

//

// Simplification rule

//

// If the follow is irrefutable then we can ignore all items after an irrefutable item, since

// we'll never be able to backtrack into them.

// E.g.: (a\*|b\*)c\* === a\*c\*

//

bool simplified = false;

if (isFollowIrrefutable && isThisIrrefutable)

{

for (AltNode\* curr = this; curr != 0; curr = curr->tail)

{

if (curr->head->isThisIrrefutable && curr->tail != 0)

{

curr->tail = 0;

simplified = true;

break;

}

}

}

if (simplified)

{

Assert(!isFirstExact);

// Recalculate firstSet. Since it can only get smaller, and alternative could not have had an exact

// first set, this recalculation does not make any decisions already made based on the current firstSet

// unsound.

// NOTE: Is it worth recalculating the WillNotProgess/WillNotRegress bools?

firstSet = Anew(compiler.ctAllocator, UnicodeCharSet);

for (AltNode\* curr = this; curr != 0; curr = curr->tail)

firstSet->UnionInPlace(compiler.ctAllocator, \*curr->head->firstSet);

}

//

// Annotate items

//

isDeterministic = true;

for (AltNode\* curr = this; curr != 0; curr = curr->tail)

{

curr->head->AnnotatePass4(compiler);

if (!curr->head->isDeterministic)

isDeterministic = false;

}

//

// Compilation scheme: Switch/Chain/Set, not isOptional

//

// If no item can match empty and all items' FIRST sets are pairwise disjoint then we can

// commit to an item using a 1 char lookahead. We can fall-through to the last

// item without guarding it since it will fail if the next character cannot match.

// E.g.: (abc|def)

//

{

// Pass 1: Items cannot match empty, accumulate counts

bool fires = true;

bool allCompact = true;

bool allSimpleOneChar = true;

int numItems = 0;

uint totalChars = 0;

for (AltNode\* curr = this; curr != 0; curr = curr->tail)

{

if (curr->head->thisConsumes.CouldMatchEmpty())

{

fires = false;

break;

}

numItems++;

if (!curr->head->firstSet->IsCompact())

allCompact = false;

if (!curr->head->IsSimpleOneChar())

allSimpleOneChar = false;

totalChars += curr->head->firstSet->Count();

}

if (fires)

{

// To go from two to one items requires the first item

// to be irrefutable, in which case it could match empty and this rule won't fire.

Assert(numItems > 1);

// Step 2: Check FIRST sets are disjoint

if (totalChars == firstSet->Count())

{

// \*\*COMMIT\*\*

if (allSimpleOneChar)

// This will probably never fire since the parser has already converted alts-of-chars/sets

// to sets. We include it for symmetry with below.

scheme = Set;

else if (allCompact && totalChars <= Switch20Inst::MaxCases)

{

// Can use a switch instruction to jump to item

scheme = Switch;

switchSize = totalChars;

}

else

// Must use a chain of jump instructions to jump to item

scheme = Chain;

isOptional = false;

return;

}

}

}

//

// Compilation scheme: None/Switch/Chain/Set, isOptional

//

// Condition (1):

// If some items are empty-only, the rest (if any) cannot match empty, follow cannot match empty, and

// all items' FIRST sets are pairwise disjoint and disjoint from the FOLLOW set, then we can commit to

// either a non-empty item or to the empty item using a 1 char lookahead. In this case we just emit each

// non-empty item with a guard, and fall-through to follow if no guard fires.

// E.g.: (abc||def)h

//

// Condition (2):

// If some items are empty-only, the rest (if any) cannot match empty, follow is irrefutable, and all

// items' FIRST sets are pairwise disjoint, then we can commit to either a non-empty item or to the empty

// item using a 1 char lookahead, provided each non-empty item obeys the condition:

// \*\* the item can't fail if given an arbitrary input starting with a character in its FIRST set \*\*

// Currently, we can prove that only for IsSimpleOneChar items, though more analysis could widen the class.

// Again, we emit each non-empty item with a guard, and fall-through to follow if no guard fires.

// E.g.: ([abc]|)a\*

//

// Condition (3):

// If all items are empty-only, we can commit to a single empty-only item

{

// Pass 1

bool fires = false;

bool allSimpleOneChar = true;

bool allCompact = true;

int numNonEmpty = 0;

uint totalChars = 0;

for (AltNode\* curr = this; curr != 0; curr = curr->tail)

{

if (curr->head->IsEmptyOnly())

fires = true;

else if (curr->head->thisConsumes.CouldMatchEmpty())

{

fires = false;

break;

}

else

{

numNonEmpty++;

if (!curr->head->IsSimpleOneChar())

allSimpleOneChar = false;

if (!curr->head->firstSet->IsCompact())

allCompact = false;

totalChars += curr->head->firstSet->Count();

}

}

if (fires)

{

// The firing condition is not strong enough yet.

fires = false;

// Check conditions (2) and (3) first because they're faster, then check condition (1).

if (numNonEmpty == 0 || isFollowIrrefutable && allSimpleOneChar && totalChars == firstSet->Count())

{

fires = true;

}

else if (!followConsumes.CouldMatchEmpty())

{

// Check whether all FIRST sets are pairwise disjoint

// and disjoint from the FOLLOW set.

CharSet<Char> unionSet;

unionSet.UnionInPlace(compiler.ctAllocator, \*firstSet);

unionSet.UnionInPlace(compiler.ctAllocator, \*followSet);

if (totalChars + followSet->Count() == unionSet.Count())

fires = true;

}

if (fires)

{

// \*\*COMMIT\*\*

if (numNonEmpty == 0)

scheme = None;

else if (allSimpleOneChar)

scheme = Set;

else if (numNonEmpty > 1 && allCompact && totalChars <= Switch20Inst::MaxCases)

{

switchSize = totalChars;

scheme = Switch;

}

else

scheme = Chain;

isOptional = true;

return;

}

}

}

//

// Compilation scheme: Trie

//

// If alt is equivalent to the form:

// (literal1|...|literaln)

// (we expand items with embedded character classes such as a[bc]d to (abd|acd)) and either:

// - follow is irrefutable and no later literal is a proper prefix of an earlier literal

// (and we may ignore later literals which have an earlier literal as proper prefix)

// E.g.: (ab|ac|abd)a\* === (ab|ac)a\*

// or:

// - follow is not irrefutable and no literal is a proper prefix of any other literal

// and the branching factor of the resulting trie is smallish

// E.g.: (abc|abd|abe)f

// then we can use a character trie to match the appropriate item.

//

{

// Pass 1: Items must be structurally appropriate and not result in too many alternatives after expansion

bool fires = true;

for (AltNode\* curr = this; curr != 0; curr = curr->tail)

{

uint numAlts = 1;

if (!curr->head->IsCharTrieArm(compiler, numAlts))

{

fires = false;

break;

}

if (numAlts > maxTrieArmExpansion)

{

fires = false;

break;

}

}

if (fires)

{

// Pass 2: Attempt to construct the trie, checking for prefixes.

CharTrie trie;

for (AltNode\* curr = this; curr != 0; curr = curr->tail)

{

if (!curr->head->BuildCharTrie(compiler, &trie, 0, isFollowIrrefutable))

{

fires = false;

break;

}

}

if (fires)

{

// \*\*COMMIT\*\*

// If follow is irrefutable and first item is empty, the trie would be of depth zero.

// However, in this case, the first simplification rule would have replaced the alt with a

// single empty item, and the 'None' compilation scheme would have been selected above.

//

// Similarly, if all alternations are empty and follow is refutable, the trie would be

// of depth zero, and the 'None' compilation scheme would have been selected above.

Assert(!trie.IsDepthZero());

if (trie.IsDepthOne())

{

// This case will fire if follow is irrefutable and all non length one items have an

// earlier one-character item as prefix. In this case we don't need the trie: the

// firstSet has all the information.

isOptional = false;

scheme = Set;

}

else

{

// Root of trie will live in compile-time allocator, but body will be in run-time allocator

runtimeTrie = Anew(compiler.ctAllocator, RuntimeCharTrie);

runtimeTrie->CloneFrom(compiler.rtAllocator, trie);

scheme = Trie;

}

return;

}

}

}

//

// Compilation scheme: Try

//

scheme = Try;

isDeterministic = false; // NON-DETERMINISTIC

}

bool AltNode::SupportsPrefixSkipping(Compiler& compiler) const

{

return false;

}

Node\* AltNode::HeadSyncronizingNode(Compiler& compiler)

{

return 0;

}

CharCount AltNode::MinSyncronizingLiteralLength(Compiler& compiler, int& numLiterals) const

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

// Here, we ignore nodes with length 1, which are Char nodes. The way the Alt node synchronization

// is currently implemented, it expects all nodes to be Literal nodes. It requires quite a bit of

// refactoring to have Alt nodes support Char nodes for synchronization, so Char nodes are ignored

// for now.

int localNumLiterals = numLiterals;

CharCount minLen = head->MinSyncronizingLiteralLength(compiler, localNumLiterals);

if (minLen <= 1)

return 0;

for (AltNode\* curr = tail; curr != 0; curr = curr->tail)

{

CharCount thisLen = curr->head->MinSyncronizingLiteralLength(compiler, localNumLiterals);

if (thisLen <= 1)

return 0;

minLen = min(minLen, thisLen);

}

numLiterals = localNumLiterals;

if (minLen <= 1)

{

return 0;

}

return minLen;

}

void AltNode::CollectSyncronizingLiterals(Compiler& compiler, ScannersMixin& scanners) const

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

for (const AltNode\* curr = this; curr != 0; curr = curr->tail)

curr->head->CollectSyncronizingLiterals(compiler, scanners);

}

void AltNode::BestSyncronizingNode(Compiler& compiler, Node\*& bestNode)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

if (IsBetterSyncronizingNode(compiler, bestNode, this))

bestNode = this;

}

void AltNode::AccumDefineGroups(Js::ScriptContext\* scriptContext, int& minGroup, int& maxGroup)

{

PROBE\_STACK(scriptContext, Js::Constants::MinStackRegex);

for (AltNode \*curr = this; curr != 0; curr = curr->tail)

curr->head->AccumDefineGroups(scriptContext, minGroup, maxGroup);

}

void AltNode::Emit(Compiler& compiler, CharCount& skipped)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

Assert(skipped == 0);

switch (scheme)

{

case Try:

{

//

// Compilation scheme:

//

// Try((If|Match)(Char|Set))? L2

// <item 1>

// Jump Lexit

// L2: Try((If|Match)(Char|Set))? L3

// <item 2>

// Jump Lexit

// L3: <item 3>

// Lexit:

//

Assert(!isOptional);

int numItems = 0;

for (AltNode\* curr = this; curr != 0; curr = curr->tail)

numItems++;

Assert(numItems >= 1);

// Each item other than last needs to jump to exit on success

Label\* jumpFixups = AnewArray(compiler.ctAllocator, Label, (numItems - 1));

Label lastTryFixup = 0;

int item = 0;

for (AltNode\* curr = this; curr != 0; curr = curr->tail, item++)

{

if (item > 0)

// Fixup previous Try

compiler.DoFixup(lastTryFixup, compiler.CurrentLabel());

CharCount itemSkipped = 0;

if (item < numItems-1)

{

// HEURISTIC: if the first set of the alternative is exact or small, and the

// alternative does not match empty, then it's probably worth using

// a Try(If|Match)(Char|Set)

if (curr->head->firstSet != 0 &&

!curr->head->thisConsumes.CouldMatchEmpty() &&

(curr->head->isFirstExact || curr->head->firstSet->Count() <= maxCharsForConditionalTry))

{

if (curr->head->SupportsPrefixSkipping(compiler))

{

if (curr->head->firstSet->IsSingleton())

lastTryFixup = compiler.GetFixup(&EMIT(compiler, TryMatchCharInst, curr->head->firstSet->Singleton())->failLabel);

else

{

TryMatchSetInst\* const i = EMIT(compiler, TryMatchSetInst);

i->set.CloneFrom(compiler.rtAllocator, \*curr->head->firstSet);

lastTryFixup = compiler.GetFixup(&i->failLabel);

}

itemSkipped = 1;

}

else

{

if (curr->head->firstSet->IsSingleton())

lastTryFixup = compiler.GetFixup(&EMIT(compiler, TryIfCharInst, curr->head->firstSet->Singleton())->failLabel);

else

{

TryIfSetInst\* const i = EMIT(compiler, TryIfSetInst);

i->set.CloneFrom(compiler.rtAllocator, \*curr->head->firstSet);

lastTryFixup = compiler.GetFixup(&i->failLabel);

}

}

}

else

lastTryFixup = compiler.GetFixup(&EMIT(compiler, TryInst)->failLabel);

}

curr->head->Emit(compiler, itemSkipped);

if (item < numItems-1)

jumpFixups[item] = compiler.GetFixup(&EMIT(compiler, JumpInst)->targetLabel);

}

// Fixup jumps

for (item = 0; item < numItems-1; item++)

compiler.DoFixup(jumpFixups[item], compiler.CurrentLabel());

break;

}

case None:

{

Assert(isOptional);

// Nothing to emit

break;

}

case Trie:

{

//

// Compilation scheme:

//

// MatchTrie <trie>

//

EMIT(compiler, MatchTrieInst)->trie = \*runtimeTrie;

break;

}

case Switch:

{

//

// Compilation scheme:

//

// Switch(AndConsume)?(10|20)(<dispatch to each arm>)

// Fail (if non-optional)

// Jump Lexit (if optional)

// L1: <item1>

// Jump Lexit

// L2: <item2>

// Jump Lexit

// L3: <item3>

// Lexit:

//

Assert(switchSize <= Switch20Inst::MaxCases);

int numItems = 0;

bool allCanSkip = true;

for (AltNode\* curr = this; curr != 0; curr = curr->tail)

{

if (curr->head->thisConsumes.CouldMatchEmpty())

{

Assert(isOptional);

}

else

{

numItems++;

if (!curr->head->SupportsPrefixSkipping(compiler))

allCanSkip = false;

}

}

Assert(numItems > 1);

// Each item other than last needs to jump to exit on success

Label\* jumpFixups = AnewArray(compiler.ctAllocator, Label, (numItems - 1));

// We must remember where each item begins to fixup switch

Label\* caseLabels = AnewArray(compiler.ctAllocator, Label, numItems);

// We must fixup the switch arms

Label switchLabel = compiler.CurrentLabel();

Assert(switchSize <= Switch20Inst::MaxCases);

if (allCanSkip)

{

if (switchSize > Switch10Inst::MaxCases)

EMIT(compiler, SwitchAndConsume20Inst);

else

EMIT(compiler, SwitchAndConsume10Inst);

}

else

{

if (switchSize > Switch10Inst::MaxCases)

EMIT(compiler, Switch20Inst);

else

EMIT(compiler, Switch10Inst);

}

Label defaultJumpFixup = 0;

if (isOptional)

// Must fixup default jump to exit

defaultJumpFixup = compiler.GetFixup(&EMIT(compiler, JumpInst)->targetLabel);

else

compiler.Emit<FailInst>();

// Emit each item

int item = 0;

for (AltNode\* curr = this; curr != 0; curr = curr->tail)

{

if (!curr->head->thisConsumes.CouldMatchEmpty())

{

if (allCanSkip)

skipped = 1;

caseLabels[item] = compiler.CurrentLabel();

curr->head->Emit(compiler, skipped);

if (item < numItems - 1)

jumpFixups[item] = compiler.GetFixup(&EMIT(compiler, JumpInst)->targetLabel);

item++;

}

}

// Fixup exit labels

if (isOptional)

compiler.DoFixup(defaultJumpFixup, compiler.CurrentLabel());

for (item = 0; item < numItems - 1; item++)

compiler.DoFixup(jumpFixups[item], compiler.CurrentLabel());

// Fixup the switch entries

item = 0;

for (AltNode\* curr = this; curr != 0; curr = curr->tail)

{

if (!curr->head->thisConsumes.CouldMatchEmpty())

{

Char entries[CharSet<Char>::MaxCompact];

int count = curr->head->firstSet->GetCompactEntries(CharSet<Char>::MaxCompact, entries);

Assert(count > 0);

for (int i = 0; i < count; i++)

{

if (allCanSkip)

{

if (switchSize > Switch10Inst::MaxCases)

compiler.L2I(SwitchAndConsume20, switchLabel)->AddCase(entries[i], caseLabels[item]);

else

compiler.L2I(SwitchAndConsume10, switchLabel)->AddCase(entries[i], caseLabels[item]);

}

else

{

if (switchSize > Switch10Inst::MaxCases)

compiler.L2I(Switch20, switchLabel)->AddCase(entries[i], caseLabels[item]);

else

compiler.L2I(Switch10, switchLabel)->AddCase(entries[i], caseLabels[item]);

}

}

item++;

}

}

break;

}

case Chain:

{

//

// Compilation scheme:

//

// JumpIfNot(Char|Set) L2

// <item1>

// Jump Lexit

// L2: JumpIfNot(Char|Set) L3

// <item2>

// Jump Lexit

// L3: <item3> (if non-optional)

// L3: JumpIfNot(Char|Set) Lexit (if optional)

// <item3> (if optional)

// Lexit:

//

int numItems = 0;

for (AltNode\* curr = this; curr != 0; curr = curr->tail)

{

if (curr->head->thisConsumes.CouldMatchEmpty())

{

Assert(isOptional);

}

else

numItems++;

}

Assert(numItems > 0);

// Each item other than last needs to jump to exit on success

Label\* jumpFixups = AnewArray(compiler.ctAllocator, Label, (numItems - 1));

Label lastJumpFixup = 0;

int item = 0;

for (AltNode\* curr = this; curr != 0; curr = curr->tail)

{

if (!curr->head->thisConsumes.CouldMatchEmpty())

{

if (item > 0)

// Fixup previous Jump

compiler.DoFixup(lastJumpFixup, compiler.CurrentLabel());

CharCount itemSkipped = 0;

if (item < numItems-1 || isOptional)

{

if (curr->head->firstSet->IsSingleton())

{

if (curr->head->SupportsPrefixSkipping(compiler))

{

lastJumpFixup = compiler.GetFixup(&EMIT(compiler, MatchCharOrJumpInst, curr->head->firstSet->Singleton())->targetLabel);

itemSkipped = 1;

}

else

lastJumpFixup = compiler.GetFixup(&EMIT(compiler, JumpIfNotCharInst, curr->head->firstSet->Singleton())->targetLabel);

}

else

{

if (curr->head->SupportsPrefixSkipping(compiler))

{

MatchSetOrJumpInst\* const i = EMIT(compiler, MatchSetOrJumpInst);

i->set.CloneFrom(compiler.rtAllocator, \*curr->head->firstSet);

lastJumpFixup = compiler.GetFixup(&i->targetLabel);

itemSkipped = 1;

}

else

{

JumpIfNotSetInst\* const i = EMIT(compiler, JumpIfNotSetInst);

i->set.CloneFrom(compiler.rtAllocator, \*curr->head->firstSet);

lastJumpFixup = compiler.GetFixup(&i->targetLabel);

}

}

}

curr->head->Emit(compiler, itemSkipped);

if (item < numItems-1)

jumpFixups[item] = compiler.GetFixup(&EMIT(compiler, JumpInst)->targetLabel);

item++;

}

}

// Fixup jumps to exit

for (item = 0; item < numItems-1; item++)

compiler.DoFixup(jumpFixups[item], compiler.CurrentLabel());

if (isOptional)

// Fixup last Jump to exit

compiler.DoFixup(lastJumpFixup, compiler.CurrentLabel());

break;

}

case Set:

{

//

// Compilation scheme:

//

// Match(Char|Set) (non optional)

// OptMatch(Char|Set) (optional)

//

if (isOptional)

{

if (firstSet->IsSingleton())

EMIT(compiler, OptMatchCharInst, firstSet->Singleton());

else

EMIT(compiler, OptMatchSetInst)->set.CloneFrom(compiler.rtAllocator, \*firstSet);

}

else

{

if (firstSet->IsSingleton())

EMIT(compiler, MatchCharInst, firstSet->Singleton());

else

EMIT(compiler, MatchSetInst<false>)->set.CloneFrom(compiler.rtAllocator, \*firstSet);

}

break;

}

}

}

CharCount AltNode::EmitScan(Compiler& compiler, bool isHeadSyncronizingNode)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

Assert(!isHeadSyncronizingNode);

//

// Compilation scheme:

//

// SyncToLiteralsAndBackup

//

SyncToLiteralsAndBackupInst\* i =

EMIT(

compiler,

SyncToLiteralsAndBackupInst,

compiler.GetScriptContext()->GetRecycler(),

compiler.GetProgram(),

prevConsumes);

CollectSyncronizingLiterals(compiler, \*i);

return 0;

}

bool AltNode::IsOctoquad(Compiler& compiler, OctoquadIdentifier\* oi)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

if (tail == 0 || tail->tail != 0)

// Must be exactly two alts

return false;

for (AltNode\* curr = this; curr != 0; curr = curr->tail)

{

if (!oi->BeginConcat())

return false;

if (!curr->head->IsOctoquad(compiler, oi))

return false;

}

return true;

}

bool AltNode::IsCharTrieArm(Compiler& compiler, uint& accNumAlts) const

{

return false;

}

bool AltNode::BuildCharTrie(Compiler& compiler, CharTrie\* trie, Node\* cont, bool isAcceptFirst) const

{

Assert(false);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void AltNode::Print(DebugWriter\* w, const Char\* litbuf) const

{

w->PrintEOL(L"Alt()");

PrintAnnotations(w);

w->PrintEOL(L"{");

w->Indent();

for (const AltNode \*curr = this; curr != 0; curr = curr->tail)

curr->head->Print(w, litbuf);

w->Unindent();

w->PrintEOL(L"}");

}

#endif

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

// DefineGroupNode

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

CharCount DefineGroupNode::LiteralLength() const

{

return 0;

}

bool DefineGroupNode::IsCharOrPositiveSet() const

{

return false;

}

CharCount DefineGroupNode::TransferPass0(Compiler& compiler, const Char\* litbuf)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

Assert(groupId > 0 && groupId < compiler.program->numGroups);

return body->TransferPass0(compiler, litbuf);

}

void DefineGroupNode::TransferPass1(Compiler& compiler, const Char\* litbuf)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

body->TransferPass1(compiler, litbuf);

}

bool DefineGroupNode::IsRefiningAssertion(Compiler& compiler)

{

return false;

}

void DefineGroupNode::AnnotatePass0(Compiler& compiler)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

body->AnnotatePass0(compiler);

isWord = body->isWord;

}

void DefineGroupNode::AnnotatePass1(Compiler& compiler, bool parentNotInLoop, bool parentAtLeastOnce, bool parentNotSpeculative, bool parentNotNegated)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

features = HasDefineGroup;

body->AnnotatePass1(compiler, parentNotInLoop, parentAtLeastOnce, parentNotSpeculative, parentNotNegated);

features |= body->features;

thisConsumes = body->thisConsumes;

firstSet = body->firstSet;

isFirstExact = body->isFirstExact;

isThisIrrefutable = body->isThisIrrefutable;

isThisWillNotProgress = body->isThisWillNotProgress;

isThisWillNotRegress = body->isThisWillNotRegress;

isNotInLoop = parentNotInLoop;

isAtLeastOnce = parentAtLeastOnce;

isNotSpeculative = parentNotSpeculative;

isNotNegated = parentNotNegated;

}

void DefineGroupNode::AnnotatePass2(Compiler& compiler, CountDomain accumConsumes, bool accumPrevWillNotProgress, bool accumPrevWillNotRegress)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

prevConsumes = accumConsumes;

isPrevWillNotProgress = accumPrevWillNotProgress;

isPrevWillNotRegress = accumPrevWillNotRegress;

body->AnnotatePass2(compiler, accumConsumes, accumPrevWillNotProgress, accumPrevWillNotRegress);

}

void DefineGroupNode::AnnotatePass3(Compiler& compiler, CountDomain accumConsumes, CharSet<Char>\* accumFollow, bool accumFollowIrrefutable, bool accumFollowEOL)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

followConsumes = accumConsumes;

followSet = accumFollow;

isFollowIrrefutable = accumFollowIrrefutable;

isFollowEOL = accumFollowEOL;

body->AnnotatePass3(compiler, accumConsumes, accumFollow, accumFollowIrrefutable, accumFollowEOL);

hasInitialHardFailBOI = body->hasInitialHardFailBOI;

}

void DefineGroupNode::AnnotatePass4(Compiler& compiler)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

body->AnnotatePass4(compiler);

isDeterministic = body->isDeterministic;

// If the follow is irrefutable and we're not in an assertion, then we are not going to backtrack beyond this point, so

// we don't need to save the group before updating it

noNeedToSave = isFollowIrrefutable && isNotSpeculative;

// Compilation scheme: Chomp

//

// Body consists of a loop node with a Chomp compilation scheme.

if(body->tag == NodeTag::Loop)

{

const LoopNode \*const loop = static\_cast<const LoopNode \*>(body);

if(loop->scheme == LoopNode::CompilationScheme::Chomp && loop->repeats.lower <= 1 && loop->repeats.IsUnbounded())

{

// \*\*COMMIT\*\*

scheme = Chomp;

return;

}

}

// Compilation scheme: Fixed

//

// Body has fixed width, so don't need a Begin instruction to keep track of the input start offset of the group.

if (body->thisConsumes.IsFixed())

{

// \*\*COMMIT\*\*

scheme = Fixed;

return;

}

// Compilation scheme: BeginEnd

//

// If both the body and the follow are irrefutable, we're not in any loops, and we're not in an assertion,

// then we don't need to save the group before updating it.

// \*\*COMMIT\*\*

scheme = BeginEnd;

}

bool DefineGroupNode::SupportsPrefixSkipping(Compiler& compiler) const

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

if (scheme != Fixed)

// We can't skip over part of the match if the BeginDefineGroup must capture it's start

return false;

return body->SupportsPrefixSkipping(compiler);

}

Node\* DefineGroupNode::HeadSyncronizingNode(Compiler& compiler)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

if (scheme != Fixed)

// Can't skip BeginDefineGroup

return 0;

return body->HeadSyncronizingNode(compiler);

}

CharCount DefineGroupNode::MinSyncronizingLiteralLength(Compiler& compiler, int& numLiterals) const

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

return body->MinSyncronizingLiteralLength(compiler, numLiterals);

}

void DefineGroupNode::CollectSyncronizingLiterals(Compiler& compiler, ScannersMixin& scanners) const

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

body->CollectSyncronizingLiterals(compiler, scanners);

}

void DefineGroupNode::BestSyncronizingNode(Compiler& compiler, Node\*& bestNode)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

body->BestSyncronizingNode(compiler, bestNode);

}

void DefineGroupNode::AccumDefineGroups(Js::ScriptContext\* scriptContext, int& minGroup, int& maxGroup)

{

PROBE\_STACK(scriptContext, Js::Constants::MinStackRegex);

if (groupId < minGroup)

minGroup = groupId;

if (groupId > maxGroup)

maxGroup = groupId;

body->AccumDefineGroups(scriptContext, minGroup, maxGroup);

}

void DefineGroupNode::Emit(Compiler& compiler, CharCount& skipped)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

switch (scheme)

{

case Chomp:

{

// Compilation scheme:

//

// Chomp(Char|Set)Group

Assert(body->tag == NodeTag::Loop);

const LoopNode \*const loop = static\_cast<const LoopNode \*>(body);

const CharSet<Char> \*const loopBodyFirstSet = loop->body->firstSet;

const CountDomain &repeats = loop->repeats;

Assert(repeats.lower <= 1 && repeats.IsUnbounded());

if(loopBodyFirstSet->IsSingleton())

{

const Char c = loopBodyFirstSet->Singleton();

if(repeats.lower == 0)

EMIT(compiler, ChompCharGroupInst<ChompMode::Star>, c, groupId, noNeedToSave);

else

EMIT(compiler, ChompCharGroupInst<ChompMode::Plus>, c, groupId, noNeedToSave);

}

else

{

Assert(repeats.lower <= 1 && repeats.IsUnbounded());

RuntimeCharSet<Char> \*runtimeSet;

if(repeats.lower == 0)

runtimeSet = &EMIT(compiler, ChompSetGroupInst<ChompMode::Star>, groupId, noNeedToSave)->set;

else

runtimeSet = &EMIT(compiler, ChompSetGroupInst<ChompMode::Plus>, groupId, noNeedToSave)->set;

runtimeSet->CloneFrom(compiler.rtAllocator, \*loopBodyFirstSet);

}

break;

}

case Fixed:

{

// Compilation scheme:

//

// <body>

// DefineGroup

Assert(body->thisConsumes.IsFixed());

body->Emit(compiler, skipped);

EMIT(compiler, DefineGroupFixedInst, groupId, body->thisConsumes.lower, noNeedToSave);

break;

}

case BeginEnd:

{

// Compilation scheme:

//

// BeginDefineGroup

// <body>

// EndDefineGroup

Assert(skipped == 0);

EMIT(compiler, BeginDefineGroupInst, groupId);

body->Emit(compiler, skipped);

EMIT(compiler, EndDefineGroupInst, groupId, noNeedToSave);

break;

}

}

}

CharCount DefineGroupNode::EmitScan(Compiler& compiler, bool isHeadSyncronizingNode)

{

Assert(false);

return 0;

}

bool DefineGroupNode::IsOctoquad(Compiler& compiler, OctoquadIdentifier\* oi)

{

return false;

}

bool DefineGroupNode::IsCharTrieArm(Compiler& compiler, uint& accNumAlts) const

{

return false;

}

bool DefineGroupNode::BuildCharTrie(Compiler& compiler, CharTrie\* trie, Node\* cont, bool isAcceptFirst) const

{

Assert(false);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void DefineGroupNode::Print(DebugWriter\* w, const Char\* litbuf) const

{

w->PrintEOL(L"DefineGroup(%d)", groupId);

PrintAnnotations(w);

w->PrintEOL(L"{");

w->Indent();

body->Print(w, litbuf);

w->Unindent();

w->PrintEOL(L"}");

}

#endif

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

// MatchGroupNode

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

CharCount MatchGroupNode::LiteralLength() const

{

return 0;

}

bool MatchGroupNode::IsCharOrPositiveSet() const

{

return false;

}

CharCount MatchGroupNode::TransferPass0(Compiler& compiler, const Char\* litbuf)

{

Assert(groupId > 0 && groupId < compiler.program->numGroups);

return 0;

}

void MatchGroupNode::TransferPass1(Compiler& compiler, const Char\* litbuf)

{

}

bool MatchGroupNode::IsRefiningAssertion(Compiler& compiler)

{

return false;

}

void MatchGroupNode::AnnotatePass0(Compiler& compiler)

{

isWord = false;

}

void MatchGroupNode::AnnotatePass1(Compiler& compiler, bool parentNotInLoop, bool parentAtLeastOnce, bool parentNotSpeculative, bool parentNotNegated)

{

features = HasMatchGroup;

thisConsumes.lower = 0;

thisConsumes.upper = CharCountFlag;

firstSet = compiler.standardChars->GetFullSet();

isFirstExact = false;

isThisIrrefutable = false;

isThisWillNotProgress = true;

isThisWillNotRegress = true;

isNotInLoop = parentNotInLoop;

isAtLeastOnce = parentAtLeastOnce;

isNotSpeculative = parentNotSpeculative;

isNotNegated = parentNotNegated;

}

void MatchGroupNode::AnnotatePass2(Compiler& compiler, CountDomain accumConsumes, bool accumPrevWillNotProgress, bool accumPrevWillNotRegress)

{

prevConsumes = accumConsumes;

isPrevWillNotProgress = accumPrevWillNotProgress;

isPrevWillNotRegress = accumPrevWillNotRegress;

}

void MatchGroupNode::AnnotatePass3(Compiler& compiler, CountDomain accumConsumes, CharSet<Char>\* accumFollow, bool accumFollowIrrefutable, bool accumFollowEOL)

{

followConsumes = accumConsumes;

followSet = accumFollow;

isFollowIrrefutable = accumFollowIrrefutable;

isFollowEOL = accumFollowEOL;

}

void MatchGroupNode::AnnotatePass4(Compiler& compiler)

{

isDeterministic = true;

}

bool MatchGroupNode::SupportsPrefixSkipping(Compiler& compiler) const

{

return false;

}

Node\* MatchGroupNode::HeadSyncronizingNode(Compiler& compiler)

{

return 0;

}

CharCount MatchGroupNode::MinSyncronizingLiteralLength(Compiler& compiler, int& numLiterals) const

{

return 0;

}

void MatchGroupNode::CollectSyncronizingLiterals(Compiler& compiler, ScannersMixin& scanners) const

{

Assert(false);

}

void MatchGroupNode::BestSyncronizingNode(Compiler& compiler, Node\*& bestNode)

{

}

void MatchGroupNode::AccumDefineGroups(Js::ScriptContext\* scriptContext, int& minGroup, int& maxGroup)

{

}

void MatchGroupNode::Emit(Compiler& compiler, CharCount& skipped)

{

Assert(skipped == 0);

//

// Compilation scheme:

//

// MatchGroup

//

EMIT(compiler, MatchGroupInst, groupId);

}

CharCount MatchGroupNode::EmitScan(Compiler& compiler, bool isHeadSyncronizingNode)

{

Assert(false);

return 0;

}

bool MatchGroupNode::IsOctoquad(Compiler& compiler, OctoquadIdentifier\* oi)

{

return false;

}

bool MatchGroupNode::IsCharTrieArm(Compiler& compiler, uint& accNumAlts) const

{

return false;

}

bool MatchGroupNode::BuildCharTrie(Compiler& compiler, CharTrie\* trie, Node\* cont, bool isAcceptFirst) const

{

Assert(false);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void MatchGroupNode::Print(DebugWriter\* w, const Char\* litbuf) const

{

w->PrintEOL(L"MatchGroup(%d)", groupId);

PrintAnnotations(w);

}

#endif

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

// LoopNode

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

CharCount LoopNode::LiteralLength() const

{

return 0;

}

bool LoopNode::IsCharOrPositiveSet() const

{

return false;

}

CharCount LoopNode::TransferPass0(Compiler& compiler, const Char\* litbuf)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

Assert(repeats.upper == CharCountFlag || repeats.upper > 0);

Assert(repeats.upper == CharCountFlag || repeats.upper >= repeats.lower);

Assert(!(repeats.lower == 1 && repeats.upper == 1));

return body->TransferPass0(compiler, litbuf);

}

void LoopNode::TransferPass1(Compiler& compiler, const Char\* litbuf)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

body->TransferPass1(compiler, litbuf);

}

bool LoopNode::IsRefiningAssertion(Compiler& compiler)

{

return false;

}

void LoopNode::AnnotatePass0(Compiler& compiler)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

body->AnnotatePass0(compiler);

isWord = !repeats.CouldMatchEmpty() && body->isWord;

}

void LoopNode::AnnotatePass1(Compiler& compiler, bool parentNotInLoop, bool parentAtLeastOnce, bool parentNotSpeculative, bool parentNotNegated)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

features = HasLoop;

isNotInLoop = parentNotInLoop;

isAtLeastOnce = parentAtLeastOnce;

isNotSpeculative = parentNotSpeculative;

isNotNegated = parentNotNegated;

body->AnnotatePass1(compiler, false, parentAtLeastOnce && repeats.lower > 0, parentNotSpeculative, isNotNegated);

features |= body->features;

thisConsumes = body->thisConsumes;

thisConsumes.Mult(repeats);

firstSet = body->firstSet;

isFirstExact = repeats.lower > 0 && body->isFirstExact;

isThisIrrefutable = repeats.CouldMatchEmpty() || body->isThisIrrefutable;

// Caution: Even if a greedy loop has a 'isThisWillNotProgress' body, if the body has choicepoints then

// a backtrack could resume execution at an earlier loop iteration, which may then continue to repeat

// the loop beyond the input offset which triggered the backtrack. Ideally we'd use the body's isDeterministic

// flag to tell us when that can't happen, but it's not available till pass 4, so we must make do with

// a simple-minded structural approximation.

isThisWillNotProgress = (isGreedy || repeats.IsExact(1)) && body->isThisWillNotProgress && body->IsObviouslyDeterministic();

isThisWillNotRegress = (!isGreedy || repeats.IsExact(1)) && body->isThisWillNotRegress && body->IsObviouslyDeterministic();

}

void LoopNode::AnnotatePass2(Compiler& compiler, CountDomain accumConsumes, bool accumPrevWillNotProgress, bool accumPrevWillNotRegress)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

prevConsumes = accumConsumes;

isPrevWillNotProgress = accumPrevWillNotProgress;

isPrevWillNotRegress = accumPrevWillNotRegress;

// May have already gone through loop when starting body

CountDomain bodyConsumes = body->thisConsumes;

CharCountOrFlag prevMax = repeats.upper;

if (prevMax != CharCountFlag)

prevMax--;

CountDomain prevLoops(0, prevMax);

bodyConsumes.Mult(prevLoops);

accumConsumes.Add(bodyConsumes);

body->AnnotatePass2(compiler, accumConsumes, accumPrevWillNotProgress && isThisWillNotProgress, accumPrevWillNotRegress && isThisWillNotRegress);

}

void LoopNode::AnnotatePass3(Compiler& compiler, CountDomain accumConsumes, CharSet<Char>\* accumFollow, bool accumFollowIrrefutable, bool accumFollowEOL)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

followConsumes = accumConsumes;

followSet = accumFollow;

isFollowIrrefutable = accumFollowIrrefutable;

isFollowEOL = accumFollowEOL;

// May go through loop again when leaving body

CountDomain bodyConsumes = body->thisConsumes;

CharCountOrFlag nextMax = repeats.upper;

if (nextMax != CharCountFlag)

nextMax--;

CountDomain nextLoops(0, nextMax);

bodyConsumes.Mult(nextLoops);

accumConsumes.Add(bodyConsumes);

CharSet<Char>\* innerFollow = Anew(compiler.ctAllocator, UnicodeCharSet);

innerFollow->UnionInPlace(compiler.ctAllocator, \*accumFollow);

innerFollow->UnionInPlace(compiler.ctAllocator, \*body->firstSet);

/\*

All of the following must be true for the loop body's follow to be irrefutable:

The loop's follow is irrefutable.

The loop can complete the required minimum number of iterations of the body without backtracking into a completed

iteration of the body.

- If repeats.lower == 0, the required minimum number of iterations is met without executing the body

- If repeats.lower == 1

- If the first iteration of the body fails, there is no previous iteration of the body to backtrack into

- After completing the first iteration of the body, the loop cannot reject the first iteration for not

making progress because the iteration is required for the loop to succeed

- If repeats.lower >= 2

- If the second iteration of the body fails, it will backtrack into the first iteration of the body

- To prevent this, the body must be irrefutable

After completing the required minimum number of iterations of the body, the loop cannot reject a subsequent

completed iteration of the body for not making progress.

- If !isGreedy || repeats.IsFixed(), there will not be any more iterations of the body, as it will proceed to

the irrefutable follow

- If !body->thisConsumes.CouldMatchEmpty(), subsequent iterations of the body cannot complete without making

progress

\*/

const bool isBodyFollowIrrefutable =

accumFollowIrrefutable &&

(repeats.lower <= 1 || body->isThisIrrefutable) &&

(!isGreedy || !body->thisConsumes.CouldMatchEmpty() || repeats.IsFixed());

body->AnnotatePass3(compiler, accumConsumes, innerFollow, isBodyFollowIrrefutable, false);

hasInitialHardFailBOI = body->hasInitialHardFailBOI;

}

void LoopNode::AnnotatePass4(Compiler& compiler)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

body->AnnotatePass4(compiler);

isDeterministic = body->isDeterministic;

//

// Loops can be defined by unfolding:

// r\* === (rr\*|)

// r\*? === (|rr\*?)

// Thus many of the optimizations for alternatives carry over to loops.

//

//

// Compilation scheme: None

//

// If overall loop is empty-only then emit nothing.

// (Parser has already eliminated loops with upper == 0, so this can only happen if the body is empty-only)

//

// If loop is non-greedy with lower 0 and follow is irrefutable, then loop body will never be executed

// no emit nothing.

//

if (body->IsEmptyOnly() ||

(!isGreedy && repeats.lower == 0 && isFollowIrrefutable))

{

// \*\*COMMIT\*\*

scheme = None;

return;

}

//

// Compilation scheme: Chain/Try

//

// If loop is greedy, with lower 0 and upper 1, then we'd like to treat it as body|<empty> so as to avoid all the loop

// overhead. However, if the body could match empty, then a match against empty must be treated as an 'iteration' of the

// loop body which made no progress. So we treat as a general loop in that case. Otherwise, we may inline two of

// AltNode's compilation schemes:

// Examples:

// - /(a\*)?/.exec("") must leave group 1 undefined rather than empty.

// - /(?:a||b)?/.exec("b") chooses the empty alt, then must backtrack due to no progress, and match 'b'.

// This is not the same as /a||b|/, as picking the first empty alt would result in success.

//

// (cf AltNode's None/Switch/Chain/Set, isOptional compilation scheme)

// If body cannot match empty, follow cannot match empty, and the body FIRST set is disjoint from the FOLLOW

// set, then we can commit to the body using a 1 char lookahead.

//

// If body cannot match empty, and follow is irrefutable, then we can commit to the body using a 1 char

// lookahead provided:

// \*\* the body can't fail if given an arbitrary input starting with a character in its FIRST set \*\*

//

// (cf AltNode's Try compilation scheme)

// Otherwise, protect body by a Try instruction.

//

if (isGreedy && repeats.lower == 0 && repeats.upper == 1 && !body->thisConsumes.CouldMatchEmpty())

{

// \*\*COMMIT\*\*

// Note that the FIRST of the loop is already the union of the body FIRST and the loop FOLLOW

if (!body->thisConsumes.CouldMatchEmpty() &&

((!followConsumes.CouldMatchEmpty() && firstSet->Count() == body->firstSet->Count() + followSet->Count()) ||

(isFollowIrrefutable && body->IsSimpleOneChar())))

{

if (body->IsSimpleOneChar())

scheme = Set;

else

scheme = Chain;

}

else

{

scheme = Try;

isDeterministic = false; // NON-DETERMINISTIC

}

return;

}

//

// Compilation scheme: Chomp/ChompGroupLastChar

//

// If the body is a simple-one-char, or a group of a simple-one-char, and either:

// - follow is non-empty and FIRST and FOLLOW are disjoint

// - loop is greedy and follow is irrefutable

// - follow is EOL

// then consume up to upper number of characters in FIRST and fail if number consumed is not >= lower.

//

if (body->IsSimpleOneChar() || (body->tag == DefineGroup && ((DefineGroupNode\*)body)->body->IsSimpleOneChar()))

{

if (!followConsumes.CouldMatchEmpty())

{

CharSet<Char> unionSet;

CharCount totalChars = 0;

unionSet.UnionInPlace(compiler.ctAllocator, \*body->firstSet);

totalChars += body->firstSet->Count();

unionSet.UnionInPlace(compiler.ctAllocator, \*followSet);

totalChars += followSet->Count();

if (totalChars == unionSet.Count())

{

// \*\*COMMIT\*\*

if (body->tag == DefineGroup)

{

noNeedToSave = isFollowIrrefutable && isNotInLoop && isNotSpeculative;

scheme = ChompGroupLastChar;

}

else

scheme = Chomp;

return;

}

}

if ((isGreedy && isFollowIrrefutable) || isFollowEOL)

{

// \*\*COMMIT\*\*

if (body->tag == DefineGroup)

{

noNeedToSave = isFollowIrrefutable && isNotInLoop && isNotSpeculative;

scheme = ChompGroupLastChar;

}

else

scheme = Chomp;

return;

}

}

//

// Compilation scheme: Guarded

//

// If body cannot match empty, follow cannot match empty, and FIRST of body and FOLLOW are

// disjoint then can use 1 char lookahead to decide whether to commit to another loop body.

// (If the loop body fails then we know the follow will fail even with one more/fewer iterations of the

// loop body, so we can let that failure propagate without needing to push choicepoints.)

//

if (!body->thisConsumes.CouldMatchEmpty() && !followConsumes.CouldMatchEmpty())

{

CharSet<Char> unionSet;

CharCount totalChars = 0;

unionSet.UnionInPlace(compiler.ctAllocator, \*body->firstSet);

totalChars += body->firstSet->Count();

unionSet.UnionInPlace(compiler.ctAllocator, \*followSet);

totalChars += followSet->Count();

if (totalChars == unionSet.Count())

{

// \*\*COMMIT\*\*

scheme = Guarded;

return;

}

}

//

// Compilation scheme: Fixed/FixedSet/FixedGroupLastIteration

//

// If loop is greedy, body is deterministic, non-zero fixed width, and either does not define any groups

// or has one outermost group, then we can keep track of the backtracking information in constant space.

//

// If body does have an outer group, we can avoid saving the existing group contents if the follow

// is irrefutable, we're not in an outer loop, and we're not in an assertion.

//

if (isGreedy && body->isDeterministic && !body->thisConsumes.CouldMatchEmpty() && body->thisConsumes.IsFixed())

{

if (body->tag == DefineGroup)

{

DefineGroupNode\* bodyGroup = (DefineGroupNode\*)body;

if (!bodyGroup->body->ContainsDefineGroup())

{

// \*\*COMMIT\*\*

scheme = FixedGroupLastIteration;

noNeedToSave = isFollowIrrefutable && isNotInLoop && isNotSpeculative;

isDeterministic = false; // NON-DETERMINISTIC;

return;

}

}

else if (body->IsSimpleOneChar())

{

// \*\*COMMIT\*\*

scheme = FixedSet;

isDeterministic = false; // NON-DETERMINISTIC

return;

}

else if (!body->ContainsDefineGroup())

{

// \*\*COMMIT\*\*

scheme = Fixed;

isDeterministic = false; // NON-DETERMINISTIC

return;

}

}

//

// Compilation scheme: GreedyNoBacktrack

//

// If loop is greedy with lower == 0 and upper == inf, the loop body is deterministic and does not define

// groups, and follow is irrefutable, then we will never have to try fewer iterations of the loop once

// entering the follow. Thus we only need one continuation record on the stack to protect against failure

// for each attempt at the loop body.

//

if (isGreedy && repeats.lower == 0 && repeats.upper == CharCountFlag && body->isDeterministic && !body->ContainsDefineGroup() && isFollowIrrefutable)

{

// \*\*COMMIT\*\*

scheme = GreedyNoBacktrack;

return;

}

//

// Compilation scheme: BeginEnd

//

scheme = BeginEnd;

isDeterministic = false; // NON-DETERMINISTIC

}

bool LoopNode::SupportsPrefixSkipping(Compiler& compiler) const

{

return false;

}

Node\* LoopNode::HeadSyncronizingNode(Compiler& compiler)

{

return 0;

}

CharCount LoopNode::MinSyncronizingLiteralLength(Compiler& compiler, int& numLiterals) const

{

return 0;

}

void LoopNode::CollectSyncronizingLiterals(Compiler& compiler, ScannersMixin& scanners) const

{

Assert(false);

}

void LoopNode::BestSyncronizingNode(Compiler& compiler, Node\*& bestNode)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

if (repeats.lower > 0)

body->BestSyncronizingNode(compiler, bestNode);

// else: can't be sure loop will be taken

}

void LoopNode::AccumDefineGroups(Js::ScriptContext\* scriptContext, int& minGroup, int& maxGroup)

{

PROBE\_STACK(scriptContext, Js::Constants::MinStackRegex);

body->AccumDefineGroups(scriptContext, minGroup, maxGroup);

}

void LoopNode::Emit(Compiler& compiler, CharCount& skipped)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

Assert(skipped == 0);

switch (scheme)

{

case BeginEnd:

{

//

// Compilation scheme:

//

// Lloop: BeginLoop Lexit

// <loop body>

// RepeatLoop Lloop

// Lexit:

//

int minBodyGroupId = compiler.program->numGroups;

int maxBodyGroupId = -1;

body->AccumDefineGroups(compiler.scriptContext, minBodyGroupId, maxBodyGroupId);

Label beginLabel = compiler.CurrentLabel();

Label fixup = compiler.GetFixup(&EMIT(compiler, BeginLoopInst, compiler.NextLoopId(), repeats, !isNotInLoop, !body->isDeterministic, minBodyGroupId, maxBodyGroupId, isGreedy)->exitLabel);

body->Emit(compiler, skipped);

EMIT(compiler, RepeatLoopInst, beginLabel);

compiler.DoFixup(fixup, compiler.CurrentLabel());

break;

}

case None:

{

// Nothing to emit

break;

}

case Chomp:

{

//

// Compilation scheme:

//

// Chomp(Char|Set)(Star|Plus|Bounded)

//

if(body->firstSet->IsSingleton())

{

if(repeats.lower <= 1 && repeats.IsUnbounded())

{

if(repeats.lower == 0)

EMIT(compiler, ChompCharInst<ChompMode::Star>, body->firstSet->Singleton());

else

EMIT(compiler, ChompCharInst<ChompMode::Plus>, body->firstSet->Singleton());

}

else

EMIT(compiler, ChompCharBoundedInst, body->firstSet->Singleton(), repeats);

}

else

{

if(repeats.lower <= 1 && repeats.IsUnbounded())

{

if(repeats.lower == 0)

EMIT(compiler, ChompSetInst<ChompMode::Star>)->set.CloneFrom(compiler.rtAllocator, \*body->firstSet);

else

EMIT(compiler, ChompSetInst<ChompMode::Plus>)->set.CloneFrom(compiler.rtAllocator, \*body->firstSet);

}

else

EMIT(compiler, ChompSetBoundedInst, repeats)->set.CloneFrom(compiler.rtAllocator, \*body->firstSet);

}

break;

}

case ChompGroupLastChar:

{

//

// Compilation scheme:

//

// ChompSetGroup

//

Assert(body->tag == DefineGroup);

DefineGroupNode\* bodyGroup = (DefineGroupNode\*)body;

EMIT(compiler, ChompSetBoundedGroupLastCharInst, repeats, bodyGroup->groupId, noNeedToSave)->set.CloneFrom(compiler.rtAllocator, \*body->firstSet);

break;

}

case Guarded:

{

//

// Compilation scheme:

//

// Lloop: BeginLoopIf(Char|Set) Lexit

// <loop body>

// RepeatLoopIf(Char|Set) Lloop

// Lexit:

//

int minBodyGroupId = compiler.program->numGroups;

int maxBodyGroupId = -1;

body->AccumDefineGroups(compiler.scriptContext, minBodyGroupId, maxBodyGroupId);

Label beginLabel = compiler.CurrentLabel();

Label exitFixup;

if (body->firstSet->IsSingleton())

exitFixup = compiler.GetFixup(&EMIT(compiler, BeginLoopIfCharInst, body->firstSet->Singleton(), compiler.NextLoopId(), repeats, !isNotInLoop, !body->isDeterministic, minBodyGroupId, maxBodyGroupId)->exitLabel);

else

{

BeginLoopIfSetInst\* i = EMIT(compiler, BeginLoopIfSetInst, compiler.NextLoopId(), repeats, !isNotInLoop, !body->isDeterministic, minBodyGroupId, maxBodyGroupId);

i->set.CloneFrom(compiler.rtAllocator, \*body->firstSet);

exitFixup = compiler.GetFixup(&i->exitLabel);

}

body->Emit(compiler, skipped);

if (body->firstSet->IsSingleton())

EMIT(compiler, RepeatLoopIfCharInst, beginLabel);

else

EMIT(compiler, RepeatLoopIfSetInst, beginLabel);

compiler.DoFixup(exitFixup, compiler.CurrentLabel());

break;

}

case Fixed:

{

//

// Compilation scheme:

//

// Lloop: BeginLoopFixed Lexit

// <loop body>

// RepeatLoopFixed Lloop

// Lexit:

//

Assert(!body->ContainsDefineGroup());

Assert(body->thisConsumes.IsFixed());

Assert(body->thisConsumes.lower > 0);

Assert(body->isDeterministic);

Label beginLabel = compiler.CurrentLabel();

Label fixup = compiler.GetFixup(&EMIT(compiler, BeginLoopFixedInst, compiler.NextLoopId(), repeats, !isNotInLoop, body->thisConsumes.lower)->exitLabel);

body->Emit(compiler, skipped);

EMIT(compiler, RepeatLoopFixedInst, beginLabel);

compiler.DoFixup(fixup, compiler.CurrentLabel());

break;

}

case FixedSet:

{

//

// Compilation scheme:

//

// LoopSet

//

Assert(body->IsSimpleOneChar());

EMIT(compiler, LoopSetInst, compiler.NextLoopId(), repeats, !isNotInLoop)->set.CloneFrom(compiler.rtAllocator, \*body->firstSet);

break;

}

case FixedGroupLastIteration:

{

//

// Compilation scheme:

//

// Lloop: BeginLoopFixedGroupLastIteration Lexit

// <loop body>

// RepeatLoopFixedGroupLastIteration Lloop

// Lexit:

//

Assert(body->tag == DefineGroup);

DefineGroupNode\* bodyGroup = (DefineGroupNode\*)body;

Assert(body->thisConsumes.IsFixed());

Assert(body->thisConsumes.lower > 0);

Assert(body->isDeterministic);

Label beginLabel = compiler.CurrentLabel();

Label fixup = compiler.GetFixup(&EMIT(compiler, BeginLoopFixedGroupLastIterationInst, compiler.NextLoopId(), repeats, !isNotInLoop, body->thisConsumes.lower, bodyGroup->groupId, noNeedToSave)->exitLabel);

bodyGroup->body->Emit(compiler, skipped);

EMIT(compiler, RepeatLoopFixedGroupLastIterationInst, beginLabel);

compiler.DoFixup(fixup, compiler.CurrentLabel());

break;

}

case GreedyNoBacktrack:

{

//

// Compilation scheme:

//

// Lloop: BeginGreedyLoopNoBacktrack Lexit

// <loop body>

// RepeatGreedyLoopNoBacktrack Lloop

// Lexit:

//

Assert(!body->ContainsDefineGroup());

Assert(isGreedy);

Assert(repeats.lower == 0);

Assert(repeats.upper == CharCountFlag);

Assert(body->isDeterministic);

Label beginLabel = compiler.CurrentLabel();

Label fixup = compiler.GetFixup(&EMIT(compiler, BeginGreedyLoopNoBacktrackInst, compiler.NextLoopId())->exitLabel);

body->Emit(compiler, skipped);

EMIT(compiler, RepeatGreedyLoopNoBacktrackInst, beginLabel);

compiler.DoFixup(fixup, compiler.CurrentLabel());

break;

}

case Set:

{

//

// Compilation scheme:

//

// OptMatch(Char|Set)

//

Assert(!body->ContainsDefineGroup() || !body->thisConsumes.CouldMatchEmpty());

if (body->firstSet->IsSingleton())

EMIT(compiler, OptMatchCharInst, body->firstSet->Singleton());

else

EMIT(compiler, OptMatchSetInst)->set.CloneFrom(compiler.rtAllocator, \*body->firstSet);

break;

}

case Chain:

{

//

// Compilation scheme:

//

// JumpIfNot(Char|Set) Lexit

// <body>

// Lexit:

//

//

Assert(!body->ContainsDefineGroup() || !body->thisConsumes.CouldMatchEmpty());

Label jumpFixup = 0;

CharCount bodySkipped = 0;

if (body->firstSet->IsSingleton())

{

if (body->SupportsPrefixSkipping(compiler))

{

jumpFixup = compiler.GetFixup(&EMIT(compiler, MatchCharOrJumpInst, body->firstSet->Singleton())->targetLabel);

bodySkipped = 1;

}

else

jumpFixup = compiler.GetFixup(&EMIT(compiler, JumpIfNotCharInst, body->firstSet->Singleton())->targetLabel);

}

else

{

if (body->SupportsPrefixSkipping(compiler))

{

MatchSetOrJumpInst\* const i = EMIT(compiler, MatchSetOrJumpInst);

i->set.CloneFrom(compiler.rtAllocator, \*body->firstSet);

jumpFixup = compiler.GetFixup(&i->targetLabel);

bodySkipped = 1;

}

else

{

JumpIfNotSetInst\* const i = EMIT(compiler, JumpIfNotSetInst);

i->set.CloneFrom(compiler.rtAllocator, \*body->firstSet);

jumpFixup = compiler.GetFixup(&i->targetLabel);

}

}

body->Emit(compiler, bodySkipped);

compiler.DoFixup(jumpFixup, compiler.CurrentLabel());

break;

}

case Try:

{

//

// Compilation scheme:

//

// Try((If|Match)(Char|Set))? Lexit

// <loop body>

// Lexit:

//

Assert(!body->ContainsDefineGroup() || !body->thisConsumes.CouldMatchEmpty());

Label tryFixup = 0;

CharCount bodySkipped = 0;

// HEURISTIC: if the first set of the body is exact or small, and the

// body does not match empty, then it's probably worth using

// a Try(If|Match)(Char|Set)

if (!body->thisConsumes.CouldMatchEmpty() &&

(body->isFirstExact || body->firstSet->Count() <= maxCharsForConditionalTry))

{

if (body->SupportsPrefixSkipping(compiler))

{

if (body->firstSet->IsSingleton())

tryFixup = compiler.GetFixup(&EMIT(compiler, TryMatchCharInst, body->firstSet->Singleton())->failLabel);

else

{

TryMatchSetInst\* const i = EMIT(compiler, TryMatchSetInst);

i->set.CloneFrom(compiler.rtAllocator, \*body->firstSet);

tryFixup = compiler.GetFixup(&i->failLabel);

}

bodySkipped = 1;

}

else

{

if(body->firstSet->IsSingleton())

tryFixup = compiler.GetFixup(&EMIT(compiler, TryIfCharInst, body->firstSet->Singleton())->failLabel);

else

{

TryIfSetInst\* const i = EMIT(compiler, TryIfSetInst);

i->set.CloneFrom(compiler.rtAllocator, \*body->firstSet);

tryFixup = compiler.GetFixup(&i->failLabel);

}

}

}

else

tryFixup = compiler.GetFixup(&EMIT(compiler, TryInst)->failLabel);

body->Emit(compiler, bodySkipped);

// Fixup Try

compiler.DoFixup(tryFixup, compiler.CurrentLabel());

break;

}

}

}

CharCount LoopNode::EmitScan(Compiler& compiler, bool isHeadSyncronizingNode)

{

Assert(false);

return 0;

}

bool LoopNode::IsOctoquad(Compiler& compiler, OctoquadIdentifier\* oi)

{

return false;

}

bool LoopNode::IsCharTrieArm(Compiler& compiler, uint& accNumAlts) const

{

return false;

}

bool LoopNode::BuildCharTrie(Compiler& compiler, CharTrie\* trie, Node\* cont, bool isAcceptFirst) const

{

Assert(false);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void LoopNode::Print(DebugWriter\* w, const Char\* litbuf) const

{

w->Print(L"Loop(");

repeats.Print(w);

w->PrintEOL(L", %s)", isGreedy ? L"greedy" : L"non-greedy");

PrintAnnotations(w);

w->PrintEOL(L"{");

w->Indent();

body->Print(w, litbuf);

w->Unindent();

w->PrintEOL(L"}");

}

#endif

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

// AssertionNode

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

CharCount AssertionNode::LiteralLength() const

{

return 0;

}

bool AssertionNode::IsCharOrPositiveSet() const

{

return false;

}

CharCount AssertionNode::TransferPass0(Compiler& compiler, const Char\* litbuf)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

return body->TransferPass0(compiler, litbuf);

}

void AssertionNode::TransferPass1(Compiler& compiler, const Char\* litbuf)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

body->TransferPass1(compiler, litbuf);

}

bool AssertionNode::IsRefiningAssertion(Compiler& compiler)

{

return !isNegation;

}

void AssertionNode::AnnotatePass0(Compiler& compiler)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

isWord = false;

body->AnnotatePass0(compiler);

}

void AssertionNode::AnnotatePass1(Compiler& compiler, bool parentNotInLoop, bool parentAtLeastOnce, bool parentNotSpeculative, bool parentNotNegated)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

features = HasAssertion;

body->AnnotatePass1(compiler, parentNotInLoop, parentAtLeastOnce, false, parentNotNegated && !isNegation);

features |= body->features;

thisConsumes.Exact(0);

if (isNegation)

firstSet = compiler.standardChars->GetFullSet();

else

firstSet = body->firstSet;

isFirstExact = false;

if (isNegation)

// This will always fail

isThisIrrefutable = false;

else

// If body is irrefutable overall assertion is irrefutable

isThisIrrefutable = body->isThisIrrefutable;

isThisWillNotProgress = true;

isThisWillNotRegress = true;

isNotInLoop = parentNotInLoop;

isAtLeastOnce = parentAtLeastOnce;

isNotSpeculative = parentNotSpeculative;

isNotNegated = parentNotNegated;

}

void AssertionNode::AnnotatePass2(Compiler& compiler, CountDomain accumConsumes, bool accumPrevWillNotProgress, bool accumPrevWillNotRegress)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

prevConsumes = accumConsumes;

isPrevWillNotProgress = accumPrevWillNotProgress;

isPrevWillNotRegress = accumPrevWillNotRegress;

body->AnnotatePass2(compiler, accumConsumes, accumPrevWillNotProgress, accumPrevWillNotRegress);

}

void AssertionNode::AnnotatePass3(Compiler& compiler, CountDomain accumConsumes, CharSet<Char>\* accumFollow, bool accumFollowIrrefutable, bool accumFollowEOL)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

followConsumes = accumConsumes;

followSet = accumFollow;

isFollowIrrefutable = accumFollowIrrefutable;

isFollowEOL = accumFollowEOL;

// Can't say anything about what the assertion body will see at its end

CountDomain innerConsumes;

CharSet<Char>\* innerFollow = compiler.standardChars->GetFullSet();

// We can never backtrack into the body of an assertion (the continuation stack is cut)

body->AnnotatePass3(compiler, innerConsumes, innerFollow, true, false);

hasInitialHardFailBOI = body->hasInitialHardFailBOI;

}

void AssertionNode::AnnotatePass4(Compiler& compiler)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

body->AnnotatePass4(compiler);

// Even if body is non-deterministic we cut the choicepoints on exit from the assertion,

// so overall assertion is deterministic.

isDeterministic = true;

//

// Compilation scheme: Fail

//

// If body is irrefutable, assertion will always fail (and will leave groups empty).

//

if (isNegation && body->isThisIrrefutable)

{

// \*\*\*COMMIT\*\*\*

scheme = Fail;

return;

}

//

// Compilation scheme: Succ

//

// If body is irrefutable, assertion will always succeed. If it does not define groups

// we can eliminate it altogether.

//

if (!isNegation && body->isThisIrrefutable && !body->ContainsDefineGroup())

{

// \*\*COMMIT\*\*

scheme = Succ;

return;

}

//

// Compilation scheme: BeginEnd

//

scheme = BeginEnd;

}

bool AssertionNode::SupportsPrefixSkipping(Compiler& compiler) const

{

return false;

}

Node\* AssertionNode::HeadSyncronizingNode(Compiler& compiler)

{

return 0;

}

CharCount AssertionNode::MinSyncronizingLiteralLength(Compiler& compiler, int& numLiterals) const

{

return 0;

}

void AssertionNode::CollectSyncronizingLiterals(Compiler& compiler, ScannersMixin& scanners) const

{

Assert(false);

}

void AssertionNode::BestSyncronizingNode(Compiler& compiler, Node\*& bestNode)

{

}

void AssertionNode::AccumDefineGroups(Js::ScriptContext\* scriptContext, int& minGroup, int& maxGroup)

{

PROBE\_STACK(scriptContext, Js::Constants::MinStackRegex);

body->AccumDefineGroups(scriptContext, minGroup, maxGroup);

}

void AssertionNode::Emit(Compiler& compiler, CharCount& skipped)

{

PROBE\_STACK(compiler.scriptContext, Js::Constants::MinStackRegex);

Assert(skipped == 0);

switch (scheme)

{

case BeginEnd:

{

//

// Compilation scheme:

//

// BeginAssertion Lexit

// <body>

// EndAssertion

// Lexit:

//

int minBodyGroupId = compiler.program->numGroups;

int maxBodyGroupId = -1;

body->AccumDefineGroups(compiler.scriptContext, minBodyGroupId, maxBodyGroupId);

Label fixup = compiler.GetFixup(&EMIT(compiler, BeginAssertionInst, isNegation, minBodyGroupId, maxBodyGroupId)->nextLabel);

body->Emit(compiler, skipped);

EMIT(compiler, EndAssertionInst);

compiler.DoFixup(fixup, compiler.CurrentLabel());

break;

}

case Succ:

{

// Nothing to emit

break;

}

case Fail:

{

//

// Compilation scheme:

//

// Fail

//

EMIT(compiler, FailInst);

break;

}

}

}

CharCount AssertionNode::EmitScan(Compiler& compiler, bool isHeadSyncronizingNode)

{

Assert(false);

return 0;

}

bool AssertionNode::IsOctoquad(Compiler& compiler, OctoquadIdentifier\* oi)

{

return false;

}

bool AssertionNode::IsCharTrieArm(Compiler& compiler, uint& accNumAlts) const

{

return false;

}

bool AssertionNode::BuildCharTrie(Compiler& compiler, CharTrie\* trie, Node\* cont, bool isAcceptFirst) const

{

Assert(false);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void AssertionNode::Print(DebugWriter\* w, const Char\* litbuf) const

{

w->PrintEOL(L"Assertion(%s)", isNegation ? L"negative" : L"positive");

PrintAnnotations(w);

w->PrintEOL(L"{");

w->Indent();

body->Print(w, litbuf);

w->Unindent();

w->PrintEOL(L"}");

}

#endif

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

// Compiler

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

Compiler::Compiler

( Js::ScriptContext\* scriptContext

, ArenaAllocator\* ctAllocator

, ArenaAllocator\* rtAllocator

, StandardChars<Char>\* standardChars

, Program\* program

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, DebugWriter\* w

, RegexStats\* stats

#endif

)

: scriptContext(scriptContext)

, ctAllocator(ctAllocator)

, rtAllocator(rtAllocator)

, standardChars(standardChars)

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, w(w)

, stats(stats)

#endif

, program(program)

, instBuf(0)

, instLen(0)

, instNext(0)

, nextLoopId(0)

{}

void Compiler::CaptureNoLiterals(Program\* program)

{

program->rep.insts.litbuf = 0;

program->rep.insts.litbufLen = 0;

}

void Compiler::CaptureLiterals(Node\* root, const Char\* litbuf)

{

// Program will own literal buffer. Prepare buffer and nodes for case-invariant matching if necessary.

CharCount finalLen = root->TransferPass0(\*this, litbuf);

program->rep.insts.litbuf = finalLen == 0 ? 0 : RecyclerNewArrayLeaf(scriptContext->GetRecycler(), Char, finalLen);

program->rep.insts.litbufLen = 0;

root->TransferPass1(\*this, litbuf);

Assert(program->rep.insts.litbufLen == finalLen);

}

void Compiler::EmitAndCaptureSuccInst(Recycler\* recycler, Program\* program)

{

program->rep.insts.insts = (uint8\*)RecyclerNewLeaf(recycler, SuccInst);

program->rep.insts.instsLen = sizeof(SuccInst);

program->numLoops = 0;

}

void Compiler::CaptureInsts()

{

program->rep.insts.insts = RecyclerNewArrayLeaf(scriptContext->GetRecycler(), uint8, instNext);

program->rep.insts.instsLen = instNext;

memcpy\_s(program->rep.insts.insts, instNext, instBuf, instNext);

program->numLoops = nextLoopId;

}

void Compiler::FreeBody()

{

if (instBuf != 0)

{

ctAllocator->Free(instBuf, instLen);

instBuf = 0;

instLen = 0;

instNext = 0;

}

}

void Compiler::CompileEmptyRegex

( Program\* program

, RegexPattern\* pattern

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, DebugWriter\* w

, RegexStats\* stats

#endif

)

{

program->tag = Program::InstructionsTag;

CaptureNoLiterals(program);

EmitAndCaptureSuccInst(pattern->GetScriptContext()->GetRecycler(), program);

}

void Compiler::Compile

( Js::ScriptContext\* scriptContext

, ArenaAllocator\* ctAllocator

, ArenaAllocator\* rtAllocator

, StandardChars<Char>\* standardChars

, Program \*program

, Node\* root

, const Char\* litbuf

, RegexPattern\* pattern

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, DebugWriter\* w

, RegexStats\* stats

#endif

)

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

if (w != 0)

{

w->PrintEOL(L"REGEX AST /%s/ {", program->source);

w->Indent();

root->Print(w, litbuf);

w->Unindent();

w->PrintEOL(L"}");

w->Flush();

}

#endif

Compiler compiler

( scriptContext

, ctAllocator

, rtAllocator

, standardChars

, program

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, w

, stats

#endif

);

bool compiled = false;

if (REGEX\_CONFIG\_FLAG(RegexOptimize))

{

// SPECIAL CASE: Octoquad/trigrams

// (must handle before converting to case-insensitive form since the later interferes with octoquad pattern recognizer)

if (OctoquadIdentifier::Qualifies(program))

{

int numCodes;

char localCodeToChar[TrigramAlphabet::AlphaCount];

char localCharToCode[TrigramAlphabet::AsciiTableSize];

char (\*codeToChar)[TrigramAlphabet::AlphaCount];

char (\*charToCode)[TrigramAlphabet::AsciiTableSize];

TrigramAlphabet \*trigramAlphabet = scriptContext->GetTrigramAlphabet();

if(trigramAlphabet)

{

numCodes = TrigramAlphabet::AlphaCount;

codeToChar = &trigramAlphabet->alpha;

charToCode = &trigramAlphabet->alphaBits;

}

else

{

numCodes = 0;

codeToChar = &localCodeToChar;

charToCode = &localCharToCode;

}

OctoquadIdentifier oi(numCodes, \*codeToChar, \*charToCode);

// We haven't captured literals yet: temporarily set the program's litbuf to be the parser's litbuf

Assert(program->rep.insts.litbuf == 0);

program->rep.insts.litbuf = (Char\*)litbuf;

if (root->IsOctoquad(compiler, &oi) && oi.IsOctoquad())

{

program->rep.insts.litbuf = 0;

oi.InitializeTrigramInfo(scriptContext, pattern);

program->tag = Program::OctoquadTag;

program->rep.octoquad.matcher = OctoquadMatcher::New(scriptContext->GetRecycler(), standardChars, program->GetCaseMappingSource(), &oi);

compiled = true;

}

else

program->rep.insts.litbuf = 0;

}

}

if (!compiled)

{

if (REGEX\_CONFIG\_FLAG(RegexOptimize))

{

Char c;

if (root->IsSingleChar(compiler, c))

{

// SPECIAL CASE: c

program->tag = Program::SingleCharTag;

program->rep.singleChar.c = c;

}

else if (root->IsBoundedWord(compiler))

{

// SPECIAL CASE: \b\w+\b

program->tag = Program::BoundedWordTag;

}

else if (root->IsLeadingTrailingSpaces(compiler,

program->rep.leadingTrailingSpaces.beginMinMatch,

program->rep.leadingTrailingSpaces.endMinMatch))

{

// SPECIAL CASE: ^\s\*|\s\*$

program->tag = Program::LeadingTrailingSpacesTag;

}

else if (root->IsBOILiteral2(compiler))

{

program->tag = Program::BOILiteral2Tag;

program->rep.boiLiteral2.literal = \*(DWORD \*)litbuf;

}

else

{

program->tag = Program::InstructionsTag;

compiler.CaptureLiterals(root, litbuf);

root->AnnotatePass0(compiler);

root->AnnotatePass1(compiler, true, true, true, true);

// Nothing comes before or after overall pattern

CountDomain consumes(0);

// Match could progress from lhs (since we try successive start positions), but can never regress

root->AnnotatePass2(compiler, consumes, false, true);

// Anything could follow an end of pattern match

CharSet<Char>\* follow = standardChars->GetFullSet();

root->AnnotatePass3(compiler, consumes, follow, true, false);

root->AnnotatePass4(compiler);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

if (w != 0)

{

w->PrintEOL(L"REGEX ANNOTATED AST /%s/ {", program->source);

w->Indent();

root->Print(w, program->rep.insts.litbuf);

w->Unindent();

w->PrintEOL(L"}");

w->Flush();

}

#endif

CharCount skipped = 0;

// If the root Node has a hard fail BOI, we should not emit any synchronize Nodes

// since we can easily just search from the beginning.

if (root->hasInitialHardFailBOI == false)

{

// If the root Node doesn't have hard fail BOI but sticky flag is present don't synchronize Nodes

// since we can easily just search from the beginning. Instead set to special InstructionTag

if ((program->flags & StickyRegexFlag) != 0)

{

compiler.SetBOIInstructionsProgramForStickyFlagTag();

}

else

{

Node\* bestSyncronizingNode = 0;

root->BestSyncronizingNode(compiler, bestSyncronizingNode);

Node\* headSyncronizingNode = root->HeadSyncronizingNode(compiler);

if ((bestSyncronizingNode == 0 && headSyncronizingNode != 0) ||

(bestSyncronizingNode != 0 && headSyncronizingNode == bestSyncronizingNode))

{

// Scan and consume the head, continue with rest assuming head has been consumed

skipped = headSyncronizingNode->EmitScan(compiler, true);

}

else if (bestSyncronizingNode != 0)

{

// Scan for the synchronizing node, then backup ready for entire pattern

skipped = bestSyncronizingNode->EmitScan(compiler, false);

Assert(skipped == 0);

// We're synchronizing to a non-head node; if we have to back up, then try to synchronize to a character

// in the first set before running the remaining instructions

if (!bestSyncronizingNode->prevConsumes.CouldMatchEmpty()) // must back up at least one character

skipped = root->EmitScanFirstSet(compiler);

}

else

{

// Optionally scan for a character in the overall pattern's FIRST set, possibly consume it,

// then match all or remainder of pattern

skipped = root->EmitScanFirstSet(compiler);

}

}

}

root->Emit(compiler, skipped);

compiler.Emit<SuccInst>();

compiler.CaptureInsts();

}

}

else

{

program->tag = Program::InstructionsTag;

compiler.CaptureLiterals(root, litbuf);

CharCount skipped = 0;

root->Emit(compiler, skipped);

compiler.Emit<SuccInst>();

compiler.CaptureInsts();

}

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

if (w != 0)

{

w->PrintEOL(L"REGEX PROGRAM /%s/ ", program->source);

program->Print(w);

w->Flush();

}

#endif

}

}

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

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

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

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

//

// Regex parsing and AST-to-AST transformations/analysis

//

#pragma once

namespace UnifiedRegex

{

// FORWARD

class Compiler;

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

// Node

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

struct Node : protected Chars<wchar\_t>

{

// Optimization heuristics

static const int maxSyncToSetSize = 256;

static const int preferredMinSyncToLiteralLength = 3;

static const int maxNumSyncLiterals = ScannersMixin::MaxNumSyncLiterals;

static const int minRemainLengthForTest = 4;

static const int maxCharsForConditionalTry = 20;

static const int maxTrieArmExpansion = 16;

enum NodeTag : uint16

{

// SimpleNode

Empty, // (|...), etc

BOL, // ^

EOL, // $

// WordBoundaryNode

WordBoundary, // \b, \B

// MatchLiteralNode

MatchLiteral, // abc, non-empty

// MatchCharNode

MatchChar, // a

// ConcatNode

Concat, // e e

// AltNode

Alt, // e | e

// DefineGroupNode

DefineGroup, // (e)

// MatchGroupNode

MatchGroup, // \1

// LoopNode

Loop, // e\*, e+, e{n,m}, e\*?, e+?, e{n,m}?

// MatchSetNode

MatchSet, // [...], [^...], \b, \B, \d, \D, \s, \S, \w, \W, .

// AssertionNode

Assertion // (?=e), (?!e)

};

enum Features : uint16

{

HasEmpty = 1 << Empty,

HasBOL = 1 << BOL,

HasEOL = 1 << EOL,

HasWordBoundary = 1 << WordBoundary,

HasMatchLiteral = 1 << MatchLiteral,

HasMatchChar = 1 << MatchChar,

HasConcat = 1 << Concat,

HasAlt = 1 << Alt,

HasDefineGroup = 1 << DefineGroup,

HasMatchGroup = 1 << MatchGroup,

HasLoop = 1 << Loop,

HasMatchSet = 1 << MatchSet,

HasAssertion = 1 << Assertion

};

NodeTag tag;

// Features of this and all child nodes

uint16 features;

// See comment for firstSet

bool isFirstExact : 1;

// True if pattern can never fail

bool isThisIrrefutable : 1;

// True if following patterns can never fail

bool isFollowIrrefutable : 1;

// True if pattern matches one or more word characters

bool isWord : 1;

// True if pattern will not consume more characters on backtracking

bool isThisWillNotProgress : 1;

// True if pattern will not consume fewer characters on backtracking

bool isThisWillNotRegress : 1;

// True if previous patterns will not consume more characters on backtracking

bool isPrevWillNotProgress : 1;

// True if previous patterns will not consume fewer characters on backtracking

bool isPrevWillNotRegress : 1;

// True if $ always follows pattern (and we are not in multi-line mode)

bool isFollowEOL : 1;

// True if pattern is deterministic (ie will never push a choicepoint during execution)

// Determined in pass 4.

bool isDeterministic : 1;

// True if pattern is not in a loop context

bool isNotInLoop : 1;

// True if pattern will be matched against at least one segment of the input (ie will be executed at least once)

bool isAtLeastOnce : 1;

// True if pattern does not appear in an assertion

bool isNotSpeculative : 1;

// True if known to not be in a negative assertion context

// (We do not play any games with double-negation)

bool isNotNegated : 1;

// True if this contains a hard fail BOI

bool hasInitialHardFailBOI : 1;

uint dummy : 17;

// NOTE: The bodies of the following sets are allocated in the compile-time allocator and must be cloned

// into the run-time allocator if they end up being used by an instruction.

// NOTE: Sets may be aliased between nodes, and may be one of the standard sets.

// Upper bound of FIRST characters of this pattern.

// - Pattern will \*never\* match first characters not in this set

// - If isFirstExact, pattern will \*always\* match first characters in this set (but may fail on later characters)

// - If !isFirstExact, pattern \*may\* match first characters in this set, or may fail.

CharSet<Char> \*firstSet;

// Upper bound of FOLLOW characters of this pattern.

CharSet<Char> \*followSet;

// Range of number of characters already consumed before this pattern

CountDomain prevConsumes;

// Range of number of characters consumed by this pattern

CountDomain thisConsumes;

// Range of number of character consumed after this pattern

CountDomain followConsumes;

inline Node(NodeTag tag)

: tag(tag)

, features(0)

, firstSet(0)

, isFirstExact(false)

, followSet(0)

, isThisIrrefutable(false)

, isFollowIrrefutable(false)

, isWord(false)

, isThisWillNotProgress(false)

, isThisWillNotRegress(false)

, isPrevWillNotProgress(false)

, isPrevWillNotRegress(false)

, isFollowEOL(false)

, isDeterministic(false)

, isNotInLoop(false)

, isAtLeastOnce(false)

, isNotSpeculative(false)

, isNotNegated(false)

, hasInitialHardFailBOI(false)

{

}

//

// Parse-time helpers

//

virtual CharCount LiteralLength() const = 0;

virtual void AppendLiteral(CharCount& litbufNext, CharCount litbufLen, \_\_inout\_ecount(litbufLen) Char\* litbuf) const;

virtual bool IsCharOrPositiveSet() const = 0;

// Transfer pass 0:

// - synthesize the total number of characters required to store all literals, including case-invariant

// expansions where required

// - adjust match char nodes to account for case invariance if necessary

virtual CharCount TransferPass0(Compiler& compiler, const Char\* litbuf) = 0;

// Transfer pass 1:

// - transfer literals from given litbuf into newLitbuf, advancing nextLit as we go

// - adjust set nodes to account for case invariance if necessary

virtual void TransferPass1(Compiler& compiler, const Char\* litbuf) = 0;

//

// Compile-time helpers

//

// True if firstSet of this node can be used as the followSet of a previous node, even though this node may

// accept empty. True only for simple assertions.

virtual bool IsRefiningAssertion(Compiler& compiler) = 0;

// Annotation pass 0:

// - bottom-up: isWord

// - refine WordBoundary nodes where possible

virtual void AnnotatePass0(Compiler& compiler) = 0;

// Annotation pass 1:

// - top-down: isNotInLoop, isAtLeastOnce, isNotSpeculative, isNotNegated

// - bottom-up: features, thisConsumes, firstSet, isFirstExact, isThisIrrefutable, isThisWillNotProgress, isThisWillNotRegress

virtual void AnnotatePass1(Compiler& compiler, bool parentNotInLoop, bool parentAtLeastOnce, bool parentNotSpeculative, bool parentNotNegated) = 0;

// Annotation pass 2

// - left-to-right: prevConsumes, isPrevWillNotProgress, isPrevWillNotRegress.

virtual void AnnotatePass2(Compiler& compiler, CountDomain accumConsumes, bool accumPrevWillNotProgress, bool accumPrevWillNotRegress) = 0;

// Annotation pass 3

// - right-to-left: followConsumes, followSet, isFollowIrrefutable, isFollowEOL

virtual void AnnotatePass3(Compiler& compiler, CountDomain accumConsumes, CharSet<Char>\* accumFollow, bool accumFollowIrrefutable, bool accumFollowEOL) = 0;

// Annotation pass 4

// - possibly simplify the node in-place

// - decide on the compilation scheme for each node, possibly recording it within node-specific fields

// - bottom-up: isDeterministic

virtual void AnnotatePass4(Compiler& compiler) = 0;

// Return true if pattern can be complied assuming some fixed-length prefix of a matching input string has already been consumed

virtual bool SupportsPrefixSkipping(Compiler& compiler) const = 0;

// Return the Match(Char|Literal|Set) at the start of pattern, or 0 if no such unique node

virtual Node\* HeadSyncronizingNode(Compiler& compiler) = 0;

// Count how many literals are in pattern and return their minimum length. Returns 0

// if pattern not in a form which can be used by a SyncToLiterals instruction.

virtual CharCount MinSyncronizingLiteralLength(Compiler& compiler, int& numLiterals) const = 0;

// Collect the literals counted by above and build scanners for them.

virtual void CollectSyncronizingLiterals(Compiler& compiler, ScannersMixin& scanners) const = 0;

// Find a MatchLiteral or Alt of MatchLiterals which must appear at least once in input string for pattern

// to match, and which has the shortest prevConsumes.

virtual void BestSyncronizingNode(Compiler& compiler, Node\*& bestNode) = 0;

// Accumulate the range of groups definitions in pattern.

// NOTE: minGroup must be > largest group, and maxGroup must be < 0 on topmost call

virtual void AccumDefineGroups(Js::ScriptContext\* scriptContext, int& minGroup, int& maxGroup) = 0;

// Emit code to consume this pattern. The first skipped characters of pattern have been consumed by context.

virtual void Emit(Compiler& compiler, CharCount& skipped) = 0;

// Emit code to scan forward for the first occurence of pattern, or hard fail if no such occurence.

// - if isHeadSyncronizingNode, also consume the occurence and leave input pointer at first char after it

// - otherwise, leave input pointer at the latest point of input which could match the overall pattern

// (ie rewind from start of occurence accerding to the prevConsumes range)

// - may actually do nothing if nothing worthwhile to scan to

// Return number of characters consumed.

virtual CharCount EmitScan(Compiler& compiler, bool isHeadSyncronizingNode) = 0;

CharCount EmitScanFirstSet(Compiler& compiler);

inline bool IsObviouslyDeterministic() { return (features & (HasAlt | HasLoop)) == 0; }

inline bool ContainsAssertion() { return (features & (HasBOL | HasEOL | HasWordBoundary | HasAssertion)) != 0; }

inline bool ContainsDefineGroup() { return (features & HasDefineGroup) != 0; }

inline bool ContainsMatchGroup() { return (features & HasMatchGroup) != 0; }

inline bool IsSimple() { return !ContainsAssertion() && !ContainsDefineGroup(); }

inline bool IsSimpleOneChar() { return IsSimple() && !isThisIrrefutable && isFirstExact && thisConsumes.IsExact(1); }

inline bool IsEmptyOnly() { return IsSimple() && isThisIrrefutable && thisConsumes.IsExact(0); }

static bool IsBetterSyncronizingNode(Compiler& compiler, Node\* curr, Node\* proposed);

//

// Recognizers

//

// Is regex c

bool IsSingleChar(Compiler& compiler, Char& outChar) const;

// Is regex \b\w+\b?

bool IsBoundedWord(Compiler& compiler) const;

// Is regex ^\s\*|\s\*$

bool IsLeadingTrailingSpaces(Compiler& compiler, CharCount& leftMinMatch, CharCount& rightMinMatch) const;

// Is regex ^literal

bool IsBOILiteral2(Compiler& compiler) const;

// Can this regex be recognized by an Octoquad/Megamatch matcher? Ie is in grammar:

// octoquad ::= atom{8} '|' atom{8}

// atom ::= A | '['...charset drawn from A's...']'

// and A is a set of exactly four ASCII characters

virtual bool IsOctoquad(Compiler& compiler, OctoquadIdentifier\* oi) = 0;

// Can this regex be recognized by a CharTrie structure? Ie is in grammar:

// triearm ::= atom\*

// atom ::= c | '[' ... ']'

// and factoring out sets does not exceed arm limit

virtual bool IsCharTrieArm(Compiler& compiler, uint& accNumAlts) const = 0;

// Assuming above returned true, expand 'trie' node to include all literals recognized in this regex, and

// continue expanding from each leaf using given 'cont' regex. Return false if any trie node has too many

// children.

// - If isAcceptFirst is true, ignore any literals which are proper extensions of a literal already in

// the trie, but return false if any later literal is a prefix of an earlier literal.

// (If the follow of the alt we are turning into a trie is irrefutable, we can simply stop at the

// first shortest match).

// - Otherwise, return false if any literal is a proper prefix of any other literal, irrespective of order.

virtual bool BuildCharTrie(Compiler& compiler, CharTrie\* trie, Node\* cont, bool isAcceptFirst) const = 0;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

virtual void Print(DebugWriter\* w, const Char\* litbuf) const = 0;

void PrintAnnotations(DebugWriter\* w) const;

#endif

};

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

#define NODE\_PRINT void Print(DebugWriter\* w, const Char\* litbuf) const override;

#else

#define NODE\_PRINT

#endif

#define NODE\_DECL CharCount LiteralLength() const override; \

bool IsCharOrPositiveSet() const override; \

CharCount TransferPass0(Compiler& compiler, const Char\* litbuf) override; \

void TransferPass1(Compiler& compiler, const Char\* litbuf) override; \

bool IsRefiningAssertion(Compiler& compiler) override; \

void AnnotatePass0(Compiler& compiler) override; \

void AnnotatePass1(Compiler& compiler, bool parentNotInLoop, bool parentAtLeastOnce, bool parentNotSpeculative, bool parentNotNegated) override; \

void AnnotatePass2(Compiler& compiler, CountDomain accumConsumes, bool accumPrevWillNotProgress, bool accumPrevWillNotRegress) override; \

void AnnotatePass3(Compiler& compiler, CountDomain accumConsumes, CharSet<Char>\* accumFollow, bool accumFollowIrrefutable, bool accumFollowEOL) override; \

void AnnotatePass4(Compiler& compiler) override; \

bool SupportsPrefixSkipping(Compiler& compiler) const override; \

Node\* HeadSyncronizingNode(Compiler& compiler) override; \

CharCount MinSyncronizingLiteralLength(Compiler& compiler, int& numLiterals) const override; \

void CollectSyncronizingLiterals(Compiler& compiler, ScannersMixin& scanners) const override; \

void BestSyncronizingNode(Compiler& compiler, Node\*& bestNode) override; \

void Emit(Compiler& compiler, CharCount& skipped) override; \

CharCount EmitScan(Compiler& compiler, bool isHeadSyncronizingNode) override; \

void AccumDefineGroups(Js::ScriptContext\* scriptContext, int& minGroup, int& maxGroup) override; \

bool IsOctoquad(Compiler& compiler, OctoquadIdentifier\* oi) override; \

bool IsCharTrieArm(Compiler& compiler, uint& accNumAlts) const override; \

bool BuildCharTrie(Compiler& compiler, CharTrie\* trie, Node\* cont, bool isAcceptFirst) const override; \

NODE\_PRINT

struct SimpleNode : Node

{

inline SimpleNode(NodeTag tag)

: Node(tag)

{

}

NODE\_DECL

};

struct WordBoundaryNode : Node

{

bool isNegation;

bool mustIncludeEntering; // isNegation => false

bool mustIncludeLeaving; // isNegation => false

inline WordBoundaryNode(bool isNegation)

: Node(WordBoundary)

, isNegation(isNegation)

, mustIncludeEntering(false)

, mustIncludeLeaving(false)

{

}

NODE\_DECL

};

struct MatchLiteralNode : Node

{

CharCount offset; // into literal buffer (initially parser's, then program's)

CharCount length; // always > 1

bool isEquivClass; // True if each consecutive triplet of characters of literal represents equivalence

// class of characters to match against. Actual literal length will be 3 times the length

// given above

inline MatchLiteralNode(CharCount offset, CharCount length)

: Node(MatchLiteral)

, offset(offset)

, length(length)

, isEquivClass(false)

{

}

NODE\_DECL

void AppendLiteral(CharCount& litbufNext, CharCount litbufLen, \_\_inout\_ecount(litbufLen) Char\* litbuf) const override;

};

struct MatchCharNode : Node

{

Char cs[CaseInsensitive::EquivClassSize];

bool isEquivClass; // true if above characters represent equivalence class of characters, otherwise only

// first character is significant

inline MatchCharNode(Char c)

: Node(MatchChar)

, isEquivClass(false)

{

cs[0] = c;

#if DBG

for (int i = 1; i < CaseInsensitive::EquivClassSize; i++)

cs[i] = (Char)-1;

#endif

}

NODE\_DECL

void AppendLiteral(CharCount& litbufNext, CharCount litbufLen, \_\_inout\_ecount(litbufLen) Char\* litbuf) const override;

CompileAssert(CaseInsensitive::EquivClassSize == 4);

static void Emit(Compiler& compiler, \_\_in\_ecount(4) Char \* cs, bool isEquivClass);

private:

CompileAssert(CaseInsensitive::EquivClassSize == 4);

static CharCount FindUniqueEquivs(

const Char equivs[CaseInsensitive::EquivClassSize],

\_\_out\_ecount(4) Char uniqueEquivs[CaseInsensitive::EquivClassSize]);

};

struct MatchSetNode : Node

{

bool isNegation;

bool needsEquivClass;

CharSet<Char> set; // contents always owned by compile-time allocator

// Set must be filled in

inline MatchSetNode(bool isNegation, bool needsEquivClass = true)

: Node(MatchSet)

, isNegation(isNegation)

, needsEquivClass(true)

{

}

NODE\_DECL

};

struct ConcatNode : Node

{

Node\* head; // never a concat node

ConcatNode\* tail; // null for end, overall always length > 1, never consecutive literals/chars

inline ConcatNode(Node\* head, ConcatNode\* tail)

: Node(Concat)

, head(head)

, tail(tail)

{

}

NODE\_DECL

};

struct AltNode sealed : Node

{

Node\* head; // never an alt node

AltNode\* tail; // null for end, overall always length > 1, never consecutive chars/sets

enum CompilationScheme

{

Try, // Push choicepoint, try item, backtrack to next item on failure

None, // No items (deterministic)

Trie, // Match using trie of literals (deterministic)

Switch, // Switch using 1 char lookahead (deterministic)

Chain, // Chain of JumpIfNot(Char|Set) using 1 char lookahead (deterministic)

Set // (Opt?)Match(Char|Set) (deterministic)

};

// Following determined in pass 4

RuntimeCharTrie\* runtimeTrie; // significant only if scheme == Trie

CompilationScheme scheme;

bool isOptional; // significant only if scheme == None|Switch|Chain|Set

int switchSize; // significant only if scheme == Switch

inline AltNode(Node\* head, AltNode\* tail)

: Node(Alt)

, head(head)

, tail(tail)

, scheme(Try)

, runtimeTrie(0)

, isOptional(false)

, switchSize(0)

{

}

NODE\_DECL

};

struct DefineGroupNode : Node

{

Node\* body;

int groupId;

enum CompilationScheme

{

Chomp, // Chomp matching characters and capture all into a group at the end

Fixed, // Capture fixed-length group at end

BeginEnd // Wrap by begin/end instructions

};

// Following determined in pass 4

CompilationScheme scheme;

bool noNeedToSave;

inline DefineGroupNode(int groupId, Node\* body)

: Node(DefineGroup)

, groupId(groupId)

, body(body)

, scheme(BeginEnd)

, noNeedToSave(false)

{

}

NODE\_DECL

};

struct MatchGroupNode : Node

{

int groupId;

inline MatchGroupNode(int groupId)

: Node(MatchGroup)

, groupId(groupId)

{

}

NODE\_DECL

};

struct LoopNode : Node

{

Node\* body;

CountDomain repeats;

bool isGreedy;

enum CompilationScheme

{

BeginEnd, // Push choicepoints for each unravelling (greedy) or deferred unravelling (non-greedy) of body

None, // Loop matches empty only (deterministic)

Chomp, // Chomp matching characters (deterministic)

ChompGroupLastChar, // Chomp matching characters and bind the last char to group (deterministic)

Guarded, // Use 1 char lookahead to control loop repeats (deterministic)

Fixed, // Loop body is non-zero fixed width, deterministic, group-free

FixedSet, // Loop body is MatchSet

FixedGroupLastIteration, // Loop body is non-zero fixed width, deterministic, loop body has one outer group

GreedyNoBacktrack, // Can keep track of backtracking info in constant space (deterministic)

Set, // Treat r? as r|<empty>, emit as for AltNode::Set

Chain, // Treat r? as r|<empty>, emit as for AltNode::Chain

Try // Treat r? as r|<empty>, emit as for AltNode::Try

};

// Following determined in pass 4

bool noNeedToSave; // defined for ChompGroupLastChar/FixedGroupLastIteration only

CompilationScheme scheme;

inline LoopNode(CharCount lower, CharCountOrFlag upper, bool isGreedy, Node\* body)

: Node(Loop)

, repeats(lower, upper)

, isGreedy(isGreedy)

, body(body)

, scheme(BeginEnd)

{

}

NODE\_DECL

};

struct AssertionNode : Node

{

Node\* body;

bool isNegation;

enum CompilationScheme

{

BeginEnd, // Protect assertion with begin/end instructions

Succ, // Assertion will always succeeed, without binding groups

Fail // Assertion will always fail

};

// Following determined in pass 4

CompilationScheme scheme;

inline AssertionNode(bool isNegation, Node\* body)

: Node(Assertion)

, isNegation(isNegation)

, body(body)

, scheme(BeginEnd)

{

}

NODE\_DECL

};

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

// Compiler

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

class Compiler : private Chars<wchar\_t>

{

friend Node;

friend SimpleNode;

friend WordBoundaryNode;

friend MatchLiteralNode;

friend MatchCharNode;

friend ConcatNode;

friend AltNode;

friend DefineGroupNode;

friend MatchGroupNode;

friend LoopNode;

friend MatchSetNode;

friend AssertionNode;

private:

static const CharCount initInstBufSize = 128;

Js::ScriptContext\* scriptContext;

// Arena for nodes and items needed only during compliation

ArenaAllocator\* ctAllocator;

// Arena for literals, sets and items needed during runtime

ArenaAllocator\* rtAllocator;

StandardChars<Char>\* standardChars;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

DebugWriter\* w;

RegexStats\* stats;

#endif

Program\* program;

uint8\* instBuf; // in compile-time allocator, owned by compiler

CharCount instLen; // size of instBuf in bytes

CharCount instNext; // offset to emit next instruction into

int nextLoopId;

private:

uint8\* Emit(size\_t size);

template <typename T>

T\* Emit();

// The instruction buffer may move, so we need to remember label fixup's relative to the instruction base

// rather than as machine addresses

inline Label Compiler::GetFixup(Label\* pLabel)

{

Assert((uint8\*)pLabel >= instBuf && (uint8\*)pLabel < instBuf + instNext);

return (Label)((uint8\*)pLabel - instBuf);

}

inline void Compiler::DoFixup(Label fixup, Label label)

{

Assert(fixup < instNext);

Assert(label <= instNext);

\*(Label\*)(instBuf + fixup) = label;

}

inline Label Compiler::CurrentLabel()

{

return instNext;

}

template <typename T>

inline T\* LabelToInstPointer(Inst::InstTag tag, Label label)

{

Assert(label + sizeof(T) <= instNext);

Assert(((Inst\*)(instBuf + label))->tag == tag);

return (T\*)(instBuf + label);

}

inline int Compiler::NextLoopId()

{

return nextLoopId++;

}

inline Js::ScriptContext \*GetScriptContext() const

{

return scriptContext;

}

inline Program \*GetProgram() const

{

return program;

}

void SetBOIInstructionsProgramTag()

{

Assert(this->program->tag == Program::InstructionsTag

|| this->program->tag == Program::BOIInstructionsTag);

Assert(this->CurrentLabel() == 0);

this->program->tag = Program::BOIInstructionsTag;

}

void SetBOIInstructionsProgramForStickyFlagTag()

{

Assert(this->program->tag == Program::InstructionsTag

|| this->program->tag == Program::BOIInstructionsForStickyFlagTag);

Assert(this->CurrentLabel() == 0);

AssertMsg((this->program->flags & StickyRegexFlag) != 0, "Shouldn't set BOIInstructionsForStickyFlagTag, if sticky is false.");

this->program->tag = Program::BOIInstructionsForStickyFlagTag;

}

static void CaptureNoLiterals(Program\* program);

void CaptureLiterals(Node\* root, const Char \*litbuf);

static void EmitAndCaptureSuccInst(Recycler\* recycler, Program\* program);

void CaptureInsts();

void FreeBody();

Compiler

( Js::ScriptContext\* scriptContext

, ArenaAllocator\* ctAllocator

, ArenaAllocator\* rtAllocator

, StandardChars<Char>\* standardChars

, Program\* program

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, DebugWriter\* w

, RegexStats\* stats

#endif

);

public:

static void CompileEmptyRegex

( Program\* program

, RegexPattern\* pattern

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, DebugWriter\* w

, RegexStats\* stats

#endif

);

static void Compile

( Js::ScriptContext\* scriptContext

, ArenaAllocator\* ctAllocator

, ArenaAllocator\* rtAllocator

, StandardChars<Char>\* standardChars

, Program\* program

, Node\* root

, const Char\* litbuf

, RegexPattern\* pattern

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, DebugWriter\* w

, RegexStats\* stats

#endif

);

};

}

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

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

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

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

M(Resume)

M(RestoreLoop)

M(RestoreGroup)

M(ResetGroup)

M(ResetGroupRange)

M(RepeatLoop)

M(PopAssertion)

M(RewindLoopFixed)

M(RewindLoopSet)

M(RewindLoopFixedGroupLastIteration)

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

// 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 UnifiedRegex

{

enum RegexFlags : uint8

{

NoRegexFlags = 0,

IgnoreCaseRegexFlag = 1 << 0,

GlobalRegexFlag = 1 << 1,

MultilineRegexFlag = 1 << 2,

UnicodeRegexFlag = 1 << 3,

StickyRegexFlag = 1 << 4,

AllRegexFlags = (1 << 5) - 1

};

}

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

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

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

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

// M(TagName)

// ClassName == TagName##Inst

// MTemplate(TagName, TemplateDeclaration, GenericClassName, SpecializedClassName)

M(Fail)

M(Succ)

M(Jump)

M(JumpIfNotChar)

M(MatchCharOrJump)

M(JumpIfNotSet)

M(MatchSetOrJump)

M(Switch10)

M(Switch20)

M(SwitchAndConsume10)

M(SwitchAndConsume20)

M(BOITest)

M(EOITest)

M(BOLTest)

M(EOLTest)

M(WordBoundaryTest)

M(MatchChar)

M(MatchChar2)

M(MatchChar3)

M(MatchChar4)

MTemplate(MatchSet, template<bool IsNegation>, MatchSetInst, MatchSetInst<false>)

MTemplate(MatchNegatedSet, template<bool IsNegation>, MatchSetInst, MatchSetInst<true>)

M(MatchLiteral)

M(MatchLiteralEquiv)

M(MatchTrie)

M(OptMatchChar)

M(OptMatchSet)

M(SyncToCharAndContinue)

M(SyncToChar2SetAndContinue)

MTemplate(SyncToSetAndContinue, template<bool IsNegation>, SyncToSetAndContinueInst, SyncToSetAndContinueInst<false>)

MTemplate(SyncToNegatedSetAndContinue, template<bool IsNegation>, SyncToSetAndContinueInst, SyncToSetAndContinueInst<true>)

M(SyncToChar2LiteralAndContinue)

M(SyncToLiteralAndContinue)

M(SyncToLinearLiteralAndContinue)

M(SyncToLiteralEquivAndContinue)

M(SyncToLiteralEquivTrivialLastPatCharAndContinue)

M(SyncToCharAndConsume)

M(SyncToChar2SetAndConsume)

MTemplate(SyncToSetAndConsume, template<bool IsNegation>, SyncToSetAndConsumeInst, SyncToSetAndConsumeInst<false>)

MTemplate(SyncToNegatedSetAndConsume, template<bool IsNegation>, SyncToSetAndConsumeInst, SyncToSetAndConsumeInst<true>)

M(SyncToChar2LiteralAndConsume)

M(SyncToLiteralAndConsume)

M(SyncToLinearLiteralAndConsume)

M(SyncToLiteralEquivAndConsume)

M(SyncToLiteralEquivTrivialLastPatCharAndConsume)

M(SyncToCharAndBackup)

MTemplate(SyncToSetAndBackup, template<bool IsNegation>, SyncToSetAndBackupInst, SyncToSetAndBackupInst<false>)

MTemplate(SyncToNegatedSetAndBackup, template<bool IsNegation>, SyncToSetAndBackupInst, SyncToSetAndBackupInst<true>)

M(SyncToChar2LiteralAndBackup)

M(SyncToLiteralAndBackup)

M(SyncToLinearLiteralAndBackup)

M(SyncToLiteralEquivAndBackup)

M(SyncToLiteralEquivTrivialLastPatCharAndBackup)

M(SyncToLiteralsAndBackup)

M(MatchGroup)

M(BeginDefineGroup)

M(EndDefineGroup)

M(DefineGroupFixed)

M(BeginLoop)

M(RepeatLoop)

M(BeginLoopIfChar)

M(BeginLoopIfSet)

M(RepeatLoopIfChar)

M(RepeatLoopIfSet)

M(BeginLoopFixed)

M(RepeatLoopFixed)

M(LoopSet)

M(BeginLoopFixedGroupLastIteration)

M(RepeatLoopFixedGroupLastIteration)

M(BeginGreedyLoopNoBacktrack)

M(RepeatGreedyLoopNoBacktrack)

MTemplate(ChompCharStar, template<ChompMode Mode>, ChompCharInst, ChompCharInst<ChompMode::Star>)

MTemplate(ChompCharPlus, template<ChompMode Mode>, ChompCharInst, ChompCharInst<ChompMode::Plus>)

MTemplate(ChompSetStar, template<ChompMode Mode>, ChompSetInst, ChompSetInst<ChompMode::Star>)

MTemplate(ChompSetPlus, template<ChompMode Mode>, ChompSetInst, ChompSetInst<ChompMode::Plus>)

MTemplate(ChompCharGroupStar, template<ChompMode Mode>, ChompCharGroupInst, ChompCharGroupInst<ChompMode::Star>)

MTemplate(ChompCharGroupPlus, template<ChompMode Mode>, ChompCharGroupInst, ChompCharGroupInst<ChompMode::Plus>)

MTemplate(ChompSetGroupStar, template<ChompMode Mode>, ChompSetGroupInst, ChompSetGroupInst<ChompMode::Star>)

MTemplate(ChompSetGroupPlus, template<ChompMode Mode>, ChompSetGroupInst, ChompSetGroupInst<ChompMode::Plus>)

M(ChompCharBounded)

M(ChompSetBounded)

M(ChompSetBoundedGroupLastChar)

M(Try)

M(TryIfChar)

M(TryMatchChar)

M(TryIfSet)

M(TryMatchSet)

M(BeginAssertion)

M(EndAssertion)

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

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

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

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

/\*

--------------------------------------------------------------------------------------------------------------------------------

Original

--------------------------------------------------------------------------------------------------------------------------------

Pattern ::= Disjunction

Disjunction ::= Alternative | Alternative '|' Disjunction

Alternative ::= [empty] | Alternative Term

Term ::= Assertion | Atom | Atom Quantifier

Assertion ::= '^' | '$' | '\' 'b' | '\' 'B' | '(' '?' '=' Disjunction ')' | '(' '?' '!' Disjunction ')'

Quantifier ::= QuantifierPrefix | QuantifierPrefix '?'

QuantifierPrefix ::= '\*' | '+' | '?' | '{' DecimalDigits '}' | '{' DecimalDigits ',' '}' | '{' DecimalDigits ',' DecimalDigits '}'

Atom ::= PatternCharacter | '.' | '\' AtomEscape | CharacterClass | '(' Disjunction ')' | '(' '?' ':' Disjunction ')'

PatternCharacter ::= SourceCharacter but not any of { '^', '$', '\', '.', '\*', '+', '?', '(', ')', '[', ']', '{', '}', '|' }

AtomEscape ::= DecimalEscape | CharacterEscape | CharacterClassEscape

CharacterEscape ::= ControlEscape | 'c' ControlLetter | HexEscapeSequence | UnicodeEscapeSequence | IdentityEscape

ControlEscape ::= one of { 'f', 'n', 'r', 't', 'v' }

ControlLetter ::= one of { 'a'..'z', 'A'..'Z' }

IdentityEscape ::= (SourceCharacter but not IdentifierPart) | <ZWJ> | <ZWNJ>

DecimalEscape ::= DecimalIntegerLiteral [lookahead not in DecimalDigit]

CharacterClassEscape :: one of { 'd', 'D', 's', 'S', 'w', 'W' }

CharacterClass ::= '[' [lookahead not in {'^'}] ClassRanges ']' | '[' '^' ClassRanges ']'

ClassRanges ::= [empty] | NonemptyClassRanges

NonemptyClassRanges ::= ClassAtom | ClassAtom NonemptyClassRangesNoDash | ClassAtom '-' ClassAtom ClassRanges

NonemptyClassRangesNoDash ::= ClassAtom | ClassAtomNoDash NonemptyClassRangesNoDash | ClassAtomNoDash '-' ClassAtom ClassRanges

ClassAtom ::= '-' | ClassAtomNoDash

ClassAtomNoDash ::= SourceCharacter but not one of { '\', ']', '-' } | '\' ClassEscape

ClassEscape ::= DecimalEscape | 'b' | CharacterEscape | CharacterClassEscape

SourceCharacter ::= <unicode character>

HexEscapeSequence ::= 'x' HexDigit HexDigit

UnicodeEscapeSequence ::= 'u' HexDigit HexDigit HexDigit HexDigit | 'u' '{' HexDigits '}'

HexDigit ::= one of { '0'..'9', 'a'..'f', 'A'..'F' }

IdentifierStart ::= UnicodeLetter | '$' | '\_' | '\' UnicodeEscapeSequence

IdentifierPart ::= IdentifierStart | UnicodeCombiningMark | UnicodeDigit | UnicodeConnectorPunctuation | <ZWNJ> | <ZWJ>

UnicodeLetter ::= <unicode Uppercase letter> | <unicode Lowercase letter> | <unicode Titlecase letter> | <unicode Modifier letter> | <unicode Other letter> | <unicode Letter number>

UnicodeCombiningMark = <unicode Non-spacing mark> | <unicode combining spacing mark>

UnicodeDigit ::= <unicode Decimal number>

UnicodeConnectorPunctuation ::= <unicode Connector punctuation>

DecimalIntegerLiteral ::= '0' | NonZeroDigit DecimalDigits?

DecimalDigits ::= DecimalDigit | DecimalDigits DecimalDigit

DecimalDigit ::= one of { '0'..'9' }

NonZeroDigit ::= one of { '1'..'9' }

------------------

Annex B Deviations

------------------

1. The assertions (?= ) and (?! ) are treated as though they have a surrounding non-capture group, and hence can be quantified.

Other assertions are not quantifiable.

QuantifiableAssertion [added] ::= '(' '?' '=' Disjunction ')' | '(' '?' '!' Disjunction ')'

Assertion [replaced] ::= '^' | '$' | '\' 'b' | '\' 'B' | QuantifiableAssertion

Term ::= ... | QuantifiableAssertion Quantifier [added]

--------------------------------------------------------------------------------------------------------------------------------

Left factored

--------------------------------------------------------------------------------------------------------------------------------

Pattern ::= Disjunction

Disjunction ::= Alternative ('|' Alternative)\*

FOLLOW(Disjunction) = { <eof>, ')' }

Alternative ::= Term\*

FOLLOW(Alternative) = { <eof>, ')', '|' }

Term ::= ( '^' | '$' | '\' 'b' | '\' 'B' | '(' '?' '=' Disjunction ')' | '(' '?' '!' Disjunction ')' | Atom Quantifier? )

| ( PatternCharacter | '.' | '\' AtomEscape | CharacterClass | '(' Disjunction ')' | '(' '?' ':' Disjunction ')' )

( '\*' | '+' | '?' | '{' DecimalDigits (',' DecimalDigits?)? '}' ) '?'?

PatternCharacter ::= <unicode character> but not any of { '^', '$', '\', '.', '\*', '+', '?', '(', ')', '[', ']', '{', '}', '|' }

AtomEscape ::= DecimalEscape | CharacterEscape | CharacterClassEscape

CharacterEscape ::= ControlEscape | 'c' ControlLetter | HexEscapeSequence | UnicodeEscapeSequence | IdentityEscape

ControlEscape ::= one of { 'f', 'n', 'r', 't', 'v' }

ControlLetter ::= one of { 'a'..'z', 'A'..'Z' }

IdentityEscape ::= <unicode character> but not <unicode Uppercase letter>, <unicode Lowercase letter>, <unicode Titlecase letter>, <unicode Modifier letter>, <unicode Other letter>, <unicode Letter number>, '$', '\_', <unicode Non-spacing mark>, <unicode combining spacing mark>, <unicode Decimal number>, <unicode Connector punctuation>

DecimalEscape ::= DecimalIntegerLiteral [lookahead not in DecimalDigit]

CharacterClassEscape :: one of { 'd', 'D', 's', 'S', 'w', 'W' }

CharacterClass ::= '[' [lookahead not in {'^'}] ClassRanges ']' | '[' '^' ClassRanges ']'

ClassRanges ::= [empty] | NonemptyClassRanges

NonemptyClassRanges ::= ClassAtom | ClassAtom NonemptyClassRangesNoDash | ClassAtom '-' ClassAtom ClassRanges

NonemptyClassRangesNoDash ::= ClassAtom | ClassAtomNoDash NonemptyClassRangesNoDash | ClassAtomNoDash '-' ClassAtom ClassRanges

ClassAtom ::= '-' | ClassAtomNoDash

ClassAtomNoDash ::= SourceCharacter but not one of { '\', ']', '-' } | '\' ClassEscape

ClassEscape ::= DecimalEscape | 'b' | CharacterEscape | CharacterClassEscape

HexEscapeSequence ::= 'x' HexDigit{2}

UnicodeEscapeSequence ::= 'u' HexDigit{4} | 'u' '{' HexDigits{4-6} '}'

HexDigit ::= one of { '0'..'9', 'a'..'f', 'A'..'F' }

DecimalIntegerLiteral ::= '0' | NonZeroDigit DecimalDigits?

DecimalDigits ::= DecimalDigit+

DecimalDigit ::= one of { '0'..'9' }

NonZeroDigit ::= one of { '1'..'9' }

------------------

Annex B Deviations

------------------

QuantifiableAssertion [added] ::= '(' '?' '=' Disjunction ')' | '(' '?' '!' Disjunction ')'

Term ::= ... | '(' '?' '=' Disjunction ')' [removed] | '(' '?' '!' Disjunction ')' [removed] | QuantifiableAssertion Quantifier? [added]

\*/

#include "ParserPch.h"

namespace UnifiedRegex

{

ParseError::ParseError(bool isBody, CharCount pos, CharCount encodedPos, HRESULT error)

: isBody(isBody), pos(pos), encodedPos(encodedPos), error(error)

{

}

template <typename P, const bool IsLiteral>

Parser<P, IsLiteral>::Parser

( Js::ScriptContext\* scriptContext

, ArenaAllocator\* ctAllocator

, StandardChars<EncodedChar>\* standardEncodedChars

, StandardChars<Char>\* standardChars

, bool isFromExternalSource

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, DebugWriter\* w

#endif

)

: scriptContext(scriptContext)

, ctAllocator(ctAllocator)

, standardEncodedChars(standardEncodedChars)

, standardChars(standardChars)

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, w(w)

#endif

, input(0)

, inputLim(0)

, next(0)

, inBody(false)

, numGroups(1)

, nextGroupId(1) // implicit overall group always takes index 0

, litbuf(0)

, litbufLen(0)

, litbufNext(0)

, surrogatePairList(nullptr)

, currentSurrogatePairNode(nullptr)

, tempLocationOfSurrogatePair(nullptr)

, tempLocationOfRange(nullptr)

, codePointAtTempLocation(0)

, unicodeFlagPresent(false)

, caseInsensitiveFlagPresent(false)

, positionAfterLastSurrogate(nullptr)

, valueOfLastSurrogate(INVALID\_CODEPOINT)

, deferredIfNotUnicodeError(nullptr)

, deferredIfUnicodeError(nullptr)

{

if (isFromExternalSource)

FromExternalSource();

}

//

// Input buffer management

//

template <typename P, const bool IsLiteral>

void Parser<P, IsLiteral>::SetPosition(const EncodedChar\* input, const EncodedChar\* inputLim, bool inBody)

{

this->input = input;

this->inputLim = inputLim;

next = input;

this->inBody = inBody;

RestoreMultiUnits(0);

}

template <typename P, const bool IsLiteral>

inline CharCount Parser<P, IsLiteral>::Pos()

{

CharCount nextOffset = Chars<EncodedChar>::OSB(next, input);

Assert(nextOffset >= m\_cMultiUnits);

return nextOffset - (CharCount)m\_cMultiUnits;

}

template <typename P, const bool IsLiteral>

inline bool Parser<P, IsLiteral>::IsEOF()

{

return next >= inputLim;

}

template <typename P, const bool IsLiteral>

inline bool Parser<P, IsLiteral>::ECCanConsume(CharCount n = 1)

{

return next + n <= inputLim;

}

template <typename P, const bool IsLiteral>

inline typename P::EncodedChar Parser<P, IsLiteral>::ECLookahead(CharCount n = 0)

{

// Ok to look ahead to terminating 0

Assert(next + n <= inputLim);

return next[n];

}

template <typename P, const bool IsLiteral>

inline typename P::EncodedChar Parser<P, IsLiteral>::ECLookback(CharCount n = 0)

{

// Ok to look ahead to terminating 0

Assert(n + input <= next);

return \*(next - n);

}

template <typename P, const bool IsLiteral>

inline void Parser<P, IsLiteral>::ECConsume(CharCount n = 1)

{

Assert(next + n <= inputLim);

#if DBG

for (CharCount i = 0; i < n; i++)

Assert(!IsMultiUnitChar(next[i]));

#endif

next += n;

}

template <typename P, const bool IsLiteral>

inline void Parser<P, IsLiteral>::ECConsumeMultiUnit(CharCount n = 1)

{

Assert(next + n <= inputLim);

next += n;

}

template <typename P, const bool IsLiteral>

inline void Parser<P, IsLiteral>::ECRevert(CharCount n = 1)

{

Assert(n + input <= next);

next -= n;

}

//

// Helpers

//

template <typename P, const bool IsLiteral>

int Parser<P, IsLiteral>::TryParseExtendedUnicodeEscape(Char& c, bool& previousSurrogatePart, bool trackSurrogatePair = false)

{

if (!scriptContext->GetConfig()->IsES6UnicodeExtensionsEnabled())

{

return 0;

}

if (!ECCanConsume(2) || ECLookahead(0) !='{' || !standardEncodedChars->IsHex(ECLookahead(1)))

{

return 0;

}

// The first character is mandatory to consume escape sequence, so we check for it above, at this stage we can set it as we already checked.

codepoint\_t codePoint = standardEncodedChars->DigitValue(ECLookahead(1));

int i = 2;

while(ECCanConsume(i + 1) && standardEncodedChars->IsHex(ECLookahead(i)))

{

codePoint <<= 4;

codePoint += standardEncodedChars->DigitValue(ECLookahead(i));

if (codePoint > 0x10FFFF)

{

return 0;

}

i++;

}

if(!ECCanConsume(i + 1) || ECLookahead(i) != '}')

{

return 0;

}

uint consumptionNumber = i + 1;

Assert(consumptionNumber >= 3);

if (!previousSurrogatePart && trackSurrogatePair && this->scriptContext->GetConfig()->IsES6UnicodeExtensionsEnabled())

{

// Current location

TrackIfSurrogatePair(codePoint, (next - 1), consumptionNumber + 1);

}

wchar\_t other;

// Generally if this code point is a single character, then we take it and return.

// If the character is made up of two characters then we emit the first and backtrack to the start of th escape sequence;

// Following that we check if we have already seen the first character, and if so emit the second and consume the entire escape sequence.

if (codePoint < 0x10000)

{

c = UTC(codePoint);

ECConsumeMultiUnit(consumptionNumber);

}

else if (previousSurrogatePart)

{

previousSurrogatePart = false;

Js::NumberUtilities::CodePointAsSurrogatePair(codePoint, &other, &c);

ECConsumeMultiUnit(consumptionNumber);

}

else

{

previousSurrogatePart = true;

Js::NumberUtilities::CodePointAsSurrogatePair(codePoint, &c, &other);

Assert(ECLookback(1) == 'u' && ECLookback(2) == '\\');

ECRevert(2);

}

return consumptionNumber;

}

// This function has the following 'knowledge':

// - A codepoint that might be part of a surrogate pair or part of one

// - A location where that codepoint is located

// - Previously tracked part of surrogate pair (tempLocationOfSurrogatePair, and codePointAtTempLocation), which is always a surrogate lower part.

// - A pointer to the current location of the linked list

// - If a previous location is tracked, then it is of a parsed character (not given character) before current, and not same to current

// This can't be asserted directly, and has to be followed by callers. Term pass can reset with each iteration, as well as this method in cases it needs.

template <typename P, const bool IsLiteral>

void Parser<P, IsLiteral>:: TrackIfSurrogatePair(codepoint\_t codePoint, const EncodedChar\* location, uint32 consumptionLength)

{

Assert(codePoint < 0x110000);

Assert(location != nullptr);

Assert(location != this->tempLocationOfSurrogatePair);

if (Js::NumberUtilities::IsSurrogateLowerPart(codePoint))

{

this->tempLocationOfSurrogatePair = location;

this->codePointAtTempLocation = codePoint;

}

else

{

if(Js::NumberUtilities::IsSurrogateUpperPart(codePoint) && this->tempLocationOfSurrogatePair != nullptr)

{

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

consumptionLength = (uint32)(location - this->tempLocationOfSurrogatePair) + consumptionLength;

codePoint = Js::NumberUtilities::SurrogatePairAsCodePoint(codePointAtTempLocation, codePoint);

location = this->tempLocationOfSurrogatePair;

}

// At this point we can clear previous location, and then if codePoint is bigger than 0xFFFF store it, as we either received it or combined it above

this->tempLocationOfSurrogatePair = nullptr;

this->codePointAtTempLocation = 0;

}

if (codePoint > 0xFFFF)

{

this->positionAfterLastSurrogate = location + consumptionLength;

this->valueOfLastSurrogate = codePoint;

// When parsing without AST we aren't given an allocator. In addition, only the 2 lines above are used during Pass 0;

// while the bottom is used during Pass 1 (which isn't done when ParseNoAST)

if(this->ctAllocator != nullptr)

{

SurrogatePairTracker\* node = Anew(this->ctAllocator, SurrogatePairTracker, location, this->tempLocationOfRange, codePoint, consumptionLength, m\_cMultiUnits);

if (surrogatePairList == nullptr)

{

Assert(currentSurrogatePairNode == nullptr);

surrogatePairList = node;

currentSurrogatePairNode = node;

}

else

{

Assert(currentSurrogatePairNode != nullptr);

currentSurrogatePairNode->next = node;

currentSurrogatePairNode = node;

}

}

}

}

template <typename P, const bool IsLiteral>

Node\* Parser<P, IsLiteral>::CreateSurrogatePairAtom(wchar\_t lower, wchar\_t upper)

{

MatchLiteralNode \* literalNode = Anew(this->ctAllocator, MatchLiteralNode, 0, 0);

MatchCharNode lowerNode(lower);

MatchCharNode upperNode(upper);

AccumLiteral(literalNode, &lowerNode);

AccumLiteral(literalNode, &upperNode);

return literalNode;

}

// This function will create appropriate pairs of ranges and add them to the disjunction node.

// Terms used in comments:

// - A minor codePoint is smaller than major codePoint, and both define a range of codePoints above 0x10000; to avoid confusion between lower/upper denoting codeUnits composing the surrogate pair.

// - A boundary is a mod 0x400 alignment marker due to the nature of surrogate pairs representation. So the codepoint 0x10300 lies between boundaries 0x10000 and 0x10400.

// - A prefix is the range set used to represent the values from minorCodePoint to the first boundary above minorCodePoint if applicable.

// - A suffix is the range set used to represent the values from first boundary below majorCodePoint to the majorCodePoint if applicable.

// - A full range is the range set used to represent the values from first boundary above minorCodePoint to first boundary below majorCodePoint if applicable.

// The algorithm works as follows:

// 1. Determine minorBoundary (minorCodePoint - mod 0x400 +0x400) and majorBoundary (majorCodePoint - mod 0x400). minorBoundary > minorCodePoint and majorBoundary < majorCodePoint

// 2. Based on the codePoints and the boundaries, prefix, suffix, and full range is determined. Here are the rules:

// 2-a. If minorBoundary > majorBoundary, we have an inner boundary range, output just that.

// 2-b. If minorBoundary - 0x400u != minorCodepoint (i.e. codePoint doesn't lie right on a boundary to be part of a full range) we have a prefix.

// 2-c. If majorBoundary + 0x3FFu != majorCodepoint (i.e. codePoint doesn't lie right before a boundary to be part of a full range) we have a suffix.

// 2-d. We have a full range, if the two boundaries don't equal, OR the codePoints lie on the range boundaries opposite to what constitutes a prefix/suffix.

// Visual representation for sample range 0x10300 - 0x10900

// | [ \_ | ^ ] |

// 0x10000 0x10800 0x11000

// [ ] - denote the actual range

// \_ - minorBoundary

// ^ - majorBoundary

// | - other boundaries

// prefix is between [ and \_

// suffix is between ^ and ]

// full range is between \_ and ^

template <typename P, const bool IsLiteral>

AltNode\* Parser<P, IsLiteral>::AppendSurrogateRangeToDisjunction(codepoint\_t minorCodePoint, codepoint\_t majorCodePoint, AltNode \*lastAltNode)

{

Assert(minorCodePoint < majorCodePoint);

Assert(minorCodePoint >= 0x10000u);

Assert(majorCodePoint >= 0x10000u);

Assert(scriptContext->GetConfig()->IsES6UnicodeExtensionsEnabled() && unicodeFlagPresent);

wchar\_t lowerMinorCodeUnit, upperMinorCodeUnit, lowerMajorCodeUnit, upperMajorCodeUnit;

Js::NumberUtilities::CodePointAsSurrogatePair(minorCodePoint, &lowerMinorCodeUnit, &upperMinorCodeUnit);

Js::NumberUtilities::CodePointAsSurrogatePair(majorCodePoint, &lowerMajorCodeUnit, &upperMajorCodeUnit);

// These boundaries represent whole range boundaries, as in 0x10000, 0x10400, 0x10800 etc

// minor boundary is the first boundary strictly above minorCodePoint

// major boundary is the first boundary below or equal to majorCodePoint

codepoint\_t minorBoundary = minorCodePoint - (minorCodePoint % 0x400u) + 0x400u;

codepoint\_t majorBoundary = majorCodePoint - (majorCodePoint % 0x400u);

Assert(minorBoundary >= 0x10000);

Assert(majorBoundary >= 0x10000);

AltNode\* tailToAdd = nullptr;

// If the minor boundary is higher than major boundary, that means we have a range within the boundary and is less than 0x400

// Ex: 0x10430 - 0x10700 will have minor boundary of 0x10800 and major of 0x10400

// This pair will be represented in single range set.

const bool singleRange = minorBoundary > majorBoundary;

if (singleRange)

{

Assert(majorCodePoint - minorCodePoint < 0x400u);

Assert(lowerMinorCodeUnit == lowerMajorCodeUnit);

MatchCharNode\* lowerCharNode = Anew(ctAllocator, MatchCharNode, lowerMinorCodeUnit);

MatchSetNode\* setNode = Anew(ctAllocator, MatchSetNode, false, false);

setNode->set.SetRange(ctAllocator, (Char)upperMinorCodeUnit, (Char)upperMajorCodeUnit);

ConcatNode\* concatNode = Anew(ctAllocator, ConcatNode, lowerCharNode, Anew(ctAllocator, ConcatNode, setNode, nullptr));

tailToAdd = Anew(ctAllocator, AltNode, concatNode, nullptr);

}

else

{

Node\* prefixNode = nullptr, \*suffixNode = nullptr;

const bool twoConsecutiveRanges = minorBoundary == majorBoundary;

// For minorBoundary,

if (minorBoundary - minorCodePoint == 1) // Single character in minor range

{

// The prefix is only a surrogate pair atom

prefixNode = CreateSurrogatePairAtom(lowerMinorCodeUnit, upperMinorCodeUnit);

}

else if (minorCodePoint != minorBoundary - 0x400u) // Minor range isn't full

{

Assert(minorBoundary - minorCodePoint < 0x400u);

MatchCharNode\* lowerCharNode = Anew(ctAllocator, MatchCharNode, (Char)lowerMinorCodeUnit);

MatchSetNode\* upperSetNode = Anew(ctAllocator, MatchSetNode, false);

upperSetNode->set.SetRange(ctAllocator, (Char)upperMinorCodeUnit, (Char)0xDFFFu);

prefixNode = Anew(ctAllocator, ConcatNode, lowerCharNode, Anew(ctAllocator, ConcatNode, upperSetNode, nullptr));

}

else // Full minor range

{

minorBoundary -= 0x400u;

}

if (majorBoundary == majorCodePoint) // Single character in major range

{

// The suffix is only a surrogate pair atom

suffixNode = CreateSurrogatePairAtom(lowerMajorCodeUnit, upperMajorCodeUnit);

majorBoundary -= 0x400u;

}

else if (majorBoundary + 0x3FFu != majorCodePoint) // Major range isn't full

{

Assert(majorCodePoint - majorBoundary < 0x3FFu);

MatchCharNode\* lowerCharNode = Anew(ctAllocator, MatchCharNode, (Char)lowerMajorCodeUnit);

MatchSetNode\* upperSetNode = Anew(ctAllocator, MatchSetNode, false, false);

upperSetNode->set.SetRange(ctAllocator, (Char)0xDC00u, (Char)upperMajorCodeUnit);

suffixNode = Anew(ctAllocator, ConcatNode, lowerCharNode, Anew(ctAllocator, ConcatNode, upperSetNode, nullptr));

majorBoundary -= 0x400u;

}

const bool nonFullConsecutiveRanges = twoConsecutiveRanges && prefixNode != nullptr && suffixNode != nullptr;

if (nonFullConsecutiveRanges)

{

Assert(suffixNode != nullptr);

Assert(minorCodePoint != minorBoundary - 0x400u);

Assert(majorBoundary + 0x3FFu != majorCodePoint);

// If the minor boundary is equal to major boundary, that means we have a cross boundary range that only needs 2 nodes for prefix/suffix.

// We can only cross one boundary.

Assert(majorCodePoint - minorCodePoint < 0x800u);

tailToAdd = Anew(ctAllocator, AltNode, prefixNode, Anew(ctAllocator, AltNode, suffixNode, nullptr));

}

else

{

// We have 3 sets of ranges, comprising of prefix, full and suffix.

Assert(majorCodePoint - minorCodePoint >= 0x400u);

Assert((prefixNode != nullptr && suffixNode != nullptr) // Spanning more than two ranges

|| (prefixNode == nullptr && minorBoundary == minorCodePoint) // Two consecutive ranges and the minor is full

|| (suffixNode == nullptr && majorBoundary + 0x3FFu == majorCodePoint)); // Two consecutive ranges and the major is full

Node\* lowerOfFullRange;

wchar\_t lowerMinorBoundary, lowerMajorBoundary, ignore;

Js::NumberUtilities::CodePointAsSurrogatePair(minorBoundary, &lowerMinorBoundary, &ignore);

bool singleFullRange = majorBoundary == minorBoundary;

if (singleFullRange)

{

// The lower part of the full range is simple a surrogate lower char

lowerOfFullRange = Anew(ctAllocator, MatchCharNode, (Char)lowerMinorBoundary);

}

else

{

Js::NumberUtilities::CodePointAsSurrogatePair(majorBoundary, &lowerMajorBoundary, &ignore);

MatchSetNode\* setNode = Anew(ctAllocator, MatchSetNode, false, false);

setNode->set.SetRange(ctAllocator, (Char)lowerMinorBoundary, (Char)lowerMajorBoundary);

lowerOfFullRange = setNode;

}

MatchSetNode\* fullUpperRange = Anew(ctAllocator, MatchSetNode, false, false);

fullUpperRange->set.SetRange(ctAllocator, (Char)0xDC00u, (Char)0xDFFFu);

// These are added in the following order [full] [prefix][suffix]

// This is doing by prepending, so in reverse.

if (suffixNode != nullptr)

{

tailToAdd = Anew(ctAllocator, AltNode, suffixNode, tailToAdd);

}

if (prefixNode != nullptr)

{

tailToAdd = Anew(ctAllocator, AltNode, prefixNode, tailToAdd);

}

tailToAdd = Anew(ctAllocator, AltNode, Anew(ctAllocator, ConcatNode, lowerOfFullRange, Anew(ctAllocator, ConcatNode, fullUpperRange, nullptr)), tailToAdd);

}

}

if (lastAltNode != nullptr)

{

Assert(lastAltNode->tail == nullptr);

lastAltNode->tail = tailToAdd;

}

return tailToAdd;

}

template <typename P, const bool IsLiteral>

AltNode\* Parser<P, IsLiteral>::AppendSurrogatePairToDisjunction(codepoint\_t codePoint, AltNode \*lastAltNode)

{

wchar\_t lower, upper;

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

AltNode\* tailNode = Anew(ctAllocator, AltNode, CreateSurrogatePairAtom(lower, upper), nullptr);

if (lastAltNode != nullptr)

{

lastAltNode->tail = tailNode;

}

return tailNode;

}

//

// Errors

//

template <typename P, const bool IsLiteral>

void Parser<P, IsLiteral>::Fail(HRESULT error)

{

throw ParseError(inBody, Pos(), Chars<EncodedChar>::OSB(next, input), error);

}

// This doesn't throw, but stores first error code for throwing later

template <typename P, const bool IsLiteral>

void Parser<P, IsLiteral>::DeferredFailIfUnicode(HRESULT error)

{

Assert(this->scriptContext->GetConfig()->IsES6UnicodeExtensionsEnabled());

if (this->deferredIfUnicodeError == nullptr)

{

this->deferredIfUnicodeError = Anew(ctAllocator, ParseError, inBody, Pos(), Chars<EncodedChar>::OSB(next, input), error);

}

}

// This doesn't throw, but stores first error code for throwing later

template <typename P, const bool IsLiteral>

void Parser<P, IsLiteral>::DeferredFailIfNotUnicode(HRESULT error)

{

Assert(this->scriptContext->GetConfig()->IsES6UnicodeExtensionsEnabled());

if (this->deferredIfNotUnicodeError == nullptr)

{

this->deferredIfNotUnicodeError = Anew(ctAllocator, ParseError, inBody, Pos(), Chars<EncodedChar>::OSB(next, input), error);

}

}

template <typename P, const bool IsLiteral>

inline void Parser<P, IsLiteral>::ECMust(EncodedChar ec, HRESULT error)

{

// We never look for 0

Assert(ec != 0);

if (ECLookahead() != ec)

Fail(error);

ECConsume();

}

template <typename P, const bool IsLiteral>

inline wchar\_t Parser<P, IsLiteral>::NextChar()

{

Assert(!IsEOF());

// Could be an embedded 0

Char c = ReadFull<true>(next, inputLim);

// No embedded newlines in literals

if (IsLiteral && standardChars->IsNewline(c))

Fail(ERRnoSlash);

return c;

}

//

// Patterns/Disjunctions/Alternatives

//

template <typename P, const bool IsLiteral>

void Parser<P, IsLiteral>::PatternPass0()

{

this->positionAfterLastSurrogate = nullptr;

this->deferredIfNotUnicodeError = nullptr;

this->deferredIfUnicodeError = nullptr;

DisjunctionPass0(0);

}

template <typename P, const bool IsLiteral>

Node\* Parser<P, IsLiteral>::PatternPass1()

{

return DisjunctionPass1();

}

template <typename P, const bool IsLiteral>

Node\* Parser<P, IsLiteral>::UnionNodes(Node\* prev, Node\* curr)

{

if (prev->tag == Node::MatchChar)

{

MatchCharNode\* prevChar = (MatchCharNode\*)prev;

if (curr->tag == Node::MatchChar)

{

MatchCharNode\* currChar = (MatchCharNode\*)curr;

if (prevChar->cs[0] == currChar->cs[0])

// Just ignore current node

return prevChar;

else

{

// Union chars into new set

MatchSetNode\* setNode = Anew(ctAllocator, MatchSetNode, false);

setNode->set.Set(ctAllocator, prevChar->cs[0]);

setNode->set.Set(ctAllocator, currChar->cs[0]);

return setNode;

}

}

else if (curr->tag == Node::MatchSet)

{

MatchSetNode\* currSet = (MatchSetNode\*)curr;

if (currSet->isNegation)

// Can't merge

return 0;

else

{

// Union chars into new set

MatchSetNode\* setNode = Anew(ctAllocator, MatchSetNode, false);

setNode->set.Set(ctAllocator, prevChar->cs[0]) ;

setNode->set.UnionInPlace(ctAllocator, currSet->set);

return setNode;

}

}

else

// Can't merge

return 0;

}

else if (prev->tag == Node::MatchSet)

{

MatchSetNode\* prevSet = (MatchSetNode\*)prev;

if (prevSet->isNegation)

// Can't merge

return 0;

else if (curr->tag == Node::MatchChar)

{

MatchCharNode\* currChar = (MatchCharNode\*)curr;

// Include char in prev set

prevSet->set.Set(ctAllocator, currChar->cs[0]);

return prevSet;

}

else if (curr->tag == Node::MatchSet)

{

MatchSetNode\* currSet = (MatchSetNode\*)curr;

if (currSet->isNegation)

// Can't merge

return 0;

else

{

// Include chars in prev set

prevSet->set.UnionInPlace(ctAllocator, currSet->set);

return prevSet;

}

}

else

// Can't merge

return 0;

}

else

// Can't merge

return 0;

}

template <typename P, const bool IsLiteral>

void Parser<P, IsLiteral>::DisjunctionPass0(int depth)

{

AlternativePass0(depth);

while (true)

{

// Could be terminating 0

if (ECLookahead() != '|')

return;

ECConsume();

AlternativePass0(depth);

}

}

template <typename P, const bool IsLiteral>

Node\* Parser<P, IsLiteral>::DisjunctionPass1()

{

// Maintain the invariants:

// - alt lists have two or more items

// - alt list items are never alt lists (so we must inline them)

// - an alt list never contains two consecutive match-character/match-set nodes

// (so we must union consecutive items into a single set)

Node\* node = AlternativePass1();

AltNode\* last = 0;

// First node may be an alternative

if (node->tag == Node::Alt)

{

last = (AltNode\*)node;

while (last->tail != 0)

last = last->tail;

}

while (true)

{

// Could be terminating 0

if (ECLookahead() != '|')

return node;

ECConsume(); // '|'

Node\* next = AlternativePass1();

AnalysisAssert(next != nullptr);

Node\* revisedPrev = UnionNodes(last == 0 ? node : last->head, next);

if (revisedPrev != 0)

{

// Can merge next into previously seen alternative

if (last == 0)

node = revisedPrev;

else

last->head = revisedPrev;

}

else if (next->tag == Node::Alt)

{

AltNode\* nextList = (AltNode\*)next;

// Append inner list to current list

revisedPrev = UnionNodes(last == 0 ? node : last->head, nextList->head);

if (revisedPrev != 0)

{

// Can merge head of list into previously seen alternative

if (last ==0)

node = revisedPrev;

else

last->head = revisedPrev;

nextList = nextList->tail;

}

AnalysisAssert(nextList != nullptr);

if (last == 0)

node = Anew(ctAllocator, AltNode, node, nextList);

else

last->tail = nextList;

while (nextList->tail != 0)

nextList = nextList->tail;

last = nextList;

}

else

{

// Append node

AltNode\* cons = Anew(ctAllocator, AltNode, next, 0);

if (last == 0)

node = Anew(ctAllocator, AltNode, node, cons);

else

last->tail = cons;

last = cons;

}

}

}

template <typename P, const bool IsLiteral>

inline bool Parser<P, IsLiteral>::IsEndOfAlternative()

{

EncodedChar ec = ECLookahead();

// Could be terminating 0, but embedded 0 is part of alternative

return (ec == 0 && IsEOF()) || ec == ')' || ec == '|' || (IsLiteral && ec == '/');

}

template <typename P, const bool IsLiteral>

void Parser<P, IsLiteral>::EnsureLitbuf(CharCount size)

{

if (litbufLen - litbufNext < size)

{

CharCount newLen = max(litbufLen, initLitbufSize);

while (newLen < litbufNext + size)

newLen \*= 2;

litbuf = (Char\*)ctAllocator->Realloc(litbuf, litbufLen \* sizeof(Char), newLen \* sizeof(Char));

litbufLen = newLen;

}

}

template <typename P, const bool IsLiteral>

void Parser<P, IsLiteral>::AccumLiteral(MatchLiteralNode\* deferredLiteralNode, Node\* charOrLiteralNode)

{

Assert(charOrLiteralNode->tag == Node::MatchChar || charOrLiteralNode->tag == Node::MatchLiteral);

CharCount addLen = charOrLiteralNode->LiteralLength();

Assert(addLen > 0);

if (deferredLiteralNode->length == 0)

{

// Start a new literal

EnsureLitbuf(addLen);

deferredLiteralNode->offset = litbufNext;

deferredLiteralNode->length = addLen;

charOrLiteralNode->AppendLiteral(litbufNext, litbufLen, litbuf);

}

else if (deferredLiteralNode->offset + deferredLiteralNode->length == litbufNext)

{

// Keep growing the current literal

EnsureLitbuf(addLen);

charOrLiteralNode->AppendLiteral(litbufNext, litbufLen, litbuf);

deferredLiteralNode->length += addLen;

}

else if (charOrLiteralNode->tag == Node::MatchLiteral && deferredLiteralNode->offset + deferredLiteralNode->length == ((MatchLiteralNode\*)charOrLiteralNode)->offset)

{

// Absorb next literal into current literal since they are adjacent

deferredLiteralNode->length += addLen;

}

else

{

// Abandon current literal and start a fresh one (leaves gap)

EnsureLitbuf(deferredLiteralNode->length + addLen);

js\_wmemcpy\_s(litbuf + litbufNext, litbufLen - litbufNext, litbuf + deferredLiteralNode->offset, deferredLiteralNode->length);

deferredLiteralNode->offset = litbufNext;

litbufNext += deferredLiteralNode->length;

charOrLiteralNode->AppendLiteral(litbufNext, litbufLen, litbuf);

deferredLiteralNode->length += addLen;

}

}

template <typename P, const bool IsLiteral>

Node\* Parser<P, IsLiteral>::FinalTerm(Node\* node, MatchLiteralNode\* deferredLiteralNode)

{

if (node == deferredLiteralNode)

{

#if DBG

if (deferredLiteralNode->length == 0)

Assert(false);

#endif

Assert(deferredLiteralNode->offset < litbufNext);

Assert(deferredLiteralNode->offset + deferredLiteralNode->length <= litbufNext);

if (deferredLiteralNode->length == 1)

{

node = Anew(ctAllocator, MatchCharNode, litbuf[deferredLiteralNode->offset]);

if (deferredLiteralNode->offset + deferredLiteralNode->length == litbufNext)

// Reclaim last added character

litbufNext--;

// else: leave a gap in the literal buffer

}

else

node = Anew(ctAllocator, MatchLiteralNode, \*deferredLiteralNode);

deferredLiteralNode->offset = 0;

deferredLiteralNode->length = 0;

}

return node;

}

template <typename P, const bool IsLiteral>

void Parser<P, IsLiteral>::AlternativePass0(int depth)

{

while (!IsEndOfAlternative())

TermPass0(depth);

}

template <typename P, const bool IsLiteral>

Node\* Parser<P, IsLiteral>::AlternativePass1()

{

if (IsEndOfAlternative())

return Anew(ctAllocator, SimpleNode, Node::Empty);

MatchCharNode deferredCharNode(0);

MatchLiteralNode deferredLiteralNode(0, 0);

// Maintain the invariants:

// - concat lists have two or more items

// - concat list items are never concat lists

// - a concat list never contains two consecutive match-character/match-literal nodes

bool previousSurrogatePart = false;

Node\* node = TermPass1(&deferredCharNode, previousSurrogatePart);

AnalysisAssert(node != nullptr);

ConcatNode\* last = 0;

// First node may be a concat

if (node->tag == Node::Concat)

{

last = (ConcatNode\*)node;

while (last->tail != 0)

last = last->tail;

}

if (last == 0)

{

if (node->LiteralLength() > 0)

{

// Begin a new literal

AccumLiteral(&deferredLiteralNode, node);

node = &deferredLiteralNode;

}

}

else

{

if (last->head->LiteralLength() > 0)

{

// Begin a new literal

AccumLiteral(&deferredLiteralNode, last->head);

last->head = &deferredLiteralNode;

}

}

while (!IsEndOfAlternative())

{

Node\* next = TermPass1(&deferredCharNode, previousSurrogatePart);

AnalysisAssert(next != nullptr);

if (next->LiteralLength() > 0)

{

// Begin a new literal or grow the existing literal

AccumLiteral(&deferredLiteralNode, next);

if (last == 0)

{

if (node != &deferredLiteralNode)

{

// So far we have first item and the current literal

ConcatNode\* cons = Anew(ctAllocator, ConcatNode, &deferredLiteralNode, 0);

node = Anew(ctAllocator, ConcatNode, node, cons);

last = cons;

}

// else: keep growing first literal

}

else

{

if (last->head != &deferredLiteralNode)

{

// Append a new literal node

ConcatNode\* cons = Anew(ctAllocator, ConcatNode, &deferredLiteralNode, 0);

last->tail = cons;

last = cons;

}

// else: keep growing current literal

}

}

else if (next->tag == Node::Concat)

{

// Append this list to accumulated list

ConcatNode\* nextList = (ConcatNode\*)next;

if (nextList->head->LiteralLength() > 0 &&

((last == 0 && node == &deferredLiteralNode) ||

(last != 0 && last->head == &deferredLiteralNode)))

{

// Absorb the next character or literal into the current literal

// (may leave a gab in litbuf)

AccumLiteral(&deferredLiteralNode, nextList->head);

nextList = nextList->tail;

// List has at least two items

AnalysisAssert(nextList != 0);

// List should be in canonical form, so no consecutive chars/literals

Assert(nextList->head->LiteralLength() == 0);

}

if (last == 0)

node = Anew(ctAllocator, ConcatNode, FinalTerm(node, &deferredLiteralNode), nextList);

else

{

last->head = FinalTerm(last->head, &deferredLiteralNode);

last->tail = nextList;

}

while (nextList->tail != 0)

nextList = nextList->tail;

last = nextList;

// No outstanding literals

Assert(deferredLiteralNode.length == 0);

if (last->head->LiteralLength() > 0)

{

// If the list ends with a literal, transfer it into deferredLiteralNode

// so we can continue accumulating (won't leave a gab in litbuf)

AccumLiteral(&deferredLiteralNode, last->head);

// Can discard MatchLiteralNode since it lives in compile-time allocator

last->head = &deferredLiteralNode;

}

}

else

{

// Append this node to accumulated list

ConcatNode\* cons = Anew(ctAllocator, ConcatNode, next, 0);

if (last == 0)

node = Anew(ctAllocator, ConcatNode, FinalTerm(node, &deferredLiteralNode), cons);

else

{

last->head = FinalTerm(last->head, &deferredLiteralNode);

last->tail = cons;

}

last = cons;

// No outstanding literals

Assert(deferredLiteralNode.length == 0);

}

}

if (last == 0)

node = FinalTerm(node, &deferredLiteralNode);

else

last->head = FinalTerm(last->head, &deferredLiteralNode);

// No outstanding literals

Assert(deferredLiteralNode.length == 0);

return node;

}

//

// Terms

//

template <typename P, const bool IsLiteral>

Node\* Parser<P, IsLiteral>::NewLoopNode(CharCount lower, CharCountOrFlag upper, bool isGreedy, Node\* body)

{

//

// NOTE: We'd like to represent r? (i.e. r{0,1}) as r|<empty> since the loop representation has high overhead.

// HOWEVER if r contains a group definition and could match empty then we must execute as a loop

// so that group bindings are correctly reset on no progress (e.g.: /(a\*)?/.exec("")). Thus we defer

// this optimization until pass 4 of the optimizer, at which point we know whether r could match empty.

//

if (lower == 1 && upper == 1)

return body;

else if (lower == 0 && upper == 0)

// Loop is equivalent to empty. If the loop body contains group definitions they will have already been

// counted towards the overall number of groups. The matcher will initialize their contents to

// undefined, and since the loop body would never execute the inner groups could never be updated from

// undefined.

return Anew(ctAllocator, SimpleNode, Node::Empty);

else

return Anew(ctAllocator, LoopNode, lower, upper, isGreedy, body);

}

template <typename P, const bool IsLiteral>

bool Parser<P, IsLiteral>::AtQuantifier()

{

// Could be terminating 0

switch (ECLookahead())

{

case '\*':

case '+':

case '?':

return true;

case '{':

{

CharCount lookahead = 1;

while (ECCanConsume(lookahead + 1) && standardEncodedChars->IsDigit(ECLookahead(lookahead)))

lookahead++;

if (lookahead == 1 || !ECCanConsume(lookahead + 1))

return false;

switch (ECLookahead(lookahead))

{

case ',':

lookahead++;

if (ECCanConsume(lookahead + 1) && ECLookahead(lookahead) == '}')

return true;

else

{

CharCount saved = lookahead;

while (ECCanConsume(lookahead + 1) && standardEncodedChars->IsDigit(ECLookahead(lookahead)))

lookahead++;

if (lookahead == saved)

return false;

return ECCanConsume(lookahead + 1) && ECLookahead(lookahead) == '}';

}

case '}':

return true;

default:

return false;

}

}

default:

return false;

}

}

template <typename P, const bool IsLiteral>

bool Parser<P, IsLiteral>::OptNonGreedy()

{

// Could be terminating 0

if (ECLookahead() != '?')

return true;

ECConsume();

return false;

}

template <typename P, const bool IsLiteral>

CharCount Parser<P, IsLiteral>::RepeatCount()

{

CharCount n = 0;

int digits = 0;

while (true)

{

// Could be terminating 0

EncodedChar ec = ECLookahead();

if (!standardEncodedChars->IsDigit(ec))

{

if (digits == 0)

Fail(JSERR\_RegExpSyntax);

return n;

}

if (n > MaxCharCount / 10)

Fail(JSERR\_RegExpSyntax);

n \*= 10;

if (n > MaxCharCount - standardEncodedChars->DigitValue(ec))

Fail(JSERR\_RegExpSyntax);

n += standardEncodedChars->DigitValue(ec);

digits++;

ECConsume();

}

}

template <typename P, const bool IsLiteral>

void Parser<P, IsLiteral>::TermPass0(int depth)

{

PROBE\_STACK(scriptContext, Js::Constants::MinStackRegex);

// Either we have a location at the start, or the end, never both. As in between it should have been cleared if surrogate pair

// Or must be cleared if we didn't perform the check

bool clearLocationIfPresent = this->tempLocationOfSurrogatePair != nullptr;

switch (ECLookahead())

{

case '^':

case '$':

ECConsume();

return;

case '\\':

ECConsume();

if (AtomEscapePass0())

return;

break;

case '(':

// Can't combine into a single codeunit because of group present

this->tempLocationOfSurrogatePair = nullptr;

this->codePointAtTempLocation = 0;

clearLocationIfPresent = false;

ECConsume();

switch (ECLookahead())

{

case '?':

if (!ECCanConsume(2))

Fail(JSERR\_RegExpSyntax);

switch (ECLookahead(1))

{

case '=':

case '!':

case ':':

ECConsume(2);

break;

default:

numGroups++;

break;

}

break;

default:

numGroups++;

break;

}

DisjunctionPass0(depth + 1);

ECMust(')', JSERR\_RegExpNoParen);

break;

case '.':

ECConsume();

break;

case '[':

ECConsume();

this->tempLocationOfSurrogatePair = nullptr;

this->codePointAtTempLocation = 0;

this->tempLocationOfRange = next;

CharacterClassPass0();

this->tempLocationOfRange = nullptr;

ECMust(']', JSERR\_RegExpNoBracket);

break;

case ')':

case '|':

Fail(JSERR\_RegExpSyntax);

break;

case ']':

case '}':

NextChar();

break;

case '\*':

case '+':

case '?':

case '{':

if (AtQuantifier())

Fail(JSERR\_RegExpBadQuant);

else

ECConsume();

break;

case 0:

if (IsEOF())

// Terminating 0

Fail(JSERR\_RegExpSyntax);

// else fall-through for embedded 0

default:

{

const EncodedChar\* current = next;

Char c = NextChar();

if (scriptContext->GetConfig()->IsES6UnicodeExtensionsEnabled())

{

TrackIfSurrogatePair(c, current, (uint32)(next - current));

}

// Closing '/' in literals should be caught explicitly

Assert(!IsLiteral || c != '/');

}

break;

}

if (clearLocationIfPresent && this->tempLocationOfSurrogatePair != nullptr)

{

this->tempLocationOfSurrogatePair = nullptr;

this->codePointAtTempLocation = 0;

}

if (AtQuantifier())

{

switch (ECLookahead())

{

case '\*':

case '+':

case '?':

ECConsume();

OptNonGreedy();

break;

case '{':

{

ECConsume();

CharCount lower = RepeatCount();

switch (ECLookahead())

{

case ',':

ECConsume();

if (ECLookahead() == '}')

{

ECConsume();

OptNonGreedy();

}

else

{

CharCount upper = RepeatCount();

if (upper < lower)

Fail(JSERR\_RegExpSyntax);

Assert(ECLookahead() == '}');

ECConsume();

OptNonGreedy();

}

break;

case '}':

ECConsume();

OptNonGreedy();

break;

default:

Assert(false);

break;

}

break;

}

default:

Assert(false);

break;

}

}

}

template <typename P, const bool IsLiteral>

Node\* Parser<P, IsLiteral>::TermPass1(MatchCharNode\* deferredCharNode, bool& previousSurrogatePart)

{

PROBE\_STACK(scriptContext, Js::Constants::MinStackRegex);

Node\* node = 0;

bool containsSurrogatePair = false;

switch (ECLookahead())

{

case '^':

ECConsume();

return Anew(ctAllocator, SimpleNode, Node::BOL); // No quantifier allowed

case '$':

ECConsume();

return Anew(ctAllocator, SimpleNode, Node::EOL); // No quantifier allowed

case '\\':

ECConsume();

if(SurrogatePairPass1(node, deferredCharNode, previousSurrogatePart))

{

break; // For quantifier

}

else if (AtomEscapePass1(node, deferredCharNode, previousSurrogatePart))

{

return node; // No quantifier allowed

}

break; // else: fall-through for opt quantifier

case '(':

ECConsume();

switch (ECLookahead())

{

case '?':

switch (ECLookahead(1))

{

case '=':

ECConsume(2); // ?=

node = DisjunctionPass1();

Assert(ECLookahead() == ')');

ECConsume(); // )

node = Anew(ctAllocator, AssertionNode, false, node);

break; // As per Annex B, allow this to be quantifiable

case '!':

ECConsume(2); // ?!

node = DisjunctionPass1();

Assert(ECLookahead() == ')');

ECConsume(); // )

node = Anew(ctAllocator, AssertionNode, true, node);

break; // As per Annex B, allow this to be quantifiable

case ':':

ECConsume(2); // ?:

node = DisjunctionPass1();

Assert(ECLookahead() == ')');

ECConsume(); // )

break; // fall-through for opt quantifier

default:

{

// ? not yet consumed

int thisGroupId = nextGroupId++;

node = DisjunctionPass1();

Assert(ECLookahead() == ')');

ECConsume(); // )

node = Anew(ctAllocator, DefineGroupNode, thisGroupId, node);

break; // fall-through for opt quantifier

}

}

break;

default:

{

// next char not yet consumed

int thisGroupId = nextGroupId++;

node = DisjunctionPass1();

Assert(ECLookahead() == ')');

ECConsume(); // )

node = Anew(ctAllocator, DefineGroupNode, thisGroupId, node);

break; // fall-through for opt quantifier

}

}

break;

case '.':

{

ECConsume();

node = GetNodeWithValidCharacterSet('.');

break; // fall-through for opt quantifier

}

case '[':

ECConsume();

if (unicodeFlagPresent)

{

containsSurrogatePair = this->currentSurrogatePairNode != nullptr && this->currentSurrogatePairNode->rangeLocation == next;

}

node = containsSurrogatePair ? CharacterClassPass1<true>() : CharacterClassPass1<false>();

Assert(ECLookahead() == ']');

ECConsume(); // ]

break; // fall-through for opt quantifier

#if DBG

case ')':

case '|':

Assert(false);

break;

#endif

case ']':

// SPEC DEVIATION: This should be syntax error, instead accept as itself

deferredCharNode->cs[0] = NextChar();

node = deferredCharNode;

break; // fall-through for opt quantifier

#if DBG

case '\*':

case '+':

case '?':

AssertMsg(false, "Allowed only in the escaped form. These should be caught by TermPass0.");

break;

#endif

case 0:

if (IsEOF())

// Terminating 0

Fail(JSERR\_RegExpSyntax);

// else fall-through for embedded 0

default:

if(SurrogatePairPass1(node, deferredCharNode, previousSurrogatePart))

{

break; //For quantifier

}

else

{

deferredCharNode->cs[0] = NextChar();

node = deferredCharNode;

break; // fall-through for opt quantifier

}

}

Assert(node != 0);

if (AtQuantifier())

{

switch (ECLookahead())

{

case '\*':

if (node == deferredCharNode)

node = Anew(ctAllocator, MatchCharNode, \*deferredCharNode);

ECConsume();

return NewLoopNode(0, CharCountFlag, OptNonGreedy(), node);

case '+':

if (node == deferredCharNode)

node = Anew(ctAllocator, MatchCharNode, \*deferredCharNode);

ECConsume();

return NewLoopNode(1, CharCountFlag, OptNonGreedy(), node);

case '?':

if (node == deferredCharNode)

node = Anew(ctAllocator, MatchCharNode, \*deferredCharNode);

ECConsume();

return NewLoopNode(0, 1, OptNonGreedy(), node);

case '{':

{

if (node == deferredCharNode)

node = Anew(ctAllocator, MatchCharNode, \*deferredCharNode);

ECConsume();

CharCount lower = RepeatCount();

switch (ECLookahead())

{

case ',':

ECConsume();

if (ECLookahead() == '}')

{

ECConsume();

return NewLoopNode(lower, CharCountFlag, OptNonGreedy(), node);

}

else

{

CharCount upper = RepeatCount();

Assert(lower <= upper);

Assert(ECLookahead() == '}');

ECConsume(); // }

return NewLoopNode(lower, upper, OptNonGreedy(), node);

}

case '}':

ECConsume();

return NewLoopNode(lower, lower, OptNonGreedy(), node);

default:

Assert(false);

break;

}

break;

}

default:

Assert(false);

break;

}

}

return node;

}

#pragma warning(push)

#pragma warning(disable:4702) // unreachable code

template <typename P, const bool IsLiteral>

bool Parser<P, IsLiteral>::AtomEscapePass0()

{

EncodedChar ec = ECLookahead();

if (ec == 0 && IsEOF())

{

// Terminating 0

Fail(JSERR\_RegExpSyntax);

return false;

}

else if (standardEncodedChars->IsDigit(ec))

{

do

{

ECConsume();

}

while (standardEncodedChars->IsDigit(ECLookahead())); // terminating 0 is not a digit

return false;

}

else if (ECLookahead() == 'c')

{

if (standardEncodedChars->IsLetter(ECLookahead(1))) // terminating 0 is not a letter

ECConsume(2);

return false;

}

else

{

const EncodedChar \*current = next;

// An escaped '/' is ok

Char c = NextChar();

switch (c)

{

case 'b':

case 'B':

return true;

// case 'c': handled as special case above

case 'x':

if (ECCanConsume(2) &&

standardEncodedChars->IsHex(ECLookahead(0)) &&

standardEncodedChars->IsHex(ECLookahead(1)))

ECConsume(2);

break;

case 'u':

bool surrogateEncountered = false;

int lengthOfSurrogate = TryParseExtendedUnicodeEscape(c, surrogateEncountered, true);

if (lengthOfSurrogate > 0)

{

if (surrogateEncountered)

{

// If we don't have an allocator, we don't create nodes

// Asserts in place as extra checks for when we do have an allocator

Assert(this->ctAllocator == nullptr || this->currentSurrogatePairNode != nullptr);

Assert(this->ctAllocator == nullptr || current == this->currentSurrogatePairNode->location);

ECConsume(lengthOfSurrogate);

}

//Don't fall through

break;

}

else if (ECCanConsume(4) &&

standardEncodedChars->IsHex(ECLookahead(0)) &&

standardEncodedChars->IsHex(ECLookahead(1)) &&

standardEncodedChars->IsHex(ECLookahead(2)) &&

standardEncodedChars->IsHex(ECLookahead(3)))

{

if (this->scriptContext->GetConfig()->IsES6UnicodeExtensionsEnabled())

{

codepoint\_t value = (standardEncodedChars->DigitValue(ECLookahead(0)) << 12) |

(standardEncodedChars->DigitValue(ECLookahead(1)) << 8) |

(standardEncodedChars->DigitValue(ECLookahead(2)) << 4) |

(standardEncodedChars->DigitValue(ECLookahead(3)));

TrackIfSurrogatePair(value, (next - 1), 5);

}

ECConsume(4);

}

break;

}

// embedded 0 is ok

return false;

}

}

#pragma warning(pop)

template <typename P, const bool IsLiteral>

bool Parser<P, IsLiteral>::AtomEscapePass1(Node\*& node, MatchCharNode\* deferredCharNode, bool& previousSurrogatePart)

{

Assert(!IsEOF()); // checked for terminating 0 in pass 0

if (standardEncodedChars->IsDigit(ECLookahead()))

{

// As per Annex B, allow octal escapes as well as group references, disambiguate based on known

// number of groups.

if (ECLookahead() == '0')

{

// fall through for octal

}

else

{

// Could be a group reference, but only if between 1 and 5 digits which resolve to a valid group number

int n = 0;

CharCount digits = 0;

do

{

n = n \* 10 + (int)standardEncodedChars->DigitValue(ECLookahead(digits));

digits++;

}

while (digits < 5 && ECCanConsume(digits + 1) && standardEncodedChars->IsDigit(ECLookahead(digits)));

if (n >= numGroups || ECCanConsume(digits + 1) && standardEncodedChars->IsDigit(ECLookahead(digits)))

{

if (standardEncodedChars->IsOctal(ECLookahead()))

{

// fall through for octal

}

else

{

// \8 and \9 are identity escapes

deferredCharNode->cs[0] = Chars<EncodedChar>::CTW(ECLookahead());

ECConsume();

node = deferredCharNode;

return false; // not an assertion

}

}

else

{

ECConsume(digits);

node = Anew(ctAllocator, MatchGroupNode, n);

return false; // not an assertion

}

}

// Must be between 1 and 3 octal digits

Assert(standardEncodedChars->IsOctal(ECLookahead())); // terminating 0 is not an octal

uint n = 0;

CharCount digits = 0;

do

{

uint m = n \* 8 + standardEncodedChars->DigitValue(ECLookahead());

if (m > Chars<uint8>::MaxUChar) // Regex octal codes only support single byte (ASCII) characters.

break;

n = m;

ECConsume();

digits++;

}

while (digits < 3 && standardEncodedChars->IsOctal(ECLookahead())); // terminating 0 is not an octal

deferredCharNode->cs[0] = UTC((UChar)n);

node = deferredCharNode;

return false; // not an assertion

}

else if (ECLookahead() == 'c')

{

Char c;

if (standardEncodedChars->IsLetter(ECLookahead(1))) // terminating 0 is not a letter

{

c = UTC(Chars<EncodedChar>::CTU(ECLookahead(1)) % 32);

ECConsume(2);

}

else

{

// SPEC DEVIATION: For non-letters or EOF, take the leading '\' to be itself, and

// don't consume the 'c' or letter.

c = '\\';

}

deferredCharNode->cs[0] = c;

node = deferredCharNode;

return false; // not an assertion

}

else

{

Char c = NextChar();

switch (c)

{

case 'b':

node = Anew(ctAllocator, WordBoundaryNode, false);

return true; // Is an assertion

case 'B':

node = Anew(ctAllocator, WordBoundaryNode, true);

return true; // Is an assertion

case 'f':

c = '\f';

break; // fall-through for identity escape

case 'n':

c = '\n';

break; // fall-through for identity escape

case 'r':

c = '\r';

break; // fall-through for identity escape

case 't':

c = '\t';

break; // fall-through for identity escape

case 'v':

c = '\v';

break; // fall-through for identity escape

case 'd':

{

MatchSetNode \*setNode = Anew(ctAllocator, MatchSetNode, false, false);

standardChars->SetDigits(ctAllocator, setNode->set);

node = setNode;

return false; // not an assertion

}

case 'D':

{

node = GetNodeWithValidCharacterSet('D');

return false; // not an assertion

}

case 's':

{

MatchSetNode \*setNode = Anew(ctAllocator, MatchSetNode, false, false);

standardChars->SetWhitespace(ctAllocator, setNode->set);

node = setNode;

return false; // not an assertion

}

case 'S':

{

node = GetNodeWithValidCharacterSet('S');

return false; // not an assertion

}

case 'w':

{

MatchSetNode \*setNode = Anew(ctAllocator, MatchSetNode, false, false);

standardChars->SetWordChars(ctAllocator, setNode->set);

node = setNode;

return false; // not an assertion

}

case 'W':

{

node = GetNodeWithValidCharacterSet('W');

return false; // not an assertion

}

// case 'c': handled as special case above

case 'x':

if (ECCanConsume(2) &&

standardEncodedChars->IsHex(ECLookahead(0)) &&

standardEncodedChars->IsHex(ECLookahead(1)))

{

c = UTC((standardEncodedChars->DigitValue(ECLookahead(0)) << 4) |

(standardEncodedChars->DigitValue(ECLookahead(1))));

ECConsume(2);

// fall-through for identity escape

}

// Take to be identity escape if ill-formed as per Annex B

break;

case 'u':

if (unicodeFlagPresent && TryParseExtendedUnicodeEscape(c, previousSurrogatePart) > 0)

break;

else if (ECCanConsume(4) &&

standardEncodedChars->IsHex(ECLookahead(0)) &&

standardEncodedChars->IsHex(ECLookahead(1)) &&

standardEncodedChars->IsHex(ECLookahead(2)) &&

standardEncodedChars->IsHex(ECLookahead(3)))

{

c = UTC((standardEncodedChars->DigitValue(ECLookahead(0)) << 12) |

(standardEncodedChars->DigitValue(ECLookahead(1)) << 8) |

(standardEncodedChars->DigitValue(ECLookahead(2)) << 4) |

(standardEncodedChars->DigitValue(ECLookahead(3))));

ECConsume(4);

// fall-through for identity escape

}

// Take to be identity escape if ill-formed as per Annex B

break;

default:

// As per Annex B, allow anything other than newlines and above. Embedded 0 is ok

break;

}

// Must be an identity escape

deferredCharNode->cs[0] = c;

node = deferredCharNode;

return false; // not an assertion

}

}

template <typename P, const bool IsLiteral>

bool Parser<P, IsLiteral>::SurrogatePairPass1(Node\*& node, MatchCharNode\* deferredCharNode, bool& previousSurrogatePart)

{

if (!this->scriptContext->GetConfig()->IsES6UnicodeExtensionsEnabled() || !unicodeFlagPresent)

{

return false;

}

if (this->currentSurrogatePairNode != nullptr && this->currentSurrogatePairNode->location == this->next)

{

AssertMsg(!this->currentSurrogatePairNode->IsInsideRange(), "Should not be calling this pass if we are currently inside a range.");

wchar\_t lower, upper;

uint tableIndex = 0, actualHigh = 0;

codepoint\_t equivClass[CaseInsensitive::EquivClassSize];

if (caseInsensitiveFlagPresent && CaseInsensitive::RangeToEquivClass(tableIndex, this->currentSurrogatePairNode->value, this->currentSurrogatePairNode->value, actualHigh, equivClass))

{

Node \*equivNode[CaseInsensitive::EquivClassSize];

int indexForNextNode = 0;

for (int i = 0; i < CaseInsensitive::EquivClassSize; i++)

{

bool alreadyAdded = false;

for (int j = 0; j < i; j++)

{

if (equivClass[i] == equivClass[j])

{

alreadyAdded = true;

break;

}

}

if (!alreadyAdded)

{

if (Js::NumberUtilities::IsInSupplementaryPlane(equivClass[i]))

{

Js::NumberUtilities::CodePointAsSurrogatePair(equivClass[i], &lower, &upper);

equivNode[indexForNextNode] = CreateSurrogatePairAtom(lower, upper);

}

else

{

equivNode[indexForNextNode] = Anew(ctAllocator, MatchCharNode, (Char)equivClass[i]);

}

indexForNextNode ++;

}

}

Assert(indexForNextNode > 0);

if (indexForNextNode == 1)

{

node = equivNode[0];

}

else

{

AltNode \*altNode = Anew(ctAllocator, AltNode, equivNode[0], nullptr);

AltNode \*altNodeTail = altNode;

for (int i = 1; i < indexForNextNode; i++)

{

altNodeTail->tail = Anew(ctAllocator, AltNode, equivNode[i], nullptr);

altNodeTail = altNodeTail->tail;

}

node = altNode;

}

}

else

{

Js::NumberUtilities::CodePointAsSurrogatePair(this->currentSurrogatePairNode->value, &lower, &upper);

node = CreateSurrogatePairAtom(lower, upper);

}

previousSurrogatePart = false;

Assert(ECCanConsume(this->currentSurrogatePairNode->length));

ECConsumeMultiUnit(this->currentSurrogatePairNode->length);

RestoreMultiUnits(this->currentSurrogatePairNode->multiUnits);

this->currentSurrogatePairNode = this->currentSurrogatePairNode->next;

return true;

}

return false;

}

//

// Classes

//

template <typename P, const bool IsLiteral>

bool Parser<P, IsLiteral>::AtSecondSingletonClassAtom()

{

Assert(ECLookahead() == '-');

if (ECLookahead(1) == '\\')

{

switch (ECLookahead(2))

{

case 'd':

case 'D':

case 's':

case 'S':

case 'w':

case 'W':

// These all denote non-singleton sets

return false;

default:

// fall-through for singleton

break;

}

}

return true;

}

template <typename P, const bool IsLiteral>

void Parser<P, IsLiteral>::CharacterClassPass0()

{

// Could be terminating 0

if (ECLookahead() == '^')

ECConsume();

EncodedChar nextChar = ECLookahead();

const EncodedChar\* current;

codepoint\_t lastCodepoint = INVALID\_CODEPOINT;

codepoint\_t pendingRangeStart = INVALID\_CODEPOINT;

codepoint\_t pendingRangeEnd = INVALID\_CODEPOINT;

bool previousSurrogatePart = false;

while(nextChar != ']')

{

current = next;

if (nextChar == '\0' && IsEOF())

{

// Report as unclosed '['

Fail(JSERR\_RegExpNoBracket);

return;

} // Otherwise embedded '\0' is ok

else if (nextChar == '\\')

{

// Consume, as classescapepass0 expects for it to be consumed

Char outChar = NextChar();

// If previousSurrogatePart = true upon leaving this method, then we are going to pass through here twice

// This is because \u{} escape sequence was encountered that is actually 2 characters, the second time we will pass consuming entire character

if (ClassEscapePass0(outChar, previousSurrogatePart))

{

lastCodepoint = outChar;

}

else

{

// Last codepoint isn't a singleton, so no codepoint tracking for the sake of ranges is needed.

lastCodepoint = INVALID\_CODEPOINT;

// Unless we have a possible range end, cancel our range tracking.

if (pendingRangeEnd == INVALID\_CODEPOINT)

{

pendingRangeStart = INVALID\_CODEPOINT;

}

}

}

else if (nextChar == '-')

{

if (pendingRangeStart != INVALID\_CODEPOINT || lastCodepoint == INVALID\_CODEPOINT)

{

// '-' is the upper part of the range, with pendingRangeStart codepoint is the lower. Set lastCodePoint to '-' to check at the end of the while statement

// OR

// '-' is just a char, consume it and set as last char

lastCodepoint = NextChar();

}

else

{

pendingRangeStart = this->next == this->positionAfterLastSurrogate

? this->valueOfLastSurrogate

: lastCodepoint;

lastCodepoint = INVALID\_CODEPOINT;

NextChar();

}

// If we have a pattern of the form [\ud800-\udfff], we need this to be interpreted as a range.

// In order to achieve this, the two variables that we use to track surrogate pairs, namely

// tempLocationOfSurrogatePair and previousSurrogatePart, need to be in a certain state.

//

// We need to reset tempLocationOfSurrogatePair as it points to the first Unicode escape (\ud800)

// when we're here. We need to clear it in order not to have a surrogate pair when we process the

// second escape (\udfff).

//

// previousSurrogatePart is used when we have a code point in the \u{...} extended format and the

// character is in a supplementary plane. However, there is no need to change its value here. When

// such an escape sequence is encountered, the first call to ClassEscapePass0() sets the variable

// to true, but it rewinds the input back to the beginning of the escape sequence. The next

// iteration of the loop here will again call ClassEscape0() with the same character and the

// variable will this time be set to false. Therefore, the variable will always be false here.

tempLocationOfSurrogatePair = nullptr;

Assert(!previousSurrogatePart);

}

else

{

lastCodepoint = NextChar();

if (scriptContext->GetConfig()->IsES6UnicodeExtensionsEnabled())

{

TrackIfSurrogatePair(lastCodepoint, current, (uint32)(next - current));

}

}

if (!scriptContext->GetConfig()->IsES6UnicodeExtensionsEnabled())

{

// This is much easier to handle.

if (pendingRangeStart != INVALID\_CODEPOINT && lastCodepoint != INVALID\_CODEPOINT)

{

Assert(pendingRangeStart < 0x10000);

Assert(pendingRangeEnd == INVALID\_CODEPOINT);

if (pendingRangeStart > lastCodepoint)

{

Fail(JSERR\_RegExpBadRange);

}

pendingRangeStart = lastCodepoint = INVALID\_CODEPOINT;

}

}

// We have a candidate for range end, and we have a range start.

else if (pendingRangeStart != INVALID\_CODEPOINT && (lastCodepoint != INVALID\_CODEPOINT || pendingRangeEnd != INVALID\_CODEPOINT))

{

// The following will be true at the end of each surrogate pair parse.

// Note, the escape sequence \u{} is a two time parse, so this will be true on the second time around.

if (this->next == this->positionAfterLastSurrogate)

{

lastCodepoint = this->valueOfLastSurrogate;

Assert(!previousSurrogatePart);

pendingRangeEnd = lastCodepoint;

lastCodepoint = INVALID\_CODEPOINT;

}

// If we the next character is the end of range ']', then we can't have a surrogate pair.

// The current character is the range end, if we don't already have a candidate.

else if (ECLookahead() == ']' && pendingRangeEnd == INVALID\_CODEPOINT)

{

pendingRangeEnd = lastCodepoint;

}

//If we get here, and pendingRangeEnd is set. Then one of the above has caused it to be set, or the previous iteration of the loop.

if (pendingRangeEnd != INVALID\_CODEPOINT)

{

wchar\_t leftSingleChar, rightSingleChar, ignore;

if (pendingRangeStart >= 0x10000)

{

Js::NumberUtilities::CodePointAsSurrogatePair(pendingRangeStart, &ignore, &leftSingleChar);

}

else

{

leftSingleChar = (wchar\_t)pendingRangeStart;

}

if (pendingRangeEnd >= 0x10000)

{

Js::NumberUtilities::CodePointAsSurrogatePair(pendingRangeEnd, &rightSingleChar, &ignore);

}

else

{

rightSingleChar = (wchar\_t)pendingRangeEnd;

}

// Here it is a bit tricky, we don't know if we have a unicode option specified.

// If it is, then \ud800\udc00 - \ud800\udc01 is valid, otherwise invalid.

if (pendingRangeStart < 0x10000 && pendingRangeEnd < 0x10000 && pendingRangeStart > pendingRangeEnd)

{

Fail(JSERR\_RegExpBadRange);

}

else

{

if(leftSingleChar > rightSingleChar)

{

DeferredFailIfNotUnicode(JSERR\_RegExpBadRange);

}

if (pendingRangeStart > pendingRangeEnd)

{

DeferredFailIfUnicode(JSERR\_RegExpBadRange);

}

}

pendingRangeStart = pendingRangeEnd = INVALID\_CODEPOINT;

}

// The current char < 0x10000 is a candidate for the range end, but we need to iterate one more time.

else

{

pendingRangeEnd = lastCodepoint;

}

}

nextChar = ECLookahead();

}

// We should never have a pendingRangeEnd set when we exit the loop

Assert(pendingRangeEnd == INVALID\_CODEPOINT);

}

template <typename P, const bool IsLiteral>

template <bool containsSurrogates>

Node\* Parser<P, IsLiteral>::CharacterClassPass1()

{

Assert(containsSurrogates ? unicodeFlagPresent : true);

CharSet<codepoint\_t> codePointSet;

MatchSetNode defferedSetNode(false, false);

MatchCharNode defferedCharNode(0);

bool isNegation = false;

if (ECLookahead() == '^')

{

isNegation = true;

ECConsume();

}

// We aren't expecting any terminating null characters, only embedded ones that should treated as valid characters.

// CharacterClassPass0 should have taken care of terminating null.

codepoint\_t pendingCodePoint = INVALID\_CODEPOINT;

codepoint\_t pendingRangeStart = INVALID\_CODEPOINT;

EncodedChar nextChar = ECLookahead();

bool previousWasASurrogate = false;

while(nextChar != ']')

{

codepoint\_t codePointToSet = INVALID\_CODEPOINT;

// Consume ahead of time if we have two backslashes, both cases below (previously Tracked surrogate pair, and ClassEscapePass1) assume it is.

if (nextChar == '\\')

{

ECConsume();

}

// These if-blocks are the logical ClassAtomPass1, they weren't grouped into a method to simplify dealing with multiple out parameters.

if (containsSurrogates && this->currentSurrogatePairNode != nullptr && this->currentSurrogatePairNode->location == this->next)

{

codePointToSet = pendingCodePoint;

pendingCodePoint = this->currentSurrogatePairNode->value;

Assert(ECCanConsume(this->currentSurrogatePairNode->length));

ECConsumeMultiUnit(this->currentSurrogatePairNode->length);

RestoreMultiUnits(this->currentSurrogatePairNode->multiUnits);

this->currentSurrogatePairNode = this->currentSurrogatePairNode->next;

}

else if (nextChar == '\\')

{

Node\* returnedNode = ClassEscapePass1(&defferedCharNode, &defferedSetNode, previousWasASurrogate);

if (returnedNode->tag == Node::MatchSet)

{

codePointToSet = pendingCodePoint;

pendingCodePoint = INVALID\_CODEPOINT;

if (pendingRangeStart != INVALID\_CODEPOINT)

{

codePointSet.Set(ctAllocator, '-');

}

pendingRangeStart = INVALID\_CODEPOINT;

codePointSet.UnionInPlace(ctAllocator, defferedSetNode.set);

}

else

{

// Just a character

codePointToSet = pendingCodePoint;

pendingCodePoint = defferedCharNode.cs[0];

}

}

else if (nextChar == '-')

{

if (pendingRangeStart != INVALID\_CODEPOINT || pendingCodePoint == INVALID\_CODEPOINT || ECLookahead(1) == ']')

{

// - is just a char, or end of a range.

codePointToSet = pendingCodePoint;

pendingCodePoint = '-';

ECConsume();

}

else

{

pendingRangeStart = pendingCodePoint;

ECConsume();

}

}

else

{

// Just a character, consume it

codePointToSet = pendingCodePoint;

pendingCodePoint = NextChar();

}

if (codePointToSet != INVALID\_CODEPOINT)

{

if (pendingRangeStart != INVALID\_CODEPOINT)

{

if (pendingRangeStart > pendingCodePoint)

{

//We have no unicodeFlag, but current range contains surrogates, thus we may end up having to throw a "Syntax" error here

//This breaks the notion of Pass0 check for valid syntax, because we don't know if we have a unicode option

Assert(!unicodeFlagPresent);

Fail(JSERR\_RegExpBadRange);

}

codePointSet.SetRange(ctAllocator, pendingRangeStart, pendingCodePoint);

pendingRangeStart = pendingCodePoint = INVALID\_CODEPOINT;

}

else

{

codePointSet.Set(ctAllocator, codePointToSet);

}

}

nextChar = ECLookahead();

}

if (pendingCodePoint != INVALID\_CODEPOINT)

{

codePointSet.Set(ctAllocator, pendingCodePoint);

}

// At this point, we have a complete set of codepoints representing the range.

// Before performing translation of any kind, we need to do some case filling.

// At the point of this comment, there are no case mappings going cross-plane between simple

// characters (< 0x10000) and supplementary characters (>= 0x10000)

// However, it might still be the case, and this has to be handled.

// On the other hand, we don't want to prevent optimizations that expect non-casefolded sets from happening.

// At least for simple characters.

// The simple case, is when the unicode flag isn't specified, we can go ahead and return the simple set.

// Negations and case mappings will be handled later.

if (!unicodeFlagPresent)

{

Assert(codePointSet.SimpleCharCount() == codePointSet.Count());

MatchSetNode \*simpleToReturn = Anew(ctAllocator, MatchSetNode, isNegation);

codePointSet.CloneSimpleCharsTo(ctAllocator, simpleToReturn->set);

return simpleToReturn;

}

// Everything past here must be under the flag

Assert(scriptContext->GetConfig()->IsES6UnicodeExtensionsEnabled());

if (codePointSet.IsEmpty())

{

return Anew(ctAllocator, MatchSetNode, false, false);

}

Node\* prefixNode = nullptr;

Node\* suffixNode = nullptr;

CharSet<codepoint\_t> \*toUseForTranslation = &codePointSet;

// If a singleton, return a simple character

bool isSingleton = !this->caseInsensitiveFlagPresent && !isNegation && codePointSet.IsSingleton();

if (isSingleton)

{

codepoint\_t singleton = codePointSet.Singleton();

Node\* toReturn = nullptr;

if (singleton < 0x10000)

{

toReturn = Anew(ctAllocator, MatchCharNode, (wchar\_t)singleton);

}

else

{

Assert(unicodeFlagPresent);

wchar\_t lowerSurrogate, upperSurrogate;

Js::NumberUtilities::CodePointAsSurrogatePair(singleton, &lowerSurrogate, &upperSurrogate);

toReturn = CreateSurrogatePairAtom(lowerSurrogate, upperSurrogate);

}

codePointSet.Clear(ctAllocator);

return toReturn;

}

if (!this->caseInsensitiveFlagPresent)

{

// If negation, we want to complement the simple chars.

// When a set is negated, optimizations skip checking if applicable, so we can go ahead and negate it here.

CharSet<codepoint\_t> negatedSet;

if (isNegation)

{

// Complement all characters, and use it as the set toTranslate

codePointSet.ToComplement(ctAllocator, negatedSet);

}

toUseForTranslation = isNegation ? &negatedSet : &codePointSet;

if (isNegation)

{

// Clear this, as we will no longer need this.

codePointSet.FreeBody(ctAllocator);

}

}

else

{

CharSet<codepoint\_t> caseEquivalent;

codePointSet.ToEquivClass(ctAllocator, caseEquivalent);

// Equiv set can't have a reduced count of chars

Assert(caseEquivalent.Count() >= codePointSet.Count());

// Here we have a regex that has both case insensitive and unicode options.

// The range might also be negated. If it is negated, we can go ahead and negate

// the entire set as well as fill in cases, as optimizations wouldn't kick in anyways.

if (isNegation)

{

codePointSet.Clear(ctAllocator);

caseEquivalent.ToComplement(ctAllocator, codePointSet);

caseEquivalent.FreeBody(ctAllocator);

}

else

{

codePointSet.CloneFrom(ctAllocator, caseEquivalent);

}

Assert(toUseForTranslation == &codePointSet);

}

uint totalCodePointsCount = toUseForTranslation->Count();

uint simpleCharsCount = toUseForTranslation->SimpleCharCount();

if (totalCodePointsCount == simpleCharsCount)

{

MatchSetNode \*simpleToReturn = Anew(ctAllocator, MatchSetNode, isNegation);

toUseForTranslation->CloneSimpleCharsTo(ctAllocator, simpleToReturn->set);

return simpleToReturn;

}

if (simpleCharsCount > 0)

{

if (!toUseForTranslation->ContainSurrogateCodeUnits())

{

MatchSetNode \*node = Anew(ctAllocator, MatchSetNode, false, false);

toUseForTranslation->CloneSimpleCharsTo(ctAllocator, node->set);

prefixNode = node;

}

else

{

MatchSetNode \*node = Anew(ctAllocator, MatchSetNode, false, false);

toUseForTranslation->CloneNonSurrogateCodeUnitsTo(ctAllocator, node->set);

prefixNode = node;

node = Anew(ctAllocator, MatchSetNode, false, false);

toUseForTranslation->CloneSurrogateCodeUnitsTo(ctAllocator, node->set);

suffixNode = node;

}

}

Assert(unicodeFlagPresent);

AltNode \*headToReturn = prefixNode == nullptr ? nullptr : Anew(ctAllocator, AltNode, prefixNode, nullptr);

AltNode \*currentTail = headToReturn;

codepoint\_t charRangeSearchIndex = 0x10000, lowerCharOfRange = 0, upperCharOfRange = 0;

while (toUseForTranslation->GetNextRange(charRangeSearchIndex, &lowerCharOfRange, &upperCharOfRange))

{

if (lowerCharOfRange == upperCharOfRange)

{

currentTail = this->AppendSurrogatePairToDisjunction(lowerCharOfRange, currentTail);

}

else

{

currentTail = this->AppendSurrogateRangeToDisjunction(lowerCharOfRange, upperCharOfRange, currentTail);

}

if (headToReturn == nullptr)

{

headToReturn = currentTail;

}

AnalysisAssert(currentTail != nullptr);

while (currentTail->tail != nullptr)

{

currentTail = currentTail->tail;

}

charRangeSearchIndex = upperCharOfRange + 1;

}

if (suffixNode != nullptr)

{

currentTail->tail = Anew(ctAllocator, AltNode, suffixNode, nullptr);

}

toUseForTranslation->Clear(ctAllocator);

if (headToReturn != nullptr && headToReturn->tail == nullptr)

{

return headToReturn->head;

}

return headToReturn;

}

#pragma warning(push)

#pragma warning(disable:4702) // unreachable code

template <typename P, const bool IsLiteral>

bool Parser<P, IsLiteral>::ClassEscapePass0(Char& singleton, bool& previousSurrogatePart)

{

// Could be terminating 0

EncodedChar ec = ECLookahead();

if (ec == 0 && IsEOF())

{

Fail(JSERR\_RegExpSyntax);

return false;

}

else if (standardEncodedChars->IsOctal(ec))

{

uint n = 0;

CharCount digits = 0;

do

{

uint m = n \* 8 + standardEncodedChars->DigitValue(ECLookahead());

if (m > Chars<uint8>::MaxUChar) //Regex octal codes only support single byte (ASCII) characters.

break;

n = m;

ECConsume();

digits++;

}

while (digits < 3 && standardEncodedChars->IsOctal(ECLookahead())); // terminating 0 is not octal

singleton = UTC((UChar)n);

// Clear possible pair

this->tempLocationOfSurrogatePair = nullptr;

return true;

}

else

{

const EncodedChar\* location = this->tempLocationOfSurrogatePair;

// Clear it for now, otherwise to many branches to clear it on.

this->tempLocationOfSurrogatePair = nullptr;

// An escaped '/' is ok

Char c = NextChar();

switch (c)

{

case 'b':

singleton = '\b';

return true;

case 'f':

singleton = '\f';

return true;

case 'n':

singleton = '\n';

return true;

case 'r':

singleton = '\r';

return true;

case 't':

singleton = '\t';

return true;

case 'v':

singleton = '\v';

return true;

case 'd':

case 'D':

case 's':

case 'S':

case 'w':

case 'W':

return false;

case 'c':

if (standardEncodedChars->IsLetter(ECLookahead())) // terminating 0 is not a letter

{

singleton = UTC(Chars<EncodedChar>::CTU(ECLookahead()) % 32);

ECConsume();

}

else

{

if (!IsEOF())

{

EncodedChar ec = ECLookahead();

switch (ec)

{

case '-':

case ']':

singleton = c;

break;

default:

singleton = UTC(Chars<EncodedChar>::CTU(ec) % 32);

ECConsume();

break;

}

}

else

singleton = c;

}

return true;

case 'x':

if (ECCanConsume(2) &&

standardEncodedChars->IsHex(ECLookahead(0)) &&

standardEncodedChars->IsHex(ECLookahead(1)))

{

singleton = UTC((standardEncodedChars->DigitValue(ECLookahead(0)) << 4) |

(standardEncodedChars->DigitValue(ECLookahead(1))));

ECConsume(2);

}

else

singleton = c;

return true;

case 'u':

this->tempLocationOfSurrogatePair = location;

if (this->TryParseExtendedUnicodeEscape(singleton, previousSurrogatePart, true) > 0)

return true;

else if (ECCanConsume(4) &&

standardEncodedChars->IsHex(ECLookahead(0)) &&

standardEncodedChars->IsHex(ECLookahead(1)) &&

standardEncodedChars->IsHex(ECLookahead(2)) &&

standardEncodedChars->IsHex(ECLookahead(3)))

{

singleton = UTC((standardEncodedChars->DigitValue(ECLookahead(0)) << 12) |

(standardEncodedChars->DigitValue(ECLookahead(1)) << 8) |

(standardEncodedChars->DigitValue(ECLookahead(2)) << 4) |

(standardEncodedChars->DigitValue(ECLookahead(3))));

if (this->scriptContext->GetConfig()->IsES6UnicodeExtensionsEnabled())

{

// Current location

TrackIfSurrogatePair(singleton, (next - 1), 5);

}

// The above if statement, if true, will clear tempLocationOfSurrogatePair if needs to.

ECConsume(4);

}

else

singleton = c;

return true;

default:

// embedded 0 is ok

singleton = c;

return true;

}

}

}

#pragma warning(pop)

template <typename P, const bool IsLiteral>

Node\* Parser<P, IsLiteral>::ClassEscapePass1(MatchCharNode\* deferredCharNode, MatchSetNode\* deferredSetNode, bool& previousSurrogatePart)

{

// Checked for terminating 0 is pass 0

Assert(!IsEOF());

if (standardEncodedChars->IsOctal(ECLookahead()))

{

// As per Annex B, allow octal escapes instead of just \0 (and \8 and \9 are identity escapes).

// Must be between 1 and 3 octal digits.

uint n = 0;

CharCount digits = 0;

do

{

uint m = n \* 8 + standardEncodedChars->DigitValue(ECLookahead());

if (m > Chars<uint8>::MaxUChar) //Regex octal codes only support single byte (ASCII) characters.

break;

n = m;

ECConsume();

digits++;

}

while (digits < 3 && standardEncodedChars->IsOctal(ECLookahead())); // terminating 0 is not octal

deferredCharNode->cs[0] = UTC((UChar)n);

return deferredCharNode;

}

else

{

Char c = NextChar();

switch (c)

{

case 'b':

c = '\b';

break; // fall-through for identity escape

case 'f':

c = '\f';

break; // fall-through for identity escape

case 'n':

c = '\n';

break; // fall-through for identity escape

case 'r':

c = '\r';

break; // fall-through for identity escape

case 't':

c = '\t';

break; // fall-through for identity escape

case 'v':

c = '\v';

break; // fall-through for identity escape

case 'd':

standardChars->SetDigits(ctAllocator, deferredSetNode->set);

return deferredSetNode;

case 'D':

standardChars->SetNonDigits(ctAllocator, deferredSetNode->set);

return deferredSetNode;

case 's':

standardChars->SetWhitespace(ctAllocator, deferredSetNode->set);

return deferredSetNode;

case 'S':

standardChars->SetNonWhitespace(ctAllocator, deferredSetNode->set);

return deferredSetNode;

case 'w':

standardChars->SetWordChars(ctAllocator, deferredSetNode->set);

return deferredSetNode;

case 'W':

standardChars->SetNonWordChars(ctAllocator, deferredSetNode->set);

return deferredSetNode;

case 'c':

if (standardEncodedChars->IsLetter(ECLookahead())) // terminating 0 is not a letter

{

c = UTC(Chars<EncodedChar>::CTU(ECLookahead()) % 32);

ECConsume();

// fall-through for identity escape

}

else

{

// SPEC DEVIATION: For non-letters, still take lower 5 bits, e.g. [\c1] == [\x11].

// However, '-', ']', and EOF make the \c just a 'c'.

if (!IsEOF())

{

EncodedChar ec = ECLookahead();

switch (ec)

{

case '-':

case ']':

// fall-through for identity escape with 'c'

break;

default:

c = UTC(Chars<EncodedChar>::CTU(ec) % 32);

ECConsume();

// fall-through for identity escape

break;

}

}

// else: fall-through for identity escape with 'c'

}

break;

case 'x':

if (ECCanConsume(2) &&

standardEncodedChars->IsHex(ECLookahead(0)) &&

standardEncodedChars->IsHex(ECLookahead(1)))

{

c = UTC((standardEncodedChars->DigitValue(ECLookahead(0)) << 4) |

(standardEncodedChars->DigitValue(ECLookahead(1))));

ECConsume(2);

// fall-through for identity escape

}

// Take to be identity escape if ill-formed as per Annex B

break;

case 'u':

if (unicodeFlagPresent && TryParseExtendedUnicodeEscape(c, previousSurrogatePart) > 0)

break;

else if (ECCanConsume(4) &&

standardEncodedChars->IsHex(ECLookahead(0)) &&

standardEncodedChars->IsHex(ECLookahead(1)) &&

standardEncodedChars->IsHex(ECLookahead(2)) &&

standardEncodedChars->IsHex(ECLookahead(3)))

{

c = UTC((standardEncodedChars->DigitValue(ECLookahead(0)) << 12) |

(standardEncodedChars->DigitValue(ECLookahead(1)) << 8) |

(standardEncodedChars->DigitValue(ECLookahead(2)) << 4) |

(standardEncodedChars->DigitValue(ECLookahead(3))));

ECConsume(4);

// fall-through for identity escape

}

// Take to be identity escape if ill-formed as per Annex B.

break;

default:

// As per Annex B, allow anything other than newlines and above. Embedded 0 is ok.

break;

}

// Must be an identity escape

deferredCharNode->cs[0] = c;

return deferredCharNode;

}

}

//

// Options

//

template <typename P, const bool IsLiteral>

void Parser<P, IsLiteral>::Options(RegexFlags& flags)

{

while (true)

{

// Could be terminating 0

EncodedChar ec = ECLookahead();

CharCount consume;

Char c;

if (ec == 0)

// Embedded 0 not valid

return;

else if (IsLiteral &&

ec == '\\' &&

ECCanConsume(6) &&

ECLookahead(1) == 'u' &&

standardEncodedChars->IsHex(ECLookahead(2)) &&

standardEncodedChars->IsHex(ECLookahead(3)) &&

standardEncodedChars->IsHex(ECLookahead(4)) &&

standardEncodedChars->IsHex(ECLookahead(5)))

{

if (scriptContext->GetConfig()->IsES6UnicodeExtensionsEnabled())

{

Fail(JSERR\_RegExpSyntax);

return;

}

else

{

uint32 n = (standardEncodedChars->DigitValue(ECLookahead(2)) << 12) |

(standardEncodedChars->DigitValue(ECLookahead(3)) << 8) |

(standardEncodedChars->DigitValue(ECLookahead(4)) << 4) |

(standardEncodedChars->DigitValue(ECLookahead(5)));

c = UTC(n);

consume = 6;

}

}

else

{

c = Chars<EncodedChar>::CTW(ec);

consume = 1;

}

switch (c) {

case 'i':

if ((flags & IgnoreCaseRegexFlag) != 0)

{

Fail(JSERR\_RegExpSyntax);

}

flags = (RegexFlags)(flags | IgnoreCaseRegexFlag);

break;

case 'g':

if ((flags & GlobalRegexFlag) != 0)

{

Fail(JSERR\_RegExpSyntax);

}

flags = (RegexFlags)(flags | GlobalRegexFlag);

break;

case 'm':

if ((flags & MultilineRegexFlag) != 0)

{

Fail(JSERR\_RegExpSyntax);

}

flags = (RegexFlags)(flags | MultilineRegexFlag);

break;

case 'u':

// If we don't have unicode enabled, fall through to default

if (scriptContext->GetConfig()->IsES6UnicodeExtensionsEnabled())

{

if ((flags & UnicodeRegexFlag) != 0)

{

Fail(JSERR\_RegExpSyntax);

}

flags = (RegexFlags)(flags | UnicodeRegexFlag);

// For telemetry

CHAKRATEL\_LANGSTATS\_INC\_LANGFEATURECOUNT(UnicodeRegexFlagCount, scriptContext);

break;

}

case 'y':

if (scriptContext->GetConfig()->IsES6RegExStickyEnabled())

{

if ((flags & StickyRegexFlag) != 0)

{

Fail(JSERR\_RegExpSyntax);

}

flags = (RegexFlags)(flags | StickyRegexFlag);

// For telemetry

CHAKRATEL\_LANGSTATS\_INC\_LANGFEATURECOUNT(StickyRegexFlagCount, scriptContext);

break;

}

default:

if (standardChars->IsWord(c))

{

// Outer context could never parse this character. Signal the syntax error as

// being part of the regex.

Fail(JSERR\_RegExpSyntax);

}

return;

}

ECConsume(consume);

}

}

//

// Entry points

//

template <typename P, const bool IsLiteral>

Node\* Parser<P, IsLiteral>::ParseDynamic

( const EncodedChar\* body

, const EncodedChar\* bodyLim

, const EncodedChar\* opts

, const EncodedChar\* optsLim

, RegexFlags& flags )

{

Assert(!IsLiteral);

Assert(body != 0);

Assert(bodyLim >= body && \*bodyLim == 0);

Assert(opts == 0 || (optsLim >= opts && \*optsLim == 0));

// Body, pass 0

SetPosition(body, bodyLim, true);

PatternPass0();

if (!IsEOF())

Fail(JSERR\_RegExpSyntax);

// Options

if (opts != 0)

{

SetPosition(opts, optsLim, false);

Options(flags);

if (!IsEOF())

Fail(JSERR\_RegExpSyntax);

this->unicodeFlagPresent = (flags & UnifiedRegex::UnicodeRegexFlag) == UnifiedRegex::UnicodeRegexFlag;

this->caseInsensitiveFlagPresent = (flags & UnifiedRegex::IgnoreCaseRegexFlag) == UnifiedRegex::IgnoreCaseRegexFlag;

Assert(!this->unicodeFlagPresent || scriptContext->GetConfig()->IsES6UnicodeExtensionsEnabled());

}

else

{

this->unicodeFlagPresent = false;

this->caseInsensitiveFlagPresent = false;

}

// If this HR has been set, that means we have an earlier failure than the one caught above.

if (this->deferredIfNotUnicodeError != nullptr && !this->unicodeFlagPresent)

{

throw ParseError(\*deferredIfNotUnicodeError);

}

else if(this->deferredIfUnicodeError != nullptr && this->unicodeFlagPresent)

{

throw ParseError(\*deferredIfUnicodeError);

}

this->currentSurrogatePairNode = this->surrogatePairList;

// Body, pass 1

SetPosition(body, bodyLim, true);

Node\* root = PatternPass1();

Assert(IsEOF());

return root;

}

template <typename P, const bool IsLiteral>

Node\* Parser<P, IsLiteral>::ParseLiteral

( const EncodedChar\* input

, const EncodedChar\* inputLim

, CharCount& outBodyEncodedChars

, CharCount& outTotalEncodedChars

, CharCount& outBodyChars

, CharCount& outTotalChars

, RegexFlags& flags )

{

Assert(IsLiteral);

Assert(input != 0);

Assert(inputLim >= input); // \*inputLim need not be 0 because of deferred parsing

// To handle surrogate pairs properly under unicode option, we will collect information on location of the pairs

// during pass 0, regardless if the option is present. (We aren't able to get it at that time)

// During pass 1, we will use that information to correctly create appropriate nodes.

// Body, pass 0

SetPosition(input, inputLim, true);

PatternPass0();

outBodyEncodedChars = Chars<EncodedChar>::OSB(next, input);

outBodyChars = Pos();

// Options are needed for the next pass

ECMust('/', ERRnoSlash);

Options(flags);

this->unicodeFlagPresent = (flags & UnifiedRegex::UnicodeRegexFlag) == UnifiedRegex::UnicodeRegexFlag;

this->caseInsensitiveFlagPresent = (flags & UnifiedRegex::IgnoreCaseRegexFlag) == UnifiedRegex::IgnoreCaseRegexFlag;

Assert(!this->unicodeFlagPresent || scriptContext->GetConfig()->IsES6UnicodeExtensionsEnabled());

// If this HR has been set, that means we have an earlier failure than the one caught above.

if (this->deferredIfNotUnicodeError != nullptr && !this->unicodeFlagPresent)

{

throw ParseError(\*deferredIfNotUnicodeError);

}

else if(this->deferredIfUnicodeError != nullptr && this->unicodeFlagPresent)

{

throw ParseError(\*deferredIfUnicodeError);

}

// Used below to proceed to the end of the regex

const EncodedChar \*pastOptions = next;

this->currentSurrogatePairNode = this->surrogatePairList;

// Body, pass 1

SetPosition(input, inputLim, true);

Node\* root = PatternPass1();

Assert(outBodyEncodedChars == Chars<EncodedChar>::OSB(next, input));

Assert(outBodyChars == Pos());

next = pastOptions;

outTotalEncodedChars = Chars<EncodedChar>::OSB(next, input);

outTotalChars = Pos();

return root;

}

template <typename P, const bool IsLiteral>

void Parser<P, IsLiteral>::ParseLiteralNoAST

( const EncodedChar\* input

, const EncodedChar\* inputLim

, CharCount& outBodyEncodedChars

, CharCount& outTotalEncodedChars

, CharCount& outBodyChars

, CharCount& outTotalChars )

{

Assert(IsLiteral);

Assert(input != 0);

Assert(inputLim >= input); // \*inputLim need not be 0 because of deferred parsing

// Body, pass 0

SetPosition(input, inputLim, true);

PatternPass0();

outBodyEncodedChars = Chars<EncodedChar>::OSB(next, input);

outBodyChars = Pos();

// Options

ECMust('/', ERRnoSlash);

RegexFlags dummyFlags = NoRegexFlags;

Options(dummyFlags);

this->unicodeFlagPresent = (dummyFlags & UnifiedRegex::UnicodeRegexFlag) == UnifiedRegex::UnicodeRegexFlag;

this->caseInsensitiveFlagPresent = (dummyFlags & UnifiedRegex::IgnoreCaseRegexFlag) == UnifiedRegex::IgnoreCaseRegexFlag;

outTotalEncodedChars = Chars<EncodedChar>::OSB(next, input);

outTotalChars = Pos();

// If this HR has been set, that means we have an earlier failure than the one caught above.

if (this->deferredIfNotUnicodeError != nullptr && !this->unicodeFlagPresent)

{

throw ParseError(\*deferredIfNotUnicodeError);

}

else if(this->deferredIfUnicodeError != nullptr && this->unicodeFlagPresent)

{

throw ParseError(\*deferredIfUnicodeError);

}

}

template <typename P, const bool IsLiteral>

template <const bool buildAST>

RegexPattern \* Parser<P, IsLiteral>::CompileProgram

( Node\* root,

const EncodedChar\*& currentCharacter,

const CharCount totalLen,

const CharCount bodyChars,

const CharCount totalChars,

const RegexFlags flags )

{

Assert(IsLiteral);

Program\* program = nullptr;

if (buildAST)

{

const auto recycler = this->scriptContext->GetRecycler();

program = Program::New(recycler, flags);

this->CaptureSourceAndGroups(recycler, program, currentCharacter, bodyChars);

}

currentCharacter += totalLen;

Assert(GetMultiUnits() == totalLen - totalChars);

if (!buildAST)

{

return nullptr;

}

RegexPattern\* pattern = RegexPattern::New(this->scriptContext, program, true);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

RegexStats\* stats = 0;

if (REGEX\_CONFIG\_FLAG(RegexProfile))

{

stats = this->scriptContext->GetRegexStatsDatabase()->GetRegexStats(pattern);

this->scriptContext->GetRegexStatsDatabase()->EndProfile(stats, RegexStats::Parse);

}

if (REGEX\_CONFIG\_FLAG(RegexTracing))

{

DebugWriter\* tw = this->scriptContext->GetRegexDebugWriter();

tw->Print(L"// REGEX COMPILE ");

pattern->Print(tw);

tw->EOL();

}

if (REGEX\_CONFIG\_FLAG(RegexProfile))

this->scriptContext->GetRegexStatsDatabase()->BeginProfile();

#endif

ArenaAllocator\* rtAllocator = this->scriptContext->RegexAllocator();

Compiler::Compile

( this->scriptContext

, ctAllocator

, rtAllocator

, standardChars

, program

, root

, this->GetLitbuf()

, pattern

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, w

, stats

#endif

);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

if (REGEX\_CONFIG\_FLAG(RegexProfile))

this->scriptContext->GetRegexStatsDatabase()->EndProfile(stats, RegexStats::Compile);

#endif

#ifdef PROFILE\_EXEC

this->scriptContext->ProfileEnd(Js::RegexCompilePhase);

#endif

return pattern;

}

template <typename P, const bool IsLiteral>

void Parser<P, IsLiteral>::CaptureEmptySourceAndNoGroups(Program\* program)

{

Assert(program->source == 0);

program->source = L"";

program->sourceLen = 0;

program->numGroups = 1;

// Remaining to set during compilation: litbuf, litbufLen, numLoops, insts, instsLen, entryPointLabel

}

template <typename P, const bool IsLiteral>

void Parser<P, IsLiteral>::CaptureSourceAndGroups(Recycler\* recycler, Program\* program, const EncodedChar\* body, CharCount bodyChars)

{

Assert(program->source == 0);

Assert(body != 0);

// Program will own source string

program->source = RecyclerNewArrayLeaf(recycler, Char, bodyChars + 1);

// Don't need to zero out since we're writing to the buffer right here

ConvertToUnicode(program->source, bodyChars, body);

program->source[bodyChars] = 0;

program->sourceLen = bodyChars;

program->numGroups = nextGroupId;

// Remaining to set during compilation: litbuf, litbufLen, numLoops, insts, instsLen, entryPointLabel

}

template <typename P, const bool IsLiteral>

Node\* Parser<P, IsLiteral>::GetNodeWithValidCharacterSet(EncodedChar cc)

{

Node\* nodeToReturn = nullptr;

if (this->scriptContext->GetConfig()->IsES6UnicodeExtensionsEnabled() && this->unicodeFlagPresent)

{

MatchSetNode \*lowerRangeNode = Anew(ctAllocator, MatchSetNode, false, false);

lowerRangeNode->set.SetRange(ctAllocator, (Char)0xD800, (Char)0xDBFF);

MatchSetNode \*upperRangeNode = Anew(ctAllocator, MatchSetNode, false, false);

upperRangeNode->set.SetRange(ctAllocator, (Char)0xDC00, (Char)0xDFFF);

ConcatNode\* surrogateRangePairNode = Anew(ctAllocator, ConcatNode, lowerRangeNode, Anew(ctAllocator, ConcatNode, upperRangeNode, nullptr));

// The MatchSet node will be split into [0-D7FFDC00-FFFF] (minus special characters like newline, whitespace, etc.) as a prefix, and a suffix of [D800-DBFF]

// i.e. The MatchSet node can be with [0-D7FFDC00-FFFF] (minus special characters like newline, whitespace, etc.) OR [D800-DBFF]

MatchSetNode\* partialPrefixSetNode = Anew(ctAllocator, MatchSetNode, false, false);

switch (cc)

{

case '.':

standardChars->SetNonNewline(ctAllocator, partialPrefixSetNode->set);

break;

case 'S':

standardChars->SetNonWhitespace(ctAllocator, partialPrefixSetNode->set);

break;

case 'D':

standardChars->SetNonDigits(ctAllocator, partialPrefixSetNode->set);

break;

case 'W':

standardChars->SetNonWordChars(ctAllocator, partialPrefixSetNode->set);

break;

default:

AssertMsg(false, "");

}

partialPrefixSetNode->set.SubtractRange(ctAllocator, (Char)0xD800u, (Char)0xDBFFu);

MatchSetNode\* partialSuffixSetNode = Anew(ctAllocator, MatchSetNode, false, false);

partialSuffixSetNode->set.SetRange(ctAllocator, (Char)0xD800u, (Char)0xDBFFu);

AltNode\* altNode = Anew(ctAllocator, AltNode, partialPrefixSetNode, Anew(ctAllocator, AltNode, surrogateRangePairNode, Anew(ctAllocator, AltNode, partialSuffixSetNode, nullptr)));

nodeToReturn = altNode;

}

else

{

MatchSetNode\* setNode = Anew(ctAllocator, MatchSetNode, false, false);

switch (cc)

{

case '.':

standardChars->SetNonNewline(ctAllocator, setNode->set);

break;

case 'S':

standardChars->SetNonWhitespace(ctAllocator, setNode->set);

break;

case 'D':

standardChars->SetNonDigits(ctAllocator, setNode->set);

break;

case 'W':

standardChars->SetNonWordChars(ctAllocator, setNode->set);

break;

default:

AssertMsg(false, "");

}

nodeToReturn = setNode;

}

return nodeToReturn;

}

template <typename P, const bool IsLiteral>

void Parser<P, IsLiteral>::FreeBody()

{

if (litbuf != 0)

{

ctAllocator->Free(litbuf, litbufLen);

litbuf = 0;

litbufLen = 0;

litbufNext = 0;

}

}

//

// Template instantiation

//

template <typename P, const bool IsLiteral>

void UnifiedRegexParserForceInstantiation()

{

typedef typename P::EncodedChar EncodedChar;

Parser<P, IsLiteral> p

( 0

, 0

, 0

, 0

, false

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, 0

#endif

);

RegexFlags f;

CharCount a, b, c, d;

const EncodedChar\* cp = 0;

p.ParseDynamic(0, 0, 0, 0, f);

p.ParseLiteral(0, 0, a, b, c, d, f);

p.ParseLiteralNoAST(0, 0, a, b, c, d);

p.CompileProgram<true>(0, cp, a, b, c, f);

p.CompileProgram<false>(0, cp, a, b, c, f);

p.CaptureEmptySourceAndNoGroups(0);

p.CaptureSourceAndGroups(0, 0, 0, 0);

p.FreeBody();

}

void UnifiedRegexParserForceAllInstantiations()

{

UnifiedRegexParserForceInstantiation<NullTerminatedUnicodeEncodingPolicy, false>();

UnifiedRegexParserForceInstantiation<NullTerminatedUnicodeEncodingPolicy, true>();

UnifiedRegexParserForceInstantiation<NullTerminatedUTF8EncodingPolicy, false>();

UnifiedRegexParserForceInstantiation<NullTerminatedUTF8EncodingPolicy, true>();

UnifiedRegexParserForceInstantiation<NotNullTerminatedUTF8EncodingPolicy, false>();

UnifiedRegexParserForceInstantiation<NotNullTerminatedUTF8EncodingPolicy, true>();

}

}

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

// 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 UnifiedRegex

{

struct ParseError

{

bool isBody;

CharCount pos; // Position in unicode characters

CharCount encodedPos; // Position in underlying characters (eg utf-8 bytes)

HRESULT error;

ParseError(bool isBody, CharCount pos, CharCount encodedPos, HRESULT error);

};

template <typename EncodingPolicy, const bool IsLiteral>

class Parser : private EncodingPolicy, private Chars<wchar\_t>

{

private:

typedef typename EncodingPolicy::EncodedChar EncodedChar;

// A linked list node to track indices of surrogate pairs.

struct SurrogatePairTracker

{

const EncodedChar\* location;

// If this surrogate pair is inside a range, then rangeLocation isn't null.

const EncodedChar\* rangeLocation;

codepoint\_t value;

uint32 length;

size\_t multiUnits;

SurrogatePairTracker\* next;

SurrogatePairTracker(const EncodedChar\* location, codepoint\_t value, uint32 length, size\_t multiUnits)

: location(location)

, next(nullptr)

, value(value)

, length(length)

, multiUnits(multiUnits)

, rangeLocation(nullptr)

{

}

SurrogatePairTracker(const EncodedChar\* location, const EncodedChar\* rangeLocation, codepoint\_t value, uint32 length, size\_t multiUnits)

: location(location)

, next(nullptr)

, value(value)

, length(length)

, multiUnits(multiUnits)

, rangeLocation(rangeLocation)

{

}

bool IsInsideRange() const

{

return this->rangeLocation != nullptr;

}

};

static const CharCount initLitbufSize = 16;

Js::ScriptContext\* scriptContext;

// Arena for nodes and items needed only during compliation

ArenaAllocator\* ctAllocator;

// Standard characters using raw encoding character representation (eg char for utf-8)

StandardChars<EncodedChar>\* standardEncodedChars;

// Standard characters using final character representation (eg wchar\_t for Unicode)

StandardChars<Char>\* standardChars;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

DebugWriter\* w;

#endif

const EncodedChar\* input;

const EncodedChar\* inputLim;

const EncodedChar\* next;

bool inBody;

int numGroups; // determined in first parse

int nextGroupId;

// Buffer accumulating all literals.

// In compile-time allocator, must be transferred to runtime allocator when build program

Char\* litbuf;

CharCount litbufLen;

CharCount litbufNext;

// During pass 0, if /u option for regex is provided, a linked list will be built up to

// track positions of surrogate pairs in the buffer. During pass 1, these linked lists will be used

// to figure out when to output a surrogate pair node.

SurrogatePairTracker\* surrogatePairList;

SurrogatePairTracker\* currentSurrogatePairNode;

bool unicodeFlagPresent;

bool caseInsensitiveFlagPresent;

// The following two variables are used to determine if the the surrogate pair has been encountered

// First holds the temporary location, second holds the value of the codepoint

const EncodedChar\* tempLocationOfSurrogatePair;

// This will be set to a location when we are parsing a range in TermPass0, and cleared when we are out of it.

const EncodedChar\* tempLocationOfRange;

codepoint\_t codePointAtTempLocation;

// When a surrogate is added for tracking, this will be updated.

const EncodedChar\* positionAfterLastSurrogate;

codepoint\_t valueOfLastSurrogate;

// deferred error state.

ParseError\* deferredIfNotUnicodeError;

ParseError\* deferredIfUnicodeError;

private:

//

// Input buffer management

//

void SetPosition(const EncodedChar\* input, const EncodedChar\* inputLim, bool inBody);

// Current position in number of logical characters, regardless of underlying character encoding

inline CharCount Pos();

inline bool IsEOF();

inline bool ECCanConsume(CharCount n);

inline EncodedChar ECLookahead(CharCount n = 0);

inline EncodedChar ECLookback(CharCount n = 0);

inline void ECConsume(CharCount n = 1);

inline void ECConsumeMultiUnit(CharCount n = 1);

inline void ECRevert(CharCount n = 1);

//

// Helpers

//

int TryParseExtendedUnicodeEscape(Char& c, bool& previousSurrogatePart, bool trackSurrogatePair = false);

void TrackIfSurrogatePair(codepoint\_t codePoint, const EncodedChar\* location, uint32 consumptionLength);

Node\* CreateSurrogatePairAtom(wchar\_t lower, wchar\_t upper);

AltNode\* AppendSurrogateRangeToDisjunction(codepoint\_t lowerCodePoint, codepoint\_t upperCodePoint, AltNode \*lastAlttNode);

AltNode\* AppendSurrogatePairToDisjunction(codepoint\_t codePoint, AltNode \*lastAlttNode);

//

// Errors

//

void Fail(HRESULT error);

void DeferredFailIfUnicode(HRESULT error);

void DeferredFailIfNotUnicode(HRESULT error);

inline void ECMust(EncodedChar ec, HRESULT error);

inline Char NextChar();

//

// Patterns/Disjunctions/Alternatives

//

void PatternPass0();

Node\* PatternPass1();

Node\* UnionNodes(Node\* prev, Node\* curr);

void DisjunctionPass0(int depth);

Node\* DisjunctionPass1();

bool IsEndOfAlternative();

void EnsureLitbuf(CharCount size);

void AccumLiteral(MatchLiteralNode\* deferredLiteralNode, Node\* charOrLiteralNode);

Node\* FinalTerm(Node\* node, MatchLiteralNode\* deferredLiteralNode);

void AlternativePass0(int depth);

Node\* AlternativePass1();

//

// Terms

//

Node\* NewLoopNode(CharCount lower, CharCountOrFlag upper, bool isGreedy, Node\* body);

bool AtQuantifier();

bool OptNonGreedy();

CharCount RepeatCount();

void TermPass0(int depth);

Node\* TermPass1(MatchCharNode\* deferredCharNode, bool& previousSurrogatePart);

bool AtomEscapePass0();

bool AtomEscapePass1(Node\*& node, MatchCharNode\* deferredCharNode, bool& previousSurrogatePart);

bool SurrogatePairPass1(Node\*& node, MatchCharNode\* deferredCharNode, bool& previousSurrogatePart);

//

// Classes

//

bool AtSecondSingletonClassAtom();

void CharacterClassPass0();

template <bool containsSurrogates>

Node\* CharacterClassPass1();

bool ClassEscapePass0(Char& singleton, bool& previousSurrogatePart);

Node\* ClassEscapePass1(MatchCharNode\* deferredCharNode, MatchSetNode\* deferredSetNode, bool& previousSurrogatePart);

Node\* GetNodeWithValidCharacterSet(EncodedChar ch);

//

// Options

//

void Options(RegexFlags& flags);

public:

Parser

( Js::ScriptContext\* scriptContext

, ArenaAllocator\* ctAllocator

, StandardChars<EncodedChar>\* standardEncodedChars

, StandardChars<Char>\* standardChars

, bool isFromExternalSource

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, DebugWriter\* w

#endif

);

//

// Entry points

//

Node\* ParseDynamic

( const EncodedChar\* body // non null, null terminated (may contain embedded nulls)

, const EncodedChar\* bodyLim // points to terminating null of above

, const EncodedChar\* opts // may be null if no options, otherwise null terminated

, const EncodedChar\* optsLim // if above non-null, points to terminating null of above

, RegexFlags& flags );

// (\*) For ParseLiteral:

// - input string must be null terminated

// - inputLim may point to the terminating null in above or before it

// - if the later, input is known to be syntactically well-formed so that the parser

// will find the natural end of the regex literal before passing inputLim

// - input may conatin nulls before the inputLim

Node\* ParseLiteral

( const EncodedChar\* input // non null, null terminated (may contain embedded nulls)

, const EncodedChar\* inputLim // see (\*) above

, CharCount& outBodyEncodedChars // in encoded characters, not including trailing '/'

, CharCount& outTotalEncodedChars // in encoded characters, including trailing '/' and any options

, CharCount& outBodyChars // in unicode characters, not including ttrailing '/'

, CharCount& outTotalChars // in unicode characters, including trailing '/' and any options

, RegexFlags& flags );

void ParseLiteralNoAST

( const EncodedChar\* input // non null, null terminated

, const EncodedChar\* inputLim // see (\*) above

, CharCount& outBodyEncodedChars

, CharCount& outTotalEncodedChars

, CharCount& outBodyChars

, CharCount& outTotalChars );

template<const bool buildAST>

RegexPattern\* CompileProgram

( Node\* root,

const EncodedChar\*& currentCharacter,

const CharCount totalLen,

const CharCount bodyChars,

const CharCount totalChars,

const RegexFlags flags );

static void CaptureEmptySourceAndNoGroups(Program\* program);

// bodyChars is number of unicode characters in program body, which may be less than the number

// of underlying UTF-8 characters

void CaptureSourceAndGroups(Recycler\* recycler, Program\* program, const EncodedChar\* body, CharCount bodyChars);

inline const Char\* GetLitbuf() { return litbuf; }

void FreeBody();

size\_t GetMultiUnits() { return this->m\_cMultiUnits; }

};

}

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

// 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"

namespace UnifiedRegex

{

RegexPattern::RegexPattern(Js::JavascriptLibrary \*const library, Program\* program, bool isLiteral)

: library(library), isLiteral(isLiteral), isShallowClone(false)

{

rep.unified.program = program;

rep.unified.matcher = 0;

rep.unified.trigramInfo = 0;

}

RegexPattern \*RegexPattern::New(Js::ScriptContext \*scriptContext, Program\* program, bool isLiteral)

{

return

RecyclerNewFinalized(

scriptContext->GetRecycler(),

RegexPattern,

scriptContext->GetLibrary(),

program,

isLiteral);

}

void RegexPattern::Finalize(bool isShutdown)

{

if(isShutdown)

return;

const auto scriptContext = GetScriptContext();

if(!scriptContext)

return;

#if DBG

if(!isLiteral && !scriptContext->IsClosed())

{

const auto source = GetSource();

RegexPattern \*p;

Assert(

!GetScriptContext()->GetDynamicRegexMap()->TryGetValue(

RegexKey(source.GetBuffer(), source.GetLength(), GetFlags()),

&p) ||

p != this);

}

#endif

if(isShallowClone)

return;

rep.unified.program->FreeBody(scriptContext->RegexAllocator());

}

void RegexPattern::Dispose(bool isShutdown)

{

}

Js::ScriptContext \*RegexPattern::GetScriptContext() const

{

return library->GetScriptContext();

}

Js::InternalString RegexPattern::GetSource() const

{

return Js::InternalString(rep.unified.program->source, rep.unified.program->sourceLen);

}

RegexFlags RegexPattern::GetFlags() const

{

return rep.unified.program->flags;

}

int RegexPattern::NumGroups() const

{

return rep.unified.program->numGroups;

}

bool RegexPattern::IsIgnoreCase() const

{

return (rep.unified.program->flags & IgnoreCaseRegexFlag) != 0;

}

bool RegexPattern::IsGlobal() const

{

return (rep.unified.program->flags & GlobalRegexFlag) != 0;

}

bool RegexPattern::IsMultiline() const

{

return (rep.unified.program->flags & MultilineRegexFlag) != 0;

}

bool RegexPattern::IsUnicode() const

{

return GetScriptContext()->GetConfig()->IsES6UnicodeExtensionsEnabled() && (rep.unified.program->flags & UnicodeRegexFlag) != 0;

}

bool RegexPattern::IsSticky() const

{

return GetScriptContext()->GetConfig()->IsES6RegExStickyEnabled() && (rep.unified.program->flags & StickyRegexFlag) != 0;

}

bool RegexPattern::WasLastMatchSuccessful() const

{

return rep.unified.matcher != 0 && rep.unified.matcher->WasLastMatchSuccessful();

}

GroupInfo RegexPattern::GetGroup(int groupId) const

{

Assert(groupId == 0 || WasLastMatchSuccessful());

Assert(groupId >= 0 && groupId < NumGroups());

return rep.unified.matcher->GetGroup(groupId);

}

RegexPattern \*RegexPattern::CopyToScriptContext(Js::ScriptContext \*scriptContext)

{

// This routine assumes that this instance will outlive the copy, which is the case for copy-on-write,

// and therefore doesn't copy the immutable parts of the pattern. This should not be confused with a

// would be CloneToScriptContext which will would clone the immutable parts as well because the lifetime

// of a clone might be longer than the original.

RegexPattern \*result = UnifiedRegex::RegexPattern::New(scriptContext, rep.unified.program, isLiteral);

Matcher \*matcherClone = rep.unified.matcher ? rep.unified.matcher->CloneToScriptContext(scriptContext, result) : nullptr;

result->rep.unified.matcher = matcherClone;

result->isShallowClone = true;

return result;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void RegexPattern::Print(DebugWriter\* w)

{

w->Print(L"/");

Js::InternalString str = GetSource();

if (str.GetLength() == 0)

w->Print(L"(?:)");

else

{

for (charcount\_t i = 0; i < str.GetLength(); ++i)

{

const wchar\_t c = str.GetBuffer()[i];

switch(c)

{

case L'/':

w->Print(L"\\%lc", c);

break;

case L'\n':

case L'\r':

case L'\x2028':

case L'\x2029':

w->PrintEscapedChar(c);

break;

case L'\\':

Assert(i + 1 < str.GetLength()); // cannot end in a '\'

w->Print(L"\\%lc", str.GetBuffer()[++i]);

break;

default:

w->PrintEscapedChar(c);

break;

}

}

}

w->Print(L"/");

if (IsIgnoreCase())

w->Print(L"i");

if (IsGlobal())

w->Print(L"g");

if (IsMultiline())

w->Print(L"m");

if (IsUnicode())

w->Print(L"u");

if (IsSticky())

w->Print(L"y");

w->Print(L" /\* ");

w->Print(L", ");

w->Print(isLiteral ? L"literal" : L"dynamic");

w->Print(L" \*/");

}

#endif

}

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

// 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 JavascriptLibrary;

}

namespace UnifiedRegex

{

struct Program;

class Matcher;

struct TrigramInfo;

struct RegexPattern : FinalizableObject

{

struct UnifiedRep

{

Program\* program;

Matcher\* matcher;

TrigramInfo\* trigramInfo;

};

Js::JavascriptLibrary \*const library;

bool isLiteral : 1;

bool isShallowClone : 1;

union Rep

{

struct UnifiedRep unified;

} rep;

RegexPattern(Js::JavascriptLibrary \*const library, Program\* program, bool isLiteral);

static RegexPattern \*New(Js::ScriptContext \*scriptContext, Program\* program, bool isLiteral);

virtual void Finalize(bool isShutdown) override;

virtual void Dispose(bool isShutdown) override;

virtual void Mark(Recycler \*recycler) override { AssertMsg(false, "Mark called on object that isn't TrackableObject"); }

Js::ScriptContext \*GetScriptContext() const;

inline bool IsLiteral() const { return isLiteral; }

int NumGroups() const;

bool IsIgnoreCase() const;

bool IsGlobal() const;

bool IsMultiline() const;

bool IsUnicode() const;

bool IsSticky() const;

bool WasLastMatchSuccessful() const;

GroupInfo GetGroup(int groupId) const;

Js::InternalString GetSource() const;

RegexFlags GetFlags() const;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w);

#endif

RegexPattern \*CopyToScriptContext(Js::ScriptContext \*scriptContext);

};

}

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

// 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"

namespace UnifiedRegex

{

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

// CountDomain

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

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void CountDomain::Print(DebugWriter\* w) const

{

if (upper != CharCountFlag && lower == (CharCount)upper)

w->Print(L"[%u]", lower);

else

{

w->Print(L"[%u-", lower);

if (upper == CharCountFlag)

w->Print(L"inf]");

else

w->Print(L"%u]", (CharCount)upper);

}

}

#endif

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

// Matcher (inlined, called from instruction Exec methods)

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

#define PUSH(contStack, T, ...) (new (contStack.Push<T>()) T(\_\_VA\_ARGS\_\_))

#define PUSHA(assertionStack, T, ...) (new (assertionStack.Push()) T(\_\_VA\_ARGS\_\_))

#define L2I(O, label) LabelToInstPointer<O##Inst>(Inst::O, label)

#define FAIL\_PARAMETERS input, inputOffset, instPointer, contStack, assertionStack, qcTicks

#define HARDFAIL\_PARAMETERS(mode) input, inputLength, matchStart, inputOffset, instPointer, contStack, assertionStack, qcTicks, mode

// Regex QC heuristics:

// - TicksPerQC

// - Number of ticks from a previous QC needed to cause another QC. The value affects how often QC will be triggered, so

// on slower machines or debug builds, the value needs to be smaller to maintain a reasonable frequency of QCs.

// - TicksPerQcTimeCheck

// - Number of ticks from a previous QC needed to trigger a time check. Elapsed time from the previous QC is checked to

// see if a QC needs to be triggered. The value must be less than TicksPerQc and small enough to reasonably guarantee

// a QC every TimePerQc milliseconds without affecting perf.

// - TimePerQc

// - The target time between QCs

#if defined(\_M\_ARM)

const uint Matcher::TicksPerQc = 1u << 19

#else

const uint Matcher::TicksPerQc = 1u << (AutoSystemInfo::ShouldQCMoreFrequently() ? 17 : 21)

#endif

#if DBG

>> 2

#endif

;

const uint Matcher::TicksPerQcTimeCheck = Matcher::TicksPerQc >> 2;

const uint Matcher::TimePerQc = AutoSystemInfo::ShouldQCMoreFrequently() ? 50 : 100; // milliseconds

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Matcher::PushStats(ContStack& contStack, const Char\* const input) const

{

if (stats != 0)

{

stats->numPushes++;

if (contStack.Position() > stats->stackHWM)

stats->stackHWM = contStack.Position();

}

if (w != 0)

{

w->Print(L"PUSH ");

contStack.Top()->Print(w, input);

}

}

void Matcher::PopStats(ContStack& contStack, const Char\* const input) const

{

if (stats != 0)

stats->numPops++;

if (w != 0)

{

const Cont\* top = contStack.Top();

if (top == 0)

w->PrintEOL(L"<empty stack>");

else

{

w->Print(L"POP ");

top->Print(w, input);

}

}

}

void Matcher::UnPopStats(ContStack& contStack, const Char\* const input) const

{

if (stats != 0)

stats->numPops--;

if (w != 0)

{

const Cont\* top = contStack.Top();

if (top == 0)

w->PrintEOL(L"<empty stack>");

else

{

w->Print(L"UNPOP ");

top->Print(w, input);

}

}

}

void Matcher::CompStats() const

{

if (stats != 0)

stats->numCompares++;

}

void Matcher::InstStats() const

{

if (stats != 0)

stats->numInsts++;

}

#endif

\_\_inline void Matcher::QueryContinue(uint &qcTicks)

{

// See definition of TimePerQc for description of regex QC heuristics

Assert(!(TicksPerQc & TicksPerQc - 1)); // must be a power of 2

Assert(!(TicksPerQcTimeCheck & TicksPerQcTimeCheck - 1)); // must be a power of 2

Assert(TicksPerQcTimeCheck < TicksPerQc);

if(PHASE\_OFF1(Js::RegexQcPhase))

return;

if(++qcTicks & TicksPerQcTimeCheck - 1)

return;

DoQueryContinue(qcTicks);

}

\_\_inline bool Matcher::HardFail

( const Char\* const input

, const CharCount inputLength

, CharCount &matchStart

, CharCount &inputOffset

, const uint8 \*&instPointer

, ContStack &contStack

, AssertionStack &assertionStack

, uint &qcTicks

, HardFailMode mode )

{

switch (mode)

{

case BacktrackAndLater:

return Fail(FAIL\_PARAMETERS);

case BacktrackOnly:

if (Fail(FAIL\_PARAMETERS))

{

// No use trying any more start positions

matchStart = inputLength;

return true; // STOP EXECUTING

}

else

return false;

case LaterOnly:

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

if (w != 0)

w->PrintEOL(L"CLEAR");

#endif

contStack.Clear();

assertionStack.Clear();

return true; // STOP EXECUTING

case ImmediateFail:

// No use trying any more start positions

matchStart = inputLength;

return true; // STOP EXECUTING

default:

Assume(false);

}

return true;

}

\_\_inline bool Matcher::PopAssertion(CharCount &inputOffset, const uint8 \*&instPointer, ContStack &contStack, AssertionStack &assertionStack, bool succeeded)

{

AssertionInfo\* info = assertionStack.Top();

Assert(info != 0);

assertionStack.Pop();

BeginAssertionInst\* begin = L2I(BeginAssertion, info->beginLabel);

// Cut the existing continuations (we never backtrack into an assertion)

// NOTE: We don't include the effective pops in the stats

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

if (w != 0)

w->PrintEOL(L"POP TO %llu", (unsigned long long)info->contStackPosition);

#endif

contStack.PopTo(info->contStackPosition);

// succeeded isNegation action

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

// false false Fail into outer continuations (inner group bindings will have been undone)

// true false Jump to next label (inner group bindings are now frozen)

// false true Jump to next label (inner group bindings will have been undone and are now frozen)

// true true Fail into outer continuations (inner group binding MUST BE CLEARED)

if (succeeded && begin->isNegation)

ResetInnerGroups(begin->minBodyGroupId, begin->maxBodyGroupId);

if (succeeded == begin->isNegation)

{

// Assertion failed

return false;

}

else

{

// Continue with next label but from original input position

inputOffset = info->startInputOffset;

instPointer = LabelToInstPointer(begin->nextLabel);

return true;

}

}

\_\_inline void Matcher::SaveInnerGroups(

const int fromGroupId,

const int toGroupId,

const bool reset,

const Char \*const input,

ContStack &contStack)

{

if(toGroupId >= 0)

DoSaveInnerGroups(fromGroupId, toGroupId, reset, input, contStack);

}

void Matcher::DoSaveInnerGroups(

const int fromGroupId,

const int toGroupId,

const bool reset,

const Char \*const input,

ContStack &contStack)

{

Assert(fromGroupId >= 0);

Assert(toGroupId >= 0);

Assert(fromGroupId <= toGroupId);

int undefinedRangeFromId = -1;

int groupId = fromGroupId;

do

{

GroupInfo \*const groupInfo = GroupIdToGroupInfo(groupId);

if(groupInfo->IsUndefined())

{

if(undefinedRangeFromId < 0)

undefinedRangeFromId = groupId;

continue;

}

if(undefinedRangeFromId >= 0)

{

Assert(groupId > 0);

DoSaveInnerGroups\_AllUndefined(undefinedRangeFromId, groupId - 1, input, contStack);

undefinedRangeFromId = -1;

}

PUSH(contStack, RestoreGroupCont, groupId, \*groupInfo);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

PushStats(contStack, input);

#endif

if(reset)

groupInfo->Reset();

} while(++groupId <= toGroupId);

if(undefinedRangeFromId >= 0)

{

Assert(toGroupId >= 0);

DoSaveInnerGroups\_AllUndefined(undefinedRangeFromId, toGroupId, input, contStack);

}

}

\_\_inline void Matcher::SaveInnerGroups\_AllUndefined(

const int fromGroupId,

const int toGroupId,

const Char \*const input,

ContStack &contStack)

{

if(toGroupId >= 0)

DoSaveInnerGroups\_AllUndefined(fromGroupId, toGroupId, input, contStack);

}

void Matcher::DoSaveInnerGroups\_AllUndefined(

const int fromGroupId,

const int toGroupId,

const Char \*const input,

ContStack &contStack)

{

Assert(fromGroupId >= 0);

Assert(toGroupId >= 0);

Assert(fromGroupId <= toGroupId);

#if DBG

for(int groupId = fromGroupId; groupId <= toGroupId; ++groupId)

{

Assert(GroupIdToGroupInfo(groupId)->IsUndefined());

}

#endif

if(fromGroupId == toGroupId)

PUSH(contStack, ResetGroupCont, fromGroupId);

else

PUSH(contStack, ResetGroupRangeCont, fromGroupId, toGroupId);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

PushStats(contStack, input);

#endif

}

\_\_inline void Matcher::ResetGroup(int groupId)

{

GroupInfo\* info = GroupIdToGroupInfo(groupId);

info->Reset();

}

\_\_inline void Matcher::ResetInnerGroups(int minGroupId, int maxGroupId)

{

for (int i = minGroupId; i <= maxGroupId; i++)

ResetGroup(i);

}

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

// Mixins

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

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void BackupMixin::Print(DebugWriter\* w, const wchar\_t\* litbuf) const

{

w->Print(L"backup: ");

backup.Print(w);

}

#endif

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void CharMixin::Print(DebugWriter\* w, const wchar\_t\* litbuf) const

{

w->Print(L"c: ");

w->PrintQuotedChar(c);

}

#endif

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Char2Mixin::Print(DebugWriter\* w, const wchar\_t\* litbuf) const

{

w->Print(L"c0: ");

w->PrintQuotedChar(cs[0]);

w->Print(L", c1: ");

w->PrintQuotedChar(cs[1]);

}

#endif

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Char3Mixin::Print(DebugWriter\* w, const wchar\_t\* litbuf) const

{

w->Print(L"c0: ");

w->PrintQuotedChar(cs[0]);

w->Print(L", c1: ");

w->PrintQuotedChar(cs[1]);

w->Print(L", c2: ");

w->PrintQuotedChar(cs[2]);

}

#endif

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Char4Mixin::Print(DebugWriter\* w, const wchar\_t\* litbuf) const

{

w->Print(L"c0: ");

w->PrintQuotedChar(cs[0]);

w->Print(L", c1: ");

w->PrintQuotedChar(cs[1]);

w->Print(L", c2: ");

w->PrintQuotedChar(cs[2]);

w->Print(L", c3: ");

w->PrintQuotedChar(cs[3]);

}

#endif

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void LiteralMixin::Print(DebugWriter\* w, const wchar\_t\* litbuf, bool isEquivClass) const

{

if (isEquivClass)

{

w->Print(L"equivLiterals: ");

for (int i = 0; i < CaseInsensitive::EquivClassSize; i++)

{

if (i > 0)

w->Print(L", ");

w->Print(L"\"");

for (CharCount j = 0; j < length; j++)

w->PrintEscapedChar(litbuf[offset + j \* CaseInsensitive::EquivClassSize + i]);

w->Print(L"\"");

}

}

else

{

w->Print(L"literal: ");

w->PrintQuotedString(litbuf + offset, length);

}

}

#endif

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

// Char2LiteralScannerMixin

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

bool Char2LiteralScannerMixin::Match(Matcher& matcher, const wchar\_t\* const input, const CharCount inputLength, CharCount& inputOffset) const

{

if (inputLength == 0)

{

return false;

}

const uint matchC0 = Chars<wchar\_t>::CTU(cs[0]);

const uint matchC1 = Chars<wchar\_t>::CTU(cs[1]);

const wchar\_t \* currentInput = input + inputOffset;

const wchar\_t \* endInput = input + inputLength - 1;

while (currentInput < endInput)

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

while (true)

{

const uint c1 = Chars<wchar\_t>::CTU(currentInput[1]);

if (c1 != matchC1)

{

if (c1 == matchC0)

{

break;

}

currentInput += 2;

if (currentInput >= endInput)

{

return false;

}

continue;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

// Check the first character

const uint c0 = Chars<wchar\_t>::CTU(\*currentInput);

if (c0 == matchC0)

{

inputOffset = (CharCount)(currentInput - input);

return true;

}

if (matchC0 == matchC1)

{

break;

}

currentInput +=2;

if (currentInput >= endInput)

{

return false;

}

}

// If the second character in the buffer matches the first in the pattern, continue

// to see if the next character has the second in the pattern

currentInput++;

while (currentInput < endInput)

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

const uint c1 = Chars<wchar\_t>::CTU(currentInput[1]);

if (c1 == matchC1)

{

inputOffset = (CharCount)(currentInput - input);

return true;

}

if (c1 != matchC0)

{

currentInput += 2;

break;

}

currentInput++;

}

}

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Char2LiteralScannerMixin::Print(DebugWriter\* w, const wchar\_t \* litbuf) const

{

Char2Mixin::Print(w, litbuf);

w->Print(L" (with two character literal scanner)");

}

#endif

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

// ScannerMixinT

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

template <typename ScannerT>

void ScannerMixinT<ScannerT>::FreeBody(ArenaAllocator\* rtAllocator)

{

scanner.FreeBody(rtAllocator, length);

}

template <typename ScannerT>

\_\_inline bool

ScannerMixinT<ScannerT>::Match(Matcher& matcher, const wchar\_t \* const input, const CharCount inputLength, CharCount& inputOffset) const

{

Assert(length <= matcher.program->rep.insts.litbufLen - offset);

return scanner.Match<1>

( input

, inputLength

, inputOffset

, matcher.program->rep.insts.litbuf + offset

, length

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, matcher.stats

#endif

);

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

template <typename ScannerT>

void ScannerMixinT<ScannerT>::Print(DebugWriter\* w, const wchar\_t\* litbuf, bool isEquivClass) const

{

LiteralMixin::Print(w, litbuf, isEquivClass);

w->Print(L" (with %s scanner)", ScannerT::GetName());

}

#endif

// explicit instantiation

template ScannerMixinT<TextbookBoyerMoore<wchar\_t>>;

template ScannerMixinT<TextbookBoyerMooreWithLinearMap<wchar\_t>>;

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

// EquivScannerMixinT

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

template <uint lastPatCharEquivClassSize>

\_\_inline bool EquivScannerMixinT<lastPatCharEquivClassSize>::Match(Matcher& matcher, const wchar\_t\* const input, const CharCount inputLength, CharCount& inputOffset) const

{

Assert(length \* CaseInsensitive::EquivClassSize <= matcher.program->rep.insts.litbufLen - offset);

CompileAssert(lastPatCharEquivClassSize >= 1 && lastPatCharEquivClassSize <= CaseInsensitive::EquivClassSize);

return scanner.Match<CaseInsensitive::EquivClassSize, lastPatCharEquivClassSize>

( input

, inputLength

, inputOffset

, matcher.program->rep.insts.litbuf + offset

, length

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, matcher.stats

#endif

);

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

template <uint lastPatCharEquivClassSize>

void EquivScannerMixinT<lastPatCharEquivClassSize>::Print(DebugWriter\* w, const wchar\_t\* litbuf) const

{

\_\_super::Print(w, litbuf, true);

w->Print(L" (last char equiv size:%d)", lastPatCharEquivClassSize);

}

// explicit instantiation

template struct EquivScannerMixinT<1>;

#endif

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

// ScannerInfo

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

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void ScannerInfo::Print(DebugWriter\* w, const wchar\_t\* litbuf) const

{

ScannerMixin::Print(w, litbuf, isEquivClass);

}

#endif

ScannerInfo\* ScannersMixin::Add(Recycler \*recycler, Program \*program, CharCount offset, CharCount length, bool isEquivClass)

{

Assert(numLiterals < MaxNumSyncLiterals);

return program->AddScannerForSyncToLiterals(recycler, numLiterals++, offset, length, isEquivClass);

}

void ScannersMixin::FreeBody(ArenaAllocator\* rtAllocator)

{

for (int i = 0; i < numLiterals; i++)

{

infos[i]->FreeBody(rtAllocator);

#if DBG

infos[i] = 0;

#endif

}

#if DBG

numLiterals = 0;

#endif

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void ScannersMixin::Print(DebugWriter\* w, const wchar\_t\* litbuf) const

{

w->Print(L"literals: {");

for (int i = 0; i < numLiterals; i++)

{

if (i > 0)

w->Print(L", ");

infos[i]->Print(w, litbuf);

}

w->Print(L"}");

}

#endif

template<bool IsNegation>

void SetMixin<IsNegation>::FreeBody(ArenaAllocator\* rtAllocator)

{

set.FreeBody(rtAllocator);

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

template<bool IsNegation>

void SetMixin<IsNegation>::Print(DebugWriter\* w, const wchar\_t\* litbuf) const

{

w->Print(L"set: ");

if (IsNegation)

w->Print(L"not ");

set.Print(w);

}

#endif

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void HardFailMixin::Print(DebugWriter\* w, const wchar\_t\* litbuf) const

{

w->Print(L"hardFail: %s", canHardFail ? L"true" : L"false");

}

#endif

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void GroupMixin::Print(DebugWriter\* w, const wchar\_t\* litbuf) const

{

w->Print(L"groupId: %d", groupId);

}

#endif

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void ChompBoundedMixin::Print(DebugWriter\* w, const wchar\_t\* litbuf) const

{

w->Print(L"repeats: ");

repeats.Print(w);

}

#endif

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void JumpMixin::Print(DebugWriter\* w, const wchar\_t\* litbuf) const

{

w->Print(L"targetLabel: L%04x", targetLabel);

}

#endif

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void BodyGroupsMixin::Print(DebugWriter\* w, const wchar\_t\* litbuf) const

{

w->Print(L"minBodyGroupId: %d, maxBodyGroupId: %d", minBodyGroupId, maxBodyGroupId);

}

#endif

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void BeginLoopMixin::Print(DebugWriter\* w, const wchar\_t\* litbuf) const

{

w->Print(L"loopId: %d, repeats: ", loopId);

repeats.Print(w);

w->Print(L", exitLabel: L%04x, hasOuterLoops: %s, hasInnerNondet: %s", exitLabel, hasOuterLoops ? L"true" : L"false", hasInnerNondet ? L"true" : L"false");

}

#endif

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void RepeatLoopMixin::Print(DebugWriter\* w, const wchar\_t\* litbuf) const

{

w->Print(L"beginLabel: L%04x", beginLabel);

}

#endif

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void TryMixin::Print(DebugWriter\* w, const wchar\_t\* litbuf) const

{

w->Print(L"failLabel: L%04x", failLabel);

}

#endif

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void FixedLengthMixin::Print(DebugWriter\* w, const wchar\_t\* litbuf) const

{

w->Print(L"length: %u", length);

}

#endif

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void NoNeedToSaveMixin::Print(DebugWriter\* w, const wchar\_t\* litbuf) const

{

w->Print(L"noNeedToSave: %s", noNeedToSave ? L"true" : L"false");

}

#endif

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void SwitchCase::Print(DebugWriter\* w) const

{

w->Print(L"case ");

w->PrintQuotedChar(c);

w->PrintEOL(L": Jump(L%04x)", targetLabel);

}

#endif

template <int n>

void SwitchMixin<n>::AddCase(wchar\_t c, Label targetLabel)

{

Assert(numCases < MaxCases);

int i;

\_\_analysis\_assume(numCases < MaxCases);

for (i = 0; i < numCases; i++)

{

Assert(cases[i].c != c);

if (cases[i].c > c)

break;

}

\_\_analysis\_assume(numCases < MaxCases);

for (int j = numCases; j > i; j--)

cases[j] = cases[j - 1];

cases[i].c = c;

cases[i].targetLabel = targetLabel;

numCases++;

}

void UnifiedRegexSwitchMixinForceAllInstantiations()

{

{

SwitchMixin<10> x;

x.AddCase(0, 0);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

x.Print(0, 0);

#endif

}

{

SwitchMixin<20> x;

x.AddCase(0, 0);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

x.Print(0, 0);

#endif

}

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

template <int n>

void SwitchMixin<n>::Print(DebugWriter\* w, const wchar\_t\* litbuf) const

{

w->EOL();

w->Indent();

for (int i = 0; i < numCases; i++)

cases[i].Print(w);

w->Unindent();

}

#endif

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

// FailInst

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

\_\_inline bool FailInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

return matcher.Fail(FAIL\_PARAMETERS);

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int FailInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->PrintEOL(L"L%04x: Fail()", label);

return sizeof(\*this);

}

#endif

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

// SuccInst

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

\_\_inline bool SuccInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

GroupInfo\* info = matcher.GroupIdToGroupInfo(0);

info->offset = matchStart;

info->length = inputOffset - matchStart;

return true; // STOP MATCHING

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int SuccInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->PrintEOL(L"L%04x: Succ()", label);

return sizeof(\*this);

}

#endif

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

// JumpInst

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

\_\_inline bool JumpInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

instPointer = matcher.LabelToInstPointer(targetLabel);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int JumpInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: Jump(", label);

JumpMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// JumpIfNotCharInst (optimized instruction)

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

\_\_inline bool JumpIfNotCharInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (inputOffset < inputLength && input[inputOffset] == c)

instPointer += sizeof(\*this);

else

instPointer = matcher.LabelToInstPointer(targetLabel);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int JumpIfNotCharInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: JumpIfNotChar(", label);

CharMixin::Print(w, litbuf);

w->Print(L", ");

JumpMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// MatchCharOrJumpInst (optimized instruction)

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

\_\_inline bool MatchCharOrJumpInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (inputOffset < inputLength && input[inputOffset] == c)

{

inputOffset++;

instPointer += sizeof(\*this);

}

else

instPointer = matcher.LabelToInstPointer(targetLabel);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int MatchCharOrJumpInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: MatchCharOrJump(", label);

CharMixin::Print(w, litbuf);

w->Print(L", ");

JumpMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// JumpIfNotSetInst (optimized instruction)

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

\_\_inline bool JumpIfNotSetInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (inputOffset < inputLength && set.Get(input[inputOffset]))

instPointer += sizeof(\*this);

else

instPointer = matcher.LabelToInstPointer(targetLabel);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int JumpIfNotSetInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: JumpIfNotSet(", label);

SetMixin::Print(w, litbuf);

w->Print(L", ");

JumpMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// MatchSetOrJumpInst (optimized instruction)

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

\_\_inline bool MatchSetOrJumpInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (inputOffset < inputLength && set.Get(input[inputOffset]))

{

inputOffset++;

instPointer += sizeof(\*this);

}

else

instPointer = matcher.LabelToInstPointer(targetLabel);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int MatchSetOrJumpInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: MatchSetOrJump(", label);

SetMixin::Print(w, litbuf);

w->Print(L", ");

JumpMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// Switch10Inst (optimized instruction)

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

\_\_inline bool Switch10Inst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

if (inputOffset >= inputLength)

return matcher.Fail(FAIL\_PARAMETERS);

#if 0

int l = 0;

int h = numCases - 1;

while (l <= h)

{

int m = (l + h) / 2;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (cases[m].c == input[inputOffset])

{

instPointer = matcher.LabelToInstPointer(cases[m].targetLabel);

return false;

}

else if (cases[m].c < input[inputOffset])

l = m + 1;

else

h = m - 1;

}

#else

const int localNumCases = numCases;

for (int i = 0; i < localNumCases; i++)

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (cases[i].c == input[inputOffset])

{

instPointer = matcher.LabelToInstPointer(cases[i].targetLabel);

return false;

}

else if (cases[i].c > input[inputOffset])

break;

}

#endif

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int Switch10Inst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: Switch10(", label);

SwitchMixin<MaxCases>::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// Switch20Inst (optimized instruction)

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

\_\_inline bool Switch20Inst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

if (inputOffset >= inputLength)

return matcher.Fail(FAIL\_PARAMETERS);

#if 0

int l = 0;

int h = numCases - 1;

while (l <= h)

{

int m = (l + h) / 2;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (cases[m].c == input[inputOffset])

{

instPointer = matcher.LabelToInstPointer(cases[m].targetLabel);

return false;

}

else if (cases[m].c < input[inputOffset])

l = m + 1;

else

h = m - 1;

}

#else

const int localNumCases = numCases;

for (int i = 0; i < localNumCases; i++)

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (cases[i].c == input[inputOffset])

{

instPointer = matcher.LabelToInstPointer(cases[i].targetLabel);

return false;

}

else if (cases[i].c > input[inputOffset])

break;

}

#endif

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int Switch20Inst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: Switch20(", label);

SwitchMixin<MaxCases>::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// SwitchAndConsume10Inst (optimized instruction)

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

\_\_inline bool SwitchAndConsume10Inst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

if (inputOffset >= inputLength)

return matcher.Fail(FAIL\_PARAMETERS);

#if 0

int l = 0;

int h = numCases - 1;

while (l <= h)

{

int m = (l + h) / 2;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (cases[m].c == input[inputOffset])

{

inputOffset++;

instPointer = matcher.LabelToInstPointer(cases[m].targetLabel);

return false;

}

else if (cases[m].c < input[inputOffset])

l = m + 1;

else

h = m - 1;

}

#else

const int localNumCases = numCases;

for (int i = 0; i < localNumCases; i++)

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (cases[i].c == input[inputOffset])

{

inputOffset++;

instPointer = matcher.LabelToInstPointer(cases[i].targetLabel);

return false;

}

else if (cases[i].c > input[inputOffset])

break;

}

#endif

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int SwitchAndConsume10Inst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: SwitchAndConsume10(", label);

SwitchMixin<MaxCases>::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// SwitchAndConsume20Inst (optimized instruction)

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

\_\_inline bool SwitchAndConsume20Inst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

if (inputOffset >= inputLength)

return matcher.Fail(FAIL\_PARAMETERS);

#if 0

int l = 0;

int h = numCases - 1;

while (l <= h)

{

int m = (l + h) / 2;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (cases[m].c == input[inputOffset])

{

inputOffset++;

instPointer = matcher.LabelToInstPointer(cases[m].targetLabel);

return false;

}

else if (cases[m].c < input[inputOffset])

l = m + 1;

else

h = m - 1;

}

#else

const int localNumCases = numCases;

for (int i = 0; i < localNumCases; i++)

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (cases[i].c == input[inputOffset])

{

inputOffset++;

instPointer = matcher.LabelToInstPointer(cases[i].targetLabel);

return false;

}

else if (cases[i].c > input[inputOffset])

break;

}

#endif

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int SwitchAndConsume20Inst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: SwitchAndConsume20(", label);

SwitchMixin<MaxCases>::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// BOITestInst

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

\_\_inline bool BOITestInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

if (inputOffset > 0)

{

if (canHardFail)

// Clearly trying to start from later in the input won't help, and we know backtracking can't take us earlier in the input

return matcher.HardFail(HARDFAIL\_PARAMETERS(ImmediateFail));

else

return matcher.Fail(FAIL\_PARAMETERS);

}

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int BOITestInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: BOITest(", label);

HardFailMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// EOITestInst

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

\_\_inline bool EOITestInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

if (inputOffset < inputLength)

{

if (canHardFail)

// We know backtracking can never take us later in the input, but starting from later in the input could help

return matcher.HardFail(HARDFAIL\_PARAMETERS(LaterOnly));

else

return matcher.Fail(FAIL\_PARAMETERS);

}

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int EOITestInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: EOITest(", label);

HardFailMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// BOLTestInst

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

\_\_inline bool BOLTestInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (inputOffset > 0 && !matcher.standardChars->IsNewline(input[inputOffset - 1]))

return matcher.Fail(FAIL\_PARAMETERS);

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int BOLTestInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->PrintEOL(L"L%04x: BOLTest()", label);

return sizeof(\*this);

}

#endif

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

// EOLTestInst

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

\_\_inline bool EOLTestInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (inputOffset < inputLength && !matcher.standardChars->IsNewline(input[inputOffset]))

return matcher.Fail(FAIL\_PARAMETERS);

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int EOLTestInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->PrintEOL(L"L%04x: EOLTest()", label);

return sizeof(\*this);

}

#endif

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

// WordBoundaryTestInst

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

\_\_inline bool WordBoundaryTestInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

const bool prev = inputOffset > 0 && matcher.standardChars->IsWord(input[inputOffset - 1]);

const bool curr = inputOffset < inputLength && matcher.standardChars->IsWord(input[inputOffset]);

if (isNegation == (prev != curr))

return matcher.Fail(FAIL\_PARAMETERS);

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int WordBoundaryTestInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->PrintEOL(L"L%04x: WordBoundaryTest(isNegation: %s)", label, isNegation ? L"true" : L"false");

return sizeof(\*this);

}

#endif

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

// MatchCharInst

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

\_\_inline bool MatchCharInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (inputOffset >= inputLength || input[inputOffset] != c)

return matcher.Fail(FAIL\_PARAMETERS);

inputOffset++;

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int MatchCharInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: MatchChar(", label);

CharMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// MatchChar2Inst

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

\_\_inline bool MatchChar2Inst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (inputOffset >= inputLength || (input[inputOffset] != cs[0] && input[inputOffset] != cs[1]))

return matcher.Fail(FAIL\_PARAMETERS);

inputOffset++;

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int MatchChar2Inst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: MatchChar2(", label);

Char2Mixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// MatchChar3Inst

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

\_\_inline bool MatchChar3Inst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (inputOffset >= inputLength || (input[inputOffset] != cs[0] && input[inputOffset] != cs[1] && input[inputOffset] != cs[2]))

return matcher.Fail(FAIL\_PARAMETERS);

inputOffset++;

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int MatchChar3Inst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: MatchChar3(", label);

Char3Mixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// MatchChar4Inst

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

\_\_inline bool MatchChar4Inst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (inputOffset >= inputLength || (input[inputOffset] != cs[0] && input[inputOffset] != cs[1] && input[inputOffset] != cs[2] && input[inputOffset] != cs[3]))

return matcher.Fail(FAIL\_PARAMETERS);

inputOffset++;

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int MatchChar4Inst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: MatchChar4(", label);

Char4Mixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// MatchSetInst

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

template<bool IsNegation>

\_\_inline bool MatchSetInst<IsNegation>::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (inputOffset >= inputLength || set.Get(input[inputOffset]) == IsNegation)

return matcher.Fail(FAIL\_PARAMETERS);

inputOffset++;

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

template<bool IsNegation>

int MatchSetInst<IsNegation>::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: MatchSet(", label);

SetMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// MatchLiteralInst

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

\_\_inline bool MatchLiteralInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

Assert(length <= matcher.program->rep.insts.litbufLen - offset);

if (length > inputLength - inputOffset)

return matcher.Fail(FAIL\_PARAMETERS);

const Char \*const literalBuffer = matcher.program->rep.insts.litbuf;

const Char \* literalCurr = literalBuffer + offset;

const Char \* inputCurr = input + inputOffset;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (\*literalCurr != \*inputCurr)

{

inputOffset++;

return matcher.Fail(FAIL\_PARAMETERS);

}

const Char \*const literalEnd = literalCurr + length;

literalCurr++;

inputCurr++;

while (literalCurr < literalEnd)

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (\*literalCurr != \*inputCurr++)

{

inputOffset = (CharCount)(inputCurr - input);

return matcher.Fail(FAIL\_PARAMETERS);

}

literalCurr++;

}

inputOffset = (CharCount)(inputCurr - input);

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int MatchLiteralInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: MatchLiteral(", label);

LiteralMixin::Print(w, litbuf, false);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// MatchLiteralEquivInst

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

\_\_inline bool MatchLiteralEquivInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

if (length > inputLength - inputOffset)

return matcher.Fail(FAIL\_PARAMETERS);

const Char \*const literalBuffer = matcher.program->rep.insts.litbuf;

CharCount literalOffset = offset;

const CharCount literalEndOffset = offset + length \* CaseInsensitive::EquivClassSize;

Assert(literalEndOffset <= matcher.program->rep.insts.litbufLen);

CompileAssert(CaseInsensitive::EquivClassSize == 4);

do

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (input[inputOffset] != literalBuffer[literalOffset]

&& input[inputOffset] != literalBuffer[literalOffset + 1]

&& input[inputOffset] != literalBuffer[literalOffset + 2]

&& input[inputOffset] != literalBuffer[literalOffset + 3])

{

return matcher.Fail(FAIL\_PARAMETERS);

}

inputOffset++;

literalOffset += CaseInsensitive::EquivClassSize;

}

while (literalOffset < literalEndOffset);

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int MatchLiteralEquivInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: MatchLiteralEquiv(", label);

LiteralMixin::Print(w, litbuf, true);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// MatchTrieInst (optimized instruction)

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

\_\_inline bool MatchTrieInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

if (!trie.Match

( input

, inputLength

, inputOffset

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, matcher.stats

#endif

))

return matcher.Fail(FAIL\_PARAMETERS);

instPointer += sizeof(\*this);

return false;

}

void MatchTrieInst::FreeBody(ArenaAllocator\* rtAllocator)

{

trie.FreeBody(rtAllocator);

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int MatchTrieInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->PrintEOL(L"L%04x: MatchTrie(", label);

trie.Print(w);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// OptMatchCharInst (optimized instruction)

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

\_\_inline bool OptMatchCharInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (inputOffset < inputLength && input[inputOffset] == c)

inputOffset++;

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int OptMatchCharInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: OptMatchChar(", label);

CharMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// OptMatchSetInst (optimized instruction)

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

\_\_inline bool OptMatchSetInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (inputOffset < inputLength && set.Get(input[inputOffset]))

inputOffset++;

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int OptMatchSetInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: OptMatchSet(", label);

SetMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// SyncToCharAndContinueInst (optimized instruction)

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

\_\_inline bool SyncToCharAndContinueInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

const Char matchC = c;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

while (inputOffset < inputLength && input[inputOffset] != matchC)

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

inputOffset++;

}

matchStart = inputOffset;

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int SyncToCharAndContinueInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: SyncToCharAndContinue(", label);

CharMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// SyncToChar2SetAndContinueInst (optimized instruction)

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

\_\_inline bool SyncToChar2SetAndContinueInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

const Char matchC0 = cs[0];

const Char matchC1 = cs[1];

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

while (inputOffset < inputLength && input[inputOffset] != matchC0 && input[inputOffset] != matchC1)

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

inputOffset++;

}

matchStart = inputOffset;

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int SyncToChar2SetAndContinueInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: SyncToChar2SetAndContinue(", label);

Char2Mixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// SyncToSetAndContinueInst (optimized instruction)

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

template<bool IsNegation>

\_\_inline bool SyncToSetAndContinueInst<IsNegation>::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

const RuntimeCharSet<Char>& matchSet = set;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

while (inputOffset < inputLength && matchSet.Get(input[inputOffset]) == IsNegation)

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

inputOffset++;

}

matchStart = inputOffset;

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

template<bool IsNegation>

int SyncToSetAndContinueInst<IsNegation>::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: SyncToSetAndContinue(", label);

SetMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// SyncToLiteralAndContinueInst (optimized instruction)

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

template <typename ScannerT>

\_\_inline bool SyncToLiteralAndContinueInstT<ScannerT>::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

if (!Match(matcher, input, inputLength, inputOffset))

return matcher.HardFail(HARDFAIL\_PARAMETERS(ImmediateFail));

matchStart = inputOffset;

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

template <typename ScannerT>

int SyncToLiteralAndContinueInstT<ScannerT>::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: SyncToLiteralAndContinue(", label);

ScannerT::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

// explicit instantiation

template struct SyncToLiteralAndContinueInstT<Char2LiteralScannerMixin>;

template struct SyncToLiteralAndContinueInstT<ScannerMixin>;

template struct SyncToLiteralAndContinueInstT<ScannerMixin\_WithLinearCharMap>;

template struct SyncToLiteralAndContinueInstT<EquivScannerMixin>;

template struct SyncToLiteralAndContinueInstT<EquivTrivialLastPatCharScannerMixin>;

#endif

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

// SyncToCharAndConsumeInst (optimized instruction)

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

\_\_inline bool SyncToCharAndConsumeInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

const Char matchC = c;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

while (inputOffset < inputLength && input[inputOffset] != matchC)

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

inputOffset++;

}

if (inputOffset >= inputLength)

return matcher.HardFail(HARDFAIL\_PARAMETERS(ImmediateFail));

matchStart = inputOffset++;

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int SyncToCharAndConsumeInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: SyncToCharAndConsume(", label);

CharMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// SyncToChar2SetAndConsumeInst (optimized instruction)

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

\_\_inline bool SyncToChar2SetAndConsumeInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

const Char matchC0 = cs[0];

const Char matchC1 = cs[1];

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

while (inputOffset < inputLength && (input[inputOffset] != matchC0 && input[inputOffset] != matchC1))

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

inputOffset++;

}

if (inputOffset >= inputLength)

return matcher.HardFail(HARDFAIL\_PARAMETERS(ImmediateFail));

matchStart = inputOffset++;

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int SyncToChar2SetAndConsumeInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: SyncToChar2SetAndConsume(", label);

Char2Mixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// SyncToSetAndConsumeInst (optimized instruction)

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

template<bool IsNegation>

\_\_inline bool SyncToSetAndConsumeInst<IsNegation>::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

const RuntimeCharSet<Char>& matchSet = set;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

while (inputOffset < inputLength && matchSet.Get(input[inputOffset]) == IsNegation)

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

inputOffset++;

}

if (inputOffset >= inputLength)

return matcher.HardFail(HARDFAIL\_PARAMETERS(ImmediateFail));

matchStart = inputOffset++;

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

template<bool IsNegation>

int SyncToSetAndConsumeInst<IsNegation>::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: SyncToSetAndConsume(", label);

SetMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// SyncToLiteralAndConsumeInst (optimized instruction)

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

template <typename ScannerT>

\_\_inline bool SyncToLiteralAndConsumeInstT<ScannerT>::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

if (!Match(matcher, input, inputLength, inputOffset))

return matcher.HardFail(HARDFAIL\_PARAMETERS(ImmediateFail));

matchStart = inputOffset;

inputOffset += ScannerT::GetLiteralLength();

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

template <typename ScannerT>

int SyncToLiteralAndConsumeInstT<ScannerT>::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: SyncToLiteralAndConsume(", label);

ScannerT::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

// explicit instantiation

template struct SyncToLiteralAndConsumeInstT<Char2LiteralScannerMixin>;

template struct SyncToLiteralAndConsumeInstT<ScannerMixin>;

template struct SyncToLiteralAndConsumeInstT<ScannerMixin\_WithLinearCharMap>;

template struct SyncToLiteralAndConsumeInstT<EquivScannerMixin>;

template struct SyncToLiteralAndConsumeInstT<EquivTrivialLastPatCharScannerMixin>;

#endif

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

// SyncToCharAndBackupInst (optimized instruction)

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

\_\_inline bool SyncToCharAndBackupInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

if (backup.lower > inputLength - matchStart)

// Even match at very end doesn't allow for minimum backup

return matcher.HardFail(HARDFAIL\_PARAMETERS(ImmediateFail));

if(inputOffset < nextSyncInputOffset)

{

// We have not yet reached the offset in the input we last synced to before backing up, so it's unnecessary to sync

// again since we'll sync to the same point in the input and back up to the same place we are at now

instPointer += sizeof(\*this);

return false;

}

if (backup.lower > inputOffset - matchStart)

// No use looking for match until minimum backup is possible

inputOffset = matchStart + backup.lower;

const Char matchC = c;

while (inputOffset < inputLength && input[inputOffset] != matchC)

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

inputOffset++;

}

if (inputOffset >= inputLength)

return matcher.HardFail(HARDFAIL\_PARAMETERS(ImmediateFail));

nextSyncInputOffset = inputOffset + 1;

if (backup.upper != CharCountFlag)

{

// Backup at most by backup.upper for new start

CharCount maxBackup = inputOffset - matchStart;

matchStart = inputOffset - min(maxBackup, (CharCount)backup.upper);

}

// else: leave start where it is

// Move input to new match start

inputOffset = matchStart;

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int SyncToCharAndBackupInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: SyncToCharAndBackup(", label);

CharMixin::Print(w, litbuf);

w->Print(L", ");

BackupMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// SyncToSetAndBackupInst (optimized instruction)

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

template<bool IsNegation>

\_\_inline bool SyncToSetAndBackupInst<IsNegation>::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

if (backup.lower > inputLength - matchStart)

// Even match at very end doesn't allow for minimum backup

return matcher.HardFail(HARDFAIL\_PARAMETERS(ImmediateFail));

if(inputOffset < nextSyncInputOffset)

{

// We have not yet reached the offset in the input we last synced to before backing up, so it's unnecessary to sync

// again since we'll sync to the same point in the input and back up to the same place we are at now

instPointer += sizeof(\*this);

return false;

}

if (backup.lower > inputOffset - matchStart)

// No use looking for match until minimum backup is possible

inputOffset = matchStart + backup.lower;

const RuntimeCharSet<Char>& matchSet = set;

while (inputOffset < inputLength && matchSet.Get(input[inputOffset]) == IsNegation)

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

inputOffset++;

}

if (inputOffset >= inputLength)

return matcher.HardFail(HARDFAIL\_PARAMETERS(ImmediateFail));

nextSyncInputOffset = inputOffset + 1;

if (backup.upper != CharCountFlag)

{

// Backup at most by backup.upper for new start

CharCount maxBackup = inputOffset - matchStart;

matchStart = inputOffset - min(maxBackup, (CharCount)backup.upper);

}

// else: leave start where it is

// Move input to new match start

inputOffset = matchStart;

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

template<bool IsNegation>

int SyncToSetAndBackupInst<IsNegation>::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: SyncToSetAndBackup(", label);

SetMixin::Print(w, litbuf);

w->Print(L", ");

BackupMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// SyncToLiteralAndBackupInst (optimized instruction)

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

template <typename ScannerT>

\_\_inline bool SyncToLiteralAndBackupInstT<ScannerT>::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

if (backup.lower > inputLength - matchStart)

// Even match at very end doesn't allow for minimum backup

return matcher.HardFail(HARDFAIL\_PARAMETERS(ImmediateFail));

if(inputOffset < nextSyncInputOffset)

{

// We have not yet reached the offset in the input we last synced to before backing up, so it's unnecessary to sync

// again since we'll sync to the same point in the input and back up to the same place we are at now

instPointer += sizeof(\*this);

return false;

}

if (backup.lower > inputOffset - matchStart)

// No use looking for match until minimum backup is possible

inputOffset = matchStart + backup.lower;

if (!Match(matcher, input, inputLength, inputOffset))

return matcher.HardFail(HARDFAIL\_PARAMETERS(ImmediateFail));

nextSyncInputOffset = inputOffset + 1;

if (backup.upper != CharCountFlag)

{

// Set new start at most backup.upper from start of literal

CharCount maxBackup = inputOffset - matchStart;

matchStart = inputOffset - min(maxBackup, (CharCount)backup.upper);

}

// else: leave start where it is

// Move input to new match start

inputOffset = matchStart;

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

template <typename ScannerT>

int SyncToLiteralAndBackupInstT<ScannerT>::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: SyncToLiteralAndBackup(", label);

ScannerT::Print(w, litbuf);

w->Print(L", ");

BackupMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

// explicit instantiation

template struct SyncToLiteralAndBackupInstT<Char2LiteralScannerMixin>;

template struct SyncToLiteralAndBackupInstT<ScannerMixin>;

template struct SyncToLiteralAndBackupInstT<ScannerMixin\_WithLinearCharMap>;

template struct SyncToLiteralAndBackupInstT<EquivScannerMixin>;

template struct SyncToLiteralAndBackupInstT<EquivTrivialLastPatCharScannerMixin>;

#endif

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

// SyncToLiteralsAndBackupInst (optimized instruction)

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

\_\_inline bool SyncToLiteralsAndBackupInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

if (backup.lower > inputLength - matchStart)

// Even match at very end doesn't allow for minimum backup

return matcher.HardFail(HARDFAIL\_PARAMETERS(ImmediateFail));

if(inputOffset < nextSyncInputOffset)

{

// We have not yet reached the offset in the input we last synced to before backing up, so it's unnecessary to sync

// again since we'll sync to the same point in the input and back up to the same place we are at now

instPointer += sizeof(\*this);

return false;

}

if (backup.lower > inputOffset - matchStart)

// No use looking for match until minimum backup is possible

inputOffset = matchStart + backup.lower;

int besti = -1;

CharCount bestMatchOffset = 0;

for (int i = 0; i < numLiterals; i++)

{

CharCount thisMatchOffset = inputOffset;

if (infos[i]->isEquivClass ?

(infos[i]->scanner.Match<CaseInsensitive::EquivClassSize>

( input

, inputLength

, thisMatchOffset

, matcher.program->rep.insts.litbuf + infos[i]->offset

, infos[i]->length

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, matcher.stats

#endif

)) :

(infos[i]->scanner.Match<1>

( input

, inputLength

, thisMatchOffset

, matcher.program->rep.insts.litbuf + infos[i]->offset

, infos[i]->length

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, matcher.stats

#endif

)))

{

if (besti < 0 || thisMatchOffset < bestMatchOffset)

{

besti = i;

bestMatchOffset = thisMatchOffset;

}

}

}

if (besti < 0)

// No literals matched

return matcher.HardFail(HARDFAIL\_PARAMETERS(ImmediateFail));

nextSyncInputOffset = bestMatchOffset + 1;

if (backup.upper != CharCountFlag)

{

// Set new start at most backup.upper from start of literal

CharCount maxBackup = bestMatchOffset - matchStart;

matchStart = bestMatchOffset - min(maxBackup, (CharCount)backup.upper);

}

// else: leave start where it is

// Move input to new match start

inputOffset = matchStart;

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int SyncToLiteralsAndBackupInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: SyncToLiteralsAndBackup(", label);

ScannersMixin::Print(w, litbuf);

w->Print(L", ");

BackupMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// MatchGroupInst

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

\_\_inline bool MatchGroupInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

GroupInfo\* const info = matcher.GroupIdToGroupInfo(groupId);

if (!info->IsUndefined() && info->length > 0)

{

if (info->length > inputLength - inputOffset)

return matcher.Fail(FAIL\_PARAMETERS);

CharCount groupOffset = info->offset;

const CharCount groupEndOffset = groupOffset + info->length;

bool isCaseInsensitiveMatch = (matcher.program->flags & IgnoreCaseRegexFlag) != 0;

bool isCodePointList = (matcher.program->flags & UnicodeRegexFlag) != 0;

// This is the only place in the runtime machinery we need to convert characters to their equivalence class

if (isCaseInsensitiveMatch && isCodePointList)

{

auto getNextCodePoint = [=](CharCount &offset, CharCount endOffset, codepoint\_t &codePoint) {

if (endOffset <= offset)

{

return false;

}

Char lowerPart = input[offset];

if (!Js::NumberUtilities::IsSurrogateLowerPart(lowerPart) || offset + 1 == endOffset)

{

codePoint = lowerPart;

offset += 1;

return true;

}

Char upperPart = input[offset + 1];

if (!Js::NumberUtilities::IsSurrogateUpperPart(upperPart))

{

codePoint = lowerPart;

offset += 1;

}

else

{

codePoint = Js::NumberUtilities::SurrogatePairAsCodePoint(lowerPart, upperPart);

offset += 2;

}

return true;

};

codepoint\_t equivs[CaseInsensitive::EquivClassSize];

while (true)

{

codepoint\_t groupCodePoint;

bool hasGroupCodePoint = getNextCodePoint(groupOffset, groupEndOffset, groupCodePoint);

if (!hasGroupCodePoint)

{

break;

}

// We don't need to verify that there is a valid input code point since at the beginning

// of the function, we make sure that the length of the input is at least as long as the

// length of the group.

codepoint\_t inputCodePoint;

getNextCodePoint(inputOffset, inputLength, inputCodePoint);

bool doesMatch = false;

if (!Js::NumberUtilities::IsInSupplementaryPlane(groupCodePoint))

{

auto toCanonical = [&](codepoint\_t c) {

return matcher.standardChars->ToCanonical(

CaseInsensitive::MappingSource::CaseFolding,

static\_cast<wchar\_t>(c));

};

doesMatch = (toCanonical(groupCodePoint) == toCanonical(inputCodePoint));

}

else

{

uint tblidx = 0;

uint acth = 0;

CaseInsensitive::RangeToEquivClass(tblidx, groupCodePoint, groupCodePoint, acth, equivs);

CompileAssert(CaseInsensitive::EquivClassSize == 4);

doesMatch =

inputCodePoint == equivs[0]

|| inputCodePoint == equivs[1]

|| inputCodePoint == equivs[2]

|| inputCodePoint == equivs[3];

}

if (!doesMatch)

{

return matcher.Fail(FAIL\_PARAMETERS);

}

}

}

else if (isCaseInsensitiveMatch)

{

do

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

auto toCanonical = [&](CharCount &offset) {

return matcher.standardChars->ToCanonical(CaseInsensitive::MappingSource::UnicodeData, input[offset++]);

};

if (toCanonical(groupOffset) != toCanonical(inputOffset))

{

return matcher.Fail(FAIL\_PARAMETERS);

}

}

while (groupOffset < groupEndOffset);

}

else

{

do

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (input[groupOffset++] != input[inputOffset++])

return matcher.Fail(FAIL\_PARAMETERS);

}

while (groupOffset < groupEndOffset);

}

}

// else: trivially match empty string

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int MatchGroupInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: MatchGroup(", label);

GroupMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// BeginDefineGroupInst

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

\_\_inline bool BeginDefineGroupInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

GroupInfo \*const groupInfo = matcher.GroupIdToGroupInfo(groupId);

Assert(groupInfo->IsUndefined());

groupInfo->offset = inputOffset;

Assert(groupInfo->IsUndefined());

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int BeginDefineGroupInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: BeginDefineGroup(", label);

GroupMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// EndDefineGroupInst

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

\_\_inline bool EndDefineGroupInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

if (!noNeedToSave)

{

// UNDO ACTION: Restore group on backtrack

PUSH(contStack, ResetGroupCont, groupId);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.PushStats(contStack, input);

#endif

}

GroupInfo \*const groupInfo = matcher.GroupIdToGroupInfo(groupId);

Assert(groupInfo->IsUndefined());

Assert(inputOffset >= groupInfo->offset);

groupInfo->length = inputOffset - groupInfo->offset;

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int EndDefineGroupInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: EndDefineGroup(", label);

GroupMixin::Print(w, litbuf);

w->Print(L", ");

NoNeedToSaveMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// DefineGroupFixedInst (optimized instruction)

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

\_\_inline bool DefineGroupFixedInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

if (!noNeedToSave)

{

// UNDO ACTION: Restore group on backtrack

PUSH(contStack, ResetGroupCont, groupId);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.PushStats(contStack, input);

#endif

}

GroupInfo \*const groupInfo = matcher.GroupIdToGroupInfo(groupId);

Assert(groupInfo->IsUndefined());

groupInfo->offset = inputOffset - length;

groupInfo->length = length;

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int DefineGroupFixedInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: DefineGroupFixed(", label);

GroupMixin::Print(w, litbuf);

w->Print(L", ");

FixedLengthMixin::Print(w, litbuf);

w->Print(L", ");

NoNeedToSaveMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// BeginLoopInst

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

\_\_inline bool BeginLoopInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

LoopInfo\* loopInfo = matcher.LoopIdToLoopInfo(loopId);

// If loop has outer loops, the continuation stack may have choicepoints from an earlier "run" of this loop

// which, when backtracked to, may expect the loopInfo state to be as it was at the time the choicepoint was

// pushed.

// - If the loop is greedy with deterministic body, there may be Resumes into the follow of the loop, but

// they won't look at the loopInfo state so there's nothing to do.

// - If the loop is greedy, or if it is non-greedy with lower > 0, AND it has a non-deterministic body,

// we may have Resume entries which will resume inside the loop body, which may then run to a

// RepeatLoop, which will then look at the loopInfo state. However, each iteration is protected by

// a RestoreLoop by RepeatLoopInst below. (\*\*\*\*)

// - If the loop is non-greedy there may be a RepeatLoop on the stack, so we must restore the loopInfo

// state before backtracking to it.

if (!isGreedy && hasOuterLoops)

{

PUSH(contStack, RestoreLoopCont, loopId, \*loopInfo);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.PushStats(contStack, input);

#endif

}

// The loop body must always begin with empty inner groups

// - if the loop is not in an outer they will be empty due to the reset when the match began

// - if the loop is in an outer loop, they will have been reset by the outer loop's RepeatLoop instruction

#if DBG

for (int i = minBodyGroupId; i <= maxBodyGroupId; i++)

{

Assert(matcher.GroupIdToGroupInfo(i)->IsUndefined());

}

#endif

loopInfo->number = 0;

loopInfo->startInputOffset = inputOffset;

if (repeats.lower == 0)

{

if (isGreedy)

{

// CHOICEPOINT: Try one iteration of body, if backtrack continue from here with no iterations

PUSH(contStack, ResumeCont, inputOffset, exitLabel);

instPointer += sizeof(\*this);

}

else

{

// CHOICEPOINT: Try no iterations of body, if backtrack do one iteration of body from here

Assert(instPointer == (uint8\*)this);

PUSH(contStack, RepeatLoopCont, matcher.InstPointerToLabel(instPointer), inputOffset);

instPointer = matcher.LabelToInstPointer(exitLabel);

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.PushStats(contStack, input);

#endif

}

else

{

// Must match minimum iterations, so continue to loop body

instPointer += sizeof(\*this);

}

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int BeginLoopInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: BeginLoop(", label);

BeginLoopMixin::Print(w, litbuf);

w->Print(L", ");

BodyGroupsMixin::Print(w, litbuf);

w->PrintEOL(L", greedy: %s)", isGreedy ? L"true" : L"false");

return sizeof(\*this);

}

#endif

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

// RepeatLoopInst

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

\_\_inline bool RepeatLoopInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

BeginLoopInst\* begin = matcher.L2I(BeginLoop, beginLabel);

LoopInfo\* loopInfo = matcher.LoopIdToLoopInfo(begin->loopId);

// See comment (\*\*\*\*) above.

if (begin->hasInnerNondet)

{

PUSH(contStack, RestoreLoopCont, begin->loopId, \*loopInfo);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.PushStats(contStack, input);

#endif

}

loopInfo->number++;

if (loopInfo->number < begin->repeats.lower)

{

// Must match another iteration of body.

loopInfo->startInputOffset = inputOffset;

if(begin->hasInnerNondet)

{

// If it backtracks into the loop body of an earlier iteration, it must restore inner groups for that iteration.

// Save the inner groups and reset them for the next iteration.

matcher.SaveInnerGroups(begin->minBodyGroupId, begin->maxBodyGroupId, true, input, contStack);

}

else

{

// If it backtracks, the entire loop will fail, so no need to restore groups. Just reset the inner groups for

// the next iteration.

matcher.ResetInnerGroups(begin->minBodyGroupId, begin->maxBodyGroupId);

}

instPointer = matcher.LabelToInstPointer(beginLabel + sizeof(BeginLoopInst));

}

else if (inputOffset == loopInfo->startInputOffset && loopInfo->number > begin->repeats.lower)

{

// The minimum number of iterations has been satisfied but the last iteration made no progress.

// - With greedy & deterministic body, FAIL so as to undo that iteration and restore group bindings.

// - With greedy & non-deterministic body, FAIL so as to try another body alternative

// - With non-greedy, we're trying an additional iteration because the follow failed. But

// since we didn't consume anything the follow will fail again, so fail

//

return matcher.Fail(FAIL\_PARAMETERS);

}

else if (begin->repeats.upper != CharCountFlag && loopInfo->number >= (CharCount)begin->repeats.upper)

{

// Success: proceed to remainder.

instPointer = matcher.LabelToInstPointer(begin->exitLabel);

}

else if (begin->isGreedy)

{

// CHOICEPOINT: Try one more iteration of body, if backtrack continue from here with no more iterations

PUSH(contStack, ResumeCont, inputOffset, begin->exitLabel);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.PushStats(contStack, input);

#endif

loopInfo->startInputOffset = inputOffset;

// If backtrack, we must continue with previous group bindings

matcher.SaveInnerGroups(begin->minBodyGroupId, begin->maxBodyGroupId, true, input, contStack);

instPointer = matcher.LabelToInstPointer(beginLabel + sizeof(BeginLoopInst));

}

else

{

// CHOICEPOINT: Try no more iterations of body, if backtrack do one more iteration of body from here

PUSH(contStack, RepeatLoopCont, beginLabel, inputOffset);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.PushStats(contStack, input);

#endif

instPointer = matcher.LabelToInstPointer(begin->exitLabel);

}

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int RepeatLoopInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: RepeatLoop(", label);

RepeatLoopMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// BeginLoopIfCharInst (optimized instruction)

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

\_\_inline bool BeginLoopIfCharInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (inputOffset < inputLength && input[inputOffset] == c)

{

// Commit to at least one iteration of loop

LoopInfo\* loopInfo = matcher.LoopIdToLoopInfo(loopId);

// All inner groups must begin reset

#if DBG

for (int i = minBodyGroupId; i <= maxBodyGroupId; i++)

{

Assert(matcher.GroupIdToGroupInfo(i)->IsUndefined());

}

#endif

loopInfo->number = 0;

instPointer += sizeof(\*this);

return false;

}

if (repeats.lower > 0)

return matcher.Fail(FAIL\_PARAMETERS);

instPointer = matcher.LabelToInstPointer(exitLabel);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int BeginLoopIfCharInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: BeginLoopIfChar(", label);

CharMixin::Print(w, litbuf);

w->Print(L", ");

BeginLoopMixin::Print(w, litbuf);

w->Print(L", ");

BodyGroupsMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// BeginLoopIfSetInst (optimized instruction)

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

\_\_inline bool BeginLoopIfSetInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (inputOffset < inputLength && set.Get(input[inputOffset]))

{

// Commit to at least one iteration of loop

LoopInfo\* loopInfo = matcher.LoopIdToLoopInfo(loopId);

// All inner groups must be begin reset

#if DBG

for (int i = minBodyGroupId; i <= maxBodyGroupId; i++)

{

Assert(matcher.GroupIdToGroupInfo(i)->IsUndefined());

}

#endif

loopInfo->startInputOffset = inputOffset;

loopInfo->number = 0;

instPointer += sizeof(\*this);

return false;

}

if (repeats.lower > 0)

return matcher.Fail(FAIL\_PARAMETERS);

instPointer = matcher.LabelToInstPointer(exitLabel);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int BeginLoopIfSetInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: BeginLoopIfSet(", label);

SetMixin::Print(w, litbuf);

w->Print(L", ");

BeginLoopMixin::Print(w, litbuf);

w->Print(L", ");

BodyGroupsMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// RepeatLoopIfCharInst (optimized instruction)

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

\_\_inline bool RepeatLoopIfCharInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

BeginLoopIfCharInst\* begin = matcher.L2I(BeginLoopIfChar, beginLabel);

LoopInfo\* loopInfo = matcher.LoopIdToLoopInfo(begin->loopId);

if (begin->hasInnerNondet)

{

// May end up backtracking into loop body for iteration just completed: see above.

PUSH(contStack, RestoreLoopCont, begin->loopId, \*loopInfo);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.PushStats(contStack, input);

#endif

}

loopInfo->number++;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (inputOffset < inputLength && input[inputOffset] == begin->c)

{

if (begin->repeats.upper != CharCountFlag && loopInfo->number >= (CharCount)begin->repeats.upper)

{

// If the loop body's first set and the loop's follow set are disjoint, we can just fail here since

// we know the next character in the input is in the loop body's first set.

return matcher.Fail(FAIL\_PARAMETERS);

}

// Commit to one more iteration

if(begin->hasInnerNondet)

{

// If it backtracks into the loop body of an earlier iteration, it must restore inner groups for that iteration.

// Save the inner groups and reset them for the next iteration.

matcher.SaveInnerGroups(begin->minBodyGroupId, begin->maxBodyGroupId, true, input, contStack);

}

else

{

// If it backtracks, the entire loop will fail, so no need to restore groups. Just reset the inner groups for

// the next iteration.

matcher.ResetInnerGroups(begin->minBodyGroupId, begin->maxBodyGroupId);

}

instPointer = matcher.LabelToInstPointer(beginLabel + sizeof(BeginLoopIfCharInst));

return false;

}

if (loopInfo->number < begin->repeats.lower)

return matcher.Fail(FAIL\_PARAMETERS);

// Proceed to exit

instPointer = matcher.LabelToInstPointer(begin->exitLabel);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int RepeatLoopIfCharInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: RepeatLoopIfChar(%d, ", label);

RepeatLoopMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// RepeatLoopIfSetInst (optimized instruction)

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

\_\_inline bool RepeatLoopIfSetInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

BeginLoopIfSetInst\* begin = matcher.L2I(BeginLoopIfSet, beginLabel);

LoopInfo\* loopInfo = matcher.LoopIdToLoopInfo(begin->loopId);

if (begin->hasInnerNondet)

{

// May end up backtracking into loop body for iteration just completed: see above.

PUSH(contStack, RestoreLoopCont, begin->loopId, \*loopInfo);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.PushStats(contStack, input);

#endif

}

loopInfo->number++;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (inputOffset < inputLength && begin->set.Get(input[inputOffset]))

{

if (begin->repeats.upper != CharCountFlag && loopInfo->number >= (CharCount)begin->repeats.upper)

{

// If the loop body's first set and the loop's follow set are disjoint, we can just fail here since

// we know the next character in the input is in the loop body's first set.

return matcher.Fail(FAIL\_PARAMETERS);

}

// Commit to one more iteration

if(begin->hasInnerNondet)

{

// If it backtracks into the loop body of an earlier iteration, it must restore inner groups for that iteration.

// Save the inner groups and reset them for the next iteration.

matcher.SaveInnerGroups(begin->minBodyGroupId, begin->maxBodyGroupId, true, input, contStack);

}

else

{

// If it backtracks, the entire loop will fail, so no need to restore groups. Just reset the inner groups for

// the next iteration.

matcher.ResetInnerGroups(begin->minBodyGroupId, begin->maxBodyGroupId);

}

instPointer = matcher.LabelToInstPointer(beginLabel + sizeof(BeginLoopIfSetInst));

return false;

}

if (loopInfo->number < begin->repeats.lower)

return matcher.Fail(FAIL\_PARAMETERS);

// Proceed to exit

instPointer = matcher.LabelToInstPointer(begin->exitLabel);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int RepeatLoopIfSetInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: RepeatLoopIfSet(", label);

RepeatLoopMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// BeginLoopFixedInst (optimized instruction)

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

\_\_inline bool BeginLoopFixedInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

LoopInfo\* loopInfo = matcher.LoopIdToLoopInfo(loopId);

// If loop is contained in an outer loop, continuation stack may already have a RewindLoopFixed entry for

// this loop. We must make sure it's state is preserved on backtrack.

if (hasOuterLoops)

{

PUSH(contStack, RestoreLoopCont, loopId, \*loopInfo);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.PushStats(contStack, input);

#endif

}

// startInputOffset will stay here for all iterations, and we'll use number of length to figure out

// where in the input to rewind to

loopInfo->number = 0;

loopInfo->startInputOffset = inputOffset;

if (repeats.lower == 0)

{

// CHOICEPOINT: Try one iteration of body. Failure of body will rewind input to here and resume with follow.

Assert(instPointer == (uint8\*)this);

PUSH(contStack, RewindLoopFixedCont, matcher.InstPointerToLabel(instPointer), true);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.PushStats(contStack, input);

#endif

}

// else: Must match minimum iterations, so continue to loop body. Failure of body signals failure of entire loop.

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int BeginLoopFixedInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: BeginLoopFixed(", label);

BeginLoopMixin::Print(w, litbuf);

w->Print(L", ");

FixedLengthMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// RepeatLoopFixedInst (optimized instruction)

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

\_\_inline bool RepeatLoopFixedInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

BeginLoopFixedInst\* begin = matcher.L2I(BeginLoopFixed, beginLabel);

LoopInfo\* loopInfo = matcher.LoopIdToLoopInfo(begin->loopId);

loopInfo->number++;

if (loopInfo->number < begin->repeats.lower)

{

// Must match another iteration of body. Failure of body signals failure of the entire loop.

instPointer = matcher.LabelToInstPointer(beginLabel + sizeof(BeginLoopFixedInst));

}

else if (begin->repeats.upper != CharCountFlag && loopInfo->number >= (CharCount)begin->repeats.upper)

{

// Matched maximum number of iterations. Continue with follow.

if (begin->repeats.lower < begin->repeats.upper)

{

// Failure of follow will try one fewer iterations (subject to repeats.lower).

// Since loop body is non-deterministic and does not define groups the rewind continuation must be on top of the stack.

Cont \*top = contStack.Top();

Assert(top != 0);

Assert(top->tag == Cont::RewindLoopFixed);

RewindLoopFixedCont\* rewind = (RewindLoopFixedCont\*)top;

rewind->tryingBody = false;

}

// else: we never pushed a rewind continuation

instPointer = matcher.LabelToInstPointer(begin->exitLabel);

}

else

{

// CHOICEPOINT: Try one more iteration of body. Failure of body will rewind input to here and

// try follow.

if (loopInfo->number == begin->repeats.lower)

{

// i.e. begin->repeats.lower > 0, so continuation won't have been pushed in BeginLoopFixed

PUSH(contStack, RewindLoopFixedCont, beginLabel, true);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.PushStats(contStack, input);

#endif

}

instPointer = matcher.LabelToInstPointer(beginLabel + sizeof(BeginLoopFixedInst));

}

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int RepeatLoopFixedInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: RepeatLoopFixed(", label);

RepeatLoopMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// LoopSetInst (optimized instruction)

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

\_\_inline bool LoopSetInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

LoopInfo\* loopInfo = matcher.LoopIdToLoopInfo(loopId);

// If loop is contained in an outer loop, continuation stack may already have a RewindLoopFixed entry for

// this loop. We must make sure it's state is preserved on backtrack.

if (hasOuterLoops)

{

PUSH(contStack, RestoreLoopCont, loopId, \*loopInfo);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.PushStats(contStack, input);

#endif

}

// startInputOffset will stay here for all iterations, and we'll use number of length to figure out

// where in the input to rewind to

loopInfo->startInputOffset = inputOffset;

// Consume as many elements of set as possible

const RuntimeCharSet<Char>& matchSet = set;

const CharCount loopMatchStart = inputOffset;

const CharCountOrFlag repeatsUpper = repeats.upper;

const CharCount inputEndOffset =

static\_cast<CharCount>(repeatsUpper) >= inputLength - inputOffset

? inputLength

: inputOffset + static\_cast<CharCount>(repeatsUpper);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

while (inputOffset < inputEndOffset && matchSet.Get(input[inputOffset]))

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

inputOffset++;

}

loopInfo->number = inputOffset - loopMatchStart;

if (loopInfo->number < repeats.lower)

return matcher.Fail(FAIL\_PARAMETERS);

if (loopInfo->number > repeats.lower)

{

// CHOICEPOINT: If follow fails, try consuming one fewer characters

Assert(instPointer == (uint8\*)this);

PUSH(contStack, RewindLoopSetCont, matcher.InstPointerToLabel(instPointer));

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.PushStats(contStack, input);

#endif

}

// else: failure of follow signals failure of entire loop

// Continue with follow

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int LoopSetInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: LoopSet(loopId: %d, ", label, loopId);

repeats.Print(w);

w->Print(L", hasOuterLoops: %s, ", hasOuterLoops ? L"true" : L"false");

SetMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// BeginLoopFixedGroupLastIterationInst (optimized instruction)

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

\_\_inline bool BeginLoopFixedGroupLastIterationInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

Assert(matcher.GroupIdToGroupInfo(groupId)->IsUndefined());

LoopInfo\* loopInfo = matcher.LoopIdToLoopInfo(loopId);

// If loop is contained in an outer loop, continuation stack may already have a RewindLoopFixedGroupLastIteration entry

// for this loop. We must make sure it's state is preserved on backtrack.

if (hasOuterLoops)

{

PUSH(contStack, RestoreLoopCont, loopId, \*loopInfo);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.PushStats(contStack, input);

#endif

}

// If loop is contained in an outer loop or assertion, we must reset the group binding if we backtrack all the way out

if (!noNeedToSave)

{

PUSH(contStack, ResetGroupCont, groupId);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.PushStats(contStack, input);

#endif

}

// startInputOffset will stay here for all iterations, and we'll use number of length to figure out

// where in the input to rewind to

loopInfo->number = 0;

loopInfo->startInputOffset = inputOffset;

if (repeats.lower == 0)

{

// CHOICEPOINT: Try one iteration of body. Failure of body will rewind input to here and resume with follow.

Assert(instPointer == (uint8\*)this);

PUSH(contStack, RewindLoopFixedGroupLastIterationCont, matcher.InstPointerToLabel(instPointer), true);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.PushStats(contStack, input);

#endif

}

// else: Must match minimum iterations, so continue to loop body. Failure of body signals failure of entire loop.

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int BeginLoopFixedGroupLastIterationInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: BeginLoopFixedGroupLastIteration(", label);

BeginLoopMixin::Print(w, litbuf);

w->Print(L", ");

FixedLengthMixin::Print(w, litbuf);

w->Print(L", ");

GroupMixin::Print(w, litbuf);

w->Print(L", ");

NoNeedToSaveMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// RepeatLoopFixedGroupLastIterationInst (optimized instruction)

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

\_\_inline bool RepeatLoopFixedGroupLastIterationInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

BeginLoopFixedGroupLastIterationInst\* begin = matcher.L2I(BeginLoopFixedGroupLastIteration, beginLabel);

LoopInfo\* loopInfo = matcher.LoopIdToLoopInfo(begin->loopId);

loopInfo->number++;

if (loopInfo->number < begin->repeats.lower)

{

// Must match another iteration of body. Failure of body signals failure of the entire loop.

instPointer = matcher.LabelToInstPointer(beginLabel + sizeof(BeginLoopFixedGroupLastIterationInst));

}

else if (begin->repeats.upper != CharCountFlag && loopInfo->number >= (CharCount)begin->repeats.upper)

{

// Matched maximum number of iterations. Continue with follow.

if (begin->repeats.lower < begin->repeats.upper)

{

// Failure of follow will try one fewer iterations (subject to repeats.lower).

// Since loop body is non-deterministic and does not define groups the rewind continuation must be on top of the stack.

Cont \*top = contStack.Top();

Assert(top != 0);

Assert(top->tag == Cont::RewindLoopFixedGroupLastIteration);

RewindLoopFixedGroupLastIterationCont\* rewind = (RewindLoopFixedGroupLastIterationCont\*)top;

rewind->tryingBody = false;

}

// else: we never pushed a rewind continuation

// Bind group

GroupInfo\* groupInfo = matcher.GroupIdToGroupInfo(begin->groupId);

groupInfo->offset = inputOffset - begin->length;

groupInfo->length = begin->length;

instPointer = matcher.LabelToInstPointer(begin->exitLabel);

}

else

{

// CHOICEPOINT: Try one more iteration of body. Failure of body will rewind input to here and

// try follow.

if (loopInfo->number == begin->repeats.lower)

{

// i.e. begin->repeats.lower > 0, so continuation won't have been pushed in BeginLoopFixed

PUSH(contStack, RewindLoopFixedGroupLastIterationCont, beginLabel, true);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.PushStats(contStack, input);

#endif

}

instPointer = matcher.LabelToInstPointer(beginLabel + sizeof(BeginLoopFixedGroupLastIterationInst));

}

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int RepeatLoopFixedGroupLastIterationInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: RepeatLoopFixedGroupLastIteration(", label);

RepeatLoopMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// BeginGreedyLoopNoBacktrackInst

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

\_\_inline bool BeginGreedyLoopNoBacktrackInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

LoopInfo\* loopInfo = matcher.LoopIdToLoopInfo(loopId);

loopInfo->number = 0;

loopInfo->startInputOffset = inputOffset;

// CHOICEPOINT: Try one iteration of body, if backtrack continue from here with no iterations

PUSH(contStack, ResumeCont, inputOffset, exitLabel);

instPointer += sizeof(\*this);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.PushStats(contStack, input);

#endif

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int BeginGreedyLoopNoBacktrackInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->PrintEOL(L"L%04x: BeginGreedyLoopNoBacktrack(loopId: %d)", label, loopId);

return sizeof(\*this);

}

#endif

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

// RepeatGreedyLoopNoBacktrackInst

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

\_\_inline bool RepeatGreedyLoopNoBacktrackInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

BeginGreedyLoopNoBacktrackInst\* begin = matcher.L2I(BeginGreedyLoopNoBacktrack, beginLabel);

LoopInfo\* loopInfo = matcher.LoopIdToLoopInfo(begin->loopId);

loopInfo->number++;

if (inputOffset == loopInfo->startInputOffset)

{

// No progress

return matcher.Fail(FAIL\_PARAMETERS);

}

else

{

// CHOICEPOINT: Try one more iteration of body, if backtrack, continue from here with no more iterations.

// Since the loop body is deterministic and group free, it wouldn't have left any continuation records.

// Therefore we can simply update the Resume continuation still on the top of the stack with the current

// input pointer.

Cont\* top = contStack.Top();

Assert(top != 0 && top->tag == Cont::Resume);

ResumeCont\* resume = (ResumeCont\*)top;

resume->origInputOffset = inputOffset;

loopInfo->startInputOffset = inputOffset;

instPointer = matcher.LabelToInstPointer(beginLabel + sizeof(BeginGreedyLoopNoBacktrackInst));

}

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int RepeatGreedyLoopNoBacktrackInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: RepeatGreedyLoopNoBacktrack(", label);

RepeatLoopMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// ChompCharInst (optimized instruction)

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

template<ChompMode Mode>

\_\_inline bool ChompCharInst<Mode>::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

const Char matchC = c;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if(Mode == ChompMode::Star || inputOffset < inputLength && input[inputOffset] == matchC)

{

while(true)

{

if(Mode != ChompMode::Star)

++inputOffset;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if(inputOffset < inputLength && input[inputOffset] == matchC)

{

if(Mode == ChompMode::Star)

++inputOffset;

continue;

}

break;

}

instPointer += sizeof(\*this);

return false;

}

return matcher.Fail(FAIL\_PARAMETERS);

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

template<ChompMode Mode>

int ChompCharInst<Mode>::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: ChompChar<%S>(", label, Mode == ChompMode::Star ? "Star" : "Plus");

CharMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// ChompSetInst (optimized instruction)

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

template<ChompMode Mode>

\_\_inline bool ChompSetInst<Mode>::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

const RuntimeCharSet<Char>& matchSet = set;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if(Mode == ChompMode::Star || inputOffset < inputLength && matchSet.Get(input[inputOffset]))

{

while(true)

{

if(Mode != ChompMode::Star)

++inputOffset;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if(inputOffset < inputLength && matchSet.Get(input[inputOffset]))

{

if(Mode == ChompMode::Star)

++inputOffset;

continue;

}

break;

}

instPointer += sizeof(\*this);

return false;

}

return matcher.Fail(FAIL\_PARAMETERS);

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

template<ChompMode Mode>

int ChompSetInst<Mode>::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: ChompSet<%S>(", label, Mode == ChompMode::Star ? "Star" : "Plus");

SetMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// ChompCharGroupInst (optimized instruction)

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

template<ChompMode Mode>

\_\_inline bool ChompCharGroupInst<Mode>::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

Assert(matcher.GroupIdToGroupInfo(groupId)->IsUndefined());

const CharCount inputStartOffset = inputOffset;

const Char matchC = c;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if(Mode == ChompMode::Star || inputOffset < inputLength && input[inputOffset] == matchC)

{

while(true)

{

if(Mode != ChompMode::Star)

++inputOffset;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if(inputOffset < inputLength && input[inputOffset] == matchC)

{

if(Mode == ChompMode::Star)

++inputOffset;

continue;

}

break;

}

if(!noNeedToSave)

{

// UNDO ACTION: Restore group on backtrack

PUSH(contStack, ResetGroupCont, groupId);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.PushStats(contStack, input);

#endif

}

GroupInfo \*const groupInfo = matcher.GroupIdToGroupInfo(groupId);

groupInfo->offset = inputStartOffset;

groupInfo->length = inputOffset - inputStartOffset;

instPointer += sizeof(\*this);

return false;

}

return matcher.Fail(FAIL\_PARAMETERS);

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

template<ChompMode Mode>

int ChompCharGroupInst<Mode>::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: ChompCharGroup<%S>(", label, Mode == ChompMode::Star ? "Star" : "Plus");

CharMixin::Print(w, litbuf);

w->Print(L", ");

GroupMixin::Print(w, litbuf);

w->Print(L", ");

NoNeedToSaveMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// ChompSetGroupInst (optimized instruction)

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

template<ChompMode Mode>

\_\_inline bool ChompSetGroupInst<Mode>::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

Assert(matcher.GroupIdToGroupInfo(groupId)->IsUndefined());

const CharCount inputStartOffset = inputOffset;

const RuntimeCharSet<Char>& matchSet = set;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if(Mode == ChompMode::Star || inputOffset < inputLength && matchSet.Get(input[inputOffset]))

{

while(true)

{

if(Mode != ChompMode::Star)

++inputOffset;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if(inputOffset < inputLength && matchSet.Get(input[inputOffset]))

{

if(Mode == ChompMode::Star)

++inputOffset;

continue;

}

break;

}

if(!noNeedToSave)

{

// UNDO ACTION: Restore group on backtrack

PUSH(contStack, ResetGroupCont, groupId);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.PushStats(contStack, input);

#endif

}

GroupInfo \*const groupInfo = matcher.GroupIdToGroupInfo(groupId);

groupInfo->offset = inputStartOffset;

groupInfo->length = inputOffset - inputStartOffset;

instPointer += sizeof(\*this);

return false;

}

return matcher.Fail(FAIL\_PARAMETERS);

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

template<ChompMode Mode>

int ChompSetGroupInst<Mode>::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: ChompSetGroup<%S>(", label, Mode == ChompMode::Star ? "Star" : "Plus");

SetMixin::Print(w, litbuf);

w->Print(L", ");

GroupMixin::Print(w, litbuf);

w->Print(L", ");

NoNeedToSaveMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// ChompCharBoundedInst (optimized instruction)

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

\_\_inline bool ChompCharBoundedInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

const Char matchC = c;

const CharCount loopMatchStart = inputOffset;

const CharCountOrFlag repeatsUpper = repeats.upper;

const CharCount inputEndOffset =

static\_cast<CharCount>(repeatsUpper) >= inputLength - inputOffset

? inputLength

: inputOffset + static\_cast<CharCount>(repeatsUpper);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

while (inputOffset < inputEndOffset && input[inputOffset] == matchC)

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

inputOffset++;

}

if (inputOffset - loopMatchStart < repeats.lower)

return matcher.Fail(FAIL\_PARAMETERS);

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int ChompCharBoundedInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: ChompCharBounded(", label);

CharMixin::Print(w, litbuf);

w->Print(L", ");

ChompBoundedMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// ChompSetBoundedInst (optimized instruction)

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

\_\_inline bool ChompSetBoundedInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

const RuntimeCharSet<Char>& matchSet = set;

const CharCount loopMatchStart = inputOffset;

const CharCountOrFlag repeatsUpper = repeats.upper;

const CharCount inputEndOffset =

static\_cast<CharCount>(repeatsUpper) >= inputLength - inputOffset

? inputLength

: inputOffset + static\_cast<CharCount>(repeatsUpper);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

while (inputOffset < inputEndOffset && matchSet.Get(input[inputOffset]))

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

inputOffset++;

}

if (inputOffset - loopMatchStart < repeats.lower)

return matcher.Fail(FAIL\_PARAMETERS);

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int ChompSetBoundedInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: ChompSetBounded(", label);

SetMixin::Print(w, litbuf);

w->Print(L", ");

ChompBoundedMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// ChompSetBoundedGroupLastCharInst (optimized instruction)

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

\_\_inline bool ChompSetBoundedGroupLastCharInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

Assert(matcher.GroupIdToGroupInfo(groupId)->IsUndefined());

const RuntimeCharSet<Char>& matchSet = set;

const CharCount loopMatchStart = inputOffset;

const CharCountOrFlag repeatsUpper = repeats.upper;

const CharCount inputEndOffset =

static\_cast<CharCount>(repeatsUpper) >= inputLength - inputOffset

? inputLength

: inputOffset + static\_cast<CharCount>(repeatsUpper);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

while (inputOffset < inputEndOffset && matchSet.Get(input[inputOffset]))

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

inputOffset++;

}

if (inputOffset - loopMatchStart < repeats.lower)

return matcher.Fail(FAIL\_PARAMETERS);

if (inputOffset > loopMatchStart)

{

if (!noNeedToSave)

{

PUSH(contStack, ResetGroupCont, groupId);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.PushStats(contStack, input);

#endif

}

GroupInfo \*const groupInfo = matcher.GroupIdToGroupInfo(groupId);

groupInfo->offset = inputOffset - 1;

groupInfo->length = 1;

}

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int ChompSetBoundedGroupLastCharInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: ChompSetBoundedGroupLastChar(", label);

SetMixin::Print(w, litbuf);

w->Print(L", ");

ChompBoundedMixin::Print(w, litbuf);

w->Print(L", ");

GroupMixin::Print(w, litbuf);

w->Print(L", ");

NoNeedToSaveMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// TryInst

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

\_\_inline bool TryInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

// CHOICEPOINT: Resume at fail label on backtrack

PUSH(contStack, ResumeCont, inputOffset, failLabel);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.PushStats(contStack, input);

#endif

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int TryInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: Try(", label);

TryMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// TryIfCharInst (optimized instruction)

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

\_\_inline bool TryIfCharInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (inputOffset < inputLength && input[inputOffset] == c)

{

// CHOICEPOINT: Resume at fail label on backtrack

PUSH(contStack, ResumeCont, inputOffset, failLabel);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.PushStats(contStack, input);

#endif

instPointer += sizeof(\*this);

return false;

}

// Proceed directly to exit

instPointer = matcher.LabelToInstPointer(failLabel);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int TryIfCharInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: TryIfChar(", label);

CharMixin::Print(w, litbuf);

w->Print(L", ");

TryMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// TryMatchCharInst (optimized instruction)

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

\_\_inline bool TryMatchCharInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (inputOffset < inputLength && input[inputOffset] == c)

{

// CHOICEPOINT: Resume at fail label on backtrack

PUSH(contStack, ResumeCont, inputOffset, failLabel);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.PushStats(contStack, input);

#endif

inputOffset++;

instPointer += sizeof(\*this);

return false;

}

// Proceed directly to exit

instPointer = matcher.LabelToInstPointer(failLabel);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int TryMatchCharInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: TryMatchChar(", label);

CharMixin::Print(w, litbuf);

w->Print(L", ");

TryMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// TryIfSetInst (optimized instruction)

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

\_\_inline bool TryIfSetInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (inputOffset < inputLength && set.Get(input[inputOffset]))

{

// CHOICEPOINT: Resume at fail label on backtrack

PUSH(contStack, ResumeCont, inputOffset, failLabel);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.PushStats(contStack, input);

#endif

instPointer += sizeof(\*this);

return false;

}

// Proceed directly to exit

instPointer = matcher.LabelToInstPointer(failLabel);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int TryIfSetInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: TryIfSet(", label);

SetMixin::Print(w, litbuf);

w->Print(L", ");

TryMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// TryMatchSetInst (optimized instruction)

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

\_\_inline bool TryMatchSetInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.CompStats();

#endif

if (inputOffset < inputLength && set.Get(input[inputOffset]))

{

// CHOICEPOINT: Resume at fail label on backtrack

PUSH(contStack, ResumeCont, inputOffset, failLabel);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.PushStats(contStack, input);

#endif

inputOffset++;

instPointer += sizeof(\*this);

return false;

}

// Proceed directly to exit

instPointer = matcher.LabelToInstPointer(failLabel);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int TryMatchSetInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: TryMatchSet(", label);

SetMixin::Print(w, litbuf);

w->Print(L", ");

TryMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// BeginAssertionInst

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

\_\_inline bool BeginAssertionInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

Assert(instPointer == (uint8\*)this);

if (!isNegation)

{

// If the positive assertion binds some groups then on success any RestoreGroup continuations pushed

// in the assertion body will be cut. Hence if the entire assertion is backtracked over we must restore

// the current inner group bindings.

matcher.SaveInnerGroups(minBodyGroupId, maxBodyGroupId, false, input, contStack);

}

PUSHA(assertionStack, AssertionInfo, matcher.InstPointerToLabel(instPointer), inputOffset, contStack.Position());

PUSH(contStack, PopAssertionCont);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.PushStats(contStack, input);

#endif

instPointer += sizeof(\*this);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int BeginAssertionInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->Print(L"L%04x: BeginAssertion(isNegation: %s, nextLabel: L%04x, ", label, isNegation ? L"true" : L"false", nextLabel);

BodyGroupsMixin::Print(w, litbuf);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// EndAssertionInst

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

\_\_inline bool EndAssertionInst::Exec(REGEX\_INST\_EXEC\_PARAMETERS) const

{

if (!matcher.PopAssertion(inputOffset, instPointer, contStack, assertionStack, true))

// Body of negative assertion succeeded, so backtrack

return matcher.Fail(FAIL\_PARAMETERS);

// else: body of positive assertion succeeded, instruction pointer already at next instruction

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int EndAssertionInst::Print(DebugWriter\* w, Label label, const Char\* litbuf) const

{

w->PrintEOL(L"L%04x: EndAssertion()", label);

return sizeof(\*this);

}

#endif

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

// Matcher state

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

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void LoopInfo::Print(DebugWriter\* w) const

{

w->Print(L"number: %u, startInputOffset: %u", number, startInputOffset);

}

#endif

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void GroupInfo::Print(DebugWriter\* w, const Char\* const input) const

{

if (IsUndefined())

w->Print(L"<undefined> (%u)", offset);

else

{

w->PrintQuotedString(input + offset, (CharCount)length);

w->Print(L" (%u+%u)", offset, (CharCount)length);

}

}

#endif

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void AssertionInfo::Print(DebugWriter\* w) const

{

w->PrintEOL(L"beginLabel: L%04x, startInputOffset: %u, contStackPosition: $llu", beginLabel, startInputOffset, static\_cast<unsigned long long>(contStackPosition));

}

#endif

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

// ResumeCont

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

\_\_inline bool ResumeCont::Exec(REGEX\_CONT\_EXEC\_PARAMETERS)

{

inputOffset = origInputOffset;

instPointer = matcher.LabelToInstPointer(origInstLabel);

return true; // STOP BACKTRACKING

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int ResumeCont::Print(DebugWriter\* w, const Char\* const input) const

{

w->PrintEOL(L"Resume(origInputOffset: %u, origInstLabel: L%04x)", origInputOffset, origInstLabel);

return sizeof(\*this);

}

#endif

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

// RestoreLoopCont

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

\_\_inline bool RestoreLoopCont::Exec(REGEX\_CONT\_EXEC\_PARAMETERS)

{

matcher.QueryContinue(qcTicks);

\*matcher.LoopIdToLoopInfo(loopId) = origLoopInfo;

return false; // KEEP BACKTRACKING

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int RestoreLoopCont::Print(DebugWriter\* w, const Char\* const input) const

{

w->Print(L"RestoreLoop(loopId: %d, ", loopId);

origLoopInfo.Print(w);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// RestoreGroupCont

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

\_\_inline bool RestoreGroupCont::Exec(REGEX\_CONT\_EXEC\_PARAMETERS)

{

\*matcher.GroupIdToGroupInfo(groupId) = origGroupInfo;

return false; // KEEP BACKTRACKING

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int RestoreGroupCont::Print(DebugWriter\* w, const Char\* const input) const

{

w->Print(L"RestoreGroup(groupId: %d, ", groupId);

origGroupInfo.Print(w, input);

w->PrintEOL(L")");

return sizeof(\*this);

}

#endif

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

// ResetGroupCont

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

\_\_inline bool ResetGroupCont::Exec(REGEX\_CONT\_EXEC\_PARAMETERS)

{

matcher.ResetGroup(groupId);

return false; // KEEP BACKTRACKING

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int ResetGroupCont::Print(DebugWriter\* w, const Char\* const input) const

{

w->PrintEOL(L"ResetGroup(groupId: %d)", groupId);

return sizeof(\*this);

}

#endif

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

// ResetGroupRangeCont

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

\_\_inline bool ResetGroupRangeCont::Exec(REGEX\_CONT\_EXEC\_PARAMETERS)

{

matcher.ResetInnerGroups(fromGroupId, toGroupId);

return false; // KEEP BACKTRACKING

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int ResetGroupRangeCont::Print(DebugWriter\* w, const Char\* const input) const

{

w->PrintEOL(L"ResetGroupRange(fromGroupId: %d, toGroupId: %d)", fromGroupId, toGroupId);

return sizeof(\*this);

}

#endif

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

// RepeatLoopCont

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

\_\_inline bool RepeatLoopCont::Exec(REGEX\_CONT\_EXEC\_PARAMETERS)

{

matcher.QueryContinue(qcTicks);

// Try one more iteration of a non-greedy loop

BeginLoopInst\* begin = matcher.L2I(BeginLoop, beginLabel);

LoopInfo\* loopInfo = matcher.LoopIdToLoopInfo(begin->loopId);

loopInfo->startInputOffset = inputOffset = origInputOffset;

instPointer = matcher.LabelToInstPointer(beginLabel + sizeof(BeginLoopInst));

if(begin->hasInnerNondet)

{

// If it backtracks into the loop body of an earlier iteration, it must restore inner groups for that iteration.

// Save the inner groups and reset them for the next iteration.

matcher.SaveInnerGroups(begin->minBodyGroupId, begin->maxBodyGroupId, true, input, contStack);

}

else

{

// If it backtracks, the entire loop will fail, so no need to restore groups. Just reset the inner groups for

// the next iteration.

matcher.ResetInnerGroups(begin->minBodyGroupId, begin->maxBodyGroupId);

}

return true; // STOP BACKTRACKING

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int RepeatLoopCont::Print(DebugWriter\* w, const Char\* const input) const

{

w->PrintEOL(L"RepeatLoop(beginLabel: L%04x, origInputOffset: %u)", beginLabel, origInputOffset);

return sizeof(\*this);

}

#endif

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

// PopAssertionCont

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

\_\_inline bool PopAssertionCont::Exec(REGEX\_CONT\_EXEC\_PARAMETERS)

{

Assert(!assertionStack.IsEmpty());

if (matcher.PopAssertion(inputOffset, instPointer, contStack, assertionStack, false))

// Body of negative assertion failed

return true; // STOP BACKTRACKING

else

// Body of positive assertion failed

return false; // CONTINUE BACKTRACKING

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int PopAssertionCont::Print(DebugWriter\* w, const Char\* const input) const

{

w->PrintEOL(L"PopAssertion()");

return sizeof(\*this);

}

#endif

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

// RewindLoopFixedCont

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

\_\_inline bool RewindLoopFixedCont::Exec(REGEX\_CONT\_EXEC\_PARAMETERS)

{

matcher.QueryContinue(qcTicks);

BeginLoopFixedInst\* begin = matcher.L2I(BeginLoopFixed, beginLabel);

LoopInfo\* loopInfo = matcher.LoopIdToLoopInfo(begin->loopId);

if (tryingBody)

{

tryingBody = false;

// loopInfo->number is the number of iterations completed before trying body

Assert(loopInfo->number >= begin->repeats.lower);

}

else

{

// loopInfo->number is the number of iterations completed before trying follow

Assert(loopInfo->number > begin->repeats.lower);

// Try follow with one fewer iteration

loopInfo->number--;

}

// Rewind input

inputOffset = loopInfo->startInputOffset + loopInfo->number \* begin->length;

if (loopInfo->number > begin->repeats.lower)

{

// Un-pop the continuation ready for next time

contStack.UnPop<RewindLoopFixedCont>();

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.UnPopStats(contStack, input);

#endif

}

// else: Can't try any fewer iterations if follow fails, so leave continuation as popped and let failure propagate

instPointer = matcher.LabelToInstPointer(begin->exitLabel);

return true; // STOP BACKTRACKING

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int RewindLoopFixedCont::Print(DebugWriter\* w, const Char\* const input) const

{

w->PrintEOL(L"RewindLoopFixed(beginLabel: L%04x, tryingBody: %s)", beginLabel, tryingBody ? L"true" : L"false");

return sizeof(\*this);

}

#endif

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

// RewindLoopSetCont

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

\_\_inline bool RewindLoopSetCont::Exec(REGEX\_CONT\_EXEC\_PARAMETERS)

{

matcher.QueryContinue(qcTicks);

LoopSetInst\* begin = matcher.L2I(LoopSet, beginLabel);

LoopInfo\* loopInfo = matcher.LoopIdToLoopInfo(begin->loopId);

// >loopInfonumber is the number of iterations completed before trying follow

Assert(loopInfo->number > begin->repeats.lower);

// Try follow with one fewer iteration

loopInfo->number--;

// Rewind input

inputOffset = loopInfo->startInputOffset + loopInfo->number;

if (loopInfo->number > begin->repeats.lower)

{

// Un-pop the continuation ready for next time

contStack.UnPop<RewindLoopSetCont>();

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.UnPopStats(contStack, input);

#endif

}

// else: Can't try any fewer iterations if follow fails, so leave continuation as popped and let failure propagate

instPointer = matcher.LabelToInstPointer(beginLabel + sizeof(LoopSetInst));

return true; // STOP BACKTRACKING

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int RewindLoopSetCont::Print(DebugWriter\* w, const Char\* const input) const

{

w->PrintEOL(L"RewindLoopSet(beginLabel: L%04x)", beginLabel);

return sizeof(\*this);

}

#endif

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

// RewindLoopFixedGroupLastIterationCont

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

\_\_inline bool RewindLoopFixedGroupLastIterationCont::Exec(REGEX\_CONT\_EXEC\_PARAMETERS)

{

matcher.QueryContinue(qcTicks);

BeginLoopFixedGroupLastIterationInst\* begin = matcher.L2I(BeginLoopFixedGroupLastIteration, beginLabel);

LoopInfo\* loopInfo = matcher.LoopIdToLoopInfo(begin->loopId);

GroupInfo\* groupInfo = matcher.GroupIdToGroupInfo(begin->groupId);

if (tryingBody)

{

tryingBody = false;

// loopInfo->number is the number of iterations completed before current attempt of body

Assert(loopInfo->number >= begin->repeats.lower);

}

else

{

// loopInfo->number is the number of iterations completed before trying follow

Assert(loopInfo->number > begin->repeats.lower);

// Try follow with one fewer iteration

loopInfo->number--;

}

// Rewind input

inputOffset = loopInfo->startInputOffset + loopInfo->number \* begin->length;

if (loopInfo->number > 0)

{

// Bind previous iteration's body

groupInfo->offset = inputOffset - begin->length;

groupInfo->length = begin->length;

}

else

groupInfo->Reset();

if (loopInfo->number > begin->repeats.lower)

{

// Un-pop the continuation ready for next time

contStack.UnPop<RewindLoopFixedGroupLastIterationCont>();

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

matcher.UnPopStats(contStack, input);

#endif

}

// else: Can't try any fewer iterations if follow fails, so leave continuation as popped and let failure propagate

instPointer = matcher.LabelToInstPointer(begin->exitLabel);

return true; // STOP BACKTRACKING

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

int RewindLoopFixedGroupLastIterationCont::Print(DebugWriter\* w, const Char\* const input) const

{

w->PrintEOL(L"RewindLoopFixedGroupLastIteration(beginLabel: L%04x, tryingBody: %s)", beginLabel, tryingBody ? L"true" : L"false");

return sizeof(\*this);

}

#endif

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

// Matcher

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

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void ContStack::Print(DebugWriter\* w, const Char\* const input) const

{

for(Iterator it(\*this); it; ++it)

{

w->Print(L"%4llu: ", static\_cast<unsigned long long>(it.Position()));

it->Print(w, input);

}

}

#endif

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void AssertionStack::Print(DebugWriter\* w, const Matcher\* matcher) const

{

for(Iterator it(\*this); it; ++it)

{

it->Print(w);

}

}

#endif

Matcher::Matcher(Js::ScriptContext\* scriptContext, RegexPattern\* pattern)

: pattern(pattern)

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

, program(pattern->rep.unified.program)

, groupInfos(nullptr)

, loopInfos(nullptr)

, previousQcTime(0)

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, stats(0)

, w(0)

#endif

{

const auto recycler = scriptContext->GetRecycler();

// Don't need to zero out - the constructor for GroupInfo should take care of it

groupInfos = RecyclerNewArrayLeaf(recycler, GroupInfo, program->numGroups);

if (program->numLoops > 0)

{

loopInfos = RecyclerNewArrayLeafZ(recycler, LoopInfo, program->numLoops);

}

}

Matcher \*Matcher::New(Js::ScriptContext\* scriptContext, RegexPattern\* pattern)

{

return RecyclerNew(scriptContext->GetRecycler(), Matcher, scriptContext, pattern);

}

Matcher \*Matcher::CloneToScriptContext(Js::ScriptContext \*scriptContext, RegexPattern \*pattern)

{

Matcher \*result = New(scriptContext, pattern);

if (groupInfos)

{

size\_t size = program->numGroups \* sizeof(GroupInfo);

js\_memcpy\_s(result->groupInfos, size, groupInfos, size);

}

if (loopInfos)

{

size\_t size = program->numLoops \* sizeof(LoopInfo);

js\_memcpy\_s(result->loopInfos, size, loopInfos, size);

}

return result;

}

#if DBG

const uint32 contTags[] = {

#define M(O) Cont::O,

#include "RegexContcodes.h"

#undef M

};

const uint32 minContTag = contTags[0];

const uint32 maxContTag = contTags[(sizeof(contTags) / sizeof(uint32)) - 1];

#endif

void Matcher::DoQueryContinue(const uint qcTicks)

{

// See definition of TimePerQc for description of regex QC heuristics

const uint before = previousQcTime;

const uint now = GetTickCount();

if((!before || now - before < TimePerQc) && qcTicks & TicksPerQc - 1)

return;

previousQcTime = now;

TraceQueryContinue(now);

// Query-continue can be reentrant and run the same regex again. To prevent the matcher and other persistent objects

// from being reused reentrantly, save and restore them around the QC call.

class AutoCleanup

{

private:

RegexPattern \*const pattern;

Matcher \*const matcher;

RegexStacks \* regexStacks;

public:

AutoCleanup(RegexPattern \*const pattern, Matcher \*const matcher) : pattern(pattern), matcher(matcher)

{

Assert(pattern);

Assert(matcher);

Assert(pattern->rep.unified.matcher == matcher);

pattern->rep.unified.matcher = nullptr;

const auto scriptContext = pattern->GetScriptContext();

regexStacks = scriptContext->SaveRegexStacks();

}

~AutoCleanup()

{

pattern->rep.unified.matcher = matcher;

const auto scriptContext = pattern->GetScriptContext();

scriptContext->RestoreRegexStacks(regexStacks);

}

} autoCleanup(pattern, this);

pattern->GetScriptContext()->GetThreadContext()->CheckScriptInterrupt();

}

void Matcher::TraceQueryContinue(const uint now)

{

if(!PHASE\_TRACE1(Js::RegexQcPhase))

return;

Output::Print(L"Regex QC");

static uint n = 0;

static uint firstQcTime = 0;

++n;

if(firstQcTime)

Output::Print(L" - frequency: %0.1f", static\_cast<double>(n \* 1000) / (now - firstQcTime));

else

firstQcTime = now;

Output::Print(L"\n");

Output::Flush();

}

bool Matcher::Fail(const Char\* const input, CharCount &inputOffset, const uint8 \*&instPointer, ContStack &contStack, AssertionStack &assertionStack, uint &qcTicks)

{

if (!contStack.IsEmpty())

{

if (!RunContStack(input, inputOffset, instPointer, contStack, assertionStack, qcTicks))

{

return false;

}

}

Assert(assertionStack.IsEmpty());

groupInfos[0].Reset();

return true; // STOP EXECUTION

}

\_\_inline bool Matcher::RunContStack(const Char\* const input, CharCount &inputOffset, const uint8 \*&instPointer, ContStack &contStack, AssertionStack &assertionStack, uint &qcTicks)

{

while (true)

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

PopStats(contStack, input);

#endif

Cont\* cont = contStack.Pop();

if (cont == 0)

break;

Assert(cont->tag >= minContTag && cont->tag <= maxContTag);

// All these cases RESUME EXECUTION if backtracking finds a stop point

const Cont::ContTag tag = cont->tag;

switch (tag)

{

#define M(O) case Cont::O: if (((O##Cont\*)cont)->Exec(\*this, input, inputOffset, instPointer, contStack, assertionStack, qcTicks)) return false; break;

#include "RegexContcodes.h"

#undef M

default:

Assert(false); // should never be reached

return false; // however, can't use complier optimization if we wnat to return false here

}

}

return true;

}

#if DBG

const uint32 instTags[] = {

#define M(TagName) Inst::TagName,

#define MTemplate(TagName, ...) M(TagName)

#include "RegexOpcodes.h"

#undef M

#undef MTemplate

};

const uint32 minInstTag = instTags[0];

const uint32 maxInstTag = instTags[(sizeof(instTags) / sizeof(uint32)) - 1];

#endif

\_\_inline void Matcher::Run(const Char\* const input, const CharCount inputLength, CharCount &matchStart, CharCount &nextSyncInputOffset, ContStack &contStack, AssertionStack &assertionStack, uint &qcTicks)

{

CharCount inputOffset = matchStart;

const uint8 \*instPointer = program->rep.insts.insts;

Assert(instPointer != 0);

while (true)

{

Assert(inputOffset >= matchStart && inputOffset <= inputLength);

Assert(instPointer >= program->rep.insts.insts && instPointer < program->rep.insts.insts + program->rep.insts.instsLen);

Assert(((Inst\*)instPointer)->tag >= minInstTag && ((Inst\*)instPointer)->tag <= maxInstTag);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

if (w != 0)

Print(w, input, inputLength, inputOffset, instPointer, contStack, assertionStack);

InstStats();

#endif

const Inst \*inst = (const Inst\*)instPointer;

const Inst::InstTag tag = inst->tag;

switch (tag)

{

#define MBase(TagName, ClassName) \

case Inst::TagName: \

if (((const ClassName \*)inst)->Exec(\*this, input, inputLength, matchStart, inputOffset, nextSyncInputOffset, instPointer, contStack, assertionStack, qcTicks)) \

return; \

break;

#define M(TagName) MBase(TagName, TagName##Inst)

#define MTemplate(TagName, TemplateDeclaration, GenericClassName, SpecializedClassName) MBase(TagName, SpecializedClassName)

#include "RegexOpcodes.h"

#undef MBase

#undef M

#undef MTemplate

default:

Assert(false);

\_\_assume(false);

}

}

}

#if DBG

void Matcher::ResetLoopInfos()

{

for (int i = 0; i < program->numLoops; i++)

loopInfos[i].Reset();

}

#endif

\_\_inline bool Matcher::MatchHere(const Char\* const input, const CharCount inputLength, CharCount &matchStart, CharCount &nextSyncInputOffset, ContStack &contStack, AssertionStack &assertionStack, uint &qcTicks)

{

// Reset the continuation and assertion stacks ready for fresh run

// NOTE: We used to do this after the Run, but it's safer to do it here in case unusual control flow exits

// the matcher without executing the clears.

contStack.Clear();

// assertionStack may be non-empty since we can hard fail directly out of matcher without popping assertion

assertionStack.Clear();

Assert(contStack.IsEmpty());

Assert(assertionStack.IsEmpty());

ResetInnerGroups(0, program->numGroups - 1);

#if DBG

ResetLoopInfos();

#endif

Run(input, inputLength, matchStart, nextSyncInputOffset, contStack, assertionStack, qcTicks);

// Leave the continuation and assertion stack memory in place so we don't have to alloc next time

return WasLastMatchSuccessful();

}

\_\_inline bool Matcher::MatchSingleCharCaseInsensitive(const Char\* const input, const CharCount inputLength, CharCount offset, const Char c)

{

CaseInsensitive::MappingSource mappingSource = program->GetCaseMappingSource();

// If sticky flag is present, break since the 1st character didn't match the pattern character

if ((program->flags & StickyRegexFlag) != 0)

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

CompStats();

#endif

if (MatchSingleCharCaseInsensitiveHere(mappingSource, input, offset, c))

{

GroupInfo\* const info = GroupIdToGroupInfo(0);

info->offset = offset;

info->length = 1;

return true;

}

else

{

ResetGroup(0);

return false;

}

}

while (offset < inputLength)

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

CompStats();

#endif

if (MatchSingleCharCaseInsensitiveHere(mappingSource, input, offset, c))

{

GroupInfo\* const info = GroupIdToGroupInfo(0);

info->offset = offset;

info->length = 1;

return true;

}

offset++;

}

ResetGroup(0);

return false;

}

\_\_inline bool Matcher::MatchSingleCharCaseInsensitiveHere(

CaseInsensitive::MappingSource mappingSource,

const Char\* const input,

const CharCount offset,

const Char c)

{

return (standardChars->ToCanonical(mappingSource, input[offset]) == standardChars->ToCanonical(mappingSource, c));

}

\_\_inline bool Matcher::MatchSingleCharCaseSensitive(const Char\* const input, const CharCount inputLength, CharCount offset, const Char c)

{

// If sticky flag is present, break since the 1st character didn't match the pattern character

if ((program->flags & StickyRegexFlag) != 0)

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

CompStats();

#endif

if (input[offset] == c)

{

GroupInfo\* const info = GroupIdToGroupInfo(0);

info->offset = offset;

info->length = 1;

return true;

}

else

{

ResetGroup(0);

return false;

}

}

while (offset < inputLength)

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

CompStats();

#endif

if (input[offset] == c)

{

GroupInfo\* const info = GroupIdToGroupInfo(0);

info->offset = offset;

info->length = 1;

return true;

}

offset++;

}

ResetGroup(0);

return false;

}

\_\_inline bool Matcher::MatchBoundedWord(const Char\* const input, const CharCount inputLength, CharCount offset)

{

const StandardChars<Char>& stdchrs = \*standardChars;

if (offset >= inputLength)

{

ResetGroup(0);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

CompStats();

#endif

if ((offset == 0 && stdchrs.IsWord(input[0])) ||

(offset > 0 && (!stdchrs.IsWord(input[offset - 1]) && stdchrs.IsWord(input[offset]))))

{

// Already at start of word

}

// If sticky flag is present, return false since we are not at the beginning of the word yet

else if ((program->flags & StickyRegexFlag) == StickyRegexFlag)

{

ResetGroup(0);

return false;

}

else

{

if (stdchrs.IsWord(input[offset]))

{

// Scan for end of current word

while (true)

{

offset++;

if (offset >= inputLength)

{

ResetGroup(0);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

CompStats();

#endif

if (!stdchrs.IsWord(input[offset]))

break;

}

}

// Scan for start of next word

while (true)

{

offset++;

if (offset >= inputLength)

{

ResetGroup(0);

return false;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

CompStats();

#endif

if (stdchrs.IsWord(input[offset]))

break;

}

}

GroupInfo\* const info = GroupIdToGroupInfo(0);

info->offset = offset;

// Scan for end of word

do

{

offset++;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

CompStats();

#endif

}

while (offset < inputLength && stdchrs.IsWord(input[offset]));

info->length = offset - info->offset;

return true;

}

\_\_inline bool Matcher::MatchLeadingTrailingSpaces(const Char\* const input, const CharCount inputLength, CharCount offset)

{

GroupInfo\* const info = GroupIdToGroupInfo(0);

Assert(offset <= inputLength);

Assert((program->flags & MultilineRegexFlag) == 0);

if (offset >= inputLength)

{

Assert(offset == inputLength);

if (program->rep.leadingTrailingSpaces.endMinMatch == 0 ||

(offset == 0 && program->rep.leadingTrailingSpaces.beginMinMatch == 0))

{

info->offset = offset;

info->length = 0;

return true;

}

info->Reset();

return false;

}

const StandardChars<Char> &stdchrs = \*standardChars;

if (offset == 0)

{

while (offset < inputLength && stdchrs.IsWhitespaceOrNewline(input[offset]))

{

offset++;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

CompStats();

#endif

}

if (offset >= program->rep.leadingTrailingSpaces.beginMinMatch)

{

info->offset = 0;

info->length = offset;

return true;

}

}

Assert(inputLength > 0);

const CharCount initOffset = offset;

offset = inputLength - 1;

while (offset >= initOffset && stdchrs.IsWhitespaceOrNewline(input[offset]))

{

// This can never underflow since initOffset > 0

Assert(offset > 0);

offset--;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

CompStats();

#endif

}

offset++;

CharCount length = inputLength - offset;

if (length >= program->rep.leadingTrailingSpaces.endMinMatch)

{

info->offset = offset;

info->length = length;

return true;

}

info->Reset();

return false;

}

\_\_inline bool Matcher::MatchOctoquad(const Char\* const input, const CharCount inputLength, CharCount offset, OctoquadMatcher\* matcher)

{

if (matcher->Match

( input

, inputLength

, offset

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, stats

#endif

))

{

GroupInfo\* const info = GroupIdToGroupInfo(0);

info->offset = offset;

info->length = TrigramInfo::PatternLength;

return true;

}

else

{

ResetGroup(0);

return false;

}

}

\_\_inline bool Matcher::MatchBOILiteral2(const Char\* const input, const CharCount inputLength, CharCount offset, DWORD literal2)

{

if (offset == 0 && inputLength >= 2)

{

CompileAssert(sizeof(Char) == 2);

const Program \* program = this->program;

if (program->rep.boiLiteral2.literal == \*(DWORD \*)input)

{

GroupInfo\* const info = GroupIdToGroupInfo(0);

info->offset = 0;

info->length = 2;

return true;

}

}

ResetGroup(0);

return false;

}

bool Matcher::Match

( const Char\* const input

, const CharCount inputLength

, CharCount offset

, Js::ScriptContext \* scriptContext

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, RegexStats\* stats

, DebugWriter\* w

#endif

)

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

this->stats = stats;

this->w = w;

#endif

Assert(offset <= inputLength);

bool res;

bool loopMatchHere = true;

Program const \*prog = this->program;

bool isStickyPresent = this->pattern->IsSticky();

switch (prog->tag)

{

case Program::BOIInstructionsTag:

if (offset != 0)

{

groupInfos[0].Reset();

res = false;

break;

}

// fall through

case Program::BOIInstructionsForStickyFlagTag:

AssertMsg(prog->tag == Program::BOIInstructionsTag || isStickyPresent, "prog->tag should be BOIInstructionsForStickyFlagTag if sticky = true.");

loopMatchHere = false;

// fall through

case Program::InstructionsTag:

{

previousQcTime = 0;

uint qcTicks = 0;

// This is the next offset in the input from where we will try to sync. For sync instructions that back up, this

// is used to avoid trying to sync when we have not yet reached the offset in the input we last synced to before

// backing up.

CharCount nextSyncInputOffset = offset;

RegexStacks \* regexStacks = scriptContext->RegexStacks();

// Need to continue matching even if matchStart == inputLim since some patterns may match an empty string at the end

// of the input. For instance: /a\*$/.exec("b")

do

{

// Let there be only one call to MatchHere(), as that call expands the interpreter loop in-place. Having

// multiple calls to MatchHere() would bloat the code.

res = MatchHere(input, inputLength, offset, nextSyncInputOffset, regexStacks->contStack, regexStacks->assertionStack, qcTicks);

} while(!res && loopMatchHere && ++offset <= inputLength);

break;

}

case Program::SingleCharTag:

if (this->pattern->IsIgnoreCase())

{

res = MatchSingleCharCaseInsensitive(input, inputLength, offset, prog->rep.singleChar.c);

}

else

{

res = MatchSingleCharCaseSensitive(input, inputLength, offset, prog->rep.singleChar.c);

}

break;

case Program::BoundedWordTag:

res = MatchBoundedWord(input, inputLength, offset);

break;

case Program::LeadingTrailingSpacesTag:

res = MatchLeadingTrailingSpaces(input, inputLength, offset);

break;

case Program::OctoquadTag:

res = MatchOctoquad(input, inputLength, offset, prog->rep.octoquad.matcher);

break;

case Program::BOILiteral2Tag:

res = MatchBOILiteral2(input, inputLength, offset, prog->rep.boiLiteral2.literal);

break;

default:

Assert(false);

\_\_assume(false);

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

this->stats = 0;

this->w = 0;

#endif

return res;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Matcher::Print(DebugWriter\* w, const Char\* const input, const CharCount inputLength, CharCount inputOffset, const uint8\* instPointer, ContStack &contStack, AssertionStack &assertionStack) const

{

w->PrintEOL(L"Matcher {");

w->Indent();

w->Print(L"program: ");

w->PrintQuotedString(program->source, program->sourceLen);

w->EOL();

w->Print(L"inputPointer: ");

if (inputLength == 0)

w->PrintEOL(L"<empty input>");

else if (inputLength > 1024)

w->PrintEOL(L"<string too large>");

else

{

w->PrintEscapedString(input, inputOffset);

if (inputOffset >= inputLength)

w->Print(L"<<<>>>");

else

{

w->Print(L"<<<");

w->PrintEscapedChar(input[inputOffset]);

w->Print(L">>>");

w->PrintEscapedString(input + inputOffset + 1, inputLength - inputOffset - 1);

}

w->EOL();

}

if (program->tag == Program::BOIInstructionsTag || program->tag == Program::InstructionsTag)

{

w->Print(L"instPointer: ");

((const Inst\*)instPointer)->Print(w, InstPointerToLabel(instPointer), program->rep.insts.litbuf);

w->PrintEOL(L"groups:");

w->Indent();

for (int i = 0; i < program->numGroups; i++)

{

w->Print(L"%d: ", i);

groupInfos[i].Print(w, input);

w->EOL();

}

w->Unindent();

w->PrintEOL(L"loops:");

w->Indent();

for (int i = 0; i < program->numLoops; i++)

{

w->Print(L"%d: ", i);

loopInfos[i].Print(w);

w->EOL();

}

w->Unindent();

w->PrintEOL(L"contStack: (top to bottom)");

w->Indent();

contStack.Print(w, input);

w->Unindent();

w->PrintEOL(L"assertionStack: (top to bottom)");

w->Indent();

assertionStack.Print(w, this);

w->Unindent();

}

w->Unindent();

w->PrintEOL(L"}");

w->Flush();

}

#endif

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

// Program

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

Program::Program(RegexFlags flags)

: source(0)

, sourceLen(0)

, flags(flags)

, numGroups(0)

, numLoops(0)

{

tag = InstructionsTag;

rep.insts.insts = 0;

rep.insts.instsLen = 0;

rep.insts.litbuf = 0;

rep.insts.litbufLen = 0;

rep.insts.scannersForSyncToLiterals = 0;

}

Program \*Program::New(Recycler \*recycler, RegexFlags flags)

{

return RecyclerNew(recycler, Program, flags);

}

ScannerInfo \*\*Program::CreateScannerArrayForSyncToLiterals(Recycler \*const recycler)

{

Assert(tag == InstructionsTag);

Assert(!rep.insts.scannersForSyncToLiterals);

Assert(recycler);

return

rep.insts.scannersForSyncToLiterals =

RecyclerNewArrayZ(recycler, ScannerInfo \*, ScannersMixin::MaxNumSyncLiterals);

}

ScannerInfo \*Program::AddScannerForSyncToLiterals(

Recycler \*const recycler,

const int scannerIndex,

const CharCount offset,

const CharCount length,

const bool isEquivClass)

{

Assert(tag == InstructionsTag);

Assert(rep.insts.scannersForSyncToLiterals);

Assert(recycler);

Assert(scannerIndex >= 0);

Assert(scannerIndex < ScannersMixin::MaxNumSyncLiterals);

Assert(!rep.insts.scannersForSyncToLiterals[scannerIndex]);

return

rep.insts.scannersForSyncToLiterals[scannerIndex] =

RecyclerNewLeaf(recycler, ScannerInfo, offset, length, isEquivClass);

}

void Program::FreeBody(ArenaAllocator\* rtAllocator)

{

if(tag != InstructionsTag || !rep.insts.insts)

return;

Inst \*inst = reinterpret\_cast<Inst \*>(rep.insts.insts);

const auto instEnd = reinterpret\_cast<Inst \*>(reinterpret\_cast<uint8 \*>(inst) + rep.insts.instsLen);

Assert(inst < instEnd);

do

{

switch(inst->tag)

{

#define MBase(TagName, ClassName) \

case Inst::TagName: \

{ \

const auto actualInst = static\_cast<ClassName \*>(inst); \

actualInst->FreeBody(rtAllocator); \

inst = actualInst + 1; \

break; \

}

#define M(TagName) MBase(TagName, TagName##Inst)

#define MTemplate(TagName, TemplateDeclaration, GenericClassName, SpecializedClassName) MBase(TagName, SpecializedClassName)

#include "RegexOpcodes.h"

#undef MBase

#undef M

#undef MTemplate

default:

Assert(false);

\_\_assume(false);

}

} while(inst < instEnd);

Assert(inst == instEnd);

#if DBG

rep.insts.insts = 0;

rep.insts.instsLen = 0;

#endif

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Program::Print(DebugWriter\* w)

{

w->PrintEOL(L"Program {");

w->Indent();

w->PrintEOL(L"source: %s", source);

w->Print(L"flags: ");

if ((flags & GlobalRegexFlag) != 0) w->Print(L"global ");

if ((flags & MultilineRegexFlag) != 0) w->Print(L"multiline ");

if ((flags & IgnoreCaseRegexFlag) != 0) w->Print(L"ignorecase");

if ((flags & UnicodeRegexFlag) != 0) w->Print(L"unicode");

if ((flags & StickyRegexFlag) != 0) w->Print(L"sticky");

w->EOL();

w->PrintEOL(L"numGroups: %d", numGroups);

w->PrintEOL(L"numLoops: %d", numLoops);

switch (tag)

{

case BOIInstructionsTag:

case InstructionsTag:

{

w->PrintEOL(L"instructions: {");

w->Indent();

if (tag == BOIInstructionsTag)

{

w->PrintEOL(L" BOITest(hardFail: true)");

}

uint8\* instsLim = rep.insts.insts + rep.insts.instsLen;

uint8\* curr = rep.insts.insts;

while (curr != instsLim)

curr += ((Inst\*)curr)->Print(w, (Label)(curr - rep.insts.insts), rep.insts.litbuf);

w->Unindent();

w->PrintEOL(L"}");

}

break;

case SingleCharTag:

w->Print(L"special form: <match single char ");

w->PrintQuotedChar(rep.singleChar.c);

w->PrintEOL(L">");

break;

case BoundedWordTag:

w->PrintEOL(L"special form: <match bounded word>");

break;

case LeadingTrailingSpacesTag:

w->PrintEOL(L"special form: <match leading/trailing spaces: minBegin=%d minEnd=%d>",

rep.leadingTrailingSpaces.beginMinMatch, rep.leadingTrailingSpaces.endMinMatch);

break;

case OctoquadTag:

w->Print(L"special form: <octoquad ");

rep.octoquad.matcher->Print(w);

w->PrintEOL(L">");

break;

}

w->Unindent();

w->PrintEOL(L"}");

}

#endif

#define M(...)

#define MTemplate(TagName, TemplateDeclaration, GenericClassName, SpecializedClassName) template struct SpecializedClassName;

#include "RegexOpcodes.h"

#undef M

#undef MTemplate

}

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

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

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

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

//

// Regex programs and their execution context

//

#pragma once

namespace UnifiedRegex

{

typedef CharCount Label;

// FORWARD

struct ScannerInfo;

class ContStack;

class AssertionStack;

class OctoquadMatcher;

enum class ChompMode : uint8

{

Star, // min = 0, max = infinite

Plus // min = 1, max = infinite

};

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

// Programs

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

struct Program : private Chars<wchar\_t>

{

friend class Compiler;

friend struct MatchLiteralNode;

friend struct AltNode;

friend class Matcher;

friend struct LoopInfo;

template <typename ScannerT>

friend struct SyncToLiteralAndConsumeInstT;

template <typename ScannerT>

friend struct SyncToLiteralAndContinueInstT;

template <typename ScannerT>

friend struct SyncToLiteralAndBackupInstT;

template <typename ScannerT>

friend struct ScannerMixinT;

template <uint lastPatCharEquivClassSize>

friend struct EquivScannerMixinT;

#define M(TagName) friend struct TagName##Inst;

#define MTemplate(TagName, TemplateDeclaration, GenericClassName, ...) TemplateDeclaration friend struct GenericClassName;

#include "RegexOpcodes.h"

#undef M

#undef MTemplate

public:

// Copy of original text of regex (without delimiting '/'s or trailing flags), null terminated.

// In run-time allocator, owned by program

Char\* source;

CharCount sourceLen; // length in wchar\_t's, NOT including terminating null

// Number of capturing groups (including implicit overall group at index 0)

int numGroups;

int numLoops;

RegexFlags flags;

private:

enum ProgramTag : uint8

{

InstructionsTag,

BOIInstructionsTag,

BOIInstructionsForStickyFlagTag,

SingleCharTag,

BoundedWordTag,

LeadingTrailingSpacesTag,

OctoquadTag,

BOILiteral2Tag

};

ProgramTag tag;

struct Instructions

{

// Instruction array, in run-time allocator, owned by program, never null

uint8\* insts;

CharCount instsLen; // in bytes

// Literals

// In run-time allocator, owned by program, may be 0

CharCount litbufLen; // length of litbuf in wchar\_t's, no terminating null

Char\* litbuf;

// These scanner infos are used by ScannersMixin, which is used by only SyncToLiteralsAndBackupInst. There will only

// ever be only one of those instructions per program. Since scanners are large (> 1 KB), for that instruction they

// are allocated on the recycler with pointers stored here to reference them.

ScannerInfo \*\*scannersForSyncToLiterals;

};

struct SingleChar

{

Char c;

uint8 padding[sizeof(Instructions) - sizeof(Char)];

};

struct Octoquad

{

OctoquadMatcher\* matcher;

uint8 padding[sizeof(Instructions) - sizeof(void\*)];

};

struct BOILiteral2

{

DWORD literal;

uint8 padding[sizeof(Instructions) - sizeof(DWORD)];

};

struct LeadingTrailingSpaces

{

CharCount beginMinMatch;

CharCount endMinMatch;

uint8 padding[sizeof(Instructions) - (sizeof(CharCount) \* 2)];

};

struct Other

{

uint8 padding[sizeof(Instructions)];

};

union

{

Instructions insts;

SingleChar singleChar;

Octoquad octoquad;

BOILiteral2 boiLiteral2;

LeadingTrailingSpaces leadingTrailingSpaces;

Other other;

} rep;

public:

Program(RegexFlags flags);

static Program \*New(Recycler \*recycler, RegexFlags flags);

static size\_t GetOffsetOfTag() { return offsetof(Program, tag); }

static size\_t GetOffsetOfRep() { return offsetof(Program, rep); }

static size\_t GetOffsetOfBOILiteral2Literal() { return offsetof(BOILiteral2, literal); }

static ProgramTag GetBOILiteral2Tag() { return ProgramTag::BOILiteral2Tag; }

ScannerInfo \*\*CreateScannerArrayForSyncToLiterals(Recycler \*const recycler);

ScannerInfo \*AddScannerForSyncToLiterals(

Recycler \*const recycler,

const int scannerIndex,

const CharCount offset,

const CharCount length,

const bool isEquivClass);

void FreeBody(ArenaAllocator\* rtAllocator);

inline CaseInsensitive::MappingSource GetCaseMappingSource() const

{

return (flags & UnicodeRegexFlag) != 0

? CaseInsensitive::MappingSource::CaseFolding

: CaseInsensitive::MappingSource::UnicodeData;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w);

#endif

};

class Matcher;

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

// CountDomain

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

struct CountDomain : private Chars<wchar\_t>

{

CharCount lower;

CharCountOrFlag upper; // CharCountFlag => unbounded

inline CountDomain() : lower(0), upper(CharCountFlag) {}

inline CountDomain(CharCount exact) : lower(exact), upper(exact) {}

inline CountDomain(CharCount lower, CharCountOrFlag upper) : lower(lower), upper(upper) {}

inline void Exact(CharCount n)

{

lower = upper = n;

}

inline void Unknown()

{

lower = 0;

upper = CharCountFlag;

}

inline void Lub(const CountDomain& other)

{

lower = min(lower, other.lower);

upper = upper == CharCountFlag || other.upper == CharCountFlag ? CharCountFlag : max(upper, other.upper);

}

inline void Add(const CountDomain& other)

{

lower = lower + other.lower;

upper = upper == CharCountFlag || other.upper == CharCountFlag ? CharCountFlag : upper + other.upper;

}

inline void Sub(const CountDomain& other)

{

lower = other.upper == CharCountFlag || other.upper > lower ? 0 : lower - other.upper;

upper = upper == CharCountFlag ? CharCountFlag : (other.lower > upper ? 0 : upper - other.lower);

}

inline void Mult(const CountDomain& other)

{

if (lower != 0)

{

CharCount maxOther = MaxCharCount / lower;

if (other.lower > maxOther)

// Clip to maximum

lower = MaxCharCount;

else

lower \*= other.lower;

}

if (upper != 0 && upper != CharCountFlag)

{

if (other.upper == CharCountFlag)

upper = CharCountFlag;

else

{

CharCount maxOther = MaxCharCount / upper;

if (other.upper > maxOther)

// Clip to 'unbounded'

upper = CharCountFlag;

else

upper \*= other.upper;

}

}

}

inline bool CouldMatchEmpty() const

{

return lower == 0;

}

inline bool IsUnbounded() const

{

return upper == CharCountFlag;

}

inline bool IsFixed() const

{

return lower == upper;

}

inline bool IsExact(CharCount n) const

{

return lower == n && upper == n;

}

inline bool IsGreaterThan(const CountDomain& other) const

{

return other.upper != CharCountFlag && lower > other.upper;

}

inline bool IsLessThan(const CountDomain& other) const

{

return upper != CharCountFlag && upper < other.lower;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w) const;

#endif

};

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

// Mix-in types

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

// Contains information about how much to back up after syncing to a literal (for the SyncTo... instructions)

struct BackupMixin

{

const CountDomain backup; // range of characters to backup, if upper is CharCountFlag then backup to existing matchStart

inline BackupMixin(const CountDomain& backup) : backup(backup) {}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w, const wchar\_t\* litbuf) const;

#endif

};

struct CharMixin

{

wchar\_t c;

inline CharMixin(wchar\_t c) : c(c) {}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w, const wchar\_t\* litbuf) const;

#endif

};

struct Char2Mixin

{

wchar\_t cs[2];

inline Char2Mixin(wchar\_t c0, wchar\_t c1) { cs[0] = c0; cs[1] = c1; }

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w, const wchar\_t\* litbuf) const;

#endif

};

struct Char3Mixin

{

wchar\_t cs[3];

inline Char3Mixin(wchar\_t c0, wchar\_t c1, wchar\_t c2) { cs[0] = c0; cs[1] = c1; cs[2] = c2; }

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w, const wchar\_t\* litbuf) const;

#endif

};

struct Char4Mixin

{

wchar\_t cs[4];

inline Char4Mixin(wchar\_t c0, wchar\_t c1, wchar\_t c2, wchar\_t c3) { cs[0] = c0; cs[1] = c1; cs[2] = c2; cs[3] = c3; }

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w, const wchar\_t\* litbuf) const;

#endif

};

struct LiteralMixin

{

CharCount offset; // into program's literal buffer

CharCount length; // in wchar\_t's

inline LiteralMixin(CharCount offset, CharCount length) : offset(offset), length(length) {}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w, const wchar\_t\* litbuf, bool isEquivClass) const;

#endif

};

template<bool IsNegation>

struct SetMixin

{

RuntimeCharSet<wchar\_t> set; // contents always lives in run-time allocator

// set must always be cloned from source

void FreeBody(ArenaAllocator\* rtAllocator);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w, const wchar\_t\* litbuf) const;

#endif

};

struct Char2LiteralScannerMixin : Char2Mixin

{

// scanner must be setup

Char2LiteralScannerMixin(CharCount offset, CharCount length) : Char2Mixin(0, 0) { Assert(length == 2); }

void Setup(wchar\_t c0, wchar\_t c1) { cs[0] = c0; cs[1] = c1; }

CharCount GetLiteralLength() const { return 2; }

bool Match(Matcher& matcher, const wchar\_t\* const input, const CharCount inputLength, CharCount& inputOffset) const;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w, const wchar\_t\* litbuf) const;

#endif

};

template <typename ScannerT>

struct ScannerMixinT : LiteralMixin

{

ScannerT scanner;

// scanner must be setup

ScannerMixinT(CharCount offset, CharCount length) : LiteralMixin(offset, length) {}

CharCount GetLiteralLength() const { return length; }

bool Match(Matcher& matcher, const wchar\_t\* const input, const CharCount inputLength, CharCount& inputOffset) const;

void FreeBody(ArenaAllocator\* rtAllocator);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w, const wchar\_t\* litbuf, bool isEquivClass = false) const;

#endif

};

typedef ScannerMixinT<TextbookBoyerMoore<wchar\_t>> ScannerMixin;

typedef ScannerMixinT<TextbookBoyerMooreWithLinearMap<wchar\_t>> ScannerMixin\_WithLinearCharMap;

template <uint lastPatCharEquivCLassSize>

struct EquivScannerMixinT : ScannerMixin

{

// scanner must be setup

EquivScannerMixinT(CharCount offset, CharCount length) : ScannerMixin(offset, length) {}

bool Match(Matcher& matcher, const wchar\_t\* const input, const CharCount inputLength, CharCount& inputOffset) const;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w, const wchar\_t\* litbuf) const;

#endif

};

typedef EquivScannerMixinT<CaseInsensitive::EquivClassSize> EquivScannerMixin;

typedef EquivScannerMixinT<1> EquivTrivialLastPatCharScannerMixin;

struct ScannerInfo : ScannerMixin

{

bool isEquivClass;

// scanner must be setup

inline ScannerInfo(CharCount offset, CharCount length, bool isEquivClass) : ScannerMixin(offset, length), isEquivClass(isEquivClass) {}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w, const wchar\_t\* litbuf) const;

#endif

};

struct ScannersMixin

{

static const int MaxNumSyncLiterals = 4;

int numLiterals;

ScannerInfo\*\* infos;

// scanner mixins must be added

inline ScannersMixin(Recycler \*const recycler, Program \*const program)

: numLiterals(0), infos(program->CreateScannerArrayForSyncToLiterals(recycler))

{

}

// Only used at compile time

ScannerInfo\* Add(Recycler \*recycler, Program \*program, CharCount offset, CharCount length, bool isEquivClass);

void FreeBody(ArenaAllocator\* rtAllocator);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w, const wchar\_t\* litbuf) const;

#endif

};

struct HardFailMixin

{

bool canHardFail;

inline HardFailMixin(bool canHardFail) : canHardFail(canHardFail) {}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w, const wchar\_t\* litbuf) const;

#endif

};

struct GroupMixin

{

const int groupId;

inline GroupMixin(int groupId) : groupId(groupId) {}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w, const wchar\_t\* litbuf) const;

#endif

};

struct ChompBoundedMixin

{

const CountDomain repeats; // if upper is CharCountFlag, consume as many characters as possible

inline ChompBoundedMixin(const CountDomain& repeats) : repeats(repeats) {}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w, const wchar\_t\* litbuf) const;

#endif

};

struct JumpMixin

{

Label targetLabel;

// targetLabel must always be fixed up

inline JumpMixin()

{

#if DBG

targetLabel = (Label)-1;

#endif

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w, const wchar\_t\* litbuf) const;

#endif

};

struct BodyGroupsMixin

{

int minBodyGroupId;

int maxBodyGroupId;

inline BodyGroupsMixin(int minBodyGroupId, int maxBodyGroupId) : minBodyGroupId(minBodyGroupId), maxBodyGroupId(maxBodyGroupId) {}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w, const wchar\_t\* litbuf) const;

#endif

};

struct BeginLoopMixin

{

int loopId;

const CountDomain repeats;

bool hasOuterLoops;

bool hasInnerNondet;

Label exitLabel;

// exitLabel must always be fixed up

inline BeginLoopMixin(int loopId, const CountDomain& repeats, bool hasOuterLoops, bool hasInnerNondet)

: loopId(loopId), repeats(repeats), hasOuterLoops(hasOuterLoops), hasInnerNondet(hasInnerNondet)

{

#if DBG

exitLabel = (Label)-1;

#endif

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w, const wchar\_t\* litbuf) const;

#endif

};

struct RepeatLoopMixin

{

Label beginLabel; // label of the BeginLoopX instruction

inline RepeatLoopMixin(Label beginLabel) : beginLabel(beginLabel) {}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w, const wchar\_t\* litbuf) const;

#endif

};

struct TryMixin

{

Label failLabel;

// failLabel must always be fixed up

inline TryMixin()

{

#if DBG

failLabel = (Label)-1;

#endif

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w, const wchar\_t\* litbuf) const;

#endif

};

struct FixedLengthMixin

{

CharCount length;

inline FixedLengthMixin(CharCount length) : length(length) {}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w, const wchar\_t\* litbuf) const;

#endif

};

struct NoNeedToSaveMixin

{

bool noNeedToSave;

inline NoNeedToSaveMixin(bool noNeedToSave) : noNeedToSave(noNeedToSave) {}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w, const wchar\_t\* litbuf) const;

#endif

};

struct SwitchCase

{

wchar\_t c;

Label targetLabel;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w) const;

#endif

};

template <int n>

struct SwitchMixin

{

static const int MaxCases = n;

int numCases;

// numCases cases, in increasing character order

SwitchCase cases[MaxCases];

// Cases must always be added

inline SwitchMixin() : numCases(0)

{

#if DBG

for (int i = 0; i < MaxCases; i++)

{

cases[i].c = (wchar\_t)-1;

cases[i].targetLabel = (Label)-1;

}

#endif

}

// Only used at compile time

void AddCase(wchar\_t c, Label targetLabel);

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w, const wchar\_t\* litbuf) const;

#endif

};

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

// Instructions

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

struct Inst : protected Chars<wchar\_t>

{

enum InstTag : uint32

{

#define M(TagName) TagName,

#define MTemplate(TagName, ...) M(TagName)

#include "RegexOpcodes.h"

#undef M

#undef MTemplate

};

InstTag tag;

inline Inst(InstTag tag) : tag(tag) {}

void FreeBody(ArenaAllocator\* rtAllocator) {}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

virtual int Print(DebugWriter\*w, Label label, const Char\* litbuf) const = 0;

#endif

};

#define INST\_BODY\_FREE(T) \

void FreeBody(ArenaAllocator\* rtAllocator) \

{ \

T::FreeBody(rtAllocator); \

Inst::FreeBody(rtAllocator); \

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

#define INST\_BODY\_PRINT virtual int Print(DebugWriter\*w, Label label, const Char\* litbuf) const override;

#else

#define INST\_BODY\_PRINT

#endif

#define REGEX\_INST\_EXEC\_PARAMETERS Matcher& matcher, const Char\* const input, const CharCount inputLength, CharCount &matchStart, CharCount& inputOffset, CharCount &nextSyncInputOffset, const uint8\*& instPointer, ContStack &contStack, AssertionStack &assertionStack, uint &qcTicks

#define INST\_BODY bool Exec(REGEX\_INST\_EXEC\_PARAMETERS) const; \

INST\_BODY\_PRINT

//

// Control flow

//

struct FailInst : Inst

{

inline FailInst() : Inst(Fail) {}

INST\_BODY

};

struct SuccInst : Inst

{

inline SuccInst() : Inst(Succ) {}

INST\_BODY

};

struct JumpInst : Inst, JumpMixin

{

// targetLabel must always be fixed up

inline JumpInst() : Inst(Jump), JumpMixin() {}

INST\_BODY

};

struct JumpIfNotCharInst : Inst, CharMixin, JumpMixin

{

// targetLabel must always be fixed up

inline JumpIfNotCharInst(Char c) : Inst(JumpIfNotChar), CharMixin(c), JumpMixin() {}

INST\_BODY

};

struct MatchCharOrJumpInst : Inst, CharMixin, JumpMixin

{

// targetLabel must always be fixed up

inline MatchCharOrJumpInst(Char c) : Inst(MatchCharOrJump), CharMixin(c), JumpMixin() {}

INST\_BODY

};

struct JumpIfNotSetInst : Inst, SetMixin<false>, JumpMixin

{

// set must always be cloned from source

// targetLabel must always be fixed up

inline JumpIfNotSetInst() : Inst(JumpIfNotSet), JumpMixin() {}

INST\_BODY

INST\_BODY\_FREE(SetMixin)

};

struct MatchSetOrJumpInst : Inst, SetMixin<false>, JumpMixin

{

// set must always be cloned from source

// targetLabel must always be fixed up

inline MatchSetOrJumpInst() : Inst(MatchSetOrJump), JumpMixin() {}

INST\_BODY

INST\_BODY\_FREE(SetMixin)

};

struct Switch10Inst : Inst, SwitchMixin<10>

{

// Cases must always be added

inline Switch10Inst() : Inst(Switch10), SwitchMixin() {}

INST\_BODY

};

struct Switch20Inst : Inst, SwitchMixin<20>

{

// Cases must always be added

inline Switch20Inst() : Inst(Switch20), SwitchMixin() {}

INST\_BODY

};

struct SwitchAndConsume10Inst : Inst, SwitchMixin<10>

{

// Cases must always be added

inline SwitchAndConsume10Inst() : Inst(SwitchAndConsume10), SwitchMixin() {}

INST\_BODY

};

struct SwitchAndConsume20Inst : Inst, SwitchMixin<20>

{

// Cases must always be added

inline SwitchAndConsume20Inst() : Inst(SwitchAndConsume20), SwitchMixin() {}

INST\_BODY

};

//

// Built-in assertions

//

struct BOITestInst : Inst, HardFailMixin

{

inline BOITestInst(bool canHardFail) : Inst(BOITest), HardFailMixin(canHardFail) {}

INST\_BODY

};

struct EOITestInst : Inst, HardFailMixin

{

inline EOITestInst(bool canHardFail) : Inst(EOITest), HardFailMixin(canHardFail) {}

INST\_BODY

};

struct BOLTestInst : Inst

{

inline BOLTestInst() : Inst(BOLTest) {}

INST\_BODY

};

struct EOLTestInst : Inst

{

inline EOLTestInst() : Inst(EOLTest) {}

INST\_BODY

};

struct WordBoundaryTestInst : Inst

{

bool isNegation;

inline WordBoundaryTestInst(bool isNegation) : Inst(WordBoundaryTest), isNegation(isNegation) {}

INST\_BODY

};

//

// Matching

//

struct MatchCharInst : Inst, CharMixin

{

inline MatchCharInst(Char c) : Inst(MatchChar), CharMixin(c) {}

INST\_BODY

};

struct MatchChar2Inst : Inst, Char2Mixin

{

inline MatchChar2Inst(Char c0, Char c1) : Inst(MatchChar2), Char2Mixin(c0, c1) {}

INST\_BODY

};

struct MatchChar3Inst : Inst, Char3Mixin

{

inline MatchChar3Inst(Char c0, Char c1, Char c2) : Inst(MatchChar3), Char3Mixin(c0, c1, c2) {}

INST\_BODY

};

struct MatchChar4Inst : Inst, Char4Mixin

{

inline MatchChar4Inst(Char c0, Char c1, Char c2, Char c3) : Inst(MatchChar4), Char4Mixin(c0, c1, c2, c3) {}

INST\_BODY

};

template<bool IsNegation>

struct MatchSetInst : Inst, SetMixin<IsNegation>

{

// set must always be cloned from source

inline MatchSetInst() : Inst(IsNegation ? MatchNegatedSet : MatchSet) {}

INST\_BODY

INST\_BODY\_FREE(SetMixin)

};

struct MatchLiteralInst : Inst, LiteralMixin

{

inline MatchLiteralInst(CharCount offset, CharCount length) : Inst(MatchLiteral), LiteralMixin(offset, length) {}

INST\_BODY

};

struct MatchLiteralEquivInst : Inst, LiteralMixin

{

inline MatchLiteralEquivInst(CharCount offset, CharCount length) : Inst(MatchLiteralEquiv), LiteralMixin(offset, length) {}

INST\_BODY

};

struct MatchTrieInst : Inst

{

RuntimeCharTrie trie;

// Trie must always be cloned

inline MatchTrieInst() : Inst(MatchTrie) {}

void FreeBody(ArenaAllocator\* rtAllocator);

INST\_BODY

};

struct OptMatchCharInst : Inst, CharMixin

{

inline OptMatchCharInst(Char c) : Inst(OptMatchChar), CharMixin(c) {}

INST\_BODY

};

struct OptMatchSetInst : Inst, SetMixin<false>

{

// set must always be cloned from source

inline OptMatchSetInst() : Inst(OptMatchSet) {}

INST\_BODY

INST\_BODY\_FREE(SetMixin)

};

//

// Synchronization:

// SyncTo(Char|Char2Set|Set|Char2Literal|Literal|LiteralEquiv|Literals)And(Consume|Continue|Backup)

//

struct SyncToCharAndContinueInst : Inst, CharMixin

{

inline SyncToCharAndContinueInst(Char c) : Inst(SyncToCharAndContinue), CharMixin(c) {}

INST\_BODY

};

struct SyncToChar2SetAndContinueInst : Inst, Char2Mixin

{

inline SyncToChar2SetAndContinueInst(Char c0, Char c1) : Inst(SyncToChar2SetAndContinue), Char2Mixin(c0, c1) {}

INST\_BODY

};

template<bool IsNegation>

struct SyncToSetAndContinueInst : Inst, SetMixin<IsNegation>

{

// set must always be cloned from source

inline SyncToSetAndContinueInst() : Inst(IsNegation ? SyncToNegatedSetAndContinue : SyncToSetAndContinue) {}

INST\_BODY

INST\_BODY\_FREE(SetMixin)

};

template <typename ScannerT>

struct SyncToLiteralAndContinueInstT : Inst, ScannerT

{

SyncToLiteralAndContinueInstT(InstTag tag, CharCount offset, CharCount length) : Inst(tag), ScannerT(offset, length) {}

INST\_BODY

};

// Specialized version of the SyncToLiteralAndContinueInst for a length 2 literal

struct SyncToChar2LiteralAndContinueInst : SyncToLiteralAndContinueInstT<Char2LiteralScannerMixin>

{

SyncToChar2LiteralAndContinueInst(Char c0, Char c1) :

SyncToLiteralAndContinueInstT(SyncToChar2LiteralAndContinue, 0, 2) { Char2LiteralScannerMixin::Setup(c0, c1); }

};

struct SyncToLiteralAndContinueInst : SyncToLiteralAndContinueInstT<ScannerMixin>

{

// scanner must be setup

SyncToLiteralAndContinueInst(CharCount offset, CharCount length) :

SyncToLiteralAndContinueInstT(SyncToLiteralAndContinue, offset, length) {}

INST\_BODY\_FREE(ScannerMixin)

};

struct SyncToLinearLiteralAndContinueInst : SyncToLiteralAndContinueInstT<ScannerMixin\_WithLinearCharMap>

{

// scanner must be setup

SyncToLinearLiteralAndContinueInst(CharCount offset, CharCount length) :

SyncToLiteralAndContinueInstT(SyncToLinearLiteralAndContinue, offset, length) {}

INST\_BODY\_FREE(ScannerMixin\_WithLinearCharMap)

};

struct SyncToLiteralEquivAndContinueInst : SyncToLiteralAndContinueInstT<EquivScannerMixin>

{

// scanner must be setup

SyncToLiteralEquivAndContinueInst(CharCount offset, CharCount length) :

SyncToLiteralAndContinueInstT(SyncToLiteralEquivAndContinue, offset, length) {}

INST\_BODY\_FREE(EquivScannerMixin)

};

struct SyncToLiteralEquivTrivialLastPatCharAndContinueInst : SyncToLiteralAndContinueInstT<EquivTrivialLastPatCharScannerMixin>

{

// scanner must be setup

SyncToLiteralEquivTrivialLastPatCharAndContinueInst(CharCount offset, CharCount length) :

SyncToLiteralAndContinueInstT(SyncToLiteralEquivTrivialLastPatCharAndContinue, offset, length) {}

INST\_BODY\_FREE(EquivTrivialLastPatCharScannerMixin)

};

struct SyncToCharAndConsumeInst : Inst, CharMixin

{

inline SyncToCharAndConsumeInst(Char c) : Inst(SyncToCharAndConsume), CharMixin(c) {}

INST\_BODY

};

struct SyncToChar2SetAndConsumeInst : Inst, Char2Mixin

{

inline SyncToChar2SetAndConsumeInst(Char c0, Char c1) : Inst(SyncToChar2SetAndConsume), Char2Mixin(c0, c1) {}

INST\_BODY

};

template<bool IsNegation>

struct SyncToSetAndConsumeInst : Inst, SetMixin<IsNegation>

{

// set must always be cloned from source

inline SyncToSetAndConsumeInst() : Inst(IsNegation ? SyncToNegatedSetAndConsume : SyncToSetAndConsume) {}

INST\_BODY

INST\_BODY\_FREE(SetMixin)

};

template <typename ScannerT>

struct SyncToLiteralAndConsumeInstT : Inst, ScannerT

{

SyncToLiteralAndConsumeInstT(InstTag tag, CharCount offset, CharCount length) : Inst(tag), ScannerT(offset, length) {}

INST\_BODY

};

// Specialized version of the SyncToLiteralAndConsumeInst for a length 2 literal

struct SyncToChar2LiteralAndConsumeInst : SyncToLiteralAndConsumeInstT<Char2LiteralScannerMixin>

{

SyncToChar2LiteralAndConsumeInst(Char c0, Char c1) :

SyncToLiteralAndConsumeInstT(SyncToChar2LiteralAndConsume, 0, 2) { Char2LiteralScannerMixin::Setup(c0, c1); }

};

struct SyncToLiteralAndConsumeInst : SyncToLiteralAndConsumeInstT<ScannerMixin>

{

// scanner must be setup

SyncToLiteralAndConsumeInst(CharCount offset, CharCount length) :

SyncToLiteralAndConsumeInstT(SyncToLiteralAndConsume, offset, length) {}

INST\_BODY\_FREE(ScannerMixin)

};

struct SyncToLinearLiteralAndConsumeInst : SyncToLiteralAndConsumeInstT<ScannerMixin\_WithLinearCharMap>

{

// scanner must be setup

SyncToLinearLiteralAndConsumeInst(CharCount offset, CharCount length) :

SyncToLiteralAndConsumeInstT(SyncToLinearLiteralAndConsume, offset, length) {}

INST\_BODY\_FREE(ScannerMixin\_WithLinearCharMap)

};

struct SyncToLiteralEquivAndConsumeInst : SyncToLiteralAndConsumeInstT<EquivScannerMixin>

{

// scanner must be setup

SyncToLiteralEquivAndConsumeInst(CharCount offset, CharCount length) :

SyncToLiteralAndConsumeInstT(SyncToLiteralEquivAndConsume,offset, length) {}

INST\_BODY\_FREE(EquivScannerMixin)

};

struct SyncToLiteralEquivTrivialLastPatCharAndConsumeInst : SyncToLiteralAndConsumeInstT<EquivTrivialLastPatCharScannerMixin>

{

// scanner must be setup

SyncToLiteralEquivTrivialLastPatCharAndConsumeInst(CharCount offset, CharCount length) :

SyncToLiteralAndConsumeInstT(SyncToLiteralEquivTrivialLastPatCharAndConsume, offset, length) {}

INST\_BODY\_FREE(EquivTrivialLastPatCharScannerMixin)

};

struct SyncToCharAndBackupInst : Inst, CharMixin, BackupMixin

{

inline SyncToCharAndBackupInst(Char c, const CountDomain& backup) : Inst(SyncToCharAndBackup), CharMixin(c), BackupMixin(backup) {}

INST\_BODY

};

template<bool IsNegation>

struct SyncToSetAndBackupInst : Inst, SetMixin<IsNegation>, BackupMixin

{

// set must always be cloned from source

inline SyncToSetAndBackupInst(const CountDomain& backup) : Inst(IsNegation ? SyncToNegatedSetAndBackup : SyncToSetAndBackup), BackupMixin(backup) {}

INST\_BODY

INST\_BODY\_FREE(SetMixin)

};

template <typename ScannerT>

struct SyncToLiteralAndBackupInstT : Inst, ScannerT, BackupMixin

{

SyncToLiteralAndBackupInstT(InstTag tag, CharCount offset, CharCount length, const CountDomain& backup) : Inst(tag), ScannerT(offset, length), BackupMixin(backup) {}

INST\_BODY

};

// Specialized version of the SyncToLiteralAndConsumeInst for a length 2 literal

struct SyncToChar2LiteralAndBackupInst : SyncToLiteralAndBackupInstT<Char2LiteralScannerMixin>

{

SyncToChar2LiteralAndBackupInst(Char c0, Char c1, const CountDomain& backup) :

SyncToLiteralAndBackupInstT(SyncToChar2LiteralAndBackup, 0, 2, backup) { Char2LiteralScannerMixin::Setup(c0, c1); }

};

struct SyncToLiteralAndBackupInst : SyncToLiteralAndBackupInstT<ScannerMixin>

{

// scanner must be setup

SyncToLiteralAndBackupInst(CharCount offset, CharCount length, const CountDomain& backup) :

SyncToLiteralAndBackupInstT(SyncToLiteralAndBackup, offset, length, backup) {}

INST\_BODY\_FREE(ScannerMixin)

};

struct SyncToLinearLiteralAndBackupInst : SyncToLiteralAndBackupInstT<ScannerMixin\_WithLinearCharMap>

{

// scanner must be setup

SyncToLinearLiteralAndBackupInst(CharCount offset, CharCount length, const CountDomain& backup) :

SyncToLiteralAndBackupInstT(SyncToLinearLiteralAndBackup, offset, length, backup) {}

INST\_BODY\_FREE(ScannerMixin\_WithLinearCharMap)

};

struct SyncToLiteralEquivAndBackupInst : SyncToLiteralAndBackupInstT<EquivScannerMixin>

{

// scanner must be setup

SyncToLiteralEquivAndBackupInst(CharCount offset, CharCount length, const CountDomain& backup) :

SyncToLiteralAndBackupInstT(SyncToLiteralEquivAndBackup, offset, length, backup) {}

INST\_BODY\_FREE(EquivScannerMixin)

};

struct SyncToLiteralEquivTrivialLastPatCharAndBackupInst : SyncToLiteralAndBackupInstT<EquivTrivialLastPatCharScannerMixin>

{

// scanner must be setup

SyncToLiteralEquivTrivialLastPatCharAndBackupInst(CharCount offset, CharCount length, const CountDomain& backup) :

SyncToLiteralAndBackupInstT(SyncToLiteralEquivTrivialLastPatCharAndBackup, offset, length, backup) {}

INST\_BODY\_FREE(EquivTrivialLastPatCharScannerMixin)

};

struct SyncToLiteralsAndBackupInst : Inst, ScannersMixin, BackupMixin

{

// scanner mixins must be setup

inline SyncToLiteralsAndBackupInst(Recycler \*recycler, Program \*program, const CountDomain& backup)

: Inst(SyncToLiteralsAndBackup), ScannersMixin(recycler, program), BackupMixin(backup)

{

}

INST\_BODY

INST\_BODY\_FREE(ScannersMixin)

};

//

// Groups

//

struct MatchGroupInst : Inst, GroupMixin

{

inline MatchGroupInst(int groupId) : Inst(MatchGroup), GroupMixin(groupId) {}

INST\_BODY

};

struct BeginDefineGroupInst : Inst, GroupMixin

{

inline BeginDefineGroupInst(int groupId) : Inst(BeginDefineGroup), GroupMixin(groupId) {}

INST\_BODY

};

struct EndDefineGroupInst : Inst, GroupMixin, NoNeedToSaveMixin

{

inline EndDefineGroupInst(int groupId, bool noNeedToSave)

: Inst(EndDefineGroup), GroupMixin(groupId), NoNeedToSaveMixin(noNeedToSave)

{

}

INST\_BODY

};

struct DefineGroupFixedInst : Inst, GroupMixin, FixedLengthMixin, NoNeedToSaveMixin

{

inline DefineGroupFixedInst(int groupId, CharCount length, bool noNeedToSave) : Inst(DefineGroupFixed), GroupMixin(groupId), FixedLengthMixin(length), NoNeedToSaveMixin(noNeedToSave) {}

INST\_BODY

};

//

// Loops

//

struct BeginLoopInst : Inst, BeginLoopMixin, BodyGroupsMixin

{

bool isGreedy;

// exitLabel must always be fixed up

inline BeginLoopInst(int loopId, const CountDomain& repeats, bool hasOuterLoops, bool hasInnerNondet, int minBodyGroupId, int maxBodyGroupId, bool isGreedy)

: Inst(BeginLoop), BeginLoopMixin(loopId, repeats, hasOuterLoops, hasInnerNondet), BodyGroupsMixin(minBodyGroupId, maxBodyGroupId), isGreedy(isGreedy)

{}

INST\_BODY

};

struct RepeatLoopInst : Inst, RepeatLoopMixin

{

inline RepeatLoopInst(Label beginLabel) : Inst(RepeatLoop), RepeatLoopMixin(beginLabel) {}

INST\_BODY

};

struct BeginLoopIfCharInst : Inst, CharMixin, BeginLoopMixin, BodyGroupsMixin

{

// exitLabel must always be fixed up

inline BeginLoopIfCharInst(Char c, int loopId, const CountDomain& repeats, bool hasOuterLoops, bool hasInnerNondet, int minBodyGroupId, int maxBodyGroupId)

: Inst(BeginLoopIfChar), CharMixin(c), BeginLoopMixin(loopId, repeats, hasOuterLoops, hasInnerNondet), BodyGroupsMixin(minBodyGroupId, maxBodyGroupId) {}

INST\_BODY

};

struct BeginLoopIfSetInst : Inst, SetMixin<false>, BeginLoopMixin, BodyGroupsMixin

{

// set must always be cloned from source

// exitLabel must always be fixed up

inline BeginLoopIfSetInst(int loopId, const CountDomain& repeats, bool hasOuterLoops, bool hasInnerNondet, int minBodyGroupId, int maxBodyGroupId)

: Inst(BeginLoopIfSet), BeginLoopMixin(loopId, repeats, hasOuterLoops, hasInnerNondet), BodyGroupsMixin(minBodyGroupId, maxBodyGroupId) {}

INST\_BODY

INST\_BODY\_FREE(SetMixin)

};

struct RepeatLoopIfCharInst : Inst, RepeatLoopMixin

{

inline RepeatLoopIfCharInst(Label beginLabel) : Inst(RepeatLoopIfChar), RepeatLoopMixin(beginLabel) {}

INST\_BODY

};

struct RepeatLoopIfSetInst : Inst, RepeatLoopMixin

{

inline RepeatLoopIfSetInst(Label beginLabel) : Inst(RepeatLoopIfSet), RepeatLoopMixin(beginLabel) {}

INST\_BODY

};

// Loop is greedy, fixed width, deterministic body, no inner groups

struct BeginLoopFixedInst : Inst, BeginLoopMixin, FixedLengthMixin

{

// exitLabel must always be fixed up

inline BeginLoopFixedInst(int loopId, const CountDomain& repeats, bool hasOuterLoops, CharCount length)

: Inst(BeginLoopFixed), BeginLoopMixin(loopId, repeats, hasOuterLoops, false), FixedLengthMixin(length) {}

INST\_BODY

};

struct RepeatLoopFixedInst : Inst, RepeatLoopMixin

{

inline RepeatLoopFixedInst(Label beginLabel) : Inst(RepeatLoopFixed), RepeatLoopMixin(beginLabel) {}

INST\_BODY

};

// Loop is greedy, contains a MatchSet only

struct LoopSetInst : Inst, SetMixin<false>

{

int loopId;

const CountDomain repeats;

bool hasOuterLoops;

// set must always be cloned from source

inline LoopSetInst(int loopId, const CountDomain& repeats, bool hasOuterLoops)

: Inst(LoopSet), loopId(loopId), repeats(repeats), hasOuterLoops(hasOuterLoops) {}

INST\_BODY

INST\_BODY\_FREE(SetMixin)

};

// Loop is greedy, fixed width, deterministic body, one outermost group

struct BeginLoopFixedGroupLastIterationInst : Inst, BeginLoopMixin, FixedLengthMixin, GroupMixin, NoNeedToSaveMixin

{

// exitLabel must always be fixed up

inline BeginLoopFixedGroupLastIterationInst(int loopId, const CountDomain& repeats, bool hasOuterLoops, CharCount length, int groupId, bool noNeedToSave)

: Inst(BeginLoopFixedGroupLastIteration), BeginLoopMixin(loopId, repeats, hasOuterLoops, false), FixedLengthMixin(length), GroupMixin(groupId), NoNeedToSaveMixin(noNeedToSave) {}

INST\_BODY

};

struct RepeatLoopFixedGroupLastIterationInst : Inst, RepeatLoopMixin

{

inline RepeatLoopFixedGroupLastIterationInst(Label beginLabel) : Inst(RepeatLoopFixedGroupLastIteration), RepeatLoopMixin(beginLabel) {}

INST\_BODY

};

// Loop is greedy, deterministic body, lower == 0, upper == inf, follow is irrefutable, no inner groups

struct BeginGreedyLoopNoBacktrackInst : Inst

{

int loopId;

Label exitLabel;

// exitLabel must always be fixed up

inline BeginGreedyLoopNoBacktrackInst(int loopId) : Inst(BeginGreedyLoopNoBacktrack), loopId(loopId) {}

INST\_BODY

};

struct RepeatGreedyLoopNoBacktrackInst : Inst, RepeatLoopMixin

{

inline RepeatGreedyLoopNoBacktrackInst(Label beginLabel) : Inst(RepeatGreedyLoopNoBacktrack), RepeatLoopMixin(beginLabel) {}

INST\_BODY

};

template<ChompMode Mode>

struct ChompCharInst : Inst, CharMixin

{

ChompCharInst(const Char c) : Inst(Mode == ChompMode::Star ? ChompCharStar : ChompCharPlus), CharMixin(c) {}

INST\_BODY

};

template<ChompMode Mode>

struct ChompSetInst : Inst, SetMixin<false>

{

// set must always be cloned from source

ChompSetInst() : Inst(Mode == ChompMode::Star ? ChompSetStar : ChompSetPlus) {}

INST\_BODY

INST\_BODY\_FREE(SetMixin)

};

template<ChompMode Mode>

struct ChompCharGroupInst : Inst, CharMixin, GroupMixin, NoNeedToSaveMixin

{

ChompCharGroupInst(const Char c, const int groupId, const bool noNeedToSave)

: Inst(Mode == ChompMode::Star ? ChompCharGroupStar : ChompCharGroupPlus),

CharMixin(c),

GroupMixin(groupId),

NoNeedToSaveMixin(noNeedToSave)

{

}

INST\_BODY

};

template<ChompMode Mode>

struct ChompSetGroupInst : Inst, SetMixin<false>, GroupMixin, NoNeedToSaveMixin

{

// set must always be cloned from source

ChompSetGroupInst(const int groupId, const bool noNeedToSave)

: Inst(Mode == ChompMode::Star ? ChompSetGroupStar : ChompSetGroupPlus),

GroupMixin(groupId),

NoNeedToSaveMixin(noNeedToSave)

{

}

INST\_BODY

INST\_BODY\_FREE(SetMixin)

};

struct ChompCharBoundedInst : Inst, CharMixin, ChompBoundedMixin

{

inline ChompCharBoundedInst(Char c, const CountDomain& repeats) : Inst(ChompCharBounded), CharMixin(c), ChompBoundedMixin(repeats) {}

INST\_BODY

};

struct ChompSetBoundedInst : Inst, SetMixin<false>, ChompBoundedMixin

{

// set must always be cloned from source

inline ChompSetBoundedInst(const CountDomain& repeats) : Inst(ChompSetBounded), ChompBoundedMixin(repeats) {}

INST\_BODY

INST\_BODY\_FREE(SetMixin)

};

struct ChompSetBoundedGroupLastCharInst : Inst, SetMixin<false>, ChompBoundedMixin, GroupMixin, NoNeedToSaveMixin

{

// set must always be cloned from source

inline ChompSetBoundedGroupLastCharInst(const CountDomain& repeats, int groupId, bool noNeedToSave) : Inst(ChompSetBoundedGroupLastChar), ChompBoundedMixin(repeats), GroupMixin(groupId), NoNeedToSaveMixin(noNeedToSave) {}

INST\_BODY

INST\_BODY\_FREE(SetMixin)

};

//

// Choicepoints

//

struct TryInst : Inst, TryMixin

{

// failLabel must always be fixed up

inline TryInst() : Inst(Try), TryMixin() {}

INST\_BODY

};

struct TryIfCharInst : Inst, CharMixin, TryMixin

{

// failLabel must always be fixed up

inline TryIfCharInst(Char c) : Inst(TryIfChar), CharMixin(c), TryMixin() {}

INST\_BODY

};

struct TryMatchCharInst : Inst, CharMixin, TryMixin

{

// failLabel must always be fixed up

inline TryMatchCharInst(Char c) : Inst(TryMatchChar), CharMixin(c), TryMixin() {}

INST\_BODY

};

struct TryIfSetInst : Inst, SetMixin<false>, TryMixin

{

// set is always same as matching BeginLoopIfSetInst set

// failLabel must always be fixed up

inline TryIfSetInst() : Inst(TryIfSet), TryMixin() {}

INST\_BODY

INST\_BODY\_FREE(SetMixin)

};

struct TryMatchSetInst : Inst, SetMixin<false>, TryMixin

{

// set is always same as matching BeginLoopIfSetInst set

// failLabel must always be fixed up

inline TryMatchSetInst() : Inst(TryMatchSet), TryMixin() {}

INST\_BODY

INST\_BODY\_FREE(SetMixin)

};

//

// User-defined assertions

//

struct BeginAssertionInst : Inst, BodyGroupsMixin

{

bool isNegation;

Label nextLabel;

// nextLabel must always be fixed up

inline BeginAssertionInst(bool isNegation, int minBodyGroupId, int maxBodyGroupId) : Inst(BeginAssertion), isNegation(isNegation), BodyGroupsMixin(minBodyGroupId, maxBodyGroupId)

{

#if DBG

nextLabel = (Label)-1;

#endif

}

INST\_BODY

};

struct EndAssertionInst : Inst

{

inline EndAssertionInst() : Inst(EndAssertion) {}

INST\_BODY

};

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

// Matcher state

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

struct LoopInfo : protected Chars<wchar\_t>

{

CharCount number; // current iteration number

CharCount startInputOffset; // input offset where the iteration started

inline void Reset()

{

#if DBG

// So debug prints will look nice

number = 0;

startInputOffset = 0;

#endif

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w) const;

#endif

};

struct GroupInfo : protected Chars<wchar\_t>

{

CharCount offset;

CharCountOrFlag length; // CharCountFlag => group is undefined

inline GroupInfo() : offset(0), length(CharCountFlag) {}

inline GroupInfo(CharCount offset, CharCountOrFlag length) : offset(offset), length(length) {}

//This constructor will only be called by a cross-site marshalling and thus we shouldn't clear offset and length

GroupInfo(VirtualTableInfoCtorEnum) { }

inline bool IsUndefined() const { return length == CharCountFlag; }

inline CharCount EndOffset() const { Assert(length != CharCountFlag); return offset + (CharCount)length; }

inline void Reset()

{

// The start offset must not be changed when backtracking into the group

length = CharCountFlag;

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w, const Char\* const input) const;

#endif

};

struct AssertionInfo : private Chars<wchar\_t>

{

const Label beginLabel; // label of BeginAssertion instruction

CharCount startInputOffset; // input offset when begun assertion (so can rewind)

size\_t contStackPosition; // top of continuation stack when begun assertion (so can cut)

inline AssertionInfo(Label beginLabel, CharCount startInputOffset, size\_t contStackPosition)

: beginLabel(beginLabel), startInputOffset(startInputOffset), contStackPosition(contStackPosition) {}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w) const;

#endif

};

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

// Continuations

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

struct Cont : protected Chars<wchar\_t>

{

enum ContTag : uint32

{

#define M(O) O,

#include "RegexContcodes.h"

#undef M

};

ContTag tag;

inline Cont(ContTag tag) : tag(tag) {}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

virtual int Print(DebugWriter\*w, const Char\* const input) const = 0;

#endif

};

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

#define CONT\_PRINT int Print(DebugWriter\*w, const Char\* const input) const override;

#else

#define CONT\_PRINT

#endif

#define REGEX\_CONT\_EXEC\_PARAMETERS Matcher& matcher, const Char\* const input, CharCount& inputOffset, const uint8\*& instPointer, ContStack &contStack, AssertionStack &assertionStack, uint &qcTicks

#define CONT\_BODY bool Exec(REGEX\_CONT\_EXEC\_PARAMETERS); \

CONT\_PRINT

struct ResumeCont : Cont

{

CharCount origInputOffset;

Label origInstLabel;

inline ResumeCont(CharCount origInputOffset, Label origInstLabel) : Cont(Resume), origInputOffset(origInputOffset), origInstLabel(origInstLabel) {}

CONT\_BODY

};

struct RestoreLoopCont : Cont

{

int loopId;

LoopInfo origLoopInfo;

inline RestoreLoopCont(int loopId, LoopInfo& origLoopInfo) : Cont(RestoreLoop), loopId(loopId), origLoopInfo(origLoopInfo) {}

CONT\_BODY

};

struct RestoreGroupCont : Cont

{

int groupId;

GroupInfo origGroupInfo;

RestoreGroupCont(int groupId, const GroupInfo &origGroupInfo)

: Cont(RestoreGroup), groupId(groupId), origGroupInfo(origGroupInfo)

{

}

CONT\_BODY

};

struct ResetGroupCont : Cont

{

const int groupId;

ResetGroupCont(const int groupId) : Cont(ResetGroup), groupId(groupId) {}

CONT\_BODY

};

struct ResetGroupRangeCont : Cont

{

const int fromGroupId;

const int toGroupId;

ResetGroupRangeCont(const int fromGroupId, const int toGroupId)

: Cont(ResetGroupRange), fromGroupId(fromGroupId), toGroupId(toGroupId)

{

Assert(fromGroupId >= 0);

Assert(toGroupId >= 0);

Assert(fromGroupId < toGroupId);

}

CONT\_BODY

};

struct RepeatLoopCont : Cont

{

Label beginLabel; // label of BeginLoop instruction

CharCount origInputOffset; // where to go back to

inline RepeatLoopCont(Label beginLabel, CharCount origInputOffset) : Cont(RepeatLoop), beginLabel(beginLabel), origInputOffset(origInputOffset) {}

CONT\_BODY

};

struct PopAssertionCont : Cont

{

inline PopAssertionCont() : Cont(PopAssertion) {}

CONT\_BODY

};

struct RewindLoopFixedCont : Cont

{

Label beginLabel; // label of BeginLoopFixed instruction

bool tryingBody; // true if attempting an additional iteration of loop body, otherwise attempting loop follow

inline RewindLoopFixedCont(Label beginLabel, bool tryingBody) : Cont(RewindLoopFixed), beginLabel(beginLabel), tryingBody(tryingBody) {}

CONT\_BODY

};

struct RewindLoopSetCont : Cont

{

Label beginLabel; // label of LoopSet instruction

inline RewindLoopSetCont(Label beginLabel) : Cont(RewindLoopSet), beginLabel(beginLabel) {}

CONT\_BODY

};

struct RewindLoopFixedGroupLastIterationCont : Cont

{

Label beginLabel; // label of BeginLoopFixedGroupLastIteration instruction

bool tryingBody; // true if attempting an additional iteration of loop body, otherwise attempting loop follow

inline RewindLoopFixedGroupLastIterationCont(Label beginLabel, bool tryingBody) : Cont(RewindLoopFixedGroupLastIteration), beginLabel(beginLabel), tryingBody(tryingBody) {}

CONT\_BODY

};

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

// Matcher

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

class ContStack : public ContinuousPageStackOfVariableElements<Cont>, private Chars<wchar\_t>

{

public:

\_\_inline ContStack(PageAllocator \*const pageAllocator, void (\*const outOfMemoryFunc)())

: ContinuousPageStackOfVariableElements(pageAllocator, outOfMemoryFunc)

{

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w, const Char\* const input) const;

#endif

};

class AssertionStack : public ContinuousPageStackOfFixedElements<AssertionInfo>

{

public:

\_\_inline AssertionStack(PageAllocator \*const pageAllocator, void (\*const outOfMemoryFunc)())

: ContinuousPageStackOfFixedElements(pageAllocator, outOfMemoryFunc)

{

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w, const Matcher\* const matcher) const;

#endif

};

struct RegexStacks

{

RegexStacks(PageAllocator \* pageAllocator) :

contStack(pageAllocator, Js::Throw::OutOfMemory),

assertionStack(pageAllocator, Js::Throw::OutOfMemory) {};

ContStack contStack;

AssertionStack assertionStack;

};

enum HardFailMode

{

BacktrackAndLater,

BacktrackOnly,

LaterOnly,

ImmediateFail

};

class Matcher : private Chars<wchar\_t>

{

#define M(TagName) friend struct TagName##Inst;

#define MTemplate(TagName, TemplateDeclaration, GenericClassName, ...) TemplateDeclaration friend struct GenericClassName;

#include "RegexOpcodes.h"

#undef M

#undef MTemplate

#define M(O) friend O##Cont;

#include "RegexContcodes.h"

#undef M

template <typename ScannerT>

friend struct SyncToLiteralAndConsumeInstT;

template <typename ScannerT>

friend struct SyncToLiteralAndContinueInstT;

template <typename ScannerT>

friend struct SyncToLiteralAndBackupInstT;

template <typename ScannerT>

friend struct ScannerMixinT;

friend struct Char2LiteralScannerMixin;

template <uint lastPatCharEquivClassSize>

friend struct EquivScannerMixinT;

friend GroupInfo;

friend LoopInfo;

public:

static const uint TicksPerQc;

static const uint TicksPerQcTimeCheck;

static const uint TimePerQc; // milliseconds

private:

RegexPattern \*const pattern;

StandardChars<Char>\* standardChars;

const Program\* program;

GroupInfo\* groupInfos;

LoopInfo\* loopInfos;

uint previousQcTime;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

RegexStats\* stats;

DebugWriter\* w;

#endif

public:

Matcher(Js::ScriptContext\* scriptContext, RegexPattern\* pattern);

static Matcher \*New(Js::ScriptContext\* scriptContext, RegexPattern\* pattern);

bool Match

( const Char\* const input

, const CharCount inputLength

, CharCount offset

, Js::ScriptContext \* scriptContext

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, RegexStats\* stats

, DebugWriter\* w

#endif

);

\_\_inline bool WasLastMatchSuccessful() const

{

return !groupInfos[0].IsUndefined();

}

\_\_inline int NumGroups() const

{

return program->numGroups;

}

\_\_inline GroupInfo GetGroup(int groupId) const

{

return \*GroupIdToGroupInfo(groupId);

}

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void Print(DebugWriter\* w, const Char\* const input, const CharCount inputLength, CharCount inputOffset, const uint8\* instPointer, ContStack &contStack, AssertionStack &assertionStack) const;

#endif

private:

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

void PushStats(ContStack& contStack, const Char\* const input) const;

void PopStats(ContStack& contStack, const Char\* const input) const;

void UnPopStats(ContStack& contStack, const Char\* const input) const;

void CompStats() const;

void InstStats() const;

#endif

private:

\_\_inline void QueryContinue(uint &qcTicks);

void DoQueryContinue(const uint qcTicks);

public:

static void TraceQueryContinue(const uint now);

private:

// Try backtracking, or return true if should stop. There could be a match using a later starting point.

bool Fail(const Char\* const input, CharCount &inputOffset, const uint8 \*&instPointer, ContStack &contStack, AssertionStack &assertionStack, uint &qcTicks);

bool RunContStack(const Char\* const input, CharCount &inputOffset, const uint8 \*&instPointer, ContStack &contStack, AssertionStack &assertionStack, uint &qcTicks);

// As above, but control whether to try backtracking or later matches

\_\_inline bool HardFail(const Char\* const input, const CharCount inputLength, CharCount &matchStart, CharCount &inputOffset, const uint8 \*&instPointer, ContStack &contStack, AssertionStack &assertionStack, uint &qcTicks, HardFailMode mode);

\_\_inline void Run(const Char\* const input, const CharCount inputLength, CharCount &matchStart, CharCount &nextSyncInputOffset, ContStack &contStack, AssertionStack &assertionStack, uint &qcTicks);

\_\_inline bool MatchHere(const Char\* const input, const CharCount inputLength, CharCount &matchStart, CharCount &nextSyncInputOffset, ContStack &contStack, AssertionStack &assertionStack, uint &qcTicks);

// Return true if assertion succeeded

\_\_inline bool PopAssertion(CharCount &inputOffset, const uint8 \*&instPointer, ContStack &contStack, AssertionStack &assertionStack, bool isFailed);

\_\_inline Label InstPointerToLabel(const uint8\* inst) const

{

Assert(inst >= program->rep.insts.insts && inst < program->rep.insts.insts + program->rep.insts.instsLen);

return (Label)((uint8\*)inst - program->rep.insts.insts);

}

\_\_inline uint8\* LabelToInstPointer(Label label) const

{

Assert(label < program->rep.insts.instsLen);

return program->rep.insts.insts + label;

}

template <typename T>

\_\_inline T\* LabelToInstPointer(Inst::InstTag tag, Label label) const

{

Assert(label + sizeof(T) <= program->rep.insts.instsLen);

Assert(((Inst\*)(program->rep.insts.insts + label))->tag == tag);

return (T\*)(program->rep.insts.insts + label);

}

\_\_inline LoopInfo\* LoopIdToLoopInfo(int loopId)

{

Assert(loopId >= 0 && loopId < program->numLoops);

return loopInfos + loopId;

}

public:

\_\_inline GroupInfo\* GroupIdToGroupInfo(int groupId) const

{

Assert(groupId >= 0 && groupId < program->numGroups);

return groupInfos + groupId;

}

Matcher \*CloneToScriptContext(Js::ScriptContext \*scriptContext, RegexPattern \*pattern);

private:

typedef bool (UnifiedRegex::Matcher::\*ComparerForSingleChar)(const Char left, const Char right);

ComparerForSingleChar comparerForSingleChar;

// Specialized matcher for regex c - case insensitive

\_\_inline bool MatchSingleCharCaseInsensitive(const Char\* const input, const CharCount inputLength, CharCount offset, const Char c);

\_\_inline bool MatchSingleCharCaseInsensitiveHere(CaseInsensitive::MappingSource mappingSource, const Char\* const input, CharCount offset, const Char c);

// Specialized matcher for regex c - case sensitive

\_\_inline bool MatchSingleCharCaseSensitive(const Char\* const input, const CharCount inputLength, CharCount offset, const Char c);

// Specialized matcher for regex \b\w+\b

\_\_inline bool MatchBoundedWord(const Char\* const input, const CharCount inputLength, CharCount offset);

// Specialized matcher for regex ^\s\*|\s\*$

\_\_inline bool MatchLeadingTrailingSpaces(const Char\* const input, const CharCount inputLength, CharCount offset);

// Specialized matcher for octoquad patterns

\_\_inline bool MatchOctoquad(const Char\* const input, const CharCount inputLength, CharCount offset, OctoquadMatcher\* matcher);

// Specialized matcher for regex ^literal

\_\_inline bool MatchBOILiteral2(const Char \* const input, const CharCount inputLength, CharCount offset, DWORD literal2);

void SaveInnerGroups(const int fromGroupId, const int toGroupId, const bool reset, const Char \*const input, ContStack &contStack);

void DoSaveInnerGroups(const int fromGroupId, const int toGroupId, const bool reset, const Char \*const input, ContStack &contStack);

void SaveInnerGroups\_AllUndefined(const int fromGroupId, const int toGroupId, const Char \*const input, ContStack &contStack);

void DoSaveInnerGroups\_AllUndefined(const int fromGroupId, const int toGroupId, const Char \*const input, ContStack &contStack);

void ResetGroup(int groupId);

void ResetInnerGroups(int minGroupId, int maxGroupId);

#if DBG

void ResetLoopInfos();

#endif

};

}

#undef INST\_BODY\_FREE

#undef INST\_BODY\_PRINT

#undef INST\_BODY

#undef CONT\_PRINT

#undef CONT\_BODY

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

// 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"

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

namespace UnifiedRegex

{

const wchar\_t\* RegexStats::PhaseNames[RegexStats::NumPhases] = { L"parse", L"compile", L"execute" };

const wchar\_t\* RegexStats::UseNames[RegexStats::NumUses] = { L"match", L"exec", L"test", L"replace", L"split", L"search" };

RegexStats::RegexStats(RegexPattern\* pattern)

: pattern(pattern)

, inputLength(0)

, numCompares(0)

, numPushes(0)

, numPops(0)

, stackHWM(0)

, numInsts(0)

{

for (int i = 0; i < NumPhases; i++)

phaseTicks[i] = 0;

for (int i = 0; i < NumUses; i++)

useCounts[i] = 0;

}

void RegexStats::Print(DebugWriter\* w, RegexStats\* totals, Ticks ticksPerMillisecond)

{

if (pattern == 0)

w->PrintEOL(L"TOTAL");

else

pattern->Print(w);

w->EOL();

w->Indent();

for (int i = 0; i < NumPhases; i++)

{

double ms = (double)phaseTicks[i] / (double)ticksPerMillisecond;

if (totals == 0 || totals->phaseTicks[i] == 0)

w->PrintEOL(L"%-12s: %10.4fms", PhaseNames[i], ms);

else

{

double pc = (double)phaseTicks[i] \* 100.0 / (double)totals->phaseTicks[i];

w->PrintEOL(L"%-12s: %10.4fms (%10.4f%%)", PhaseNames[i], ms, pc);

}

}

for (int i = 0; i < NumUses; i++)

{

if (useCounts[i] > 0)

{

if (totals == 0 || totals->useCounts[i] == 0)

w->PrintEOL(L"#%-11s: %10I64u", UseNames[i], useCounts[i]);

else

{

double pc = (double)useCounts[i] \* 100.0 / (double)totals->useCounts[i];

w->PrintEOL(L"#%-11s: %10I64u (%10.4f%%)", UseNames[i], useCounts[i], pc);

}

}

}

if (inputLength > 0)

{

double r = (double)numCompares \* 100.0 / (double)inputLength;

if (totals == 0 || totals->numCompares == 0)

w->PrintEOL(L"numCompares : %10.4f%%", r);

else

{

double pc = (double)numCompares \* 100.0 / (double)totals->numCompares;

w->PrintEOL(L"numCompares : %10.4f%% (%10.4f%%)", r, pc);

}

}

if (totals == 0 || totals->inputLength == 0)

w->PrintEOL(L"inputLength : %10I64u", inputLength);

else

{

double pc = (double)inputLength \* 100.0 / (double)totals->inputLength;

w->PrintEOL(L"inputLength : %10I64u (%10.4f%%)", inputLength, pc);

}

if (totals == 0 || totals->numPushes == 0)

w->PrintEOL(L"numPushes : %10I64u", numPushes);

else

{

double pc = (double)numPushes \* 100.0 / (double)totals->numPushes;

w->PrintEOL(L"numPushes : %10I64u (%10.4f%%)", numPushes, pc);

}

if (totals == 0 || totals->numPops == 0)

w->PrintEOL(L"numPops : %10I64u", numPops);

else

{

double pc = (double)numPops \* 100.0 / (double)totals->numPops;

w->PrintEOL(L"numPops : %10I64u (%10.4f%%)", numPops, pc);

}

if (totals == 0 || totals->stackHWM == 0)

w->PrintEOL(L"stackHWM : %10I64u", stackHWM);

else

{

double pc = (double)stackHWM \* 100.0 / (double)totals->stackHWM;

w->PrintEOL(L"stackHWM : %10I64u (%10.4f%%)", stackHWM, pc);

}

if (totals == 0 || totals->numInsts == 0)

w->PrintEOL(L"numInsts : %10I64u", numInsts);

else

{

double pc = (double)numInsts \* 100.0 / (double)totals->numInsts;

w->PrintEOL(L"numInsts : %10I64u (%10.4f%%)", numInsts, pc);

}

w->Unindent();

}

void RegexStats::Add(RegexStats\* other)

{

for (int i = 0; i < NumPhases; i++)

phaseTicks[i] += other->phaseTicks[i];

for (int i = 0; i < NumUses; i++)

useCounts[i] += other->useCounts[i];

inputLength += other->inputLength;

numCompares += other->numCompares;

numPushes += other->numPushes;

numPops += other->numPops;

if (other->stackHWM > stackHWM)

stackHWM = other->stackHWM;

numInsts += other->numInsts;

}

RegexStats::Ticks RegexStatsDatabase::Now()

{

LARGE\_INTEGER tmp;

if (QueryPerformanceCounter(&tmp))

return tmp.QuadPart;

else

{

Assert(false);

return 0;

}

}

RegexStats::Ticks RegexStatsDatabase::Freq()

{

LARGE\_INTEGER tmp;

if (QueryPerformanceFrequency(&tmp))

{

return tmp.QuadPart / 1000;

}

else

{

Assert(false);

return 1;

}

}

RegexStatsDatabase::RegexStatsDatabase(ArenaAllocator\* allocator)

: start(0), allocator(allocator)

{

ticksPerMillisecond = Freq();

map = Anew(allocator, RegexStatsMap, allocator, 17);

}

RegexStats\* RegexStatsDatabase::GetRegexStats(RegexPattern\* pattern)

{

Js::InternalString str = pattern->GetSource();

RegexStats \*res;

if (!map->TryGetValue(str, &res))

{

res = Anew(allocator, RegexStats, pattern);

map->Add(str, res);

}

return res;

}

void RegexStatsDatabase::BeginProfile()

{

start = Now();

}

void RegexStatsDatabase::EndProfile(RegexStats\* stats, RegexStats::Phase phase)

{

stats->phaseTicks[phase] += Now() - start;

}

void RegexStatsDatabase::Print(DebugWriter\* w)

{

RegexStats totals(0);

Output::Print(L"Regular Expression Statistics\n");

Output::Print(L"=============================\n");

for (int i = 0; i < map->Count(); i++)

totals.Add(map->GetValueAt(i));

for (int i = 0; i < map->Count(); i++)

map->GetValueAt(i)->Print(w, &totals, ticksPerMillisecond);

totals.Print(w, 0, ticksPerMillisecond);

allocator->Free(w, sizeof(DebugWriter));

}

}

#endif

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

// 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

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

namespace UnifiedRegex

{

struct RegexStats

{

typedef long long Ticks;

enum Phase

{

Parse,

Compile,

Execute,

NumPhases

};

static const wchar\_t\* PhaseNames[NumPhases];

enum Use

{

Match,

Exec,

Test,

Replace,

Split,

Search,

NumUses

};

static const wchar\_t\* UseNames[NumUses];

RegexPattern\* pattern; // null => total record

// Time spent on regex

Ticks phaseTicks[NumPhases];

// How is regex used?

uint64 useCounts[NumUses];

// Total input length

uint64 inputLength;

// Total chars looked at (may be > length if backtrack, < length if using Boyer-Moore)

uint64 numCompares;

// Number of continuation stack pushes

uint64 numPushes;

// Number of continuation stack pops

uint64 numPops;

// Continuation stack high-water-mark

uint64 stackHWM;

// Number of instructions executed

uint64 numInsts;

RegexStats(RegexPattern\* pattern);

void Print(DebugWriter\* w, RegexStats\* totals, Ticks ticksPerMillisecond);

void Add(RegexStats\* other);

};

typedef JsUtil::BaseDictionary<Js::InternalString, RegexStats\*, ArenaAllocator, PrimeSizePolicy, DefaultComparer, JsUtil::DictionaryEntry> RegexStatsMap;

class RegexStatsDatabase

{

private:

RegexStats::Ticks ticksPerMillisecond;

RegexStats::Ticks start;

ArenaAllocator\* allocator;

RegexStatsMap\* map;

static RegexStats::Ticks Now();

static RegexStats::Ticks Freq();

public:

RegexStatsDatabase(ArenaAllocator\* allocator);

RegexStats\* GetRegexStats(RegexPattern\* pattern);

void BeginProfile();

void EndProfile(RegexStats\* stats, RegexStats::Phase phase);

void Print(DebugWriter\* w);

};

}

#endif

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

// 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 "rtError.h"

// PUBLIC ERROR codes

// verify HR is as-expected for the Legacy (private) error JSERR\_CantExecute

C\_ASSERT(JSCRIPT\_E\_CANTEXECUTE != JSERR\_CantExecute);

// verify the HR value (as MAKE\_HRESULT(SEVERITY\_ERROR, FACILITY\_CONTROL, 0x1393))

C\_ASSERT(JSERR\_CantExecute == 0x800A1393);

// verify HR matches between public SDK and private (.h) files

C\_ASSERT(JSCRIPT\_E\_CANTEXECUTE == JSPUBLICERR\_CantExecute);

// verify the HR value (as MAKE\_HRESULT(SEVERITY\_ERROR, FACILITY\_JSCRIPT, 0x0001))

C\_ASSERT(JSPUBLICERR\_CantExecute == 0x89020001L);

// /PUBLIC ERROR codes

// boundary check - all errNum should be capped to 10,000 (RTERROR\_STRINGFORMAT\_OFFSET) - except for VBSERR\_CantDisplayDate==32812

#define VERIFY\_BOUNDARY\_ERRNUM(name,errnum) C\_ASSERT(name == VBSERR\_CantDisplayDate || errnum < RTERROR\_STRINGFORMAT\_OFFSET);

#define RT\_ERROR\_MSG(name, errnum, str1, str2, jst, errorNumSource) VERIFY\_BOUNDARY\_ERRNUM(name, errnum)

#define RT\_PUBLICERROR\_MSG(name, errnum, str1, str2, jst, errorNumSource) VERIFY\_BOUNDARY\_ERRNUM(name, errnum)

#include "rterrors.h"

#undef RT\_PUBLICERROR\_MSG

#undef RT\_ERROR\_MSG

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

// 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

//

// Defines trappable runtime error handling.

//

#include "rterrors\_limits.h"

enum rtErrors

{

#define RT\_ERROR\_MSG(name, errnum, str1, str2, jst, errorNumSource) name = MAKE\_HR(errnum),

#define RT\_PUBLICERROR\_MSG(name, errnum, str1, str2, jst, errorNumSource) name = MAKE\_HRESULT(SEVERITY\_ERROR, FACILITY\_JSCRIPT, errnum),

#include "rterrors.h"

#undef RT\_PUBLICERROR\_MSG

#undef RT\_ERROR\_MSG

MWUNUSED\_rtError

};

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

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

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

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

#define IDS\_COMPILATION\_ERROR\_SOURCE 4096

#define IDS\_RUNTIME\_ERROR\_SOURCE 4097

#define IDS\_UNKNOWN\_RUNTIME\_ERROR 4098

#define IDS\_INFINITY 6000

#define IDS\_MINUSINFINITY 6001

#ifndef RT\_ERROR\_MSG

#define RT\_ERROR\_MSG(name, errnum, str1, str2, jst, errorNumSource)

#endif

#ifndef RT\_PUBLICERROR\_MSG

#define RT\_PUBLICERROR\_MSG(name, errnum, str1, str2, jst, errorNumSource)

#endif

#ifndef RT\_ERROR\_MSG\_UNUSED\_ENTRY

#define RT\_ERROR\_MSG\_UNUSED\_ENTRY

#endif

RT\_ERROR\_MSG(VBSERR\_None, 0, "", "", kjstError, 0)

//RT\_ERROR\_MSG(VBSERR\_ReturnWOGoSub, 3, "", "Return without GoSub")

RT\_ERROR\_MSG(VBSERR\_IllegalFuncCall, 5, "", "Invalid procedure call or argument", kjstTypeError, 0)

RT\_ERROR\_MSG(VBSERR\_Overflow, 6, "", "Overflow", kjstRangeError, 0)

RT\_ERROR\_MSG(VBSERR\_OutOfMemory, 7, "", "Out of memory", kjstError, 0)

RT\_ERROR\_MSG(VBSERR\_OutOfBounds, 9, "", "Subscript out of range", kjstRangeError, 0)

RT\_ERROR\_MSG(VBSERR\_ArrayLocked, 10, "", "This array is fixed or temporarily locked", kjstError, 0)

//RT\_ERROR\_MSG(VBSERR\_DivByZero, 11, "", "Division by zero", kjstError, 0)

RT\_ERROR\_MSG(VBSERR\_TypeMismatch, 13, "", "Type mismatch", kjstTypeError, 0)

//RT\_ERROR\_MSG(VBSERR\_OutOfStrSpace, 14, "", "Out of string space", kjstError, 0)

//RT\_ERROR\_MSG(VBSERR\_ExprTooComplex, 16, "", "Expression too complex")

//RT\_ERROR\_MSG(VBSERR\_CantContinue, 17, "", "Can't perform requested operation", kjstError, 0)

//RT\_ERROR\_MSG(VBSERR\_UserInterrupt, 18, "", "User interrupt occurred")

//RT\_ERROR\_MSG(VBSERR\_ResumeWOErr, 20, "", "Resume without error")

RT\_ERROR\_MSG(VBSERR\_OutOfStack, 28, "", "Out of stack space", kjstError, 0)

//RT\_ERROR\_MSG(VBSERR\_UndefinedProc, 35, "", "Sub or Function not defined", kjstError, 0)

//RT\_ERROR\_MSG(VBSERR\_TooManyClients, 47, "", "Too many DLL application clients")

RT\_ERROR\_MSG(VBSERR\_DLLLoadErr, 48, "", "Error in loading DLL", kjstError, 0)

//RT\_ERROR\_MSG(VBSERR\_DLLBadCallingConv, 49, "", "Bad DLL calling convention")

RT\_ERROR\_MSG(VBSERR\_InternalError, 51, "", "Internal error", kjstError, 0)

//RT\_ERROR\_MSG(VBSERR\_BadFileNameOrNumber, 52, "", "Bad file name or number", kjstError, 0)

RT\_ERROR\_MSG(VBSERR\_FileNotFound, 53, "", "File not found", kjstError, 0)

//RT\_ERROR\_MSG(VBSERR\_BadFileMode, 54, "", "Bad file mode", kjstError, 0)

//RT\_ERROR\_MSG(VBSERR\_FileAlreadyOpen, 55, "", "File already open", kjstError, 0)

RT\_ERROR\_MSG(VBSERR\_IOError, 57, "", "Device I/O error", kjstError, 0)

RT\_ERROR\_MSG(VBSERR\_FileAlreadyExists, 58, "", "File already exists", kjstError, 0)

//RT\_ERROR\_MSG(VBSERR\_BadRecordLen, 59, "", "Bad record length")

RT\_ERROR\_MSG(VBSERR\_DiskFull, 61, "", "Disk full", kjstError, 0)

//RT\_ERROR\_MSG(VBSERR\_EndOfFile, 62, "", "Input past end of file", kjstError, 0)

//RT\_ERROR\_MSG(VBSERR\_BadRecordNum, 63, "", "Bad record number")

RT\_ERROR\_MSG(VBSERR\_TooManyFiles, 67, "", "Too many files", kjstError, 0)

//RT\_ERROR\_MSG(VBSERR\_DevUnavailable, 68, "", "Device unavailable", kjstError, 0)

RT\_ERROR\_MSG(VBSERR\_PermissionDenied, 70, "", "Permission denied", kjstError, 0)

//RT\_ERROR\_MSG(VBSERR\_DiskNotReady, 71, "", "Disk not ready", kjstError, 0)

//RT\_ERROR\_MSG(VBSERR\_DifferentDrive, 74, "", "Can't rename with different drive", kjstError, 0)

RT\_ERROR\_MSG(VBSERR\_PathFileAccess, 75, "", "Path/File access error", kjstError, 0)

RT\_ERROR\_MSG(VBSERR\_PathNotFound, 76, "", "Path not found", kjstError, 0)

//RT\_ERROR\_MSG(VBSERR\_ObjNotSet, 91, "", "Object variable or With block variable not set", kjstTypeError, 0)

//RT\_ERROR\_MSG(VBSERR\_IllegalFor, 92, "", "For loop not initialized", kjstError, 0)

//RT\_ERROR\_MSG(VBSERR\_BadPatStr, 93, "", "Invalid pattern string")

//RT\_ERROR\_MSG(VBSERR\_CantUseNull, 94, "", "Invalid use of Null", kjstError, 0)

//RT\_ERROR\_MSG(VBSERR\_UserDefined, 95, "", "Application-defined or object-defined error")

RT\_ERROR\_MSG(VBSERR\_CantCreateTmpFile, 322, "", "Can't create necessary temporary file", kjstError, 0)

//RT\_ERROR\_MSG(VBSERR\_InvalidResourceFormat, 325, "", "Invalid format in resource file")

//RT\_ERROR\_MSG(VBSERR\_InvalidPropertyValue, 380, "", "Invalid property value")

//RT\_ERROR\_MSG(VBSERR\_NoSuchControlOrProperty, 423, "", "Property or method not found")

//RT\_ERROR\_MSG(VBSERR\_NotObject, 424, "", "Object required", kjstTypeError, 0)

RT\_ERROR\_MSG(VBSERR\_CantCreateObject, 429, "", "Automation server can't create object", kjstError, 0)

RT\_ERROR\_MSG(VBSERR\_OLENotSupported, 430, "", "Class doesn't support Automation", kjstError, 0)

RT\_ERROR\_MSG(VBSERR\_OLEFileNotFound, 432, "", "File name or class name not found during Automation operation", kjstError, 0)

RT\_ERROR\_MSG(VBSERR\_OLENoPropOrMethod, 438, "Object doesn't support property or method '%s'", "Object doesn't support this property or method", kjstTypeError, 0)

//RT\_ERROR\_MSG(VBSERR\_OLEAutomationError, 440, "", "Automation error", kjstError, 0)

//RT\_ERROR\_MSG(VBSERR\_LostTLB, 442, "", "Connection to type library or object library for remote process has been lost. Press OK for dialog to remove reference.")

//RT\_ERROR\_MSG(VBSERR\_OLENoDefault, 443, "", "Automation object does not have a default value")

RT\_ERROR\_MSG(VBSERR\_ActionNotSupported, 445, "", "Object doesn't support this action", kjstTypeError, 0)

RT\_ERROR\_MSG(VBSERR\_NamedArgsNotSupported, 446, "", "Object doesn't support named arguments", kjstError, 0)

RT\_ERROR\_MSG(VBSERR\_LocaleSettingNotSupported, 447, "", "Object doesn't support current locale setting", kjstError, 0)

RT\_ERROR\_MSG(VBSERR\_NamedParamNotFound, 448, "", "Named argument not found", kjstError, 0)

RT\_ERROR\_MSG(VBSERR\_ParameterNotOptional, 449, "Argument to the function '%s' is not optional", "Argument not optional", kjstTypeError, 0)

RT\_ERROR\_MSG(VBSERR\_FuncArityMismatch, 450, "", "Wrong number of arguments or invalid property assignment", kjstTypeError, 0)

RT\_ERROR\_MSG(VBSERR\_NotEnum, 451, "", "Object not a collection", kjstTypeError, 0)

//RT\_ERROR\_MSG(VBSERR\_InvalidOrdinal, 452, "", "Invalid ordinal")

RT\_ERROR\_MSG(VBSERR\_InvalidDllFunctionName, 453, "", "Specified DLL function not found", kjstError, 0)

//RT\_ERROR\_MSG(VBSERR\_CodeResourceNotFound, 454, "", "Code resource not found")

//RT\_ERROR\_MSG(VBSERR\_CodeResourceLockError, 455, "", "Code resource lock error")

//RT\_ERROR\_MSG(VBSERR\_DuplicateKey, 457, "", "This key is already associated with an element of this collection")

RT\_ERROR\_MSG(VBSERR\_InvalidTypeLibVariable, 458, "", "Variable uses an Automation type not supported in JavaScript", kjstTypeError, 0)

RT\_ERROR\_MSG(VBSERR\_ServerNotFound, 462, "", "The remote server machine does not exist or is unavailable", kjstError, 0)

//RT\_ERROR\_MSG(VBSERR\_InvalidPicture, 481, "", "Invalid picture")

//RT\_ERROR\_MSG(VBSERR\_CantAssignTo, 501, "", "Cannot assign to variable", kjstReferenceError, 0)

//RT\_ERROR\_MSG(VBSERR\_NotSafeForScripting, 502, "", "Object not safe for scripting", kjstError, 0)

//RT\_ERROR\_MSG(VBSERR\_NotSafeForInitializing, 503, "", "Object not safe for initializing", kjstError, 0)

//RT\_ERROR\_MSG(VBSERR\_NotSafeForCreating, 504, "", "Object not safe for creating", kjstError, 0)

//RT\_ERROR\_MSG(VBSERR\_InvalidReference, 505, "", "Invalid or unqualified reference")

//RT\_ERROR\_MSG(VBSERR\_ClassNotDefined, 506, "", "Class not defined")

RT\_ERROR\_MSG(VBSERR\_ComponentException, 507, "", "An exception occurred", kjstError, 0)

RT\_ERROR\_MSG(JSERR\_CantAssignThis, 5000, "", "Cannot assign to 'this'", kjstError, 0)

RT\_ERROR\_MSG(JSERR\_NeedNumber, 5001, "'%s' is not a number", "Number expected", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_NeedFunction, 5002, "'%s' is not a function", "Function expected", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_CantAsgCall, 5003, "", "Cannot assign to a function result", kjstReferenceError, 0)

RT\_ERROR\_MSG(JSERR\_NeedIndxObj, 5004, "'%s' is not an indexable object", "Cannot index object", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_NeedString, 5005, "'%s' is not a string", "String expected", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_NeedDate, 5006, "'%s' is not a date object", "Date object expected", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_NeedObject, 5007, "'%s' is null or not an object", "Object expected", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_CantAssignTo, 5008, "", "Invalid left-hand side in assignment", kjstReferenceError, 0)

RT\_ERROR\_MSG(JSERR\_UndefVariable, 5009, "'%s' is undefined", "Undefined identifier", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_NeedBoolean, 5010, "'%s' is not a boolean", "Boolean expected", kjstTypeError, 0)

// This is the legacy error code for JScript.

RT\_ERROR\_MSG(JSERR\_CantExecute, 5011, "", "Can't execute code from a freed script", kjstError, 0)

// JScript9 is to use the newer JSCRIPT\_E\_CANTEXECUTE public HResult.

RT\_PUBLICERROR\_MSG(JSPUBLICERR\_CantExecute, 1, "", "Can't execute code from a freed script", kjstError, 0)

RT\_ERROR\_MSG(JSERR\_CantDelete, 5012, "Cannot delete '%s'", "Object member expected", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_NeedVBArray, 5013, "'%s' is not a VBArray", "VBArray expected", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_NeedInternalObj, 5014, "'%s' is not a JavaScript object", "JavaScript object expected", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_NeedEnumerator, 5015, "'%s' is not an enumerator object", "Enumerator object expected", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_NeedRegExp, 5016, "'%s' is not a regular expression object", "Regular Expression object expected", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_RegExpSyntax, 5017, "", "Syntax error in regular expression", kjstSyntaxError, 0)

RT\_ERROR\_MSG(JSERR\_RegExpBadQuant, 5018, "", "Unexpected quantifier", kjstSyntaxError, 0)

RT\_ERROR\_MSG(JSERR\_RegExpNoBracket, 5019, "", "Expected ']' in regular expression", kjstSyntaxError, 0)

RT\_ERROR\_MSG(JSERR\_RegExpNoParen, 5020, "", "Expected ')' in regular expression", kjstSyntaxError, 0)

RT\_ERROR\_MSG(JSERR\_RegExpBadRange, 5021, "", "Invalid range in character set", kjstSyntaxError, 0)

RT\_ERROR\_MSG(JSERR\_UncaughtException, 5022, "", "Exception thrown and not caught", kjstError, 0)

RT\_ERROR\_MSG(JSERR\_InvalidPrototype, 5023, "", "Function does not have a valid prototype object", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_URIEncodeError, 5024, "", "The URI to be encoded contains an invalid character", kjstURIError, 0)

RT\_ERROR\_MSG(JSERR\_URIDecodeError, 5025, "", "The URI to be decoded is not a valid encoding", kjstURIError, 0)

RT\_ERROR\_MSG(JSERR\_FractionOutOfRange, 5026, "", "The number of fractional digits is out of range", kjstRangeError, 0)

RT\_ERROR\_MSG(JSERR\_PrecisionOutOfRange, 5027, "", "The precision is out of range", kjstRangeError, 0)

RT\_ERROR\_MSG(JSERR\_NeedArrayOrArg, 5028, "%s is not an Array or arguments object", "Array or arguments object expected", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_ArrayLengthConstructIncorrect, 5029, "", "Array length must be a finite positive integer", kjstRangeError, 0)

RT\_ERROR\_MSG(JSERR\_ArrayLengthAssignIncorrect, 5030, "", "Array length must be assigned a finite positive integer", kjstRangeError, 0)

RT\_ERROR\_MSG(JSERR\_NeedArrayObject, 5031, "%s is not an Array object", "Array object expected", kjstTypeError, 0)

// RETIRED ECMACP removd ;; RT\_ERROR\_MSG(JSERR\_NoCPEval, 5032, "", "'eval' is not available in the ECMA 327 Compact Profile", kjstEvalError, 0)

// RETIRED ECMACP removd ;; RT\_ERROR\_MSG(JSERR\_NoCPFunction, 5033, "", "Function constructor is not available in the ECMA 327 Compact Profile", kjstEvalError, 0)

RT\_ERROR\_MSG(JSERR\_JSONSerializeCircular, 5034, "", "Circular reference in value argument not supported", kjstError, 0)

RT\_ERROR\_MSG(JSERR\_JSONInvalidReplacer, 5035, "", "Invalid replacer argument", kjstError, 0)

RT\_ERROR\_MSG(JSERR\_InvalidAttributeTrue,5036,"'%s' attribute on the property descriptor cannot be set to 'true' on this object","",kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_InvalidAttributeFalse,5037,"'%s' attribute on the property descriptor cannot be set to 'false' on this object","",kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_ArgListTooLarge, 5038, "", "Argument list too large to apply", kjstRangeError, 0)

RT\_ERROR\_MSG(JSERR\_ConstRedeclaration, 5039, "Redeclaration of const '%s'", "Redeclaration of const property", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_CyclicProtoValue, 5040, "", "Cyclic \_\_proto\_\_ value", kjstError, 0)

RT\_ERROR\_MSG(JSERR\_CantDeleteExpr, 5041, "Calling delete on '%s' is not allowed in strict mode", "Object member not configurable", kjstTypeError, 0) // string 4

RT\_ERROR\_MSG(JSERR\_RefErrorUndefVariable, 5042, "", "Variable undefined in strict mode", kjstReferenceError, 0) // string 10

RT\_ERROR\_MSG(JSERR\_AccessCallerRestricted, 5043, "", "Accessing the 'caller' property is restricted in this context", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_AccessCallee, 5044, "", "Accessing the 'callee' property of an arguments object is not allowed in strict mode", kjstTypeError, 0) // string 2

RT\_ERROR\_MSG(JSERR\_CantAssignToReadOnly, 5045, "", "Assignment to read-only properties is not allowed in strict mode", kjstTypeError, 0) // string 5

RT\_ERROR\_MSG(JSERR\_NonExtensibleObject, 5046, "", "Cannot create property for a non-extensible object", kjstTypeError, 0) // string 6

RT\_ERROR\_MSG(JSERR\_Property\_CannotSet\_NullOrUndefined, 5047, "Unable to set property '%s' of undefined or null reference", "Object expected", kjstTypeError, JSERR\_NeedObject)

RT\_ERROR\_MSG(JSERR\_Property\_CannotGet\_NullOrUndefined, 5048, "Unable to get property '%s' of undefined or null reference", "Object expected", kjstTypeError, JSERR\_NeedObject)

RT\_ERROR\_MSG(JSERR\_Property\_CannotDelete\_NullOrUndefined, 5049, "Unable to delete property '%s' of undefined or null reference", "Object expected", kjstTypeError, JSERR\_NeedObject)

RT\_ERROR\_MSG(JSERR\_Property\_VarDate, 5050, "Unable to access property '%s': type 'VarDate' does not support user-defined properties", "Object expected", kjstTypeError, JSERR\_NeedObject)

RT\_ERROR\_MSG(JSERR\_Property\_NeedFunction, 5051, "The value of the property '%s' is not a Function object", "Function expected", kjstTypeError, JSERR\_NeedFunction)

RT\_ERROR\_MSG(JSERR\_Property\_NeedFunction\_NullOrUndefined, 5052, "The value of the property '%s' is null or undefined, not a Function object", "Function expected", kjstTypeError, JSERR\_NeedObject)

RT\_ERROR\_MSG(JSERR\_Property\_CannotHaveAccessorsAndValue, 5053, "", "Property cannot have both accessors and a value", kjstTypeError, VBSERR\_ActionNotSupported)

RT\_ERROR\_MSG(JSERR\_This\_NullOrUndefined, 5054, "%s: 'this' is null or undefined", "'this' is null or undefined", kjstTypeError, JSERR\_NeedObject) // {Locked="\'this\'"}

RT\_ERROR\_MSG(JSERR\_This\_NeedObject, 5055, "%s: 'this' is not an Object", "Object expected", kjstTypeError, JSERR\_NeedObject) // {Locked="\'this\'"}

RT\_ERROR\_MSG(JSERR\_This\_NeedFunction, 5056, "%s: 'this' is not a Function object", "Function expected", kjstTypeError, JSERR\_NeedFunction) // {Locked="\'this\'"}

RT\_ERROR\_MSG(JSERR\_This\_NeedString, 5057, "%s: 'this' is not a String object", "String expected", kjstTypeError, JSERR\_NeedString) // {Locked="\'this\'"}

RT\_ERROR\_MSG(JSERR\_This\_NeedBoolean, 5058, "%s: 'this' is not a Boolean object", "Boolean expected", kjstTypeError, JSERR\_NeedBoolean) // {Locked="\'this\'"}

RT\_ERROR\_MSG(JSERR\_This\_NeedDate, 5059, "%s: 'this' is not a Date object", "Date expected", kjstTypeError, JSERR\_NeedDate) // {Locked="\'this\'"}

RT\_ERROR\_MSG(JSERR\_This\_NeedNumber, 5060, "%s: 'this' is not a Number object", "Number expected", kjstTypeError, JSERR\_NeedNumber) // {Locked="\'this\'"}

RT\_ERROR\_MSG(JSERR\_This\_NeedVBArray, 5061, "%s: 'this' is not a VBArray object", "VBArray expected", kjstTypeError, JSERR\_NeedVBArray) // {Locked="\'this\'"}

RT\_ERROR\_MSG(JSERR\_This\_NeedInternalObject, 5062, "%s: 'this' is not a JavaScript object", "JavaScript object expected", kjstTypeError, JSERR\_NeedInternalObj) // {Locked="\'this\'"}

RT\_ERROR\_MSG(JSERR\_This\_NeedEnumerator, 5063, "%s: 'this' is not an Enumerator object", "Enumerator object expected", kjstTypeError, JSERR\_NeedEnumerator) // {Locked="\'this\'"}

RT\_ERROR\_MSG(JSERR\_This\_NeedRegExp, 5064, "%s: 'this' is not a RegExp object", "RegExp object expected", kjstTypeError, JSERR\_NeedRegExp) // {Locked="\'this\'"}

RT\_ERROR\_MSG(JSERR\_FunctionArgument\_Invalid, 5065, "%s: invalid argument", "Invalid function argument", kjstTypeError, VBSERR\_IllegalFuncCall)

RT\_ERROR\_MSG(JSERR\_FunctionArgument\_NeedObject, 5066, "%s: argument is not an Object", "Object expected", kjstTypeError, JSERR\_NeedObject)

RT\_ERROR\_MSG(JSERR\_FunctionArgument\_NeedInternalObject, 5067, "%s: argument is not a JavaScript object", "JavaScript object expected", kjstTypeError, JSERR\_NeedInternalObj)

RT\_ERROR\_MSG(JSERR\_FunctionArgument\_NeedFunction, 5068, "%s: argument is not a Function object", "Function expected", kjstTypeError, JSERR\_NeedFunction)

RT\_ERROR\_MSG(JSERR\_FunctionArgument\_NeedVBArray, 5069, "%s: argument is not a VBArray object", "VBArray expected", kjstTypeError, JSERR\_NeedVBArray)

RT\_ERROR\_MSG(JSERR\_FunctionArgument\_NullOrUndefined, 5070, "%s: argument is null or undefined", "Object expected", kjstTypeError, JSERR\_NeedObject)

RT\_ERROR\_MSG(JSERR\_FunctionArgument\_NotObjectOrNull, 5071, "%s: argument is not an Object and is not null", "Object expected", kjstTypeError, JSERR\_NeedObject)

RT\_ERROR\_MSG(JSERR\_FunctionArgument\_InvalidLength, 5072, "%s: argument does not have a valid 'length' property", "Invalid 'length' property", kjstTypeError, VBSERR\_ActionNotSupported)

RT\_ERROR\_MSG(JSERR\_FunctionArgument\_NeedArrayOrArguments, 5073, "%s: Array or arguments object expected", "Array or arguments object expected", kjstTypeError, JSERR\_NeedArrayOrArg)

RT\_ERROR\_MSG(JSERR\_Operand\_Invalid\_NeedObject, 5074, "Invalid operand to '%s': Object expected", "Invalid Operand", kjstTypeError, JSERR\_NeedObject)

RT\_ERROR\_MSG(JSERR\_Operand\_Invalid\_NeedFunction, 5075, "Invalid operand to '%s': Function expected", "Invalid Operand", kjstTypeError, JSERR\_NeedFunction)

RT\_ERROR\_MSG(JSERR\_PropertyDescriptor\_Invalid, 5076, "Invalid descriptor for property '%s'", "Invalid property descriptor", kjstTypeError, JSERR\_NeedObject)

RT\_ERROR\_MSG(JSERR\_DefineProperty\_NotExtensible, 5077, "Cannot define property '%s': object is not extensible", "Cannot define property: object is not extensible", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_DefineProperty\_NotConfigurable, 5078, "Cannot redefine non-configurable property '%s'", "Cannot redefine non-configurable property", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_DefineProperty\_NotWritable, 5079, "Cannot modify non-writable property '%s'", "Cannot modify non-writable property", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_DefineProperty\_LengthNotWritable, 5080, "Cannot modify property '%s': 'length' is not writable", "Cannot modify property: 'length' is not writable", kjstTypeError, 0) // {Locked="\'length\'"}

RT\_ERROR\_MSG(JSERR\_DefineProperty\_Default, 5081, "Cannot define property '%s'", "Cannot define property", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_InvalidTypedArray\_Constructor, 5082, "", "Typed array constructor argument is invalid", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_This\_NeedTypedArray, 5083, "", "'this' is not a typed array object", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_InvalidTypedArrayLength, 5084, "", "Invalid offset/length when creating typed array", kjstRangeError, 0)

RT\_ERROR\_MSG(JSERR\_InvalidTypedArraySubarrayLength, 5085, "", "Invalid begin/end value in typed array subarray method", kjstRangeError, 0)

RT\_ERROR\_MSG(JSERR\_TypedArray\_NeedSource, 5086, "", "Invalid source in typed array set", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_This\_NeedDataView, 5087, "", "'this' is not a DataView object", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_DataView\_NeedArgument, 5088, "Required argument %s in DataView method is not specified", "Invalid arguments in DataView", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_DataView\_InvalidOffset, 5089, "", "DataView operation access beyond specified buffer length", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_DataView\_InvalidArugment, 5090, "DataView constructor argument %s is invalid", "Invalid arguments in DataView", kjstRangeError, 0)

RT\_ERROR\_MSG(JSERR\_InvalidFunctionSignature, 5091, "The function '%s' has an invalid signature and cannot be called", "invalid function signature", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_InvalidPropertySiganture, 5092, "The property '%s' has an invalid signature and cannot be accessed", "invalid property signature", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_InvalidRTCPropertyValueIn, 5093, "The runtimeclass %s that has Windows.Foundation.IPropertyValue as default interface is not supported as input parameter type", "invalid input parameter type", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_RTCInvalidRTCPropertyValueOut, 5094, "The object with interface Windows.Foundation.IPropertyValue that has runtimeclass name %s is not supported as out parameter", "invalid ouput parameter", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_AccessArgumentsRestricted, 5095, "", "Accessing the 'arguments' property is restricted in this context", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_This\_NeedInspectableObject, 5096, "%s: 'this' is not an Inspectable Object", "Inspectable Object expected", kjstTypeError, JSERR\_NeedObject) // {Locked="\'this\'"}

RT\_ERROR\_MSG(JSERR\_FunctionArgument\_NeedWinRTChar, 5097, "%s: could not convert argument to type 'char'", "Could not convert argument to type 'char'", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_FunctionArgument\_NeedWinRTGUID, 5098, "%s: could not convert argument to type 'GUID'", "Could not convert argument to type 'GUID'", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_ReturnValue\_NeedInspectable, 5099, "%s: could not convert return value to IInspectable", "IInspectable expected", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_MissingStructProperty, 5100, "Could not convert object to struct: object missing expected property '%s'", "Could not convert object to struct: object missing expected property", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_UnknownType, 5101, "Type '%s' not found", "Unknown type", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_WinRTFunction\_TooFewArguments, 5102, "%s: function called with too few arguments", "Function called with too few arguments", kjstError, 0)

RT\_ERROR\_MSG(JSERR\_UnconstructableClass, 5103, "%s: type is not constructible", "Type is not constructible", kjstError, 0)

RT\_ERROR\_MSG(JSERR\_InvalidPropertyValue, 5104, "Could not convert value to PropertyValue: %s not supported by PropertyValue", "Could not convert value to PropertyValue: Type not supported by PropertyValue", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_InvalidIInspectable, 5105, "Could not convert value to IInspectable: %s not supported by IInspectable", "Could not convert value to IInspectable: Type not supported by IInspectable", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_OutOfDateTimeRange, 5106, "", "Could not convert Date to Windows.Foundation.DateTime: value outside of valid range", kjstRangeError, 0)

RT\_ERROR\_MSG(JSERR\_OutOfTimeSpanRange, 5107, "", "Could not convert value to Windows.Foundation.TimeSpan: value outside of valid range", kjstRangeError, 0)

RT\_ERROR\_MSG(JSERR\_This\_ReleasedInspectableObject, 5108, "%s: The Inspectable object 'this' is released and cannot be accessed", "Invalid access to already released Inspectable Object", kjstReferenceError, JSERR\_NeedObject) // {Locked="\'this\'"}

RT\_ERROR\_MSG(JSERR\_AlreadyReleasedInspectableObject, 5109, "", "Cannot release already released Inspectable Object", kjstReferenceError, JSERR\_NeedObject) // {Locked="\'this\'"}

RT\_ERROR\_MSG(JSERR\_This\_NeedWinRTType, 5110, "'this' is not of expected type: %s", "'this' is not of the expected type", kjstTypeError, JSERR\_NeedObject) // {Locked="\'this\'"}

RT\_ERROR\_MSG(JSERR\_IllegalArraySizeAndLength, 5111, "", "Illegal length and size specified for the array", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_UnexpectedMetadataFailure, 5112, "%s: an unexpected failure occurred while trying to obtain metadata information", "An unexpected failure occurred while trying to obtain metadata information", kjstError, 0)

RT\_ERROR\_MSG(JSERR\_UseBeforeDeclaration, 5113, "", "Use before declaration", kjstReferenceError, 0)

RT\_ERROR\_MSG(JSERR\_ObjectIsAlreadyInitialized, 5114, "Cannot initialize '%s' object: 'this' is already initialized as '%s' object", "Cannot re-initialize 'this', object already initialized", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_ObjectIsNonExtensible, 5115, "Cannot initialize '%s' object: 'this' is not extensible", "Cannot initialize 'this' because it is a non-extensible object", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_NeedObjectOfType, 5116, "%s: 'this' is not a %s object", "'this' is not of the expected type", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_WeakMapSetKeyNotAnObject, 5117, "%s: 'key' is not an object", "'key' is not an object", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_OptionValueOutOfRange, 5118, "Option value '%s' for '%s' is outside of valid range. Expected: %s", "Option value is outside of valid range", kjstRangeError, 0)

RT\_ERROR\_MSG(JSERR\_NeedObjectOrString, 5119, "%s is not an object or a string", "Object or string expected", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_NotAConstructor, 5120, "Function '%s' is not a constructor", "This can't be used in a new statement", kjstTypeError, 0)

//Intl Specific

RT\_ERROR\_MSG(JSERR\_LocaleNotWellFormed, 5121, "Locale '%s' is not well-formed", "Locale is not well-formed", kjstRangeError, 0)

RT\_ERROR\_MSG(JSERR\_InvalidCurrencyCode, 5122, "Currency code '%s' is invalid", "Currency code is invalid", kjstRangeError, 0)

RT\_ERROR\_MSG(JSERR\_MissingCurrencyCode, 5123, "", "Currency code was not specified", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_InvalidDate, 5124, "", "Invalid Date", kjstRangeError, 0)

RT\_ERROR\_MSG(JSERR\_IntlNotAvailable, 5125, "", "Intl is not available.", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_ArgumentOutOfRange, 5130, "%s: argument out of range", "argument out of range", kjstRangeError, 0)

RT\_ERROR\_MSG(JSERR\_ErrorOnNew, 5131, "", "Function is not a constructor", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_This\_NeedArrayIterator, 5132, "%s: 'this' is not an Array Iterator object", "Array Iterator expected", kjstTypeError, 0) // {Locked="\'this\'"}

RT\_ERROR\_MSG(JSERR\_This\_NeedMapIterator, 5133, "%s: 'this' is not an Map Iterator object", "Map Iterator expected", kjstTypeError, 0) // {Locked="\'this\'"}

RT\_ERROR\_MSG(JSERR\_This\_NeedSetIterator, 5134, "%s: 'this' is not an Set Iterator object", "Set Iterator expected", kjstTypeError, 0) // {Locked="\'this\'"}

RT\_ERROR\_MSG(JSERR\_This\_NeedStringIterator, 5135, "%s: 'this' is not an String Iterator object", "String Iterator expected", kjstTypeError, 0) // {Locked="\'this\'"}

RT\_ERROR\_MSG(JSERR\_InvalidSpreadArgument, 5140, "%s: argument cannot be spread; expected Array or Object with a 'length' property", "Argument cannot be spread; expected Array or Object with a 'length' property", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_InvalidSpreadLength, 5141, "%s: argument cannot be spread; the 'length' property must be a number or convert to a number", "Argument cannot be spread; the 'length' property must be a number or convert to a number", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_BadSuperReference, 5145, "", "'super' can only be accessed from a subclass method", kjstReferenceError, 0)

RT\_ERROR\_MSG(JSERR\_DeletePropertyWithSuper, 5146, "Unable to delete property '%s' which has a super reference", "Unable to delete property with a super reference", kjstReferenceError, 0)

RT\_ERROR\_MSG(JSERR\_DetachedTypedArray, 5147, "%s: The ArrayBuffer is detached.", "The ArrayBuffer is detached.", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_AsmJsCompileError, 5148, "%s: Compiling asm.js failed.", "Compiling asm.js failed.", kjstError, 0)

/\* Error messages for misbehaved Async Operations for use in Promise.js \*/

RT\_ERROR\_MSG(ASYNCERR\_NoErrorInErrorState, 5200, "", "Status is 'error', but getResults did not return an error", kjstError, 0)

RT\_ERROR\_MSG(ASYNCERR\_InvalidStatusArg, 5201, "", "Missing or invalid status parameter passed to completed handler", kjstError, 0)

RT\_ERROR\_MSG(ASYNCERR\_InvalidSenderArg, 5202, "", "Missing or invalid sender parameter passed to completed handler", kjstError, 0)

// Error messages for hybrid debugging

RT\_ERROR\_MSG(DIAGERR\_FunctionCallNotSupported, 5400, "", "Function evaluation is disabled while debugging native code", kjstError, 0)

RT\_ERROR\_MSG(DIAGERR\_EvaluateNotSupported, 5401, "", "Evaluation of the JavaScript expression is not supported while debugging native code", kjstError, 0)

RT\_ERROR\_MSG(JSERR\_InvalidCodePoint, 5600, "Invalid code point %s", "Invalid code point", kjstRangeError, 0)

RT\_ERROR\_MSG(JSERR\_InvalidNormalizationForm, 5601, "Normalization form '%s' is invalid. Expected one of: ['NFC', 'NFD', 'NFKC', 'NFKD'].", "Invalid normalization form. Expected one of: ['NFC', 'NFD', 'NFKC', 'NFKD']", kjstRangeError, 0)

RT\_ERROR\_MSG(JSERR\_InvalidUnicodeCharacter, 5602, "Failed to normalize: invalid or missing unicode character at index %d.", "Failed to normalize: invalid or missing unicode character.", kjstRangeError, 0)

RT\_ERROR\_MSG(JSERR\_FailedToNormalize, 5603, "Failed to normalize string.", "Failed to normalize string.", kjstRangeError, 0)

RT\_ERROR\_MSG(JSERR\_NeedArrayBufferObject, 5604, "%s is not an ArrayBuffer", "ArrayBuffer object expected", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_NeedSymbol, 5605, "'%s' is not a symbol", "Symbol expected", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_This\_NeedSymbol, 5606, "%s: 'this' is not a Symbol object", "Symbol expected", kjstTypeError, JSERR\_NeedSymbol) // {Locked="\'this\'"}

RT\_ERROR\_MSG(JSERR\_RegExpNoCurlyBracket, 5607, "", "Expected '}' in regular expression", kjstSyntaxError, 0)

RT\_ERROR\_MSG(JSERR\_NeedProxyArgument, 5608, "", "Proxy requires more than 1 arguments", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_InvalidProxyArgument, 5609, "Proxy argument %s is not a valid object", "Invalid Proxy argument", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_InvalidProxyObject, 5610, "Revocable method requires Proxy object", "Revocable method requires Proxy object", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_ErrorOnRevokedProxy, 5611, "method %s is called on a revoked Proxy object", "trap called on a revoked Proxy object", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_InconsistentTrapResult, 5612, "Invariant check failed for %s proxy trap", "Invariant check failed for proxy trap", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_PromiseSelfResolution, 5613, "", "Object used to resolve a promise creates a circular resolution", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_NeedPromise, 5614, "'%s' is not a promise", "Promise expected", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_This\_NeedPromise, 5615, "%s: 'this' is not a Promise object", "Promise expected", kjstTypeError, JSERR\_NeedPromise) // {Locked="\'this\'"}

RT\_ERROR\_MSG(JSERROR\_SetPrototypeOf, 5616, "Failed to set prototype", "Failed to set prototype", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_ObjectIsNotInitialized, 5617, "%s: Object internal state is not initialized", "Object internal state is not initialized", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_GeneratorAlreadyExecuting, 5618, "%s: Cannot execute generator function because it is currently executing", "", kjstTypeError, 0)

// SIMD\_JS

RT\_ERROR\_MSG(JSERR\_This\_NeedSimd, 5620, "%s: 'this' is not a SIMD type", "SIMD type expected", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_SimdFloat32x4TypeMismatch, 5621, "SIMD.Float32x4.%s: Invalid SIMD types for operation", "Expecting Float32x4 values", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_SimdFloat64x2TypeMismatch, 5622, "SIMD.Float64x2.%s: Invalid SIMD types for operation", "Expecting Float64x2 values", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_SimdInt32x4TypeMismatch, 5623, "SIMD.Int32x4.%s: Invalid SIMD types for operation", "Expecting Int32x4 values", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_SimdInt8x16TypeMismatch, 5624, "SIMD.Int8x16.%s: Invalid SIMD types for operation", "Expecting Int8x16 values", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_SimdLaneRangeError, 5625, "Lane index is out of range for this SIMD type", "Lane index is out of range", kjstRangeError, 0)

RT\_ERROR\_MSG(JSERR\_SimdInvalidArgType, 5626, "%s: Invalid argument type for SIMD operation", "Invalid argument type for SIMD operation", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_NeedConstructor, 5627, "'%s' is not a constructor", "Constructor expected", kjstTypeError, 0)

RT\_ERROR\_MSG(VBSERR\_CantDisplayDate, 32812, "", "The specified date is not available in the current locale's calendar", kjstRangeError, 0)

RT\_ERROR\_MSG(JSERR\_ClassThisAlreadyAssigned, 5628, "", "Multiple calls to 'super' in a class constructor are not allowed", kjstReferenceError, 0)

RT\_ERROR\_MSG(JSERR\_ClassSuperInBaseClass, 5629, "", "Unexpected call to 'super' in a base class constructor", kjstReferenceError, 0)

RT\_ERROR\_MSG(JSERR\_ClassDerivedConstructorInvalidReturnType, 5630, "", "Derived class constructor can return only object or undefined", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_ClassStaticMethodCannotBePrototype, 5631, "", "Class static member cannot be named 'prototype'", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_ClassConstructorCannotBeCalledWithoutNew, 5632, "%s: cannot be called without the new keyword", "Class constructor cannot be called without the new keyword", kjstTypeError, 0)

RT\_ERROR\_MSG(JSERR\_FunctionArgument\_FirstCannotBeRegExp, 5633, "%s: first argument cannot be a RegExp", "First argument cannot be a RegExp", kjstTypeError, 0)

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

// 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

// NOTE: all resource IDs should be < RTERROR\_STRINGFORMAT\_OFFSET (up to 10,000 resource ids)

#define RTERROR\_STRINGFORMAT\_OFFSET 10000

#define RTERROR\_PUBLIC\_RESOURCEOFFSET 20000

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

// 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>;

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

// 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"

#pragma hdrstop

#include "errstr.h"

void CopyException (EXCEPINFO \*peiDest, const EXCEPINFO \*peiSource)

{

FreeExcepInfo(peiDest);

\*peiDest = \*peiSource;

if (peiSource->bstrSource) {

peiDest->bstrSource =

SysAllocStringLen(peiSource->bstrSource, SysStringLen(peiSource->bstrSource));

}

if (peiSource->bstrDescription) {

peiDest->bstrDescription =

SysAllocStringLen(peiSource->bstrDescription, SysStringLen(peiSource->bstrDescription));

}

if (peiSource->bstrHelpFile) {

peiDest->bstrHelpFile =

SysAllocStringLen(peiSource->bstrHelpFile, SysStringLen(peiSource->bstrHelpFile));

}

}

/\*\*\*

\*BOOL FSupportsErrorInfo

\*Purpose:

\* Answers if the given object supports the Rich Error mechanism

\* on the given interface.

\*

\*Entry:

\* punk = the object

\* riid = the IID of the interface on the object

\*

\*Exit:

\* return value = BOOL

\*

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

BOOL FSupportsErrorInfo(IUnknown \*punk, REFIID riid)

{

BOOL fSupports;

ISupportErrorInfo \*psupport;

fSupports = FALSE;

if(SUCCEEDED(punk->QueryInterface(\_\_uuidof(ISupportErrorInfo), (void \*\*)&psupport)))

{

if(NOERROR == psupport->InterfaceSupportsErrorInfo(riid))

fSupports = TRUE;

psupport->Release();

}

return fSupports;

}

/\*\*\*

\*PUBLIC HRESULT GetErrorInfo

\*Purpose:

\* Filling the given EXCEPINFO structure from the contents of

\* the current OLE error object (if any).

\*

\*Entry:

\* pexcepinfo = pointer to caller allocated EXCEPINFO to fillin.

\*

\*Exit:

\* return value = HRESULT. S\_OK if obtained info, else S\_FALSE

\*

\*Note:

\* This routine assumes that the given EXCEPINFO does \*not\* contain

\* any strings that need to be freed before its contents are set.

\*

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

HRESULT GetErrorInfo(EXCEPINFO \*pexcepinfo)

{

HRESULT hr;

memset(pexcepinfo, 0, sizeof(\*pexcepinfo));

IErrorInfo \*perrinfo;

// GetErrorInfo returns S\_FALSE if there is no rich error info

// and S\_OK if there is.

if(NOERROR == (hr = GetErrorInfo(0L, &perrinfo)))

{

perrinfo->GetSource(&pexcepinfo->bstrSource);

perrinfo->GetDescription(&pexcepinfo->bstrDescription);

perrinfo->GetHelpFile(&pexcepinfo->bstrHelpFile);

perrinfo->GetHelpContext(&pexcepinfo->dwHelpContext);

perrinfo->Release();

}

return hr;

}

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

HRESULT mapping

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

template <rtErrors errnum> class ErrorTypeMap;

#define RT\_ERROR\_MSG(name, errnum, str1, str2, jst, errorNumSource) \

template <> class ErrorTypeMap<name> \

{ \

public: \

static const ErrorTypeEnum Type = jst; \

};

#define RT\_PUBLICERROR\_MSG(name, errnum, str1, str2, jst, errorNumSource) RT\_ERROR\_MSG(name, errnum, str1, str2, jst, errorNumSource)

#include "rterrors.h"

#undef RT\_PUBLICERROR\_MSG

#undef RT\_ERROR\_MSG

struct MHR

{

HRESULT hrIn;

HRESULT hrOut;

ErrorTypeEnum errorType;

};

// This table maps OLE errors to JScript errors. The comment on each line

// shows the numeric value. The table must be sorted so we can do a binary

// search on it.

#define MAPHR(in, out) { HR(in), out, ErrorTypeMap<out>::Type }

const MHR g\_rgmhr[] =

{

// FACILITY\_NULL errors

#if \_WIN32 || \_WIN64

/\*0x80004001\*/ MAPHR(E\_NOTIMPL, VBSERR\_ActionNotSupported),

/\*0x80004002\*/ MAPHR(E\_NOINTERFACE, VBSERR\_OLENotSupported),

#else

#error Neither \_\_WIN32, nor \_WIN64 is defined

#endif

// FACILITY\_DISPATCH - IDispatch errors.

/\*0x80020001\*/ MAPHR(DISP\_E\_UNKNOWNINTERFACE, VBSERR\_OLENoPropOrMethod),

/\*0x80020003\*/ MAPHR(DISP\_E\_MEMBERNOTFOUND, VBSERR\_OLENoPropOrMethod),

/\*0x80020004\*/ MAPHR(DISP\_E\_PARAMNOTFOUND, VBSERR\_NamedParamNotFound),

/\*0x80020005\*/ MAPHR(DISP\_E\_TYPEMISMATCH, VBSERR\_TypeMismatch),

/\*0x80020006\*/ MAPHR(DISP\_E\_UNKNOWNNAME, VBSERR\_OLENoPropOrMethod),

/\*0x80020007\*/ MAPHR(DISP\_E\_NONAMEDARGS, VBSERR\_NamedArgsNotSupported),

/\*0x80020008\*/ MAPHR(DISP\_E\_BADVARTYPE, VBSERR\_InvalidTypeLibVariable),

/\*0x8002000A\*/ MAPHR(DISP\_E\_OVERFLOW, VBSERR\_Overflow),

/\*0x8002000B\*/ MAPHR(DISP\_E\_BADINDEX, VBSERR\_OutOfBounds),

/\*0x8002000C\*/ MAPHR(DISP\_E\_UNKNOWNLCID, VBSERR\_LocaleSettingNotSupported),

/\*0x8002000D\*/ MAPHR(DISP\_E\_ARRAYISLOCKED, VBSERR\_ArrayLocked),

/\*0x8002000E\*/ MAPHR(DISP\_E\_BADPARAMCOUNT, VBSERR\_FuncArityMismatch),

/\*0x8002000F\*/ MAPHR(DISP\_E\_PARAMNOTOPTIONAL, VBSERR\_ParameterNotOptional),

/\*0x80020011\*/ MAPHR(DISP\_E\_NOTACOLLECTION, VBSERR\_NotEnum),

// FACILITY\_DISPATCH - Typelib errors.

/\*0x8002802F\*/ MAPHR(TYPE\_E\_DLLFUNCTIONNOTFOUND, VBSERR\_InvalidDllFunctionName),

/\*0x80028CA0\*/ MAPHR(TYPE\_E\_TYPEMISMATCH, VBSERR\_TypeMismatch),

/\*0x80028CA1\*/ MAPHR(TYPE\_E\_OUTOFBOUNDS, VBSERR\_OutOfBounds),

/\*0x80028CA2\*/ MAPHR(TYPE\_E\_IOERROR, VBSERR\_IOError),

/\*0x80028CA3\*/ MAPHR(TYPE\_E\_CANTCREATETMPFILE, VBSERR\_CantCreateTmpFile),

/\*0x80029C4A\*/ MAPHR(TYPE\_E\_CANTLOADLIBRARY, VBSERR\_DLLLoadErr),

// FACILITY\_STORAGE errors

/\*0x80030002\*/ MAPHR(STG\_E\_FILENOTFOUND, VBSERR\_OLEFileNotFound),

/\*0x80030003\*/ MAPHR(STG\_E\_PATHNOTFOUND, VBSERR\_PathNotFound),

/\*0x80030004\*/ MAPHR(STG\_E\_TOOMANYOPENFILES, VBSERR\_TooManyFiles),

/\*0x80030005\*/ MAPHR(STG\_E\_ACCESSDENIED, VBSERR\_PermissionDenied),

/\*0x80030008\*/ MAPHR(STG\_E\_INSUFFICIENTMEMORY, VBSERR\_OutOfMemory),

/\*0x80030012\*/ MAPHR(STG\_E\_NOMOREFILES, VBSERR\_TooManyFiles),

/\*0x80030013\*/ MAPHR(STG\_E\_DISKISWRITEPROTECTED, VBSERR\_PermissionDenied),

/\*0x8003001D\*/ MAPHR(STG\_E\_WRITEFAULT, VBSERR\_IOError),

/\*0x8003001E\*/ MAPHR(STG\_E\_READFAULT, VBSERR\_IOError),

/\*0x80030020\*/ MAPHR(STG\_E\_SHAREVIOLATION, VBSERR\_PathFileAccess),

/\*0x80030021\*/ MAPHR(STG\_E\_LOCKVIOLATION, VBSERR\_PermissionDenied),

/\*0x80030050\*/ MAPHR(STG\_E\_FILEALREADYEXISTS, VBSERR\_FileAlreadyExists),

/\*0x80030070\*/ MAPHR(STG\_E\_MEDIUMFULL, VBSERR\_DiskFull),

/\*0x800300FC\*/ MAPHR(STG\_E\_INVALIDNAME, VBSERR\_FileNotFound),

/\*0x80030100\*/ MAPHR(STG\_E\_INUSE, VBSERR\_PermissionDenied),

/\*0x80030101\*/ MAPHR(STG\_E\_NOTCURRENT, VBSERR\_PermissionDenied),

/\*0x80030103\*/ MAPHR(STG\_E\_CANTSAVE, VBSERR\_IOError),

// FACILITY\_ITF errors.

/\*0x80040154\*/ MAPHR(REGDB\_E\_CLASSNOTREG, VBSERR\_CantCreateObject),

/\*0x800401E3\*/ MAPHR(MK\_E\_UNAVAILABLE, VBSERR\_CantCreateObject),

/\*0x800401E6\*/ MAPHR(MK\_E\_INVALIDEXTENSION, VBSERR\_OLEFileNotFound),

/\*0x800401EA\*/ MAPHR(MK\_E\_CANTOPENFILE, VBSERR\_OLEFileNotFound),

/\*0x800401F3\*/ MAPHR(CO\_E\_CLASSSTRING, VBSERR\_CantCreateObject),

/\*0x800401F5\*/ MAPHR(CO\_E\_APPNOTFOUND, VBSERR\_CantCreateObject),

/\*0x800401FE\*/ MAPHR(CO\_E\_APPDIDNTREG, VBSERR\_CantCreateObject),

#if \_WIN32 || \_WIN64

// FACILITY\_WIN32 errors

/\*0x80070005\*/ MAPHR(E\_ACCESSDENIED, VBSERR\_PermissionDenied),

/\*0x8007000E\*/ MAPHR(E\_OUTOFMEMORY, VBSERR\_OutOfMemory),

/\*0x80070057\*/ MAPHR(E\_INVALIDARG, VBSERR\_IllegalFuncCall),

/\*0x800706BA\*/ MAPHR(\_HRESULT\_TYPEDEF\_(0x800706BA), VBSERR\_ServerNotFound),

// FACILITY\_WINDOWS

/\*0x80080005\*/ MAPHR(CO\_E\_SERVER\_EXEC\_FAILURE, VBSERR\_CantCreateObject),

#endif // \_WIN32 || \_WIN64

};

const long kcmhr = sizeof(g\_rgmhr) / sizeof(g\_rgmhr[0]);

HRESULT MapHr(HRESULT hr, ErrorTypeEnum \* errorTypeOut)

{

int imhrMin, imhrLim, imhr;

#if DEBUG

// In debug, check that all the entries in the error map table are

// sorted based on the HRESULT in ascending order. We will then binary

// search the sorted array. We need do this only once per invocation.

static BOOL fCheckSort = TRUE;

if (fCheckSort)

{

fCheckSort = FALSE;

for (imhr = 1; imhr < kcmhr; imhr++)

Assert((ulong)g\_rgmhr[imhr - 1].hrIn < (ulong)g\_rgmhr[imhr].hrIn);

}

#endif // DEBUG

if (errorTypeOut != nullptr)

{

\*errorTypeOut = kjstError;

}

if (SUCCEEDED(hr))

return NOERROR;

if (FACILITY\_CONTROL == HRESULT\_FACILITY(hr))

return hr;

for (imhrMin = 0, imhrLim = kcmhr; imhrMin < imhrLim; )

{

imhr = (imhrMin + imhrLim) / 2;

if ((ulong)g\_rgmhr[imhr].hrIn < (ulong)hr)

imhrMin = imhr + 1;

else

imhrLim = imhr;

}

if (imhrMin < kcmhr && hr == g\_rgmhr[imhrMin].hrIn)

{

if (errorTypeOut != nullptr)

{

\*errorTypeOut = g\_rgmhr[imhrMin].errorType;

}

return g\_rgmhr[imhrMin].hrOut;

}

return hr;

}

// === ScriptException ===

ScriptException::~ScriptException(void)

{

FreeExcepInfo(&ei);

}

void ScriptException::CopyInto(ScriptException \*pse)

{

pse->ichMin = ichMin;

pse->ichLim = ichLim;

CopyException(&(pse->ei), &ei);

}

void ScriptException::Free(void)

{

ichMin = ichLim = 0;

FreeExcepInfo(&ei);

}

void ScriptException::GetError(HRESULT \*phr, EXCEPINFO \*pei)

{

AssertMem(phr);

AssertMemN(pei);

if (HR(SCRIPT\_E\_RECORDED) == \*phr)

{

Assert(FAILED(HR(ei.scode)));

if (nullptr == pei)

\*phr = HR(ei.scode);

else

{

\*phr = HR(DISP\_E\_EXCEPTION);

js\_memcpy\_s(pei, sizeof(\*pei), &ei, sizeof(\*pei));

memset(&ei, 0, sizeof(ei));

if (nullptr != pei->pfnDeferredFillIn)

{

pei->pfnDeferredFillIn(pei);

pei->pfnDeferredFillIn = nullptr;

}

}

}

}

// === CompileScriptException ===

CompileScriptException::~CompileScriptException()

{

SysFreeString(bstrLine);

}

void CompileScriptException::Clear()

{

memset(this, 0, sizeof(\*this));

}

void CompileScriptException::Free()

{

ScriptException::Free();

line = ichMinLine = 0;

if (nullptr != bstrLine)

{

SysFreeString(bstrLine);

bstrLine = nullptr;

}

}

HRESULT CompileScriptException::ProcessError(IScanner \* pScan, HRESULT hr, ParseNode \* pnodeBase)

{

if (nullptr == this)

return hr;

// fill in the ScriptException structure

Clear();

ei.scode = GetScode(MapHr(hr));

// get the error string

if (FACILITY\_CONTROL != HRESULT\_FACILITY(ei.scode) ||

nullptr == (ei.bstrDescription =

BstrGetResourceString(HRESULT\_CODE(ei.scode))))

{

OLECHAR szT[50];

\_snwprintf\_s(szT, ARRAYSIZE(szT), ARRAYSIZE(szT)-1, OLESTR("error %d"), ei.scode);

if (nullptr == (ei.bstrDescription = SysAllocString(szT)))

ei.scode = E\_OUTOFMEMORY;

}

ei.bstrSource = BstrGetResourceString(IDS\_COMPILATION\_ERROR\_SOURCE);

if (nullptr == pnodeBase && nullptr != pScan)

{

// parsing phase - get the line number from the scanner

AssertMem(pScan);

this->hasLineNumberInfo = true;

pScan->GetErrorLineInfo(this->ichMin, this->ichLim, this->line, this->ichMinLine);

HRESULT hrSysAlloc = pScan->SysAllocErrorLine(this->ichMinLine, &this->bstrLine);

if( FAILED(hrSysAlloc) )

{

return hrSysAlloc;

}

if (ichMin < ichMinLine)

ichMin = ichMinLine;

}

else

{

// TODO: Variable length registers.

// Remove E\_FAIL once we have this feature.

// error during code gen - no line number info available

// E\_ABORT may result if compilation does stack probe while thread is in disabled state.

Assert(hr == JSERR\_AsmJsCompileError || hr == ERRnoMemory || hr == VBSERR\_OutOfStack || hr == E\_OUTOFMEMORY || hr == E\_FAIL || hr == E\_ABORT);

}

return SCRIPT\_E\_RECORDED;

}

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

// 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

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

Exception blocks

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

class ErrHandler;

struct ParseNode;

class COleScript;

interface IScanner;

inline void FreeExcepInfo(EXCEPINFO \*pei)

{

if (pei->bstrSource)

SysFreeString(pei->bstrSource);

if (pei->bstrDescription)

SysFreeString(pei->bstrDescription);

if (pei->bstrHelpFile)

SysFreeString(pei->bstrHelpFile);

memset(pei, 0, sizeof(\*pei));

}

void CopyException (EXCEPINFO \*pexcepinfoDest, const EXCEPINFO \*pexcepinfoSource);

BOOL FSupportsErrorInfo(IUnknown \*punk, REFIID riid);

HRESULT GetErrorInfo(EXCEPINFO \*pexcepinfo);

HRESULT MapHr(HRESULT hr, ErrorTypeEnum \* errorTypeOut = nullptr);

class SRCINFO;

class ActiveScriptError;

class ScriptException

{

public:

long ichMin;

long ichLim;

EXCEPINFO ei;

public:

ScriptException()

{ memset(this, 0, sizeof(\*this)); }

~ScriptException(void);

public:

void CopyInto(ScriptException \*pse);

void Free();

void GetError(HRESULT \*phr, EXCEPINFO \*pei); // Clears error.

};

class CompileScriptException : public ScriptException

{

public:

long line; // line number of error (zero based)

long ichMinLine; // starting char of the line

bool hasLineNumberInfo;

// TODO: if the line contains \0 character the substring following \0 will not be included:

BSTR bstrLine; // source line (if available)

public:

CompileScriptException(void) : ScriptException(), line(0), ichMinLine(0), hasLineNumberInfo(false),

bstrLine(nullptr)

{ }

~CompileScriptException();

public:

void Clear();

void Free();

void GetError(HRESULT \*phr, EXCEPINFO \*pei)

{

ScriptException::GetError(phr, pei);

Free();

}

HRESULT ProcessError(IScanner \* pScan, HRESULT hr, ParseNode \* pnodeBase);

friend class ActiveScriptError;

};

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

// 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"

namespace UnifiedRegex

{

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

// ASCIIChars

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

/\*

To get these two tables run:

ch.exe ascii.js

where ascii.js is:

----------------------------------------------------------------------

function echo(s) { WScript.Echo(s); }

var NumChars = 1 << 8;

var Word = 1 << 0;

var Newline = 1 << 1;

var Whitespace = 1 << 2;

var Letter = 1 << 3;

var Digit = 1 << 4;

var Octal = 1 << 5;

var Hex = 1 << 6;

var classes = [];

var values = [];

function cc(s) {

return s.charCodeAt(0);

}

var c;

for (c = 0; c < NumChars; c++)

{

classes[c] = 0;

values[c] = 0;

}

for (c = cc('0'); c <= cc('7'); c++)

{

classes[c] |= Word | Octal | Digit | Hex;

values[c] = c - cc('0');

}

for (c = cc('8'); c <= cc('9'); c++)

{

classes[c] |= Word | Digit | Hex;

values[c] = c - cc('0');

}

for (c = cc('a'); c <= cc('f'); c++)

{

classes[c] |= Word | Hex | Letter;

values[c] = 10 + c - cc('a');

}

for (c = cc('g'); c <= cc('z'); c++)

classes[c] |= Word | Letter;

for (c = cc('A'); c <= cc('F'); c++)

{

classes[c] |= Word | Hex | Letter;

values[c] = 10 + c - cc('A');

}

for (c = cc('G'); c <= cc('Z'); c++)

classes[c] |= Word | Letter;

classes[cc('\_')] |= Word;

classes[cc('\n')] |= Newline;

classes[cc('\r')] |= Newline;

for (c = cc('\t'); c <= cc('\r'); c++)

classes[c] |= Whitespace;

classes[cc(' ')] |= Whitespace;

classes[cc('\x85')] |= Whitespace;

classes[cc('\xa0')] |= Whitespace;

hex = "0123456789abcdef";

function toHex(n) {

return "0x" + hex[n >> 4] + hex[n & 0xf];

}

function dump(a) {

for (c = 0; c < NumChars; c++) {

if (c % 16 == 0)

str = " ";

else

str += ", ";

str += toHex(a[c]);

if (c % 16 == 15)

{

if (c < NumChars - 1)

str += ",";

echo(str);

}

}

}

echo(" const uint8 ASCIIChars::classes[] = {");

dump(classes);

echo(" };");

echo(" const uint8 ASCIIChars::values[] = {");

dump(values);

echo(" };");

----------------------------------------------------------------------

\*/

const uint8 ASCIIChars::classes[] = {

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x04, 0x06, 0x04, 0x04, 0x06, 0x00, 0x00,

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x04, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x71, 0x71, 0x71, 0x71, 0x71, 0x71, 0x71, 0x71, 0x51, 0x51, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x49, 0x49, 0x49, 0x49, 0x49, 0x49, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09,

0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x00, 0x00, 0x00, 0x00, 0x01,

0x00, 0x49, 0x49, 0x49, 0x49, 0x49, 0x49, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09,

0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x09, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x04, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00

};

const uint8 ASCIIChars::values[] = {

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00

};

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

// TrivialCaseMapper

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

const TrivialCaseMapper TrivialCaseMapper::Instance;

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

// StandardChars<wchar\_t>

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

/\*

To get the whitespaces string, run:

gawk -f spaces.gawk http://www.unicode.org/Public/UNIDATA/UnicodeData.txt

where spaces.gawk is

----------------------------------------------------------------------

BEGIN {

FS = ";";

start = -1;

last = -1;

str = "";

}

{

code = strtonum("0x" $1);

if ($3 == "Zs" || code == 0x0009 || code == 0x000B || code == 0x000C || code == 0x0020 || code == 0x00A0 || code == 0xFEFF || code == 0x000A || code == 0x000D || code == 0x2028 || code == 0x2029)

{

if (start < 0)

start = code;

else if (code > last + 1) {

str = sprintf("%s\\x%04x\\x%04x", str, start, last);

start = code;

}

last = code;

}

}

END {

str = sprintf("%s\\x%04x\\x%04x", str, start, last);

print str;

}----------------------------------------------------------------------

\*/

const int StandardChars<wchar\_t>::numDigitPairs = 1;

const wchar\_t\* const StandardChars<wchar\_t>::digitStr = L"09";

const int StandardChars<wchar\_t>::numWhitespacePairs = 11;

const wchar\_t\* const StandardChars<wchar\_t>::whitespaceStr = L"\x0009\x000d\x0020\x0020\x00a0\x00a0\x1680\x1680\x180e\x180e\x2000\x200a\x2028\x2029\x202f\x202f\x205f\x205f\x3000\x3000\xfeff\xfeff";

const int StandardChars<wchar\_t>::numWordPairs = 4;

const wchar\_t\* const StandardChars<wchar\_t>::wordStr = L"09AZ\_\_az";

const int StandardChars<wchar\_t>::numNewlinePairs = 3;

const wchar\_t\* const StandardChars<wchar\_t>::newlineStr = L"\x000a\x000a\x000d\x000d\x2028\x2029";

StandardChars<wchar\_t>::StandardChars(ArenaAllocator\* allocator)

: allocator(allocator)

, unicodeDataCaseMapper(allocator, CaseInsensitive::MappingSource::UnicodeData, &TrivialCaseMapper::Instance)

, caseFoldingCaseMapper(allocator, CaseInsensitive::MappingSource::CaseFolding, &unicodeDataCaseMapper)

, fullSet(0)

, emptySet(0)

, wordSet(0)

, nonWordSet(0)

, newlineSet(0)

, whitespaceSet(0)

{

}

void StandardChars<wchar\_t>::SetDigits(ArenaAllocator\* setAllocator, CharSet<Char> &set)

{

set.SetRanges(setAllocator, numDigitPairs, digitStr);

}

void StandardChars<wchar\_t>::SetNonDigits(ArenaAllocator\* setAllocator, CharSet<Char> &set)

{

set.SetNotRanges(setAllocator, numDigitPairs, digitStr);

}

void StandardChars<wchar\_t>::SetWhitespace(ArenaAllocator\* setAllocator, CharSet<Char> &set)

{

set.SetRanges(setAllocator, numWhitespacePairs, whitespaceStr);

}

void StandardChars<wchar\_t>::SetNonWhitespace(ArenaAllocator\* setAllocator, CharSet<Char> &set)

{

set.SetNotRanges(setAllocator, numWhitespacePairs, whitespaceStr);

}

void StandardChars<wchar\_t>::SetWordChars(ArenaAllocator\* setAllocator, CharSet<Char> &set)

{

set.SetRanges(setAllocator, numWordPairs, wordStr);

}

void StandardChars<wchar\_t>::SetNonWordChars(ArenaAllocator\* setAllocator, CharSet<Char> &set)

{

set.SetNotRanges(setAllocator, numWordPairs, wordStr);

}

void StandardChars<wchar\_t>::SetNewline(ArenaAllocator\* setAllocator, CharSet<Char> &set)

{

set.SetRanges(setAllocator, numNewlinePairs, newlineStr);

}

void StandardChars<wchar\_t>::SetNonNewline(ArenaAllocator\* setAllocator, CharSet<Char> &set)

{

set.SetNotRanges(setAllocator, numNewlinePairs, newlineStr);

}

CharSet<wchar\_t>\* StandardChars<wchar\_t>::GetFullSet()

{

if (fullSet == 0)

{

fullSet = Anew(allocator, UnicodeCharSet);

fullSet->SetRange(allocator, MinChar, MaxChar);

}

return fullSet;

}

CharSet<wchar\_t>\* StandardChars<wchar\_t>::GetEmptySet()

{

if (emptySet == 0)

{

emptySet = Anew(allocator, UnicodeCharSet);

// leave empty

}

return emptySet;

}

CharSet<wchar\_t>\* StandardChars<wchar\_t>::GetWordSet()

{

if (wordSet == 0)

{

wordSet = Anew(allocator, UnicodeCharSet);

wordSet->SetRanges(allocator, numWordPairs, wordStr);

}

return wordSet;

}

CharSet<wchar\_t>\* StandardChars<wchar\_t>::GetNonWordSet()

{

if (nonWordSet == 0)

{

nonWordSet = Anew(allocator, UnicodeCharSet);

nonWordSet->SetNotRanges(allocator, numWordPairs, wordStr);

}

return nonWordSet;

}

CharSet<wchar\_t>\* StandardChars<wchar\_t>::GetNewlineSet()

{

if (newlineSet == 0)

{

newlineSet = Anew(allocator, UnicodeCharSet);

newlineSet->SetRanges(allocator, numNewlinePairs, newlineStr);

}

return newlineSet;

}

CharSet<wchar\_t>\* StandardChars<wchar\_t>::GetWhitespaceSet()

{

if (whitespaceSet == 0)

{

whitespaceSet = Anew(allocator, UnicodeCharSet);

whitespaceSet->SetRanges(allocator, numWhitespacePairs, whitespaceStr);

}

return whitespaceSet;

}

CharSet<wchar\_t>\* StandardChars<wchar\_t>::GetSurrogateUpperRange()

{

if (surrogateUpperRange == 0)

{

surrogateUpperRange = Anew(allocator, UnicodeCharSet);

surrogateUpperRange->SetRange(allocator, (wchar\_t)0xDC00u, (wchar\_t)0xDFFFu);

}

return surrogateUpperRange;

}

}

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

// 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 UnifiedRegex

{

template <typename C>

class StandardChars {};

class ASCIIChars : public Chars<char>

{

private:

enum CharClass : uint8

{

Word = 1 << 0,

Newline = 1 << 1,

Whitespace = 1 << 2,

Letter = 1 << 3,

Digit = 1 << 4,

Octal = 1 << 5,

Hex = 1 << 6

};

static const uint8 classes[NumChars];

static const uint8 values[NumChars];

public:

\_\_inline static bool IsWord(Char c)

{

return (classes[CTU(c)] & Word) != 0;

}

\_\_inline static bool IsNewline(Char c)

{

return (classes[CTU(c)] & Newline) != 0;

}

\_\_inline static bool IsWhitespace(Char c)

{

return (classes[CTU(c)] & Whitespace) != 0;

}

\_\_inline static bool IsLetter(Char c)

{

return (classes[CTU(c)] & Letter) != 0;

}

\_\_inline static bool IsDigit(Char c)

{

return (classes[CTU(c)] & Digit) != 0;

}

\_\_inline static bool IsOctal(Char c)

{

return (classes[CTU(c)] & Octal) != 0;

}

\_\_inline static bool IsHex(Char c)

{

return (classes[CTU(c)] & Hex) != 0;

}

\_\_inline static uint DigitValue(Char c)

{

return values[CTU(c)];

}

};

template <>

class StandardChars<uint8> : Chars<uint8>

{

public:

inline StandardChars(ArenaAllocator\* allocator) {}

\_\_inline bool IsWord(Char c) const

{

return ASCIIChars::IsWord(ASCIIChars::UTC(CTU(c)));

}

\_\_inline bool IsNewline(Char c) const

{

return ASCIIChars::IsNewline(ASCIIChars::UTC(CTU(c)));

}

\_\_inline bool IsWhitespaceOrNewline(Char c) const

{

return ASCIIChars::IsWhitespace(ASCIIChars::UTC(CTU(c)));

}

\_\_inline bool IsLetter(Char c) const

{

return ASCIIChars::IsLetter(ASCIIChars::UTC(CTU(c)));

}

\_\_inline bool IsDigit(Char c) const

{

return ASCIIChars::IsDigit(ASCIIChars::UTC(CTU(c)));

}

\_\_inline bool IsOctal(Char c) const

{

return ASCIIChars::IsOctal(ASCIIChars::UTC(CTU(c)));

}

\_\_inline bool IsHex(Char c) const

{

return ASCIIChars::IsHex(ASCIIChars::UTC(CTU(c)));

}

\_\_inline uint DigitValue(Char c) const

{

return ASCIIChars::DigitValue(ASCIIChars::UTC(CTU(c)));

}

};

template <typename FallbackCaseMapper>

class CaseMapper

{

public:

CaseMapper(ArenaAllocator \*allocator, CaseInsensitive::MappingSource mappingSource, const FallbackCaseMapper \*fallbackMapper) :

toEquivs((uint64) -1),

fallbackMapper(fallbackMapper)

{

CompileAssert(sizeof(wchar\_t) == 2);

CompileAssert(sizeof(uint) > sizeof(wchar\_t));

const uint maxUChar = Chars<wchar\_t>::MaxUChar;

uint l = 0;

uint h = maxUChar;

uint tblidx = 0;

do {

uint acth;

wchar\_t equivl[CaseInsensitive::EquivClassSize];

bool isNonTrivial = CaseInsensitive::RangeToEquivClassOnlyInSource(mappingSource, tblidx, l, h, acth, equivl);

if (isNonTrivial)

{

\_\_assume(acth <= maxUChar); // property of algorithm: acth never greater than h

do

{

uint64 r = 0;

CompileAssert(sizeof(r) >= sizeof(wchar\_t) \* CaseInsensitive::EquivClassSize);

for (int i = CaseInsensitive::EquivClassSize - 1; i >= 0; i--)

{

\_\_assume(equivl[i] <= maxUChar); // property of algorithm: never map outside of range

r <<= 16;

r |= Chars<wchar\_t>::CTU(equivl[i]++);

}

toEquivs.Set(allocator, Chars<wchar\_t>::UTC(l++), r);

}

while (l <= acth);

}

else

{

l = acth + 1;

}

}

while (l <= h);

}

inline wchar\_t ToCanonical(wchar\_t c) const

{

uint64 r = toEquivs.Get(c);

return r == EQUIV\_MISSING ? fallbackMapper->ToCanonical(c) : Chars<wchar\_t>::UTC(r & 0xffff);

}

CompileAssert(CaseInsensitive::EquivClassSize == 4);

inline bool ToEquivs(wchar\_t c, \_\_out\_ecount(4) wchar\_t\* equivs) const

{

uint64 r = toEquivs.Get(c);

if (r == EQUIV\_MISSING)

{

return fallbackMapper->ToEquivs(c, equivs);

}

else

{

for (int i = 0; i < CaseInsensitive::EquivClassSize; i++)

{

equivs[i] = Chars<wchar\_t>::UTC(r & 0xffff);

r >>= 16;

}

return true;

}

}

inline bool IsTrivialString(const wchar\_t\* str, CharCount strLen) const

{

for (CharCount i = 0; i < strLen; i++)

{

if (toEquivs.Get(str[i]) != EQUIV\_MISSING)

return false;

}

return fallbackMapper->IsTrivialString(str, strLen);

}

private:

// Map character to:

// - -1 if trivial equivalence class

// - otherwise to four 16-bit fields: <equiv 4><equiv 3><equiv 2><equiv 1>

const static uint64 EQUIV\_MISSING = static\_cast<uint64>(-1);

CharMap<wchar\_t, uint64> toEquivs;

const FallbackCaseMapper \*fallbackMapper;

};

class TrivialCaseMapper

{

public:

inline wchar\_t ToCanonical(wchar\_t c) const

{

return c;

}

CompileAssert(CaseInsensitive::EquivClassSize == 4);

inline bool ToEquivs(wchar\_t c, \_\_out\_ecount(4) wchar\_t\* equivs) const

{

for (int i = 0; i < CaseInsensitive::EquivClassSize; i++)

equivs[i] = c;

return false;

}

inline bool IsTrivialString(const wchar\_t\* str, CharCount strLen) const

{

return true;

}

static const TrivialCaseMapper Instance;

};

template <>

class StandardChars<wchar\_t> : public Chars<wchar\_t>

{

private:

static const int numDigitPairs;

static const Char\* const digitStr;

static const int numWhitespacePairs;

static const Char\* const whitespaceStr;

static const int numWordPairs;

static const Char\* const wordStr;

static const int numNewlinePairs;

static const Char\* const newlineStr;

ArenaAllocator\* allocator;

typedef CaseMapper<TrivialCaseMapper> UnicodeDataCaseMapper;

const UnicodeDataCaseMapper unicodeDataCaseMapper;

typedef CaseMapper<UnicodeDataCaseMapper> CaseFoldingCaseMapper;

const CaseFoldingCaseMapper caseFoldingCaseMapper;

CharSet<Char>\* fullSet;

CharSet<Char>\* emptySet;

CharSet<Char>\* wordSet;

CharSet<Char>\* nonWordSet;

CharSet<Char>\* newlineSet;

CharSet<Char>\* whitespaceSet;

CharSet<Char>\* surrogateUpperRange;

public:

StandardChars(ArenaAllocator\* allocator);

\_\_inline bool IsWord(Char c) const

{

return CTU(c) < ASCIIChars::NumChars && ASCIIChars::IsWord(ASCIIChars::UTC(CTU(c)));

}

\_\_inline bool IsNewline(Char c) const

{

return CTU(c) < ASCIIChars::NumChars ? ASCIIChars::IsNewline(ASCIIChars::UTC(CTU(c))) : (CTU(c) & 0xfffe) == 0x2028;

}

\_\_inline bool IsWhitespaceOrNewline(Char c) const

{

if (CTU(c) < ASCIIChars::NumChars)

return ASCIIChars::IsWhitespace(ASCIIChars::UTC(CTU(c)));

else

return CTU(c) == 0x1680 || CTU(c) == 0x180e || (CTU(c) >= 0x2000 && CTU(c) <= 0x200a) ||

CTU(c) == 0x2028 || CTU(c) == 0x2029 || CTU(c) == 0x202f || CTU(c) == 0x205f ||

CTU(c) == 0x3000 || CTU(c) == 0xfeff;

}

\_\_inline bool IsLetter(Char c) const

{

return CTU(c) < ASCIIChars::NumChars && ASCIIChars::IsLetter(ASCIIChars::UTC(CTU(c)));

}

\_\_inline bool IsDigit(Char c) const

{

return CTU(c) < ASCIIChars::NumChars && ASCIIChars::IsDigit(ASCIIChars::UTC(CTU(c)));

}

\_\_inline bool IsOctal(Char c) const

{

return CTU(c) < ASCIIChars::NumChars && ASCIIChars::IsOctal(ASCIIChars::UTC(CTU(c)));

}

\_\_inline bool IsHex(Char c) const

{

return CTU(c) < ASCIIChars::NumChars && ASCIIChars::IsHex(ASCIIChars::UTC(CTU(c)));

}

\_\_inline uint DigitValue(Char c) const

{

return CTU(c) < ASCIIChars::NumChars ? ASCIIChars::DigitValue(ASCIIChars::UTC(CTU(c))) : 0;

}

void SetDigits(ArenaAllocator\* setAllocator, CharSet<Char> &set);

void SetNonDigits(ArenaAllocator\* setAllocator, CharSet<Char> &set);

void SetWhitespace(ArenaAllocator\* setAllocator, CharSet<Char> &set);

void SetNonWhitespace(ArenaAllocator\* setAllocator, CharSet<Char> &set);

void SetWordChars(ArenaAllocator\* setAllocator, CharSet<Char> &set);

void SetNonWordChars(ArenaAllocator\* setAllocator, CharSet<Char> &set);

void SetNewline(ArenaAllocator\* setAllocator, CharSet<Char> &set);

void SetNonNewline(ArenaAllocator\* setAllocator, CharSet<Char> &set);

CharSet<Char>\* GetFullSet();

CharSet<Char>\* GetEmptySet();

CharSet<Char>\* GetWordSet();

CharSet<Char>\* GetNonWordSet();

CharSet<Char>\* GetNewlineSet();

CharSet<Char>\* GetWhitespaceSet();

CharSet<Char>\* GetSurrogateUpperRange();

inline Char ToCanonical(CaseInsensitive::MappingSource mappingSource, Char c) const

{

if (mappingSource == CaseInsensitive::MappingSource::UnicodeData)

{

return unicodeDataCaseMapper.ToCanonical(c);

}

else

{

Assert(mappingSource == CaseInsensitive::MappingSource::CaseFolding);

return caseFoldingCaseMapper.ToCanonical(c);

}

}

CompileAssert(CaseInsensitive::EquivClassSize == 4);

inline bool ToEquivs(CaseInsensitive::MappingSource mappingSource, Char c, \_\_out\_ecount(4) Char\* equivs) const

{

if (mappingSource == CaseInsensitive::MappingSource::UnicodeData)

{

return unicodeDataCaseMapper.ToEquivs(c, equivs);

}

else

{

Assert(mappingSource == CaseInsensitive::MappingSource::CaseFolding);

return caseFoldingCaseMapper.ToEquivs(c, equivs);

}

}

inline bool IsTrivialString(CaseInsensitive::MappingSource mappingSource, const Char\* str, CharCount strLen) const

{

if (mappingSource == CaseInsensitive::MappingSource::UnicodeData)

{

return unicodeDataCaseMapper.IsTrivialString(str, strLen);

}

else

{

Assert(mappingSource == CaseInsensitive::MappingSource::CaseFolding);

return caseFoldingCaseMapper.IsTrivialString(str, strLen);

}

}

};

typedef UnifiedRegex::StandardChars<uint8> UTF8StandardChars;

typedef UnifiedRegex::StandardChars<wchar\_t> UnicodeStandardChars;

}

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

// 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"

namespace UnifiedRegex

{

template <typename C>

void TextbookBoyerMooreSetup<C>::Init()

{

Assert(patLen > 0);

for (uint i = 0; i < MaxCharMapLinearChars; i++)

{

lastOcc[i] = -1;

}

numLinearChars = 1;

// Always put the last character in the first index

linearChar[0] = pat[patLen - 1];

for (CharCount i = 0; i < patLen; i++)

{

if (numLinearChars <= MaxCharMapLinearChars)

{

uint j = 0;

for (; j < numLinearChars; j++)

{

if (linearChar[j] == pat[i])

{

lastOcc[j] = i;

break;

}

}

if (j == numLinearChars)

{

if (numLinearChars < MaxCharMapLinearChars)

{

linearChar[numLinearChars] = pat[i];

lastOcc[numLinearChars] = i;

}

numLinearChars++;

}

}

if (numLinearChars > MaxCharMapLinearChars)

{

break;

}

}

if (numLinearChars <= MaxCharMapLinearChars)

{

scheme = LinearScheme;

}

else

{

scheme = DefaultScheme;

}

}

template <typename C>

void TextbookBoyerMoore<C>::Setup(ArenaAllocator\* allocator, TextbookBoyerMooreSetup<C> const& info)

{

Assert(info.GetScheme() == TextbookBoyerMooreSetup<C>::DefaultScheme);

this->Setup(allocator, info.pat, info.patLen, 1);

}

template <typename C>

void TextbookBoyerMoore<C>::Setup(ArenaAllocator \* allocator, const Char \* pat, CharCount patLen, int skip)

{

// character c |-> index of last occurrence of c in pat, otherwise -1

for (CharCount i = 0; i < patLen; i++)

{

for (int j = 0; j < skip; j++)

lastOccurrence.Set(allocator, pat[i \* skip + j], i);

}

goodSuffix = TextbookBoyerMooreSetup<C>::GetGoodSuffix(allocator, pat, patLen, skip);

}

template <typename C>

int32 \* TextbookBoyerMooreSetup<C>::GetGoodSuffix(ArenaAllocator\* allocator, const Char \* pat, CharCount patLen, int skip)

{

// pat offset q |-> longest prefix of pat which is a proper suffix of pat[0..q]

// (thanks to equivalence classes being in canonical order we only need to look at the first

// character of each skip grouping in the pattern)

int32\* prefix = AnewArray(allocator, int32, patLen);

prefix[0] = 0;

int32 k = 0;

for (CharCount q = 1; q < patLen; q++)

{

while (k > 0 && pat[k \* skip] != pat[q \* skip])

k = prefix[k - 1];

if (pat[k \* skip] == pat[q \* skip])

k++;

prefix[q] = k;

}

// As above, but for rev(pat)

int32\* revPrefix = AnewArray(allocator, int32, patLen);

revPrefix[0] = 0;

k = 0;

for (CharCount q = 1; q < patLen; q++)

{

while (k > 0 && pat[(patLen - k - 1) \* skip] != pat[(patLen - q - 1) \* skip])

k = revPrefix[k - 1];

if (pat[(patLen - k - 1) \* skip] == pat[(patLen - q - 1) \* skip])

k++;

revPrefix[q] = k;

}

// pat prefix length l |-> least shift s.t. pat[0..l-1] is not mismatched

int32 \* goodSuffix = AnewArray(allocator, int32, patLen + 1);

for (CharCount j = 0; j <= patLen; j++)

goodSuffix[j] = patLen - prefix[patLen - 1];

for (CharCount l = 1; l <= patLen; l++)

{

CharCount j = patLen - revPrefix[l - 1];

int32 s = l - revPrefix[l - 1];

if (goodSuffix[j] > s)

goodSuffix[j] = s;

}

// shift above one to the left

for (CharCount j = 0; j < patLen; j++)

goodSuffix[j] = goodSuffix[j + 1];

AdeleteArray(allocator, patLen, prefix);

AdeleteArray(allocator, patLen, revPrefix);

return goodSuffix;

}

template <typename C>

void TextbookBoyerMoore<C>::FreeBody(ArenaAllocator\* allocator, CharCount patLen)

{

if(goodSuffix)

{

AdeleteArray(allocator, patLen + 1, goodSuffix);

#if DBG

goodSuffix = 0;

#endif

}

lastOccurrence.FreeBody(allocator);

}

template <uint equivClassSize, uint compareCount>

static bool MatchPatternAt(uint inputChar, wchar\_t const \* pat, CharCount index);

template <>

static bool MatchPatternAt<1, 1>(uint inputChar, wchar\_t const\* pat, CharCount index)

{

return inputChar == pat[index];

}

template <>

static bool MatchPatternAt<CaseInsensitive::EquivClassSize, CaseInsensitive::EquivClassSize>(uint inputChar, wchar\_t const \* pat, CharCount index)

{

CompileAssert(CaseInsensitive::EquivClassSize == 4);

return inputChar == pat[index \* CaseInsensitive::EquivClassSize]

|| inputChar == pat[index \* CaseInsensitive::EquivClassSize + 1]

|| inputChar == pat[index \* CaseInsensitive::EquivClassSize + 2]

|| inputChar == pat[index \* CaseInsensitive::EquivClassSize + 3];

}

template <>

static bool MatchPatternAt<CaseInsensitive::EquivClassSize, 1>(uint inputChar, wchar\_t const \* pat, CharCount index)

{

CompileAssert(CaseInsensitive::EquivClassSize == 4);

return inputChar == pat[index \* 4];

}

template <typename C>

template <uint equivClassSize, uint lastPatCharEquivClass>

bool TextbookBoyerMoore<C>::Match

( const Char \*const input

, const CharCount inputLength

, CharCount& inputOffset

, const Char\* pat

, const CharCount patLen

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, RegexStats\* stats

#endif

) const

{

Assert(input != 0);

Assert(inputOffset <= inputLength);

if (inputLength < patLen)

return false;

CharCount offset = inputOffset;

const CharCount endOffset = inputLength - (patLen - 1);

const int32\* const localGoodSuffix = goodSuffix;

const LastOccMap\* const localLastOccurrence = &lastOccurrence;

const CharCount lastPatCharIndex = (patLen - 1);

while (offset < endOffset)

{

// A separate tight loop to find the last character

while (true)

{

uint inputChar = Chars<Char>::CTU(input[offset + lastPatCharIndex]);

if (MatchPatternAt<equivClassSize, lastPatCharEquivClass>(inputChar, pat, lastPatCharIndex))

{

// Found a match. Break out of this loop and go to the match pattern loop

break;

}

// Negative case is more common,

// Write the checks so that we have a super tight loop

int lastOcc;

if (inputChar < localLastOccurrence->GetDirectMapSize())

{

if (!localLastOccurrence->IsInDirectMap(inputChar))

{

offset += patLen;

if (offset >= endOffset)

{

return false;

}

continue;

}

lastOcc = localLastOccurrence->GetDirectMap(inputChar);

}

else if (!localLastOccurrence->GetNonDirect(inputChar, lastOcc))

{

offset += patLen;

if (offset >= endOffset)

{

return false;

}

continue;

}

Assert((int)lastPatCharIndex - lastOcc >= localGoodSuffix[lastPatCharIndex]);

offset += lastPatCharIndex - lastOcc;

if (offset >= endOffset)

{

return false;

}

}

// CONSIDER: we can remove this check if we stop using TextbookBoyerMoore for one char pattern

if (lastPatCharIndex == 0)

{

inputOffset = offset;

return true;

}

// Match the rest of the pattern

int32 j = lastPatCharIndex - 1;

while (true)

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

if (stats != 0)

stats->numCompares++;

#endif

uint inputChar = Chars<Char>::CTU(input[offset + j]);

if (!MatchPatternAt<equivClassSize, equivClassSize>(inputChar, pat, j))

{

const int32 e = j - localLastOccurrence->Get((Char)inputChar);

offset += e > localGoodSuffix[j] ? e : localGoodSuffix[j];

break;

}

if (--j < 0)

{

inputOffset = offset;

return true;

}

}

}

return false;

}

// Specialized linear char map version

template <typename C>

void TextbookBoyerMooreWithLinearMap<C>::Setup(ArenaAllocator \* allocator, TextbookBoyerMooreSetup<C> const& setup)

{

Assert(setup.GetScheme() == TextbookBoyerMooreSetup<C>::LinearScheme);

lastOccurrence.Set(setup.numLinearChars, setup.linearChar, setup.lastOcc);

goodSuffix = TextbookBoyerMooreSetup<C>::GetGoodSuffix(allocator, setup.pat, setup.patLen);

}

template <typename C>

void TextbookBoyerMooreWithLinearMap<C>::FreeBody(ArenaAllocator\* allocator, CharCount patLen)

{

if(goodSuffix)

{

AdeleteArray(allocator, patLen + 1, goodSuffix);

#if DBG

goodSuffix = 0;

#endif

}

}

template <typename C>

template <uint equivClassSize>

bool TextbookBoyerMooreWithLinearMap<C>::Match

( const Char \*const input

, const CharCount inputLength

, CharCount& inputOffset

, const Char\* pat

, const CharCount patLen

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, RegexStats\* stats

#endif

) const

{

CompileAssert(equivClassSize == 1);

Assert(input != 0);

Assert(inputOffset <= inputLength);

if (inputLength < patLen)

return false;

const int32\* const localGoodSuffix = goodSuffix;

const LastOccMap\* const localLastOccurrence = &lastOccurrence;

CharCount offset = inputOffset;

const CharCount lastPatCharIndex = (patLen - 1);

const CharCount endOffset = inputLength - lastPatCharIndex;

// Using int size instead of Char value is faster

const uint lastPatChar = pat[lastPatCharIndex];

Assert(lastPatChar == localLastOccurrence->GetChar(0));

while (offset < endOffset)

{

// A separate tight loop to find the last character

while (true)

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

if (stats != 0)

stats->numCompares++;

#endif

uint inputChar = Chars<Char>::CTU(input[offset + lastPatCharIndex]);

if (inputChar == lastPatChar)

{

// Found a match. Break out of this loop and go to the match pattern loop

break;

}

// Negative case is more common,

// Write the checks so that we have a super tight loop

Assert(inputChar != localLastOccurrence->GetChar(0));

int32 lastOcc;

if (localLastOccurrence->GetChar(1) != inputChar)

{

if (localLastOccurrence->GetChar(2) != inputChar)

{

if (localLastOccurrence->GetChar(3) != inputChar)

{

offset += patLen;

if (offset >= endOffset)

{

return false;

}

continue;

}

lastOcc = localLastOccurrence->GetLastOcc(3);

}

else

{

lastOcc = localLastOccurrence->GetLastOcc(2);

}

}

else

{

lastOcc = localLastOccurrence->GetLastOcc(1);

}

Assert((int)lastPatCharIndex - lastOcc >= localGoodSuffix[lastPatCharIndex]);

offset += lastPatCharIndex - lastOcc;

if (offset >= endOffset)

{

return false;

}

}

// CONSIDER: we can remove this check if we stop using

// TextbookBoyerMoore for one char pattern

if (lastPatCharIndex == 0)

{

inputOffset = offset;

return true;

}

// Match the rest of the pattern

int32 j = lastPatCharIndex - 1;

while (true)

{

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

if (stats != 0)

stats->numCompares++;

#endif

uint inputChar = Chars<Char>::CTU(input[offset + j]);

if (inputChar != pat[j])

{

int goodSuffix = localGoodSuffix[j];

Assert(patLen <= MaxCharCount);

if (goodSuffix == (int)patLen)

{

offset += patLen;

}

else

{

const int32 e = j - localLastOccurrence->Get(inputChar);

offset += e > goodSuffix ? e : goodSuffix;

}

break;

}

if (--j < 0)

{

inputOffset = offset;

return true;

}

}

}

return false;

}

// explicit instantiation

template struct TextbookBoyerMooreSetup<wchar\_t>;

template class TextbookBoyerMoore<wchar\_t>;

template class TextbookBoyerMooreWithLinearMap<wchar\_t>;

template

bool TextbookBoyerMoore<wchar\_t>::Match<1>

( const Char \*const input

, const CharCount inputLength

, CharCount& inputOffset

, const Char\* pat

, const CharCount patLen

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, RegexStats\* stats

#endif

) const;

template

bool TextbookBoyerMoore<wchar\_t>::Match<CaseInsensitive::EquivClassSize>

( const Char \*const input

, const CharCount inputLength

, CharCount& inputOffset

, const Char\* pat

, const CharCount patLen

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, RegexStats\* stats

#endif

) const;

template

bool TextbookBoyerMoore<wchar\_t>::Match<CaseInsensitive::EquivClassSize, 1>

( const Char \*const input

, const CharCount inputLength

, CharCount& inputOffset

, const Char\* pat

, const CharCount patLen

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, RegexStats\* stats

#endif

) const;

template

bool TextbookBoyerMooreWithLinearMap<wchar\_t>::Match<1>

( const Char \*const input

, const CharCount inputLength

, CharCount& inputOffset

, const Char\* pat

, const CharCount patLen

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, RegexStats\* stats

#endif

) const;

}

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

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

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

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

// From Cormen, Leiserson and Rivest, ch 34.

#pragma once

namespace UnifiedRegex

{

template <typename C>

class TextbookBoyerMooreWithLinearMap : private Chars<C>

{

template <typename C>

friend struct TextbookBoyerMooreSetup;

private:

typedef CharMap<Char, int32, CharMapScheme\_Linear> LastOccMap;

// NOTE: We don't store the actual pattern here since it may be moved between

// constructing the scanner and running it.

LastOccMap lastOccurrence;

int32 \*goodSuffix;

public:

inline TextbookBoyerMooreWithLinearMap() : lastOccurrence(-1), goodSuffix(0) {}

// Construct Boyer-Moore tables for pattern pat:

// - pat must be of length patLen \* skip

// - if skip is > 1, then each consecutive skip characters of pattern are assumed to represent

// an equivalence class of characters in canonical order (i.e. all chars in a class are represented

// by the same sequence of skip chars)

// - otherwise this is a regular exact-match pattern

void Setup(ArenaAllocator\* allocator, TextbookBoyerMooreSetup<C> const& setup);

void FreeBody(ArenaAllocator\* allocator, CharCount patLen);

// NOTE: In the following pat and patLen must be the same as passed to Setup above

// For skip = 1

template <uint equivClassSize>

bool Match

( const Char \*const input

, const CharCount inputLength

, CharCount& inputOffset

, const Char\* pat

, const CharCount patLen

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, RegexStats\* stats

#endif

) const;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

static wchar\_t const \* GetName() { return L"linear map Boyer-Moore"; }

#endif

};

template <typename C>

class TextbookBoyerMoore : private Chars<C>

{

template <typename C>

friend struct TextbookBoyerMooreSetup;

private:

typedef CharMap<Char, int32> LastOccMap;

// NOTE: We don't store the actual pattern here since it may be moved between

// constructing the scanner and running it.

LastOccMap lastOccurrence;

int32 \*goodSuffix;

public:

inline TextbookBoyerMoore() : lastOccurrence(-1), goodSuffix(0) {}

// Construct Boyer-Moore tables for pattern pat:

// - pat must be of length patLen \* skip

// - if skip is > 1, then each consecutive skip characters of pattern are assumed to represent

// an equivalence class of characters in canonical order (i.e. all chars in a class are represented

// by the same sequence of skip chars)

// - otherwise this is a regular exact-match pattern

void Setup(ArenaAllocator\* allocator, TextbookBoyerMooreSetup<C> const& setup);

void Setup(ArenaAllocator \* allocator, const Char \* pat, CharCount patLen, int skip);

void FreeBody(ArenaAllocator\* allocator, CharCount patLen);

// NOTE: In the following pat and patLen must be the same as passed to Setup above

template <uint equivClassSize>

\_\_inline bool Match

( const Char \*const input

, const CharCount inputLength

, CharCount& inputOffset

, const Char\* pat

, const CharCount patLen

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, RegexStats\* stats

#endif

) const

{

return Match<equivClassSize, equivClassSize>(input, inputLength, inputOffset, pat, patLen

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, stats

#endif

);

}

template <uint equivClassSize, uint lastPatCharEquivClass>

bool Match

( const Char \*const input

, const CharCount inputLength

, CharCount& inputOffset

, const Char\* pat

, const CharCount patLen

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

, RegexStats\* stats

#endif

) const;

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

static wchar\_t const \* GetName() { return L"full map Boyer-Moore"; }

#endif

};

template <typename C>

struct TextbookBoyerMooreSetup : private Chars<C>

{

friend class TextbookBoyerMoore<C>;

friend class TextbookBoyerMooreWithLinearMap<C>;

enum Scheme

{

DefaultScheme,

LinearScheme

};

TextbookBoyerMooreSetup(Char const \* pat, CharCount patLen) : pat(pat), patLen(patLen) { Init(); }

Scheme GetScheme() const { return scheme; }

static int32 \* GetGoodSuffix(ArenaAllocator\* allocator, const Char \* pat, CharCount patLen, int skip = 1);

private:

void Init();

Scheme scheme;

Char const \* const pat;

CharCount const patLen;

uint numLinearChars;

Char linearChar[MaxCharMapLinearChars];

int32 lastOcc[MaxCharMapLinearChars];

};

}

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

// 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

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

\*

\* Define the token kind enum. Note that all entries from the keyword

\* table are defined first, such that the first entry has the value 0.

\*/

enum tokens

{

tkNone,

#define KEYWORD(tk,...) tk,

#define TOK\_DCL(tk,...) tk,

#include "keywords.h"

tkLimKwd,

tkLastKwd = tkLimKwd - 1,

tkEOF, // end of source code

tkIntCon, // integer literal

tkFltCon, // floating literal

tkStrCon, // string literal

tkRegExp, // regular expression literal

tkLim

};