**Bug 1263355:** Rewrite the frontend: bindings - Shu-yu Guo [:shu] <shu@mozilla.com> - 4/9/2016 [reviews] [help]

**Attachment 8780730:** Rewrite the frontend: bindings. - Shu-yu Guo [:shu] <shu@mozilla.com> - 8/12/2016 [diff] [details]

**Comment:**
With fixes for fuzz and anba tests.

Show Quick Help

**Navigation:** Overview | All Files | ~~browser_dbg_break-on-next.js~~ | ~~test_framebindings-02.js~~ | ~~test_framebindings-05.js~~ | ~~Class.h~~ | ~~GCHashTable.h~~ | ~~GCVariant.h~~ | ~~MemoryMetrics.h~~ | ~~TraceKind.h~~ | ~~TracingAPI.h~~ | ~~NamespaceImports.h~~ | ~~AsmJS.cpp~~ | ~~AsmJS.h~~ | ~~Eval.cpp~~ | ~~ModuleObject.cpp~~ | ~~ModuleObject.h~~ | ~~ReflectParse.cpp~~ | ~~RegExp.cpp~~ | ~~SelfHostingDefines.h~~ | ~~TestingFunctions.cpp~~ | ~~InlineMap.h~~ | ~~InlineTable.h~~ | ~~BytecodeCompiler.cpp~~ | ~~BytecodeCompiler.h~~ | ~~BytecodeEmitter.cpp~~ | ~~BytecodeEmitter.h~~ | ~~FoldConstants.cpp~~ | ~~FullParseHandler.h~~ | ~~NameAnalysisTypes.h~~ | ~~NameCollections.h~~ | ~~NameFunctions.cpp~~ | ~~ParseMaps-inl.h~~ | ~~ParseMaps.cpp~~ | ~~ParseMaps.h~~ | ~~ParseNode.cpp~~ | ~~ParseNode.h~~ | ~~Parser-inl.h~~ | ~~Parser.cpp~~ | ~~Parser.h~~ | ~~SharedContext.h~~ | ~~SyntaxParseHandler.h~~ | ~~Allocator.cpp~~ | ~~Barrier.cpp~~ | ~~Barrier.h~~ | ~~GCInternals.h~~ | ~~Heap.h~~ | ~~Marking.cpp~~ | ~~Marking.h~~ | ~~Nursery.cpp~~ | ~~Policy.h~~ | ~~Rooting.h~~ | ~~Statistics.cpp~~ | ~~Statistics.h~~ | ~~alias-function-closed.js~~ | ~~alias-function-not-closed.js~~ | ~~defaults-bound-to-function.js~~ | ~~defaults-call-function.js~~ | ~~defaults-destructuring-with-rest.js~~ | ~~defaults-evaluation-order.js~~ | ~~defaults-scoping.js~~ | ~~defaults-with-rest.js~~ | ~~destructuring-after-defaults.js~~ | ~~bug646968-4.js~~ | ~~bug678087.js~~ | ~~letTDZDelete.js~~ | ~~testLet.js~~ | ~~simple.js~~ | ~~Debugger-debuggees-19.js~~ | ~~Environment-getVariable-02.js~~ | ~~Environment-getVariable-05.js~~ | ~~Environment-getVariable-06.js~~ | ~~Environment-getVariable-11.js~~ | ~~Environment-getVariable-12.js~~ | ~~Environment-inspectable-01.js~~ | ~~Environment-optimizedOut-01.js~~ | ~~Environment-setVariable-01.js~~ | ~~Environment-setVariable-02.js~~ | ~~Frame-environment-02.js~~ | ~~bug-1263881-1.js~~ | ~~bug-1263881-2.js~~ | ~~bug-1263881-3.js~~ | ~~letContextualKeyword.js~~ | ~~Bailouts.cpp~~ | ~~Bailouts.h~~ | ~~BaselineBailouts.cpp~~ | ~~BaselineCompiler.cpp~~ | ~~BaselineCompiler.h~~ | ~~BaselineFrame-inl.h~~ | ~~BaselineFrame.cpp~~ | ~~BaselineFrame.h~~ | ~~BaselineFrameInfo.h~~ | ~~BaselineIC.cpp~~ | ~~BaselineIC.h~~ | ~~BaselineICList.h~~ | ~~BaselineInspector.cpp~~ | ~~BaselineInspector.h~~ | ~~BaselineJIT.cpp~~ | ~~BaselineJIT.h~~ | ~~BytecodeAnalysis.cpp~~ | ~~BytecodeAnalysis.h~~ | ~~CodeGenerator.cpp~~ | ~~CodeGenerator.h~~ | ~~CompileInfo.h~~ | ~~Ion.cpp~~ | ~~IonBuilder.cpp~~ | ~~IonBuilder.h~~ | ~~IonCaches.cpp~~ | ~~IonCaches.h~~ | ~~JitCompartment.h~~ | ~~JitFrameIterator.h~~ | ~~JitFrames.cpp~~ | ~~Lowering.cpp~~ | ~~Lowering.h~~ | ~~MIR.cpp~~ | ~~MIR.h~~ | ~~MIRGraph.cpp~~ | ~~MIRGraph.h~~ | ~~MOpcodes.h~~ | ~~MacroAssembler.cpp~~ | ~~RematerializedFrame.cpp~~ | ~~RematerializedFrame.h~~ | ~~ScalarReplacement.cpp~~ | ~~SharedIC.cpp~~ | ~~VMFunctions.cpp~~ | ~~VMFunctions.h~~ | ~~LIR-shared.h~~ | ~~LOpcodes-shared.h~~ | ~~js.msg~~ | ~~testForceLexicalInitialization.cpp~~ | ~~jsapi.cpp~~ | ~~jsapi.h~~ | ~~jscntxt.cpp~~ | ~~jscntxt.h~~ | ~~jscompartment.cpp~~ | ~~jscompartment.h~~ | ~~jsfriendapi.cpp~~ | ~~jsfriendapi.h~~ | ~~jsfun.cpp~~ | ~~jsfun.h~~ | ~~jsfuninlines.h~~ | ~~jsgc.cpp~~ | ~~jsgc.h~~ | ~~jshashutil.h~~ | ~~jsobj.cpp~~ | ~~jsobj.h~~ | ~~jsobjinlines.h~~ | ~~jsopcode.cpp~~ | ~~jsopcode.h~~ | ~~jsscript.cpp~~ | ~~jsscript.h~~ | ~~jsscriptinlines.h~~ | ~~jsstr.cpp~~ | ~~moz.build~~ | ~~run-deltablue.js~~ | ~~js.cpp~~ | ~~cross-global-implicit-this.js~~ | ~~block-scoped-functions-annex-b-eval.js~~ | ~~const-declaration-in-for-loop.js~~ | ~~for-loop.js~~ | ~~redeclaring-global-properties.js~~ | ~~for-inof-loop-const-declaration.js~~ | ~~for-loop-declaration-contains-computed-name.js~~ | ~~for-loop-declaration-contains-initializer.js~~ | ~~regress-290575.js~~ | ~~regress-367923.js~~ | ~~declarations.js~~ | ~~for-loop-destructuring.js~~ | ~~regress-610026.js~~ | ~~warning.js~~ | ~~ArgumentsObject-inl.h~~ | ~~ArgumentsObject.cpp~~ | ~~ArgumentsObject.h~~ | ~~Caches.h~~ | ~~Debugger.cpp~~ | ~~Debugger.h~~ | ~~EnvironmentObject-inl.h~~ | ~~EnvironmentObject.cpp~~ | ~~EnvironmentObject.h~~ | ~~GeneratorObject.cpp~~ | ~~GeneratorObject.h~~ | ~~GlobalObject.cpp~~ | ~~GlobalObject.h~~ | ~~HelperThreads.cpp~~ | ~~Interpreter-inl.h~~ | ~~Interpreter.cpp~~ | ~~Interpreter.h~~ | ~~MemoryMetrics.cpp~~ | ~~NativeObject.cpp~~ | ~~Opcodes.h~~ | ~~Runtime.cpp~~ | ~~Runtime.h~~ | ~~Scope.cpp~~ | ~~Scope.h~~ | ~~ScopeObject-inl.h~~ | ~~ScopeObject.cpp~~ | ~~ScopeObject.h~~ | ~~SelfHosting.cpp~~ | ~~Shape-inl.h~~ | ~~Shape.h~~ | ~~Stack-inl.h~~ | ~~Stack.cpp~~ | ~~Stack.h~~ | ~~TypeInference.cpp~~ | ~~UbiNode.cpp~~ | ~~Xdr.cpp~~ | ~~mozJSComponentLoader.cpp~~ | ~~mozJSComponentLoader.h~~ | ~~mozJSSubScriptLoader.cpp~~ | ~~XPCShellImpl.cpp~~ **Draft**

```
# HG changeset patch
# User Shu-yu Guo <shu@rfrn.org>

Bug 1263355 - Rewrite the frontend: bindings.
```

(Restored from draft; last edited 2:08:19 AM)

<Overall Comment>

**devtools/server/tests/unit/test_framebindings-02.js**

```
35    do_check_eq(vars.stopMe.value.type, "object");
36    do_check_eq(vars.stopMe.value.class, "Function");
37    do_check_true(!!vars.stopMe.value.actor);
38
```

```
39     // Skip both the global lexical scope.
```

Sentence doesn't make sense.

**devtools/server/tests/unit/test_framebindings-05.js**

```
35      do_check_eq(aResponse.ownProperties.cos.value.type,
"object");
36      do_check_eq(aResponse.ownProperties.cos.value.class,
"Function");
37 do_check_true(!!aResponse.ownProperties.cos.value.actor);
38
39      // Skip both the global lexical scope.
```

Same sentence again doesn't make sense (so I guess do a global search/replace for all of them?).

**js/public/GCVariant.h**

```
47 {
48     template <typename ConcreteVariant>
49     static void trace(JSTracer* trc, ConcreteVariant* v,
const char* name) {
50         T& thing = v->template as<T>();
51         GCPolicy<T>::trace(trc, &thing, name);
```

This shouldn't be changed the same way the other hunk did it? Seems like GCVariant<T> and GCVariant<T, U> should both trace T identically.

**js/src/asmjs/AsmJS.cpp**

```
664     MOZ_ASSERT(last->isKind(PNK_LEXICALSCOPE));
665     MOZ_ASSERT(last->isEmptyScope());
666     last = last->scopeBody();
667     MOZ_ASSERT(last->isKind(PNK_STATEMENTLIST));
668     return last;
```

Seems better to have a new variable for |last| in the assignment and afterward, with a better name, for clarity.

---

```
6704     ParseNode* switchBody = BinaryRight(switchStmt);
6705
6706     if (switchBody->isKind(PNK_LEXICALSCOPE)) {
6707         if (!switchBody->isEmptyScope())
6708             return f.fail(switchBody, "switch body may
not contain 'let' declarations");
```

Maybe s/'let'/lexical/, or 'let' or 'const' for consistency with below.

**js/src/builtin/Eval.cpp**

```
312         SourceBufferHolder::Ownership ownership =
linearChars.maybeGiveOwnershipToCaller()
                                                        ?
313 SourceBufferHolder::GiveOwnership
                                                        :
314 SourceBufferHolder::NoOwnership;
315         SourceBufferHolder srcBuf(chars, linearStr-
>length(), ownership);
316         JSScript* compiled =
frontend::CompileEvalScript(cx, cx->tempLifoAlloc(),
```

Yessss, not an option any more.

**js/src/ds/InlineTable.h**

```
4  * License, v. 2.0. If a copy of the MPL was not
distributed with this
5  * file, You can obtain one at http://mozilla.org/MPL/2.0/.
*/
6
7 #ifndef ds_InlineMap_h
8 #define ds_InlineMap_h
```

s/ds_InlineMap_h/ds_InlineTable_h/g

Does this file need |hg mv --after| treatment as well?

**js/src/frontend/BytecodeEmitter.cpp**

```
59 class LabelControl;
```

```
class LoopControl;
class TryFinallyControl;

static inline bool
```

No inline.

---

```
// A cache that tracks superfluous TDZ checks.
//
// Each basic block should have a TDZCheckCache in scope. Some NestableControl
// subclasses contain a TDZCheckCache.
class BytecodeEmitter::TDZCheckCache : public Nestable<BytecodeEmitter::TDZCheckCache>
```

As mentioned in Parser (?), privately inheriting and then using the specific desired functions out of Nestable might be better. (And I guess apply private-inheriting Nestable to every use of Nestable, while you're at it.)

---

```
template <>
bool
BytecodeEmitter::NestableControl::is<BreakableControl>() const
{
    return StatementKindIsUnlabeledBreakTarget(kind_) || kind_ == StatementKind::Label;
```

Scumbag language.

---

```
        return emittingSubroutine_;
    }
};

static inline bool
```

No inline.

---

```
MarkAllBindingsClosedOver(LexicalScope::Data& data)
{
    BindingName* names = data.names;
    for (uint32_t i = 0; i < data.length; i++)
        names[i] = BindingName(names[i].name(), true);
```

I'd prefer if BindingName::setClosedOver() existed and you used it here.

---

```
        names[i] = BindingName(names[i].name(), true);
}

// A scope that that introduces bindings.
class BytecodeEmitter::EmitterScope : public Nestable<BytecodeEmitter::EmitterScope>
```

"that that"

Given you have enclosing() shadowed by enclosing(BCE**), and enclosingInFrame() appears the preferred way to access enclosing, does Nestable need to be public here? Could it be private?

---

```
    // global scope, the NameLocation to return.
    Maybe<NameLocation> fallbackFreeNameLocation_;

    // Whether there is a corresponding EnvironmentObject on the environment
    // chain.
```

Maybe ", or whether bindings are stored in frame slots on the stack" to be clear what the alternation is.

---

```
    // The number of enclosing environments. Used for error checking.
    uint8_t environmentChainLength_;

    // The next usable slot on the frame for unaliased (i.e., not closed
```

```
    // over) bindings.
```

You went to such effort to get rid of "aliased" in favor of "closed over", so it seems very odd to mention "unaliased i.e." at all here.

---

```

    EmitterScope* enclosing(BytecodeEmitter** bce) const
{
        // There is an enclosing scope with access to
the same frame.
        if (enclosingInFrame())
            return enclosingInFrame();
```

if (EmitterScope* inFrame = enclosingInFrame())
  return inFrame;

or similar.

---

```
    }

    // The first frame slot used.
    uint32_t frameSlotStart() const {
        return enclosingInFrame() ? enclosingInFrame()-
>nextFrameSlot_ : 0;
```

if (EmitterScope* inFrame = enclosingInFrame())
  return inFrame->nextFrameSlot_;
return 0;

---

```

    NameLocation lookup(BytecodeEmitter* bce, JSAtom*
name) {
        Maybe<NameLocation> loc = lookupInCache(name);
        if (loc)
            return *loc;
```

if (Maybe<NameLocation> loc = ...)
  return *loc;

---

```

void
BytecodeEmitter::EmitterScope::dump(BytecodeEmitter*
bce)
{
    fprintf(stdout, "EmitterScope [%s] %p\n",
ScopeKindString(scope(bce)->kind()), this);
```

Hmm.  I thought dumping generally went to stderr.

---

```
{
    fprintf(stdout, "EmitterScope [%s] %p\n",
ScopeKindString(scope(bce)->kind()), this);

    for (NameLocationMap::Range r = nameCache_->all();
!r.empty(); r.popFront()) {
        NameLocation& l = r.front().value();
```

const would be nice here, if you can do it.

---

```
                "Scope notes are not needed for body-
level scopes.");
    noteIndex_ = bce->scopeNoteList.length();
    return bce->scopeNoteList.append(index(), bce-
>offset(), bce->inPrologue(),
                                     enclosingInFrame()
? enclosingInFrame()->noteIndex()
: ScopeNote::NoScopeNoteIndex);
```

Would prefer computing scope note in a way that doesn't double-call enclosingInFrame().

---

```
                if (!script->strict() && !script-
>functionHasParameterExprs()) {
                    // Check for duplicate positional
formal parameters.
                    for (BindingIter bi2(bi); bi2 &&
bi2.hasArgumentSlot(); bi2++) {
                        if (bi2.name() == name)
```

```
                        kind =
bi2.location().kind();
```

Hm, so we have this overhead even when the function has destructuring parameter that would preclude duplication. I guess we're just not going to worry about the overhead?

---

```

                    BindingLocation bindLoc =
bi.location();
                    if (bi.hasArgumentSlot() &&
                        !script->strict() &&
                        !script-
>functionHasParameterExprs())
```

And again the destructuring-case overhead.

---

```
                    BindingLocation bindLoc =
bi.location();

                    // Imports are on the environment
but are indirect
                    // bindings and must be accessed via
name instead of
                    // EnvironmentCoordinate.
```

I understand what you're getting at with "via name", but that would suggest "dynamic" to me, not a custom Import-centric mechanism. Suggest this be reworded somehow to indicate invoking that custom mechanism.

---

```
    if (!putNameInCache(bce, bi.name(), loc))
        return false;

    // There is exactly one binding in the decl env
scope.
    MOZ_ASSERT(bi.kind() ==
BindingKind::NamedLambdaCallee);
```

I'd put this above |loc|'s decl above, and I'd pass in the constant to fromBinding. Also I'd put the comment into a reason-string as second argument to MOZ_ASSERT.

---

```
        return false;

    // Resolve body-level bindings, if there are any.
    Maybe<uint32_t> lastLexicalSlot;
    if (funbox->functionScopeBindings()) {
```

Could

if (auto* bindings = funbox->functionScopeBindings)

to save some typing/reading below.

---

```
            // The only duplicate bindings that occur
are simple formal
            // parameters, in which case the last
position counts, so update the
            // location.
            if (p) {
                MOZ_ASSERT(bi.kind() ==
BindingKind::FormalParameter);
```

Should also do these, or the moral equivalent if I typo'd:

MOZ_ASSERT(!funbox->hasDestructuringArgs);
MOZ_ASSERT(!funbox->function()->hasRest());

---

```

            uint32_t atomIndex;
            if (!bce->makeAtomIndex(name, &atomIndex))
                return false;
            if (!bce->emitIndexOp(bi.bindingOp(),
```

```
atomIndex))
```

Might as well do

```
  if (!bce->emitAtomOp(name, bi.bindingOp())
    return false;
```

for concision. (I suppose your formulation skips an impossible |op == JSOP_GETPROP && atom == cx->names().length| optimization, but if that's the worry, I'd say it's better to just have callers that want the optimization -- i.e. where it might actually be possible -- perform it themselves.)

---

```
    // global scope, dynamic accesses under non-syntactic global scope.
    if (globalsc->scopeKind() == ScopeKind::Global)
        fallbackFreeNameLocation_ = Some(NameLocation::Global(BindingKind::Var));
    else
        fallbackFreeNameLocation_ = Some(NameLocation::Dynamic());
```

MOZ_ASSERT(globalsc->scopeKind() == ScopeKind::NonSyntactic);

in the alternative?

---

```
                uint32_t atomIndex;
                if (!bce->makeAtomIndex(bi.name(), &atomIndex))
                    return false;
                if (!bce->emitIndexOp(JSOP_DEFVAR, atomIndex))
                    return false;
```

Again, can simplify to emitAtomOp.

---

```
    if (!ensureCache(bce))
        return false;

    // 'with' make all accesses dynamic and unanalyzable.
    fallbackFreeNameLocation_ = Some(NameLocation::Dynamic());
```

💯 💯 💯

---

```

    if (!bce->emitInternedScopeOp(index(), JSOP_ENTERWITH))
        return false;

    return appendScopeNote(bce);
```

I'm probably just being dumb, but why doesn't this need to |checkEnvironmentChainLength(bce)| like most of the other functions that add to the environment chain do? What's the way to know when a function should add to the chain and not do that, versus when it should check?

---

```

    MaybeCheckTDZ rv = CheckTDZ;
    for (TDZCheckCache* it = enclosing(); it; it = it->enclosing()) {
        if (it->cache_) {
            if (CheckTDZMap::Ptr p = it->cache_->lookup(name)) {
```

Something's probably going to complain about this |p| shadowing the outer one, so rename this one somehow.

---

```
    if (!ensureCache(bce))
        return false;

    CheckTDZMap::AddPtr p = cache_->lookupForAdd(name);
```

```
    if (p) {
```

if (CheckTDZMap::AddPtr p = ...) {

---

```
    return true;
}

bool
BytecodeEmitter::emitCheckIsObj(CheckIsObjectKind kind)
```

Would prefer that this not be moved from its original location, please.

---

```

    EmitterScope* es = bce_->innermostEmitterScope;
    int npops = 0;

#define FLUSH_POPS() if (npops && !bce_->flushPops(&npops)) return false
```

Make this a lambda and check calling it for failures the normal way.

---

```
        *answer = true;
```

This could use a comment as to why. (Maybe PNK_FUNCTION handles everything, such that this is never reached? But if so, you should instead add the case to the MOZ_CRASH("handled by parent nodes"); list near the end of the switch.)

---

```
        break;

      case NameLocation::Kind::FrameSlot:
        if (loc.isLexical() && !emitTDZCheckIfNeeded(name, loc))
            return false;
```

if (loc.isLexical()) {
  if (!emitTDZCheckIfNeeded(name, loc))
    return false;
}

and same a few lines later.

---

```
        if (!makeAtomIndex(name, &atomIndex))
            return false;
        if (loc.isLexical() && initialize) {
            // INITGLEXICAL always gets the global lexical scope. It doesn't
            // need a BINDGNAME.
```

Mm, nice.

---

```
        if (loc.isLexical() && initialize) {
            // INITGLEXICAL always gets the global lexical scope. It doesn't
            // need a BINDGNAME.
            MOZ_ASSERT(innermostScope()->is<GlobalScope>());
            op = JSOP_INITGLEXICAL;
```

Probably worth a comment here that we don't track TDZ or anything like that because getting will go through FetchName which does an uninitialized-check regardless, and setting will go through bindg?name which binds to a RuntimeLexicalError and so will complain when the set happens.

---

```
      case NameLocation::Kind::NamedLambdaCallee:
        if (!emitRhs(this, loc, emittedBindOp))
            return false;
        // Assigning to the named lambda is a no-op in sloppy mode but
        // does in strict mode.
```

s/does/throws/ ?

---

```
            // whenever parameters are (or might, in the
```

```
case of calls to eval)
        // assigned.
        FunctionBox* funbox = sc->asFunctionBox();
        if (funbox->argumentsHasLocalBinding() &&
!funbox->hasMappedArgsObj())
            funbox->setDefinitelyNeedsArgsObj();
```

If I nop out this line and run jstests and jit-tests, there are no failures. Are you absolutely certain lines 3143-3152 are needed? Unless you can show me a test that fails without these lines and passes with them, I think you need to remove this.

---

```
BytecodeEmitter::emitTDZCheckIfNeeded(JSAtom* name,
const NameLocation& loc)
{
    // Dynamic accesses have TDZ checks built into their
VM code and should
    // never emit explicit TDZ checks.
    MOZ_ASSERT(loc.hasKnownSlot() && loc.isLexical());
```

Split into two asserts.

---

```
    if (!check)
        return false;

    // We've already emitted a check in this basic
block.
    if (!*check)
```

Would prefer an explicit == comparison to boolish conversion, 'cause the reader isn't going to know whether truthy/falsy indicates a check's needed.

---

```
    if (!emitTree(pn->pn_left))
        return false;

    // Enter the scope before pushing the switch
BreakableControl since all
    // breaks are under the this scope.
```

under the this scope

---

```
    return emit1(JSOP_DEBUGAFTERYIELD);
}

bool
BytecodeEmitter::emitInitializeFunctionSpecialName(HandlePropertyName
 name, JSOp initialOp)
```

Given this function is only used in one caller, namely emitInitializeFunctionSpecialNames, I'd make it a lambda inside that function. That also lets you hoist the inPrologue() and isFunctionBox() assertions and remove the initialOp assertion.

---

```
BytecodeEmitter::emitInitializeFunctionSpecialName(HandlePropertyName
 name, JSOp initialOp)
{
    // A special name must be slotful, either on the
frame or on the
    // call environment.
    MOZ_ASSERT(lookupName(name).hasKnownSlot());
```

MOZ_ASSERT(inPrologue());

---

```
        if (!emitterScope.enterEval(this, sc-
>asEvalContext()))
            return false;
        switchToMain();
    } else {
        MOZ_ASSERT(sc->isModuleContext());
```

Could possibly have the module case first, then the other two cases in an else, so the switchTo*() could be shared.

---

```
    FunctionBox* funbox = sc->asFunctionBox();
```

```

    // The ordering of these EmitterScopes is important.
The decl env scope
```

s/decl env/named lambda/

---

```
            // to-be-destructured value on top of the
stack.
            if (!emit1(JSOP_POP))
                return false;
        }
    } else if (emitOption == PushInitialValues) {
```

DIE PUSHINITIALVALUES DIE

---

```

    auto emitRhs = [initializer, declList]
(BytecodeEmitter* bce, const NameLocation&, bool) {
        if (!initializer && !declList->isKind(PNK_VAR))
{
            // Lexical declarations are initialized to
undefined without an
            // initializer.
```

Maybe worth checking for just !initializer, then MOZ_ASSERT(declList->isKind(PNK_LET), "initializer-less var was handled above, and consts must have initializers").

---

```
    return emit1(JSOP_POP);
}

static bool
EmitAssignmentRhs(BytecodeEmitter* bce, ParseNode* rhs,
uint8_t offset)
```

I'll be astonished if this is better off un-inlined into the users when all's said and done. But we can land this as-is.

---

```

  if_again:
    /* Emit code for the condition before pushing
stmtInfo. */
    if (!emitTree(pn->pn_kid1))
        return false;
```

The lack of something TDZCheckCache-like to guard this, akin to what condswitch handling has, means that this testcase will fail to have a TDZ check for the final return bit:

```
(function()
{
  if ({}) {}
  else if (x) {}
  else {}

  return x;

  let x;
})()
```

---

```

        if (headLexicalEmitterScope->hasEnvironment()) {
            if (!emit1(JSOP_RECREATELEXICALENV))
// ITER RESULT
                return false;
        }
```

Just the reminder to dead-zone frame slots in the !hasEnvironment case if you haven't implemented that yet -- here and for for-in and for for(;;) all.

---

```
    return emit1(JSOP_GLOBALTHIS);
}


```

```
bool
BytecodeEmitter::emitCheckDerivedClassConstructorReturn()
```

"it's so much prettier"

**js/src/frontend/BytecodeEmitter.h**

```
    };
    EmitSection prologue, main, *current;

    /* the parser */
    Parser<FullParseHandler>* const parser;
```

I wasn't sure this was actually the parser, you sure this comment has go away?

---

```
    uint32_t        arrayCompDepth; /* stack depth of array in comprehension */

    unsigned        emitLevel;      /* emitTree recursion level */

    uint32_t        bodyScopeIndex; /* index into scopeList of the body scope */
```

I...think this number is intrinsically limited to be super-small. Like, 0 to 2 for functions, 0 for eval (?), and so on. Could we use a smaller integer type, for general size-hygiene, and to hopefully catch errors somewhat earlier if something goes really awry? Maybe?

---

```
    MOZ_MUST_USE bool init();

    template <typename Predicate /* (NestableControl*) -> bool */>
    NestableControl* findInnermostNestableControl(Predicate predicate) const;
```

I think this decl is undeclared/unused.

---

```
    // Finish taking source notes in cx's notePool. If successful, the final
    // source note count is stored in the out outparam.
    MOZ_MUST_USE bool finishTakingSrcNotes(uint32_t* out);

    // control whether emitTree emits a line number note.
```

Erm, s/c/C/ back as it was?

---

```

    enum DestructuringFlavor {
        DestructuringDeclaration,
        DestructuringFormalParameterInVarScope,
        DestructuringAssignment
```

Worth brief comments by these:

"destructuring into a declaration"
"destructuring into a formal parameter, when the formal parameters contain an expression that might be evaluated (and thus require this destructuring to assign not into the innermost scope that contains the function body's vars, but into its enclosing scope for parameter defaults"
"destructuring as part of an AssignmentExpression"

or somesuch.

**js/src/frontend/FullParseHandler.h**

```
    const TokenPos& pos() {
        return tokenStream.currentToken().pos;
    }

    inline ParseNode* makeAssignmentFromArg(ParseNode* lhs, ParseNode* rhs);
```

Just inline this function in its one caller -- it's super-small.

---

```
        return tokenStream.currentToken().pos;
    }

    inline ParseNode* makeAssignmentFromArg(ParseNode* lhs, ParseNode* rhs);
    inline void replaceLastFunctionFormalParameter(ParseNode* funcpn, ParseNode* pn);
```

I'd also prefer if this were inlined in its one caller. It's a little bigger, but inlining seems clearer than having to understand when/how this is used from its name.

**js/src/frontend/NameAnalysisT ypes.h**

```
    {
        MOZ_ASSERT(JOF_OPTYPE(JSOp(*pc)) == JOF_ENVCOORD);
    }

    inline EnvironmentCoordinate() {}
```

The inlines on functions with bodies inside of classes are unnecessary.

---

```
        MOZ_ASSERT(slot < ENVCOORD_SLOT_LIMIT);
        slot_ = slot;
    }

    uint32_t hops() const {
```

Since hops is limited, it might be worth using uint8_t for its return value, with a static_assert about the limit value. Seems better than a super-overlarge type. Also for hops_ as well, to avoid some upper-bits modification behavior. And might as well flip slots_ and hops_, then, just for general packing largest-to-smallest hygiene even if it hopefully won't matter here.

---

```
    bool closedOver_;

  public:
    // Default constructor for InlineMap.
    DeclaredNameInfo() = default;
```

Maybe move this beneath the real constructor, and adjust to "Needed by InlineMap". The current wording reads like redundant typo/copypasta to me.

---

```
        // Cannot statically determine where the name lives. Needs to walk the
        // environment chain to search for the name.
        Dynamic,

        // The name lives on the global. Search for the name on the global scope.
```

"on the global" implies property, to me. Maybe "on the global or in a global lexical binding"?

---

```
  private:
    // Where the name lives.
    Kind kind_;

    // If the name is not a dynamic lookup, the kind of the binding.
```

s/is not a dynamic lookup/isn't Dynamic{,AnnexBVar}/ maybe? Best to be clear about exact kindness.

---

```
    // If the name is closed over and accessed via EnvironmentCoordinate, the
    // slot on the environment.
    //
    // Otherwise LOCALNO_LIMIT/ENVCOORD_SLOT_LIMIT.
    uint32_t slot_ : ENVCOORD_SLOT_BITS;
```

Sort the fields largest to smallest again?

---

```
                hops_ == other.hops_ && slot_ == other.slot_;
```

```
    }

    bool operator!=(const NameLocation& other) const {
        return !operator==(other);
```

Minor preference for

return !(*this == other);

---

```
        return slot_;
    }

    NameLocation addHops(uint8_t more) {
        MOZ_ASSERT(hops_ < ENVCOORD_HOPS_LIMIT - more);
```

|more| isn't a constant, so it could exceed ENVCOORD_HOPS_LIMIT and that would be Bad. Please assert it's <= ENVCOORD_HOPS_LIMIT.

**js/src/frontend/NameCollections.h**

```
    InlineMap<JSAtom*,                                  \
              RecyclableAtomMapValueWrapper<MapValue>,  \
              24,                                       \
              DefaultHasher<JSAtom*>,                   \
              SystemAllocPolicy>
```

template<typename MapValue>
using RecyclableNameMap = InlineMap<JSAtom*, RecyclableAtomMapValueWrapper<MapValue>, 24, DefaultHasher<JSAtom*>, SystemAllocPolicy>;

and then

using DeclaredNameMap = RecyclableNameMap<DeclaredNameInfo>;
using CheckTDZMap = RecyclableNameMap<MaybeCheckTDZ>;
...

Heck, I'd just use RecyclableNameMap<T> rather than have the four extra typedefs, if it were me.

---

```
        }
    }
};

#define POOLED_COLLECTION_PTR_METHODS(N, T)
\
```

This feels CRTP-able in a followup.

**js/src/frontend/ParseNode.cpp**

```
      // are non-null.
      case PNK_WITH: {
        MOZ_ASSERT(pn->isArity(PN_BINARY));
```

There's a long list of cases a dozen up that you could/should just add this to, rather than handling it uniquely, now.

---

```
                NULLCHECK(pn->pn_right = cloneParseTree(opn->pn_right));
            else
                pn->pn_right = pn->pn_left;
        }
        pn->pn_iflags = opn->pn_iflags;
```

So if we removed the last user of cloning, is this all dead code now? Or did I misunderstand and we didn't actually remove the last clone-user? (Or was it atop this patch and I'm just out of date?) Would be soooooooooo nice to just kill this.

**js/src/frontend/ParseNode.h**

```
{
    uint16_t pn_type;   /* PNK_* type */
```

```
    uint8_t pn_op;      /* see JSOp enum and jsopcode.tbl */
    uint8_t pn_arity:4; /* see ParseNodeArity enum */
    bool pn_parens:1;   /* this expr was enclosed in parens */
```

At one time MSVC wouldn't pack adjacent bitfields of differing type -- or maybe it was differing type when the two types had different signedness. Please verify MSVC packs this all into a single 32-bit location.

**js/src/frontend/Parser .cpp**

```
using mozilla::Nothing;
using mozilla::Some;
using mozilla::PodCopy;
using mozilla::PodZero;
using mozilla::Move;
```

Alphabetize.

---

```
      case DeclarationKind::Import:
        return "import";
      case DeclarationKind::BodyLevelFunction:
        return "function";
      case DeclarationKind::LexicalFunction:
```

Handle these two cases with one return, not two.

---

```

    MOZ_CRASH("Bad DeclarationKind");
}

static inline bool
```

Remove the inline, compiler should be able to figure it out -- plus putting an inline on this means if it ever becomes unused, we wouldn't know.

---

```
    declared_->remove(p);
}

void
ParseContext::Scope::removeArgumentsVarDeclaration(ParseContext* pc)
```

Possibly worth a Scope::removeDeclaration(ParseContext* pc, JSAtom* name, DeclarationKind kind) helper that removeSimpleCatchParameter and removeArgumentsVarDeclaration could both call.

---

```
            if (fun->isArrow())
                continue;
            allowNewTarget_ = true;
            allowSuperProperty_ = fun->allowSuperProperty();
            allowSuperCall_ = fun->isDerivedClassConstructor();
```

Assert that all these fields are false on entry to the function, so if there's no function on the scope chain, we're not leaving the fields unset?

---

```
            return;
        }
    }

    thisBinding_ = ThisBinding::Global;
```

Note the contrast with computeAllowSyntax: this function does always clearly set thisBinding_.

---

```
SharedContext::computeInWith(Scope* scope)
{
    for (ScopeIter si(scope); si; si++) {
        if (si.kind() == ScopeKind::With) {
            inWith_ = true;
```

Assert inWith_ is false on entry.

```
    computeAllowSyntax(enclosingScope);
    computeInWith(enclosingScope);
    computeThisBinding(enclosingScope);

    // Like all things Debugger, Debugger.Frame.eval needs special
```

s/things Debugger/things Debugger and eval/ ?

---

```
    if (!closedOverBindingsForLazy_.acquire(cx))
        return false;

    if (!sc()->strict() && !innerFunctionBoxesForAnnexB_.acquire(cx))
        return false;
```

These days we do

```
if (!sc()->strict()) {
  if (!innerFunctionBoxesForAnnexB_.acquire(cx))
    return false;
}
```

i.e. have guards on fallible operations separate from the fallible operations themselves.

---

```
UsedNameTracker::note(ExclusiveContext* cx, JSAtom* name, uint32_t scriptId, uint32_t scopeId)
{
    UsedNameMap::AddPtr p = map_.lookupForAdd(name);
    if (p) {
```

Could do

```
if (UsedNameMap::AddPtr p = map_.lookupForAdd(name)) {
 } else {
 }
```

because |p| would be in scope for both arms of the if-else.

---

```
void
FunctionBox::initStandaloneFunction(Scope* enclosingScope)
{
    // Standalone functions are Function or Generator constructors and are
    // always scoped to the global.
```

Seems like we can assert enclosingScope->kind() == ScopeKind::Global or NonSyntactic, then.

---

```
    JSFunction* fun = function();

    // Arrow functions and generator expression lambdas don't have
    // their own `this` binding.
    if (!fun->isArrow()) {
```

I'd handle arrow functions first, then everything else second, to avoid mental negation overhead.

---

```
        needsThisTDZChecks_ = sc->needsThisTDZChecks();
        thisBinding_ = sc->thisBinding();
    }

    if (sc->inWith()) {
```

Lines 501-509 are duplicated at 511-519. Rebase error?

---

```
 * Parse a top-level JS script.
 */
template <typename ParseHandler>
typename ParseHandler::Node
Parser<ParseHandler>::parse()
```

We should rename this to parseGlobalScript in a followup.

```
    if (!tokenStream.getToken(&tt, TokenStream::Operand))
        return null();
    if (tt != TOK_EOF) {
        report(ParseError, false, null(), JSMSG_GARBAGE_AFTER_INPUT,
               "script", TokenKindToDesc(tt));
```

Maybe use checkStatementsEOF() here instead? I see an error number/message distinction, but maybe we can unify those or get rid of one or the other.

```
template <typename ParseHandler>
bool
Parser<ParseHandler>::notePositionalFormalParameter(Node fn, HandlePropertyName name,
                                                    bool disallowDuplicateParams,
                                                    bool *duplicatedParam)
```

* by type

```
                                                    bool disallowDuplicateParams,
                                                    bool *duplicatedParam)
{
    AddDeclaredNamePtr p = pc->functionScope().lookupDeclaredNameForAdd(name);
    if (p) {
```

if (auto p = ...) {

will reduce |p|'s scope to just where it's needed, so no need for people to think about it after the if-else ends.

```
            }
        }

        if (disallowDuplicateParams) {
            report(ParseError, false, null(), JSMSG_BAD_DUP_ARGS);
```

Seems mildly preferable to check this before the strictness check, because it applies regardless of strictness. They could make their code non-strict to avoid the currently-earlier error...but then they'd be back to square one when they hit this.

```
    //   { var x; var x; }
    //   { { let x; } var x; }

    for (ParseContext::Scope* scope = pc->innermostScope();
         scope != pc->varScope().enclosing();
```

I'd prefer caching |pc->varScope().enclosing()| so the user doesn't have to know how much/little overhead the computation involves.

```
         scope != pc->varScope().enclosing();
         scope = scope->enclosing())
    {
        AddDeclaredNamePtr p = scope->lookupDeclaredNameForAdd(name);
        if (p) {
```

if (auto p = ...) {

```

template <typename ParseHandler>
bool
Parser<ParseHandler>::tryDeclareVarForAnnexBLexicalFunction(HandlePropertyName
```

```
 name,
bool* tryAnnexB)
```

I'd prefer if |tryAnnexB| were, universally, an enum of two values. I think you can not bother doing it right now, but I want to see this in a fairly fast followup.

---

```

template <typename ParseHandler>
bool
Parser<ParseHandler>::noteDeclaredName(HandlePropertyName
 name, DeclarationKind kind,
                                       TokenPos pos)
```

Hmm. "note" as the universal verb for reacting to anything isn't working for me.

It's fine as in "noteUse", because that's effectively just using a thing already seen, or to be seen, or not to be seen (and deoptimized accordingly).

For formal parameters, it feels to me like "add" is the right verb, as we're adding items to the internal representation of the function's arguments. It's not quite just "noting" them.

For non-parameter declarations *particularly*, I strongly want a more active verb. The parser isn't just saying, "Oh, I saw this". It's creating data structures to record the declaration, adopting any existing uses that target it, &c. "addDeclaration" or "declareName", maybe?

As regards "noteDeclaredName" specifically, I also strongly disagree with the "declared name" part of it. The strong implication of the name, to me, is that the name has already been declared, by the parser's understanding. But that's rarely true (and only for fairly uncommon duplicate declarations). Hence "declaration", or eliminating a claim the name's already been declared.

Whichever way you go (I could go for either), the other name-declaration functions should have similar names. Following either scheme, tryAddVarDeclarationForAnnexBLexicalFunction would be the worst name, but it's not much longer than noteDPFP *already* is. So I guess it could slide.

---

```

        ParseContext::Scope* scope = pc-
>innermostScope();
        AddDeclaredNamePtr p = scope-
>lookupDeclaredNameForAdd(name);

        if (p) {
```

Can unify decl/if.

---

```
            }

            // Update the DeclarationKind to make a
LexicalFunction
            // declaration that shadows the
VarForAnnexBLexicalFunction.
            p->value() = DeclaredNameInfo(kind);
```

This overwrites closedOver_, but it hasn't really been set yet, so that's fine. I think I'd either add a comment to that effect, assert !isClosedOver() as determined by the behavior of the constructor, or added an extra mutating accessor alterKind(kind)

that changes just the one field (so readers don't have to wonder whether the extra write is/isn't correct).

Pick something, just don't leave the reader guessing here. :-)

---

```
1165      }
1166
1167      case DeclarationKind::Let:
1168      case DeclarationKind::Const:
1169      case DeclarationKind::Import:
```

Move Import down below the let/const handling, and assert |name != context->names().let| and then fall through. Module code is strict mode code, so |let| is keywordish and can't reach here.

---

```
1191        // needs a special check if there is an extra var scope due to
1192        // parameter expressions.
1193        if (pc->isFunctionExtraBodyVarScopeInnermost()) {
1194            DeclaredNamePtr p = pc->functionScope().lookupDeclaredName(name);
1195            if (p && DeclarationKindIsParameter(p->value()->kind())) {
```

If we find a declaration, DKIP is always going to be true here *except* if |name == context->names().arguments|. So please make this

```
  if (DeclaredNamePtr p = pc->functionScope().lookupDeclaredName(name)) {
    DeclarationKind declKind = p->value()->kind();
    bool declIsParameter = DeclarationKindIsParameter(declKind);
    MOZ_ASSERT(declIsParameter ||
               (declKind == DeclarationKind::Var && name == context->names().arguments),
               "function scope contains only parameter bindings, arguments, and internal "
               "dot-names that can't be lexically declared");
    if (declIsParameter) {
      reportRedeclaration(...);
      return false;
    }
  }
```

This arguments-oddity appears to be tested by the existing js/src/tests/ecma_5/strict/12.14.1.js, so I don't think we need any existing test to exercise this and be sure we handle the edge case right.

---

```
1236 }
1237
1238 template <typename ParseHandler>
1239 bool
1240 Parser<ParseHandler>::noteUsedName(HandlePropertyName name)
```

Probably keep this consistent with UsedNameTracker::note, so noteUse. Or both could be noteNameUse if you want.

---

```
1250        return true;
1251
1252    // Global bindings are properties and not actual bindings, we don't need
1253    // to know if they are closed over. So no need to track used name at the
1254    // global scope. It is not incorrect to track them, this is an
```

s/,/:/ or something to avoid run-on-ness.

---

```
}

template <typename ParseHandler>
bool
Parser<ParseHandler>::hasUsedName(HandlePropertyName name)
```

usesName seems a slightly better name to me.

---

```
        }
    }

    GlobalScope::Data* bindings = nullptr;
    uint32_t numBindings = funs.length() + vars.length() + lets.length() + consts.length();
```

Add a size_t loop-counter to the loop above, and assert that it's <= UINT32_MAX before you evaluate this addition.

---

```
        BindingName* start = bindings->names;
        BindingName* cursor = start;

        // Keep track of what vars are functions. This is only used in BCE to omit
        // superfluous DEFVARs.
```

This comment is repeated a couple times in these functions. I'm not convinced all of them are precisely correct. Please examine them and adjust the comments however necessary for the specific cases -- I imagine you have copypasta right now.

---

```
        }
    }

    ModuleScope::Data* bindings = nullptr;
    uint32_t numBindings = imports.length() + vars.length() + lets.length() + consts.length();
```

Same size_t counting and assertion after loop. (And assuming there are more of these, apply similarly to subsequent new*ScopeData functions.)

---

```
            // exprs, which induces a separate var environment, should be the
            // special internal bindings.
            MOZ_ASSERT_IF(hasParameterExprs,
                          bi.name() == context->names().dotThis ||
                          bi.name() == context->names().arguments);
```

This might need to be updated for dotGenerator and friends based on the discussion we had on IRC on Monday -- I think it does? Looks to me like dotGenerator is defined in every scope up to the var scope, so I don't see why it couldn't be here, in the absence of parameter expressions forcing the extra scope into existence.

---

```
        return nullptr;

    // All evals have an implicit non-extensible lexical scope.
    ParseContext::Scope scope(this);
    if (!scope.init(pc))
```

lexicalScope seems a mildly better name, no?

---

```
    TokenKind tt;
    if (!tokenStream.getToken(&tt, TokenStream::Operand))
        return null();
    if (tt != TOK_EOF) {
        report(ParseError, false, null(), JSMSG_GARBAGE_AFTER_INPUT, "module", TokenKindToDesc(tt));
```

checkStatementsEOF also plausible here -- maybe with that provided const char* argument to cover "expression"/"statement"/"module" distinctions.

---

```
    if (!modulesc->builder.buildTables())
        return null();

    // Check exported local bindings exist and mark them as closed over.
    for (auto entry : modulesc->builder.localExportEntries()) {
```

I'd feel much safer if |entry| were a Rooted<ExportEntryObject*> than with it an under-typed |auto|.

---

```
    //
    // If we have an extra var scope due to parameter expressions and the body
    // declared 'var arguments', move the declaration to the function scope.
    DeclaredNamePtr p = varScope.lookupDeclaredName(argumentsName);
    if (p && p->value()->kind() == DeclarationKind::Var) {
```

This should test for DeclarationKind::ForOfVar as well so the following test passes (please add to the final patch, natch):

assertEq(function() { for (var arguments of []) break; return arguments; }().length, 0);

I see we already have a DeclarationKindIsVar function that *does not* simply add only ForOfVar. I'm not sure what its existing uses look like, but it might or might not be desirable to change its meaning to just the two definitely-var cases, and have a *different* function that includes the function oddball cases. In any case: we should probably have two different functions for var-testing, and make sure we're using the right one in the right case.

---

```
    }

    // Lazy functions inner to another lazy function need to be remembered the
```

remembered by?

---

```
                    return false;

handler.addFunctionFormalParameter(funcpn, destruct);
```

Move this call into noteDestructuredPositionalFormalParameter, for consistency with notePositionalFormalParameter.

---

```
    if (!fun)
        return null();

    if (synthesizedStmtForAnnexB) {
        Node synthesizedStmtList = handler.newStatementList(handler.getPosition(fun));
```

Grab a position in the synthesis part at the beginning of this function, and use that. The position you're using here is totally wrong, the '}' closing the function or something.

---

```
                    return false;
                }
                // We don't reparse global scopes, so we keep track of the one
                // possible strict violation that could occur in the directive
```

```
                // prologue -- octal escapes -- and complain now.
```

We should regroup when this lands and reevaluate this assertion. I'm not sure it's true any more now that parameter expressions are a thing.

---

```
        // Namespace imports are are not indirect bindings but lexical
        // definitions that hold a module namespace object. They are treated
        // as const variables which are initialized during the
        // ModuleDeclarationInstantiation step.
        RootedPropertyName bindingAtom(context, tokenStream.currentName());
```

Just name this |name|? "atom"'s a total misnomer.

---

```
        tokenStream.ungetToken();
    } else {
        // Annex B.3.5 has different early errors for vars in for-of loops.
        if (*isForOfp && declKind && *declKind == DeclarationKind::Var)
            *declKind = DeclarationKind::ForOfVar;
```

Hrm. I'd prefer if this were put in callers, in the code where they handle the |isForOf| case. It's only two cases, which seems preferable to foisting DeclarationKind* overhead on every caller (including the one that considers for-of to be an error).

---

```
            return null();

        // Push a temporary ForLoopLexicalHead Statement that allows for
        // lexical declarations, as they are usually allowed only in braced
        // statements.
```

Mm. Moving bracing-restrictions into the process of parsing initially will let us get rid of this, which seems pretty nice.

---

```
    // construct that is forbidden in strict mode code, but doesn't even merit a
    // warning under JSOPTION_EXTRA_WARNINGS.  See
    // https://bugzilla.mozilla.org/show_bug.cgi?id=514576#c1.
    if (pc->sc()->strict() && !report(ParseStrictError, true, null(), JSMSG_STRICT_CODE_WITH))
        return null();
```

if (strict) {
  if (!report(...))
    return null();
}

while you're changing things.

---

```

    if (!report(ParseStrictError, pc->sc()->strict(), nameNode, JSMSG_BAD_STRICT_ASSIGN, chars))
        return false;

    MOZ_ASSERT(!pc->sc()->strict(),
```

Assign pc->sc()->strict() to a local and use the local in assertion and report both?

---

```

            Node rhs;
            {
                // Clearing `inDestructuringDecl` allows name use to be noted
                // in `identifierName` See Bug: 1255167.
```

Period after `identifierName`, and capitalizing "bug" and the colon are weird.

**js/src/frontend/Parser .h**

```
46        StatementKind kind_;
47
48      public:
49        using Nestable<Statement>::enclosing;
50        using Nestable<Statement>::findNearest;
```

It's a little odd having usings of publicly-inherited functions, but it checks out. If you could privately inherit, I'd mildly prefer that so this doesn't look strange.

---

```
58
59        template <typename T>
60        T& as() {
61            MOZ_ASSERT(is<T>());
62            return static_cast<T&>(*this);
```

Please define this immediately after the definition of is(). This is super-close to a used-but-not-defined warning, but for the templates in play, and I'd rather we not risked it.

---

```
75    };
76
77    class LabelStatement : public Statement
78    {
79        RootedAtom label_;
```

Should Statement be MOZ_STACK_CLASS, at an absolute minimum for documentation purposes?

---

```
97        // Names declared in this scope. Corresponds to the union of
98        // VarDeclaredNames and LexicallyDeclaredNames in the ES spec.
99        //
100       // A 'var' declared name is a member of the declared name set of every
101       // scope in its scope contour.
```

I can hazily guess what "scope contour" might mean, but I shouldn't have to. And I don't see anywhere else that defines the term, at least not in frontend/. Expand this into a prosaic description of whatever it is is meant here, please -- maybe "every scope in which a lexical declaration having the same name as a var name would be a syntax error" or something, not sure.

---

```
122       template <typename ParseHandler>
123       explicit Scope(Parser<ParseHandler>* parser)
124         : Nestable<Scope>(&parser->pc->innermostScope_),
125           declared_(parser->context->frontendCollectionPool()),
126           id_(parser->usedNames.nextScopeId())
```

Again, declare here, define after nextScopeId is defined? I guess templates are saving you again here.

---

```
190
191           void settle() {
192               if (isVarScope_)
193                   return;
194               while (!declaredRange_.empty()) {
```

Add a comment in here that we're exposing all 'var' bindings in the var scope (and not in any other), and that bindings are exposed vars first, then lexicals, so breaking once a lexical is examined skips all the vars. Or something like that. (Is that an accurate statement? And assuming it is, pointing to what precisely enforces that ordering would be nice.)

---

```
227           }
228
229           void setClosedOver() {
230               MOZ_ASSERT(!done());
```

```
                return declaredRange_.front().value()->setClosedOver();
```

Just make the call, eliminate the return?

---

```
    Statement* innermostStatement_;

    // The innermost scope, i.e., top of the scope stack.
    //
    // The outermost scope in the stack is varScope_.
```

Is this statement true in the case of functions with parameter expressions? I'd think the outermost there would be the FunctionScope, and varScope_ would be the separate VarScope nested within it (and possibly within a parameters scope, if the parameter expressions contain a direct eval) to hold function variables.

---

```
    // scope for named lambdas.
    mozilla::Maybe<Scope> namedLambdaScope_;

    // If isFunctionBox(), the scope for the function. If there are no
    // parameter expressions, this is scope for the entire function. If there
```

is the scope

---

```
    {
        if (isFunctionBox()) {
            if (functionBox()->function()->isNamedLambda())
                namedLambdaScope_.emplace(prs);
            functionScope_.emplace(prs);
```

Possibly worth a "This emplacement ordering is necessary for proper LIFO deallocation." comment or so.

---

```
//                                  parsed?"
//
// The algorithm:
//
// Let Used by a map of names to lists.
```

Perhaps say each list starts life empty, so you can avoid the 3b wording-(IMO-)awkwardness below.

---

```
// 4. When we finish parsing a scope S in script P, for each declared name d in
//    Declared(S):
// 4a. If d is found in Used, mark d as closed over if there is a value
//     (P_d, S_d) in Used[d] such that P_d > P and S_d > S.
// 4b. Remove all values (P_d, S_d) in Used[d] such that S_d are >= S.
```

Please somehow indent the substeps for readability, and probably get rid of the numeric parts of them.

---

```
            if (uses_.empty() || uses_.back().scopeId < scopeId) {
                Use use;
                use.scriptId = scriptId;
                use.scopeId = scopeId;
                return uses_.append(use);
```

Could do

return uses_.append({ scriptId, scopeId });

which seems better/self-documenting.

---

```
        return map_.init();
    }

    uint32_t nextScriptId() {
        MOZ_ASSERT(scriptCounter_ != UINT32_MAX);
```

Add a reason-string:

"ParseContext::init should have prevented wraparound"

```
        return scriptCounter_++;
    }

    uint32_t nextScopeId() {
        MOZ_ASSERT(scopeCounter_ != UINT32_MAX);
```

And,

"ParseContext::Scope::init should have prevented wraparound"

```
        return map_.lookup(name);
    }

    template <typename Predicate /* (JSAtom*, Use) -> bool */>
    bool hasUse(Predicate p) const {
```

This function is unused.

```
        }
        return false;
    }

    MOZ_MUST_USE bool note(ExclusiveContext* cx, JSAtom* name, uint32_t scriptId, uint32_t scopeId);
```

I'd prefer "noteUse" for a name for this to the very-generic "note".

```
    RewindToken getRewindToken() const {
        RewindToken token;
        token.scriptId = scriptCounter_;
        token.scopeId = scopeCounter_;
        return token;
```

return { scriptCounter_, scopeCounter_ };

```
    // for-in/of loop head, returning the iterated expression in
    // |*forInOrOfExpression|.  (An "initial declaration" is the first
    // declaration in a declaration list: |a| but not |b| in |var a, b|, |{c}|
    // but not |d| in |let {c} = 3, d|.)
    Node declarationPattern(Node decl, DeclarationKind declKind, TokenKind tt,
```

BINDDATA IS DEAD WOO WOO WOO

```
    };

    bool checkAndMarkAsAssignmentLhs(Node pn, AssignmentFlavor flavor,
                                     PossibleError* possibleError=nullptr);
    bool matchInOrOf(bool* isForInp, bool* isForOfp, DeclarationKind* delcKind = nullptr);
```

declKind

declKind so grody here, stupid language.  :-(

```
    bool makeSetCall(Node node, unsigned errnum);

    Node cloneForInDeclarationForAssignment(Node decl);
    Node cloneForOfDeclarationForAssignment(Node decl);
```

These are no longer defined.

**js/src/frontend/SharedContext.h**

```
    // JSOP_FUNCTIONTHIS in the prologue to initialize it.
    bool hasThisBinding:1;

```

```
    // Whether this function has inner functions.
    bool hasInnerFunctions:1;
```

hasInnerFunctions is fine for the name, but the docs should clarify what "inner functions" are. Something like,

// Whether this function contains nested function syntax of any variety (arrow, expression, generator, statement, method).

---

```
    FunctionScope::Data* functionScopeBindings_;

    // Names from the extra 'var' scope of the function, if the parameter list
    // has expressions.
    VarScope::Data* extraVarScopeBindings_;
```

Could be worth comments by these fields that the Parser root handles tracing these.

---

```
    uint16_t        length;

    uint8_t         generatorKindBits_;     /* The GeneratorKind of this function. */
    bool            isGenexpLambda:1;       /* lambda from generator expression */
    bool            hasDestructuringArgs:1; /* parameter list contains destructuring expression */
```

Args, or Params? If a change is needed, followup.

---

```
    }

    bool hasExtraBodyVarScope() const {
        return hasParameterExprs && (extraVarScopeBindings_ ||
needsExtraBodyVarEnvironmentRegardlessOfBindings());
```

I'd linebreak after &&, not after ||.

---

```
    return static_cast<ModuleSharedContext*>(this);
}

inline GlobalSharedContext*
SharedContext::asGlobalContext()
```

I'd put these asFoo functions directly after each subclass.

**js/src/frontend/SyntaxParseHandler.h**

```
    MOZ_MUST_USE bool setLastFunctionFormalParameterDefault(Node funcpn, Node pn) { return true; }
    Node newFunctionDefinition() { return NodeFunctionDefinition; }
    bool setComprehensionLambdaBody(Node pn, Node body) { return true; }
    void setFunctionFormalParametersAndBody(Node pn, Node kid) {}
```

If either of these can assert what any of the Nodes are, please do so. At least for the additions you've made -- the pre-existing ones can slide for now.

---

```
    }

    Node newAssignment(ParseNodeKind kind, Node lhs, Node rhs, JSOp op)
    {
```

{ to previous line.

---

```
        return false;
    }
    JSAtom* nextLazyClosedOverBinding() {

MOZ_CRASH("SyntaxParseHandler::canSkipLazyClosedOverBindings must return false");
```

```
548     }
```

Hm, would be nice if whatever this is doing, could be done in a way that didn't require it live in the handler, if this is going to just crash. Better not to have always-crash functions if you can help it. Maybe the user of these three functions, should be a templated class/function specialized for FullParseHandler and SyntaxParseHandler, so that the entire context in which these are used is effectively done per-handler all at once.

**js/src/gc/Marking.cpp**

```
1357        break;
1358      }
1359
1360      case ScopeKind::With:
1361        break;
```

Just returning here would be clearer, IMO.

**js/src/gc/Nursery.cpp**

```
22 #include "jit/JitFrames.h"
23 #include "vm/ArrayObject.h"
24 #include "vm/Debugger.h"
25 #if defined(DEBUG)
26 #include "vm/EnvironmentObject.h"
```

Double-check this is needed ('cause most stuff in it was renamed, yet nothing in this file had to be renamed).

**js/src/jit-test/tests/arguments/defaults-call-function.js**

```
3 function f1(a=g()) {
4   function g() {
5   }
6 }
7 assertThrowsInstanceOf(f1, ReferenceError);
```

Might be worth doing s/g/h/g on the function below, then adding |this.g = () => 42; assertEq(f1(), undefined);|. Just to test the real underlying effect, not the incidental ReferenceError because there's no |g| in the enclosing environment.

**js/src/jit-test/tests/arguments/defaults-scoping.js**

```
24 }
25 var hf = h();
26 assertEq(hf('x'), 'global');
27 assertEq(hf('f'), hf);
28 assertEq(hf('var x = 3; x'), 3);
```

assertEq(x, "global"); here?

**js/src/jit-test/tests/arguments/defaults-with-rest.js**

```
17 function f3(a=rest, ...rest) {
18     assertEq(a, 1);
19     assertEqArray(rest, [2, 3, 4]);
20 }
21 assertThrowsInstanceOf(f3, ReferenceError);
```

I assume these two functions' bodies don't execute? Empty them out, please.

**js/src/jit-test/tests/basic/bug646968-4.js**

```
6 var s = "", x = {a: 1, b: 2, c: 3};
7 for (let x in eval('x'))
8     s += x;
9 assertEq(s, "");
```

Just make this

```
assertThrowsInstanceOf(function() {
  for (let x in eval("x"))
    continue;
}, ReferenceError);
```

to get rid of noise for readers.

**js/src/jit-test/tests/basic/bug678087.js**

```
   TestCase: Float64Array
}) for each(let TestCase in [TestCase]) {}
 } while (i++ < 10);
function TestCase(n, d, e, a) {}
}, ReferenceError);
```

Put the assertThrowsInstanceOf around just the for each, for precision?

**js/src/jit-test/tests/coverage/simple.js**

```
}
//FNF:1
//FNH:1
//LF:8
//LH:7
```

...I am not going to ask what/how this test works.

**js/src/jit/BaselineCompiler .cpp**

```
        masm.loadPtr(Address(callee, JSFunction::offsetOfEnvironment()), scope);
        masm.storePtr(scope, frame.addressOfEnvironmentChain());

        if (fun->needsFunctionEnvironmentObjects()) {
            // Call into the VM to create the extra environment object.
```

Maybe s/extra/proper/, since this might create multiple environments?

**js/src/jit/IonBuilder .cpp**

```
void
IonBuilder::initLocals()
{
    // Initialize all frame slots to undefined. Lexical bindings are temporal
    // dead zoned in bytecode.
```

If this implies that everything's set to |undefined| initially, then bytecode overwrites lexicals to uninitialized, that's obviously inefficient. (Tho could be that Ion write-tracking eliminates duplication that's present in bytecode.) If so, file a bug to write the proper value from the start?

---

```
                    return false;
            }

            // TODO: Parameter expression-induced extra var environment not
            // yet handled.
```

???

---

```
        // These opcodes are currently unhandled by Ion, but in principle
        // there's no reason they couldn't be.  Whenever this happens, OSR will
        // have to consider that JSOP_{FRESHEN,RECREATE}BLOCK mutates the env
        // chain -- right now MBasicBlock::environmentChain() caches the env
        // chain.  JSOP_{FRESHEN,RECREATE}BLOCK must update that stale value.
```

s/BLOCK/LEXICALENV/, twice.

---

```
    //   LOOPHEAD
    // body:
    //    ; [body]
    // [increment:]
    //   [{FRESHEN,RECREATE}LEXICALENV, if needed by a cloned block]
```

s/block/env/ or so?

**js/src/jit/IonCaches.cpp**

```
    // If we're calling a getter on the global, inline the logic for the
    // 'this' hook on the global lexical env and manually push the global.
    if (IsGlobalLexicalEnvironment(obj))
        masm.extractObject(Address(object, EnvironmentObject::offsetOfEnclosingEnvironment()),
                           object);
```

Should be braced.

**js/src/jit/MacroAssembler .cpp**

```
    *startOfUndefined = first;

    if (first != 0 && IsUninitializedLexical(templateObj->getSlot(first - 1))) {
        for (; first != 0; --first) {
            if (!IsUninitializedLexical(templateObj->getSlot(first - 1)))
```

I don't think you need the IsUninitializedLexical check on this |if|, do you? Or am I misreading at a quick skim?

**js/src/jit/RematerializedFrame.cpp**

```
InlineFrameIterator& iter,
MaybeReadFallback& fallback,
GCVector<RematerializedFrame*>& frames)
{
    Rooted<GCVector<RematerializedFrame*>> tempFrames(cx, GCVector<RematerializedFrame*>(cx));
```

Seems like we could just clear |frames| and write into it directly (and, um, why is the argument not a MutableHandle or so), but can be fixed later.

**js/src/jsapi.cpp**

```
    return !!script;
```

Seems like we should change this (and the other compiling functions) to return JSScript* in a followup. Nice to see this function behaving in a more normal fashion.

**js/src/jscompartment.cpp**

```
JSCompartment::trace(JSTracer* trc)
{
    savedStacks_.trace(trc);

    // Atoms are always tenured.
```

Is there a way to static_assert this at all?

**js/src/jscompartment.h**

```
    // names declared using FunctionDeclaration, GeneratorDeclaration, and
    // VariableDeclaration declarations in global code in this compartment.
    // Names are only added to this list, never removed -- not even if the
    // property was created as configurable by eval code, then subsequently
    // deleted.
```

Maybe "Names are only removed from this list by a |delete IdentifierReference| that successfully removes that global property." to replace the last sentence here, that no longer holds in light of more-careful spec reading.

---

```
        explicit WrapperEnum(JSCompartment* c) : js::WrapperMap::Enum(c->crossCompartmentWrappers) {}
    };

    js::LexicalEnvironmentObject* getOrCreateNonSyntacticLexicalEnvironment(
```

```
601         JSContext* cx, js::HandleObject enclosing);
```

Weird indentation. If this is to avoid 99ch, I think we either overflow, or we bump the function name to a new line separate from the return type.

**js/src/jsfun.h**

```
118                                       use the accessor! */
119                 js::LazyScript* lazy_; /* lazily compiled script, or nullptr */
120             } s;
121             JSObject*   env_;    /* environment for new activations;
122                                     use the accessor! */
```

"use the accessor!" can disappear now.

**js/src/jsopcode.cpp**

```
1332            if (!fi.isDestructured())
1333                return fi.name();
1334
1335            // Destructured arguments have no single binding name.
1336            return Atomize(cx, "(destructured parameter)", strlen("(destructured parameter)"));
```

static const char destructuredParam[] = "(destructured parameter)";
return Atomize(cx, destructuredParam, ArrayLength(destructuredParam) - 1);

**js/src/jsscript.cpp**

```
288     MOZ_CRASH("Scope not found");
289 }
290
291 enum XDRClassKind {
292     CK_RegexpObject = 0,
```

Given enums start at 0, the initializer seems unnecessary.

---

```
665         RootedScope scope(cx);
666         RootedScope enclosing(cx);
667         ScopeKind scopeKind;
668         uint32_t enclosingScopeIndex = 0;
669         for (i = 0; i != nscopes; ++i) {
```

Assuming none of these |i| are used after their loops, probably, it'd be nice in a followup mini-change to make them all loop-local.

---

```
2466 /* static */ bool
2467 JSScript::initFunctionPrototype(ExclusiveContext* cx, Handle<JSScript*> script,
2468                                 HandleFunction functionProto)
2469 {
2470     if (!partiallyInit(cx, script, 1, 0, 0, 0, 0, 0, 0))
```

Aaaaaaaughghgh that parameter list is horrifying can I have

uint32_t numSlots = ..., numFixed = ..., ...;

and whatever and then pass in those locals as arguments.

---

```
2476     if (!functionProtoScope)
2477         return false;
2478     script->scopes()->vector[0].init(functionProtoScope);
2479
2480     if (!script->createScriptData(cx, 1, 1, 0))
```

Would prefer locals for these as well.

---

```
2530
2531     PositionalFormalParameterIter fi(script);
2532     while (fi && !fi.closedOver())
```

```
        fi++;
    script->funHasAnyAliasedFormal_ = !!fi;
```

Mild preference for !fi.done().

---

```

    script->isGeneratorExp_ = false;
    script->setGeneratorKind(NotGenerator);

    // Since modules are only run once. Mark the script so that initializers
```

s/. M/, m/

---

```
    script->bindingsAccessedDynamically_ = bce->sc->bindingsAccessedDynamically();
    script->hasSingletons_ = bce->hasSingletons;

    uint64_t nslots = bce->maxFixedSlots + bce->maxStackDepth;
    if (nslots > UINT32_MAX) {
```

This doesn't do what it was supposed to -- adding two uint32_t on a compiler where |unsigned| is a 32-bit integer will hit wraparound rather than ever exceed UINT32_MAX. (Good compilers would warn about this, dunno if any of ours would on this.) Cast at least one quantity to uint64_t to make this do what you wanted.

---

```
    RootedScope clone(cx);
    for (uint32_t i = scopes.length(); i < nscopes; i++) {
        original = vector[i];
        clone = Scope::clone(cx, original, scopes[FindScopeIndex(src, *original->enclosing())]);
        if (!clone || !scopes.append(clone))
```

Might be worth reserving |nscopes - scopes.length()| extra space at the start, then infallibly append after.

---

```
    AutoObjectVector objects(cx);
    if (nobjects != 0) {
        GCPtrObject* vector = src->objects()->vector;
        RootedObject obj(cx);
        RootedObject clone(cx);
```

This shadows the |clone| declared above. I'd put the /* Scopes */ section above into a block for visual and scoping separation.

---

```
            } else {
                clone = DeepCloneObjectLiteral(cx, obj, TenuredObject);
            }

            if (!clone || !objects.append(clone))
```

Again, could reserve/infallibleAppend.

---

```
            clone = FunctionScope::clone(cx, original.as<FunctionScope>(), fun, enclosingClone);
        else
            clone = Scope::clone(cx, original, enclosingClone);

        if (!clone || !scopes.append(clone))
```

reserve/infallibleAppend

---

```

    while (bottom < top) {
        size_t mid = bottom + (top - bottom) / 2;
        const ScopeNote* note = &notes->vector[mid];
        if (note->start <= offset) {
```

We should look in a followup if this can be converted to mozilla::BinarySearch{,If}.

---

```
        return nullptr;

    JSAtom** resClosedOverBindings = res->closedOverBindings();
    for (size_t i = 0; i < res->numClosedOverBindings(); i++)
        resClosedOverBindings[i] = closedOverBindings[i];
```

You could also do

std::copy(closedOverBindings.begin(), closedOverBindings.end(), resClosedOverBindings);

if you wanted. Can't for the loop just after, tho, because it's using init() and not assignment.

---

```
    // variables and inner functions.
    size_t i, num;
    JSAtom** closedOverBindings = res->closedOverBindings();
    for (i = 0, num = res->numClosedOverBindings(); i < num; i++)
        closedOverBindings[i] = dummyAtom;
```

std::fill(closedOverBindings, closedOverBindings + res->numClosedOverBindings(),
dummyAtom);

?

**js/src/jsscript.h**

```
    js::Scope* getScope(jsbytecode* pc) const {
        // This method is used to get a scope directly using a JSOp with an
        // index. To search through ScopeNotes to look for a Scope using pc,
        // use lookupScope.
        MOZ_ASSERT(containsPC(pc) && containsPC(pc + sizeof(uint32_t)));
```

Split into two asserts. Also, if the goal of the second is to assert the whole of |JSOp(*pc)| is in-bounds, shouldn't that be + 1, for op plus four bytes of immediate index?

---

```

    inline JSFunction* getFunction(size_t index);
    JSFunction* function() const {
        if (functionNonDelazifying())
            return functionNonDelazifying();
```

Mild preference for

if (JSFunction* fun = functionNonDelazifying())
return fun;

---

```
        uint32_t version : 8;

        uint32_t shouldDeclareArguments : 1;
        uint32_t hasThisBinding : 1;
        uint32_t numClosedOverBindings : 22;
```

Oh noes my functions with 4M upvars are going to break!!!1!!cos(0)

---

```
                                    uint32_t lineno, uint32_t column);

  public:
    static const uint32_t NumClosedOverBindingsLimit = 1 << 22;
    static const uint32_t NumInnerFunctionsLimit = 1 << 20;
```

Could the 20/22 here be moved above PackedView into constants that this could reuse? Not super-

happy about repeated magic constants in two different places forty lines apart.

**js/src/jsscriptinlines.h**

```
{
    js::Scope* scope = bodyScope();
    if (scope->is<js::FunctionScope>()) {
        if (scope->environmentShape())
            return scope->environmentShape();
```

if (Shape* envShape = scope->environmentShape())
return envShape;

**js/src/jsstr.cpp**

```
#include "js/UniquePtr.h"
#if ENABLE_INTL_API
#include "unicode/unorm.h"
#endif
#include "vm/EnvironmentObject.h"
```

Given you didn't have to change anything else in this file for the move/renames, is this #include even necessary? Please remove it if it's dead.

**js/src/octane/run-deltablue.js**

```
}


BenchmarkSuite.config.doWarmup = undefined;
BenchmarkSuite.config.doDeterministic = true;
```

Was this deliberate?

**js/src/shell/js.cpp**

```
         * coincides with the end of a line.
         */
        int startline = lineno;
        typedef Vector<char, 32> CharBuffer;
        RootedObject globalLexical(cx, &cx->global()->lexicalEnvironment());
```

Can we just get rid of this now, maybe?

**js/src/tests/ecma_6/LexicalEnvironment/block-scoped-functions-annex-b-eval.js**

```
  const a = 1;
}

function f3() {
  // As above, but for let.
```

How about the trifecta? (I have no idea what this should do, tho.)

function f4() {
  try {
    throw 17;
  } catch (a) {
    eval("{ function a() {} }");
  }
}

**js/src/vm/EnvironmentObject.cpp**

```
    MOZ_ASSERT(CanBeFinalizedInBackground(kind, &class_));
    kind = gc::GetBackgroundAllocKind(kind);

    RootedNativeObject obj(cx,
        MaybeNativeObject(JSObject::create(cx, kind, heap, shape, group)));
```

Not really necessary to Rooted here.

---

```
    return create(cx, object, enclosing, nullptr);
}

static inline bool
IsUnscopableDotName(JSContext* cx, HandleId id)
```

Argument order should be flipped, because this isn't fallible. Can defer to followup, as I'm guessing this was pre-existing (but can't tell because of no move-recognition).

```
    if (!obj->setQualifiedVarObj(cx))
        return nullptr;

    RootedObject globalLexical(cx, &cx->global()->lexicalEnvironment());
    obj->initEnclosingEnvironment(globalLexical);
```

Just pass in the pointer, no need to root.

```
    gc::AllocKind allocKind = gc::GetGCObjectKind(shape->numFixedSlots());
    MOZ_ASSERT(CanBeFinalizedInBackground(allocKind, &LexicalEnvironmentObject::class_));
    allocKind = GetBackgroundAllocKind(allocKind);
    RootedNativeObject obj(cx,
        MaybeNativeObject(JSObject::create(cx, allocKind, heap, shape, group)));
```

Shouldn't need the Rooted here.

```
    if (!obj)
        return nullptr;

    obj->initFixedSlot(lambdaSlot(), ObjectValue(*callee));
    return static_cast<NamedLambdaObject*>(obj);
```

And I guess as<>() doesn't work because NLO are just LEO with a lambda, right?

```
/* static */ size_t
NamedLambdaObject::lambdaSlot()
{
    // Decl env environments have exactly one name.
    return JSSLOT_FREE(&LexicalEnvironmentObject::class_);
```

Comment needs updating. And you should be able to have a static const in NLO for this, no?

```
        return nullptr;
    }

    /*
     * In theory, every function scope contains an 'arguments' bindings.
```

Not quite -- need to clarify that:

* arrow function scopes don't have an arguments binding

Also, these cases don't have an arguments *object*

* function scopes with |arguments| in the parameter names
* functions without any parameter expressions, that contain a function declaration/generator or a lexical declaration named |arguments|

It would appear this function, by name, indicates the *first* sense, not the sense of has-no-arguments-object.

So I guess adding in (non-arrow) would suffice to fix this.

```
     */
    static bool isMissingArgumentsBinding(EnvironmentObject& env)
    {
```

```
        return isFunctionEnvironment(env) &&
               !env.as<CallObject>
().callee().nonLazyScript()->argumentsHasVarBinding();
```

And I guess, given the one use later on, this should have a test added that callee isn't an arrow function. And this is visible, so probably defer to a followup fix, and include a debugger test that checks that arrow functions lack an 'arguments' binding when observed by debugging.

---

```
    }

    /*
     * Similar to 'arguments' above, we don't add a
'this' binding to functions
     * if it's not used.
```

(and again, arrow functions don't have a 'this' binding)

Also I see isFunctionEnvironmentWithThis claims generator expression lambdas don't, either (non-standard).

---

```
     */
    static bool isMissingThisBinding(EnvironmentObject&
env)
    {
        return isFunctionEnvironmentWithThis(env) &&
               !env.as<CallObject>
().callee().nonLazyScript()->functionHasThisBinding();
```

So, hm. I *think* this definition is wrong in that it claims arrow functions "isMissingThisBinding". But they're not supposed to have one. (And genexp lambdas, non-standardly, per above.) So the name seems wrong. Maybe |thisBindingOptimizedOut|? And I guess |argumentsBindingOptimizedOut| for the other function, for consistency.

---

```
     * the debugger requests 'arguments' for a function
scope where the
     * arguments object has been optimized away (either
because the binding is
     * missing altogether or because
!ScriptAnalysis::needsArgsObj).
     */
    static bool isMissingArguments(JSContext* cx, jsid
id, EnvironmentObject& env)
```

I might rename this to isMissingArgumentsObject -- seems clearer. And I think that comports with the one use.

Or, going with the "optimized out" theme, argumentsObjectOptimizedOut?

---

```
    {
        return isArguments(cx, id) &&
isFunctionEnvironment(env) &&
               !env.as<CallObject>
().callee().nonLazyScript()->needsArgsObj();
    }
    static bool isMissingThis(JSContext* cx, jsid id,
EnvironmentObject& env)
```

And then thisOptimizedOut.

---

```
        bool isMagic = v.isMagic() && v.whyMagic() ==
JS_OPTIMIZED_ARGUMENTS;

#ifdef DEBUG
        // The |env| object here is not limited to
CallObjects but may also
        // be block scopes in case of the following:
```

s/block scopes/lexical envs/ ?

---

```
        Rooted<EnvironmentObject*> env(cx, &proxy->as<DebugEnvironmentProxy>().environment());

        if (isMissingArgumentsBinding(*env)) {
            if (!props.append(NameToId(cx->names().arguments)))
                return false;
```

Hmm, so I guess this means arrow functions wrongly appear to have an |arguments| binding, to the debugger? Followup to fix?

---

```
    EnvironmentObject& e = environment();
    return e.is<CallObject>() ||
           e.is<VarEnvironmentObject>() ||
           e.is<ModuleEnvironmentObject>() ||
           e.is<LexicalEnvironmentObject>();
```

NLO is declarative, right? And the last check here *will* subsume it, right? Maybe worth a comment.

---

```
        if (!frame.environmentChain()->is<CallObject>())
            return;

        if (frame.isFunctionFrame() && frame.callee()->isGenerator())
            return;
```

I'm probably being dumb, when is it not isFunctionFrame() here?

---

```
void
DebugEnvironments::onPopWith(AbstractFramePtr frame)
{
    DebugEnvironments* envs = frame.compartment()->debugEnvs;
    if (envs)
```

Combine decl/if?

---

```
void
DebugEnvironments::onCompartmentUnsetIsDebuggee(JSCompartment* c)
{
    DebugEnvironments* envs = c->debugEnvs;
    if (envs) {
```

Combine decl/if?

---

```
{
#ifdef DEBUG
    for (size_t i = 0; i < chain.length(); ++i) {
        assertSameCompartment(cx, chain[i]);
        MOZ_ASSERT(!chain[i]->is<GlobalObject>());
```

Oh hey, so if we have this, *definitely* WithEnvironment::THIS_SLOT always mirrors OBJECT_SLOT. Nice to see this assertion's already in place, pleasant surprise!

---

```
            return true;
        }

        if (!script->functionHasThisBinding()) {
            res.setMagic(JS_OPTIMIZED_OUT);
```

I would have thought this would happen just after the hasLexicalThis check above, and we'd just continue to the enclosing function or so. But it was this way before, so I guess this is right...

---

```
            return false;
    }

    for (; bi; bi++) {
        name = bi.name()->asPropertyName();
```

Was there a BindingKindIsLexical function we could assert here on |bi|?

**js/src/vm/EnvironmentObject.h**

```
47 * environment chain (that is, fp->environmentChain() or
48 * fun->environment()).
49 *
50 * Environments may be syntactic, i.e., corresponding to
source text, or
51 * non-syntactic, i.e., specially created by embedding.
```

by the embedding

---

```
91 *    of those objects is wrapped by a
WithEnvironmentObject.
92 *
93 *    The innermost object passed in by the embedding
becomes a qualified
94 *    variables object that captures 'var' bindings.
That is, it wraps the
95 *    holder object of 'var' bindings.
```

Would be nice to say the vector is from outermost to innermost, i.e. the *last* object's environment shadows all other objects' environments, and it holds 'var' bindings. Right now innermost/outermost doesn't clarify what the ordering is.

---

```
93 *    The innermost object passed in by the embedding
becomes a qualified
94 *    variables object that captures 'var' bindings.
That is, it wraps the
95 *    holder object of 'var' bindings.
96 *
97 *    Does not hold 'let' or 'const' bindings.
```

Maybe

WithEnvironment doesn't hold 'let' or 'const' bindings -- see LexicalEnvironmentObject below.

or some sort of thing that explains what happens instead of this possible mis-assumption.

---

```
103 *    object. While any object can be made into a
qualified variables object,
104 *    only the GlobalObject and
NonSyntacticVariablesObject are considered
105 *    unqualified variables objects.
106 *
107 *    Unlike WithEnvironmentObjects, this object is
itself the holder of 'var'
```

Unlike WEO that delegate to the object they wrap, ...

?

---

```
135 *   Global lexical scope
136 *       |
137 *   WithEnvironmentObject wrapping FakeBackstagePass
138 *       |
139 *   LexicalEnvironmentObject
```

For each of these, the bottom LEO is provided by the parser, not by explicit action of the embedder, right? It might be worth noting that explicitly, and that embeddings don't do anything to provide that LEO.

I have some hesitance about documenting these users *inside the JS engine*. I'd prefer if this documentation were in each user, and if those users could be found by searching for easily-found uses of a particular JS symbol. I guess as long as we have people hand in a *vector* of objects for a scope chain, rather than requiring them to build it up themselves, we can't really have this. :-(

---

```
167 *
168 * D. XBL
```

```
 *
 * XBL methods are compiled as functions with XUL elements on the env chain.
 * For a chain of elements e0,...,eN:
```

Seems to me this isn't just XBL, is it? Pretty sure DOM event-handler attributes like onclick and such are handled this way, with the scope looking like

global
|
lexical
|
<html> element
|
<body>
|
<form>
|

for a DOM like

<html><body><form>...

So maybe a case E needs to be added, similar to what XBL has -- or the two could be combined some.

---

```
        return getReservedSlot(ENCLOSING_ENV_SLOT).toObject();
    }

    void initEnclosingEnvironment(JSObject* enclosing) {
        initReservedSlot(ENCLOSING_ENV_SLOT, ObjectOrNullValue(enclosing));
```

Can't we init with a reference, because non-null as the comment above says? Or deal with this in a followup, perhaps.

---

```
        initReservedSlot(ENCLOSING_ENV_SLOT, ObjectOrNullValue(enclosing));
    }

    // Get or set a name contained in this environment.
    const Value& aliasedBinding(EnvironmentCoordinate ec) {
```

const HeapSlot& or a better type?

---

```
    /*
     * When an aliased formal (var accessed by nested closures) is also
     * aliased by the arguments object, it must of course exist in one
     * canonical location and that location is always the CallObject. For this
     * to work, the ArgumentsObject stores special MagicValue in its array for
```

stores *a* special MagicValue? (and which one?)

---

```
     * When an aliased formal (var accessed by nested closures) is also
     * aliased by the arguments object, it must of course exist in one
     * canonical location and that location is always the CallObject. For this
     * to work, the ArgumentsObject stores special MagicValue in its array for
     * forwarded-to- CallObject variables. This MagicValue's payload is the
```

spurious space

---

```
        initReservedSlot(THIS_VALUE_OR_SCOPE_SLOT, PrivateGCThingValue(scope));
```

```
    }

    void initScope(LexicalScope* scope) {
        MOZ_ASSERT(!isGlobal() && isSyntactic());
```

Split into two asserts for more pointed failure messages.

---

```
    // For non-extensible lexical environments, the LexicalScope that created
    // this environment. Otherwise asserts.
    LexicalScope& scope() const {
        Value v = getReservedSlot(THIS_VALUE_OR_SCOPE_SLOT);
        MOZ_ASSERT(!isExtensible() && v.isPrivateGCThing());
```

split

---

```
// assignments and unqualified bareword assignments. Its parent is always the
// global lexical environment.
//
// This is used in ExecuteInGlobalAndReturnScope and sits in front of the
// global scope to capture 'var' and bareword asignments.
```

s/to.+assignments/to store 'var' bindings, and to store fresh properties created by assignments to undeclared variables./ ?  "bareword assignments" is a bit terse for this oddballness, IMO.

---

```
    /* Return the 'o' in 'with (o)'. */
    JSObject& object() const;

    /* Return object for GetThisValue. */
    JSObject* withThis() const;
```

As mentioned on IRC, I think -- as long as users aren't providing a scope-chain vector containing a global object (i.e. is<GlobalObject>()) -- we can get rid of this, because object() and *withThis() are identical.  File a bug to kill off this unnecessary complexity.

---

```
    //
    // This logic must be in sync with the HAS_INITIAL_ENV logic in
    // InitFromBailout.

    // If a function frame's CallObject, if present, is always the initial
```

An extra if in here.

---

```
    // InitFromBailout.

    // If a function frame's CallObject, if present, is always the initial
    // environment.
    if (mozilla::IsSame<SpecificEnvironment, CallObject>::value)
```

There's enough IsSame action in this function, that it seems to me probably better to have specializations for each possible SpecificEnvironment.

```
template<class SpecificEnvironment>
inline bool
IsFrameInitialEnvironment(AbstractFramePtr frame, SpecificEnvironment& env)
{
   return false;
}

template<>
inline bool
```

```
IsFrameInitialEnvironment(AbstractFramePtr frame,
CallObject& env)
{
   return true;
}

template<>
inline bool
IsFrameInitialEnvironment(AbstractFramePtr frame,
VarEnvironmentObject& env)
{
   return frame.isEvalFrame();
}

template<>
inline bool
IsFrameInitialEnvironment(AbstractFramePtr frame,
NamedLambdaObject& env)
{
   if (frame.isFunctionFrame() &&
      ...)
   {
      ...
   }

   return false;
}
```

That reads a bit clearer to me, IMO, than one large function with bunches of dead code in it per specialization.

**js/src/vm/Interpreter .cpp**

```
{
#ifdef DEBUG
    // Pop block from scope chain.
    Scope* scope = script->lookupScope(REGS.pc);
    MOZ_ASSERT(scope && scope->is<LexicalScope>() &&
scope->as<LexicalScope>().hasEnvironment());
```

Split this up into multiple asserts.

---

```
CASE(JSOP_POPVARENV)
{
#ifdef DEBUG
    Scope* scope = script->lookupScope(REGS.pc);
    MOZ_ASSERT(scope && scope->is<VarScope>() && scope-
>as<VarScope>().hasEnvironment());
```

Split up.

**js/src/vm/Runtime.h**

```
    unsigned keepAtoms_;
    friend class js::AutoKeepAtoms;
  public:
    bool keepAtoms() {
        MOZ_ASSERT(CurrentThreadCanAccessRuntime(this));
```

This removal seems somewhat strange, for what this bug/patch are doing.

**js/src/vm/Scope.cpp**

```
#include "builtin/ModuleObject.h"
#include "gc/Allocator.h"
#include "vm/EnvironmentObject.h"
#include "vm/Runtime.h"
#include "vm/Shape-inl.h"
```

Blank line before this -inl, right?

---

```

using mozilla::Maybe;
using mozilla::Some;
using mozilla::Nothing;

```

```
using mozilla::MakeScopeExit;
```

Alphabetize.

---

```
    unsigned attrs = JSPROP_PERMANENT |
JSPROP_ENUMERATE;
    switch (bindKind) {
      case BindingKind::Const:
      case BindingKind::NamedLambdaCallee:
        attrs |= JSPROP_READONLY;
```

This

```
    if (bindKind == BindingKind::Const || bindKind ==
BindingKind::NamedLambdaCallee)
        attrs |= JSPROP_READONLY;
```

seems a lot more readable.

---

```
    for (; bi; bi++) {
        BindingLocation loc = bi.location();
        if (loc.kind() ==
BindingLocation::Kind::Environment) {
            name = bi.name();
            shape = NextEnvironmentShape(cx, name,
bi.kind(), loc.slot(), stackBase, shape);
```

Given there's only one caller, I think you inline this. You have push/pop overhead for a Rooted<StackShape>, recomputation of |base| every time, etc. with the current split, for no reason. At a minimum, get rid of the recomputation overheads, and make NextEnvironmentShape a lambda inside CreateEnvironmentShape.

---

```
}

template <typename ConcreteScope>
static typename ConcreteScope::Data*
CopyScopeData(ExclusiveContext* cx, Handle<typename
ConcreteScope::Data*> data)
```

The pervasive need to manually js_free ConcreteScope::Data* on failure to allocate a Scope to take ownership of it in this file is unfortunate. I vaguely recall you griping about this on IRC.

I guess let's run with this for now, but this function *really* feels like it should be able to return UniquePtr<ScopeData> where we have DeletePolicy<ScopeData>::operator()(Scope* p) { js_free(p); } and ScopeData is an empty base class of all ConcreteScope::Data. And then Scope::create could take that same type, and we'd avoid uintptr_t for an argument, and we could kill off the manual js_free.

---

```
CopyScopeData(ExclusiveContext* cx, Handle<typename
ConcreteScope::Data*> data)
{
    size_t dataSize = ConcreteScope::sizeOfData(data-
>length);
    uint8_t* copyBytes = cx->zone()->pod_malloc<uint8_t>
(dataSize);
    if (!copyBytes) {
```

Seems like this should reuse NewEmptyScopeData so the memory-allocating is in only one place.

---

```
{
    uint8_t* bytes = cx->zone()->pod_calloc<uint8_t>
(ConcreteScope::sizeOfData(length));
    if (!bytes)
        ReportOutOfMemory(cx);
    return reinterpret_cast<typename
ConcreteScope::Data*>(bytes);
```

Seems like you could move |data->length = length;| into this function and save some trouble for a bunch of callers.

```
        if (!envShape)
            return nullptr;
    }

    uintptr_t dataClone = 0;
```

Eeeeeugh, and the manual js_free in case of error is equally so.

```
void
Scope::dump()
{
    for (ScopeIter si(this); si; si++) {
        fprintf(stderr, "%s [%p]", ScopeKindString(si.kind()), si.scope());
```

static_cast<void*>(si.scope()) to avoid warnings with some compilers. "%p" matches only void*, not T* for any T.

```
          case ScopeKind::ParameterExpressionVar:
          case ScopeKind::Global:
          case ScopeKind::NonSyntactic:
            return scope;
          default:
```

Would prefer listing out the others here.

```
    }
}

BindingIter::BindingIter(JSScript* script)
  : BindingIter(script->bodyScope())
```

Ooh, nice, delegating constructor use.

```
    // Named lambda scopes can only have environment slots. If the callee
    // isn't closed over, it is accessed via JSOP_CALLEE.
    if (flags & IsNamedLambda) {
        // Named lambda binding is weird. Normal BindingKind ordering rules
        // don't apply.
```

lol

```
        //             consts - [data.constStart, data.length)
        init(0, 0, 0, 0, data.constStart,
             CanHaveFrameSlots | CanHaveEnvironmentSlots | flags,
             firstFrameSlot, JSSLOT_FREE(&LexicalEnvironmentObject::class_),
             data.names, data.length);
```

These init function-calls are going to be fun to read, aren't they. :-\ Guess it can't be helped.

**js/src/vm/Scope.h**

```
      : bits_(uintptr_t(name) | (closedOver ? ClosedOverFlag : 0x0))
    { }

    JSAtom* name() const {
        return (JSAtom*)(bits_ & ~FlagMask);
```

static_cast<> or reinterpret_cast<>

```
          case ScopeKind::With:
          case ScopeKind::Global:
          case ScopeKind::NonSyntactic:
            return true;
          default:
```

I'd prefer if we listed all the kinds here, rather than just have a default that anonymously sweeps up everything.

---

```
    uint32_t environmentChainLength() const;

    template <typename T>
    bool hasOnChain() const {
        for (const Scope* it = this; it; it = it->enclosing()) {
```

It might be worth noting we can't (re)use hasOnChain(T:classScopeKind_) here because not all T have their own kind, and is<T>() is specialized. I was going to suggest that change until I read further/harder.

---

```
};

//
// A lexical scope that holds let and const bindings. There are 3 kinds of
// LexicalScopes.
```

...and yet you only list two. :-) I assume you're missing NamedLambda (and StrictNamedLambda that you conflate with it)?

---

```

    static uint32_t nextFrameSlot(Scope* start);

  public:
    uint32_t firstFrameSlot() const;
```

Somewhere you need to document what "slot" means: the index of a binding within a frame. You should also document that the maximum number of slots a single frame can have is LOCALNO_LIMIT, and that while the frontend may temporarily produce setups that have more than that number of frame slots, eventually EmitterScope::checkSlotLimits will enforce this limit. I can't find documentation of this anywhere, and I had to sort of piece it together as I read various parts of this patch.

---

```
}

//
// Scope corresponding to a function. Holds var bindings and formal parameter
// names.
```

This is a bit reductive, in the presence of parameters containing any expressions. And examples would say a thousand words. How about,

"""
Holds formal parameter names and, if the function parameters contain no expressions that might possibly be evaluated, the function's var bindings. For example, in these functions, the FunctionScope will store a/b/c bindings but not d/e/f bindings:

```
function f1(a, b) {
  var c;
  let e;
  const f = 3;
}
function f2([a], b = 4, ...c) {
  var d, e, f; // stored in VarScope
}
```
"""

---

```
        // scope.
```

```
        uint32_t nextFrameSlot;

        // Frame slots [0, varFrameSlotEnd) are vars.
        uint32_t varFrameSlotEnd;
```

This field and all its users are dead, ergo should be removed. The emitter cares about what this is computing, so it should be computing it.

---

```
    BindingIter(EvalScope::Data& data, bool strict) {
        init(data, strict);
    }

    explicit BindingIter(const BindingIter& bi)
```

Can this just be

explicit BindingIter(const BindingIter&) = default;

?

---

```
{
    Scope* scope_;

  public:
    explicit ScopeIter(Scope* scope)
```

I think this could take |const Scope*|, and if it did you wouldn't need a const_cast<> in Scope::chainLength.

**js/src/vm/ScopeObject.cpp**

```
/* -*- Mode: C++; tab-width: 8; indent-tabs-mode: nil; c-basic-offset: 4 -*-
```

This file wasn't really "removed", it was moved to EnvironmentObject.cpp and substantially changed. So, there's probably some reasonable portion of it that's basically the same as it used to be.

Given that, you should inform hg that you moved this file, so that hg blame and such will annotate line-changes across the move. git doesn't have any concept of a file move, unfortunately. :-( So you'll want to import this patch into hg, then use

hg mv --after ScopeObject.cpp EnvironmentObject.cpp

and so on for the other file-moves (ScopeObject.h, ScopeObject-inl.h) to inform hg about what you've done, and only then land the patch.

**js/src/vm/Stack.cpp**

```
InterpreterRegs::setToEndOfScript()
{
    sp = fp()->base();
    // TODOshu see if debugger fails
    // pc = fp()->script()->lastPC();
```

???

**js/src/vm/Stack.h**

```

        RESUMED_GENERATOR      =        0x2,  /* frame is for a resumed generator invocation */

        /* Function prologue state */
        HAS_INITIAL_ENV        =        0x4,  /* call obj created function or var env for eval */
```

Should "call obj created function" be "call-obj" or something? This is hard to read, and thus to understand, with "call" and "created" both possibly reading as verbs.

**js/xpconnect/loader/mozJSComponentLoader .cpp**

```

```

```
931     aTableScript.set(tableScript);
932
933
934     {   // Environment for AutoEntryScript
```

This should still be scope.

**Flags:** **Requestee:**

| | | | | |
|---|---|---|---|---|
| shu: | review | ? | jorendorff@mozilla.com | suggested reviewers ▾ |
| shu: | review | ? | jwalden+bmo@mit.edu | suggested reviewers ▾ |
| shu: | review | ? | efaustbmo@gmail.com | suggested reviewers ▾ |
| | superreview | | | |
| | a11y-review | | | |
| | ui-review | | | |
| | feedback | | | |
| | approval-mozilla-aurora | | | |
| | approval-mozilla-beta | | | |
| | approval-mozilla-release | | | |
| | approval-mozilla-esr45 | | | |
| | approval-mozilla-b2g37 | | | |
| | approval-mozilla-b2g48 | | | |
| | approval-mozilla-b2g37_v2_2r | | | |
| | approval-mozilla-b2g44 | | | |
| | sec-approval | | | |
| | qa-approval | | | |
| | checkin | | | |
| addl. | review | | | |

Publish Cancel

Powered by Splinter