ClojureCLR Compiler Notes

2025.01.04

# Getting to the compiler

From the REPL:

clojure.main/repl 🡪 clojure.core/eval 🡪 clojure.lang.Compiler.eval

From code:

clojure.core/load 🡪 clojure.lang.RT.load

clojure.core/compile 🡪 clojure.core/load-one 🡪 clojure.core/load

clojure.lang.RT.load

* Try to find assembly (newer than equivalent source code)
  + Load the assembly
* Try to find source file
  + If \*compile-files\* is true: RT.Compile.
  + Otherwise RT.LoadScript
* Try to find loaded assembly with a class named \_\_Init\_\_$*sourcePath*
  + Call \_\_Init\_\_$*sourcePath*.Initialize
* Try to find a loaded assembly with *sourcePath*.cljr.dll (or cljc or clj) as an embedded resource
  + Load that Assembly and call its \_\_Init\_\_$*sourcePath*.Initialize

clojure.lang.RT.Compile => clojure.lang.Compiler.Compile

clojure.lang.RT.LoadScript => clojure.lang.Compiler.load

# Compiling

clojure.lang.Compiler.Compile

* Error if \*compile-path\* is not true
* Create GenContext with an external assembly
* Do the compile (See next).
* Save the assembly

clojure.lang.Compile.Compile (called from above)

* Generate a loader class to hold the initializer: ObjExpr objx = new ObjExpr(null);
* Augment the GenContext with that class (.WithTypeBuilder)
* Create a MethodBuilder for the Initialize method
* Set up the thread bindings for the all the compilation environment
* Call Compile1 on each form in the file being compiled
* Do cleanup on the ObjExpr:
  + Emit constants
  + Generate a constructor
  + Add custom attribute (not sure what this is for)
  + Create the type

clojure.lang.Compile.Compile1

* Macroexpand the form
* If the form looks like (do x y z …), call Compile1 on each of x, y, z, …
* Otherwise, Analyze as RHC.Eval the form, getting an Expr
* Move the keywords, vars, constants from the Expr to the initializer objx
* Emit the constant field defs into the typebuilder for the initializer class
* Emit the expr as RHC.Expression into the assembly
* Add an Opcodes.Pop at the end.
* Call expr.eval() to get the form evaluated in the runtime context.

# Loading

clojure.lang.Compiler.load

* Set up the thread binding for the compiler context (many fewer than Compile() does)
* Iterate through all forms in the file, calling clojure.lang.Compiler.eval on each.

clojure.lang.Compiler.eval(form):

* Macroexpand the form
* If the form looks like (do … ): eval each form in the body, return the last
* If the form is an IType (only types defined by deftype have this marker) or it is (x y z …) where x is not a Symbol starting with “def” –
  + Analyze as RHC.Expression: (fn [] x y z), returns an ObjExpr objx.
  + objx.Eval() => an IFn fn
  + fn.invoke()
* otherwise
  + Analyze as RHC.Eval the form getting back an Expr.
  + Return expr.eval()

# Analyze

Takes in a form and translates it into an AST, represented by the Expr that is its root. It’s mostly a big case statement based on what the form looks like.

If the form is a LazySeq, realize it. If empty, use an empty list. Transfer the meta-information to the resulting value.

Then switch based on the form that we now have:

* Null 🡪 NilExprInstance
* Boolean value 🡪 TrueExprInstance or FalseExprInstance, depending
* Symbol 🡪 call AnalyzeSymbol
* Keyword 🡪 call RegisterKeyword
* IsNumeric 🡪 NumberExpr.Parse
* String 🡪 StringExpr
* IPersistentCollection, but not an IRecord, IType, and the count is 0 🡪 EmptyExpr, but wrap in a MetaExpr if the form has meta-information.
* ISeq 🡪 call AnalyzeSeq
* IPersistentVector 🡪 VectorExpr
* IRecord 🡪 ConstantExpr
* IType 🡪 ConstantExpr
* IPersistentMap 🡪 MapExpr
* IPersistentSet 🡪 SetExpr
* Otherwise 🡪 ConstantExpr (and pray it is something that ConstantExpr can handle)

Most of these are straightforward, except for AnalyzeSymbol and AnalyzeSeq (and a little bit of RegisterKeyword).

## AnalyzeSymbol

There are some interesting side-effects that can come from this.

One must know that there are some Vars that have thread-local bindings used to keep certain pieces of information available. These include:

* LocalEnvVar,: (maybe) holds a map from Symbols to LocalBindings
* MethodVar: (maybe) holds an ObjMethod, the method we are currently compiling. The local bindings will be local to this method.

We’ll analyze these in more detail later.

Steps in AnalyzeSymbol:

* If the symbol does not have namespace, check LocalEnvVar to see if the Symbol is for a local variable. (namespace-qualified symbols are never locals) If so, there can be side-effects:
  + If the local binding has index 0, then the local is ‘this’. Set the UsesThis flag on the ObjMethod to true.
  + Add the local binding to the ClosesOver list for the method. Recurse up the chain of parent methods (nesting is possible).
  + Also perhaps note that binding’s index might be added to the method’s LocalsUsedInCatchFinally list. (More on this later, too.)
* Check to see if the Symbol names a type. (This is more complicated than you might think. We have to make sure that the symbol’s namespace is not an actual namespace, or an alias for another namespace in the current namespace. Also, the Symbol’s name can’t start with a positive digit – that’s reserved for something else.) Call HostExpr.MaybeType (more later) to see if the namespace maps to a type. If so, there are three possibilities:
  + The symbol’s name is a static field in that type: Return a StaticFieldExpr.
  + The symbol’s name is a static property in that type: Return a StaticPropertyExpr
  + Otherwise: Return a QualifiedMethodExpr
* If we get here, we are not a type or a local binding. Call Compiler.Resolve, which does the following. There are two branches of analysis, depending on whether the symbol has a namespace or not.
  + With namespace:
  + If the symbol designates an array type (e.g., String/2), return the type.
  + Look for a Var interned with this name in the current namespace.
    - If no Var: error
    - If the Var has a difference namespace from the current namespace and it not public: error
    - Otherwise: return the associated Var
  + Without namespace:
  + If the symbol has a ‘.’ in it, but not as the first character, or the name ends with ‘]’, try to find a type with that name. Return the type or null.
  + If the symbol is ‘ns’, return the Var for ‘ns’.
  + If the symbol is ‘in-ns’, return the Var for ‘in-ns’
  + If the symbol is name of the compileStub (more on this, much later), return the compileStubClassVar.
  + See if the current namespace has a mapping. If it does, return the mapped value. If not, throw an error, unless \*allow-unresolved-vars\* is true, in which case just return the symbol. (\*allow-unresolved-vars\* supposedly was introduced for some early version of ClojureScript. Not sure why we would bother anymore.)
* We have the following possible returns from Compiler.Resolve.
  + A Var – if this is a macro – error. If the var is marked as ‘const’, say it has value X, then Analyze (quote X), which will use the ConstantExpr.Parser to generate a constant. Otherwise, register the Var (check the VarsVar to see if it already registered, if not, add it to that map, and call RegisterConstant on the Var. A lot more on this later.) Return a VarExpr.
  + A Type – return a ConstantExpr for the type.
  + A Symbol – this only happens when \*allow-unresolved-vars\* is true. Return an UnresolvedVarExpr.
  + Otherwise: Resolve failed. Throw error “Unable to resolve symbol”

## AnalyzeSeq

We have something (x y z … )

* Macro-expand the form. If we get back something different, call Analyze recursively on that thing.
* Macro-expansion didn’t change it, so we are still (x y z …). Let’s rename this (op x y …). We want to discriminate based on op.
* Op is null – error “Can’t call nil”
* Check if op is marked as inline.
  + If op is a symbol used as a local variable, it is not inline.
  + If op is a Var or op is a Symbol mapped to a not-private Var in the current namespace and the var has :inline metadata and the number of items in the form after the op is one of the list inline arities, then it is inline.
  + If it is inline, apply the inline form to the parameters and Analyze the result.
* If op = fn\*, parse an FnExpr.
* If op is a special form (def, loop\*, recur\*, let\*, if, … ), call the parser for that special form.
* Otherwise: call InvokeExpr.Parse.

# Compiler state

The compiler maintains some of its state in dynamically bound Vars. In addition, in many places a GenContext is passed around. Similarly, an ObjExpr is passed around.

Compiler.load() establishes the following dynamic Var bindings:

SourcePathVar, relativePath,  
SourceVar, sourceName,  
RT.ReadEvalVar, true   
RT.CurrentNSVar, RT.CurrentNSVar.deref(),  
RT.UncheckedMathVar, RT.UncheckedMathVar.deref(),  
RT.WarnOnReflectionVar, RT.WarnOnReflectionVar.deref(),  
RT.DataReadersVar, RT.DataReadersVar.deref(),  
MethodVar, null,  
LocalEnvVar, null,  
LoopLocalsVar, null,  
NextLocalNumVar, 0,  
CompilerContextVar, EvalContext,  
CompilerActiveVar, false

Compiler.compile() does all those bindings and adds:

ConstantsVar, PersistentVector.EMPTY,  
ConstantIdsVar, new IdentityHashMap(),  
KeywordsVar, PersistentHashMap.EMPTY,  
VarsVar, PersistentHashMap.EMPTY,

Compiler.compile() establishes different bindings for two of them:

CompilerContextVar, context,  
CompilerActiveVar, true

Certain of these Vars are part of the Clojure ecosystem.

|  |  |
| --- | --- |
| RT.ReadEvalVar | clojure.core/\*read-eval\* |
| RT.CurrentNSVar | clojure.core/\*ns\* |
| RT.UncheckedMathVar | clojure.core/\*unchecked-math\* |
| RT.WarnOnReflectionVar | clojure.core/\*warn-on-reflection\* |
| RT.DataReadersVar | clojure.core/\*data-readers\* |
| SourcePathVar | clojure.core/\*file\* |
| SourceVar | clojure.core/\*source-path\* |

Establishing thread-local bindings for these prevents changes to root values having an effect during loading/compiling.

Don’t stare too long at the last two. If they seem backwards, well, yes. Don’t ask me. That’s the way it was in the Java code. \*file\* has a doc-string associated with it: "The path of the file being evaluated, as a String. When there is no file, e.g. in the REPL, the value is not defined." \*source-path\* does not. Clojuredocs.org has an entry, but the official data is blank. Someone did make a note: “Contains the name (not the full path, for that see https://clojuredocs.org/clojure.core/\*file\*) of the compilation unit that is currently being compiled.”

For Compiler.load(), the only non-standard (not part of clojure.core, and hence specific to the compile/load activity) Vars are CompilerContextVar and CompilerActiveVar. The value of CompilerContextVar will be a GenContext, to be described below. CompilerActiveVar is true/false.

In addition, there are a number of Vars that are used during compilation that are dynamically bound only when needed:

InTryBlockVar   
InCatchFinallyVar   
MethodReturnContextVar   
NoRecurVar  
KeywordCallsitesVar  
ProtocolCallsitesVar  
VarCallsitesVar

The following are not in the JVM version, just the CLR version.

CompileStubSymVar   
CompileStubClassVar   
CompileStubOrigClassVar  
CompilingDefTypeVar

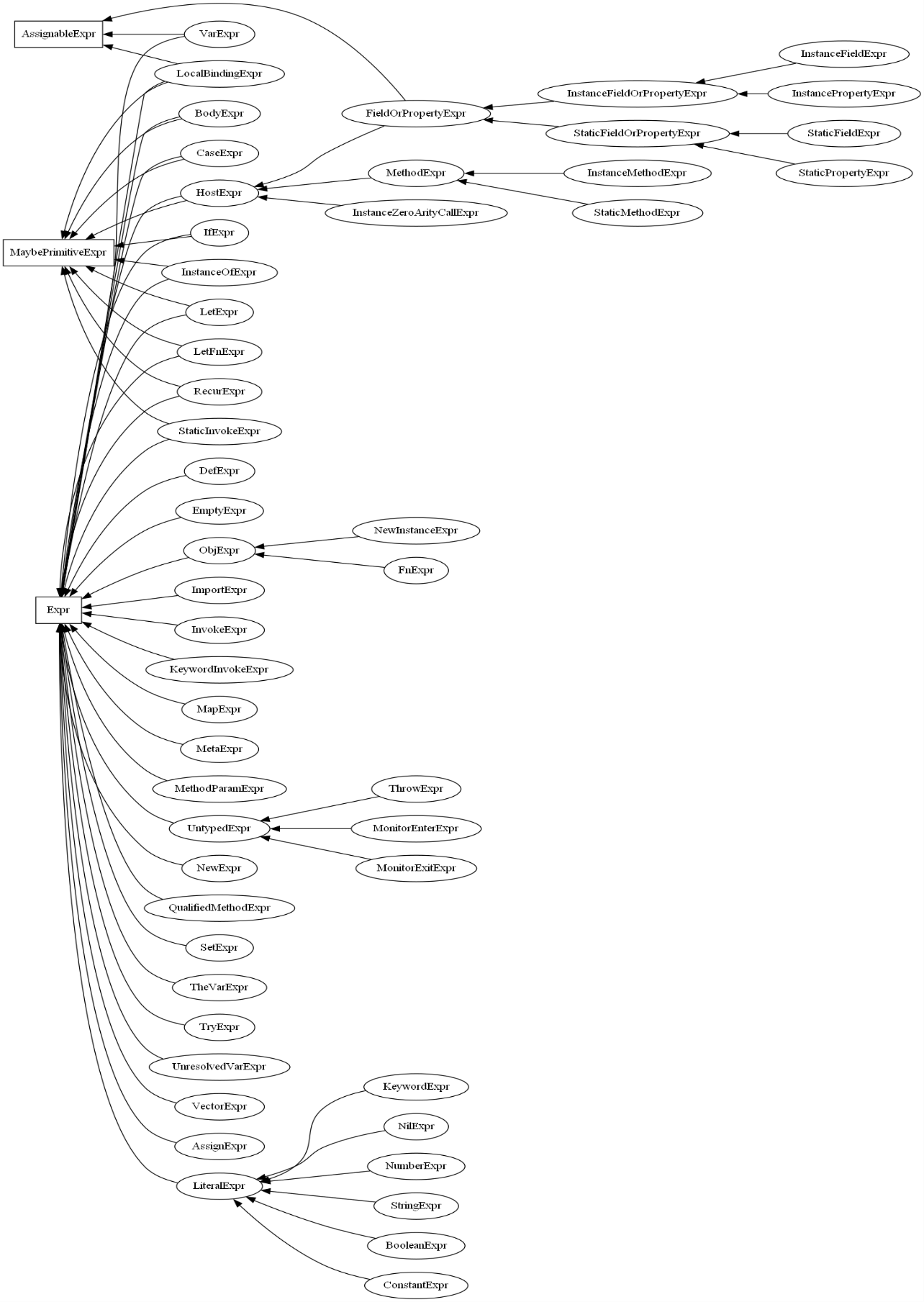
I won’t attempt to define all of these here. Clearly we have a lot of state being passed around in a very distributed fashion.

The most important of these are MethodVar and CompilerContextVar.

MethodVar is set when we are in the process of compiling an fn\* form, i.e., a function definition. Specifically, when we are compiling a specific arity overload of the fn. Typically, this is accessed to get information such as the set of local variables.

CompilerContextVar holds a GenContext. A GenContext holds the context of code generation-related information, such as the AssemblyBuilder, ModuleBuilder, TypeBuilder, etc., currently in use.

# Expr types



MaybePrimitiveExpr is a marker interface. Expr is the root of all evil.

public interface Expr  
 {  
 bool HasClrType { get; }  
 Type ClrType { get; }  
 object Eval();  
 void Emit(RHC rhc, ObjExpr objx, CljILGen ilg);  
 bool HasNormalExit();  
 }

HasClrType and ClrType are used to indicate the Type of the result of the expression. One is not supposed to call ClrType unless HasClrType is true. Eval() is used when the AST is called to evaluate the form. Emit() is used generate IL code using the provided CLjILGen object (an overload on the regular ILGen).

AssignExpr defines methods EvalAssign and EmitAssign, for use when the set! is applied to the thing.

Generally, the classes derived from Expr have a similar form:

* A static Parse method used to parse the form in question. This does a bunch of work and then finally calls the constructor to create the node.
* Fields holding the relevant data
* A constructor. Usually one does not call the constructor directly – we work through the Parse method.
* Definitions of the Expr methods.

The Parse methods mostly have the signature: Expr Parse(ParserContext pcon, object form). The ParserContext holds an RHC and flag indicating whether we are in an assignment context. In the original Clojure(JVM) compiler, the only thing passed was the RHC. In ClojureCLR, I found it useful to pass an extra flag.

The RHC = “Rich Hickey Constant” is an enum defined as

public enum RHC  
 {  
 Statement, // value ignored  
 Expression, // value required  
 Return, // tail position relative to enclosing recur frame  
 Eval  
 }

(Just called Compiler.C in Clojure(JVM).) RHC.Statement vs RHC.Expression mostly matters during code generation – the Emit method takes an RHC. For many types of expression, generating the code under RHC.Statement means the value will not be used, so a Pop instruction is issued to clear the expression’s result from the operand stack.

The value RHC.Eval is a little more interesting. It indicates that we are in an evaluation context, meaning that we will eventually execute the AST using the Eval method on the root node. There are some node types that cannot be evaluated; these are often types that have complicated control flow. Often the solution in the parser is to wrap the entire form inside an fn an call Analyze on that instead. (You also see situations where the RHC.Eval is switched to RHC.Expression on subform analysis.) Here is an example from LetExpr.Parse:

if (pcon.Rhc == RHC.Eval  
 || (pcon.Rhc == RHC.Expression && isLoop))  
 return Compiler.Analyze(  
 pcon,   
 RT.list(RT.list(Compiler.FnOnceSym,   
 PersistentVector.EMPTY,   
 form)),   
 "let\_\_" + RT.nextID());

There are many classes derived from Expr. Fortunately, a considerable number of them are not too complicated. Consider VectorExpr. One will be created because a vector was in the source code, for example, if we were compiling [x (+ y 12) z] – perhaps the last form in a function that is returning a vector of values. VectorExpr.Parse iterates through the elements of the vector, and calls Analyze on each, collecting the results (in a vector, as it turns out). We would end up with a vector containing, perhaps, a LocalBindingExpr, a StaticMethodExpr, and a VarExpr (assuming x is a local variable, and z is not a local variable). The + would translate as a call to the static method Numbers.add. When Emit is called, it will generate code for each of these elements, emit code to make an array, and call RT.vector, which takes an array and creates an IPersistentVector. (Actually there is an efficiency hack for small argument counts, but that’s just gravy.)

In the easy mode, one would include:

|  |  |  |  |
| --- | --- | --- | --- |
| AssignExpr | BodyExpr | BooleanExpr | ConstantExpr |
| EmptyExpr | ImportExpr | InstanceOfExpr | KeywordExpr |
| LocalBindingExpr | MapExpr | MetaExpr | MonitorEnterExpr |
| MonitorExitExpr | NilExpr | NumberExpr | SetExpr |
| StringExpr | TheVarExpr | ThrowExpr | UnresolvedVarExpr |
| VectorExpr | VarExpr |  |  |

Slightly harder are those that have more complicated control flow, more parsing variations, or more case analysis (particularly for type propagation).

|  |  |  |  |
| --- | --- | --- | --- |
| CaseExpr | DefExpr | LetExpr | RecurExpr |
| LetFnExpr | IfExpr | TryExpr |  |

There are many that deal with making interop calls:

|  |  |  |
| --- | --- | --- |
| HostExpr |  |  |
| FieldOrPropertyExpr | InstancePropertyExpr | InstanceFieldExpr |
| StaticFieldOrPropertyExpr | StaticPropertyExpr | StaticFieldExpr |
| MethodExpr | InstanceMethodExpr | StaticMethodExpr |
| InstanceZeroArityCallExpr |  |  |
| QualifiedMethodExpr |  |  |

There are those that deal with defining functions. These have associated classes that help with dealing with each arity’s method:

|  |  |
| --- | --- |
| ObjExpr | (ObjMethod) |
| FnExpr | (FnMethod) |
| NewInstanceExpr | (NewInstanceMethod) |

And the Expr classes dealing with making sense of regular invocations (f x y z …)

|  |  |  |
| --- | --- | --- |
| InvokeExpr | KeywordInvokeExpr | StaticInvokeExpr |