

# Type System

You should explore the rest of the [Boo Compiler](#) so this is more understandable, especially the [Compiler Steps](#), the [Abstract Syntax Tree](#) structures, and the [Boo Parser](#).

Say you have a line of code like:

```
m = x.something()
```

What is m? What is x? What does something do? You're in the same boat that the compiler is in when it processes code. To us and the compiler, those are just words or names. They are references to something, but to what we don't know yet.

Even without knowing what the names refer to, the parser can tell certain things that it uses to generate the [Abstract Syntax Tree](#). The above code is a statement (not an enum or class or import...). Seeing the "X = Y" form it knows the statement is a binary assignment expression. Seeing the "X.Y" form it knows that "something" should refer to some member of "x". Seeing the parentheses () it knows we are invoking either a method or other callable object named "something", or if "something" is a type like a class, we are invoking its constructor (like "m = SomeNameSpace.SomeClass()").

So if we wanted to generate the equivalent AST by hand, we would say something like:

```
stmt = ExpressionStatement()  
  
b = BinaryExpression()  
b.Operator = BinaryOperatorType.Assign  
b.Left = ReferenceExpression(Name: "m")  
  
invoke = MethodInvocationExpression()  
invoke.Target =  
MemberReferenceExpression(Name: "something",  
Target:  
ReferenceExpression(Name: "x"))  
  
b.Right = invoke  
  
stmt.Expression = b
```

So now the compiler has to find out "what" everything in the AST is. By "what" I mean some type of code object that exists in either some external assembly or that is defined somewhere else in your code. Besides a "Name" property, AST nodes like ReferenceExpressions have an "Entity" property that will store information about the kind of type that needs to be created for that node in the [EmitAssembly](#) step.

What are the different kinds of types a node can possibly be? See [EntityType.cs](#). A name might refer to a system type (which means class, enum, interface, or struct), or a method, field, property, whatever.

Now in our example we'll skip ahead to the [ProcessMethodBodiesWithDuckTyping](#) step in the compiler pipeline (actually ProcessMethodBodies, its superclass). We can do this because our example code doesn't define any new types on its own (we have no "class" statement for example). Any new types that have been created or imported were handled in earlier steps like [BindTypeDefinitions](#), [BindBaseTypes](#), and [BindTypeMembers](#).

In [ProcessMemberBodies.cs](#) the compiler visits the 2 simple reference expressions "m" and "x" in the OnReferenceExpression method.

It retrieves the appropriate entity by calling the Resolve(name) method in [NameResolutionService.cs](#)

The name resolution service asks the type system service, does this name refer to a built-in primitive (like "int" or "date")? If yes, we know the entity type because we have a hashtable mapping primitive names to their corresponding entity types (which correspond to real .NET/Mono types like System.Int32 or System.DateTime).

If no, then it starts a hierarchical search through each namespace context in which the reference expression is enclosed. Each namespace may have its own hashtable mapping names to entity types. Let's say the line of code is in the global namespace (actually that code will have been moved inside a "Main" method inside a module, see the [IntroduceModuleClasses](#) step). To resolve "m" and "x" it has to start with the global or module-level namespace. Back in the [InitializeNameResolutionService](#) step, a global namespace and module namespaces were created. When asked to resolve a name, these namespaces will search the external assembly references for a type matching that name, or the internal modules in your code for any types you have created yourself, like new classes.

The "something" member reference expression is processed in the ProcessMemberReferenceExpression method of ProcessMemberBodies.cs. It asks the target of the member reference ("x") for its namespace (unless "something" is a type itself and so we ask for its constructor). Expressions or types have their own namespaces (see [INamespace.cs](#) and IType and other interfaces contained in [IEntity.cs](#)), which may store a list of child entities they contain, and can retrieve an entity type given a name.

## Watch it happen

To see the names being bound to their respective types, run the boo compiler (booc.exe) with the "-vvv" option. This very verbose option spits all the references and their corresponding entities during the compile pipeline.

## Type Inference

A little on [Type Inference](#).

You can see though that if "m" was not declared earlier in the code, then the compiler cannot find out its type until it finds out the type of the "something" member reference. If "m" was declared earlier (like, "m as MyClass"), then when the compiler visits that declaration it will bind the type created for MyClass to "m". If the compiler then visits the binary expression and finds the type that "something" returns is not assignable to the type of "m" then it will complain. In our sample line of code, it is mandatory at least that "x" is defined elsewhere (perhaps it is a class type or a namespace), "x" contains an entity matching the name "something", and "something" refers to either a method or callable entity (if for example "x" is a class), or else "something" must be a type entity like a class itself with an accessible constructor method ("m = SomeNameSpace.SomeClass()").

## Creating New Types

When you create a new class with a line like "class MyClass...", for example, the boo compiler creates a new instance of the [InternalClass entity](#) and sets the ClassDefinition's Entity property to that object. The ClassDefinition also stores the name you used ("MyClass"). The InternalClass handles name resolution and later is used in the [Emit Assembly](#) step to help generate the correct IL assembly code to define the type you created.

## Relation to Macro Processing

See [Syntactic Macros](#). Understanding the type system and other features of the [Boo Compiler](#), it is easier to understand how AST macros work, and their limitations. Currently, macros are processed before the types are processed. A lot of times a macro is just rearranging the AST structure or adding new references to save typing. But say you need to know the type of a parameter passed to your macro. The Entity property will be null at that point. Boo may eventually incorporate "type-safe" macros that are processed later in the compiler pipeline after the type system has done its thing.

Look at the "with" macro on the [Syntactic Macros](#) page and you'll understand why it has to check for an underscore "\_" at the beginning of a referenceexpression in order to know whether or not that reference should be turned into a memberreferenceexpression targeting fooInstanceWithReallyLongName. It can't use a leading period like Visual Basic because that is an illegal name. And we can't simply use no prefix (i.e., "f1" instead of "\_f1") like some other languages do, because how would we distinguish which references refer to members of fooInstanceWithReallyLongName and which do not. We can't since we do not know the type of fooInstanceWithReallyLongName at that point.

A type safe macro that is processed after the type system would have to be more careful in how it processes the AST, so as to not break the name-type bindings. Remember that the correct type is determined according to the hierarchical structure of a node's enclosing namespaces. If the nodes are rearranged, the name might really refer to a completely different type if the name exists in multiple namespaces. And if you move the AST node for "x.something()" before "x = MyClass()", then x is undefined at first and should have been a type error.