SymPL Implementation on the Dynamic Language Runtime

Bill Chiles

This draft still needs a final proofreading pass!

1 Introduction

This document demonstrates how to implement a very simple language, SymPL, using the Dynamic Language Runtime (DLR) as it ships in .NET 4.0. The reader should have some basic familiarity with language implementation concepts such as compiler components and runtime support. Mostly curiosity will be enough for reading this document. The Sympl implementation does not show production quality .NET interoperability such as IronPython has. For example, SymPL mostly just binds to .NET functions by matching argument and parameter counts and whether the runtime argument types are assignable to the parameter types. For serious language implementers, this document assumes you have deeper .NET awareness for how types and the CLR work. The goal for serious language implementers is to make you aware of how to get started building a language on the DLR.

The SymPL language implementation demonstrates the following:

- Using DLR Expression Trees (which include LINQ Expression Trees v1) for code generation
- Using DLR dynamic dispatch caching
- Building runtime binders
- Building dynamic objects and dynamic meta-objects
- Supporting dynamic language/object interoperability
- Very simple hosting with application supplied global objects
- Basic arithmetic and comparison operators with fast DLR dispatch caching
- Control flow
- Name binding within Sympl
- Method and top-level function invocation with fast DLR dispatch caching
- Very simple .NET interoperability for runtime method and operation binding
- Closures
- Assignment with various left-hand-side expressions

Before reading this document you may want to read dlr-overview.doc from www.codeplex.com/dlr to get a general grasp of the DLR. You should read, either before or simultaneously with this document, the sites-binders-dynobj-interop.doc document from the same web site. This document refers to the latter now and then for more detailed background.

1.1 Sources

All the source code for the DLR releases regularly on www.codeplex.com/dlr, with weekly source updates at least. The full sources for the language used in this document are under <installdir>\languages\sympl\.

As a side note, the implementation was started in IronPython as an experiment in using the DLR functionality from a dynamic language. Using a dynamic language with a REPL would provide some productivity gains in exploring implementation techniques and .NET-isms. The experiment was beneficial for flushing out some implementation and design issues in IronPython and DLR. It showed where the dynamic language made some things easier. It also showed some things that were more straightforward if done in C# or VB due to the nature of the DLR APIs and some hurdles created by IronPython itself being implemented on the DLR. Since the code for both are in the source tree, you can see some of the IronPython comments that note where the C# implementation was easier.

1.2 Walkthrough Organization

The walkthrough starts with Hello World and the entire infrastructure needed for it. There is a surprising amount required if you do not have a built-in print function. Having a built in print function would not be very interesting. The rest of the document flows as the Sympl implementation actually evolved. One might argue for a different organization, say, with all the runtime binder descriptions in one place. However, the fundamental point of this document is to walk you through how to start your own language.

There's a fairly natural order to how things were added that shows one good way to proceed. We build some fundamental infrastructure in a basic end-to-end scenario, Hello World. Then we add function definitions and invocations, some simple control flow, and some built-in types and constants. At this point, there's enough language to write a little library. Also, as we incrementally add instantiation, instance member access, and indexing, we can write interesting programs or tests. Later topics require more work and DLR concepts, so doing some easier features to flesh out Sympl after Hello World works well.

2 Quick Language Overview

This is a high-level description of language features to set the stage for starting to discuss the implementation. For a more detailed description (no language lawyer formal specification :-)), see section 10. For quick view of the simple hosting and runtime API, see section 22.

SymPL stands for symbolic programming language. It is pronounced as "simple" and is usually written as "Sympl". It looks like a Lisp variant, but it lacks several of Lisp's semantics. For example, Sympl does NOT:

- evaluate identifiers by indirecting through a symbol's value cell (Sympl is like Common Lisp lexical variables)
- have symbols with distinct value and function value cells (only value cells)
- *read* all code into a list literal, have a *readtable*, read macros, etc.
- allow all identifiers to be redefined (Sympl has keywords)

Sympl has very minimal language features and simple .NET interoperability for instructional purposes only. Sympl has:

- pure expression-based semantics (easy with Expression Trees v2)
- a simplified module mechanism akin to python's
- hosting model for executing files and snippets within host-provided scopes
- top-level functions and global variables within file modules
- closures (free with the DLR)
- basic data types: int, string, double
- basic control flow: if, loop, function call
- basic arithmetic, Boolean, and comparison operations
- infix dot for accessing data and function members
- case-insensitive identifiers

3 Walkthrough of Hello World

We're going to start with walking through the infrastructure needed for Hello World. This section explains a lot of important concepts in the DLR needed to start implementing a language using the DLR. Later sections discuss the remaining concepts, which come in two forms as Sympl gains features. One is different kinds of dynamic call sites and runtime binders, and the other is simple syntax tree translations to Expression Trees. Do NOT skip this section just because it is Hello World!

This is the Sympl program to get working end-to-end:

```
(import system)
(system.console.writeline "hey")
```

If the above code were in test.sympl, the host code snippet is (see program.cs):

```
static void Main(string[] args) {
    string dllPath = typeof(object).Assembly.Location;
    Assembly asm = Assembly.LoadFile(dllPath);
    Sympl s = new Sympl(new Assembly[] { asm });
    s.ExecuteFile("test.sympl");
```

This two line Sympl program requires a surprising amount of support if you do not cut corners such as having a built-in print function. The needed language implementation pieces are:

- Lexical scanning and Parsing to an abstract syntax tree (AST).
- Analysis -- is the program well-formed semantically and to what variables do identifiers refer.
- Code generation -- made much simpler by using the DLR's Expression Trees. Sympl emits Expressions directly from its analysis pass over an input program.
- Hosting -- how you cause a Sympl program to execute and give it access .NET assemblies, namespaces, and types.
- Reflection over .NET assemblies and types.
- Runtime helper functions -- for example, performing import and binding helpers.
- Dynamic object helpers.
- File modules or scopes -- do files provide some name isolation as in python or none at all as in elisp. Sympl has file globals. Import brings names from .NET assemblies and host globals into a file module in Sympl.
- Keyword forms -- for example, import, defun, let*, new, set, +, <, and so on.
- Function call.
- Name resolution and global variable lookup.
- Member access -- system.console
- Member invocation -- console.writeline

This document focuses on the DLR pieces and key points of learning. It explains some of the implementation or language semantics as needed to make sure you know why Sympl needed some piece of infrastructure. This document won't dwell on every detail, and won't explain components such as the lexer or parser. These are pretty simple and straightforward even if you've never implemented them before, just check out lexer.cs and parser.cs.

3.1 Quick Code Overview

This document excerpts relevant code while discussing parts of the Sympl implementation. If you want to see the code snippets in situ, this is a breakdown of the code.

Lexer.cs -- as you'd expect, this is the lexical scanner. The main entry points are GetToken and PutToken.

Parser.cs -- again, as you'd expect, the parser takes a stream of tokens from the lexer and builds a syntax tree (AST). The IronPython implementation of Sympl just has top-level functions in the parser.py module, but the lexer.py defines a class to hold tokens for PutToken. The main entry points are ParseFile and ParseExpr, which take TextReaders.

Etgen.cs -- Sympl collapses the semantic analysis and code generation phases into one. Sympl's analysis is pretty trivial and combined with using Expression Trees for your code generation, there's no reason to be more complicated. If you had Python's lexical rules, or classes with smarter identifier resolution semantics, you would need to make a first pass over the AST to analyze and bind names. The main entry point is AnalyzeExpr.

Runtime.cs -- here is where Sympl defines all the runtime binders, runtime helper functions, dynamic object helpers, and some built-in types such as Symbol and Cons. Runtime.py has some of the code in it that sympl.cs has so that the python modules layer nicely, but since all the C# code is in one namespace, some classes sit in the file in which they are used.

Sympl.cs -- primarily this holds the Sympl hosting class which you instantiate to run Sympl code. This also has some runtime support, either instance methods on Sympl or support classes used by the hosting class. Some of this latter code is in runtime.py for python module layering that wasn't necessary in C#.

Program.cs -- main just runs the test.sympl file and then drops into a REPL for testing and playing with code in the test module. This can be useful for trying out one expression at time and changing breakpoints in VS to watch the implementation flow.

3.2 Hosting, Globals, and .NET Namespaces Access

This section talks about hosting only a bit for now, as hosting relates to host Globals and accessing .NET namespaces and types. There is a Sympl class that programs can instantiate to create a Sympl runtime. When you create the runtime, you provide one or more assemblies whose namespaces and types Sympl programs can access when executing within this runtime. The Sympl runtime has a Globals table. Hosts and add globals to this table, and Sympl fills it with the available namespaces and types.

Sympl leverages the DLR's ExpandoObjects as a trivial implementation of its Globals table. Sympl does not use a .NET dictionary for a couple of reasons. The first is that Sympl needs a DLR based dynamic object to hold its globals so that programs can use late binding to access variables. Sympl uses the DLR's DynamicExpression for GetMember operations with runtime binder metadata to look up names case-INsensitively. This approach means Sympl modules can interoperate as dynamic objects with other languages and libraries. The second reason to use ExpandoObjects is that the DLR has a fast implementation for accessing members.

The Sympl.Globals table is the root of a tree of ExpandoObjects. Each interior ExpandoObject node represents one .NET namespace. For each sub namespace within it, there is a nested

ExpandoObject representing each sub namespace. The leaf nodes are TypeModel objects that Sympl defines. Sympl cannot store .NET RuntimeType objects because when Sympl programs dot into the type objects, the Sympl code wants to bind names to the types represented by the RuntimeType objects. The code does not want to bind names to the members of the type RuntimeType. We'll go through this in detail below.

This is Sympl's reflection code, which is pretty simple, while IronPython's is much richer and also tuned for startup performance (from sympl.cs):

```
    public void AddAssemblyNamesAndTypes() {
        foreach (var assm in _assemblies) {
            foreach (var typ in assm.GetExportedTypes()) {
                string[] names = typ.FullName.Split('.');
                var table = _globals;
                for (int i = 0; i < names.Length - 1; i++) {
                    string name = names[i].ToLower();
                    if (DynamicObjectHelpers.HasMember(
                            (IDynamicMetaObjectProvider)table, name)) {
                        table = (ExpandoObject)(DynamicObjectHelpers
                                                    .GetMember(table,
                                                               name));
                    } else {
                        var tmp = new ExpandoObject();
                        DynamicObjectHelpers.SetMember(table,
name,tmp);
                        table = tmp;
                    }
                }
                DynamicObjectHelpers.SetMember(table,
                                               names[names.Length - 1],
                                               new TypeModel(typ));
            }
        }
    }
```

3.2.1 DLR Dynamic Binding and Interoperability -- a Very Quick Description

Before explaining how Sympl fills the Sympl.Globals table with models of namespaces and types, you need to be familiar with how dynamic CallSites work. This is a brief, high-level description, but you can see the sites-binders-dynobj-interop.doc document on www.codeplex.com/dlr for details of the concepts and mechanisms. CallSites are one of the top benefits of using the DLR. They are the basis of the DLR's fast dynamic method caching. They work together with DLR runtime binders and dynamic objects to enable languages and dynamic library objects to interoperate.

The DLR has a notion of dynamic operations that extend beyond method invocation. There are twelve kinds of operations designed for interoperability, such as GetMember, SetMember, InvokeMember, GetIndex, Invoke, CreateInstance, BinaryOperation, UnaryOperation, and so on. See diagram below. They require runtime binding to inform a CallSite how to perform an operation given the object or objects that flowed into the call site. Binders represent the language that owns the CallSite as well as metadata to inform the binding search. Binders return a sort of rule, entailing how to perform the operation and when the particular implementation is appropriate to use. CallSites can cache these rules to avoid future binding searches on successive executions of the CallSite.

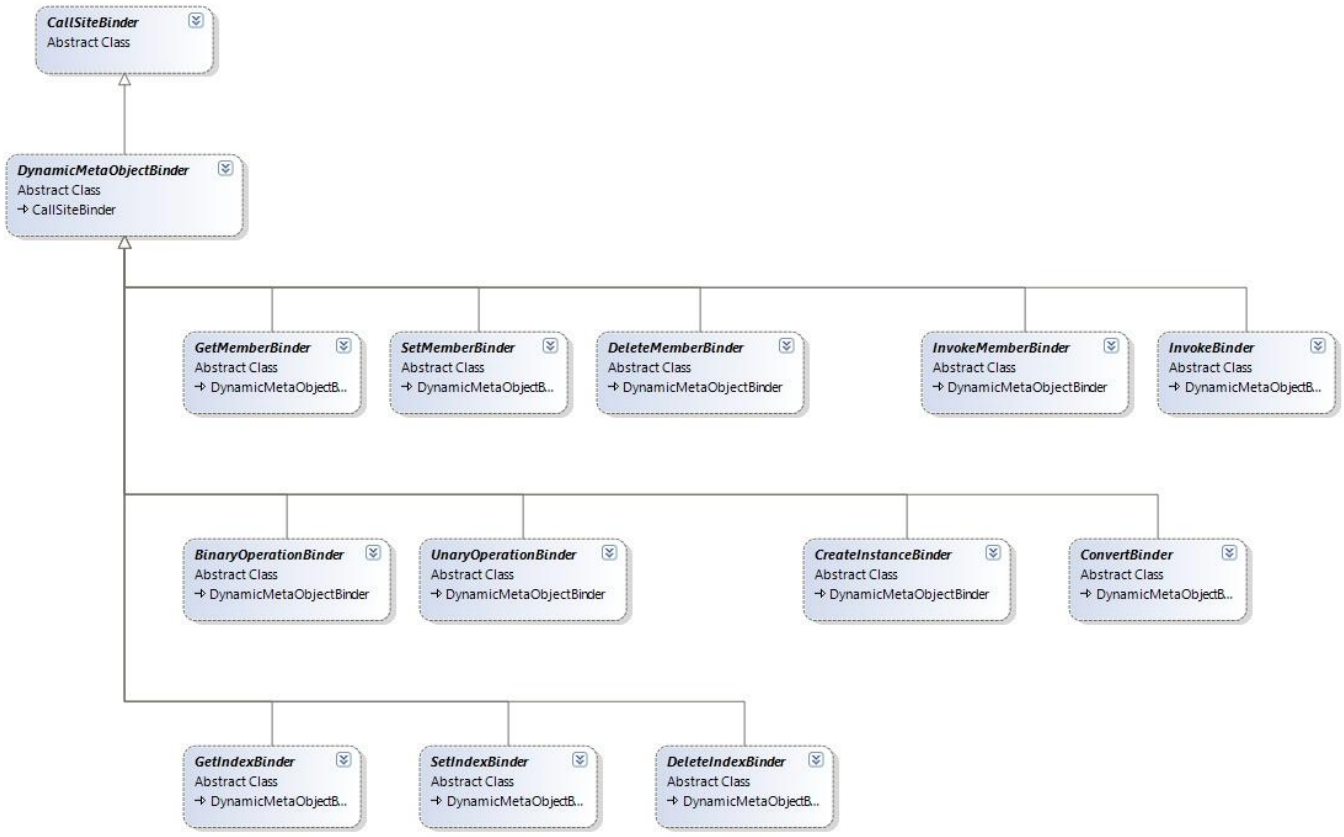
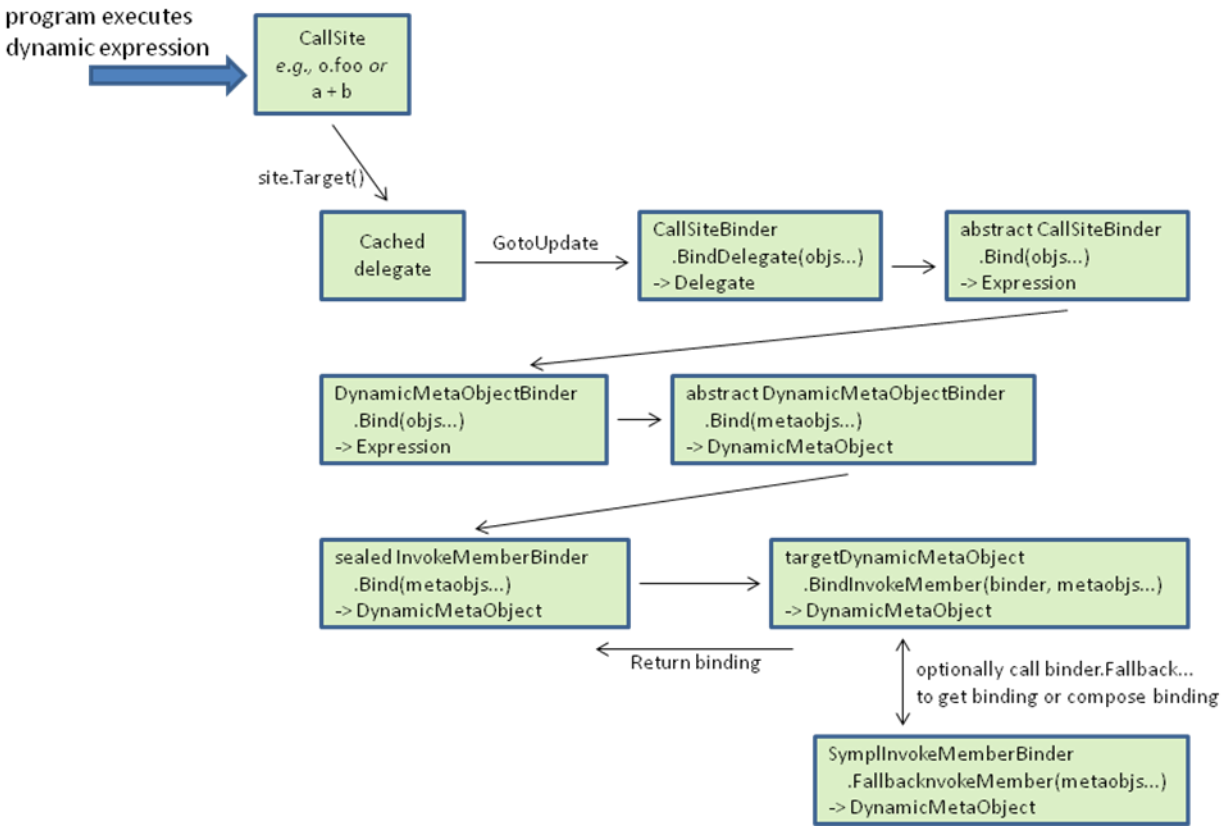


Figure 1: Twelve Standard Interop Binders

This is the basic flow end-to-end for binding a dynamic operation in the DLR when there is a cache miss (explained in detail below):



1) Your language compiles a method invocation or binary operation to an Expression Tree DynamicExpression, which the DLR compiles to a CallSite. The DynamicExpression's operation is defined by the CallSiteBinder you placed in the expression and the metadata contained in that binder instance.

2) When execution of the containing program comes to the call site, the DLR has emitted code to invoke the CallSite's Target delegate. Initially, this target delegate simply calls on the binder you provided to get a rule for how to perform the operation. Ignoring more advanced options, there are no implementations cached at this point. Any time there is a cache miss, the CallSite calls on the binder to search for how to perform the operation.

3) The CallSite calls BindDelegate, which the base CallSiteBinder type handles. You almost never need to implement at this level. You'll probably only need to use DynamicMetaObjectBinders, which allow you to participate in the DLR's interoperability protocol. The CallSiteBinder base implementation of BindDelegate calls its abstract Bind method to get an Expression that embodies the rule (a test for when the rule is valid and the implementation of the operation given the current arguments).

4) DynamicMetaObjectBinder provides a base implementation of Bind that wraps all the arguments in DynamicMetaObjects and calls its own abstract Bind overload that takes DynamicMetaObjects. This is where dynamic objects, which implement IDynamicMetaObjectProvider, get to insinuate themselves in the binding process. Meta-objects represent a value in the binding process for binders that derive from

DynamicMetaObjectBinder. DynamicMetaObjects support dynamic objects being king of their own binding logic. They also are the means of interoperability across languages and libraries, as well as allowing several operations to combine into one (for very advanced cases). Meta-objects also represent the results of binding, which is explained further in the sites-binders-dynobj-interop.doc document. When DynamicMetaObjectBinder calls Bind on DynamicMetaObjects, execution flows to one of the twelve binder types (see diagram) that define the standard interoperability operations. InvokeMemberBinder is an example.

5) The twelve standard interoperability binder types all provide sealed implementations of the Bind method that takes DynamicMetaObjects. These implementations call on the first argument to ask it to provide a binding for the operation. This is part of the interoperability protocol. The DLR gives precedence to the object, represented by its DynamicMetaObject, to determine how to perform the operation. The DynamicMetaObject may take various actions, including falling back to the binder to compute the binding. The DLR's default DynamicMetaObjects, which wrap regular .NET objects, always just calls on the language binders to get a binding.

   a) The target DynamicMetaObject may return a new DynamicMetaObject that represents the result of the operation. The result is the Expression property of the DynamicMetaObject. Combined with the result DynamicMetaObject's restrictions, CallSiteBinder can form a rule as a delegate. For example, the target object for an InvokeMember operation might be an IronRuby object. It would look at the metadata on the binder (the name to invoke and the ignoreCase flag). If it could find the member, and it is callable, it would provide the binding for how to perform the operation on this kind of object.

   b) The target DynamicMetaObject might not be able to find the member, so it would call your binder's FallbackInvokeMember. The dynamic object got first chance to handle the operation, but it calls on the language that owns the compiled line of code. There are two reasons for this part of the protocol. The first is that the language binder might manufacture a member on the object, such as standard members that all objects in a particular language are expected to have. The second reason is that even if your language doesn't add members to all objects, like Sympl, the language's binder gets the chance to return a DynamicMetaObject representing an error that is in the expected style of the language. The language binder has a couple of responsibilities or options, such as COM support and handling DynamicMetaObjects that have no concrete value at binding time. Sympl adds these to its binders late in Sympl's evolution.

   c) The target DynamicMetaObject might know how to perform GetMember but not InvokeMember, like Python meta-objects. Then it could make a DynamicMetaObject representing the result of performing a GetMember operation and call the binder's FallbackInvoke method. This gives the language owning the line of code the opportunity to apply its semantics for argument conversions to match a callable object. For example, does it convert nil to false or throw an error that you cannot bind the call.

6) NOTE, DynamicMetaObjects and binders should NEVER throw when they fail to bind an operation. Of course, they throw if there's some internal integrity error, but when they cannot produce a binding for an operation, they should return a DynamicMetaObject representing a Throw expression of an appropriate error so that the binding process finishes smoothly. The reason is that another DynamicMetaObject or binder down the chain of execution might handle the operation. There are situations described later where

DynamicMetaObjects call on binders to get their rule or error meta-object to fold those into the result the DynamicMetaObject produces.

7) Now suppose execution flows into your language's binder via a call to FallbackOP (for some OP, see the standard interoperability binders' members). This happens when the DynamicMetaObject, as described above, fails to find a binding and asks the owning language to do the binding. OP is the root name of the binder, which is also the name of one of the twelve base interoperability operations (GetMember, InvokeMember, etc.). By convention, DynamicMetaObjects should fall back to the language's binder for .NET binding. They fall back to the language's binder, so it can apply the language's semantics for how it resolves methods and operations to interoperate with .NET. Finally, your language's binder is the last stop, and the binder returns a DynamicMetaObject representing how to perform the binding, or a DynamicMetaObject representing a language-specific error.

8) When your language's binder (that derives from DynamicMetaObjectBinder) returns, or the target argument's meta-object returns, then DynamicMetaObjectBinder's Bind returns. Execution unwinds back to the DLR's base binder methods. The DLR combines the resulting DynamicMetaObject's restrictions and Expression property into a rule that gets compiled into a delegate. This delegate becomes the new target delegate of the CallSite. If this call site ever needs to repeat this binding process, it moves this delegate into a site specific L1 cache where it can be found and re-used faster than repeating the binding process, compiling Expression Trees, JIT'ing, and so on. The rule also gets put into a binder specific L2 cache when the delegate was created. Since you can re-use the same binder in multiple call sites (for example, the same SymplBinaryOperationBinder instance with Operator Add can be used for all CallSites in Sympl code), other Add call sites then benefit from cached rules/delegates that would be returned for the same kinds of arguments in any Add site.

A key point, which you can see in the Hello Word end-to-end scenario, is that often binding an operation for a call site never involves your language's binder. The Sympl binders don't become relevant until we actually implement an Invoke operation to call a Sympl function definition. Then again later, the SymplGetMemberBinder and SymplInvokeMemberBinder will actually perform work instead of having DynamicMetaObjects handle the binding.

All the work described above, as well as the actual work to inspect runtime types and .NET reflection information, is why CallSite caching is so important to performance for dynamic languages.

3.2.2 DynamicObjectHelpers

Now that you're familiar with how binding works end-to-end, this section describes how Sympl fills in the Globals table with namespace and type model objects. See the code excerpt above for AddAssemblyNamesAndTypes that uses DynamicObjectHelpers (defined in runtime.cs). DynamicObjectHelpers employs a degenerate usage of CallSites and binders to provide HasMember, GetMember, and SetMember methods.

Sympl needs to use DynamicObjectHelpers because it reflects over namespaces and types, and it has the names of entities in variables. The DLR and dynamic objects are designed to be used from languages when they compile expressions like "obj.foo" and "obj.foo = x". "foo" is the name of a member of obj, so the language can use a DyanmicExpression and put "foo" in the metadata of a GetMemberBinder or SetMemberBinder. However, when the string "foo" is the

value of a variable at runtime, and you need to see of the dynamic object obj has a member named "foo", you need to build something like DynamicObjectHelpers.

DynamicObjectHelpers explicitly creates a CallSite so that it can encode a request of the dynamic object. DynamicObjectHelpers.GetMember must put a binder in the CallSite, even though the expectation is that the dynamic object would handle the operation. The binder's only metadata is the name to look up. There is also metadata for whether to ignore case, but DoHelpersGetMemberBinder inherently sets that to true. GetMember invokes the CallSite as shown below (from runtime.cs):

```
    static private Dictionary<string,
                              CallSite<Func<CallSite, object, object>>>
            _getSites = new ...;

    internal static object GetMember(IDynamicMetaObjectProvider o,
                                     string name) {
        CallSite<Func<CallSite, object, object>> site;
        if (! DynamicObjectHelpers._getSites.TryGetValue(
                                                 name, out site)) {
            site = CallSite<Func<CallSite, object, object>>
                            .Create(new DoHelpersGetMemberBinder(name));
            DynamicObjectHelpers._getSites[name] = site;
        }
        return site.Target(site, o);
```

Except for the last line, all this code is just for caching and fetching the CallSite object with the binder that has the right metadata (that is, the name to look up). We cache these since it speeds up Sympl's launch time 4-5X depending on whether it is the C# or IronPython implementation.

If the requested member is not present in object o, the dynamic object's meta-object will call FallbackGetMember on the binder, DoHelperGetMemberBinder. There are two cases for this binder in this situation. The first is that the DynamicMetaObject provided a way to look up the name, via errorSuggestion, in case the DoHelperGetMemberBinder can't. ErrorSuggestion is discussed later when we drill into real Sympl runtime binders. If the errorSuggestion is null, then this binder returns a DynamicMetaObject result that represents a sentinel value for HasMember. You can see the FallbackGetMember method for DoHelperGetMemberBinder in runtime.cs:

```
    public override DynamicMetaObject FallbackGetMember(
          DynamicMetaObject target, DynamicMetaObject errorSuggestion)
{
        return errorSuggestion ??
               new DynamicMetaObject(

Expression.Constant(DynamicObjectHelpers.Sentinel),
                    target.Restrictions.Merge(
                        BindingRestrictions.GetTypeRestriction(
                            target.Expression, target.LimitType)));
```

Sympl's DynamicObjectHelpers also needs to support a HasMember method, which the DLR's interoperability protocol does not support. Sympl implements this with the sentinel value mentioned above and a HasMember implemented as follows:

```
internal static bool HasMember(IDynamicMetaObjectProvider o,
                               string name) {
    return (DynamicObjectHelpers.GetMember(o, name) !=
            DynamicObjectHelpers.Sentinel);
```

If you look at DoHelpersSetMemberBinder, it simply returns a DynamicMetaObject that represents an error because if the dynamic object didn't handle the SetMember operation, then Sympl doesn't know how to store members into it. Some languages like IronPython do manufacture members that can be set on objects, but Sympl does not do that.

It is worth noting that using CallSites and binders in this way means you are not enjoying the benefits of caching. Since you create a call site and binder each time, there are no successive executions of the same call site where caching would help. Sometimes something like DynamicObjectHelpers is just what you need, but for production quality languages like IronPython, you would need to re-use the call sites or do something like IronPython does for performance.

3.2.3 TypeModels and TypeModelMetaObjects

As stated above, Sympl needs to wrap RuntimeType objects in Sympl's TypeModel objects. If Sympl did not do this, then Sympl code such as "console.writeline" (from the Hello Word snippet) would flow a RuntimeType object into an InvokeMember call site. Then Sympl's SymplInvokeMemberBinder would look for the member "writeline" on the type RuntimeType. When Sympl wraps RuntimeType objects, then instances of TypeModel flow into the call site, and a TypeModelMetaObject actually handles the binding as described in section 3.2.1.

TypeModel is very simple. Each instance holds a RuntimeType object. The only functionality of the TypeModel as an IDynamicMetaObjectProvider is to return a DynamicMetaObject that represents the TypeModel instance during the binding process in the previous section. This is most of the class from sympl.cs:

```
public class TypeModel : IDynamicMetaObjectProvider {
    ...
    DynamicMetaObject IDynamicMetaObjectProvider
                          .GetMetaObject(Expression parameter) {
        return new TypeModelMetaObject(parameter, this);
```

TypeModelMetaObject is only used in Hello World for its BindInvokeMember when binding the call console.writeline. By the time Sympl's implementation is done, its TypeModelMetaObject needs a BindGetMember and a BindCreateInstance, which later sections describe.

The IronPython implementation of these types, and some of the uses of them in runtime binders was much trickier than the C# implementation. This is because everything you define in IronPython is already an IDynamicMetaObjectProvider with IronPython meta-objects. It's great that IronPython supports implementing these types, but it's tricky. See the IronPython implementation for comments and explanations.

3.2.4 TypeModelMetaObject's BindInvokeMember -- Finding a Binding

TypeModelMetaObject's BindInvokeMember is where the real work happens for console.writeline. Sympl uses very similar logic throughout its binders. InvokeMember binding

is very similar to CreateInstance, Invoke, GetIndex, and SetIndex for matching arguments to parameters. Here is the code from sympl.cs, which is described in detail below:

```
    public override DynamicMetaObject BindInvokeMember(
            InvokeMemberBinder binder, DynamicMetaObject[] args) {
        var flags = BindingFlags.IgnoreCase | BindingFlags.Static |
                    BindingFlags.Public;
        var members = ReflType.GetMember(binder.Name, flags);
        if ((members.Length == 1) && (members[0] is PropertyInfo ||
                                      members[0] is FieldInfo)){
            // Code deleted, not implemented yet.
        } else {
            // Get MethodInfos with right arg counts.
            var mi_mems = members.
                Select(m => m as MethodInfo).
                Where(m => m is MethodInfo &&
                           ((MethodInfo)m).GetParameters().Length ==
                               args.Length);
            // See if params are assignable from args.
            List<MethodInfo> res = new List<MethodInfo>();
            foreach (var mem in mi_mems) {
                if (RuntimeHelpers.ParametersMatchArguments(
                                       mem.GetParameters(), args)) {
                    res.Add(mem);
                }
            }
            // When MOs can't bind, they fall back to binders.
            if (res.Count == 0) {
                var typeMO =
                        RuntimeHelpers.GetRuntimeTypeMoFromModel(this);
                var result = binder.FallbackInvokeMember(typeMO, args,
                                                         null);
                return result;
            }
            var restrictions =
RuntimeHelpers.GetTargetArgsRestrictions(
                this, args, true);
            var callArgs =
                RuntimeHelpers.ConvertArguments(
                args, res[0].GetParameters());
            return new DynamicMetaObject(
                   RuntimeHelpers.EnsureObjectResult(
                       Expression.Call(res[0], callArgs)),
               restrictions);
```

Sympl takes the name from the binder's metadata and looks for all public, static members on the type represented by the target meta-object. TypeModelMetaObjects also looks for members ignoring case because Sympl is a case-INsensitive language.

Next, TypeModelMetaObject's BindInvokeMember filters for all the members that are MethodInfos and have the right number of arguments. Ignore the checks for properties or fields that have delegate values for now. Then the binding logic filters for the MethodInfos that have parameters that can be bound given the types of arguments present at this invocation of the call site. There are a few things to note (see next few paragraphs), but essentially the matching test in ParametersMatchArguments is (in runtime.cs):

```
MethodInfo.GetParameters[i].IsAssignableFrom(args[i].LimitType)
```

DynamicMetaObjects have a RuntimeType and a LimitType. You can disregard RuntimeType because if the DynamicMetaObject has a concrete value (HasValue is true) as opposed to just an Expression, then RuntimeType is the same as LimitType; otherwise, RuntimeType is null. LimitType comes from either the Type property of the expression with which the DynamicMetaObject was created, or LimitType is the actual specific runtime type of the object represented by the DynamicMetaObject. There is always an expression, even if it is just a ParameterExpression for a local or temporary value.

The function ParametersMatchArguments also tests whether any of the arguments is a TypeModel object. This is more important to understand when Sympl adds instantiating and indexing generic types, but for now, it is enough to know that ParametersMatchArguments matches TypeModels if the parameter type is Type:

```
...
if (paramType == typeof(Type) &&
    (args[i].LimitType == typeof(TypeModel)))
    continue;  // accept as matching
...
```

You might think you could just use the Expression.Call factory to find a matching member. This is not generally a good idea for a few reasons. The first is that any real language should provide its own semantics for finding a most applicable method and resolving overloads when more than one could apply. The second is that Expression.Call can't handle special language semantics such as handling TypeModels or mapping nil to Boolean false. The third is that even though Sympl doesn't care and just picks the first matching MethodInfo, Expression.Call throws an error if it finds more than one matching.

The last point to make now about matching parameters is that Sympl's very simple logic could be improved in two ways to enable more .NET interoperability. First, Sympl could check whether an argument DynamicMetaObject has a value (HasValue is true), and if the value is null. In this case, Sympl could test if the parameter's type is a class or interface, or if the parameter's type is a generic type and also Nullable<T>. These conditions would mean null matches while Sympl's current code does not find a match. The second improvement is to notice the last parameter is an array of some type for which all the remaining arguments match the element type. Sympl could then use Expression.NewArrayInit to create an expression collecting the last of the arguments for passing to the matching MethodInfo's method. Production-quality languages will do these kinds of tests as well as other checks such as built-in or user conversions.

If there are applicable methods, Sympl binds the InvokeMember operation. Sympl does not try to match a most specific method. In fact, in Hello World Sympl actually invokes the object overload on WriteLine, not the string overload due to the order .NET Reflection happens to serve up the MethodInfos. The result of binding is a DynamicMetaObject with an expression that implements the member invocation and restrictions for when the implementation is valid. See the next section for a discussion of argument conversions and restrictions. Note that the MethodCallExpression may be wrapped by EnsureObjectResult (see below) because CallSites (except for Convert Operations) must return something strictly typed that's assignable to object.

The DLR enforces that rules produced by DynamicMetaObjects and binders have implementation expressions of type object (or a reference type). This is necessary because a site potentially mixes rules from various objects and languages, and the CallSite must in effect

unify all the expression result types in the code it produces and compiles into a delegate. Rather than have various arbitrary policies and conversions, the DLR simply says all CallSites that use binders that derive from DynamicMetaObjectBinder must have object results. Sympl uses this little helper from RuntimeHelpers in runtime.cs, which is pretty self-explanatory:

```
public static Expression EnsureObjectResult (Expression expr) {
    if (! expr.Type.IsValueType)
        return expr;
    if (expr.Type == typeof(void))
        return Expression.Block(
                    expr, Expression.Default(typeof(object)));
    else
        return Expression.Convert(expr, typeof(object));
```

If BindInvokeMember finds no matching MethodInfos, then it falls back to the binder passed to it in the parameters. Falling back is explained at a high level in section 3.2.1. In this particular case, Sympl generates an expression to represent fetching the underlying RuntimeType from the TypeModel object by calling GetRuntimeTypeMoFromModel. This helper function is explained further when we describe how Sympl supports instantiating arrays and generic types. For now, you can see that since Sympl falls back to the binder, which might not be a Sympl binder. Sympl needs to pass a .NET Type representation since the binder may not know what a TypeModel is.

3.2.5 TypeModelMetaObject.BindInvokeMember -- Restrictions and Conversions

Nearing the end of binding for Hello World, BindInvokeMember needs to create restrictions and argument conversions to return a new DynamicMetaObject representing how to perform the operation. The restrictions form the test part of the rule. They are true when the resulting DynamicMetaObject's Expression property is a valid implementation of the operation in the CallSite. The conversions are necessary even though the code only selected methods to which the arguments are assignable. Just because the arguments types are assignable to the parameter types according to the test, Expression Trees and their compilation require strict typing and explicit conversion.

This is the code that generates the restrictions (from runtime.cs), more on this below:

```
public static BindingRestrictions GetTargetArgsRestrictions(
        DynamicMetaObject target, DynamicMetaObject[] args,
        bool instanceRestrictionOnTarget){
    var restrictions = target.Restrictions
                            .Merge(BindingRestrictions
                                        .Combine(args));
    if (instanceRestrictionOnTarget) {
        restrictions = restrictions.Merge(
            BindingRestrictions.GetInstanceRestriction(
                target.Expression,
                target.Value
            ));
    } else {
        restrictions = restrictions.Merge(
            BindingRestrictions.GetTypeRestriction(
                target.Expression,
                target.LimitType
            ));
    }
```

```
        for (int i = 0; i < args.Length; i++) {
            BindingRestrictions r;
            if (args[i].HasValue && args[i].Value == null) {
                r = BindingRestrictions.GetInstanceRestriction(
                        args[i].Expression, null);
            } else {
                r = BindingRestrictions.GetTypeRestriction(
                        args[i].Expression, args[i].LimitType);
            }
            restrictions = restrictions.Merge(r);
        }
        return restrictions;
    }
```

You need to be careful to gather all the restrictions from arguments and from the target object first. The binding computations BindInvokeMember does with DynamicMetaObject LimitType and Value properties depend on those restrictions. Having those restrictions come first is like testing that an argument is not null and has a specific type, or perhaps is a non-empty collection, before you test details about members or elements.

Next the restrictions should include a test for the exact value, GetInstanceRestriction, or a test for the type of the value, GetTypeRestriction. You can see in the Sympl code discussed later how this is used, but basically if you're operating on an instance member, your rule should be good for any object of that type. That means use a type restriction. If you're operating on a type, as the TypeModelMetaObject represents, you want to restrict the rule to the exact instance, that is, the exact type model object.

The last bit of restrictions code needs to be consistent with any argument conversions used so that the rule is only applicable in exactly those situations. Because you often need to convert all the arguments to their corresponding parameter types, you also need to add restrictions that say the rule is only valid when the arguments match those types. The Expression Tree compiler removes any unnecessary conversions.

This is the conversion code Sympl uses (from runtime.cs):

```
    public static Expression[] ConvertArguments(
                DynamicMetaObject[] args, ParameterInfo[] ps) {
        Debug.Assert(args.Length == ps.Length,
                     "Internal: args are not same len as params?!");
        Expression[] callArgs = new Expression[args.Length];
        for (int i = 0; i < args.Length; i++) {
            Expression argExpr = args[i].Expression;
            if (args[i].LimitType == typeof(TypeModel) &&
                ps[i].ParameterType == typeof(Type)) {
                // Get arg.ReflType
                argExpr =
GetRuntimeTypeMoFromModel(args[i]).Expression;
            }
            argExpr = Expression.Convert(argExpr, ps[i].ParameterType);
            callArgs[i] = argExpr;
        }
        return callArgs;
    }
```

There two ways in which the restrictions and argument conversions code appear to be different. The first is where the restrictions code tests for null values. Nulls need to be handled as instance restriction, but the conversions don't need to change in any way. The second difference is that DynamicMetaObjects representing Sympl's TypeModels need to be wrapped in expressions that get the .NET's RuntimeType values. The restrictions ensure the argument is of type TypeModel so that the wrapping expression works.

Note, with TypeModelMetaObject.BindInvokeMember implemented, we can define a SymplInvokeMemberBinder that does no work at all. We could write a little bit of code that fetches the Console TypeModel object from Sympl.Globals and passes it to a hand-coded CallSite invocation. The instance of SymplInvokeMemberBinder would only serve to pass the metadata, the name "writeline" and ignoreCase, to TypeModelMetaObject's BindInvokeMember method. Incremental testing and exploration like this were easier in the IronPython implementation due to its REPL.

3.3 Import Code Generation and File Module Scopes

Sympl adopts a pythonic model of file namespaces. Each file implicitly has its own globals scope. Importing another file's code creates a global name in the current global scope that is bound to the object representing the other file's global scope. There is support for member access to get to the other file's globals. You can also import names from the host provided Sympl.Globals table and .NET namespaces. Doing so binds file globals to the host objects or objects representing .NET namespaces and types. See section 21.6 for more details.

Sympl leverages the DLR's ExpandoObjects to represent file scopes. When Sympl compiles a free reference identifier, it emits an Expression Tree with a DynamicExpression that performs a GetMember operation on the ExpandoObject for the file's globals. The ExpandoObject's meta-object computes the runtime binding for Sympl and returns the value of the variable. There is no implicit chain of global scopes. The look up does NOT continue to the host globals table. Sympl could do this easily with a runtime helper function that first looked in the file's globals and then the Sympl.Globals.

Sympl compiles an Import keyword form to a call to RuntimeHelpers.SymplImport. It takes a list of names to successively look up starting with Sympl.Globals and resulting in a namespace or TypeModel. SymplImport also takes a Sympl runtime instance and a file module instance (more on that below in the section on Sympl.ExecuteFile). The helper then stores the last name in the list to the file's scope with the last value retrieved. If you look at SymplImport, you'll see Sympl employs the DynamicObjectHelpers again for accessing the Sympl.Globals and for storing into the file scope.

This is the code from etgen.cs to analyze and emit code for Import:

```
public static Expression AnalyzeImportExpr(SymplImportExpr expr,
                                           AnalysisScope scope) {
    if (!scope.IsModule) {
        throw new InvalidOperationException(
            "Import expression must be a top level expression.");
    }
    return Expression.Call(
        typeof(RuntimeHelpers).GetMethod("SymplImport"),
```

```
            scope.RuntimeExpr,
            scope.ModuleExpr,
            Expression.Constant(expr.NamespaceExpr.Select(id =>
id.Name)
                                                    .ToArray()),
            Expression.Constant(expr.MemberNames.Select(id => id.Name)
                                                .ToArray()),
            Expression.Constant(expr.Renames.Select(id => id.Name)
                                            .ToArray()));
```

Analysis passes an AnalysisScope chain throughout the analysis phase. The chain starts with a scope representing the file's top level. This scope has a ParameterExpression that will be bound to a Sympl runtime instance and a second ParameterExpresson for a file scope instance. Enforcing that the Import form is at top level in the file is unnecessary since this function could follow the scope chain to the first scope to get the RuntimeExpr and ModuleExpr properties. This restriction was just a design choice for Sympl.

The Expression.Call is quite simple, passing the MethodInfo for RuntimeHelpers.SymplImport, the parameter expressions for the execution context, and any name arguments to the Import form. Each set of names in the AST for the Import syntax is a collection of IdOrKeywordTokens from lexer.cs. AnalyzeImportExpr iterates over them to lift the string names to pass to the runtime helper function.

Emitting the code to call SymplImport in the IronPython implementation is not this easy. You can't call the Call factory with MethodInfo for IronPython methods. You need to emit an Invoke DynamicExpression with a no-op InvokeBinder. See the code for comments and explanation.

3.4 Function Call and Dotted Expression Code Generation

We've looked at most of the infrastructure needed for Hello Word at this point, ignoring lexical scanning and parsing. So far, Sympl has a runtime instance object, reflection modeling, dynamic object helpers, a TypeModel meta-object with InvokeMember binding logic, code generation for Import, and runtime helpers for Import and binding. Once Sympl can parse and emit code in general for keyword forms, function calls, and dotted expressions, Hello Word will be fully functional with no short cuts to be filled in later.

It is worth a quick comment on Sympl's parser design and syntax for calls. Following an open parenthesis is either a keyword (import, new, elt, set, +, *, etc.) or an expression that results in a callable object. If the first token after the parenthesis is a keyword, Sympl dispatches to a keyword form analysis function, such as shown above for Import. If following the parenthesis is a dotted expression, Sympl analyzes this into a series of dynamic GetMember and InvokeMember operations. Since the Sympl syntactic form is a parenthesis, and the dotted expression is the first sub expression, it must result in an InvokeMember operation. In any other position, the dotted expression would result in a GetMember operation. If the expression following the parenthesis is not a keyword or dotted expression, Sympl analyzes the expression to get a callable object. Sympl emits a dynamic expression with an Invoke operation on the callable object. For more information on function calls see section 21.4.1.

3.4.1 Analyzing Function and Member Invocations

Here's the analysis and code generation for general function calls from etgen.cs, described in detail below:

```
    public static DynamicExpression AnalyzeFunCallExpr(
            SymplFunCallExpr expr, AnalysisScope scope) {
        if (expr.Function is SymplDottedExpr) {
            SymplDottedExpr dottedExpr =
(SymplDottedExpr)expr.Function;
            Expression objExpr;
            int length = dottedExpr.Exprs.Length;
            if (length > 1) {
                objExpr = AnalyzeDottedExpr(
                    new SymplDottedExpr(
                           dottedExpr.ObjectExpr,

RuntimeHelpers.RemoveLast(dottedExpr.Exprs)),
                    scope
                );
            } else {
                objExpr = AnalyzeExpr(dottedExpr.ObjectExpr, scope);
            }
            List<Expression> args = new List<Expression>();
            args.Add(objExpr);
            args.AddRange(expr.Arguments
                              .Select(a => AnalyzeExpr(a, scope)));
            // last expr must be an id
            var lastExpr = (SymplIdExpr)(dottedExpr.Exprs.Last());
            return Expression.Dynamic(
                scope.GetRuntime().GetInvokeMemberBinder(
                    new InvokeMemberBinderKey(
                            lastExpr.IdToken.Name,
                            new CallInfo(expr.Arguments.Length))),
                        typeof(object),
                        args);
        } else {
            var fun = AnalyzeExpr(expr.Function, scope);
            List<Expression> args = new List<Expression>();
            args.Add(fun);
            args.AddRange(expr.Arguments
                              .Select(a => AnalyzeExpr(a, scope)));
            return Expression.Dynamic(
                scope.GetRuntime()
                     .GetInvokeBinder(
                          new CallInfo(expr.Arguments.Length)),
                typeof(object),
                args);
```

Looking at the outer else branch first for simplicity, you see the DynamicExpression for a callable object. The CallInfo metadata on the Invoke binder counts only the argument expressions that you would normally think of as arguments to a function call. However, the 'args' variable includes as its first element the callable (function) object. This is often a confusing point for first-time DynamicExpression users.

There are a couple of aside points to make now in Sympl's evolving implementation. The first is to ignore that the code calls GetInvokeBinder. Also, in the IF's consequent branch for the dotted expression case, ignore the call to GetInvokeMemberBinder. Imagine these are just calls to the constructors, which is what the code actually was at this point in Sympl's evolving implementation:

```
new SymplInvokeBinder(callinfo)
new SymplInvokeMemberBinder(name, callinfo)
```

The Get... methods produce canonical binders, a single binder instance used on every call site with the same metadata. This is important for DLR L2 caching of rules. See section 18 for how Sympl returns canonical binders and why, and see sites-binders-dynobj-interop.doc for more details on CallSite rule caching. The second point is that right now the SymplInvokeMemberBinder doesn't do any work, other than convey the identifier name and CallInfo as metadata. We know the TypeModelMetaObject will provide the implementation at runtime for how to invoke "console.writeline". Execution will never get to the Invoke DynamicExpression until later, see section 5.

In Sympl, it doesn't matter what the expression is that produces the callable object for a function call. It could be an **IF**, a **loop**, **try**, etc. There are two big reasons Sympl uses a dynamic expression rather than an Expression.Call or Expression.Invoke node. Sympl may need to do some argument conversions or general binding logic (such as converting TypeModel objects to RuntimeType objects), and it is good to isolate that to the binder. Also, using a DynamicExpression supports language and library interoperability. For example, an IronPython callable object might flow into the call site for the dynamic expression, and an IronPython DynamicMetaObject would handle the Invoke operation.

Back to the outer IF's consequent branch, the function expression is a dotted expression. AnalyzeFunCallExpr analyzes the expression to get a target object (below is an explanation of dotted expression analysis). Next the function call analysis generates code for all the argument expressions. Then it builds a DynamicExpression with an InvokeMember operation. The metadata for InvokeMember operations is the member name, whether to ignore case, and argument count. Sympl's binders inherently set ignoreCase to true. As with the Invoke operation, the binder's CallInfo metadata counts only the argument expressions that you would normally think of as arguments to a function call. However, the 'args' variable includes as its first element the 'objExpr' resulting from the dotted expression.

AnalyzeFunCallExpr picks off the simple case in the else branch, when the length of dotted expressions is only one. This is when there is an expression, a dot, and an identifier. In the general case, when length > 1, AnalyzeFunCallExpr creates a new SymplDottedExpr AST node that excludes the last member of the dotted expression. Analyzing the first N-1 results in an Expression that produces a target object for an InvokeMember operation. The member to invoke, or last bit of the dotted expression, must be an identifier.

3.4.2 Analyzing Dotted Expressions

Here is the code for analyzing dotted expressions, from etgen.cs:

```
public static Expression AnalyzeDottedExpr(SymplDottedExpr expr,
                                           AnalysisScope scope) {
    var curExpr = AnalyzeExpr(expr.ObjectExpr, scope);
    foreach (var e in expr.Exprs) {
        if (e is SymplIdExpr) {
            curExpr = Expression.Dynamic(
```

```
                    new SymplGetMemberBinder(
                            ((SymplIdExpr)e).IdToken.Name),
                            typeof(object),
                            curExpr);
            } else if (e is SymplFunCallExpr) {
                var call = (SymplFunCallExpr)e;
                List<Expression> args = new List<Expression>();
                args.Add(curExpr);
                args.AddRange(call.Arguments
                                  .Select(a => AnalyzeExpr(a, scope)));
                curExpr = Expression.Dynamic(
                            new SymplInvokeMemberBinder(
                                  ((SymplIdExpr)call.Function).IdToken
                                                              .Name,
                                  new CallInfo(call.Arguments.Length)),
                            typeof(object),
                            args);
            } else {
                throw new InvalidOperationException(
                    "Internal: dotted must be IDs or Funs.");
            }
        }
        return curExpr;
```

AnalyzeDottedExpr just loops over all the sub expressions and turns them into dynamic expressions with GetMember or InvokeMember operations. Each iteration updates curExpr, which always represents the target object resulting from the previous GetMember or InvokeMember operation. GetMember takes the name, and Sympl's binder inherently ignores case. InvokeMember takes the name and the CallInfo, just as described above, and the Sympl binder inherently ignores case.

3.4.3 What Hello World Needs

In the Hello World end-to-end example, the function call to "system.console.writeline" will turn into three operations. The first is a file scope global lookup for "system". The second is a GetMember operation for "console" on the ExpandoObject modeling the System namespace. Then the function call is actually an InvokeMember operation of "writeline" on the Console TypeModel.

As stated above SymplInvokeMemberBinder doesn't do any real work now, other than convey the identifier name and CallInfo as metadata. Execution will never get to the Invoke DynamicExpression in AnalyzeFunCallExpr until later, see section 5.

3.5 Identifier and File Globals Code Generation

Analyzing identifiers is pretty easy, so this section explains the whole process for lexical and file globals. Any syntax that can introduce a scope (function declaration and let* since we don't have classes or other binding forms) adds an AnalysisScope to the head of the chain before analyzing its sub expressions. It also adds to the AnalysisScope all the declared identifiers, mapping them to ParameterExpressions. At run time when execution enters a binding contour represented by one of these scopes, the locals (represented by the ParameterExpressions) get set to argument values or initialization values.

ParameterExpressions are how you represent variables in ExpressionTrees. The ParameterExpression objects appear in parameter lists of LambdaExpressions and variable lists of BlockExpressions as a means of declaring the variables. To reference a variable, use the same instance of a ParameterExpression in a sub expression. Note, ParameterExpressions optionally take a name, but he name is for debugging or pretty printing a tree only. The name has no semantic bearing whatsoever.

When resolving an identifier at code generation time, Sympl searches the current AnalysisScope chain for the name. If the name is found, code generation uses the ParameterExpression to which the name mapped. There's no concern for references crossing lambda boundaries and lifting variables to closures since Expression Trees handle closures automatically. If the name in hand cannot be found by the time the search gets to the file module AnalysisScope, then it is a global reference.

Here's the code for AnalyzeIdExpr from etgen.cs:

```
public static Expression AnalyzeIdExpr(SymplIdExpr expr,
                                       AnalysisScope scope) {
    if (expr.IdToken.IsKeywordToken) {
        // Ignore for now, discussed later with keyword literals.
    } else {
        var param = FindIdDef(expr.IdToken.Name, scope);
        if (param != null) {
            return param;
        } else {
            return Expression.Dynamic(
               new SymplGetMemberBinder(expr.IdToken.Name),
               typeof(object),
               scope.GetModuleExpr());
```

Initially this function only contained the code in the else branch, which is where it handles lexicals and globals. FindIdDef does the AnalysisScope chain search. If it returns a ParameterExpression, then that's the result of AnalyzeIdExpr. Otherwise, AnalyzeIdExpr emits a GetMember DynamicExpression operation. The target object is a file's module object, which gets bound at run time to a variable represented by a ParameterExpression. The ParameterExpression is stored in the AnalysisScope representing the file's top level code. Since the file scope is an ExpandoObject, it's DynamicMetaObject will handle the name lookup for the call site that the DynamicExpression compile into.

The SymplGetMemberBinder at this point just conveys the metadata of the name and that the look up is case-INsensitive. The Hello World end-to-end scenario doesn't need the binder to do any real binding. Later when Sympl adds the ability to create instances of .NET types, then we'll look at SymplGetMemberBinder (see section 11).

3.6 Sympl.ExecuteFile and Finally Running Code

There's been a long row to hoe getting real infrastructure for the Hello World example. See the list in section 3. There's a very good foundation to Sympl now, and a lot of DLR concepts have been exposed. Now it is time to wrap all that up and make it work end-to-end from instantiating a Sympl runtime, to compiling test.sympl (just the first two lines), to running the code.

At a high level, program.cs calls Sympl.ExecuteFile on the test.sympl file. Sympl parses all the expressions in the file, wraps them in a LambdaExpression, compiles it, and then executes it. Let's look at the code before we walk through it:

```
public ExpandoObject ExecuteFile(string filename) {
    return ExecuteFile(filename, null);
}

public ExpandoObject ExecuteFile(string filename,
                                 string globalVar) {
    var moduleEO = CreateScope();
    ExecuteFileInScope(filename, moduleEO);
    globalVar = globalVar ??
                Path.GetFileNameWithoutExtension(filename);
    DynamicObjectHelpers.SetMember(this._globals, globalVar,
                                   moduleEO);
    return moduleEO;
}

public void ExecuteFileInScope(string filename,
                               ExpandoObject moduleEO) {
    var f = new StreamReader(filename);
    DynamicObjectHelpers.SetMember(moduleEO, "__file__",
                                   Path.GetFullPath(filename));
    try {
        var asts = new Parser().ParseFile(f);
        var scope = new AnalysisScope(
            null,
            filename,
            this,
            Expression.Parameter(typeof(Sympl), "symplRuntime"),
            Expression.Parameter(typeof(ExpandoObject),
                                 "fileModule"));
        List<Expression> body = new List<Expression>();
        foreach (var e in asts) {
            body.Add(ETGen.AnalyzeExpr(e, scope));
        }
        var moduleFun = Expression
                        .Lambda<Action<Sympl, ExpandoObject>>(
            fType,
            Expression.Block(body),
            scope.RuntimeExpr,
            scope.ModuleExpr
        );
        var d = moduleFun.Compile();
        d(this, moduleEO);
    } finally {
        f.Close();
    }
}
```

ExecuteFile creates a scope, an ExpandoObject, for the file globals. This ExpandoObject is also the result of ExecuteFile so that the host can access globals created by executing the Sympl source code. For example, the host could hook up a global function as a command handler or event hook. Before returning, ExecuteFile uses the file name, "test", to add a binding in Sympl.Globals to the file scope object. This doesn't matter for Hello World, but later when the

test file imports lists.sympl and other libraries, it will be important for accessing those libraries. You can see the use of DynamicObjectHelpers again for accessing members on the runtime's globals and the file globals.

To do the work of parsing and compiling, ExecuteFile calls ExecuteFileInScope on the file and module scope. This function creates a new Parser to parse the file. Then it creates an AnalysisScope with the filename, Sympl runtime instance, and ParameterExpressions for generated code to refer to the runtime instance and file module scope. Using the AnalysisScope, ExecuteFileInScope analyzes all the expression in the file.

ExecuteFieInScope uses an Action type that takes a Sympl runtime and an ExpandoObject for the file scope. It uses this Action type to create a LambdaExpression with a BlockExpression that holds the top-level expressions from test.sympl. The LambdaExpression takes two arguments, represented by the ParameterExpression objects placed in the AnalysisScope.

The last thing to do is compile the LambdaExpression and then invoke it. The invocation passes 'this' which is the Sympl runtime instance and the file scope object. This is how all code runs ultimately, that is, wrapped in a LambdaExpression with whatever context is needed passed in.

4 Assignment to Globals and Locals

We already discussed lexical and globals in general in section 3.5. This section discusses adding variable assignment to Sympl. This starts with the keyword **set**, for which we won't discuss lexical scanning or parsing. As a reminder, Sympl is expression-based, and everything returns value. The **set** keyword form returns the value that is stored.

Let's look at the AnalyzeAssignExpr from etgen.cs:

```
    public static Expression AnalyzeAssignExpr(SymplAssignExpr expr,
                                               AnalysisScope scope) {
        if (expr.Location is SymplIdExpr) {
            var idExpr = (SymplIdExpr)(expr.Location);
            var lhs = AnalyzeExpr(expr.Location, scope);
            var val = AnalyzeExpr(expr.Value, scope);
            var param = FindIdDef(idExpr.IdToken.Name, scope);
            if (param != null) {
                return Expression.Assign(
                           lhs,
                           Expression.Convert(val, param.Type)
                );
            } else {
                var tmp = Expression.Parameter(typeof(object),
                                               "assignTmpForRes");
                return Expression.Block(
                   new[] { tmp },
                   Expression.Assign(
                       tmp,
                       Expression.Convert(val, typeof(object))
                   ),
                   Expression.Dynamic(
                       scope.GetRuntime()
                            .GetSetMemberBinder(idExpr.IdToken.Name),
                       typeof(object),
                       scope.GetModuleExpr(),
```

```
                    tmp
                ),
                tmp
            );
        }
    // Ignore rest of function for now, discussed later with elt
    // keyword and SetMember.
```

There are only two cases to implement at this point, lexical variables and file global variables. Later, Sympl adds setting indexed locations and .NET members. The key here is the chain of AnalysisScopes and that Expression Trees provide automatic closure environments if lifting is needed. FindIdDef (code is in etgen.cs) searches up the chain until it finds a scope with the identifier mapped to a ParameterExpression. If it finds the name, then it is a lexical variable. For lexical identifiers AnalyzeAssignExpr emits an Assign node, which guarantees to return the value stored. You also need to ensure the val expression converts to the ParameterExpression's type. The Assign factory method would throw if the Expression types were inconsistent.

If FindIdDef finds no scope mapping the identifier to a ParameterExpression, then the variable is a file global. As described in section 3.3, Sympl leverages the DLR's ExpandoObjects to represent file scopes. Sympl uses a Dynamic expression with one of its SymplSetMemberBinders, which carries the identifier name as metadata. Sympl's binders also set ignoreCase to true implicitly. We'll discuss the use of the BlockExpression after digging into the DynamicExpression a bit more.

There are a couple of points to make now in Sympl's evolving implementation. The first is to ignore GetSetMemberBinder. Imagine this is just a call to the constructor:

```
new SymplSetMemberBinder(idExpr.IdToken.Name)
```

GetSetMemberBinder produces canonical binders, a single binder instance used on every call site with the same metadata. This is important for DLR L2 caching of rules. See section 18 for how Sympl does this and why, and see sites-binders-dynobj-interop.doc for more details on CallSite rule caching. The second point is that right now the SymplSetMemberBinder doesn't do any other work, other than convey the identifier name as metadata. We know the ExpandoObject's DynamicMetaObject will provide the implementation at runtime for how to store the value as a member.

Sympl is a case-INsensitive language. Sympl is case-preserving with identifiers stored in tokens and in binder metadata. Preserving case provides a bit more opportunity for interoperability. For example, if a Sympl file module flowed into some IronPython code, and it did case-sensitive lookups, the python code is more likely to just work. As another example, in the IronPython implementation of Sympl, where the Cons class is implemented in IronPython, it is IronPython's DynamicMetaObject that looks up the members First and Rest. IronPython still has a bug that it ignores ignoreCase on binders. While Sympl preserves case in the metadata, it uses lowercase as the canonical representation of identifiers in AnalysisScopes.

There's more code to the globals branch than the SetMember DynamicExpression. Sympl's semantics is to return a value from every expression, and it returns the value stored from assignments. Sympl wraps the DynamicExpression in a Block with a temporary variable to ensure it only evaluates the value expression once. The Block has as its last expression the ParameterExpression for the temporary variable so that the BlockExpression returns the value stored. This code is for example now, but when written ExpandoObject didn't return the values

it stored. The convention for binders and DynamicMetaObjects is that they should return rules for SetMember and SetIndex operations that result in the value stored.

5 Function Definition and Dynamic Invocations

This section focuses on defining Sympl functions and invoking them. Section 3.4 on function call and dotted expressions discusses Sympl semantics and AnalyzeFunCallExpr quite a bit. It may be useful to read the Sympl language definition about function calls, section 21.4.1.

5.1 Defining Functions

Sympl uses the **defun** keyword form to define functions. **Defun** takes a name as the first argument and a list of parameter names as the second. These are non-evaluated contexts in the Sympl code. The rest of a **defun** form is a series of expressions. The last expression executed in a Sympl function provides the result value of the function. Sympl does not currently support a **return** keyword form, but you'll see the implementation is plumbed for its support. You could look at how Sympl implements **break** for loops and probably add this in 20 minutes or so as an exercise. You can see example function definitions in the .sympl files.

Code generation for defun is pretty easy. At a high-level, you add the parameter names to a new, nested AnalysisScope, analyze the sub expressions, and then emit an assignment to a file scope global whose value is a LambdaExpression. When the code in a file is gathered into an outer Lambda and compiled, all the contained lambdas get compiled (see section 3.6 for more information on executing Sympl files). Here's the code from etgen.cs, which is discussed more below:

```
public static DynamicExpression AnalyzeDefunExpr
        (SymplDefunExpr expr, AnalysisScope scope) {
    if (!scope.IsModule) {
        throw new InvalidOperationException(
            "Use Defmethod or Lambda when not defining " +
            "top-level function.");
    }
    return Expression.Dynamic(
               scope.GetRuntime().GetSetMemberBinder(expr.Name),
               typeof(object),
               scope.ModuleExpr,
               AnalyzeLambdaDef(expr.Params, expr.Body, scope,
                                "defun " + expr.Name));
}

private static Expression AnalyzeLambdaDef
        (IdOrKeywordToken[] parms, SymplExpr[] body,
         AnalysisScope scope, string description) {
    var funscope = new AnalysisScope(scope, description);
    funscope.IsLambda = true;  // needed for return support.
    var paramsInOrder = new List<ParameterExpression>();
    foreach (var p in parms) {
        var pe = Expression.Parameter(typeof(object), p.Name);
        paramsInOrder.Add(pe);
        funscope.Names[p.Name.ToLower()] = pe;
    }
```

```
var bodyexprs = new List<Expression>();
foreach (var e in body) {
    bodyexprs.Add(AnalyzeExpr(e, funscope));
}
var funcTypeArgs = new List<Type>();
for (int i = 0; i < parms.Length + 1; i++) {
    funcTypeArgs.Add(typeof(object));
}
return Expression.Lambda(
            Expression.GetFuncType(funcTypeArgs.ToArray()),
            Expression.Block(bodyexprs),
            paramsInOrder);
```

Looking at the helper AnalyzeLambdaDef first, functions definitions need to push an AnalysisScope on the chain. The scope can serve a few purposes. Sympl uses it to hold the locals for the function's parameters. Any references to these names, barring any intervening **let*** bindings, will resolve to the ParameterExpressions stored in this new AnalysisScope. References from nested **lambda** expressions will automatically become closure environment references thanks to Expression Trees.

The new AnalysisScope also has IsLambda set to true as plumbing for adding **return** keyword forms. Then analyzing a **return** form would just search the AnalysisScope chain for a first IsLambda scope and use a return LabelTarget stored in the scope. The body of the lambda would also need to be wrapped in a LabelExpression that used the return LabelTarget.

After storing ParameterExpressions for the locals, AnalyzeDefunExpr analyzes all the body expressions in the context of the new AnalysisScope.

To create a LambdaExpression, Sympl needs to create an array of Types determined from the parameter definitions. Of course, in Sympl, these are all object.

Now AnalyzeDefunExpr can emit the code. There is a SetMember DynamicExpression to store into the ExpandoObject for the file module. It stores a binding for the function's name to the LambdaExpression. The LambdaExpression is just a BlockExpression of the body expressions. As explained a couple of times now, just assume GetSetMemberBinder is a call to the constructor for SymplSetMemberBinder and see section 18 for an explanation. Note, still at this point in the evolution of Sympl, its SetMemberBinder doesn't really do any work, other than convey the name and ignoreCase metadata.

This is the code from AnalyzeFunCallExpr in etgen.cs (discussed further in section 3.4.1) for invoking Sympl functions (or any callable object from another language, a library, or a delegate from .NET):

```
var fun = AnalyzeExpr(expr.Function, scope);
List<Expression> args = new List<Expression>();
args.Add(fun);
args.AddRange(expr.Arguments.Select(a => AnalyzeExpr(a, scope)));
return Expression.Dynamic(
    scope.GetRuntime()
         .GetInvokeBinder(new CallInfo(expr.Arguments.Length)),
    typeof(object),
    args);
```

5.2 SymplInvokeBinder and Binding Function Calls

At runtime, when trying to call a Sympl function, a delegate will flow into the CallSite. The default .NET meta-object will call FallbackInvoke on SymplInvokeBinder. This code is much simpler than the code for binding InvokeMember we looked at before. As a reminder, if the object that flows into the CallSite is some dynamic object that's callable (for example, an IronPython runtime function or first class type object), then its DynamicMetaObject's BindInvoke will produce a rule for invoking the object with the given arguments.

Here's the code for SymplInvokeBinder from runtime.cs, which is discussed in detail below:

```
public class SymplInvokeBinder : InvokeBinder {
    public SymplInvokeBinder(CallInfo callinfo) : base(callinfo) {
    }

    public override DynamicMetaObject FallbackInvoke(
            DynamicMetaObject targetMO, DynamicMetaObject[] argMOs,
            DynamicMetaObject errorSuggestion) {
        // ... Deleted COM support and checking for Defer for now ...
        if (targetMO.LimitType.IsSubclassOf(typeof(Delegate))) {
            var parms = targetMO.LimitType.GetMethod("Invoke")
                                          .GetParameters();
            if (parms.Length == argMOs.Length) {
                var callArgs = RuntimeHelpers.ConvertArguments(
                                                argMOs, parms);
                var expression = Expression.Invoke(
                    Expression.Convert(targetMO.Expression,
                                       targetMO.LimitType),
                    callArgs);
                return new DynamicMetaObject(
                    RuntimeHelpers.EnsureObjectResult(expression),
                    BindingRestrictions.GetTypeRestriction(
                                        targetMO.Expression,
                                        targetMO.LimitType));
            }
        }
        return errorSuggestion ??
            RuntimeHelpers.CreateThrow(
                targetMO, argMOs,
                BindingRestrictions.GetTypeRestriction(
                                    targetMO.Expression,
                                    targetMO.LimitType),
                typeof(InvalidOperationException),
                "Wrong number of arguments for function -- " +
                    targetMO.LimitType.ToString() + " got " +
                    argMOs.ToString());
```

Let's first talk about what we aren't talking about now. This code snippet omits the code to check if the target is a COM object and to use built-in COM support. See section 19 for information adding this to your binders. The snippet also omits some very important code that protects binders and DynamicMetaObjects from infinitely looping due to producing bad rules. It is best to discuss this in one place, so see section 20 for how the infinite loop happens and how to prevent it for all binders.

Because language binders are expected to provide the rules for binding to static .NET objects, and the convention is that DynamicMetaObjects fall back to binders for this purpose, first this code checks for a Delegate type. This code doesn't do much for Sympl invocations. Dynamic objects fall back to languages for Delegates because the language needs to control implicit argument conversions, may provide special mappings like nil to false, and so on.

FallbackInvoke then checks the parameter and argument counts. If the delegate looks callable, Sympl optimistically converts each argument to the corresponding parameter type. See section 3.2.5 on TypeModelMetaObject's BindInvokeMember for details on ConvertArguments. Sympl doesn't need to do more argument and parameter type matching here. This method doesn't need the information for resolving methods, and users will get an error about not being able to convert the argument to the function's parameter type as appropriate.

Sympl then creates code to convert the target object to the specific Delegate sub type, which is the LimitType of the DynamicMetaObject. See section 3.2.4 for a discussion of using LimitType over RuntimeType. It may seem odd to convert the object to the type that LimitType reports it to be, but the type of the meta-object's expression might be more general and require an explicit Convert node to satisfy the strict typing of the Expression Tree factory or the actual emitted code that executes. The Expression Tree compiler removes unnecessary Convert nodes.

Then FallbackInvoke creates an InvokeExpression. This expression passes through EnsureObjectResult in case it needs to be wrapped to ensure it is strictly typed as assignable to object. For more information, see section 3.2.4. Then FallbackInvoke wraps this expression in the DynamicMetaObject result with the restrictions for when this rule is valid. The restrictions for Sympl's Invoke are much simpler than the InvokeMember restrictions. This rule should apply to any Delegate with the same type since the rule converts the target and arguments to the types captures in the Delegate type.

If the parameters do not match the arguments, then Sympl either uses the suggested result or creates a DynamicMetaObject result that throws an Exception. See section 6 for a discussion of CreateThrow and restrictions. ErrorSuggestion is more interesting to discuss as it relates to member fetching and setting since it is most likely null in InvokeBinder.FallbackInvoke. If the target callable object were dynamic, then its meta-object would have directly returned a rule, rather than fall back with a result to use if the language didn't know how to invoke the dynamic object.

6 CreateThrow Runtime Binding Helper

As we noted in the high-level description of execution flow when searching for a binding rule, DynamicMetaObjects and binders should NEVER throw when they fail to bind an operation. Of course, they throw if there's some internal integrity error, but when they cannot produce a binding for an operation, they should return a DynamicMetaObject representing a Throw expression of an appropriate error so that the binding process finishes smoothly. The reason is that another DynamicMetaObject or binder down the chain of execution might handle the operation. There are situations described later where DynamicMetaObjects call on binders to get their rule or error meta-object to fold those into the result the DynamicMetaObject produces.

Since there's enough boiler plate code to creating this ThrowExpression, Sympl has a runtime helper function in the RuntimeHelpers class from runtime.cs:

```
    public static DynamicMetaObject CreateThrow
            (DynamicMetaObject target, DynamicMetaObject[] args,
             BindingRestrictions moreTests,
             Type exception, params object[] exceptionArgs) {
        Expression[] argExprs = null;
        Type[] argTypes = Type.EmptyTypes;
        int i;
        if (exceptionArgs != null) {
            i = exceptionArgs.Length;
            argExprs = new Expression[i];
            argTypes = new Type[i];
            i = 0;
            foreach (object o in exceptionArgs) {
                Expression e = Expression.Constant(o);
                argExprs[i] = e;
                argTypes[i] = e.Type;
                i += 1;
            }
        }
        ConstructorInfo constructor =
                            exception.GetConstructor(argTypes);
        if (constructor == null) {
            throw new ArgumentException(
                "Type doesn't have constructor with a given
signature");
        }
        return new DynamicMetaObject(
            Expression.Throw(
                Expression.New(constructor, argExprs),
                typeof(object)),
            target.Restrictions.Merge(
               BindingRestrictions.Combine(args)).Merge(moreTests));
```

This function takes the target and arguments that were passed to a binder's FallbackX method. Since this ThrowExpression is a binding result (and therefore a rule), it is important to put the right restrictions on it. Otherwise, the rule might fire as a false positives and throw when a successful binding could be found. CreateThrow takes moreTests for this purpose. The function also takes the exception type and arguments for the ThrowExpression.

If there are any arguments to the Exception constructor, CreateThrow gathers their types and Constant expressions for each value. Then it looks up the constructor using the argument types.

CreateThrow passes the object type to the Throw factory to ensure the ThrowExpression's Type property represents the object type. Though Throw never really returns, its factory has this overload precisely to support the strict typing model of Expression Trees. The DLR CallSites use the Expression.Type property also to ensure DynamicMetaObjects and binders return implementations for operations that is strictly typed as assignable to object. For more information on why, see section 3.2.4.

The resulting DynamicMetaObject has default restrictions like Sympl puts on almost all rules its binders produce. Since the restrictions for throwing need to match the restrictions a binder would have produced for a positive rule, there are a couple of calls to CreateThrow that pass several restrictions they got from RuntimeHelpers.GetTargetArgsRestrictions. These restrictions duplicate the default restrictions CreateThrow adds, but fortunately the DLR removes duplicates restrictions when it produces the final expression in DynamicMetaObjectBinder.

7 A Few Easy, Direct Translations to Expression Trees

Next the Sympl implementation gained some easy constructs that are either direct translations to Expression Trees or features that leverage code already written for Sympl.

7.1 Let* Binding

Let* binding creates a shadowing bind of an identifier name. For example, a function has a parameter named "x" that comes in set to 10. You **let*** bind "x" to 20, and maybe you set it to other values. When you exit the scope of the **let*** keyword form, references to "x" see the value 10, barring any sets or other intervening **let*** bindings. **Let*** takes one or more lexical bindings and a sequence of sub expressions. It results in the value of the last sub expression executed. The bindings occur serially so that later initialization expressions can refer to earlier bindings. Here's an example:

```
(let* ((var1 init1)
       (var2 init2))
   e1 e2 e3)
```

Here's the code from etgen.cs that handles this keyword form:

```
public static Expression AnalyzeLetStarExpr(SymplLetStarExpr expr,
                                            AnalysisScope scope) {
    var letscope = new AnalysisScope(scope, "let*");
    // Analyze bindings.
    List<Expression> inits = new List<Expression>();
    List<ParameterExpression> varsInOrder =
        new List<ParameterExpression>();
    foreach (var b in expr.Bindings) {
        // Need richer logic for mvbind
        var v = Expression.Parameter(typeof(object),
                                     b.Variable.Name);
        varsInOrder.Add(v);
        inits.Add(
            Expression.Assign(
                v,
                Expression.Convert(AnalyzeExpr(b.Value, letscope),
                                   v.Type))
        );
        letscope.Names[b.Variable.Name.ToLower()] = v;
    }
    List<Expression> body = new List<Expression>();
    foreach (var e in expr.Body) {
        body.Add(AnalyzeExpr(e, letscope));
    }
    inits.AddRange(body);
    return Expression.Block(typeof(object), varsInOrder.ToArray(),
                            inits);
```

This code pushes a new, nested AnalysisScope just like Sympl did for function definition parameters. However, AnalyzeLetStarExpr handles the nested scope differently. It adds each variable to the nested scope after analyzing the initialization expression for the variable. This is obvious from the stand point that the initialization expression cannot refer to the variable for which it is producing the initial value. However, to implement let* semantics instead of let

semantics (which is effectively "parallel" assignment), you need to make sure you add each variable to the scope so that successive variable initialization expressions can refer to previous variables.

The rest of the code just analyzes each sub expression in the nested scope and wraps them in a BlockExpression, which returns the value of the last sub expression.

7.2 Lambda Expressions and Closures

Sympl has lambda keyword forms for first class functions. These are just like defun keyword forms, except they have no name. The results of lambda expression can be assigned to lexical or global variables, and assigning one to a global variable would be the same as using defun. To create local functions that are recursive, you need to create a let* binding of some variable, define a lambda that refers to that variable, and then assign the lambda's result to the variable.

As mentioned previously, closures are automatic with Expression Trees. Any uses references to ParameterExpressions that were established as lambda parameters or Block variables outside of the referencing lambda get lifted to closure environments as needed.

The code for lambda expression is AnalyzeLambdaExpr in etgen.cs. It just calls the helper method used by AnalyzeDefunExpr discussed in section 5.

7.3 Conditional (IF) Expressions

Sympl has an **IF** keyword expression. **IF** takes two or three arguments. If the first argument is true (that is, NOT **nil** or **false**), the **IF** results in the value produced by the second argument. If the first argument is **nil** or **false**, then **IF** results in the value of the third argument. If there is no third argument, then the value is **false**.

To have something easy to test at this point in Sympl's implementation, we added an **eq** test and a few keyword literal constants (**nil**, **true**, **false**), which are described in sub sections below.

Here's the code for **IF** from etgen.cs, which is described further below:

```
public static Expression AnalyzeIfExpr (SymplIfExpr expr,
                                        AnalysisScope scope) {
    Expression alt = null;
    if (expr.Alternative != null) {
        alt = AnalyzeExpr(expr.Alternative, scope);
    } else {
        alt = Expression.Constant(false);
    }
    return Expression.Condition(
               WrapBooleanTest(AnalyzeExpr(expr.Test, scope)),
               Expression.Convert(AnalyzeExpr(expr.Consequent,
                                              scope),
                                  typeof(object)),
               Expression.Convert(alt, typeof(object)));
}

private static Expression WrapBooleanTest (Expression expr) {
    var tmp = Expression.Parameter(typeof(object), "testtmp");
    return Expression.Block(
        new ParameterExpression[] { tmp },
```

```
            new Expression[]
                    {Expression.Assign(tmp, Expression
                                                .Convert(expr,
typeof(object))),
                     Expression.Condition(
                         Expression.TypeIs(tmp, typeof(bool)),
                         Expression.Convert(tmp, typeof(bool)),
                         Expression.NotEqual(
                            tmp,
                            Expression.Constant(null,
                                                typeof(object))))});
```

The first thing AnalyzeIfExpr does is get an expression for the alternative branch or third argument to IF. This defaults to the constant false. Then it simply emits a Conditional Expression with the analyzed sub expressions, forcing everything to type object so that the Conditional factory method sees consistent types and also emits code to return a value.

AnalyzeIfExpr uses WrapBoolenTest. It emits an Expression that captures Sympl's truth value semantics (anything that is not nil or false is true). WrapBooleanTest uses a Block and temporary variable to evaluate the test expression only once. Then it uses an inner ConditionalExpression. If the type of the test expression is bool, then just convert the value to bool. Otherwise, return whether the test value is not null (our runtime representation of nil).

7.4 Eq Expressions

To have something to test along with IF at this point in Sympl's evolution, we added the **eq** keyword form. We can implement this quickly as a runtime helper rather than getting into BinaryOperationBinders and design in the parser. **Eq** has the semantics of returning **true** if the arguments are integers with the same value, or the arguments are the same identical object in memory. To implement **eq**, Sympl just emits a call to it the runtime helper (from etgen.cs):

```
    public static Expression AnalyzeEqExpr (SymplEqExpr expr,
                                            AnalysisScope scope) {
        var mi = typeof(RuntimeHelpers).GetMethod("SymplEq");
        return Expression.Call(mi, Expression.Convert(
                                       AnalyzeExpr(expr.Left, scope),
                                       typeof(object)),
                                   Expression.Convert(
                                       AnalyzeExpr(expr.Right, scope),
                                       typeof(object)));
```

Sympl needs to place the ConvertExpressions in the argument list to satisfy the Call factory and make sure conversions are explicit for the Expression Tree compiler and .NET CLR.

Emitting the code to call SymplEq in the IronPython implementation is not this easy. You can't call the Call factory with MethodInfo for IronPython methods. You need to emit an Invoke DynamicExpression with a no-op InvokeBinder. See the code for comments and explanation.

This is the code for the runtime helper function, from the class RuntimeHelpers in runtime.cs:

```
    public static bool SymplEq (object x, object y) {
        if (x == null)
            return y == null;
```

```
        else if (y == null)
            return x == null;
        else {
            var xtype = x.GetType();
            var ytype = y.GetType();
            if (xtype.IsPrimitive && xtype != typeof(string) &&
                ytype.IsPrimitive && ytype != typeof(string))
                return x.Equals(y);
            else
                return object.ReferenceEquals(x, y);
```

There's not much to explain further since It is a pretty direct implementation of the semantics described above.

7.5 Loop Expressions

Adding loops is almost straightforward translations to Expression Trees. Sympl only has one kind of loop. It repeats forever until the code calls the **break** keyword form. It would be trivial to add **continue** since it is directly supported by Expression Trees like **break**, but demonstrating **break** is enough to show you what to do. For a whileloop and foreach example, see the Expression Tree spec on www.codeplex.com/dlr .

Here's the code for AnalyzeLoopExpr and AnalyzeBreakExpr from etgen.cs, which are described further below:

```
    public static Expression AnalyzeLoopExpr (SymplLoopExpr expr,
                                              AnalysisScope scope) {
        var loopscope = new AnalysisScope(scope, "loop ");
        loopscope.IsLoop = true; // needed for break and continue
        loopscope.LoopBreak = Expression.Label(typeof(object),
                                               "loop break");
        int len = expr.Body.Length;
        var body = new Expression[len];
        for (int i = 0; i < len; i++) {
            body[i] = AnalyzeExpr(expr.Body[i], loopscope);
        }
        return Expression.Loop(Expression.Block(typeof(object), body),
                               loopscope.LoopBreak);
    }

    public static Expression AnalyzeBreakExpr (SymplBreakExpr expr,
                                       AnalysisScope scope) {
        var loopscope = _findFirstLoop(scope);
        if (loopscope == null)
            throw new InvalidOperationException(
                           "Call to Break not inside loop.");
        Expression value;
        if (expr.Value == null)
            value = Expression.Constant(null, typeof(object));
        else
            // Ok if value jumps to break label.
            value = AnalyzeExpr(expr.Value, loopscope);
        return Expression.Break(loopscope.LoopBreak, value,
                                typeof(object));
```

AnalyzeLoopExpr needs to push a new AnalysisScope on the chain. This enables AnalyzeBreakExpr to confirm the break is within a loop and to find the LabelTarget to use to escape from the innermost containing loop expression. Sympl makes the LabelTarget have type object because Sympl loops are expressions that can produce values. This is explained further below.

AnalyzeLoopExpr then just analyzes the sequence of body expressions and emits a LoopExpression. The LoopExpression takes one expression, hence the Block wrapping the body expressions. The factory also takes the LabelTarget that is used to jump past the end of the loop. The break target's type becomes the value of LoopExpression.Type.

Sympl's break keyword form optionally takes one argument, the result value for the loop. AnalyzeBreakExpr finds the innermost containing loop. Then it analyzes the value expression if there is one, or just uses null (which is Sympl's nil value). The Break factory returns a GotoExpression node that compiles to a jump to the loop's break label target, carrying along the value. The object type argument to the Break factory is required because the factory doesn't use the LabelTarget's Type property to set the GotoExpresson's Type property. Without the object type argument, the Goto would have type void. If the code had an IF expression with a branch that called break, then the ConvertExpressions that Sympl wraps around IF branches would throw because they couldn't convert void (from the break expression) to object.

8 Literal Expressions

This section describes some Sympl features added along with **eq** before **IF** and **loop**. These features are quoted symbols, quoted lists, and literal keyword constants. They enable more testing of **IF** and **loop**. This section also describes integer and string literals which of course were added at the beginning along with the lexer. You could skip this section unless you're curious about the built-in list objects and their runtime helpers.

8.1 Integers and Strings

These come out of the parser as SymplLiteralExprs with the value tucked in them. This snippet from AnalyzeExpr in etgen is the code generation:

```
if (expr is SymplLiteralExpr) {
    return Expression.Constant(((SymplLiteralExpr)expr).Value)
```

8.2 Keyword Constants

Literal keyword constants are nil, false, and true. Sympl includes nil for its built-in lists, and it includes true and false for easier .NET interoperability. Sympl could do more work in its runtime binders to map nil to false for Boolean parameters, and map anything else to true for Boolean parameters. Adding this to Sympl didn't seem to add any new lessons given the TypeModel mappings binders do.

These literal keywords come out of the parser as SymplIdExprs. Section 3.5 discussed part of AnalyzeIdExpr from etgen.cs, but it omitted the literal keywords branch shown here:

```
public static Expression AnalyzeIdExpr(SymplIdExpr expr,
                                       AnalysisScope scope) {
```

```
        if (expr.IdToken.IsKeywordToken) {
            if (expr.IdToken == KeywordToken.Nil)
                return Expression.Constant(null, typeof(object));
            else if (expr.IdToken == KeywordToken.True)
                return Expression.Constant(true);
            else if (expr.IdToken == KeywordToken.False)
                return Expression.Constant(false);
            else
                throw new InvalidOperationException(
                    "Internal: unrecognized keyword literal
constant.");
        } else {
            var param = FindIdDef(expr.IdToken.Name, scope);
            if (param != null) {
                return param;
            } else {
                return Expression.Dynamic(
                   new SymplGetMemberBinder(expr.IdToken.Name),
                   typeof(object),
                   scope.GetModuleExpr());
```

Handling this is straightforward; just turn them into ConstantExpressions with obvious .NET representations.

8.3 Quoted Lists and Symbols

Sympl does stand for Symbolic Programming Language, so it needs to have symbols and built-in lists as a nod to its ancestry in Lisp-like languages. Sympl also includes these because it provided a nice domain for writing a little Sympl library, lists.sympl, and demonstrating loading libraries and cross-module access. Quoted literals are symbols, numbers, strings, lists, or lists of these things. A Symbol is an object with a name that's interned into a Sympl runtime's symbol table. All Symbols with the same name are the same object, or eq. Sympl's Symbols are similar to Lisp's, but don't have all the same slots.

8.3.1 AnalyzeQuoteExpr -- Code Generation

The high-level idea is that quoted constants are literal constants, so Sympl builds the constant and burns it into the resulting Expression Tree as a ConstantExpression. Here's the code for AnalyzeQuoteExpr and its helper from etgen.cs:

```
    public static Expression AnalyzeQuoteExpr(SymplQuoteExpr expr,
                                              AnalysisScope scope) {
        return Expression.Constant(MakeQuoteConstant(
                                       expr.Expr, scope.GetRuntime()));
    }

    private static object MakeQuoteConstant(object expr,
                                            Sympl symplRuntime) {
        if (expr is SymplListExpr) {
            SymplListExpr listexpr = (SymplListExpr)expr;
            int len = listexpr.Elements.Length;
            var exprs = new object[len];
            for (int i = 0; i < len; i++) {
                exprs[i] = MakeQuoteConstant(listexpr.Elements[i],
```

```
                                                symplRuntime);
            }
            return Cons._List(exprs);
        } else if (expr is IdOrKeywordToken) {
            return symplRuntime.MakeSymbol(
                                    ((IdOrKeywordToken)expr).Name);
        } else if (expr is LiteralToken) {
            return ((LiteralToken)expr).Value;
        } else {
            throw new InvalidOperationException(
                "Internal: quoted list has -- " + expr.ToString());
```

As stated above, AnalyzeQuoteExpr just creates a ConstantExpression. MakeQuoteConstant does the work, and it takes a Sympl runtime instance both for runtime helper functions and to intern symbols as they are created.

Skipping the first case for a moment, if the expression is an identifier or keyword, Sympl interns its name to create a Symbol as the resulting constant. If the expression is a literal constant (string, integer, nil, false, true), the resulting constant is just the value.

If the expression is a SymplListExpr, then MakeQuoteConstant recurses to make constants out of the elements. Then it calls the runtime helper function Cons._List to create a Sympl built-in list as the constant to emit. See the next section for more on lists and this helper function.

8.3.2 Cons and List Keyword Forms and Runtime Support

Sympl's lists have the structure of Lisp lists, formed from chaining Cons cells together. A Cons cell has two pointers, conventionally with the first pointing to data and the second pointing to the rest of the list (another Cons cell). You can also create a Cons cell that just points to two objects even if the second object is not a Cons cell (or list tail).

Sympl provides a **cons** keyword form and a **list** keyword form for creating lists:

```
(cons 'a (cons 'b nil)) --> (a b)
(cons 'a (cons 'b 'c)) --> (a b . c)
(list 'a 'b 3 "c") --> (a b 3 "c")
(cons 'a (list 2 '(b c) 3)) --> (a 2 (b c) 3)
```

Here is the code for analyzing **cons** and **list** from etgen.cs:

```
public static Expression AnalyzeConsExpr (SymplConsExpr expr,
                                          AnalysisScope scope) {
    var mi = typeof(RuntimeHelpers).GetMethod("MakeCons");
    return Expression.Call(mi, Expression.Convert(
                                    AnalyzeExpr(expr.Left, scope),
                                    typeof(object)),
                               Expression.Convert(
                                   AnalyzeExpr(expr.Right, scope),
                                   typeof(object)));
}

public static Expression AnalyzeListCallExpr
        (SymplListCallExpr expr, AnalysisScope scope) {
    var mi = typeof(Cons).GetMethod("_List");
    int len = expr.Elements.Length;
    var args = new Expression[len];
    for (int i = 0; i < len; i++) {
```

```
            args[i] = Expression.Convert(AnalyzeExpr(expr.Elements[i],
                                                     scope),
                                         typeof(object));
        }
        return Expression.Call(mi, Expression
                                       .NewArrayInit(typeof(object),
                                                     args));
```

Cons is just a Call node for invoking RuntimeHelpers.MakeCons. Sympl analyzes the left and right expressions, and wraps them in Convert nodes to satisfy the Call factory and make sure conversions are explicit for the Expression Tree compiler and .NET CLR. Emitting the code to call MakeCons in the IronPython implementation is not this easy. You can't call the Call factory with MethodInfo for IronPython methods. You need to emit an Invoke DynamicExpression with a no-op InvokeBinder. See the code for comments and explanation.

The List keyword form analyzes all the arguments and ultimately just emits a Call node to Cons._List. Because the helper function takes a params array, Sympl emits a NewArrayInit node that results in an array of objects. Each of the element arguments needs to be wrapped in a Convert node to make all the strict typing consistent for Expression Trees. The same comment as above holds about this code generation being easier in C# than IronPython.

You can look at RuntimeHelpers.MakeCons and Cons._List in runtime.cs to see the code. It's not worth excerpting for this document, but there are a couple of comments to make. The reason Sympl has MakeCons at this point in Sympl's implementation evolution is that Sympl does not have type instantiation. Also, without some basic SymplGetMemberBinder or SymplSetMemberBinder that just looked up the name and assumed it was a property, for example, Sympl couldn't have some of the tests it had at this time with Cons members. The IronPython implementation at this point still didn't need the binder to do more than convey the name because the IronPython implementation of Cons had a DynamicMetaObject for free from IronPython, which handled the binding. Since IronPython ignores the ignoreCase flag, the tests had to be written with uppercase First and Rest names.

9 Importing Sympl Libraries and Accessing and Invoking Their Globals

Most of the ground work for importing libraries and accessing other file module's globals has been done. Section 3.3 on Import code generation discusses some of how this works too. Since file scopes are ExpandoObjects, their DynamicMetaObjects handle the member accesses. When Sympl imports another file of Sympl code, Sympl stores into the importing file's globals scope the name of the file (no directory or extension) using the imported file's module object (an ExpandoObject) as the value. Sympl code can then just dot into these like any other object with members.

Sympl adapts a trick from python for loading files from the same directory that contains the Sympl file doing the importing . You can see in the code excerpt in section 3.6 that Sympl stores into each file's module globals a special name, "__file__", bound to the full path of the source file. If you look at the RuntimeHelpers.SymplImport in runtime.cs, you can see it fetches this name to build a path for importing a library of Sympl code.

The rest of the support for fetching library globals just falls out of emitting GetMember DynamicExpressions. At this point in Sympl's implementation, the SymplGetMemberBinder still

just conveys the name and ignoreCase metadata for binding. That changes after Sympl adds type instantiation, coming up next.

The InvokeMember story is a bit different. Consider this Sympl code:

```
(imports lists)
(lists.append '(a b c) '(1 2 3))
```

Sympl compiles "(lists.append" to an InvokeMember DynamicExpression. The meta-object for the 'lists' module's ExpandoObject gets calls to bind the InvokeMember operation. ExpandoObject's don't supply a 'this' or otherwise have a notion of InvokeMember. They do a GetMember using the name on the binder and then call the binder's FallbackInvoke method. FallbackInvoke can just invoke the object received as the target argument. FallbackInvoke for some languages might check to see if the object is a special runtime callable object with special calling conventions or properties. This is described more fully in section 14.

At this point in Sympl's implementation evolution, its SymplInvokeMemberBinder had a FallbackInvokeMember method that just returned the result of CreateThrow. Its FallbackInvoke method was essentially:

```
    return new DynamicMetaObject(
                    Expression.Dynamic(
                        new SymplInvokeBinder(new
CallInfo(args.Length)),
                        typeof(object),
                        argexprs),
                    targetMO.Restrictions.Merge(
                        BindingRestrictions.Combine(args)));
```

This was not only enough to get cross-library calls working, it became the final implementation. The code in SymplInvokeBinder.FallbackInvoke (discussed in section 5.2) is doing all the work in the nested CallSite resulting from this DynamicExpression.

10 Type instantiation

Sympl has a **new** keyword form. It takes as its first argument an expression that results in a type. Sympl code can get types in one of two ways, importing them from the hosting globals table in the Sympl runtime instance or from the result of a .NET call. One code path has to bind Sympl's TypeModel objects to constructors (TypeModelMetaObject.BindCreateInstance), and the other handles more direct .NET binding (SymplCreateInstanceBinder's FallbackCreateInstance method). The rest of the arguments to the **new** keyword form are used to find an appropriate constructor to call.

10.1 New Keyword Form Code Generation

The analysis and code generation for New is pretty easy since all the work is in runtime binders and the TypeModelMetaObject. Here's the code from etgen.cs:

```
    public static Expression AnalyzeNewExpr(SymplNewExpr expr,
                                            AnalysisScope scope) {

        List<Expression> args = new List<Expression>();
```

```
        args.Add(AnalyzeExpr(expr.Type, scope));
        args.AddRange(expr.Arguments.Select(
                                        a => AnalyzeExpr(a, scope)));
        return Expression.Dynamic(
            scope.GetRuntime().GetCreateInstanceBinder(
                                new CallInfo(expr.Arguments.Length)),
            typeof(object),
            args);
```

AnalyzeNewExpr just analyzes the type expression and all the arguments to build a list of arguments for a DynamicExpression with a SymplCreateInstanceBinder. The metadata for binding is a description of the arguments. For Sympl, this is just an argument count, but note that the count does not include the target type expression even though it is in the 'args' variable passed to the Dynamic factory method.

For now, ignore GetCreateInstanceBinder. Imagine this is just a call to the constructor:

```
new SymplCreateInstanceBinder(CallInfo)
```

GetCreateInstanceBinder produces canonical binders, a single binder instance used on every call site with the same metadata. This is important for DLR L2 caching of rules. See section 18 for how Sympl makes canonical binders and why, and see sites-binders-dynobj-interop.doc for more details on CallSite rule caching.

This DynamicExpression has a result type of object. You might think Sympl could statically type this CallSite to the type of the instance being created. However, the type is unknown until run time when some expression results in a first class type object. Therefore, as with all Dynamic expressions in Sympl, the type is object.

10.2 Binding CreateInstance Operations in TypeModelMetaObject

One path that type instantiation can take in Sympl is from code like the following:

```
(set x (new System.Text.StringBuilder "hello"))
```

In this case one of Sympl's TypeModel objects (representing StringBuilder) flows into the CallSite into which AnalyzeNewExpr's CreateInstance DynamicExpression compiles. Then TypeModelMetaObject's BindCreateInstance produces a rule for creating the StringBuilder.

Here's the code from sympl.cs (the python code is in runtime.py):

```
    public override DynamicMetaObject BindCreateInstance(
               CreateInstanceBinder binder, DynamicMetaObject[] args) {
        var constructors = ReflType.GetConstructors();
        var ctors = constructors.
            Where(c => c.GetParameters().Length == args.Length);
        List<ConstructorInfo> res = new List<ConstructorInfo>();
        foreach (var c in ctors) {
            if (RuntimeHelpers.ParametersMatchArguments(
                                    c.GetParameters(),
                                    args)) {
                res.Add(c);
            }
        }
        if (res.Count == 0) {
            return binder.FallbackCreateInstance(
                        RuntimeHelpers.GetRuntimeTypeMoFromModel(this),
```

```
                    args);
        }
        var restrictions = RuntimeHelpers.GetTargetArgsRestrictions(
                            this, args, true);
        var ctorArgs =
            RuntimeHelpers.ConvertArguments(
            args, res[0].GetParameters());
        return new DynamicMetaObject(
           Expression.New(res[0], ctorArgs),
           restrictions);
```

First BindCreateInstance gets the underlying RuntimeType's constructors and finds those with marching parameter counts. Then Sympl filters for constructors with matching parameters as discussed in section 3.2.4 on TypeModelMetaObject's BindInvokeMember method.

If no constructors match, then Sympl falls back to the language binder after converting the TypeModel to a meta-object representing the RuntimeType object. See the sub section below on instantiating arrays for information on GetRuntimeTypeMoFromModel. Falling back may seem futile, but in addition to other language binders having richer matching rules that might succeed, the convention is to fall back to the binder to get a language-specific error for failing to bind.

Finally, BindCreateInstance gathers restrictions for the rule it produces and converts the arguments, as discussed in section 3.2.5 for TypeModelMetaObject's BindInvokeMember method. Then BindCreateInstance returns the DynamicMetaObject whose restrictions and Expression property (using the Expression Tree New factory) form a rule for creating instances of the target type. The resulting expression does not need to go through EnsureObjectResult since creating an instance necessarily returns objects.

10.3 Binding CreateInstance Operations in FallbackCreateInstance

One path that type instantiation can take in Sympl is from code like the following:

```
;; x is a StringBuilder instance from the previous section
(set y (new (x.GetType) (x.ToString)))
```

In this case one of the DLR's default DynamicMetaObjects for static .NET objects (representing the RuntimeType object for StringBuilder) calls the SymplCreateInstanceBinder's FallbackCreateInstance method.

Here's the code from runtime.cs:

```
public class SymplCreateInstanceBinder : CreateInstanceBinder {
    public SymplCreateInstanceBinder(CallInfo callinfo)
        : base(callinfo) {
    }

    public override DynamicMetaObject FallbackCreateInstance(
                        DynamicMetaObject target,
                        DynamicMetaObject[] args,
                        DynamicMetaObject errorSuggestion) {
        // ... Deleted checking for Defer for now ...
        if (!typeof(Type).IsAssignableFrom(target.LimitType)) {
                return errorSuggestion ??
                    RuntimeHelpers.CreateThrow(
```

```
                target, args, BindingRestrictions.Empty,
                typeof(InvalidOperationException),
                       "Type object must be used when " +
                       "creating instance -- " +
                        args.ToString());
}
var type = target.Value as Type;
Debug.Assert(type != null);
var constructors = type.GetConstructors();
// Get constructors with right arg counts.
var ctors = constructors.
    Where(c => c.GetParameters().Length == args.Length);
List<ConstructorInfo> res = new List<ConstructorInfo>();
foreach (var c in ctors) {
    if (RuntimeHelpers.ParametersMatchArguments(
                            c.GetParameters(),
                            args)) {
        res.Add(c);
    }
}
var restrictions =
    RuntimeHelpers.GetTargetArgsRestrictions(
        target, args, true);
if (res.Count == 0) {
    return errorSuggestion ??
        RuntimeHelpers.CreateThrow(
           target, args, restrictions,
           typeof(MissingMemberException),
                  "Can't bind create instance -- " +
                      args.ToString());
}
var ctorArgs =
    RuntimeHelpers.ConvertArguments(
    args, res[0].GetParameters());
return new DynamicMetaObject(
   Expression.New(res[0], ctorArgs),
   restrictions);
```

Let's first talk about what we aren't talking about now. This code snippet omits some very important code that protects binders and DynamicMetaObjects from infinitely looping due to producing bad rules. It is best to discuss this in one place, so see section 20 for how the infinite loop happens and how to prevent it for all binders.

This coded is nearly the same as TypeModelMetaObject's BindCreateInstance discussed in the previous section. One difference to note is that while DynamicMetaObjects can fall back to binders for errors or potentially more binding searching, the binder creates Throw expressions. FallbackCreateInstance has to gather the target and argument restrictions before deciding to return an error DynamicMetaObject result so that it can ensure it uses the same restrictions it would use in a positive result. See section 6 for a discussion of CreateThrow and restrictions.

10.4 Instantiating Arrays and GetRuntimeTypeMoFromModel

Because Sympl has no built-in notion of arrays (similar to IronPython), you create arrays in Sympl like you would in IronPython:

```
(System.Array.CreateInstance System.String 3)
```

This expression turns into an InvokeMember DynamicExpression. The resulting CallSite gets a Sympl TypeModel object for System.String. This is an example of why ConvertArguments and GetTargetArgsRestrictions conspire to match TypeModel objects to parameters of type Type and convert the former to the latter, as discussed in section 3.2.5.

Here's the code for RuntimeHelpers.GetRuntimeTypeMoFromModel from runtime.cs, which converts a DynamicMetaObject holding a TypeModel value to one holding a RuntimeType value:

```
public static DynamicMetaObject GetRuntimeTypeMoFromModel
                  (DynamicMetaObject typeModelMO) {
    Debug.Assert((typeModelMO.LimitType == typeof(TypeModel)),
                 "Internal: MO is not a TypeModel?!");
    // Get tm.ReflType
    var pi = typeof(TypeModel).GetProperty("ReflType");
    Debug.Assert(pi != null);
    return new DynamicMetaObject(
        Expression.Property(
            Expression.Convert(typeModelMO.Expression,
                               typeof(TypeModel)),
            pi),
        typeModelMO.Restrictions.Merge(
            BindingRestrictions.GetTypeRestriction(
                typeModelMO.Expression, typeof(TypeModel))));
```

The key here is to refrain from lifting the RuntimeType value out of the TypeModel and burning it into the rule as a ConstantExpression. Instead this function returns an Expression in the DynamicMetaObject that fetches the ReflType property from the result of the Expression in the TypeModel's meta-object. This allows the rule to work more generally.

11 SymplGetMemberBinder and Binding .NET Instance Members

Now that Sympl can instantiate types it is worth fleshing out its GetMemberBinder. At runtime, when trying to access a member of a .NET static object, the default .NET meta-object will call FallbackGetMember on SymplGetMemberBinder. This code is much simpler than the code for binding InvokeMember we looked at before. As a reminder, if the object that flows into the CallSite is some dynamic object, then it's DynamicMetaObject's BindGetMember will produce a rule for fetching the member.

Here's the code from runtime.cs, which is described further below:

```
public class SymplGetMemberBinder : GetMemberBinder {
    public SymplGetMemberBinder(string name) : base(name, true) {
    }

    public override DynamicMetaObject FallbackGetMember(
            DynamicMetaObject targetMO,
            DynamicMetaObject errorSuggestion) {
        // ... Deleted checking for COM and need to Defer for now ...
        var flags = BindingFlags.IgnoreCase | BindingFlags.Static |
                    BindingFlags.Instance | BindingFlags.Public;
        var members = targetMO.LimitType.GetMember(this.Name, flags);
        if (members.Length == 1) {
```

```
            return new DynamicMetaObject(
                RuntimeHelpers.EnsureObjectResult(
                  Expression.MakeMemberAccess(
                    Expression.Convert(targetMO.Expression,
                                       members[0].DeclaringType),
                    members[0])),
                BindingRestrictions.GetTypeRestriction(
                                       targetMO.Expression,
                                       targetMO.LimitType));
        } else {
            return errorSuggestion ??
                RuntimeHelpers.CreateThrow(
                    targetMO, null,
                    BindingRestrictions.GetTypeRestriction(
                                           targetMO.Expression,
                                           targetMO.LimitType),
                    typeof(MissingMemberException),
                    "cannot bind member, " + this.Name +
                        ", on object " + targetMO.Value.ToString());
```

Let's first talk about what we aren't talking about now. This code snippet omits the code to check if the target is a COM object and to use built-in COM support. See section 19 for information adding this to your binders. The snippet also omits some very important code that protects binders and DynamicMetaObjects from infinitely looping due to producing bad rules. It is best to discuss this in one place, so see section 20 for how the infinite loop happens and how to prevent it for all binders.

FallbackGetMember uses .NET reflection to get the member with the name in the binder's metadata. If there's exactly one, the binder returns a DynamicMetaObject result with a MemberExpression. It uses the MakeMemberAccess factory which figures out whether the member is a property or a field, and otherwise throws. FallbackGetMember includes a ConvertExpression to ensure the target expression is the member's DeclaringType (expressions tend to flow through Sympl as type object). The operation implementation expression passes through EnsureObjectResult in case it needs to be wrapped to ensure it is strictly typed as assignable to object. For more information, see section 3.2.4.

For restrictions, this binder doesn't need the binding helpers since it only has to test the target object. The restriction uses the target's LimitType, see section 3.2.4 for a discussion of using LimitType over RuntimeType. The restriction does not use DeclaringType because the rule was built using members from LimitType. You might think we need a restriction for the name to prevent the rule from firing for any name. However, this rule is only good on CallSites that point to this binder, and it only returns rules for this one name.

If there isn't exactly one member, then Sympl either uses the suggested result or creates a DynamicMetaObject result that throws an Exception. See section 6 for a discussion of CreateThrow and restrictions. See section 12 for a discussion of errorSuggestion arguments to binders.

12 ErrorSuggestion Arguments to Binder FallbackX Methods

Recall that the convention in the DLR is for dynamic objects to call FallbackX methods so that the language owning the line of code that produced the CallSite can perform .NET static binding.

Furthermore, from the description of the flow of execution when searching for a binding (see section 3.2.1), the DLR calls on DynamicMetaObjects first. You might ask yourself, should the object look for dynamic members first or static members, and how should this negotiation happen with the language binder.

The errorSuggestion parameter provides a mechanism for allowing dynamic objects to choose static first or not. To post-process dynamic members, the DynamicMetaObject can compute a binding rule and pass it to the binder via errorSuggestion. The binder will search for a static member on the .NET type and return a rule if it is successful. The binder will return the errorSuggestion if it is not null to allow the dynamic object to return a rule for a dynamic member. The pre-process dynamic members, the DynamicMetaObject can just return a rule on success and fallback to the binder on failure with a null errorSuggestion.

That's the basic concept, but the dance the DyanamicMetaObject needs to perform with the binder is a bit more intricate for total correctness in the post-process dynamic member case. This is described more fully in the sites-binders-dynobj-interop.doc document on www.codeplex.com/dlr, but here's a quick description. Recall the convention is to fall back to the binder for a language specific error if the dynamic member lookup fails. The meta-object calls on the binder twice to build a complete rule. The first call on the binder gets a successful static member lookup expression or a language-specific error. The DynamicMetaObject uses this expression in its "else" clause for its expression that looks up a dynamic member. Now the DynamicMetaObject calls the binder's FallbackX method again with the combined expression as the errorSuggestion. The binder now binds again (of course, producing exactly what it did before), but it now has an errorSuggestion when the static binding fails that looks up dynamically (after the static lookup) and has a language-specific error branch.

13 SymplSetMemberBinder and Binding .NET Instance Members

At runtime, when trying to set a member of a .NET static object, the default .NET meta-object will call FallbackSetMember on SymplSetMemberBinder. There is more code to setting members than getting them, and by convention, the resulting DynamicMetaObject's expression needs to ensure it returns the value stored. As a reminder, if the object that flows into the CallSite is some dynamic object, then it's DynamicMetaObject's BindSetMember will produce a rule for setting the member.

Here's the code for SymplSetMemberBinder's FallbackSetMember from runtime.cs, which is essentially all there is to the class:

```
public override DynamicMetaObject FallbackSetMember(
            DynamicMetaObject targetMO, DynamicMetaObject value,
            DynamicMetaObject errorSuggestion) {
    // ... Deleted checking for COM and need to Defer for now ...
    var flags = BindingFlags.IgnoreCase | BindingFlags.Static |
                BindingFlags.Instance | BindingFlags.Public;
    var members = targetMO.LimitType.GetMember(this.Name, flags);
    if (members.Length == 1) {
        MemberInfo mem = members[0];
        Expression val;
        if (mem.MemberType == MemberTypes.Property)
            val = Expression.Convert(
                                value.Expression,
```

```
                                    ((PropertyInfo)mem).PropertyType);
            else if (mem.MemberType == MemberTypes.Field)
                val = Expression.Convert(value.Expression,
                                         ((FieldInfo)mem).FieldType);
            else
                return (errorSuggestion ??
                        RuntimeHelpers.CreateThrow(
                            targetMO, null,
                            BindingRestrictions.GetTypeRestriction(
                                targetMO.Expression,
                                targetMO.LimitType),
                            typeof(InvalidOperationException),
                            "Sympl only supports setting Properties " +
                                "and fields at this time."));
            return new DynamicMetaObject(
                RuntimeHelpers.EnsureObjectResult(
                  Expression.Assign(
                    Expression.MakeMemberAccess(
                        Expression.Convert(targetMO.Expression,
                                           members[0].DeclaringType),
                        members[0]),
                    val)),
                 BindingRestrictions.GetTypeRestriction(
                                         targetMO.Expression,
                                         targetMO.LimitType));
        } else {
            return errorSuggestion ??
                RuntimeHelpers.CreateThrow(
                    targetMO, null,
                        BindingRestrictions.GetTypeRestriction(
                                               targetMO.Expression,
                                               targetMO.LimitType),
                    typeof(MissingMemberException),
                    "IDynObj member name conflict.");
```

Let's first talk about what we aren't talking about now. This code snippet omits the code to check if the target is a COM object and to use built-in COM support. See section 19 for information adding this to your binders. The snippet also omits some very important code that protects binders and DynamicMetaObjects from infinitely looping due to producing bad rules. It is best to discuss this in one place, so see section 20 for how the infinite loop happens and how to prevent it for all binders.

FallbackSetMember uses .NET reflection to get the member with the name in the binder's metadata. If there's exactly one, the binder needs to confirm the kind of member for two reasons. The first is to make sure the kind of member is supported for setting in Sympl. The second is due to .NET's reflection API not having a single name for getting the type of values the member can store. FallbackSetMember creates a ConvertExpression to convert the value to the member's type. To be more correct or consistent, the code should check for the property or field being of type Type and the value being a TypeModel, similar to what ConvertArguments does. In this case, it could build an expression like the helper method GetRuntimeTypeMoFromModel does.

The FallbackSetMember returns a DynamicMetaObject result with an Assign node. The left hand side argument is the same expression created in SymplGetMemberBinder, so see that

description for uses of DeclaringType and LimitType in the expression and restrictions. The right hand side is the ConvertExpression discuss above. Expression Tree Assign nodes guarantee returning the value stored, so the binder complies with that convention. The operation implementation expression passes through EnsureObjectResult in case it needs to be wrapped to ensure it is strictly typed as assignable to object. For more information, see section 3.2.4.

For restrictions, this binder doesn't need the binding helpers since it only has to test the target object. Also, since the code doesn't make any decisions based on the property or field's type, having restrictions consistent with the conversions isn't necessary here. If the code were conditional on whether the property or field was assignable, then the result would need more restrictions. You might think we need a restriction for the name to prevent the rule from firing for any name. However, this rule is only good on CallSites that point to this binder, and it only returns rules for this one name.

If there isn't exactly one member, or if the member is not a property or field, then Sympl either uses the suggested result or creates a DynamicMetaObject result that throws an Exception. See section 6 for a discussion of CreateThrow and restrictions. See section 12 for a discussion of errorSuggestion arguments to binders.

14 SymplInvokeMemberBinder and Binding .NET Member Invocations

In Sympl member invocations look like this:

```
some-expr.property.(method "arg").property
(x.tostring)
(x.select (lambda (e) e.Name))
(expr.property.(method).method arg1 arg2) ;; two InvokeMembers
```

At runtime, when trying to invoke a member of a .NET static object, the default .NET meta-object will call FallbackInvokMember on SymplInvokeMemberBinder. This code is nearly the same as TypeModelMetaObject's BindInvokeMember, with a couple of changes. If the object that flows into the CallSite is a dynamic object, then its meta-object's BindInvokeMember will produce a rule for invoking the member, as TypeModel's meta-object in Hello World.

InvokeMemberBinders need FallbackInvokeMember and FallbackInvoke methods. The second came up in Sympl when cross-library function invocation started working (see section 9) due to ExpandObject's implementation of InvokeMember. Some languages and dynamic objects do not or cannot perform InvokeMember operations. They can turn InvokeMember into a GetMember and a call to the InvokeMemberBinder's FallbackInvoke method. IronPython does this, passing a DynamicMetaObject that results in a callable object that is closed over the InvokeMember operation's target object.

14.1 FallbackInvokeMember

This code is nearly the same as TypeModelMetaObject's BindInvokeMember, with a couple of changes, such as using CreateThrow rather than falling back to a binder. The details are discussed below the code for SymplInvokeMemberBinder's FallbackInvokeMember from runtime.cs:

```
public override DynamicMetaObject FallbackInvokeMember(
        DynamicMetaObject targetMO, DynamicMetaObject[] args,
```

```
            DynamicMetaObject errorSuggestion) {
        // ... Deleted checking for COM and need to Defer for now ...
        var flags = BindingFlags.IgnoreCase | BindingFlags.Instance |
                    BindingFlags.Public;
        var members = targetMO.LimitType.GetMember(this.Name, flags);
        if ((members.Length == 1) && (members[0] is PropertyInfo ||
                                      members[0] is FieldInfo)){
            // Code deleted, not implemented yet.
        } else {
            // Get MethodInfos with right arg counts.
            var mi_mems = members.
                Select(m => m as MethodInfo).
                Where(m => m is MethodInfo &&
                           ((MethodInfo)m).GetParameters().Length ==
                               args.Length);
            List<MethodInfo> res = new List<MethodInfo>();
            foreach (var mem in mi_mems) {
                if (RuntimeHelpers.ParametersMatchArguments(
                                       mem.GetParameters(), args)) {
                    res.Add(mem);
                }
            }
            var restrictions =
RuntimeHelpers.GetTargetArgsRestrictions(
                                                  targetMO, args,
false);
            if (res.Count == 0) {
                return errorSuggestion ??
                    RuntimeHelpers.CreateThrow(
                        targetMO, args, restrictions,
                        typeof(MissingMemberException),
                        "Can't bind member invoke -- " +
                             args.ToString());
            }
            var callArgs = RuntimeHelpers.ConvertArguments(
                                        args,
                                        res[0].GetParameters());
            return new DynamicMetaObject(
               RuntimeHelpers.EnsureObjectResult(
                 Expression.Call(
                    Expression.Convert(targetMO.Expression,
                                       targetMO.LimitType),
                    res[0], callArgs)),
               restrictions);
```

Let's first talk about what we aren't talking about now. This code snippet omits the code to check if the target is a COM object and to use built-in COM support. See section 19 for information adding this to your binders. The snippet also omits some very important code that protects binders and DynamicMetaObjects from infinitely looping due to producing bad rules. It is best to discuss this in one place, so see section 20 for how the infinite loop happens and how to prevent it for all binders.

Sympl takes the name from the binder's metadata and looks for all public, instance members on the LimitType of the value represented by this meta-object. You could just as easily decide to

bind to static members here as well if your language had those semantics. Because Sympl is a case-INsensitive language, the flags include IgnoreCase.

You could also bind to data members that held sub types of Delegate. You'd then emit code to fetch the member, similar to the expression in GetRuntimeTypeMoFromModel, and use an Invoke DynamicExpression. This nests a CallSite and defers binding to SymplInvokeBinder's FallbackInvoke method, similar to what SymplInvokeMemberBinder's FallbackInvoke does. Sympl doesn't bind to data members with delegate values just to simplify the sample.

FallbackInvokeMember filters for only the members that are MethodInfos and have the right number of arguments. Then the binding logic filters for the MethodInfos that have parameters that can be bound given the kinds of arguments present at this invocation of the call site. See section 3.2.4 for a discussion of matching parameters in the filtered MethodInfos and choosing the overload to invoke because it is the same logic here.

If FallbackInvokeMember finds no matching MethodInfos, then it either uses the suggested result or creates a DynamicMetaObject result that throws an Exception. See section 6 for a discussion of CreateThrow and restrictions. ErrorSuggestion is discussed in section 12.

The rest of this function is almost exactly TypeModelMetaObject's BindInvokeMember. See section 3.2.5 for a discussion of restrictions and argument conversions for the resulting DynamicMetaObject's MethodCallExpression. See section 3.2.4 for a discussion of EnsureObjectResult. One difference to point out is that FallbackInvokeMember needs to convert the target object to the specific LimitType of the DynamicMetaObject. See section 3.2.4 for a discussion of using LimitType over RuntimeType. It may seem odd to convert the object to the type that LimitType reports it to be, but the type of the meta-object's expression might be more general and require an explicit Convert node to satisfy the strict typing of the Expression Tree factory or the actual emitted code that executes. The Expression Tree compiler removes unnecessary Convert nodes.

14.2 FallbackInvoke

This method exists for languages and dynamic objects do not or cannot perform InvokeMember operations. Instead, they can turn InvokeMember into a GetMember and a call to the InvokeMemberBinder's FallbackInvoke method. The DLR's ExpandoObjects do this, which section 9 discusses to get cross-module top-level function calls working. IronPython uses FallbackInvoke, passing a DynamicMetaObject that results in a callable object that is closed over the InvokeMember operation's target object.

Here's the code for SymplInvokeMemberBinder's FallbackInvoke from runtime.cs:

```
    public override DynamicMetaObject FallbackInvoke(
            DynamicMetaObject targetMO, DynamicMetaObject[] args,
            DynamicMetaObject errorSuggestion) {
        var argexprs = new Expression[args.Length + 1];
        for (int i = 0; i < args.Length; i++) {
            argexprs[i + 1] = args[i].Expression;
        }
        argexprs[0] = targetMO.Expression;
        return new DynamicMetaObject(
                       Expression.Dynamic(
                           new SymplInvokeBinder(
                                   new CallInfo(args.Length)),
```

```
                typeof(object),
                argexprs),
            targetMO.Restrictions.Merge(
                BindingRestrictions.Combine(args)));
```

The target meta-object passed to FallbackInvoke is a callable object, not the target object passed to FallbackInvokeMember that might have a member with the name in the binder's metadata. There are no checks here for COM objects because no callable COM object should flow into FallbackInvoke.

or whether FallbackInvoke needs to Defer to a nested CallSite. No , and FallbackInvoke effectively always defers to a nested CallSite (section 20).

FallbackInvoke just bundles the target and args into an array to pass to the Dynamic factory method. By returning a DynamicMetaObject with a DynamicExpression, FallbackInvoke is creating a nested CallSite, so regardless of whether any argument meta-objects need to defer computation, this code works (see section 20 for information on Defer). As with all CallInfos, the count of arguments does not include the target object even though it is in the arguments array passed to Dynamic.

The restrictions are simple too, but it is important to collect them and propagate them to the new DynamicMetaObject. There's no need to add other restrictions since no other argument conditions were used to compute a binding. This method is just deferring to the SymplInvokeBinder's FallbackInvoke method to figure out a binding.

Note, Sympl calls the SymplInvokeBinder constructor here rather than calling GetInvokeBinder from an instance of the Sympl runtime class (see section 18). This means the CallSite resulting from the DynamicExpression will not share any L2 caching with other call sites. At this point in the execution of a Sympl program, Sympl binders do not have access to the Sympl instance on whose behalf the Sympl code is running. Sympl could have added a property to the binder to stash the Sympl runtime instance when creating the InvokeMember DynamicExpression in AnalyzeFunCallExpr, but you want to do that very carefully to make sure you don't hold onto working set unintentionally. Sympl could have used GetInvokeBinder in AnalyzeFunCallExpr and tucked one into the SymplInvokeMemberBinder instance in case it was needed. There are various ways to handle this, but for the sample, losing L2 cache sharing here is acceptable.

15 Indexing Expressions: GetIndex and SetIndex

Sympl supports indexing its built-in lists, arrays, and indexers/indexed properties. Expression Trees v1 had an ArrayIndex factory that would return either a BinaryExpression for single-dimensional arrays or a MethodCallExpression for multi-dimensional arrays. These now exist only for LINQ backward compatibility. All new code should use the ArrayAccess or MakeIndex factories that return IndexExpressions. Expression Trees v2 support IndexExpressions everywhere, including the left hand side of assignments and as byref arguments.

SymplGetIndexBinder and SymplSetIndexBinder both use the RuntimeHelpers method GetIndexingExpression. It does most of the work for FallbackGetIndex. FallbackSetIndex has to do some extra work.

15.1 SymplGetIndexBinder's FallbackGetIndex

Here's the code from runtime.cs, which is described further below:

```
public override DynamicMetaObject FallbackGetIndex(
             DynamicMetaObject target, DynamicMetaObject[] indexes,
             DynamicMetaObject errorSuggestion) {
    // ... Deleted checking for COM and need to Defer for now ...
    // Give good error for Cons.
    if (target.LimitType == typeof(Cons)) {
        if (indexes.Length != 1)
            return errorSuggestion ??
                RuntimeHelpers.CreateThrow(
                     target, indexes, BindingRestrictions.Empty,
                     typeof(InvalidOperationException),
                     "Indexing list takes single index.  " +
                        "Got " + indexes.Length.ToString());
    }
    var indexingExpr =
            RuntimeHelpers.EnsureObjectResult(
                RuntimeHelpers.GetIndexingExpression(target,
                                                     indexes));
    var restrictions = RuntimeHelpers.GetTargetArgsRestrictions(
                                            target, indexes, false);
    return new DynamicMetaObject(indexingExpr, restrictions);
```

Let's first talk about what we aren't talking about now. This code snippet omits the code to check if the target is a COM object and to use built-in COM support. See section 19 for information adding this to your binders. The snippet also omits some very important code that protects binders and DynamicMetaObjects from infinitely looping due to producing bad rules. It is best to discuss this in one place, so see section 20 for how the infinite loop happens and how to prevent it for all binders.

As we said before, GetIndexingExpression does most of the work here. Before calling it, FallbackGetIndex checks if the indexing is for Sympl built-in lists and whether the argument count is right. After calling GetIndexingExpression, the expression passes through EnsureObjectResult in case it needs to be wrapped to ensure it is strictly typed as assignable to object. For more information, see section 3.2.4. FallbackGetIndex uses the binding helper GetTargetArgsRestrictions to get restrictions and returns the resulting DynamicMetaObject with the indexing expression and restrictions.

15.2 GetIndexingExpression

SymplGetIndexBinder and SymplSetIndexBinder both use the RuntimeHelpers method GetIndexingExpression. It does most of the work FallbackGetIndex. FallbackSetIndex has to do some extra work.

Here's the code from RuntimeHelpers in runtime.cs, which is further explained below:

```
public static Expression GetIndexingExpression(
                              DynamicMetaObject target,
                              DynamicMetaObject[] indexes) {
    Debug.Assert(target.HasValue &&
                 target.LimitType != typeof(Array));
```

```
        var indexExpressions = indexes.Select(
            i => Expression.Convert(i.Expression, i.LimitType))
            .ToArray();
        // HANDLE CONS
        if (target.LimitType == typeof(Cons)) {
            // Call RuntimeHelper.GetConsElt
            var args = new List<Expression>();
            // The first argument is the list
            args.Add(
                Expression.Convert(
                    target.Expression,
                    target.LimitType)
            );
            args.AddRange(indexExpressions);
            return Expression.Call(
                typeof(RuntimeHelpers),
                "GetConsElt",
                null,
                args.ToArray());
        // HANDLE ARRAY
        } else if (target.LimitType.IsArray) {
            // the target has an array type
            return Expression.ArrayAccess(
                Expression.Convert(target.Expression,
                                   target.LimitType),
                indexExpressions
            );
        // HANDLE INDEXERS
        } else {
            var props = target.LimitType.GetProperties();
            var indexers = props.
                Where(p => p.GetIndexParameters().Length >
0).ToArray();
            indexers = indexers.
                Where(idx => idx.GetIndexParameters().Length ==
                             indexes.Length).ToArray();

            var res = new List<PropertyInfo>();
            foreach (var idxer in indexers) {
                if (RuntimeHelpers.ParametersMatchArguments(
                                       idxer.GetIndexParameters(),
                                       indexes)) {
                    // all parameter types match
                    res.Add(idxer);
                }
            }
            if (res.Count == 0) {
                return Expression.Throw(
                    Expression.New(
                        typeof(MissingMemberException)
                            .GetConstructor(new Type[]
                                                { typeof(string) }),
                        Expression.Constant(
                           "Can't bind because there is no " +
                               "matching indexer.")
                    )
                );
```

```
            }
            return Expression.MakeIndex(
                Expression.Convert(target.Expression,
target.LimitType),
                res[0], indexExpressions);
```

The first thing GetIndexingExpression does is get ConvertExpressions for all the indexing arguments. It converts them to the specific LimitType of the DynamicMetaObject. See section 3.2.4 for a discussion of using LimitType over RuntimeType. It may seem odd to convert the object to the type that LimitType reports it to be, but the type of the meta-object's expression might be more general and require an explicit Convert node to satisfy the strict typing of the Expression Tree factory or the actual emitted code that executes. The Expression Tree compiler removes unnecessary Convert nodes.

The first kind of indexing Sympl supports is its built-in lists (target's LimitType is Cons). In this case the GetIndexingExpression creates a ConvertExpression for the target to its LimitType, just like the index arguments discussed above. Then it takes the converted target and argument expressions to create a MethodCallExpression for RuntimeHelpers.GetConsElt. You can see this runtime helper in runtime.cs.

The second kind of indexing Sympl supports is arrays (target's LimitType IsArray). In this case GetIndexingExpression also creates a ConvertExpression for the target to its LimitType. Then it takes the converted target and argument expressions to create an IndexExpression.

The third kind of indexing Sympl supports is looking for an indexer or indexed property. GetIndexingExpression gets the target's properties and filters for those whose parameter count matches the index arguments count. Then it filters for matching parameter types, which is described in section 3.2.4. If no properties match, GetIndexingExpression returns a Throw expression (doesn't use CreateThrow here since it returns a DynamicMetaObject). Finally GetIndexingExpression calls MakeIndex to return an IndexExpression. It also converts the target to its LimitType, as discussed above.

15.3 SymplSetIndexBinder's FallbackSetIndex

Here's the code from runtime.cs, which is described further below:

```
    public override DynamicMetaObject FallbackSetIndex(
                DynamicMetaObject target, DynamicMetaObject[] indexes,
                DynamicMetaObject value,
                DynamicMetaObject errorSuggestion) {
        // ... Deleted checking for COM and need to Defer for now ...
        Expression valueExpr = value.Expression;
        if (value.LimitType == typeof(TypeModel)) {
            valueExpr = RuntimeHelpers.GetRuntimeTypeMoFromModel(value)
                                      .Expression;
        }
        Debug.Assert(target.HasValue &&
                     target.LimitType != typeof(Array));
        Expression setIndexExpr;
        if (target.LimitType == typeof(Cons)) {
            if (indexes.Length != 1) {
                return errorSuggestion ??
                    RuntimeHelpers.CreateThrow(
```

```
                        target, indexes, BindingRestrictions.Empty,
                        typeof(InvalidOperationException),
                        "Indexing list takes single index.  " +
                           "Got " + indexes);
            }
            // Call RuntimeHelper.SetConsElt
            List<Expression> args = new List<Expression>();
            // The first argument is the list
            args.Add(
                Expression.Convert(
                    target.Expression,
                    target.LimitType)
            );
            // The second argument is the index.
            args.Add(Expression.Convert(indexes[0].Expression,
                                        indexes[0].LimitType));
            // The last argument is the value
            args.Add(Expression.Convert(valueExpr, typeof(object)));
            // Sympl helper returns value stored.
            setIndexExpr = Expression.Call(
                typeof(RuntimeHelpers),
                "SetConsElt",
                null,
                args.ToArray());
        } else {
            Expression indexingExpr =
                        RuntimeHelpers.GetIndexingExpression(target,

indexes);
            setIndexExpr = Expression.Assign(indexingExpr, valueExpr);
        }
        BindingRestrictions restrictions =
             RuntimeHelpers.GetTargetArgsRestrictions(target, indexes,
                                                      false);
        return new DynamicMetaObject(
            RuntimeHelpers.EnsureObjectResult(setIndexExpr),
            restrictions);
```

Let's first talk about what we aren't talking about now. This code snippet omits the code to check if the target is a COM object and to use built-in COM support. See section 19 for information adding this to your binders. The snippet also omits some very important code that protects binders and DynamicMetaObjects from infinitely looping due to producing bad rules. It is best to discuss this in one place, so see section 20 for how the infinite loop happens and how to prevent it for all binders.

At a high level FallbackSetIndex examines the value meta-object, then gets an expression to set the index, then forms restrictions, and lastly returns the resulting DynamicMetaObject representing the bound operation. The value processing is just a check for whether to convert TypeModel to a RuntimeType meta-object. See section 3.2.5 for a discussion of the restrictions. The only difference here is the false value to get a type restriction on the target.

To determine the indexing expression, FallbackSetIndex checks for the target being a Cons. If it is, the binder needs to call the RuntimeHelpers.SetConsElt method. The binder first checks the number of arguments and whether it should call CreateThrow. See section 6 for a discussion of CreateThrow and restrictions. As discussed with GetIndexingExpression, the binder creates a

ConvertExpression to convert the target to its LimitType, and does the same for the index arguments. The binder converts the value to the Type object because that's what the runtime helper takes. Finally, the resulting indexing expression for setting a Cons element is a MethodCallExpression for SetConsElt. This helper returns the value stored to be in compliance with the convention for binders and meta-objects.

In the alternative branch, FallbackSetIndex uses the GetIndexingExpression binding helper. The binder then wraps that in an Assign node. This guarantees returning the value stored also.

In either case, the operation implementation expression passes through EnsureObjectResult in case it needs to be wrapped to ensure it is strictly typed as assignable to object. For more information, see section 3.2.4.

16 Generic Type Instantiation

Instantiating .NET generic types in Sympl may not be obvious. It is similar to IronPython, but IronPython provides a bit of syntactic sugar to make it easier. Here are a couple of examples in Sympl:

```
;;; Create List<int>
(set types (system.array.CreateInstance system.type 1))
(set (elt types 0) System.Int32)
(new (system.collections.generic.list`1.MakeGenericType types)))

;;; Create a Dictionary<string,int>
(set types (system.array.CreateInstance system.type 2))
(set (elt types 0) system.string)
(set (elt types 1) system.int32)
(new (system.collections.generic.dictionary`2.MakeGenericType
      types)))
```

Sympl could do two things to make this better. The first is to provide nicer name mappings to hide the .NET true names of types, that is, hide the backquote-integer naming. Sympl's identifiers are so flexible, this wasn't necessary or worth demonstrating in the example. The second affordance would be some keyword form like the following:

```
(new (generic-type system.collections.generic.list system.int32))
```

IronPython supports indexing syntax on generic type names, and it also supports cleaner name mapping to avoid the backquote characters in its identifiers.

The above working examples in Sympl work because of the binding logic discussed several times regarding mapping TypeModel to RuntimeType. See ParametersMatchArguments and GetRuntimeTypeMoFromModel discussions in other sections.

17 Arithmetic, Comparison, and Boolean Operators

Sympl supports addition, subtraction, multiplication, division, less than, greater than, equals, and not equal as the obvious corresponding Expression Tree nodes. It support **and**, **or**, and **not** logical operators (see section 21.5 for their semantics). Here are example Sympl expressions:

```
(+ 3 5)
(+ (* 10 x) (/ y 2))
```

```
    (< x max-fixnum)
    (or arg default-value)
    (= (not (and a b))
       (or (not a) (not b)))
```

17.1 Analysis and Code Generation for Binary Operations

All these operators are all keyword forms that AnalyzeBinaryExpr in etgen.cs analyzes. It picks off **and** and **or** to handle specially since they are conditionally evaluating and need to support Sympl's truth value semantics (see section 7.3 on **IF** expressions). Otherwise, AnalyzeBinaryExpr just emits a BinaryOperation DynamicExpression using the operator stored in the SymplBinaryExpr AST node.

Here's the code for AnalyzeBinaryExpr, which is discussed further below:

```
    public static Expression AnalyzeBinaryExpr(SymplBinaryExpr expr,
                                               AnalysisScope scope) {
        if (expr.Operation == ExpressionType.And) {
            return AnalyzeIfExpr(
                new SymplIfExpr(
                    expr.Left, expr.Right, null),
                scope);
        } else if (expr.Operation == ExpressionType.Or) {
            // (Let* (tmp1 x)
            //    (If tmp1 tmp1
            //        (Let* (tmp2 y) (If tmp2 tmp2))))
            IdOrKeywordToken tmp2 = new IdOrKeywordToken(
                "__tmpLetVariable2");
            var tmpExpr2 = new SymplIdExpr(tmp2);
            var binding2 = new LetBinding(tmp2, expr.Right); ;
            var ifExpr2 = new SymplIfExpr(
                tmpExpr2, tmpExpr2, null);
            var letExpr2 = new SymplLetStarExpr(
                    new[] { binding2 },
                    new[] { ifExpr2 });
            // Build outer let*
            IdOrKeywordToken tmp1 = new IdOrKeywordToken(
                "__tmpLetVariable1");
            var tmpExpr1 = new SymplIdExpr(tmp1);
            LetBinding binding1 = new LetBinding(tmp1, expr.Left); ;
            SymplExpr ifExpr1 = new SymplIfExpr(
                tmpExpr1, tmpExpr1, letExpr2);
            return AnalyzeLetStarExpr(
                new SymplLetStarExpr(
                    new[] { binding1 },
                    new[] { ifExpr1 }
                ),
                scope
            );
        }
        return Expression.Dynamic(

scope.GetRuntime().GetBinaryOperationBinder(expr.Operation),
                typeof(object),
                AnalyzeExpr(expr.Left, scope),
                AnalyzeExpr(expr.Right, scope));
```

Because **and** and **or** have equivalent semantic to **IF** (with some temporary bindings for **or**), the code above creates ASTs for **IF** and re-uses the AnalyzeIfExpr and AnalyzeLetStarExpr. Sympl could also have "open coded" the equivalent Expression Tree code generation here.

If the operation is other than **and** and **or**, AnalyzeBinaryExpr emits a DynamicExpression with a SymplBinaryOperationBinder. See section 18 for a discussion of why this method calls GetBinaryOperationBinder rather than just calling the constructor. The reason Sympl uses a DynamicExpression when it only supports the static built-in semantics of Expression Trees is for interoperability with dynamic languages from other languages or libraries that might flow though a Sympl program.

17.2 Analysis and Code Generation for Unary Operations

The only unary operation Sympl supports is logical negation. Here's the code for AnalyzeUnaryExpr in etgen.cs:

```
public static Expression AnalyzeUnaryExpr(SymplUnaryExpr expr,
                                          AnalysisScope scope) {
    if (expr.Operation == ExpressionType.Not) {
        return Expression.Not(WrapBooleanTest(
                                  AnalyzeExpr(expr.Operand,
                                              scope)));
        }
    return Expression.Dynamic(
                scope.GetRuntime()
                     .GetUnaryOperationBinder(expr.Operation),
                typeof(object),
                AnalyzeExpr(expr.Operand, scope));
```

Execution never reaches the DynamicExpression result. This is there as plumbing and an example should Sympl support other unary operations, such as binary or arithmetic negation.

Sympl's logical **not** translates directly to an Expression Tree Not node as long as the operand expression has a Boolean type. Because of Sympl's truth semantics (see section 7.3 on **IF** expressions), AnalyzeUnaryExpr calls WrapBooleanTest which results in an Expression with Type bool.

17.3 SymplBinaryOperationBinder

Binding binary operations is pretty easy in Sympl. Here's the code for the binder's FallbackBinaryOperation in runtime.cs:

```
public override DynamicMetaObject FallbackBinaryOperation(
            DynamicMetaObject target, DynamicMetaObject arg,
            DynamicMetaObject errorSuggestion) {
    var restrictions = target.Restrictions.Merge(arg.Restrictions)
            .Merge(BindingRestrictions.GetTypeRestriction(
                target.Expression, target.LimitType))
            .Merge(BindingRestrictions.GetTypeRestriction(
                arg.Expression, arg.LimitType));
    return new DynamicMetaObject(
        RuntimeHelpers.EnsureObjectResult(
```

```
                Expression.MakeBinary(
                  this.Operation,
                  Expression.Convert(target.Expression,
target.LimitType),
                  Expression.Convert(arg.Expression, arg.LimitType))),
              restrictions);
```

This function gathers all the restrictions for the arguments and merges them with the target's restrictions. Then it also merges in restrictions to ensure the arguments are the same LimitType as they have during this pass through the CallSite; the rule produced is only good for those types.

Then FallbackBinaryOperation returns a DynamicMetaObject with a BinaryExpression. The arguments are wrapped in ConvertExpressions to ensure they have the strict typing required for the BinaryExpression node returned. Recall that the argument's expression type may be more general than the actual LimitType they have at run time. The result BinaryExpression also passes through EnsureObjectResult in case it needs to be wrapped to ensure it is strictly typed as assignable to object. For more information, see section 3.2.4.

17.4 SymplUnaryOperationBinder

As stated above, Sympl never really uses its UnaryOperationBinder. It exists a plumbing for future unary features and as an example. It is exactly like FallbackBinaryOperation except that it only has one argument to process.

Here's the code form runtime.cs.

```
    public override DynamicMetaObject FallbackUnaryOperation(
                DynamicMetaObject target,
                DynamicMetaObject errorSuggestion) {
        return new DynamicMetaObject(
            RuntimeHelpers.EnsureObjectResult(
              Expression.MakeUnary(
                this.Operation,
                Expression.Convert(target.Expression,
target.LimitType),
                target.LimitType)),
             target.Restrictions.Merge(
                 BindingRestrictions.GetTypeRestriction(
                     target.Expression, target.LimitType)));
```

18 Canonical Binders or L2 Cache Sharing

This section will briefly describe the DLR's caching for dynamic dispatch and explain the code that calls GetXBinder helpers rather than calling a SymplXBinder constructor. For a full discussion of caching, see sites-binders-dynobj-interop.doc on www.codeplex.com/dlr .

As described in section 3.2.1, when you use a DynamicExpression, the Expression Tree compiler creates a CallSite to cache rules for how to perform the CallSite's operation. The delegate that is stored in the CallSite's Target property is considered to be the L0 cache. When the rule in this delegate fails, the CallSite looks to the rules it has cached on the CallSite object (for now, up to ten such rules). These ten rules are considered to be the L1 cache.

When none of the rules seen previously by a CallSite match the current arguments, before the CallSite starts the binding search described in section 3.2.1, it looks at internal state of the binder attached to the CallSite. A binder instance caches up to 100 rules that it has produced. This is considered to be the L2 cache, and it can be shared across multiple CallSites. The idea is that, for example, any call site performing addition with the same metadata on the binder should share rules across an entire program. It is highly likely they are all just adding integers, strings, or a small number of combinations of argument types. Because producing a rule is expensive, you want to save that work once one call site has paid the price.

Sympl maps all binders with the same operation and metadata to a single such binder. It does this per Sympl runtime instance. Having canonical binders per Sympl runtime instance isn't really needed, but it is a good example of what real languages likely need to do. There might be global options per runtime instance, such as whether integers roll over to bignums, or whether division is truncating or float-producing, and so on.

The code for the GetXBinder methods is pretty straightforward. This section just makes a couple of points about how it creates canonical binders, but you can see the code in sympl.cs, which is well commented. You need to hash on the binder type's metadata, which is a little bit of work for InvokeMember binders. All the others just hash on the member name string or a CallInfo object (for which the DLR provides GetHashCode and Equals). For InvokeMember binders, you need to make a key class, InvokeMemberBinderKey, that holds a name and a CallInfo, and then implement GetHashCode and Equals.

Sympl hashes name strings with default .NET semantics and '==' equality, thus comparing case-senstively. The reason is that while Sympl is case-INsensitive, it is case-preserving in the binder metadata, as explained in section 4. This allows for more opportunity of language and library interoperability. Since a DynamicMetaObject might return a rule for looking up a member based on the case-sensitive spelling (that is, ignoring the ignoreCase metadata), Sympl needs to ensure no two CallSites with different spellings could share L2 rules.

19 Binding COM Objects

This support has been moved to Codeplex only (after the writing of the Sympl sample and this document). This is in the Microsoft.Dynamic.dll. It is the COM binding C# uses, but C# uses it privately now. Sympl has been updated with a new using statement to continue working. Note that this is the one bit of functionality that Sympl demonstrates in the base csharp directory that is not shipping functionality in .NET 4.0.

Having your language work late bound using IDispatch on COM objects is very easy. You can look at any of the FallbackX methods to see how Sympl does this, but here's an example from FallbackGetMember in runtime.cs:

```
public override DynamicMetaObject FallbackGetMember(
        DynamicMetaObject targetMO,
        DynamicMetaObject errorSuggestion) {

    // First try COM binding.
    DynamicMetaObject result;
    if (ComBinder.TryBindGetMember(this, targetMO, out result,
                                   true)) {
        return result;
```

You can look at the code for ComBinder, and for each TryBindX member, add coded similar to the above to your corresponding XBinder.

TryBindGetMember is the only one with the odd Boolean flag. It accommodates the distinction between languages like Python and C#. Python has a strict model of getting a member and then calling it; there is no invoke member. If ComBinder.TryBindGetMember can prove the member is parameterless and a data member, then it can produce a rule for eager evaluation. If either test fails, then passing true means TryBindGetMember returns a rule for lazily fetching the value as a callable wrapper. When the flag is false, ComBider always eagerly evaluates, and if the member requires parameters that aren't available, then ComBinder returns a rule that throws. C# passes false here. It doesn't really matter for Sympl, which just copied this code from IronPython.

20 Using Defer When MetaObjects Have No Value

Binders should always protect themselves by checking whether all the DynamicMetaObjects passed to their FallbackX methods have values (see HasValue property). As a conservative implementation, if any meta-object is lacking a concrete value, then call the binder's Defer method on all the meta-objects passed in. The Defer method results in a DynamicMetaObject with an Expression that creates a nested CallSite. This allows the DynamicMetaObject expressions to be evaluated and to flow into the nested site with values. If binders did not protect themselves in this way, an infinite loop results as they produce a rule that fails when the CallSite executes it, which forces a binding update, which causes the same target DynamicMetaObject to fallback with no value, which causes the binder to produce a bad rule, and so on.

Let's look at why DynamicMetaObjects might not have values and then look at a real situation in the Sympl code that infinitely loops without HasValue checks. Sometimes dynamic languages produce partial results or return rules for performing part of an operation but then need the language binder to do the rest. A prime example when interoperating with IronPython is how it handles InvokeMember. It will fetch the member, package it in an IronPython callable object represented as an IDynamicMetaObjectProvider, and then call FallbackInvoke on your binder. The dynamic object has no value, just an expression capturing how to get the callable object. Anytime a language or DynamicMetaObject needs to return a dynamic object to a FallbackX method, it should never place a value in the DynamicMetaObject it passes to the FallbackX method. Doing so would cause the FallbackX method to try to do static .NET binding on the object, but of course, that's not right since the static nature of the object is the carrier for the dynamic nature.

For a concrete example within Sympl's implementation, consider this line of code from indexing.sympl:

```
(set l (new (System.Collections.Generic.List`1.MakeGenericType
            types)))
```

Without SymplInvokeMemberBinder.FallbackInvokeMember testing whether the DynamicMetaObjects passed to it have values and deferring, it would infinitely loop with TypeModelMetaObject.BindInvokeMember for the "MakeGenericType" member. BindINvokeMember would fall back with no value (as shown below), and the binder would

produce a binding result whose rule restrictions would fail. The CallSite would then try to update the rule. The TypeModelMetaObject would fall back again with no value, and this would repeat forever.

Before adding the HasValue check to the Sympl binders, the runtime helper function GetRuntimeTypeMoFromModel had to supply a value to the meta-object it produced. This is not always possible or the right thing to do, but it worked for GetRuntimeTypeMoFromModel because it could produce a regular .NET static object for the binder that was consistent with an instance restriction on the type object.

```
public static DynamicMetaObject GetRuntimeTypeMoFromModel
    (DynamicMetaObject typeModelMO) {
     Debug.Assert((typeModelMO.LimitType == typeof(TypeModel)),
                  "Internal: MO is not a TypeModel?!");
     // Get tm.ReflType
     var pi = typeof(TypeModel).GetProperty("ReflType");
     Debug.Assert(pi != null);
     return new DynamicMetaObject(
         Expression.Property(
             Expression.Convert(typeModelMO.Expression,
                                typeof(TypeModel)),
             pi),
         typeModelMO.Restrictions.Merge(
             BindingRestrictions.GetTypeRestriction(
                 typeModelMO.Expression, typeof(TypeModel)))//,
         //((TypeModel)typeModelMO.Value).ReflType
     );
```

When the highlight code above gets comment out, the code below is what prevents the FallbackInvokeMember function from infintely looping through the CallSite, trying to bind with the TypeModel's meta-object:

```
public override DynamicMetaObject FallbackInvokeMember(
        DynamicMetaObject targetMO, DynamicMetaObject[] args,
        DynamicMetaObject errorSuggestion) {
    // ... code deleted for example ...
    if (!targetMO.HasValue || args.Any((a) => !a.HasValue)) {
        var deferArgs = new DynamicMetaObject[args.Length + 1];
        for (int i = 0; i < args.Length; i++) {
            deferArgs[i + 1] = args[i];
        }
        deferArgs[0] = targetMO;
        return Defer(deferArgs);
```

Every FallbackX method on all your binders should protect themselves by checking all arguments for HasValue. If HasValue is false for any, then call Defer as shown in the Sympl binders. Note, the above code is the most complicated, and for FallbackGetMember, it is just this:

```
        if (!targetMO.HasValue) return Defer(targetMO);
```

21 SymPL Language Description

The following sub sections contain very brief descriptions of language features and semantics. Some sections have more details, such as the description of **Import**, but most loosely describe the construct, such as try-catch.

21.1 High-level

Sympl is expression-based. The last expression executed in a function produces the return value of the function. All control flow constructs have result values. The executed branch of an **If** provides the value of **If**. If the **break** from a loop has a value, it is the result of the **loop** expression; otherwise, nil is.

There is support for top-level functions and lambda expressions. There is no **flet**, but you can get recursive lambdas by letting a variable be nil, then setting it to the result of a lambda expression, which can refer to the variable for recursive calls.

Sympl is a case-INsensitive language for identifiers.

Sympl does not demonstrate class definitions. See section 21.7 for more information.

21.2 Lexical Aspects

Identifiers may contain any character except the following:

```
( ) " ; , ' @ \ .
```

Sympl should disallow backquote in case adding macro support later would be interesting. For now it allows backquote in identifiers since .NET raw type names allow that character. Like many languages, Sympl could provide a mapping from simpler names to those names that include backquote (for example, map List to List`1). Doing so didn't seem to add any teaching about the DLR, and Sympl identifiers are flexible enough to avoid the work now.

Due to the lack of infix operators (other than period), there's no issue with expressions like "a+b", which is an identifier in Sympl.

An identifier may begin with a backslash, when quoting a keyword for use as an identifier.

Sympl has integer and floats (.NET doubles) for numeric literals.

Doubles aren't in yet.

Strings are delimited by double quotes and use backslash as an escape character.

Apostrophes quote list literals, symbols, integers, and other literals. It is only necessary to quote lists and symbols (distinguishes them from identifiers).

Comments are line-oriented and begin with a semi-colon.

21.3 Built-in Types

Sympl has these built-in types:

- Integers -- .NET int32 (no bignums)

- Floats -- .NET doubles NOT ADDED YET
- Strings -- .NET immutable strings
- Boolean -- **true** and **false** literal keywords for .NET interop. Within Sympl, anything that is not **nil** or **false** is true.
- Lists -- Cons cell based lists (with First and Rest instead of Car and Cdr :-))
- Symbols -- interned names in a Sympl runtime instance
- Lambdas -- .NET dynamic methods via Expression Trees v2 Lambda nodes

Lists and symbols were added as a nod to the name SymPL (Symbolic Programming Language), but they show having a language specific runtime object representation that might need to be handled specially with runtime helper functions or binding rules. They also provided a nice excuse for writing a library of functions that actually do something, as well as showing importing Sympl libraries.

21.4 Control Flow

Control flow in Sympl consists of function call, lexical exits, conditionals, loops, and try/catch.

21.4.1 Function Call

A function call has the following form (parentheses are literal, curlies group, asterisks are regular expression notation, and square brackets indicate optional syntax):

```
(expr [{.id | . invokemember}* .id] expr*)
Invokemember :: (id expr*)
```

A function call evaluates the first expression to get a value:

```
(foo 2 "three")
(bar)
((lambda (x) (print x)) 5)
```

The first expression may be a dotted expression. If an identifier follows the dot, it must be a member of the previously obtained value, evaluating left to right. If a period is followed by invoke member syntax, the identifier in the invoke member syntax must name a member of the previously obtained value, and the member must be callable.

These two expressions are equivalent, but the first is preferred for style:

```
(obj.foo.bar x y)
obj.foo.(bar x y)
```

The second should only be used in this sort of situation:

```
obj.(foo y).bar ;;bar is property
(obj.(foo y).bar ...) ;;bar is method or callable member
((obj.foo y) . bar ...) ;; also works but odd nested left parens
```

The first form, (obj.foo.bar.baz x y), has the following semantics when baz is a method (where tmp holds the implicit 'this'):

```
(let* ((tmp obj.foo.bar))
  (tmp.baz x y))
```

The form has these semantics if baz is property with a callable value:

```
(let* ((tmp obj.foo.bar.baz))
  (tmp x y)) ;; no implicit 'this'
```

Sympl prefers implicit this invocation over a callable value in that it first tries to invoke the member with an implicit this, and then tries to call with just the arguments passed.

21.4.2 Conditionals

If expressions have the following form (parentheses are literal and square brackets indicate optional syntax):

```
(if expr expr [expr])
```

The first expr is the test condition. If it is neither nil nor false, then the second (or consequent) expr executes to produce the value of the **If**. If the condition is false or nil, and there is no third expr, then the **If** returns false; otherwise, it executes the third (or alternative) expr to produce the value of the **If**.

21.4.3 Loops

There is one **loop** expression which has the form (plus is regular expression notation):

```
(loop expr+)
```

Loops may contain **break** expressions, which have the following forms (square brackets indicate optional syntax):

```
(break [expr])
```

Break expressions do not return. They transfer control to the end of the loop. Break exits the loop and produces a value for the loop if it has an argument. Otherwise, the loop returns nil.

Sympl may consider adding these too (with break and continue as well):

```
(for ([id init-expr [step-expr]]) ;;while is (for () (test) ...)
     (test-expr [result-expr])
  expr*)

(foreach (id seq-expr [result-expr]) expr*)
```

21.4.4 Try/Catch/Finally and Throw

Still need to add Try/Catch/Finally/Throw. These are pretty easy, direct translations like 'if' and 'loop' since Expression Trees v2 directly supports these expressions.

These have semantics as supported by DLR Expression Trees. A **try** has the following form (parentheses are literal, asterisks are regular expression notation, and square brackets indicate optional syntax):

```
(try <expr>
  [(catch (<var> <type>) <body>)] *
  [(finally <body>)] )
```

21.5 Built-in Operations

These are the built-in operations in Sympl, all coming in the form of keyword forms as the examples for each shows:

- Function definition: **defun** keyword form. Sympl uses the **defun** keyword form to define functions. It takes a name as the first argument and a list of parameter names as the second. These are non-evaluated contexts in the Sympl code. The rest of a **defun** form is a series of expressions. The last expression to execute in a function is the return value of the function. Sympl does not currently support a **return** keyword form, but you'll see the implementation is plumbed for its support.

```
(defun name (param1 param2) (do-stuff param1) param2)
```

- Assignment: **set** keyword form

```
(set x 5)
(set (elt arr 0) "bill")  ; works for aggregate types, indexers, and Sympl lists
(set o.bar 3)
```

- arithmetic: **+**, **-**, *****, **/** keyword forms, where each requires two arguments

```
(* (+ x 5) (- y z))
```

- Boolean: **and**, **or**, **not** keyword forms. For each operand, any value that is not **nil** or **false**, it is true. **And** is conditional, so it is equivalent to (if e1 e2). **Or** is conditional, so it is equivalent to (let* ((tmp1 e1)) (if tmp1 tmp1 (let* ((tmp2 e2)) (if tmp2 tmp2)))).
- Comparisons: **=**, **!=**, **<**, **>**, **eq** keyword forms. All but **eq** have the semantics of Expression Trees v2 nodes. **Eq** returns true if two objects are reference equal, and for integers, returns true if they are numerically the same value.
- Indexing: **elt** keyword form

```
(elt "bill" 2)
(elt '(a b c) 1)
(elt dot-net-dictionary "key")
```

- Object instantiation: **new** keyword form

```
(new system.text.stringbuilder "hello world!")
(set types (system.array.createinstance system.type 1))
(set (elt types 0) system.int32)
(new (system.collections.generic.list`1.MakeGenericType types))
```

- Object member access: uses infix dot/period syntax

```
o.foo
o.foo.bar
(set o.blah 5)
```

21.6 Globals, Scopes, and Import

The hosting object, Sympl, has a Globals dictionary. It holds globals the host makes available to an executing script. It also holds names for namespace and types added from assemblies when instantiating the Sympl hosting object. For example, if mscorlib.dll is passed to Sympl, then Sympl.Globals binds "system" to an ExpandoObject, which in turn binds "io" to an ExpandoObject, which in turn binds "textreader" to a model of the TextReader type.

21.6.1 File Scopes and Import

There is an implicit scope per file. Free references in expressions resolve to the file's implicit scope. Bindings are created in a file's scope by setting identifiers that do not resolve to a lexical scope. There also is an **import** expression that binds file scope variables to values brought into the file scope from the host's Sympl.Globals table or from loading other files.

Import has the following form (parentheses are literal, curlies group, asterisks are regular expression notation, and square brackets indicate optional syntax):

```
(import id[.id]* [{id | (id [id]*)} [{id | (id [id]*)}]] )
```

The first ID must be found in the Sympl.Globals table available to the executing code, or the ID must indicate a filename (with implicit extension .sympl) in the executing file's directory. If the first argument to **import** is a sequence of dotted IDs, then they must evaluate to an object via Sympl.Globals. The effect is that **import** creates a new file scope variable with the same name as the last dotted identifier and the value it had in the dotted expression.

If the second argument is supplied, it is a member of the result of the first expression, and the effect is that this member is imported and assigned to a file scope variable with the same name. If the second argument is a list of IDs, then each is a member of the first argument resulting in a new variable created for each one with the corresponding name.

If the third argument is supplied, it must match the count of IDs in the second argument. The third set of IDs specifies the names of the variables to create in the file's scope, setting each to the values of the corresponding members from the second list.

If the first ID is not found in Sympl.Globals, and it names a file in the executing file's directory, then that file is executed in its own file scope. Then in the file's scope that contains the **import** expression, **import** creates a variable with the name ID and the imported file's scope as the value. A file's scope is a dynamic object you can fetch members from. The second and third arguments have the same effect as specified above, but the member values come from the file's scope rather than an object fetched from Sympl.Globals.

The IDs may be keywords without any backslash quoting. Note, that if such an identifier is added to Globals, referencing locations in code will need to be quoted with the backslash.

Import returns nil.

21.6.2 Lexical Scoping

Sympl has strongly lexically scoped identifiers for referencing variables. Some variables have indefinite extent due to closures. Variables are introduced via function parameters or **let*** bindings. Function parameters can be referenced anywhere in a function where they are not shadowed by a **let*** binding. **Let*** variables can be referenced within the body of the **let*** expression. For example,

```
(defun foo (x)
   (system.console.writeline x)
   (let ((x 5))
     (system.console.writeline x))
     (set x 10))
     (system.console.writeline x))
   (system.console.writeline x))
(foo 3)
```

prints 3, then 5, then 10, then 3 .

Each time execution enters a **let*** scope, there are distinct bindings to new variables semantically. For example, if a **let*** were inside a **loop**, and you saved closures in each iteration of the loop, they would close over distinct variables.

21.6.3 Closures

Sympl has closure support. If you have a **lambda** expression in a function or within a **let***, and you reference a parameter or let binding from within the lambda, Sympl closes over that binding. If a **let*** were inside a **loop**, and you saved closures in each iteration of the loop, they would close over distinct variables. The great thing about Expression Trees is this is free to the language implementer!

21.7 Why No Classes

Sympl does not demonstrate classes. Sympl could have showed using an ExpandoObject to describe class members, and used a derivation of DynamicObject to represent instances since DynamicObject can support invoke member. Sympl could have stayed simple by requiring static class members to be access via classname.staticmember. It also could require an explicit 'self' or 'this' parameter on any instance methods (and using self.instancemember). This may have been cute, but it wouldn't have demonstrated anything real on the path to implementing good .NET interop.

Real languages need to use .NET reflection to emit real classes into a dynamic assembly. You need to do this to derive from .NET types and to pass your class instances into .NET static libraries. Performance is also better when you can burn some members into class fields or properties for faster access.

21.8 Keywords

The following are keywords in Sympl:

- Import
- Defun, Lambda
- Return (not currently used)
- Let*, Block
- Set
- New
- +, -, *, /
- =, !=, <, >
- Or, And, Not
- If
- Loop, Break, Continue (continue not currently used)
- Try, Catch, Finally, Throw (not currently used)
- Elt
- List, Cons, First, Rest
- Nil, True, False

21.9 Example Code (mostly from test.sympl)

```
(import system.windows.forms)

(defun nconc (lst1 lst2)
   (if (eq lst2 nil)
       lst1
       (if (eq lst1 nil)
           lst2
           (block (if (eq lst1.Rest nil)
                      (set lst1.Rest lst2)
                      (nconc lst1.Rest lst2))
                  lst1))))

(defun reverse (l)
   (let* ((reverse-aux nil))
      (set reverse-aux
          (lambda (remainder result)
             (if remainder
                (reverse-aux remainder.Rest
                             (cons remainder.First result))
                result)))
      (reverse-aux l nil)))

(import system)
(system.console.WriteLine "hey")

(defun print (x)
   (if (eq x nil)
       (system.console.writeline "nil")
       (system.console.writeline x))
   x)

(print nil)
(print 3)
(print (print "cool"))
(set blah 5)
(print blah)

(defun foo2 (x)
   (print x)
   (let* ((x "let x")
          (y 7))
      ;; shadow binding local names
      (print x)
      (print y)
      (set x 5)
      (print x))
   (print x)
   (print blah)
   ;; shadow binding global names
   (let* ((blah "let blah"))
      (print blah)
      (set blah "bill")
      (print blah))
   (print blah)
   (set blah 17))
```

```
((lambda (z) (princ "non ID expr fun: ") (print z)) "yes")
(set closure (let* ((x 5))
               (lambda (z) (princ z) (print x))))
(closure "closure: ")

(print nil)
(print true)
(print false)

(print (list x alist (list blah "bill" (list 'dev "martin") 10) 'todd))

(if (eq '(one).Rest nil) ;_getRest nil)
    (print "tail was nil"))
(if (eq '(one two) nil)
    (print "whatever")
    (print "(one two) is not nil"))

;; Sympl library of list functions.
(import lists)

(set steve (cons 'steve 'grunt))
(set db (list (cons 'bill 'pm) (cons 'martin 'dev) (cons 'todd 'test)
              steve))

(print (lists.assoc 'todd db))

(print (lists.member steve db))

(let* ((x '(2 6 8 9 4 10)))
   (print
      (loop
         (if (eq x.First 9)
             (break x.Rest)
             (set x x.Rest)))))

(set x (new System.Text.StringBuilder "hello"))
(x.Append " world!")
(print (x.ToString))
(print (x.ToString 0 5))
(set y (new (x.GetType) (x.ToString)))
(print (y.ToString))
(print y.Length)
```

22 Runtime and Hosting

Sympl provides very basic hosting, essentially execute file and execute string. You can instantiate a Sympl runtime with a list of assemblies whose namespaces and types will be available to Sympl code. You can execute files in a host-provided scope or module, or let Sympl create a new scope for each execution. You can execute strings with Sympl expressions in a host-provided scope, or one previously obtained from executing a file.

There is a host globals table. Sympl programs can import names from the globals table, making those names become file module globals. When you execute a file, it name (without directories

or extension) becomes an entry in the host globals table. Other Sympl programs can then import that name to access libraries of Sympl code.

22.1 Class Summary

This class represents a Sympl runtime. It is not intentionally thread-safe or necessarily hardened in any way.

```
public ExpandoObject ExecuteFile(string filename)
public ExpandoObject ExecuteFile(string filename,
                                 string globalVar)
```

ExecuteFile reads the file and executes its contents in a fresh scope or module. Then it adds a variable in the Globals table based on the file's name (no directory or extension), so that importing can refer to the name to access the file's module. When globalVar is supplied, this is the name entered into the Globals table instead of the file's base name. These functions also add a variable to the module called "__file__" with the file's full pathname, which importing uses to load file names relative to the current file executing.

```
public void ExecuteFileInScope(string filename,
                               ExpandoObject moduleEO)
```

ExecuteFile is just like ExecuteFile except that it uses the scope provided, and it does not add a name to Globals.

```
public object ExecuteExpr(string expr_str,
                          ExpandoObject moduleEO)
```

ExecuteExpr reads the string for one expression, then executes it in the provided scope or module. It returns the value of the expression.

```
public static ExpandoObject CreateScope()
```

CreateScope returns a new module suitable for executing files and expression in.

23 Appendixes

These sections show using DLR APIs that are available on codeplex.com only for now. Some will move into CLR versions beyond 4.0. These APIs let you provide more features for your language more easily, such as generators (functions with yield expressions) or richer .NET binding logic by using the DefaultBinder. These APIs sometimes improve performance over the base Sympl implementation, such as using the namespace/type trackers that IronPython uses.

The source is in the ...\languages\sympl\csharp-cponly directory, separated from the version that only depends on CLR 4.0 APIs, for a cleaner code sample. This will be more useful when this version supports more codeplex-only APIs. While the changes for each appendix may not need to be big, it should be cleaner to have sources that are not riddled with "#if cponly". Two sets of sources support using 'diff' tools to see changes more readily.

23.1 Supporting the DLR Hosting APIs

This section shows a minimal LanguageContext implementation so that applications can host Sympl using the common hosting model provided by the DLR. This means Sympl could seamlessly be added to any host using these APIs for multi-language scripting support. The Main function in program.cs shows executing Python and Ruby code in the same scope or module that Sympl uses, cross-language interoperability, and accessing host supplied globals.

The changes comprise one small new file and a few tweaks to a couple of other files. The changes show using the hosting model's Globals table where hosts inject global bindings for script code to access the hosts' object models. The changes show how to represent your language as a DLR ScriptEngine and how to run code in the DLR hosting model. They also show how Sympl can expose extension services to the host.

A LanguageContext can participate in hosting by providing more functionality than what is shown here. For example, your language can support a tokenizing/colorizing service, error formatting service, ObjectOperations, execute program semantics, configuration settings such as search paths, and so on (see the dlr-spec-hosting.doc on codeplex.com/dlr). The Sympl sample shows how to get code to run in a host that supports common DLR Hosting APIs and how to provide extension services.

23.1.1 Main and Example Host Consumer

Let's start top-down and look at how program.cs changed in Main. This is a portion of the code:

```
static void Main(string[] args)
{
    var setup = new ScriptRuntimeSetup();
    string qualifiedname =
        typeof(SymplSample.Hosting.SymplLangContext)
            .AssemblyQualifiedName;
    setup.LanguageSetups.Add(new LanguageSetup(
        qualifiedname, "Sympl", new[] { "sympl" },
        new[] { ".sympl" }));
    setup.LanguageSetups.Add(
        IronPython.Hosting.Python.CreateLanguageSetup(null));
    setup.LanguageSetups.Add(IronRuby.Ruby.CreateRubySetup());
    var dlrRuntime = new ScriptRuntime(setup);
    var engine = dlrRuntime.GetEngine("sympl");

    string filename = @"..\..\Languages\sympl\examples\test.sympl";
    var feo = engine.ExecuteFile(filename);
    Console.WriteLine("ExecuteExpr ... ");
    engine.Execute("(print 5)", feo);

    // Get Python and Ruby engines
    var pyeng = dlrRuntime.GetEngine("Python");
    var rbeng = dlrRuntime.GetEngine("Ruby");
    // Run some Python and Ruby code in our shared Sympl module.
    pyeng.Execute("def pyfoo(): return 1", feo);
    rbeng.Execute("def rbbar; 2; end", feo);
    // Call those objects from Sympl.
    Console.WriteLine("pyfoo returns " +
                      (engine.Execute("(pyfoo)", feo)).ToString());
    Console.WriteLine("rbbar returns " +
```

```
                            (engine.Execute("(rbbar)", feo)).ToString());

        // Consume host supplied globals via DLR Hosting.
        dlrRuntime.Globals.SetVariable("DlrGlobal", new int[] { 3, 7
});
        engine.Execute("(import dlrglobal)", feo);
        engine.Execute("(print (elt dlrglobal 1))", feo);

        // Drop into the REPL ...
        ... deleted code ...
        var s = engine.GetService<Sympl>();
        while (true) {
            ... deleted code ...
            try {
                object res = engine.Execute(exprstr, feo);
                exprstr = "";
                prompt = ">>> ";
                if (res == s.MakeSymbol("exit")) return;
                Console.WriteLine(res);
            ... deleted code ...
```

This code creates a ScriptRuntimeSetup and fills it in with specific LanguageSetups for Sympl, IronPython, and Ironruby. An application may offer script engines to its users language-independently and work with newly added engines without changing its code. Application can do this by using an app.config file. See the dlr-spec-hosting.doc on codeplex.com/dlr. This example uses the convenience LanguageSetup factory functions from IronPython and IronRuby to get default setups for just those languages. If you use an app.config file, ScriptRuntime has a factory method that reads the app.config file and returns a configured ScriptRuntime.

Next the Main function runs the same Sympl test file as the other version of Sympl, followed by running IronPython and IronRuby code in the same file module or scope. Then Main executes Sympl expressions that use the IronPython and IronRuby objects. The example could have also loaded IronPython and IronRuby files and added the ScriptScopes they returned to the feo module's globals. Calling across modules and "dotting" into members to get at IronPython and IronRuby objects would also work.

Lastly, Main installs host globals using the DLR's common hosting APIs. It binds the name dlrglobal to a .NET array. Then it executes Sympl code to import the name into the file's module, and then executes code to use the values imported. This is the IronPython model of accessing DLR host globals. IronRuby simply does global lookups by chaining up to the DLR's globals table automatically.

The last bit worth noting is that the Sympl support for DLR Hosting APIs supports fetching a service from the Sympl engine. The Sympl engine lets the host fetch the inner Sympl hosting object, which happens to be the hosting object first defined in the other version of Sympl. The Main function uses this to make Sympl symbol objects so that the REPL can continue testing for 'exit to stop the session.

23.1.2 Runtime.cs Changes

The changes to runtime.cs are few and really simple. The first is that Sympl no longer uses ExpandoObjects for its file modules. It must use the language-implementation Scope type. Sympl must wrap this to make it into an IDynamicMetaObjectProvider because its file modules

are just dynamic objects and can be passed around as such. The only change in runtime.cs for this support is renaming ExpandoObject to IDynamicMetaObjectProvider.

The second change is in the SymplImport runtime helper function. When importing a single name (that is, not a dotted name sequence), SymplImport now looks in Sympl.DlrGlobals if the name is not in Sympl.Globals. The branch where there are multiple names to look up, successively fetching a member of the previous resulting object, should change too. It should handle this same check for DlrGlobals for the first name. It also should change in all versions of Sympl to not assume the objects are dynamic objects. It could explicitly create a CallSite with a SymplGetMemberBinder to be able to handle any kind of object (dynamic or static .NET).

23.1.3 Sympl.cs Changes

There are three basic changes to sympl.cs. It needs to change ExpandoObject to IDynamicMetaObjectProvider for the file module becoming DLR Scopes, as mentioned in the previous section. The Sympl class needs to provide a couple more entry points for how the DLR Hosting APIs call on the new Sympl LanguageContext to parse and run code. Lastly, the Sympl class now keeps a reference to the DLR's ScriptRuntime.Globals Scope so that Sympl programs can import host supplied globals for the host's object model.

The new entry points needed support the LanguageContext's CompileSourceCode method (see the section on dlrhosting.cs). The Sympl class now has ParseFileToLamba and ParseExprToLambda. Creating these was a straightforward extraction refactoring of ExecuteFile and Execute Expr, so there's nothing new to explain about how that code works.

The last change was to make the Sympl class constructor take a DLR Scope, which is the ScriptRuntime's Globals table. See the section on changes to runtime.cs for more information, but the change support host globals was a new else-if branch and a couple of lines of code.

23.1.4 Why Not Show Using ScriptRuntime.Globals Namespace Reflection

You might notice that the Sympl hosting class still calls AddAssemblyNamespacesAndTypes to build its own reflection modeling. It could use the DLR's ScriptRuntime.Globals which has names bound similarly for namespaces and types. There are a couple of issues with Sympl's using the DLR's reflection.

The first is that the DLR Globals scope effectively includes two dictionaries, a regular DLR Scope and NamespaceTracker (or a TypeTracker). The DLR's Globals scope looks names up both dictionaries. The NamespaceTracker object has a bug in that it fails to look up SymbolIDs case-INsensitively, which the Scope object does correctly. This means the Sympl expression "(import system)" fails to find the name "system". One way to work around this would have been to fetch all the keys from the Globals scope and compare each one for a case-INsenstive match. If there was only one, then return successfully. Furthermore, Sympl's binders would need updating to explicitly notice when a NamespaceTracker or TypeTracker flowed into the call site (say, for "system.console" binding) and have special binding logic for the trackers.

The second reason Sympl continues to build its own reflection modeling is that the DLR may fix the NamespaceTracker and related types to flow as dynamic objects with correct lookup behavior. The DLR might remove them entirely and no longer push them into the ScriptRuntime's Globals. Each language would then have to have its own reflection modeling

like Sympl. The DLR would either have helpers, or people could copy code from IronPython to get the benefits of the tracker objects.

23.1.5 The New DlrHosting.cs File

This file is where the more interesting new code is. Primarily this file defines a LanguageContext, which is the representation of a language or execution engine to the DLR Hosting APIs. The LanguageContext in Sympl does two things. It supports compiling code, which can be run in a new DLR scope or in a provided scope. It also provides a service that returns the inner Sympl hosting object. The LanguageContext could do more, such as supporting a tokenizing/colorizing service, error formatting service, ObjectOperations, execute program semantics, etc (see the dlr-spec-hosting.doc on codeplex.com/dlr). The Sympl sample mostly just shows how to get code to run in a host that supports common DLR Hosting APIs.

Here is the code for the SymplLangContext in dlrhosting.cs:

```
    public sealed class SymplLangContext : LanguageContext {
        private readonly Sympl _sympl;

        public SymplLangContext(ScriptDomainManager manager,
                                IDictionary<string, object> options)
            : base(manager) {
            _sympl = new Sympl(manager.GetLoadedAssemblyList(),
                               manager.Globals);
        }

        protected override ScriptCode CompileSourceCode(
                SourceUnit sourceUnit, CompilerOptions options,
                ErrorSink errorSink) {
            using (var reader = sourceUnit.GetReader()) {
                try {
                    switch (sourceUnit.Kind) {
                        case SourceCodeKind.SingleStatement:
                        case SourceCodeKind.Expression:
                        case SourceCodeKind.AutoDetect:
                        case SourceCodeKind.InteractiveCode:
                            return new SymplScriptCode(
                                _sympl,
                                _sympl.ParseExprToLambda(reader),
                                sourceUnit);
                        case SourceCodeKind.Statements:
                        case SourceCodeKind.File:
                            return new SymplScriptCode(
                                _sympl,

sympl.ParseFileToLambda(sourceUnit.Path,
                                                        reader),
                                sourceUnit);
                        default:
                            throw Assert.Unreachable;
                    }
                }
                catch (Exception e) {
                    errorSink.Add(sourceUnit, e.Message,
                                  SourceSpan.None, 0,
```

```
                                    Severity.FatalError);
                return null;

        public override TService GetService<TService>(
                                            params object[] args) {
            if (typeof(TService) == typeof(Sympl)) {
                return (TService)(object)_sympl;
            }
            return base.GetService<TService>(args);
        }
```

The key method is CompileSourceCode, which returns a SymplScriptCode object that is discussed below. There are several kinds of code the DLR might ask the language to compile. These kinds allow languages to set initial parser state (such as SourceCodeKind.Expression vs. SingleStatement) or to apply special semantics for magic interactive loop syntax or variables (SourceCodeKind.InteractiveCode). Sympl buckets all of these into either the Expression or File kind because Sympl doesn't need finer-grain distinctions. These two branches just call the new entry points in the Sympl class.

Regarding the catch block, a more serious language implementation would have a specific type of exception for parse errors. It would also pass the errorSink down into the parser and add messages while doing tighter error recovery and continuing to parse when possible. For the Sympl example, it just catches the first error, passes it to the errorSink, and punts.

The SymplScriptCode returned from CompileSourceCode above represents code to the DLR Hosting APIs. Sympl defines its ScriptCode as follows:

```
    public sealed class SymplScriptCode : ScriptCode {
        private readonly Expression<Func<Sympl,
                                         IDynamicMetaObjectProvider,
                                         object>>
                         _lambda;
        private readonly Sympl _sympl;
        private Func<Sympl, IDynamicMetaObjectProvider, object>
                _compiledLambda;

        public SymplScriptCode(
             Sympl sympl,
             Expression<Func<Sympl, IDynamicMetaObjectProvider,
object>>
                 lambda,
             SourceUnit sourceUnit)
             : base(sourceUnit) {
            _lambda = lambda;
            _sympl = sympl;
        }

        public override object Run() {
            return Run(new Scope());
        }

        public override object Run(Scope scope) {
            if (_compiledLambda == null) {
```

```
            _compiledLambda = _lambda.Compile();
        }
        var module = new SymplModuleDlrScope(scope);
        if (this.SourceUnit.Kind == SourceCodeKind.File) {
            DynamicObjectHelpers.SetMember(
                module, "__file__",
                Path.GetFullPath(this.SourceUnit.Path));
        }
        return _compiledLambda(_sympl, module);
```

There are three interesting notes on this code. The first is that we parse to a type of lambda just like Sympl does in the other version. In this version the type of the LambdaExpression is the same for both executing files and executing expressions because the DLR hosting expects a return value in both cases. In the other version, Sympl's file LambdaExpressions returned void. Now Sympl just returns null from each file lambda. The next point is that Sympl needs to wrap the Scope passed to the Run method. Sympl does this so that the DLR scope passed to Run can be passed around in Sympl code as a dynamic object. The last point is that before invoking the compiled lambda, the Run method stores the file module variable "__file__" so that Sympl's 'import' expressions work correctly. Sympl.ExecuteFile still stores "__file__" also.

Lastly, DlrHosting.cs defines a wrapper class for DLR language-implementation Scopes so that Sympl can pass them around as dynamic objects:

```
    public sealed class SymplModuleDlrScope : DynamicObject {
        private readonly Scope _scope;

        public SymplModuleDlrScope(Scope scope) {
            _scope = scope;
        }

        public override bool TryGetMember(GetMemberBinder binder,
                                          out object result) {
            return_scope.TryGetName(

SymbolTable.StringToCaseInsensitiveId(binder.Name),
                        out result);
        }

        public override bool TrySetMember(SetMemberBinder binder,
                                          object value) {
            _scope.SetName(SymbolTable.StringToId(binder.Name), value);
            return true;
```

To get an easy implementation of IDynamicMetaObjectProvider, Sympl uses the DynamicObject type. It is a convenience type for library authors that enables you to avoid implementing a binder and generating expression trees. DynamicObject implements the IDynamicMetaObjectProvider interface by always returning a rule that calls Try... methods on the DynamicObject.

Sympl only needs to support GetMember and SetMember operations since it doesn't have any special InvokeMember semantics or other special behaviors for its file scopes. As described in the main sections of this document, Sympl stores IDs as they are spelled and stored in the binder's metadata. However, Sympl looks IDs up case-INsensitively. Since the underlying DLR

Scope objects use SymbolID objects to represent names, Symp has to map the binder's Name property to an appropriate SymbolID object.

23.2 Using the Codeplex.com DefaultBinder for rich .NET interop

Using the default binder gives you a complete .NET interoperability story. It has many customization hooks to tailor how it binds to .NET. IronPython and IronRuby use it as well as other language implementations. We'll demonstrate using it to get richer .NET binding and to convert Sympl nil to .NET False, TypeModel to .NET RuntimeType, and Cons to some .NET sequence types.

TBD

23.3 Using Codeplex.com Namespace/Type Trackers instead of ExpandoObjects

For a faster and richer implementation of how Sympl provides access to .NET namespaces and types, we use the DLR's reflection trackers.

TBD

23.4 Using Codeplex.com GeneratorFunctionExpression

It is very easy to add generators or what C# calls iterators (functions with 'yield' expressions) to your language. The support on Codeplex for this is very solid and used by IronPython and other languages. The only reason it didn't ship in CLR 4.0 is that the implementations has a couple of python-specific features that we did not have time to parameterize into a general model for usage.

TBD