1. An Introduction to Wyvern

Here is a “Hello, World!” program in Wyvern:

```
require stdout

stdout.print("Hello, World!")
```

This program already illustrates several basic aspects of Wyvern. First, Wyvern is object-oriented: `stdout` is an object, and we are invoking the `print` method on it. For expressions, much of Wyvern’s syntax is similar to Java’s.

A second notable feature of Wyvern illustrateed by this program is that the `stdout` object is not ambiently available to programs, nor can it be imported in the sense of a Java `import` statement. Because `stdout` does I/O, it is a *resource*. A primary goal of Wyvern’s module system is helping developers to reason about the use of resources. Thus even a simple script such as Hello World must declare the resources it requires in order to execute. This allows engineers to determine at a glance what kind of I/O a program might do, and provides a basis for making a decision about whether to run this program in a particular situation. In this case, even without looking at the actual code, we know that this program may write to the standard output stream, but will not access the file system or access the network.

2. Objects and Modules in Wyvern

Let us further illustrate objects and modules in Wyvern through a simple logging module:

```
resource module logging

import wyvern.collections.list
require files

resource type Log
    def log(x:String):Unit

def makeLog(path:String):Log
    val logFile = files.openForAppend(path)
    val messageList = list.make()
    new
        def log(x:String):Unit
            messageList.append(x)
            logFile.print(x)
```

A module starts with the module keyword and a name that denotes the module. The logging module needs access to the file system in order to write to log files, so it declares that it requires the `files` resource. Because this module requires a resource, it becomes a resource itself, in that other modules can use it to access the file system via log-related operations. Thus we must declare it to be a `resource` module.

In our example, the log stores logged messages in an internal list, perhaps so it can be accessed programmatically (through an interface not shown here). Thus we need to `import` a `list` abstraction. We could `require` the `list` module, but this is unnecessary: the `list` module does not provide any access to I/O, and it does not have any internal state (`List` objects created by the `list` module are stateful in our example, but that is not relevant for module imports). It is thus not a resource module, and can therefore be used by any module in the program with a simple `import` statement similar to Java's.

The import statement uses a fully-qualified name similar to package names in Java (in the future, this fully-qualified name will be given at the head of the module definition, rather than just a simple name, but this is not fully specified or implemented yet). A protocol similar to Java's is used to find a file defining that module: specifically, we check the library directory and the current directory for subdirectories named by the qualifier and search recursively down until we find the file list.wyv. In this example, the library directory is tools/src/wyvern/lib and so the import statement binds to the file wyvern/collections/list.wyv within this directory tree.

Next, we declare a type for `Log` objects. The interface is pretty simple: a single method `log` taking a `String` argument is defined using the `def` keyword. The return type of `Unit`–equivalent to `void` in Java–is specified at the end of the method declaration. Note that the members of the `Log` type are described in indented lines, rather than (for example) within curly braces. We mark the type as being a resource type, again because a `Log` object encapsulates the ability to write to a particular log file.

Finally, we define a `makeLog` method in order to create a `Log` object. The implementation of the function is given in indented lines. The first line declares a local variable, `logFile`, which is the file we will be logging to. Likewise, we declare a variable for the list where we will store the log messages. The last line that is indented at the function implementation level is returned by the function–in this case a `new` expression that actually creates the `Log` object. The type of object to be created is inferred based on the return type of the method. We implement the object's methods in indented lines following the `new` expression.

Now we can write a program that uses the logging module:

```
require files
import examples.logging
import examples.logApplication

val logger = logging(files)
val client = logApplication(logger)

client.run()
```

Since the `logging` module defined earlier requires a `files` in order to operate, and since `files` represents a resource that we can only get from the operating system, our program must also `require files`. We can then import the code for the `logging` module (specifying the fully qualified location `examples.logging`) and then instantiate the code by passing the `files` module to fulfill its requirement. We bind the resulting instantiated module to `logger`, which we can use in the rest of the code as if it were an object.

The main program that uses the `logger` is defined in a module `examples.logApplication` (not shown). It requires the `logger` and so we must instantiate `logApplication` with the `logger` as an argument, and then bind the result to the name `client`. We can then execute the application code by invoking `client.run()`.

3. Reasoning about Security in Wyvern

The program code above is notable for the security analysis it facilitates based on the principle of least privilege. The overall program requires the `files` resource, but the only use of this resource is to instantiate the `logging` module. Wyvern is a capability-safe language, so the `logApplication` module cannot ambiently import the `files` without getting permission from the main module. Furthermore, Wyvern does not provide unconstrained global state which could, in a language like Java, be used to share the `files` between modules in a hard-to-analyze way. We can conclude that `logApplication` does not directly use the `files` simply by observing that it is not passed to `logApplication` upon instantiation, and further observing that it is not exposed in the interface of the `logging` module.

Of course, `logApplication` can write to a log file by using the `logging` module, and this is a use of the file system. But by our knowledge of the `logging` module, or by a brief inspection of its code, we can see that it provides only a restricted mode of use of the file system: reading files is not permitted, and writing files is only possible in append mode. A refinement of the `logging` module could impose additional restrictions, such as restricting log writes to particular directories.

Thus the combination of the capability-safe nature of the Wyvern language, the `require` construct, and the careful structuring of the source program, we get a strong guarantee about the behavior of `logApplication`: the only I/O effect it has is logging, which is a refinement of file system access, which is a refinement of general I/O. Furthermore, we paid a relatively low cost to facilitate this reasoning. In the `logging` module (and in `logApplication`, if we were to examine it) the code is no more verbose than you would expect of code in another typed language such as Java: basically we just have to use the `require` keyword in place of `import` and we have to mark types and modules with the `resource` keyword as appropriate. The overhead in the main module is a substantially higher, as we have to instantiate modules explicitly and thread resources through them.

Note: The logging example above does not run yet. In particular, we have not yet implemented the `files` module.

4. Module Bodies

Module bodies are made up of four kinds of core declarations: `val`, `var`, `def`, and `type`, as well as expressions. The `val` and `var` declarations and the expressions in a module body are evaluated in sequence, and the variables defined earlier in the sequence are in scope in later declarations and expressions. In contrast, `def` and `type` declarations do not evaluate, and therefore these declaration forms can be safely used to define mutually recursive functions and types. Each sequence of declarations that consists exclusively of `def` and `type` is therefore treated as a mutually recursive block, so that the definition or type defined in each of the declarations is in scope in all the other declarations.

To understand why we allow recursive `def` and `type` declarations but do not allow this for `val` and `var` declarations, consider the following example:

```
def foo() = baz()
val bar = foo()
def baz() = bar
```

When we try to initialize the `bar` value, we call `foo()`, which in turn invokes `baz()`. However, `baz()` reads the `bar` variable, which is what we are defining, so there is no well-defined result. Languages such as Java handle this by initializing `bar` to `null` at first and then writing a permanent value to it after the initializer executes. However, in order to avoid null pointer errors, Wyvern does not allow `null` as a value. Languages such as Haskell would use a special "black hole" value and signal a run-time error in cases such as the above. We avoid this semantics as it adds complexity and means the program can fail at run time. Of course, infinite loops can still exist in Wyvern, but they come from recursive functions, never recursively defined values.

5. Anonymous Functions

Anonymous functions can be defined in Wyvern using the syntax:

```
(x:Int) => x + 1
```

This is a shorthand for creating an object with an `apply` method that has the same arguments and body:

```
new
    def apply(x:Int):Int = x + 1
```

which is an instance of the following type:

```
type IntToIntFn
    def apply(x:Int):Int
```

Anonymous functions can also have multiple parameters:

```
(x:Int,y:Int) => x + y
```

or no parameters:

```
() => 7
```

We expect to define a convenient shorthand for function types in the near future.