1. An Introduction to Wyvern

Here is a "Hello, World!" program in Wyvern:

```
import stdout

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

This program already illustrates a couple of 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.

Second, system resources such as the standard output object, `stdout`, are not ambiently available to programs, but must be explicitly imported. 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.

In planned, but not yet implemented, functionality, using system resources such as `stdout` will involve the use of the `require` construct defined below.

2. Anonymous Functions

Anonymous functions can be defined in Wyvern using the syntax:

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

We can bind the expresison above to a variable and invoke it:

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

and the result will be 2.

Anonymous functions can also have multiple parameters:

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

or no parameters:

```
() => 7
```

3. Functions in Wyvern

Consider the definition of the `factorial` function in Wyvern:

```
import stdout

def factorial(n:Int):Int
```

```
    (n < 2).ifTrue(
        () => 1,
        () => n * factorial(n-1)
    )

stdout.print("factorial(15) = ")
stdout.printInt(factorial(15))
```

A function is defined with the `def` keyword, and its argument and return types are given in Algol-like syntax. Functions defined with `def` are recursive, so we can call `factorial` in the body. The example illustrates how an integer comparison `n-2` is a boolean object, on which we can invoke the `ifTrue` method. This method takes two functions, one of which is evaluated in the true case and one of which is evaluated in the false case.

Note that `factorial(15)` would overflow in languages such as Java in which the default integer types is represented using only 32 bits. In Wyvern, `Int` means an arbitrary precision integer.

4. Objects and Object Types in Wyvern

We can define a sumable integer list type as follows:

```
type IntList
    def sum():Int
```

The `type` keyword declares a new object type, called `IntList` in this case. The public methods available in the type are listed below, but no method bodies may be given as we are defining a type, not an implementation.

We can implement a constant representing the empty list and a constructor for creating a larger list out of a smaller one as follows:

```
val empty:IntList = new
    def sum():Int = 0

def cons(elem:Int,rest:IntList):IntList = new
    def sum():int = elem + rest.sum()

cons(3,cons(4,empty)).sum() // evalutes to 7
```

The `new` expression creates an object with the methods given. In the example above, we just have one method, `sum()`, which evaluates to `0` in the case of the `empty` list and sums up the integers in the list otherwise.

5. Anonymous Functions as Objects

The anonymous function syntax described above is actually 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
```

Unsurprisingly, the type above can be abbreviated `Int -> Int`, as in many other languages with good support for functional programming.

6. Mutable State and Resource Types

Types with mutable state can be defined, but need to be marked as `resource` types:

```
resource type Cell
    def set(newValue:Int):Unit
    def get():Int

def makeCell(initVal:Int):Cell = new
    var value : Int = initVal
    def set(newValue:Int):Unit
        this.value = newValue
    def get():Int = value

val c = makeCell(5)
c.get() // evalutes to 5
c.set(3)
c.get() // evalutes to 3
```

Here `makeCell` uses a `new` statement to create an object with a `var` field `value`. `var` fields are assignable, so the `set` funtion is implemented to assign the `value` field of the receiver object `this` to the passed-in argument. Note that we must initialize a `var` field with an initial value. If we had not declared `Cell` to be a `resource` type, we would get an error because the `new` expression creates a stateful object that is a resource.

In the example above, `Unit` is used as the return type of functions that do not return any interesting value.

7. Modules

We can define the `Cell` abstraction above in a module:

```
module cell
```

```
resource type Cell
    def set(newValue:Int):Unit
    def get():Int

def make(initVal:Int):Cell = new
    var value : Int = initVal
    def set(newValue:Int):Unit
        this.value = newValue
    def get():Int = value
```

In Wyvern, analogously to Java, a module named `m` should be stored in a file `m.wyv` (we expect that the implementation will enforce this in the near future). The file system forms a hierarchical namespace with one name per directory that allows us to find modules by their qualified name. Thus, assuming we have the module above defined in a file `myPackage/mySubPackage/cell.wyv`, we can use it in a program as follows:

```
import myPackage.mySubPackage.cell

val myCell : cell.Cell = cell.make(3)
myCell.set(7)
myCell.get() // evalutes to 7
```

Here the import statement takes a fully qualified name and uses this to find the file defining module `cell`. The module is actually an object that gets bound to the name `cell`. We can invoke `make()` on `cell` just as if it were a method. Types such as `Cell` defined in the `cell` module can be referred to by their qualified names, i.e. `cell.Cell`. In fact, types can be defined as members of an object as well, and the same qualified syntax can be used to refer to them. So modules are not special semantically: they are just a convenient syntax for defining an object. Consider what is the type of `cell`? The answer could be written as follows:

```
type TCell
    resource type Cell
        def set(newValue:Int):Unit
        def get():Int

    def make(initVal:Int):this.Cell
```

8. Resource Modules

Just as objects with state must be given a resource type, stateful modules have resource type. A `resource` module is one that captures state in its implementation. The `cell` module is not a `resource` module; although it can be used to create stateful `Cell`s, the module itself does not capture state. Here is a module that does:

```
resource module cellModule

var value : Int = 0
def set(newValue:Int):Unit
    value = newValue
def get():Int = value
```

Wyvern does not allow implicitly shared global state, because this often causes problems in software development. So `cellModule` does not evaluate to an object, but rather a function that, when invoked, yields a fresh object with its own copy of the internal state defined by the module. We can use `cellModule` in a program as follows:

```
import myPackage.cellModule

val m1 = cellModule()
val m2 = cellModule()
m1.set(1)
m2.set(2)
m1.get() // evalutes to 1
m2.get() // evalutes to 2
```

In this example you can see that we have instantiated the `resource` module `cellModule` twice, and each instance of the module has its own internal state.

9. Module Parameters

If resource modules are functions, we expect to be able to pass parameters–and so we can. First let's define the type that `cellModule` returns. For convenience, we will put this type in a file `TCellModule.wyt` (here `.wyt` stands for Wyvern Type):

```
resource type TCellModule
    def set(newValue:Int):Unit
    def get():Int
```

Here is a client of the cellModule:

```
resource module cellClient

require myPackage.TCellModule as cell

def addOne():Unit
    cell.set(cell.get()+1)

def getValue():Int = cell.get()
```

In order to define a parameter of a module, we use a `require` statement in

place of an import. The `require` statement specifies the type that we expect the module parameter to be, and gives the parameter the name `cell`. If the `as cell` part is left out, the parameter will just be bound to the type name; this is somewhat inelegant, but does not cause a conflict because the namespace of types and values are separate in Wyvern.

Now we can use the two resource modules together in a program:

```
import myPackage.cellModule
import myPackage.cellClient

val client = cellClient(cellModule())
client.addOne()
client.getValue() // evalutes to 1
```

10. Dynamic Types

Wyvern is intended to be a mostly statically typed language. However, while getting type members to work, we implemented a `Dyn` type that partially implements dynamic types. Specifically, `Dyn` is a subtype of any type, and any type is a subtype of `Dyn`. (Note that subtyping is not transitive where `Dyn` is involved, as this would effectively collapse the type system to a single type.)

We recommend avoiding `Dyn` where possible, and gradually transitioning existing `Dyn` code to remove use of this construct. If we do keep this in the long term, we need to think about how it interacts with resource types.

11. Declaration Sequences and Mutual Recursion

Programs 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 program 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:

```
type IntCell
    def get():Int

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

```
bar.get()
```

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 if the black hole is ever used, as in the `bar.get` statement at the end. 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.

12. Some Examples

The Wyvern standard library files are in subdirectories of `tools/src/wyvern/lib`. For example, `stdout` is defined in `stdout.wyv`, within that directory. The definition of `stdout` uses some Java helper code defined in the Java class `wyvern.stdlib.support.Stdio`.

An example of a utility library that provides a small part of a regular expression package is in `wyvern/util/matching/regex.wyv`. The design approximately follows the corresponding Scala library.

An example of a data structure library is `wyvern/collections/list.wyv`. Also see `wyvern/option.wyv`.

Examples that are working test cases are in subdirectories of `tools/src/wyvern/tools/tests`. For example, `rosetta/hello.wyv`, `rosetta/fibonacci.wyv`, and `rosetta/factorial.wyv` illustrate writing to standard out, integer operations and tests, and recursion. These examples are in the `rosetta` subdirectory because they are inspired by the Rosetta Code project.

For a semi-realistic example that shows instantiation of resource modules, see `modules/safeSQLdriver.wyv`.

All of the above examples are tested by the Wyvern regression test suite that is run as part of `ant test` when building Wyvern.