1. Why Wyvern?

Wyvern is a new programming language exploring how to help software engineers build software better at scale. Our primary aim is to do research that discovers and validates new principles for designing engineering-focused programming languages. Part of the "validation" bit is actually writing interesting programs in Wyvern, and so we hope to also make it a great language to write code in—for us, and for you!

Wyvern's design incorporates a lot of great ideas from prior languages: it is a statically-typed, garbage-collected general-purpose applications programming language with excellent support for both object-oriented and functional programming. However, Wyvern is most interesting because of the new ideas it explores. Most of these are best illustrated through the examples below, but here's a brief overview of Wyvern's most interesting design features and what motivates them.

- Large programs must be composed from parts, so Wyvern has an advanced module system with features echoing those of Standard ML's module system. However, modern programs often load and (re-)compose modules at run time, so Wyvern modules and functors are first-class objects and functions, respectively, providing programmers with the power and flexibility they need.
- No one language can be good at everything, so Wyvern is extensible: libraries can define new syntax for the abstractions that they provide, and that syntax can be embedded seamlessly in Wyvern expressions.
- Engineers need to control the access that untrusted code has to resources such as the network or file system, and so Wyvern's module system is the first to be designed from the ground up to be capability-safe. We are designing an effect system that leverages these capabilities to provide lightweight and automatically-checked control of resources.
- More broadly, understanding and controlling the software architecture of a program is critical to understanding its properties and evolving it over time. Thus we are building a way of expressing software architecture as an integrated part of Wyvern programs, including a static view building on the module system and a dynamic view showing run-time components and the connections between them. Wyvern architectures are "live" in that changing the architecture specification affects the program semantics, and they are "trusted" in that they are guaranteed to be an accurate abstraction of what the program does.

2. Hello, World! in Wyvern

Here is a "Hello, World!" program in Wyvern (`examples/rosetta/hello.wyv`):

```
require 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 required from the operating system. 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.

3. Anonymous Functions

Wyvern has good support for functional programming, and 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
```

Function types can be denoted with an arrow, and we can annotate a variable with this type. If we annotate the type of the variable we are binding to the function, we can leave out the type annotation (and even the parentheses) on the function's argument:

```
val annotatedAddOne : Int -> Int = x => x + 1
```

This also works if we pass an anonymous function to a higher-order function:

```
val invokeIt = (f:Int -> Int, x:Int) => f(x)
invokeIt(x => x+1, 5)
```

The code above can be found in `examples/introductory/functions.wyv`

4. Functions in Wyvern

Consider the definition of the `factorial` function in Wyvern (`examples/rosetta/factorial.wyv`):

```
require 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.

Wyvern provides two nicer ways to write the `if` statement above:

```
def fact(n:Int):Int
    if (n < 2)
            1
        else
            n * fact(n-1)

def fact2(n:Int):Int
    if (n < 2) { 1 } else { n * fact2(n-1) }
```

In the first case, we put the `then` computation on a new, indented line, followed by `else` (also indented, but not as much), and then the `else` computation. In the second case, we put it all on one line of code, but put the `then` and `else` computations in curly braces to set them off.

It turns out that both versions of the `if` function can be user-defined. In the first case, Wyvern takes all the indented code and passes it as an argument to the `if` function. In the second case, Wyvern passes the code in each set of curly braces as a separate argument to the `if...else...` function.

If you'd like to see how these `if` functions are actually user-defined, look in the Wyvern standard prelude (`stdlib/prelude.wyv`):

```
import metadata wyvern.IfTSL
val ifelseARG = (cond:Boolean, tt:IfTSL.FnExpr, ff:IfTSL.FnExpr) => IfTSL.doif(cond, tt, ff)
val if = (cond:Boolean, blocks:IfTSL.Blocks) => IfTSL.doifblk(cond, blocks)
```

The import statement loads the `IfTSL` module from the `wyvern` package. The `metadata` keyword indicates that the library defines new syntax, in this case for `then` and `else` clauses of an `if` statement. We define an `if` shorthand for calling the `doifblk` function from that library. The `ifelseARG` function is how `if...else...` is written: keywords separated by a single argument are concatenated, and whenever more than one argument separates keywords (or comes after them all) `ARG` is added to the name. Parsers associated with the `IfTSL.FnExpr` and `IfTSL.Blocks` are defined in `stdlib/wyvern/IfTSL.wyv`.

5. Objects and Object Types in Wyvern

We can define a sumable integer list type as follows (`examples/introductory/objects.wyv`):

```
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.

6. Strings and Characters

String literals can be written in quotes, using the same escapes as in Java. Strings support several operations, including `==`, `<`, `>`, `length()`, and `charAt(Int)`. The last of these returns a `Character`, which supports `==`, `<`, and `>` operations. A simple program illustrating these is in `examples/introductory/strings.wyv`

Wyvern will support character literals but doesn't yet.

7. 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
```

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

8. Mutable State and Resource Types

Types with mutable state can be defined, but need to be marked as `resource` types (`examples/introductory/cell.wyv`):

```
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 = this.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.

9. Modules

We can define the `Cell` abstraction above in a module (`examples/modules/cell.wyv`):

```
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 = this.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. In this case, within the `examples` directory of the Wyvern distribution we have the directory `modules` that contains `cell.wyv`, so we can use it in a program as follows (see `examples/modules/cellClient.wyv`):

```
import modules.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
```

Wyvern files that define a type use a `.wyt` extension (for Wyvern Type), and you can find the above definition at `examples/modules/TCell.wyt`.

10. Functors

Although modules are objects, they cannot contain any state and they cannot encapsulate system resources. How can we express designs similar to those in other languages, in which modules do these things? We can do so by defining *functors*. A functor is a module that is defined as a function: it takes zero or more arguments and returns an object. For example, we can define a functor that returns a stateful `Cell` object (`examples/modules/cellAsModule.wyv`):

```
module def cellAsModule()

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 `cellAsModule` 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. The `module def` syntax indicates this; it is reminiscent of the `def` syntax for defining functions. The word *functor* that we used from this is based on a similar construct in Standard ML (and inspired by category theory, for the mathematically inclined). We can use `cellAsModule` in a program as follows (`examples/modules/cellModuleClient.wyv`):

```
import modules.cellAsModule

val m1 = cellAsModule()
val m2 = cellAsModule()
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 called the functor `cellAsModule` twice, and each resulting object instance has its own internal state.

11. Module Parameters

If objects are produced by functors (module functions), we expect to be able to pass parameters–and so we can. First let's define the type that `cellAsModule` returns. For convenience, we will put this type in a file `TCellAsModule.wyt` (here `.wyt` stands for Wyvern Type):

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

Here is a client of the cellAsModule (`examples/modules/cellClientFunctor.wyv`):

```
module def cellClientFunctor(cell : modules.TCellAsModule)

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

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

We can put module parameters in between the parentheses in the definition of the functor, specifying the type in the usual way.

Now we can use `cellAsModule` together with `cellClientFunctor` in a program (`examples/modules/cellClientMain.wyv`):

```
import myPackage.cellAsModule
import myPackage.cellClientFunctor

val client = cellClientFunctor(cellAsModule())
client.addOne()
client.getValue() // evalutes to 1
```

12. Dynamic Types

Wyvern is intended to be a mostly statically typed language. However, while getting parameterized types 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 now that parameterized types work well, we are 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.

13. 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:

```
// NOTE: this example does not typecheck
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.

14. Some More Examples

The Wyvern standard library files are in subdirectories of `stdlib`. For example, earlier we used the `IfTSL` library, defined in `stdlib/wyvern/IfTSL.wyv`. When importing files from the standard library, leave out `stdlib`; i.e. in the example above, just use `import wyvern.IfTSL`

Platform-specific definitions are in the `platform` subdirectory of `stdlib`, in a sub-subdirectory named after the platform (e.g. `java`). For example, `stdout` is defined in `stdlib/platform/java/stdout.wyv` for the `java` platform; there is an analogous definition for the `python` platform too. The definition of `stdout` for `java` uses some Java helper code defined in the Java class `wyvern.stdlib.support.Stdio`. These libraries should be imported without using `stdlib.platform.java`.

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`.

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