The Zilk Programming Language

Ilkka Kokkarinen, ilkka.kokkarinen@gmail.com

Version 0.04 "Simon says", March 31 2005

Table of Contents

1.

Zilk is a dynamic programming language heavily inspired by Ruby and, to a lesser extent, Python. It supports object-oriented programming, inheritance and polymorphism so that both classes and methods are first-class objects of Zilk.

Zilk runs on top of Java so that Zilk programs can dynamically create and use Java classes and objects with the exact same syntax as ordinary Zilk classes and object. A Java class can even be a superclass to a subclass written in Zilk. This makes Zilk an enjoyable alternative to Java while getting to use (most) of the Java API, including Swing, regular expressions, threads etc. etc. Of course Zilk functions, classes and objects can similarly be accessed and manipulated from Java. (I eventually want to write a proper Zilk API for this, since right now this access would take place directly through the main class of Zilk.)

I started writing Zilk about a month ago as an academic curiosity to see how creating a programming language would go. As I kept writing it, the whole thing just kept getting more interesting and powerful, plus it gave me a higher appreciation of programming in Java. Garbage collection and polymorphism made things a lot easier compared to how they would have been in C or other such lower-level languages.

I have decided to release Zilk under the GNU Lesser General Public License so that other people can use it and perhaps extend it any way they want, but absolutely no warranties are given or implied. Lots of bugs probably remain in the code, so use it at your own risk.

2. Variables and values

The syntax of Zilk is very free and inspired by my experiences in teaching Java programming to beginners. Unlike in many other languages, Zilk statements do not have to be separated by semicolons or other such separators, although separators can be used if they are needed or preferred. Even so, Zilk is not sensitive to whitespace, but any kind of indentation and whitespace policy can be used (as opposed to Python).

Zilk is a dynamically typed language. Therefore the programmer does not have to (and cannot) declare types of variables in the Zilk program. Any variable can at any time refer to any type of object. A variable comes to existence at the first assignment. Trying to use a nonexistent variable terminates the program with an error message.

As of this writing, Zilk built-in types include integer, floating point number, character, boolean, string and list. The list is actually an arraylist which can be indexed in constant time, but which can grow freely while it automatically takes care of its memory management behind the scenes. In the spirit of the free syntax of Zilk, the elements of a list can be separated just by whitespace. Commas can also be used for this purpose, if this feels more familiar.

The following example program demonstrates these types. Note that Zilk consciously throws away several harmful traditions in programming language syntax, especially the tradition of using = for assignment and == for equality testing. In Zilk, the operator `is` is used for assignment, and `eq` means the equality testing (and `neq` is the inequality testing). As synonyms for these, you can also use the operator `:=` for assignment and the operators = and `!=` for equality and inequality testing.

```
n is nil            # the nil object
i is 99             # integer
f is 1.2345         # floating point number
b is true           # true and false are the boolean values
s is "Hello world"  # string literals are inside either single or double quotes
c is ?$             # character constants begin with a ?
println c           # outputs $
li is [7 i [f b s]] # a heterogeneous list
println li          # outputs [7 99 [1.2345 true Hello world]]
println li.length   # outputs 3
println li[-2]      # outputs 99
```

The keywords `println` and `print` can be used to output any value, respectively with and without the trailing newline. They evaluate to the value of the object that generated the output, not to the string representation that was actually output.

To convert a value of one type explicitly to another type, use the attributes `int`, `float`, `char` and `boolean` as applicable. To explicitly convert a value to a string, add it to the empty string. (This doesn't work for lists, because adding anything to a list makes it a new element with the list. For lists you should therefore use the list method `join`.)

Zilk strings can be delimited with either single or double quotes. If single quotes are used, double quotes do not need to be escaped inside the string, and vice versa. Character constants are denoted by a question mark followed by the character, for example `?$` for the dollar sign and `?\n` for the newline.

Zilk lists are untyped, so each element of the list can be any Zilk object. Both strings and lists have the attribute `length` for accessing their length, and they allow indexing to access their individual elements. As usual, the index of the first element is zero. Unlike in Java, negative indices can be used to access characters or elements counting from the end.

```
num is [1 2 3]
```

```
num[6] is 99    # uh oh, we go past the end of the array
println num     # outputs [1 2 3 nil nil nil 99]
println num[-5] # outputs 3
```

If the index goes past the end of the string, the empty string is returned, whereas a list automatically grows to accommodate the new element, filling up the new elements with the special value `nil`.

To create a multidimensional list of given size, use the method `$List.create`. This method can take an arbitrary number of dimensions. To create a one-dimensional list with given initial values, use the class `$List` as a function and pass the desired elements as its parameters.

```
println $List.create(2 3)       # outputs [[nil nil nil][nil nil nil]]
println $List(42 1.0 'Hello')   # outputs [42 1.0 Hello]
```

Outside the parameters of a function call and the lists that are explicitly created with square brackets, commas implicitly create lists from arbitrary statements. This can be used to create multiple simultaneous assignments in the style of Python and Ruby.

```
a,b is 12-2,5*4
println a,b                     # outputs [10 20]
a,b is b,a                      # swaps two variables
println a,b                     # outputs [20 10]
```

Unlike in Python, implicitly created lists are real mutable lists instead of immutable tuples, which Zilk does not have at all. Note also that the comma has a lower precedence than anything else except the assignment operator `is`, so writing something like

```
assert a,b,c eq 1,2,3
```

would be the same as writing

```
assert a,b,(c eq 1),2,3
```

which is probably not what was meant. This assertion always succeeds, since the expression that follows it is an implicitly constructed five-element list, which is considered a true value.

As in Java, adding anything else (except a list) to a string temporarily converts it to a string and creates a new string from concatenation of these two. For the lists, adding two lists concatenates them on the same level, whereas concatenating a list with anything else adds that something as a new element to the list. To insert a list into the other list as an element, use the method `append`:

```
println [1 2 3] + 4.0           # outputs [1 2 3 4.0]
println [1 2 3] + [4 5 6]       # outputs [1 2 3 4 5 6]
println [1 2 3].append(4)       # outputs [1 2 3 4]
println [1 2 3].append([4 5 6]) # outputs [1 2 3 [4 5 6]]
println [1 2] + 'Hello'         # outputs [1 2 Hello]
```

The class `Dictionary` implements a dictionary using a hash table. Its elements can be accessed with indexing, and both keys and values can be any objects. However, changing the value of the key object may result in erroneous behavior, so it is best to use only the immutable types such as integers and strings as keys.

```
m is Dictionary()
m["Tom"] is 180
m["Bob"] is 190
println m["Tom"]      # outputs 180
println m["Bob"]      # outputs 190
```

As in Ruby, ranges between two limiting values can be created with "`..`" and "`...`" operators. The

two-dot version defines an inclusive range, and the three-dot version defines an exclusive range that does not include the last value. Ranges can be used to index lists and strings.

```
a is [1 2 3 4 5 6 7]
println a[0..5]       # outputs [1,2,3,4,5,6]
println a[0...5]      # outputs [1,2,3,4,5]
println a[2..-2]      # outputs [3,4,5,6]
println a[2...-2]     # outputs [3,4,5]
```

To delete a variable or a field, use the operator `del` followed by the name of the variable. Just like in Python, the same operator can be used to erase an element or several elements from the list, indexed either with an integer or an integer range.

```
a is 99               # create a new variable
println a             # outputs 99
del a                 # remove the variable
println a             # outputs nil
del a                 # does nothing
println a             # outputs nil
b is [1 2 3 4 5 6 7]  # a list to work on
del b[2]              # delete element in place 2
println b             # outputs [1,2,4,5,6,7]
del b[2..-2]          # delete all elements whose indices are in range 2..-2
println b             # outputs [1,2,6,7]
```

3. Control structures

Like Ruby, Zilk makes no distinction between expressions and statements. Every statement ultimately evaluates to some value that can be used.

```
a is if b > c [1 2] else 'Hello' endif  # makes a to be either [1 2] or 'Hello'
```

As you can see, Zilk has the traditional `if-else-endif` structure for conditions and the traditional `while-endwhile` structure for loops.

```
a is 50
while a > 0
  if a mod 3 eq 0     # check if a is divisible by three
    print a + " "
  endif
  a is a-1
endwhile
```

(However, see the later section “Iterator methods” for a better way to stepping through lists and integer ranges.)

In Zilk, each structure that starts with the keyword `X` is terminated with the appropriate keyword `endX`. You can also use a generic keyword `end` to terminate any structure, if you are not worried about mismatching the nested ends. Any such structure can also be empty. Note that there is no keyword `begin`: the structure type begins the structure.

Both `if` and `while` evaluate to the value of the last statement in the body, or if the body of the `while` loop is not executed at all, to `false`. In either structure, the condition does not have to be wrapped inside parenthesis, but of course any expression can we wrapped inside parentheses if this is required by clarity or resolving ambiguity.

Instead of the keywords `if` and `while`, symmetric keywords `unless` and `until` can be used to negate the condition. Unlike Python and Ruby, Zilk does not support using these as statement modifiers

in the style of Ruby and Python, since there is no way to make the syntax unambiguous without requiring delimiters between statements.

The statement `break` jumps out of the innermost `while`- or `until`-loop that the execution is currently in, and the statement `continue` jumps back to the beginning of that loop. Unlike in most other languages, these statements also work from inside functions. If a function is called inside a loop so that the function body contains a `break` statement that is not itself surrounded by a loop inside the function body, the statement immediately terminates *both* the call and the loop in the calling outer scope.

For the built-in types and object, only the empty list, boolean value `false` and the special object `nil` are considered to be false values in conditions. All other values of the built-in types are considered true values, and this includes zeros and empty strings. Every function object is also considered to be a true value. For objects constructed from user-defined classes, the method `isTrue` determines if the object is considered a true value.

Note that instead of the traditional cartoon character swearing, Zilk uses English words for the common operators. The basic logical operators are `and`, `or` and `not`, the remainder of integer division is computed with `mod`, and the bitwise operators of integers are denoted with `band`, `bor` and `bnot`. The logical operators `and` and `or` are guaranteed to evaluate from left to right and short-circuit the result so that the second parameter is evaluated only if the first parameter did not determine the result.

```
println 42 or false   # outputs 42
println false or 42   # outputs 42
println true or 42    # outputs true
println 42 or true    # outputs 42
println 42 and false  # outputs false
println false and 42  # outputs false
println 42 and true   # outputs true
println true and 42   # outputs 42
```

When combining these operators, note that unlike many other languages, both `and` and `or` have the same precedence in Zilk. So if you don't want your mixed expression to be evaluated from left to right, use the parentheses to impose your desired order.

Even though Zilk is otherwise insensitive to whitespace, you cannot use a space before the left parenthesis in function calls and indexing. This restriction is necessary to make the syntax unambiguous. Otherwise, for example, the list `[a [2]]` could be read in two different ways: either it is a one-element list that contains the element computed by indexing the variable `a` with `2`, or a two-element list that contains first the value of variable `a` and then a single-element list `[2]`.

4. Functions in Zilk

The keyword `def` can be used to define a function. It is followed by the name of the function and the parameters listed inside *vertical bars* (**note**: not parentheses!). If the function takes no parameters, the empty pair of bars can be omitted. At the function call, parentheses are used. (Empty parentheses cannot be similarly omitted in the actual function call, so that Zilk remains able to distinguish between the function call and just accessing the function object.) The body of the function ends with the keyword `enddef` or simply just `end`.

```
def cube |a|
  a*a*a
enddef

println cube(3)  # outputs 27
```

```
alias is cube    # assign another variable point to the function object
alias(4)         # outputs 64
```

The value of the last expression in the function body is the return value of the function, so no specific `return` statement is needed. However, a `return` statement can be used inside a function, where it immediately terminates the function execution. Since every Zilk function call evaluates to a value, the keyword `return` cannot be used alone, but it *must* be followed by the expression that gives the actual return value.

Functions are real first-class objects in Zilk, so variables can reference them same as integers or strings. Functions can be given as parameters to other functions, stored in lists and dictionaries, returned as results from functions calls etc.

To define an anonymous function, use the keyword `lambda`. Unlike functions defined with `def`, anonymous functions remember their surrounding context, as the following classic functional programming example shows.

```
def makeAccumulator |value|
  lambda |add| value is value + add endlambda
enddef

acc is makeAccumulator(5)
println acc(2) # outputs 7
println acc(3) # outputs 10
```

A function can be called with more arguments than it has been declared to take. These extra arguments are collected into a list named `rest`. This makes it pretty easy to write a function that takes an arbitrary number of parameters and collects them into a list:

```
def toList() rest enddef
```

If you precede a list with an ampersand, its contents are exploded into parameters.

```
def sum |a b c d e| a+b+c+d+e enddef
println sum(&[1 2] 3 &[4 5])  # outputs 15, same as sum(1 2 3 4 5)
list is [2 3 4]               # list to explode in the next line
println sum(1 &list 5)        # also outputs 15
```

The list explosion with ampersand also works for lists that are outside parameter lists, regardless of whether they are explicitly or implicitly defined:

```
assert 1,2,3,4,5 eq [&[1 2 3] &[4] &[5]]   # true
assert 1,2,3,4,5 eq 1, &[2 3], &[4], &[5]  # true
```

5. Iterator methods

In addition to the traditional procedural control structures, Zilk lists themselves offer methods that can be used to conveniently iterate over them. The method `each(f)` evaluates the function `f` for each element inside the object, and returns the result of the last evaluation. The method `map(f)` applies the function `f` to each element inside the container and returns a list consisting of the results. The method `filter(f)` creates a list of elements `x` for which `f(x)` evaluates to a true value. The method `accumulate(f init)` accumulates the result of a two-parameter function for the elements of the list, starting with the initial value `init`.

```
a is [1 2 3 4 5 6 7 8 9 10]
println a.map({ |x| x*x })                  # list of squares
println a.filter({ |x| x mod 2 eq 0 })      # filter even numbers
```

```
println a.reverse()                          # (take a guess)
println a.accumulate({|x y| x+y} 0)          # outputs 55 (sum of elements)
```

Each of these methods can also be given two extra parameters `first` and `last` that the method uses to limit the operation to only to the inclusive range `(first..last)` of the list.

The methods `eachWithIndex` and `filterIndices` work like `each` and `filter`, but they construct the list of indices instead of the list of actual elements.

Ruby enthusiasts such as myself are probably happy to see that curly braces can be used instead of `lambda` and `endlambda`. They are considered the same down in the tokenization level.

```
(1..10).each(lambda |x| print x+" " endlambda)
(1..10).each({|x| print x+" "})
```

Warning: Ruby-style postfix block passing does not work, so you cannot do

```
(1..10).each {|x| print x+" " }     # This doesn't work in Zilk same as in Ruby
```

because in Zilk this is just two consecutive statements: the first one accesses the method `each` in the range object `(1..10)` (without calling this method), and the second one defines a `lambda` function which is immediately discarded. This makes calling functions with a single lambda expression parameter slightly annoying, because you have to write the both ordinary parenthesis (that surround the argument list) and the curly braces (that surround the lambda expression) inside them.

The next example illustrates some convenient methods of strings. The method `split` splits the string into an array of strings at each occurrence of the separator. The iterator `eachLine(f)` executes the function `f` for each line of the string. The methods `startsWith(other)` and `endsWith(other)` can be used to check whether the string `other` is a prefix or suffix of this string. The method `chomp(suffix)` removes the `suffix` from the end of the string if it is there, and otherwise returns the strings as it was.

```
a is "Hello world can you hear me".split(" ")
println a        # outputs [Hello,world,can,you,hear,me]
lines is 0
"Hello there,\nwe are\nthe robots".eachLine({ lines is lines + 1 })
println lines    # outputs 3
println "Hello world".startsWith("Hell")   # outputs true
println "Hello world".endsWith("orld")     # outputs true
println "Hello world".chomp("rld")         # outputs Hello wo
println "Hello world".chomp("forld")       # outputs Hello world
```

Note that like in Java and unlike in Python and Ruby, Zilk strings are immutable, so `chomp` and other such seemingly destructive methods create and return a new string object to represent the result. As in Ruby, methods that actually change the first object have their names ending with the exclamation mark, whereas methods that create a new object for the result are named normally. For the Zilk built-in types, such destructive methods are only defined for lists, since other built-in types of Zilk are immutable.

```
list is [1 2 3 4]
println list.append(42)         # outputs [1 2 3 4 42]
println list                    # outputs [1 2 3 4]
println list.append!(42)        # outputs [1 2 3 4 42]
println list                    # outputs [1 2 3 4 42]
```

Note that unlike in Ruby, the destructive and the nondestructive versions of the method behave similarly with respect to return values. (In Ruby, destructive versions of methods return `nil` if they didn't change anything.) Other useful list methods are `partition` and `sort`, illustrated below. The

method `sort` uses the operator < by default, but it can be given some other comparator to use. This comparator must be a function that returns true if its first argument is less than its second argument. The method `join` is the inverse of the string method `split`.

```
def isEven|x| x mod 2 eq 0 enddef
list is [45 99 22 23 0 445, -33 9232]
both is list.partition(isEven)  # returns a list of two lists
println both[0]                 # outputs [22,0,9232]
println both[1]                 # outputs [45,99,23,445,-33]
println list.sort               # outputs [-33,0,22,23,45,99,445,9232]
println list                    # outputs [45 99 22 23 0 445, -33 9232]
list.sort!                      #
println list                    # outputs [-33,0,22,23,45,99,445,9232]
println list.join("$")          # outputs -33$0$22$23$45$99$445$9232
```

6. Classes in Zilk

A class is defined with the keyword `class`, followed by the body of statements that define the class and the keyword `endclass`. At the time of class definition, the statements are executed inside the class context. These statements can therefore be any statements, not just method definitions and variable assignments. Their effects persist in the class context. For example, nested classes are just ordinary classes that are defined in a class context instead of some other context. (However, such classes behave the same way as the *static nested classes* of Java, that is, they must access the fields and methods of the surrounding class through an object.)

Note that with the exception of global variables (which start with a character `$`), assigning to a variable inside a function automatically makes it a local variable. Therefore in a method where you want to assign into an object field instead of creating local variable of the same name, you should either precede it with the reference `this` (Java style) or start its name with the character `@` (Ruby style). To access a class variable from a method, either precede it with the reference `klass` or as in Ruby, start its name with the characters `@@`.

```
class Person

  @@count is 0 # a class variable (also works without the @@)

  println "Now executing the body of class Person"

  def initialize |name, height, bestfriend|
    println 'Constructor of the class Person'
    @name is name                  # Ruby style works
    this.height is height          # Java style also works
    this.bestfriend is bestfriend
    this.i is Inner(height)
    @@count is @@count + 1         # @@ denotes a class variable in a method
  enddef

  def greet
    println "Hello, I am " + this.name + " and I am " + @height + " cm tall."
    if this.bestfriend
      println "My best friend is " + @bestfriend.name + ". Go on, dude."
      this.bestfriend.greet()
    else
      println "I have no best friend."
    endif
  enddef
```

```
  # Here we have a class inside another class. This works the same way as
  # the static nested classes in Java.
  class Inner
    def initialize |x person|
        println 'Constructor of the inner class'
        @x is x
        @person is person
    enddef
    def output
      println 'My own value is ' + @x
      println 'Name of my person is ' + @person.name
    enddef
  endclass

  def getCount klass.count enddef
  def useInner() this.i.output() enddef

  println "Class body Person finished"

endclass
```

To construct an object from a class, use the name of the class as a function. The parameters that you provide are passed on to the constructor.

```
p is Person("Tommy", 190, 0)
q is Person("Billy", 180, p)
p.greet()  # Tommy introduces himself
q.greet()  # Billy introduces himself and then his best friend Tommy
```

The method named `initialize` is the constructor for the class. When an object is constructed in Zilk, the constructors of the superclasses are automatically called in descending order down the inheritance chain starting from `$Object`. The constructor arguments are passed on to the superclass constructor as they are by default, but if you want to use different constructors for the superclass, define the method `toSuper` in the subclass. When this method gets automatically called with the subclass constructor parameters, it should return a list of arguments intended for the superclass.

When an object is created, it recognizes everything that has been defined in the class body. New attributes can be dynamically added to both classes and objects. Definining an attribute for a class makes it visible for all objects constructed from that class, whereas defining an attribute for a single object makes it only visible for that object.

```
def p.greet
  println "Hi there, my name is " + @name + "."
  println @@count + " people have been created."
enddef
p.greet()     # new greeting
q.greet()     # old greeting from the class Person
del p.greet   # remove the new greeting
p.greet()     # old greeting from the class Person
```

As in Java, inheritance is indicated by following the name of the class with the keyword `extends` and the name of the superclass. If a class does not extend any superclass explicitly, it implicitly extends the global superclass `$Object`.

To check if the given object has been constructed from a given class, use the operator `instanceof`. The operator follows the inheritance chain where necessary.

```
println p instanceof Person        # outputs true
println [1 2 3] instanceof Person  # outputs false
```

```
println p instanceof $Object       # outputs true
```

If the operator `instanceof` is applied to two classes instead of an object and a class, it checks if the first class is a proper subclass of the second. (What really happens is that `instanceof` follows the `klass` references of objects, and in the `ZClass` objects this reference points to the superclass object.)

7. Methods in classes

To get the arithmetic and comparison operators of Zilk to work for your class, override the following methods that correspond to them. (Perhaps in the future Zilk will allow Ruby-style naming these methods exactly the same as the operator that they override, but not yet.) Note that for order comparisons, you only need to define the operators < and `eq`, since the other four follow automatically from these. (Actually they would follow automatically from the operator < alone if it was a total order. However, equality testing is meaningful for many objects for which the order comparison cannot be reasonably implemented as a total relation, so it is good to have equality testing as an independent operation.)

```
class OperatorOverrider

  # arithmetic
  def add |other| ...           # +
  def subtract |other| ...      # -
  def multiply |other| ...      # *
  def divide |other| ...        # /
  def modulus |other| ...       # mod
  def negate ...                # unary minus

  # comparisons
  def lessThan |other| ...      # <
  def equals |other| ...        # eq
  def isTrue ...                # truth value for conditions

  # indexing
  def getIndex |key| ...        # b is a[idx]
  def setIndex |key value| ...  # a[idx] is b
  def delIndex |key| ...        # del a[idx]

  # other
  def iterator ...              # get an iterator into this object
  def toString ...              # convert this object to a string
  def toFunction ...            # use this object as a callable
  def toSuper ...               # convert subclass constructor arguments
                                # to superclass constructor arguments
end
```

To allow the object to be used as a callable function, define the method `toFunction` in it.

To allow `each` and its derivative iterator methods work for the objects of your class, define the method `iterator` in it. This method should return some object that works as the iterator to your class. The iterator should respond to methods `hasNext` that is true if there are elements remaining to be iterated over, and `next` that returns the value that the iterator is currently at and advances the iterator one step forward. The method `iterator` can take two parameters `first` and `last` to limit to range of iteration, but some iterators might choose not to do anything with these parameters.

Object methods can also be extracted and assigned to variables for later use, perhaps for passing them

to other functions as parameters. Of course, the whole point of these extracted methods is that they remember what object and the version of the method they were bound to, as the next example shows.

```
class Holder
  def initialize|x| @x is x enddef
  def output println @x enddef
endclass

a is Holder(3)      # Holder object with value 3
b is Holder(5)      # Holder object with value 5
a.output()          # outputs 3
b.output()          # outputs 5
m1 is a.output      # take the method output of a
m2 is b.output      # take the method output of b
m1()                # outputs 3
m2()                # outputs 5
def Holder.output
  println @x * @x   # redefine the method output of class Holder
enddef
a.output()          # outputs 9
b.output()          # outputs 25
m1()                # still outputs 3, as it is bound to old version
m2()                # still outputs 5, as it is bound to old version
```

Zilk supports dynamic dispatching of methods with the special method `call`. This method uses its first parameter, which has to be a string, as the name of the actual method to be called for the rest of the parameters. For example, after the assignment `methodName is "foo"`, each of the following expressions does the exact same thing:

```
a.foo(42)
a.call("foo", 42)
a.call(methodName, 42)
a.call("call", "call", "call", methodName, 42)
```

Note that `call` can be extracted for later use, as it is itself a pure Zilk method.

8. Exception handling

Zilk's exception handling is straightforward and works like in Java, except that there is no way to distinguish between different types of exceptions in compile time. (For this reason, Ruby-style keyword `rescue` is intentionally used instead of the Java-style keyword `catch`.) Thrown exceptions can be any Zilk objects whatsoever, and it is the responsibility of the catching block to decide what it does with the object that it caught.

You can throw any Zilk object whatsoever as an exception with the keyword `throw`, followed by the expression that evaluates to the object. Uncaught exceptions terminate the current function immediately and return to the calling level.

The `try-rescue-finally` block tries to execute the block of code that follows the keyword `try`. If this block of code does not throw exceptions, the `rescue` block is skipped. Otherwise the execution immediately transfers to the beginning of the `rescue` block, and the caught object appears to the current context under the name `caught`. Either way, the `finally` block is executed after that, even if the `rescue` block decides to throw an exception of its own. Both `rescue` and `finally` blocks are optional after the `try` block, but the whole thing again has to end with either `end` or `endtry`.

```
def tosser |x|  # this function will throw an exception when called
  try
```

```
    println "Throwing " + x
    throw(x)
    println "This text won't really print..."
  finally
    println "... but this will"
  endtry
end

def middle |x|  # an intermediate function that has finally, but no rescue
  try
    tosser (x)  # this call will throw an exception
    println "This text will not print."
  finally
    println "But again, this will."
  endtry
enddef

try
  middle(55)
rescue
  println "Caught an object " + caught    # outputs ...55
  println "Now I will throw it forward."
  throw caught                            # will be a run-time error if uncaught
finally
  println "finally of outside"
endtry
```

Some runtime errors of Zilk throw an exception that you can catch and handle. These exception objects s are constructed from standard classes, so you can use `instanceof` to check from the object `caught` what type of error actually occurred. The following predefined exception classes are all subclasses of the class `$Exception`.

- `$JavaError` indicates that using Java class or creation of Java object was unsuccessful.
- `$ParseError` indicates that dynamic compilation failed.
- `$IndexError` indicates that indexing was done with an illegal index.
- `$ArithmeticError` indicates an illegal arithmetic operation, e.g. division by zero.
- `$AssignmentError` indicates an assignment to something that you cannot assign to.
- `$AssertionError` indicates that an assertion failed.
- `$UnknownNameError` indicates an attempt to use the value of an undefined variable.

9. The dynamic nature of Zilk

As in Ruby, function and class declarations are not compile-time constructs but they are themselves executable expressions. Function and class declarations can occur anywhere in the program, for example inside conditional statements, not just at the top level.

The program is parsed and executed in a single pass from beginning to end, as it is a linear chain of expressions that make no references to the future at the time of their execution. (Of course a the body of a function `foo` may refer to another function `bar` defined later, but as long as `bar` has been defined at the time `foo` actually gets executed, this is not a problem.) As a side effect, a statement can modify the names that exist in various scopes, assigning them to refer to dynamically created objects. A function declaration such as

```
def sum |a b c|
  a+b+c
end
```

is essentially an assignment that makes the variable `sum` in the current scope refer to a function object constructed from the function body. (Even if this statement is inside a loop, the function body is parsed only once. After all, the loop statement and all of its contents have already been fully parsed before its execution can begin.)

In some dynamic aspects, Zilk actually goes further than even Ruby. For example, Zilk allows defining classes inside functions and methods. Like the `lambda` expressions, classes also remember their static context. In the following example, the function `where` is defined both the global context and again inside the function `bar`. When a class is defined inside a function, the class remembers the context of the function, even after when it is returned from the said function which has terminated. Even though there is another version of `where` available in the outer scope, the class still internally uses the inner scope's version of that method.

```
def where println "outside bar" enddef

def bar

  def where println "inside bar" enddef

  class T
    def see
      where() # let's see where we are
    end
  end

  b is T()    # construct an object from T
  b.see()     # outputs "inside bar"
  T           # return the class as function result

end

K is bar()    # call the method, get back a class and call it K
a is K()      # construct an object from it
a.see()       # still outputs "inside bar"
```

This would probably be a very powerful mechanism for metaclass programming. For example, a factory method can dynamically construct a class (or a `lambda` function, for that matter, since both are called the same way) which is then returned to the caller who uses it to construct new objects.

Note also that in Zilk both `def` and `class` accept arbitrary compound names for function and class names, and in `class` the superclass can be given by any expression. Let's assume that you have a list called `myClasses` whose each element is a class. When you want to create a new class, you can dynamically choose its superclass and write something like

```
class myNewClass extends myClasses[idx]
  ...         # myNewClass definition goes here
endclass
```

to use the value of the variable `idx` to choose which class is used as the superclass.

10. Executing code from elsewhere

Since both the Zilk parser and the execution environment run simultaneously on top of Java, the parser

can be used to generate new Zilk code at runtime. The keyword `compile` takes a string object and compiles it into an anonymous function of zero parameters.

```
f is compile "println 'Hello world!'"
f() # outputs Hello world!
```

To execute a Zilk program that resides in another file, use the function `execute` that takes the filename as parameter. The function looks for the file in the directories listed in the environment variable `PATH`. After that program has finished, its results remain in the context where `execute` was called, because `execute` is really a lambda expression.

For debugging, the keyword `assert` evaluates the expression that follows it and throws a Zilk exception if the expression evaluates to a false value.

The header file *std.zil* defines a small interactive Zilk shell that can be started with by calling the function `irz`. The shell displays a prompt from the global variable `$PROMPT` and reads, parses and executes one line of code at the time. This shell is very primitive, since unlike Ruby's `irb` or the Python shell it doesn't automatically allow multiline statements. (To create a multiline statement, put a backslash after the line.) To exit from the shell, simply use the statement `break`.

11. Using Java from Zilk

To import a Java class to be used inside Zilk, call the function `java` with the full name of the class in a string. If you don't give the full name but just the class name, the function uses the global list `$IMPORT` to determine which packages it looks for the class. After that, you can use the class pretty much as you would use a Zilk class, with some small differences emerging from the fact that Java distinguishes between a field and a method whereas Zilk does not. But you can, for example, use the Java class as the superclass of your own Zilk class.

This possibility to use Java classes directly from Zilk makes interaction with Java code pretty easy. (It also saves work for me in having to implement most of the standard libraries of Zilk, since the perfectly good underlying Java classes are already there!) For example, the class `Dictionary` that was seen earlier is just `java.util.HashMap` with the Zilk indexing methods `getIndex`, `setIndex` and `delIndex` added on top of it. Therefore other useful `HashMap` methods such as `size()` can be directly used for `Dictionary`.

For a first example of how to use standard Java classes in Zilk, let's say that we want to use the random number generator. Well, Java already has one that is easy to use.

```
Random is java("java.util.Random")
rng is Random()
(0..10).each( { println rng.nextInt() + "\t" + rng.nextDouble() } )
```

Since Java does not have the keyword `rest` to capture the extra arguments in a function call, the number of parameters and their types must match the signature of the Java method exactly, although certain reasonable substitutions for the types are allowed. Argument exploding works the same for Java methods as for the Zilk methods.

Zilk can automatically convert its own data types to the corresponding ordinary Java types that it then uses when calling Java methods. Symmetrically, the Java method's return values are automatically converted into the corresponding Zilk data types. Java's built-in types are converted to corresponding Zilk basic types, Java arrays are converted to Zilk lists, Zilk objects are taken back to Zilk exactly as they are, and Java objects of other types are wrapped inside a Zilk shell object so you can still call their methods in Zilk and pass them as arguments to other Java methods. If these conversions do not work

the way you need, you need to write a Java class to do the appropriate conversion for you.

One important application of Java in Zilk is to use Swing to give your Zilk program a graphical user interface. To listen to the events of Swing components, just use the helper class `ZSwingHelper`. An instance of `ZSwingHelper` can be set to listen to events of a Swing component and route them to appropriate Zilk methods to handle. For example, to listen to mouse events of a Swing component `c`, call the method `addMouse(c)` in the instance of `ZSwingHelper`. You can then use `setMethod` to register the actual Zilk method to be executed when the Swing component emits the given type of event. The registered Zilk method receives the original Java event object as a parameter and can use its methods to extract all the available information about the event.

The following example opens two `JFrame` windows, each of which contains a button, a textfield and a panel that the Zilk program draws a filled red rectangle into. The window listens to its own mouse events and the action events of the button and the textfield, and responds to them appropriately.

```
# Make javax.swing visible to us
$IMPORT.append!("javax.swing")

JFrame is java("JFrame")
JButton is java("JButton")
JTextField is java("JTextField")
FlowLayout is java("FlowLayout")
ZPanel is java("ZPanel")
Dimension is java("Dimension")
Color is java("Color")
ZSwingHelper is java("ZSwingHelper")

class MyFrame extends JFrame

  @@eventTunnel is ZSwingHelper()

  def toSuper |title x y| [title] enddef

  def initialize |title x y|
    @title is title
    setSize(400 200)
    setLocation(x y)
    getContentPane().setLayout(FlowLayout())
    setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE)
    # Create and add the button.
    @b is JButton("Click me")
    getContentPane().add(@b)
    # Create and add the textfield.
    @t is JTextField(20)
    getContentPane().add(@t)
    # Create and add the panel.
    @p is ZPanel(this.myPaint)
    @p.setPreferredSize(Dimension(100 100))
    @p.repaint()
    setVisible(true)

    # Make the event tunnel listen to the component events.
    @@eventTunnel.addAction(@t)
    @@eventTunnel.addAction(@b)
    @@eventTunnel.addMouse(this)
    @@eventTunnel.addWindowFocus(this)

    # Set up the Zilk methods to call in case of an event.
    @@eventTunnel.setMethod("actionPerformed", @b, this.click)
    @@eventTunnel.setMethod("actionPerformed", @t, this.text)
```

```
    @@eventTunnel.setMethod("mousePressed", this, this.pressed)
    @@eventTunnel.setMethod("windowGainedFocus", this, this.gainedFocus)

  enddef

  def text
    println "Received input: " + @t.getText()
    t.setText("")
  enddef

  def click
    println "Button click in " + @title
  enddef

  def pressed |me|
    println "Mouse pressed at " + me.getX() + " " + me.getY()
  enddef

  def gainedFocus
    println @title + " gained focus"
  enddef

  def myPaint |g|
    g.setColor(Color.RED)
    g.fillRect(20 20 50 50)
  enddef

endclass

MyFrame("First" 200 200)
MyFrame("Second" 600 300)
```

At the moment of writing this document, the class `ZSwingHelper` recognizes the following event interfaces: `ActionListener`, `MouseListener`, `KeyListener`, `MouseMotionListener`, `ItemListener`, `FocusListener`, `WindowListener` and `WindowFocusListener`.

What if you want to do your own painting? As you can see from the previous example, the class `ZPanel` extends the class `JPanel` so that when you construct a `ZPanel` object in Zilk, you pass it a function that itself gets a `Graphics` object and does the drawing there. The method `paintComponent` of `ZPanel` calls this Zilk method to do the actual drawing.

Last but not least, if you have written a Zilk program that uses AWT or Swing for its graphical user interface, it is very easy to run it as an applet. The package `org.zilk` contains an applet class `ZAppletMaker` that initializes the Zilk environment and launches a given Zilk program. Just copy the files *Zilk.jar*, *std.zil* and whatever file that contains your Zilk program (let's assume here that it is called *myprogram.zil*) to the same folder that your web page is in, and in the HTML file of the web page insert something like

```
<applet code="org/zilk/ZAppletMaker.class"
        file="myprogram.zil"
        archive="Zilk.jar"></applet>
```

If your program happens to be split in multiple files, no problem. Just put them to the same directory where the previous files are. When a Zilk program knows that it is running as an applet, `execute` loads the files over the network instead of trying to load them from the local filesystem.

12. Using Zilk from Java

To run Zilk code from your Java program, the main class `org.zilk.Zilk` has four static methods. First, the method

```
public static void initializeZilkEnvironment() throws ZRuntimeError
```

must be used to initialize the Zilk environment. After that, you can create blank Zilk contexts by calling the method

```
public static ZContext createContext()
```

To execute a Zilk file in a given context, call the method

```
public static ZObject execute(String filename, ZContext context)
  throws ZRuntimeException, ZParseError
```

where `filename` is the name of the file that you want to execute, and `context` is the Zilk context where you want to execute it in. Whatever names and definitions the execution of this file creates will persist in the `context` after the execution has finished. The value of the last evaluated expression in the file is returned from the call as a `ZObject`.

To compile Zilk code that resides in a Java string, the method to call is

```
public static ZFunction compile(String input, ZContext context)
  throws ZParseError
```

that compiles the `input` and gives you back a lambda expression that takes no parameters and is bound to the given `context`, and its body consists of the Zilk code in the `input` string.

To call a `ZFunction` object, use its method `call` with an array of `ZObject`s that you pass it as arguments. If you want to use zero arguments (which you probably usually do with a lambda expressions, although of course their local variable `rest` would capture the extra arguments) you can use the overloaded version of `call` that takes no arguments. You get back a `ZObject` that represents the result.

The class `ZObject` (which is a superclass of `ZFunction` and other classes that represent the runtime objects of Zilk) has a couple of useful methods (in addition to `call`, which works for all Zilk objects instead of only functions) that can be called from other Java programs..

When you call a Zilk function and intend to pass arguments to it, all of these arguments must be Zilk objects of the type `ZObject`. For this purpose, the static method

```
public static ZObject createFromJavaObject(Object javaObj)
```

in the class `ZObject` constructs a Zilk object that corresponds to the given Java object `java`. A Java `String` becomes a Zilk string, and so on. To convert a Zilk object back to a corresponding Java object, use its method

```
public Object asJavaObject()
```

13. Input and output

The Zilk keywords `println` and `print` output the value of the following expression to the standard output with and without a trailing newline, respectively.

Of course, all of the stream, reader and writer and networking classes in the Java packages `java.io`

and `java.net` are readily available in Zilk for more complex I/O. Especially the often-used class `BufferedReader` (which is used to read characters and lines of characters from any input stream or reader) has been extended in *std.zil* to allow an iterator (though only one at the time, since the reader object is itself its own iterator) and the method `eachLine`. Therefore we can write something like

```
br is BufferedReader(FileReader("file.txt"))
Character is java("Character")
br.each({ |x|
  unless Character.isWhitespace(x)
    print x
  endunless
})
```

to output the contents of *file.txt* with all the whitespace characters removed.

Note that the iterator methods of `BufferedReader` close the stream when all of the stream has been read, so further attempts to read from the same stream would result in an error.

For reading character input from `$STDIN`, some simple functions have been defined in *std.zil*. The function `read()` reads and returns the next character (or `nil` if the stream has ended), and the function `readLine()` reads and returns the next line.

14. Troubleshooting

Question: What is wrong with these statements?

```
5.each ({println "Hello"})   # should output five hellos
execute ('test.zil')         # executes a file
```

Answer: You can't put a space between the method name (in this case, `each` and `execute`) and the left parenthesis that starts the argument list. Otherwise you have two separate statements that really do nothing (well, they evaluate to objects which are then immediately discarded).

This is the price that we pay for not requiring statement separators in Zilk. Ruby is admittely much prettier in this respect, but to compensate Zilk can syntactically distinguish between some things that Ruby cannot.

Question: Can I assign to or otherwise modify `this`, `klass`, `super` and `rest`?

Answer: Trying to assign to the first three special variables is a compile-time error. You can assign to `rest` no problem, but when `rest` is the empty list, do not modify its contents, since in this case, the empty list object is shared for efficiency reasons instead of creating it anew each time a function call takes place. (The empty lists are not shared in general.)

Question: Can I extend `$Integer` et al. and construct objects from them?

Answer: Yes, but for the immutable objects such as integers and strings, there is no way to change the contents of the objects once it has been created, not even in the method `initialize`. The object gets created before its constructor is executed, so there is no way to write the constructor of `$Integer` or any of its subclasses in a way that would allow

```
a is $Integer(42)   # construct a new $Integer object
println a           # perhaps ought to output 42 (but will output 0)
```

since the integer object that `a` refers to gets initialized to zero. The constructor of `$List` has been defined so that it builds the initial list from the arguments that it receives:

```
a is $List(42, 'Hello', true)
```

```
println a           # outputs [42 Hello true]
```

Question: I want to use this neat Java class that I have, but it has a method name or a field that is also a Zilk keyword! What can I do?

Answer: No problem. Zilk does not consider identifiers that are inside a compound name to be keywords. So if you always use that unfortunately-named method through a reference, it works as intended. For example, `System.out.println()` means calling the method `println` of object `System.out`. Note that keywords are only ignored inside the compound name, not at the beginning, so `println.foo()` would always be a syntax error. (Just use `this` or `klass` or some dummy reference to create a compound name where needed.)

Question: How does Zilk look for a name that is used in a method?

Answer: To find a (non-global) name, Zilk first looks at the local context of the method. If the name can be found there, return the object associated to it and terminate.

Otherwise, the search starts climbing from the object `this` towards `$Object`. At each level of this climb, if the name is found at the local context of the object, return the object associated to it, otherwise follow the pointer `klass` to the next level. (In each object, `klass` points to the class object that the object was constructed from, and in each class object, `klass` points to the superclass of that class.)

If the name cannot be found that way, the search does the same climb again, but now looks at the *static contexts* of the objects along the way. If this search is also unsuccessful, the search terminates and throws a Zilk exception.

Global names that start with the character `$` are immediately searched for in the global context. If they are not found, they are considered to be `nil` and no exception is thrown even if they don't really exist.

In methods, starting a variable name with `@` and `@@` mean exactly the same thing (down to the tokenization level of the parser) as prefixing the variable name with keywords `this` and `klass`. These special variables exist inside every method, and they are `nil` in functions that do not belong to an object.

Question: If I execute a superclass method for an object constructed from the subclass (either explicitly through `super` or because the method has not been overriden in the subclass that the object was originally constructed from), are `klass` and `super` inside that method relative to object `this` or to the class from which the method comes from?

Answer: The class from which the method came from.

Question: I am writing a delegate class that contains a list of methods that it should forward each call to. Why doesn't this simple method work as intended: the methods in `@callList` do all get called no problem, but they get zero arguments no matter what?

```
def toFunction   # a delegate is called with the normal function syntax
  @callList.each( {|f| f(*rest)} )
enddef
```

Answer: That is because the variable `rest` in the lambda block refers to the rest of the arguments given to the lambda block (and since there are none, `rest` is always the empty list), not to the rest of the arguments that were given to `toFunction`. To get this method to work, write it like this:

```
def toFunction
  tmpRest is rest
  @callList.each({|f| f(*tmpRest)})
enddef
```

Such a class `Delegate` is defined in *std.zil*. The constructor of this class can be given any number of arguments which it turns to the initial list of the methods to forward its own arguments to. The class `Delegate` also has the methods `addMethod |f|` and `removeMethod |f|` for adding new methods into the delegate and removing the existing ones.

Question: How do set a value of a variable if it is not defined yet, but leave it to its old value otherwise?

Answer: Use the `try-rescue` structure:

```
try nonexistent                 # a nonexistent variable
rescue nonexistent is 99
endtry
assert nonexistent eq 99

a is 10
try a                           # an already declared variable
rescue a is 99                  # execution does not get here
endtry
assert a eq 10
```

Question: How do I find out what fields an object has and what methods it responds to?

Answer: Use the attributes `dir` and `locals`. The first returns a list of all names that the object and its `klass` chain contain, and the second returns the list of names in the local context of the object.

```
println 2.locals    # outputs []
println 2.dir       # outputs [accumulate,append,append!,...]
```

Note that currently these two won't find the builtin attributes such as `int` or even `dir` or `locals` themselves. I will hopefully fix this in the future once I rethink the Zilk object model and its implementation more carefully.

Question: Is Zilk thread-safe, say if I use `java.lang.Thread` to launch multiple Zilk threads?

Answer: I have done my best attempt to make Zilk thread-safe and have actually been quite conservative in mutual exclusion, but I can't fully guarantee thread-safety at this point. The Zilk interpreted is bound to have lots of bugs remaining.

By the way, every Zilk object is itself a Java `Runnable`, so you can give a Zilk function as constructor parameter to `Thread` to be executed in the background thread. However, note that this executes the function in a new context, so you probably want this function to be a lambda expression or bound to some object.

Question: If I extend a Java class in Zilk, for example `javax.swing.JPanel` so that I redefine the method `paint(Graphics g)` in my Zilk subclass of `JPanel`, does the Java class polymorphically recognize this overriding method to draw my panel on screen?

Answer: No. Zilk simulates classes on a different level that is one level of abstraction above the Java's mechanism. (Zilk actually implements extending a Java class using composition instead of “real” inheritance.)

15. Global variables and options

The header file *std.zil* and the Zilk initialization process define some global variables.

- `$Object`, `$Integer`, `$Float`, `$Char`, `$Boolean`, `$List`, `$String`, `$Range`, `$Function` and `$JavaClass` are the class types of the built-in Zilk objects.

- `$ARGS` contains the list of command line arguments given to the program.
- `$IMPORT` is the list of Java packages where classes are searched for.
- `$LINE` is the line number where the program execution is currently at. (Currently, the line numbering only works for parse errors: for runtime errors, the reported line number refers to the line where the offending call originated. I will fix this in a later version.)
- `$APPLET` is the Java applet object if the program is currently being run as an applet, and `nil` otherwise.
- `$LISTSEPARATOR`, `$LISTLEFTBRACKET` and `$LISTRIGHTBRACKET` determine how lists are output. These are initially comma and the square brackets. They are evaluated once each time before a list is output.
- `$STDIN` and `$STDOUT` are the `InputStream` and `OutputStream` objects corresponding to standard input and output.
- `$EXTENSION` is the file extension for Zilk programs, initially set to `.zil`.

Names that start with two underscore characters are internal to Zilk and therefore should not be used in your own programs.

The following standard Java classes are defined in *std.zil* to be Zilk classes with the same names: `File` (unless the program is running as an applet), `System`, `InputStreamReader`, `BufferedReader` and `PrintWriter`.

The main method of Zilk resides in the file *Zilk.java*. It recognizes the following command line options before the filename and the command line arguments that follow it:

- `-c`

  Just parse the code and check its syntax, but do not actually execute the program.

- `-e`

  Execute the following code directly. Example: `Zilk -e "println 'Hello world'"`

- `-t`

  Time the parsing and execution, output the elapsed time at the end of execution.

- `-v`

  Turn verbose mode on. (This doesn't do anything yet.)

16. About the Zilk class hierarchy

The Zilk classes reside in the package `org.zilk`. The Zilk parser and execution environment consists of an inheritance hierarchy of classes plus a few auxiliary classes that reside outside this hierarchy. The superclass of this main hierarchy is the abstract class `ZExpression`, whose subclasses correspond to different nodes in the syntax tree. Each subclass `X` of `ZExpression` must implement the method

```
abstract public ZObject evaluate(ZContext context) throws ZRuntimeError
```

to appropriately evaluate the syntax tree whose root is an object of the type from `X`, using the context given in the parameter `context`.

The class `ZObject` and its subclasses (`ZList`, `ZInt` etc.) correspond to the objects that come to existence during the execution of the Zilk program. The class `ZObject` is also a subclass of

ZExpression, so that some objects constructed from subclasses of ZObject (namely, the constant objects from the classes ZInt, ZFloat, ZChar and ZBoolean) can serve as leaf nodes in the syntax tree. (Admittedly this is not very pure, but does no harm.) In addition to the method evaluate, the class ZObject contains a bunch of methods that are needed to do the computations at runtime.

The class ZParser takes care of parsing a program string into a syntax tree of the type ZExpression. It is fully written by hand (not the best decision, as I learned during the debugging phase) and uses the straightforward recursive descent parsing algorithm. The class ZTokenizer does the lexical analysis and reads and returns one token (of the type ZToken) to the parser when asked.

The important method of ZParser is

```
public ZExpression parse(Reader input, boolean executeImmediately,ZContext context)
throws ZParseError, ZRuntimeError
```

which reads the input and constructs the syntax tree for it. The parameter executeImmediately determines if the code should be executed as it is read one statement at the time.

The class ZContext encapsulates the context in which code tree is executed. The static field ZContext.GLOBALS is used to store the global context, which is the default.

The exception classes ZParseError and ZRuntimeError correspond to errors that can occur during parsing and evaluation. As of this writing,

For executing Zilk code as an application, the main method of Zilk resides in the class Zilk. This method reads the command line parameters and executes the Zilk file. The main class Zilk also contains the very important method

```
public static void initializeZilkEnvironment()
```

which must be executed before any Zilk code can be parsed or run (the ZParser method parse makes sure of this), as it creates the Zilk objects for the class $Object and its subclasses and then executes the header file *std.zil*.

The class ZRuntimeException is the superclass of exceptions that can be thrown during the execution of a Zilk program. It has three subclasses. ZException is used for exceptions that are supposed to be caught and handled by the Zilk program itself. ZRuntimeError is used for errors that the Zilk program is not supposed to handle itself (although these should be changed to be ZException objects wherever possible). ZControl is a class that is used to implement Zilk statements break and continue.

17. To do

Use caching to speed up both Java method lookup and variable lookup inside a context. Optimize list operations and concatenation.

Make line numbering work correctly also for runtime errors. That is, make each ZExpression object remember what file and line it came from.