NAME

coroutine, yield, yieldto − Create and produce values from coroutines

SYNOPSIS

coroutine *name command* ?*arg...*?
yield ?*value*?
yieldto *command* ?*arg...*?
name ?*value...*?

DESCRIPTION

The **coroutine** command creates a new coroutine context (with associated command) named *name* and executes that context by calling *command*, passing in the other remaining arguments without further interpretation. Once *command* returns normally or with an exception (e.g., an error) the coroutine context *name* is deleted.

Within the context, values may be generated as results by using the **yield** command; if no *value* is supplied, the empty string is used. When that is called, the context will suspend execution and the **coroutine** command will return the argument to **yield**. The execution of the context can then be resumed by calling the context command, optionally passing in the *single* value to use as the result of the **yield** call that caused the context to be suspended. If the coroutine context never yields and instead returns conventionally, the result of the **coroutine** command will be the result of the evaluation of the context.

The coroutine may also suspend its execution by use of the **yieldto** command, which instead of returning, cedes execution to some command called *command* (resolved in the context of the coroutine) and to which *any number* of arguments may be passed. Since every coroutine has a context command, **yieldto** can be used to transfer control directly from one coroutine to another (this is only advisable if the two coroutines are expecting this to happen) but *any* command may be the target. If a coroutine is suspended by this mechanism, the coroutine processing can be resumed by calling the context command optionally passing in an arbitrary number of arguments. The return value of the **yieldto** call will be the list of arguments passed to the context command; it is up to the caller to decide what to do with those values.

The recommended way of writing a version of **yield** that allows resumption with multiple arguments is by using **yieldto** and the **return** command, like this:

```
proc yieldm {value} {
    yieldto return -level 0 $value
}
```

The coroutine can also be deleted by destroying the command *name*, and the name of the current coroutine can be retrieved by using **info coroutine**. If there are deletion traces on variables in the coroutine's implementation, they will fire at the point when the coroutine is explicitly deleted (or, naturally, if the command returns conventionally).

At the point when *command* is called, the current namespace will be the global namespace and there will be no stack frames above it (in the sense of **upvar** and **uplevel**). However, which command to call will be determined in the namespace that the **coroutine** command was called from.

EXAMPLES

This example shows a coroutine that will produce an infinite sequence of even values, and a loop that consumes the first ten of them.

```
proc allNumbers {} {
    yield
    set i 0
    while 1 {
        yield $i
        incr i 2
    }
```

```
}
coroutine nextNumber allNumbers
for {set i 0} {$i < 10} {incr i} {
   puts "received [nextNumber]"
}
rename nextNumber {}
```

In this example, the coroutine acts to add up the arguments passed to it.

```
coroutine accumulator apply {{} {
   set x 0
   while 1 {
      incr x [yield $x]
   }
}}
for {set i 0} {$i < 10} {incr i} {
   puts "$i -> [accumulator $i]"
}
```

This example demonstrates the use of coroutines to implement the classic Sieve of Eratosthenes algorithm for finding prime numbers. Note the creation of coroutines inside a coroutine.

```
proc filterByFactor {source n} {
   yield [info coroutine]
   while 1 {
      set x [$source]
      if {$x % $n} {
         yield $x
      }
   }
}
coroutine allNumbers apply {{} {while 1 {yield [incr x]}}}
coroutine eratosthenes apply {c {
   yield
   while 1 {
      set n [$c]
      yield $n
      set c [coroutine prime$n filterByFactor $c $n]
   }
}} allNumbers
for {set i 1} {$i <= 20} {incr i} {
   puts "prime#$i = [eratosthenes]"
}
```

This example shows how a value can be passed around a group of three coroutines that yield to each other:

```
proc juggler {name target {value ""}} {
   if {$value eq ""} {
      set value [yield [info coroutine]]
   }
   while {$value ne ""} {
      puts "$name : $value"
      set value [string range $value 0 end-1]
      lassign [yieldto $target $value] value
   }
}
```

```
coroutine j1 juggler Larry [
   coroutine j2 juggler Curly [
      coroutine j3 juggler Moe j1]] "Nyuck!Nyuck!Nyuck!"
```

DETAILED SEMANTICS

This example demonstrates that coroutines start from the global namespace, and that *command* resolution happens before the coroutine stack is created.

```
proc report {where level} {
   # Where was the caller called from?
   set ns [uplevel 2 {namespace current}]
   yield "made $where $level context=$ns name=[info coroutine]"
}
proc example {} {
   report outer [info level]
}
namespace eval demo {
   proc example {} {
      report inner [info level]
   }
   proc makeExample {} {
      puts "making from [info level]"
      puts [coroutine coroEg example]
   }
   makeExample
}
```

Which produces the output below. In particular, we can see that stack manipulation has occurred (comparing the levels from the first and second line) and that the parent level in the coroutine is the global namespace. We can also see that coroutine names are local to the current namespace if not qualified, and that coroutines may yield at depth (e.g., in called procedures).

```
making from 2
made inner 1 context=:: name=::demo::coroEg
```

SEE ALSO

apply(n), info(n), proc(n), return(n)

KEYWORDS

coroutine, generator