Generator Expressions

Note: If you are not familiar with the generator concept, this Generator Breakdown page should be read before continuning.

Generator expressions are defined through the pattern:

```
<expression> for <declarations> in
<iterator> [if|unless <condition>]
```

Generator expressions can be used as return values:

```
def GetCompletedTasks():
      return t for t in _tasks if
t.IsCompleted
```

Generator expressions can be stored in variables:

```
oddNumbers = i for i in range(10) if i % 2
```

Generator expressions can be used as arguments to functions:

```
print(join(i*2 for i in range(10) if 0 == i
% 2))
```

In all cases the evaluation of each inner expression happens only **on demand** as the generator is consumed by a for in loop.

Generator expressions capture their enclosing environment (like closures do) and thus are able to affect it by updating variables:

```
i = 0
a = (++i)*j for j in range(3)
print(join(a)) # prints "0 2 6"
print(i) # prints 3
```

As well as being affected by changes to captured variables:

```
i = 1
a = range(3)
generator = i*j for j in a
print(join(generator)) # prints "0 1 2"

i = 2
a = range(5)
print(join(generator)) # prints "0 2 4 6 8"
```

Remarks

boo's variable capturing behavior differs in behavior from python's in a subtle but I think good way:

```
import System

functions = Math.Sin, Math.Cos

a = []
for f in functions:
   a.Add(f(value) for value in range(3))

for iterator in a:
   print(join(iterator))
```

This program properly prints the sins followed by the cosines of 0, 1, 2 because **for** controlled variable references (such as f) in boo generator expressions as well as in closures are **bound early**.

If you don't know what the python behavior would be check this document.