Extending FUNC with Generators

Isaac Getto + David Abrahams

What are Generators?

```
def evens(n):
    i = 0
    while i < n:
        if i % 2 == 0:
            yield i
        i += 1


evens_list = [e for e in evens(10)]
```

Lazy Evaluation

With generator:

```
def evens(n):
    i = 0
    while i < n:
        if i % 2 == 0:
            yield i
        i += 1

# print 5 even numbers
i = 0
for e in evens(1000):
    print e
    i += 1
    if i == 5:
        return
```

With list

```
def evens(n):
    i = 0
    A = []
    while i < n:
        if i % 2 == 0:
            A.append(i)
        i += 1
    return A

# print 5 even numbers
i = 0
for e in evens(1000):
    print e
    i += 1
    if i == 5:
        return
```

Doesn’t need end condition

With generator:

```
def evens():
    i = 0
    while True:
        if i % 2 == 0:
            yield i
        i += 1

# print 5 even numbers
i = 0
for e in evens():
    print e
    i += 1
    if i == 5:
        return
```

With list

```
def evens():
    i = 0
    A = []
    while True:
        if i % 2 == 0:
            A.append(i)
        i += 1
    return A

# print 5 even numbers
i = 0
for e in evens(1000):
    print e
    i += 1
    if i == 5:
        return
```

Goal syntax

Generator expression:

```
( gen (x) expr)


( gen (x) (yield 1 2))


(let (mygen (( gen (x) (yield 1 2))))
  some_let_expr
)

(let (mygen (( gen (x) (yield 1 2))))
  (+ (mygen 1) (mygen _)
)
```

Yield expression:

```
( yield first next )
```

How to loop?

Generator expression:

```
(let
 ((upgen
  (gen (x)
   (let (( up
    (fun up (x)
    (yield
     x
     (up (+ x 1))
   )))) (up x)))))
 (+ (upgen 1) (upgen 1))
)
```

Declare the generator

Recursive function yields x, then calls itself with x+1

Let expression is a call to the recursive function

Call generator with initial argument 1

Stack Based Evaluation

We use a stack and a value variable

(1)

ELiteral(1)

Currently Evaluating:

Stack:

1. ELiteral(1)

Value: None

Stack Based Evaluation

We use a stack and a value variable

(1)

ELiteral(1)

Currently Evaluating: ELiteral(1)

Stack:

Value: None

Stack Based Evaluation

We use a stack and a value variable

(1)

ELiteral(1)

Currently Evaluating:

Stack:

Value: VInteger(1)

Stack Based Evaluation

We use a stack and a value variable

(+ 1 2)

EPlus(ELiteral(1), ELiteral(2))

Currently Evaluating:

Stack:

1. EPlus(ELiteral(1), ELiteral(2))

Value: None

Stack Based Evaluation

We use a stack and a value variable

(+ 1 2)

EPlus(ELiteral(1), ELiteral(2))

Currently Evaluating:
EPlus(ELiteral(1), ELiteral(2))

Stack:

Value: None

Stack Based Evaluation

We use a stack and a value variable

(+ 1 2)

EPlus(ELiteral(1), ELiteral(2))

Currently Evaluating:

Stack:

1. ELiteral(1)
2. EPlus2(ELiteral(2))

Value: None

Stack Based Evaluation

We use a stack and a value variable

(+ 1 2)

EPlus(ELiteral(1), ELiteral(2))

Currently Evaluating: ELiteral(1)

Stack:

1. EPlus2(ELiteral(2))

Value: None

Stack Based Evaluation

We use a stack and a value variable

(+ 1 2)

EPlus(ELiteral(1), ELiteral(2))

Currently Evaluating:

Stack:

1. EPlus2(ELiteral(2))

Value: VInteger(1)

Stack Based Evaluation

We use a stack and a value variable

(+ 1 2)

EPlus(ELiteral(1), ELiteral(2))

Currently Evaluating:
EPlus2(ELiteral(2))

Stack:

1.

Value: VInteger(1)

Stack Based Evaluation

We use a stack and a value variable

(+ 1 2)

EPlus(ELiteral(1), ELiteral(2))

Currently Evaluating:

Stack:

1. ELiteral(2)
2. EPlus3(VInteger(1))

Value:

Stack Based Evaluation

We use a stack and a value variable

(+ 1 2)

EPlus(ELiteral(1), ELiteral(2))

Currently Evaluating:

Stack:

1. EPlus3(VInteger(1))

Value: VInteger(2)

Stack Based Evaluation

We use a stack and a value variable

(+ 1 2)

EPlus(ELiteral(1), ELiteral(2))

Currently Evaluating:

Stack:

Value: VInteger(3)

Stack Based Evaluation

We use a stack and a value variable

(if false 1 2)

EIf(ELiteral(false), ELiteral(1), ELiteral(2))

Currently Evaluating:

Stack:

1. EIf(ELiteral(false), ELiteral(1), ELiteral(2))

Value:

Stack Based Evaluation

We use a stack and a value variable

(if false 1 2)

EIf(ELiteral(false), ELiteral(1), ELiteral(2))

Currently Evaluating:
EIf(ELiteral(false), ELiteral(1), ELiteral(2))

Stack:

Value:

Stack Based Evaluation

We use a stack and a value variable

(if false 1 2)

EIf(ELiteral(false), ELiteral(1), ELiteral(2))

Currently Evaluating:

Stack:

1. ELiteral(false)
2. EIf2(ELiteral(1), ELiteral(2))

Value:

Stack Based Evaluation

We use a stack and a value variable

(if false 1 2)

EIf(ELiteral(false), ELiteral(1), ELiteral(2))

Currently Evaluating:

Stack:

1. EIf2(ELiteral(1), ELiteral(2))

Value: false

Stack Based Evaluation

We use a stack and a value variable

(if false 1 2)

EIf(ELiteral(false), ELiteral(1), ELiteral(2))

Currently Evaluating: EIf2(ELiteral(1), ELiteral(2))

Stack:

Value: false

Stack Based Evaluation

We use a stack and a value variable

(if false 1 2)

EIf(ELiteral(false), ELiteral(1), ELiteral(2))

Currently Evaluating:

Stack:

1. ELiteral(2)

Value:

Stack Based Evaluation

We use a stack and a value variable

(if false 1 2)

EIf(ELiteral(false), ELiteral(1), ELiteral(2))

Currently Evaluating:

Stack:

Value: 2

Stack Evaluation Demo

(+ 1 2)

=> **3**

Generators Demo

$f(x) = 2 * x$

```
(let ((
  double_gen
  (gen (x) (
    let ((
      double
      (fun double (x) (yield x (double (* 2 x))))))
        (double x)
    )
  )
))
(+ (double_gen 2) (double_gen 2)))
```

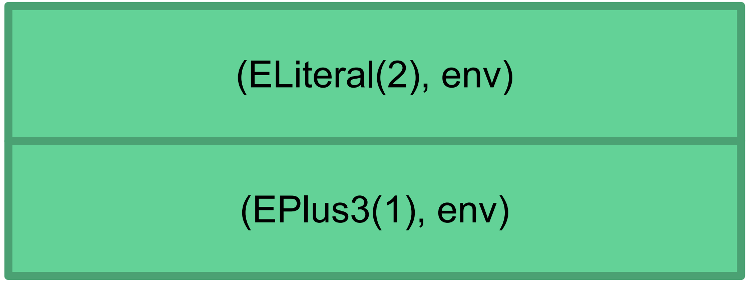
=> 2 + 4 = **6**

How it works

```
var stack = new Stack[(Exp, Env)]
```

```
until the stack is empty:
    take the expression off the top
    evaluate
    repeat
```

(+ 1 2)



(ELiteral(2), env)

(EPlus3(1), env)

How it works

```
while (!stack.isEmpty) {
  val (e, env) = stack.pop()

  e match {
    case ELiteral(v) => value = v
    case EId(s) => value = env.lookup(s)
    case EPlus(e1, e2) => {
      stack.push((new EPlus2(e2), env))
      stack.push((e1, env))
    }
    case EPlus2(e2) => {
      stack.push((new EPlus3(value), env))
      stack.push((e2, env))
    }
    case EPlus3(v) => {
      value = new VInteger(v.getInt() + value.getInt())
    }

    . . .
}
```

How it works

load

save

```
    e match {
      . . .
      case EApply3(vf) => {
        vf match {
          . . .
          case VGen(param, body, env) => {
            if (nextExp == null) {
              var e = env.push(param, value)
              stack.push((body, e))
            } else {
              stack.push((nextExp, nextEnv))
            }
          }
        }
      }
      case EYield(e1, e2) => {
        stack.push((e1, env))
        nextExp = e2
        nextEnv = env
      }
    }
}
```

Next Steps - Simplify Syntax

```
(let ((
 double_gen
 (gen (x) (
  let ((
   double
   (fun double (x) (yield x (double (* 2 x))))))
    (double x)
  )
 )
))
(+ (double_gen 1) (double_gen 1)))
```

```
doubles = list(2 * n for n in range(50))
```

Next Steps - No Parameters

```
(let ((
 double_gen
 (gen (x) (
  let ((
   double
   (fun double (x) (yield x (double (* x 2))))))
    (double x)
  )
 )
))
(+ (double_gen 1) (double_gen 1)))
```

useless

Next Steps - Less Restrictive & Multiple Generators

- Make yield more general

- Right now we can only create 1 generator at time