Assignment 2 Review

Supporting Control Flow

- Extend your AST interpreter to support return. Two general ways of doing it.

Longjmp / Setjmp

- (In C) Longjmp/Setjmp : Set a jump point upon function entry, and when you execute the return you longjmp to the set jump point.
- (In Java/ML) Throw exception.
- (In Scheme) Request continuation.

Augmenting eval's return value

- Change eval to return:
  1. Result(value)
  2. Return(value)
- Now in the interpreter whenever you call eval on a subexpression, you check if its a Result or a Return.
- If it is a Result than proceed as before.
- If it is a Return than directly return the wrapped value.
- Haskell weenies can wrap all this up in a Monad.

Not All Languages are Equal

- For manipulating control flow: Scheme continuations is the most powerful construct out there.

Object Cloning

- Semantics of Cloning:
  - Shallow clone all the fields.
    - Parent is either cloned or not cloned. Good reasons for either.
- Deep cloning:
  - Must detail how cycles in the object graph are handled.

Object Cloning

- Cannot be written as a pure Feeny function. Requires interpreter support.
- Make a copy of the Environment object and copy all the pointers over.

Object Cloning: Negative Consequences

- The most powerful language features of all are what invariants are guaranteed by your language.

Object Cloning: Negative Consequences

- Goal: Make a pair function:

```
val p = pair()
p.get-x()
p.get-y()
p.set-x(4)
p.set-y(4)
```

Object Cloning: Negative Consequences

```
defn pair () :
   object :
      var a = array(2, 0)
      method get-x () :
         this.a[0]
      method get-y () :
         this.a[1]
   method set-x (x) :
      this.a[0] = x
   method set-y (y) :
      this.a[1] = y
```

Object Cloning: Negative Consequences

```
defn pair () :
   object :
      var a = array(2, 0)
      method get-x () :
         this.a[0]
      method get-y () :
         this.a[1]
   method set-x (x) :
      this.a[0] = x
   method set-y (y) :
      this.a[1] = y
```

```
defn pair () :
   object :
      var x = 0
      var y = 0
      method get-x () :
         this.x
      method get-y () :
         this.y
   method set-x (x) :
      this.x = x
   method set-y (y) :
      this.y = y
```

Object Cloning: Negative Consequences

```
defn pair () :
   object :
      var a = array(2, 0)
      method get-x () :
         this.a[0]
      method get-y () :
         this.a[1]
   method set-x (x) :
      this.a[0] = x
   method set-y (y) :
      this.a[1] = y
```

```
defn pair () :
   object :
      var x = 0
      var y = 0
      method get-x () :
         this.x
      method get-y () :
         this.y
   method set-x (x) :
      this.x = x
   method set-y (y) :
      this.y = y
```

The program is not invariant to this transformation!!!

Invariants are Features

- Feeny programs are memory safe.
- No segmentation faults ever!

Invariants are Features

- You cannot write malicious programs using Feeny.
- Can download any arbitrary Feeny program from the internet and start running it.

Invariants are Features

```
while i < 10 :
    do stuff …

do more stuff …
```

i >= 10 is guaranteed during this code!

Bytecode Interpreter / Compiler : Value Representation

Bytecode Interpreter: Value Representation