A Lambda Interpreter

- We studied *Lambda Calculus* as a foundation for functional programs
- Now we implement the operational semantics defined by the λ-calculus in the form of an *interpreter*
- An interpreter for a programming language is a function that, when applied to a term, performs the actions required to evaluate the expression:

$$interpreter : Term \rightarrow Value$$

- Lambda Interpreter: β-reduction + Evaluation Strategy
  - Call-By-Name
  - Call-By-Value
  - Call-By-Need

Representation of Lambda Terms

- Syntax of Lambda-Calculus:

$$\begin{array}{ll} \text{Names} & x \in \mathcal{N} \\ \text{Terms} & E \ ::= \ x \mid \lambda x.E \mid E\,E' \end{array}$$

- Names are implemented with strings
- Lambda terms are represented as trees:

```
val Tree = {
    def Name(x)      = { def match(v) = v.Name(x)      }
    def Lambda(x, t) = { def match(v) = v.Lambda(x, t) }
    def Apply(t, t') = { def match(v) = v.Apply(t, t') }
}
```

Evaluation of Lambda Terms

- Semantics of λ-calculus:

$$(\beta) \quad (\lambda x.E)E' \quad \to \quad [E'/x]\; E$$

- Definition of substitutions $[E/x]$:

$$
\begin{array}{lcll}
[E/x]\; x & = & E & \\
[E/x]\; y & = & y & (x \neq y) \\
[E/x]\; \lambda x.E' & = & \lambda x.E' & \\
[E/x]\; \lambda y.E' & = & \lambda y.[E/x]E' & (x \neq y, y \notin \mathrm{fn}(E)) \\
[E/x]\; (E'\; E'') & = & ([E/x]E')\; ([E/x]E'') &
\end{array}
$$

Implementing Substitutions

We implement a substitution as a visitor:

```
def subst(y, t) = {
    def Name(x)      = if (x == y) t else Tree.Name(x)
    def Lambda(x,t') = {
        val z = newvar
        Tree.Lambda(z,t'.match(subst(x,Tree.Name(z)))
                        .match(subst(y,t)))
    }
    def Apply(t1,t2) = Tree.Apply(t1.match(subst(y,t)),
                                  t2.match(subst(y,t)))
}
```

The function `newvar` returns a fresh variable when called.

Substitutions

Example usage:

```
> val t = Tree.Lambda("x",Tree.Apply(Tree.Apply(Tree.Name("x"),
                                  Tree.Name("y")), Tree.Name("y")))
'val t = <record id=2, adr=1, type=(match)>'
> t.match(toString)
"(x|((x y) y))"
> t.match(subst("y", Tree.Lambda("z", Tree.Name("z"))))
   .match(toString)
"(x|((x (z|z)) (z|z)))"
```

Implementing β-Reduction

- Here's an implementation of the β-reduction rule with call-by-name evaluation strategy:

```
val reduce = {
    def Name(x) = Tree.Name(x)
    def Lambda(x, t) = Tree.Lambda(x, t)
    def Apply(t, t') = {
        t.match(reduce).match {
          def Name(x) = Tree.Apply(Tree.Name(x), t')
          def Lambda(x, b) = b.match(subst(x, t')).match(reduce)
          def Apply(s, t) = Tree.Apply(Tree.Apply(s, t), t')
        }
    }
}
```

A Call-By-Value Lambda Interpreter

- What changes if we are using the call-by-value evaluation strategy?

```
val reduce = {
    def Name(x) = Tree.Name(x)
    def Lambda(x, t) = Tree.Lambda(x, t)
    def Apply(t, t') = {
        t.match(reduce).match {
          def Name(x) = Tree.Apply(Tree.Name(x), t')
          def Lambda(x, b) = b.match(subst(x, t'.match(reduce)))
                              .match(reduce)
          def Apply(s, t) = Tree.Apply(Tree.Apply(s, t), t')
        }
    }
}
```

- This is more efficient as call-by-name, because the argument is evaluated only once

Environments

- Implementing β-reduction directly with substitutions has several disadvantages:
  - α-renaming is needed, and
  - it is very inefficient
- Solution: Environments
  - An environment is a finite mapping from names to values
  - Instead of substituting a parameter name with the actual argument in the β-reduction rule, a mapping $name \rightarrow argument$ is added to the environment and the body of the λ-abstraction is evaluated using this extended environment
  - A new mapping for a name shadows an existing mapping for the same name

An Interpreter with Environments

```
def eval(env) = {
  def Name(x) = env(x)
  def Lambda(x, t) = Tree.Lambda(x, t)
  def Apply(t, t’) = {
    t.match(eval(env)).match {
      def Name(x) = Tree.Name(x)
      def Lambda(x, b) = b.match(
                           eval(enter(x, t’.match(eval(env)), env)))
      def Apply(t, t’) = Tree.Apply(t, t’)
    }
  }
}
```

Is this implementation correct?

Just think about evaluating $(\lambda x.\lambda y.x)\ a\ b$

We need Closures

- When a function is applied, it's body is evaluated in an environment that binds the formal parameter to the argument of the application (β-reduction)
- If the body reduces to a function (λ-abstraction), it has to retain it's bindings of free variables. It must be a closed entity, independent of the environment in which it is used: *Closure*
- Consequence: evaluation of a λ-abstraction yields a closure (which binds all free variables within the abstraction)
- A closure is represented by a tupel: $(\lambda - abstraction, environment)$

A Call-By-Value Interpreter with Environments

- From now on we consider global free variables as illegal; i.e. evaluating the term $(\lambda x.x)a$ yields an error, since a is not defined
- We distinguish between terms and values: the interpreter maps terms to values; an environment maps names to values.

```
val Value = {
  def Closure(x,t,env) = { def match(v) = v.Closure(x,t,env) }
}
```

- Let's write an `eval` function that implements the interpreter:

```
def eval(env) = {
  def Name(x) = env(x)
  def Lambda(x, t) = Value.Closure(x, t, env)
  def Apply(t, t') = t.match(eval(env))
                      .match(apply(t'.match(eval(env))))
}
```

A Call-By-Value Interpreter with Environments

Here's the missing `apply` function:

```
def apply(arg) = {
    def Closure(x,t,env) = t.match(eval(enter(x,arg,env)))
}
```

Your Task

- Try to find a good representation for environments
- Write an interpreter for an extension of the call-by-value λ-calculus in Funnel
- Here's the abstract syntax:

Names	x
Numbers	i
Terms	$E \ ::= \ x \mid i \mid \lambda x.E \mid E\,E'$

- In addition to the pure λ-calculus your interpreter should support integer numbers and the initial environment should offer basic operations $plus$, $minus$, $times$, div, etc. on numbers