Environments

Past three weeks
How to use essential language constructs?

- Data Types
- Recursion
- Higher-Order Functions

Next two weeks
How to implement language constructs?

- Local variables and scope
- Environments and Closures
- Type Inference

Interpreter
How do we represent and evaluate a program?
Roadmap: The Nano Language

Features of Nano:

1. Arithmetic
2. Variables
3. Let-bindings
4. Functions
5. Recursion
1. **Nano: Arithmetic**

A “grammar” of arithmetic expressions:
e ::= n
    | e1 + e2
    | e1 - e2
    | e1 * e2

<table>
<thead>
<tr>
<th>Expressions</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>==&gt; 4</td>
</tr>
<tr>
<td>4 + 12</td>
<td>==&gt; 16</td>
</tr>
<tr>
<td>(4+12) - 5</td>
<td>==&gt; 11</td>
</tr>
</tbody>
</table>

Representing Arithmetic Expressions and Values
Lets represent arithmetic expressions as type

```haskell
data Expr
  = ENum Int       -- ^ n
  | EAdd Expr Expr -- ^ e1 + e2
  | ESub Expr Expr -- ^ e1 - e2
  | EMul Expr Expr  -- ^ e1 * e2
```

Lets represent arithmetic values as a type

```haskell
type Value = Int
```
Evaluating Arithmetic Expressions

We can now write a Haskell function to evaluate an expression:

```haskell
eval :: Expr -> Value
eval (ENum n) = n
eval (EAdd e1 e2) = eval e1 + eval e2
eval (ESub e1 e2) = eval e1 - eval e2
eval (EMul e1 e2) = eval e1 * eval e2
```
Alternative representation

Lets pull the operators into a separate type

```haskell
data Binop = Add                  -- ^ `+`
  | Sub                   -- ^ `-`
  | Mul                   -- ^ `*`

data Expr = ENum Int            -- ^ n
  | EBin Binop Expr Expr  -- ^ e1 `op` e2
```
**QUIZ**

Evaluator for alternative representation

```haskell
eval :: Expr -> Value
eval (ENum n) = n
eval (EBin op e1 e2) = evalOp op (eval e1) (eval e2)
```

What is a suitable type for `evalOp`?

```haskell
{- 1 -} evalOp :: BinOp -> Value
{- 2 -} evalOp :: BinOp -> Value -> Value -> Value
{- 3 -} evalOp :: BinOp -> Expr -> Expr -> Value
{- 4 -} evalOp :: BinOp -> Expr -> Expr -> Expr
{- 5 -} evalOp :: BinOp -> Expr -> Value
```
The Nano Language

Features of Nano:

1. Arithmetic [done]
2. Variables
3. Let-bindings
4. Functions
5. Recursion

[Diagram showing relationships between Expression, Value, and Type with run-time and compile-time arrows]
2. Nano: Variables

Let’s add variables and let bindings!

\[
es ::= \text{n} \quad \text{-- \textit{OLD}}
\]
\[
| \ e_1 + e_2
\]
\[
| \ e_1 - e_2
\]
\[
| \ e_1 \times e_2
\]
\[
| \ x \quad \text{-- \textit{NEW}}
\]

-- \textit{variables}

Let’s extend our datatype
type Id = String

data Expr
  = ENum Int          -- OLD
  | EBin Binop Expr Expr
    -- NEW
  | EVar Id            -- variables

**QUIZ**

What should the following expression evaluate to?

\[ x + 1 \]

(A) 0

(B) 1

(C) Error
Environment

An expression is evaluated in an environment

- A phone book which maps variables to values

[ "x" := 0, "y" := 12, ... ] → lookup

A type for environments

type Env = [(Id, Value)]

Evaluation in an Environment

We write
(eval env expr) ==> value

to mean

When expr is evaluated in environment env the result is value **

That is, when we have variables, we modify our evaluator to take an input environment env in which expr must be evaluated.

`eval :: Env -> Expr -> Value`

`eval env expr = ... value-of-expr-in-env...`

First, let's update the evaluator for the arithmetic cases ENum and EBin

`eval :: Env -> Expr -> Value`

`eval env (ENum n) = ???`

`eval env (EBin op e1 e2) = ???`

**QUIZ**

What is a suitable value such that
eval [ "x" := 0, "y" := 12, ...] (x + 1) ==> ?value

(A) 0
(B) 1
(C) Error

---

**QUIZ**

What is a suitable env such that

```
  eval env (x + 1) ==> 10
```

(A) []
(B) [x := 0, y := 9]
(C) \[x := 9, \ y := 0\]

(D) \[x := 9, \ y := 10, \ z := 666\]

(E) \[y := 10, \ z := 666, \ x := 9\]

**Evaluating Variables**

Using the above intuition, let's update our evaluator to handle variables i.e. the EVar case:

```java
eval env (EVar x) = ???
```

Let's confirm that our `eval` is ok!
envA = []
envB = ["x" := 0 , "y" := 9]
envC = ["x" := 9 , "y" := 0]
envD = ["x" := 9 , "y" := 10 , "z" := 666]
envE = ["y" := 10 , "z" := 666 , "x" := 9 ]

-- >>> eval envA (EBin Add (EVar "x") (ENum 1))
-- >>> eval envB (EBin Add (EVar "x") (ENum 1))
-- >>> eval envC (EBin Add (EVar "x") (ENum 1))
-- >>> eval envD (EBin Add (EVar "x") (ENum 1))
-- >>> eval envE (EBin Add (EVar "x") (ENum 1))

The Nano Language

Features of Nano:

1. Arithmetic expressions [done]
2. Variables [done]
3. Let-bindings
4. Functions
5. Recursion
2. Nano: Variables

Let’s add variables and `let` bindings!
\[ e ::= n \quad \text{-- OLD} \\
| e_1 + e_2 \\
| e_1 - e_2 \\
| e_1 \times e_2 \\
| x \\
| \text{let } x = e_1 \text{ in } e_2 \quad \text{-- NEW} \]

Let's extend our datatype

\text{type} \ Id = \text{String} \\

\text{data} \ Expr \\
= \text{ENum Int} \quad \text{-- OLD} \\
| \text{EBin Binop Expr Expr} \\
| \text{EVar Id} \\
| \text{ELet Id Expr Expr} \quad \text{-- NEW} \\
\quad \text{let } x = e_1 \text{ in } e_2 \\

How should we extend \text{eval}?

\textbf{QUIZ}

What \textit{should} the following expression evaluate to?
(A) Error

(B) 1

(C) 0

**QUIZ**

What *should* the following expression evaluate to?
let x = 0
in
  let y = 100
in
  x + y

(A) Error
(B) 0
(C) 1
(D) 100
(E) 101
What *should* the following expression evaluate to?

```plaintext
let x = 0
in
  let x = 100
  in
    x + 1
```

(A) Error

(B) 100

(C) 101
What *should* the following expression evaluate to?

```plaintext
let x = 0
in
  (let x = 100 in
   in
     x + 1
   )
 +
 x

(A) Error
(B) 1
(C) 101
(D) 102
(E) 2
```
**Principle: Static/Lexical Scoping**

Every variable *use* gets its value from a unique *definition*:

- “Nearest” *let* -binder in program *text*

“Static” means you can tell *without running the program*

Great for readability and debugging

1. Define *local* variables
2. Be sure *where* each variable got its value

Don’t have to scratch head to figure where a variable got “assigned”

How to *implement* static scoping?
Quiz

Lets re-evaluate the quizzes!

```
-- env

let x = 0
in
  -- ?? what env to use for `x + 1`?
  x + 1
```

(A) env

(B) []

(C) [ ("x" := 0) ]

(D) ("x" := 0) : env

(E) env ++ ["x" := 0]
let $x = 0$
in

let $y = 100$
in

-- ?? what env to use for `x + y`?

$x + y$

(A) ("x" := 0) : env

(B) ("y" := 100) : env

(C) ("y" := 100) : ("x" := 0) : env

(D) ("x" := 0) : ("y" := 100) : env

(E) [("y" := 100), ("x" := 0)]

**QUIZ**

Lets re-evaluate the quizzes!
-- env

let x = 0

in -- ("x" := 0) : env

  let x = 100

  in -- ??? what env to use for `x + 1`?

  x + 1

(A) ("x" := 0) : env

(B) ("x" := 100) : env

(C) ("x" := 100) : ("x" := 0) : env

(D) ("x" := 0) : ("x" := 100) : env

(E) [("x" := 100)]

---

Extending Environments

Let's fill in `eval` for the `let x = e1 in e2` case!
eval env (ELet x e1 e2) = ???

1. Evaluate e1 in env to get a value v1
2. Extend environment with value for x i.e. to (x := v1) : env
3. Evaluate e2 using extended environment.

Let's make sure our tests pass!

Run-time Errors

Haskell function to evaluate an expression:
eval :: Env -> Expr -> Value

eval env (Num n) = n

eval env (Var x) = lookup x env -- (A)

eval env (Bin op e1 e2) = eval0p op v1 v2 -- (B)
    where
      v1 = eval env e1 - (C)
      v2 = eval env e2 -- (C)

eval env (Let x e1 e2) = eval env1 e2
    where
      v1 = eval env e1
      env1 = extend env x v1 -- (D)

**QUIZ**

Will eval env expr always return a value? Or, can it crash?

(A) operation at A may fail

(B) operation at B may fail

(C) operation at C may fail
(D) operation at D may fail

(E) nah, its all good..., always returns a Value

Free vs bound variables

Undefined Variables

How do we make sure lookup doesn’t cause a run-time error?

Bound Variables

Consider an expression \texttt{let } x = e1 \texttt{ in } e2

- An occurrence of \texttt{x} is \texttt{bound} in \texttt{e2}
- i.e. when occurrence of form `let x = ... in ... x ...`

- i.e. when `x` occurs “under” a `let` binding for `x`.

**Free Variables**

An occurrence of `x` is **free** in `e` if it is **not bound** in `e`

**Closed Expressions**

An expression `e` is **closed** in environment `env`:

- If all free variables of `e` are defined in `env`

**Successful Evaluation**

`lookup` will never fail

- If `eval env e` is only called on `e` that is closed in `env`
QUIZ

Which variables occur free in the expression?

```plaintext
let y = (let x = 2
        in x ) + x

in
  let x = 3
  in
    x + y
```

(A) None

(B) x

(C) y

(D) x and y
Exercise

Consider the function

\[
\text{evaluate} :: \text{Expr} \rightarrow \text{Value} \\
\text{evaluate } e \\
\text{ | isOk } e = \text{eval emptyEnv } e \\
\text{| otherwise } = \text{error "Sorry! bad expression, it will crash `eval`!"} \\
\text{ where} \\
\text{emptyEnv } = [] \\
\text{ | has NO bindings}
\]

What should isOk check for? (Try to implement it for nano ...)

The Nano Language

Features of Nano:

1. Arithmetic expressions \(\text{[done]}\)
2. Variables \(\text{[done]}\)
3. Let-bindings \(\text{[done]}\)
4. Functions \(\text{[done]}\)
5. Recursion
Nano: Functions

Let’s add

- *lambda abstraction* (aka function definitions)
- *application* (aka function calls)

\[
\begin{align*}
e & ::= n & \quad \text{-- } OG \\
& | \ e_1 \ `\text{op}\` e_2 \\
& | \ x \\
& | \ \text{let } x = e_1 \ \text{in } e_2 & \quad \text{-- } NEW \\
& | \ \backslash x \ \rightarrow e & \quad \text{-- } \text{abstraction} \\
& | \ e_1 \ e_2 & \quad \text{-- } \text{application}
\end{align*}
\]

**Example**

```
let incr = \x \rightarrow x + 1

in
incr 10
```

**Representation**
data Expr
  = ENum Int               -- OLD
  | EBin Binop Expr Expr
  | EVar Id
  | ELet Id Expr Expr
                -- NEW
  | ???                -- abstraction |x -> e
  | ???                -- application (e1 e2)

Representation

data Expr
  = ENum Int               -- OLD
  | EBin Binop Expr Expr
  | EVar Id
  | ELet Id Expr Expr
                -- NEW
  | ELam Id Expr                -- abstraction |x -> e
  | EApp Expr Expr              -- application (e1 e2)
Example

```ocaml
let incr = \x -> x + 1
in
  incr 10
```

is represented as

```ocaml
ELet "incr" (ELam "x" (EBin Add (EVar "x") (ENum 1)))
  (
    EApp (EVar "incr") (ENum 10)
  )
```

Functions are Values

Recall the trinity
But... what is the value of a function?

Let's build some intuition with examples.
QUIZ

What does the following expression evaluate to?

```plaintext
let incr = \x -> x + 1  -- abstraction ("definition")

in

  incr 10  -- application ("call")
```

(A) Error/Undefined

(B) 10

(C) 11

(D) 0

(E) 1
What is the Value of $\text{incr}$?

- Is it an Int?
- Is it a Bool?
- Is it a ???

What information do we need to store (in the Env) about $\text{incr}$?

A Function’s Value is its Code
let incr = \x -> x + 1

-- env

in

-- ("incr" := <code>) : env

incr 10

-- evaluate <code> with parameter := 10

What information do we store about <code>?

---

**A Call’s Value**

How to evaluate the “call” incr 10?

1. Lookup the <code>i.e. <param, body> for incr (stored in the environment),

2. Evaluate body with param set to 10!
Two kinds of Values

We now have two kinds of Values

\[ v ::= n \quad \text{-- OLD} \]
\[ \mid <x, e> \quad \text{-- <param, body>} \]

1. Plain Int (as before)
2. A function’s “code”: a pair of “parameter” and “body-expression”

\[ \text{data Value} \]
\[ = \text{VInt Int} \quad \text{-- OLD} \]
\[ \mid \text{VCode Id Expr} \quad \text{-- <x, e>} \]
Evaluating Lambdas and Applications

eval :: Env -> Expr -> Value

-- OLD

eval env (ENum n) = ???
eval env (EVar x)  = ???
eval env (EBin op e1 e2) = ???
eval env (ELet x e1 e2) = ???

-- NEW

eval env (ELam x e) = ???
eval env (EApp e1 e2) = ???

Lets make sure our tests work properly!

exLam1 = ELet "incr" (ELam "x" (EBin Add (EVar "x") (ENum 1)))

  (EApp (EVar "incr") (ENum 10))

-- >>> eval [] exLam1
-- 11

QUIZ

What should the following evaluate to?
let c = 1
in
  let inc = \x -> x + c
  in
    inc 10

(A) Error/Undefined

(B) 10

(C) 11

(D) 0

(E) 1
exLam2 = ELet "c" (ENum 1)
  (ELet "incr" (ELam "x" (EBin Add (EVar "x") (EVar "c")))
   (EApp (EVar "incr") (ENum 10))
  )

-- >>> eval [] exLam2
-- ??

**QUIZ**

And what should *this* expression evaluate to?

```haskell
let c = 1
in
  let inc = \x -> x + c
  in
    let c = 100
    in
      inc 10
```

(A) Error/Undefined

(B) 110
The “Immutability Principle”

A function’s behavior should never change

- A function must always return the same output for a given input

Why?

> myFunc 10
0

> myFunc 10
10

Oh no! How to find the bug? Is it
In myFunc or
In a global variable or
In a library somewhere else or
...

My worst debugging nightmare

Colbert “Immutability Principle” (https://youtu.be/CWqzLgDc030?t=628)

**The Immutability Principle**

How does our eval work?

```plaintext
exLam3 = ELet "c" (ENum 1)
   (ELet "incr" (ELam "x" (EBin Add (EVar "x") (EVar "c")))
   (ELet "c" (ENum 100)
    (EApp (EVar "incr") (ENum 10)
       )
   )
)

-- >>> eval [] exLam3
-- ???
```
Oops?

```
let c = 1
in -- ["c" := 1]
    let inc = \x -> x + c
    in -- ["inc" := <x, x+c>, c := 1]
        let c = 100
        in -- ["c" := 100, "inc" := <x, x+c>, "c" := 1] <<< env
        inc 10
```

And so we get

```
eval env (inc 10)

    ==> eval ("x" := 10 : env) (x + c)

    ==> 10 + 100

    ==> 110
```

Ouch.

**Enforcing Immutability with Closures**

How to enforce immutability principle
Key Idea: Closures

At definition: Freeze the environment the function’s value

At call: Use the frozen environment to evaluate the body

Ensures that \texttt{inc 10} always evaluates to the same result!

```haskell
let c = 1
in
  let inc = \x -> x + c
  in
    let c = 100
    in
      inc 10
```

Now we evaluate
eval env (inc 10)

  ==> eval ("x" := 10 : frozenv) (x + c)  where frozenv = ["c" := 1]

  ==> 10 + 1

  ==> 1

tada!

**Representing Closures**

Lets change the Value datatype to also store an Env

```
data Value
    = VInt  Int          -- OLD
    | VClos Env Id Expr   -- <frozenv, param, body>
```

**Evaluating Function Definitions**

How should we fix the definition of eval for ELam?

```
eval :: Env -> Expr -> Value

eval env (ELam x e) = ???
```
**Hint:** What value should we *bind* `incr` to in our example above?

(Recall **At definition freeze** the environment the function’s value)

### Evaluating Function Calls

How should we fix the definition of `eval` for `EApp`?

```
  eval :: Env -> Expr -> Value

  eval env (EApp e1 e2) = ???
```

(Recall **At call:** Use the *frozen* environment to evaluate the *body*)

**Hint:** What value should we *evaluate* `incr 10` to?

1. Evaluate `incr` to get `<frozen, "x", x + c>`
2. Evaluate `10` to get `10`
3. Evaluate `x + c` in `x:=10 : frozen`
Let's generalize that recipe!

1. Evaluate e1 to get <froze \text{nv}, \text{param}, \text{body}>
2. Evaluate e2 to get v2
3. Evaluate body in \text{param} := v2 : \text{froze \text{nv}}

**Immutability Achieved**

Let's put our code to the test!

```haskell
exLam3 = ELet "c" (ENum 1)
  (ELet "incr" (ELam "x" (EBin Add (EVar "x") (EVar "c")))
   (ELet "c" (ENum 100)
    (EApp (EVar "incr") (ENum 10))
   )
  )

-- >>> eval [] exLam3
-- ???
```

**QUIZ**
What should the following evaluate to?

```haskell
let add = \x -> (\y -> x + y)
in
  let add10 = add 10
  in
    let add20 = add 20
    in
      (add10 100) + (add20 1000)
```

TODO

*Functions Returning Functions Achieved!*

```haskell
exLam4 = ...
```

```haskell
-- >>> eval [] exLam4
```

TODO

*QUIZ*

What should the following evaluate to?
let add = \x -> (\y -> x + y)
in
  let add10 = add 10
  in
    let doTwice = \f -> (\x -> f (f x))
    in
      doTwice add10 100

TODO

*Functions Accepting Functions Achieved!*

exLam4 = ...

-- >>> eval [] exLam4

TODO

*The Nano Language*
Features of Nano:

1. Arithmetic expressions [done]
2. Variables [done]
3. Let-bindings [done]
4. Functions [done]
5. Recursion

... You figure it out Hw4 ... :-)

You