2. Nano: Variables

Let’s add variables and let bindings!

\[ e ::= n \quad \text{-- OLD} \]
\[ | \quad e_1 + e_2 \]
\[ | \quad e_1 - e_2 \]
\[ | \quad e_1 * e_2 \]
\[ | \quad x \quad \text{-- NEW} \]
\[ | \quad \text{let } x = e_1 \text{ in } e_2 \]

Let’s extend our datatype

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

\textbf{data} \ Expr
\[ = \text{ENum Int} \quad \text{-- OLD} \]
\[ | \quad \text{EBin} \text{ Binop Expr Expr} \]
\[ | \quad \text{EVar} \text{ Id} \]
\[ | \quad \text{ELet} \text{ Id Expr Expr} \quad \text{-- NEW} \]

How should we extend \text{eval}?
QUIZ

What should the following expression evaluate to?

\[
\text{let } x = 0 \\
\text{in} \\
x + 1
\]

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

Elet "x" (Elnt 0)
\( \rightarrow \)
(EAdd (EVar "x") (Elnt 1))

QUIZ

What should the following expression evaluate to?

\[
\text{let } x = 0 \\
\text{in} \\
\text{let } y = 100 \\
\text{in} \\
x + y
\]

(A) Error
QUIZ

What should the following expression evaluate to?

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

(A) Error

(B) 0

(C) 1

(D) 100

(E) 101
**QUIZ**

What *should* the following expression evaluate to?

```plaintext
let x = 0
in
  ( let x = 100 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!

---

expr

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

(A) env

(B) [ ]

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

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

(E) env ++ ["x" := 0] [x := 55]

---

let x = 0
in x + 1

---

QUIZ

```
let x = 0
in
  let y = 100
  in x + y
```

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

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

Lets re-evaluate the quizzes!

```
let x = 0 in
  let x = 100 in
    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)]

---

`x + 1` needs an `x` in the environment. What should it be?
Extending Environments

Let's fill in `eval` for the `let x = e1 in e2` case!

\[
eval \text{ env (ELet } x \text{ e1 e2)} = ???
\]

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) = evalOp 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 = (x, v1) : env -- (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 `let x = e1 in e2`

- An occurrence of `x` is **bound** in `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?

```
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

\[
\text{evaluate :: Expr} \rightarrow \text{Value} \\
\text{evaluate } e \ |
\text{isOk } e = \text{eval emptyEnv } e \\
\text{otherwise } = \text{error } "\text{Sorry! bad expression, it will crash `eval `!}"
\]

where

\[
\text{emptyEnv } = [] \quad -- \text{has NO bindings}
\]

What should \text{isOk} check for? (Try to implement it for \text{nano} ...)

**The Nano Language**

Features of Nano:

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

Let’s add

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

\[
\begin{align*}
e ::= & \quad \text{-- OLD} \\
\mid & \quad n \\
\mid & \quad e_1 \`\text{op}`\ e_2 \\
\mid & \quad x \\
\mid & \quad \textbf{let} \ x = e_1 \ \textbf{in} \ e_2 \\
\mid & \quad \textbf{\{x \ -> \ e\}} \quad \text{-- abstraction} \\
\mid & \quad e_1 \ e_2 \quad \text{-- application}
\end{align*}
\]

**Example**

\[
\begin{align*}
\text{let } \text{incr} & = \{x \ -> \ x + 1\} \\
\text{in} \\
\text{incr} \ 10
\end{align*}
\]

**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

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

is represented as
Functions are Values

Recall the trinity

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

Lets build some intuition with examples.

QUIZ
What does the following expression evaluate to?

\[
\text{let } \text{incr} = \lambda x \to x + 1 \quad \text{-- abstraction ("definition")}
\]
\[
in \quad \text{incr} 10 \quad \text{-- application ("call")}
\]

(A) Error/Undefined

(B) 10

(C) 11

(D) 0

(E) 1

---

**What is the Value of \text{incr}?**

- Is it an \text{Int}?
- Is it a \text{Bool}?
- Is it a ???

**What information** do we need to store (in the \text{Env}) about \text{incr}?
A Function’s Value is its Code

```
let incr = \x -> x + 1
in incr 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),