Nano: Interpreters

CSE 130
Week ???

The plan

1. Interpreters
2. Hw4 concepts

The plan

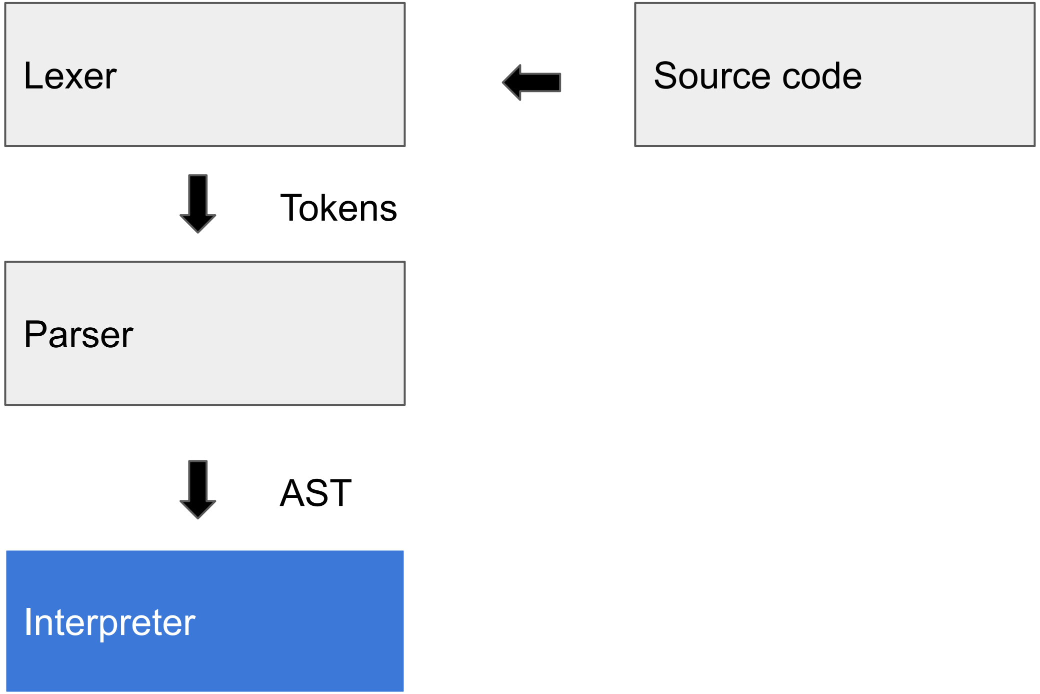
1. Interpreters
2. Hw4 concepts
   a. Environments
   b. Closures
   c. Native ops

Interpreters

Interpreters

An interpreter executes another program in some language without compilation

The big picture



A boring interpreter

```
Expr ::= IntLit Int | Add Expr Expr -- n | e1 + e2

eval :: Expr -> Int
eval e = ??
```

How do we implement `eval`?

A boring interpreter

```
Expr ::= IntLit Int | Add Expr Expr -- n | e1 + e2

eval :: Expr -> Int
eval (IntLit x)  = x
eval (Add e1 e2) = (eval e1) + (eval e2)
```

Nano is not so simple...

```
data Expr
  = EInt Int
  | EBool Bool
  | ENil                    -- []
  | EVar Id                 -- x
  | EBin Binop Expr Expr
  | EIf Expr Expr Expr      -- If e1 then e2 else e3
  | ELet Id Expr Expr       -- let x = e1 in e2
  | EApp Expr Expr          -- e1 e2
  | ELam Id Expr            -- \x. e
```

What does it mean to “evaluate” an Expr?

```
eval :: ??
eval = ??
```

What does it mean to “evaluate” an Expr?

```
eval :: Env -> Expr -> ??
eval = ??
```

Our output type needs to be able to represent any possible result -- a boolean, a list, etc...

What does it mean to “evaluate” an Expr?

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

data Value
  = VInt  Int
  | VBool Bool
  | VClos Env Id Expr         -- will discuss later
  | VNil                      -- []
  | VCons Value Value         -- x:xs
  | VPrim (Value -> Value)    -- will discuss later
```

Environments

How should we evaluate this:

```
let x = 5 in x + x
```

Environments

How should we evaluate this:

```
let x = 5 in x + x
```

We need to know the value of “x” while evaluating “x + x”

```
type Env = [(Id, Value)]
```

Environments

```
eval :: Env -> Expr -> Value
eval env e = ??
```

You might need to update the environment when recursively evaluating subexpressions

Closures

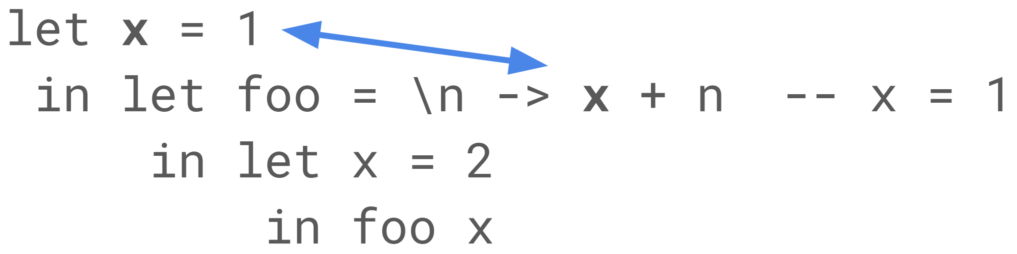
```
data Value
  = VInt  Int
  | VBool Bool
  | VClos Env Id Expr       -- will discuss later
  | VNil                    -- []
  | VCons Value Value       -- x:xs
  | VPrim (Value -> Value)  -- will discuss later
```

Why closures?

```
let x = 1
 in let foo = \n -> x + n
     in let x = 2
         in foo x
```

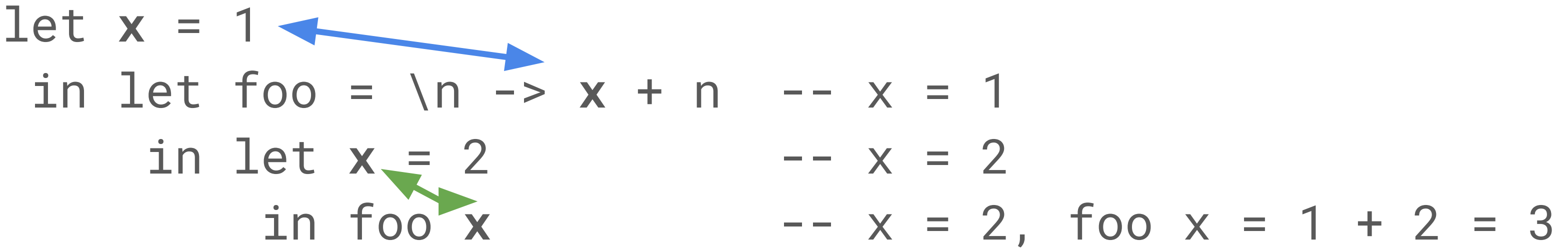
How should we evaluate this?

Why closures?



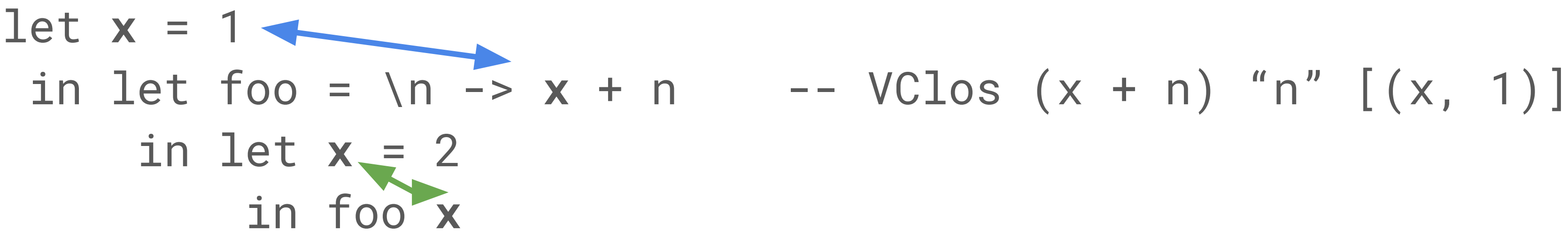
```
let x = 1
 in let foo = \n -> x + n  -- x = 1
   in let x = 2
     in foo x
```

Why closures?



```
let x = 1
 in let foo = \n -> x + n  -- x = 1
     in let x = 2          -- x = 2
         in foo x          -- x = 2, foo x = 1 + 2 = 3
```

When create a closures?



```
let x = 1
 in let foo = \n -> x + n     -- VClos (x + n) “n” [(x, 1)]
     in let x = 2
         in foo x
```

```
data Value
  = …                        -- (VClos e “x” env) is
  | VClos Env Id Expr        -- A function with argument “x”
  | …                        -- and body e that was defined
                             -- in an environment env
```

Native ops

```
data Binop = … | Cons

data ENil = … | ENil

data Value = … | VNil | VCons Value Value
```

Native ops

```
data Binop = … | Cons

data ENil = … | ENil

data Value = … | VNil | VCons Value Value
```

Now, add support for “head” and “tail”...

Native ops

Now, add support for “head” and “tail”...

How might we do this?

Native ops

We need to be able to define primitive function -- sort of a standard library

One function constructor:

```
VClos Env Id Expr
```

Native ops

We need to be able to define primitive function -- sort of a standard library

One function constructor:

```
VClos Env Id Expr
```

This won’t let us define “head” or “tail”!

The Expr type doesn’t allow us to pattern match on the list -- no way to represent these functions in our Expr language

Native ops

```
data Value = … | VPrim (Value -> Value)
```

Now you can implement normal Haskell functions over Values and use them

Stuff I haven’t talked about

Function application

```
let f = \x -> x + 1
 in f 3
```

Stuff I haven’t talked about

Function application

```
let f = \x -> x + 1 -- [(“f”, VClos [] “x” (“x” + 1))]
 in f 3
```

How does one evaluate `(f 3)`?

Stuff I haven’t talked about

Function application

```
let f = \x -> x + 1 -- [(“f”, VClos [] “x” (“x” + 1))]
 in f 3
```

Stuff I haven’t talked about

Function application

```
let factorial = \n ->
       if n <= 0
          then 1
          else n * (factorial (n - 1))
 in factorial 3
```