CS440: Programming Languages and Translators

Lecture 11

Spring 2023

Environments with functions: first try

```
let add = fun (x, y) -> x + y
let three = add (1, 2)
```

Var	Value

Environments with functions: first try

```
let add = fun (x, y) -> x + y
let three = add (1, 2)
```

Var	Value
add	fun (x, y) -> x + y

Environments with functions: first try

```
let add = fun (x, y) -> x + y
let three = add (1, 2)
```

Var	Value
add	fun (x, y) -> x + y
x	1
y	2

Environments with functions: first try

```
let add = fun (x, y) -> x + y
let three = add (1, 2)
```

Var	Value
add	fun (x, y) -> x + y
three	3

Environments with functions: first try

```
let add = fun x -> fun y -> x + y
let add1 = add 1
let three = add1 2
```

Var	Value
add	fun x -> fun y -> x + y

Environments with functions: first try

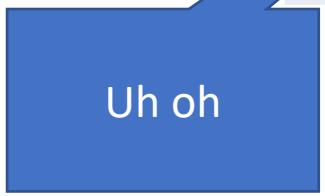
```
let add = fun x -> fun y -> x + y
let add1 = add 1
let three = add1 2
```

Var	Value
add	fun x -> fun y -> x + y
x	1

Environments with functions: first try

```
let add = fun x -> fun y -> x + y
let add1 = add 1
let three = add1 2
```

Var	Value
add	fun x -> fun y -> x + y
add1	fun y -> x + y



Environments with functions: first try

```
let x = 1 in
let f y = x + y in
let x = 2 in
f 2
```

Var	Value

Environments with functions: first try

```
let x = 1 in
let f y = x + y in
let x = 2 in
f 2
```

Var	Value
x	1

Environments with functions: first try

```
let x = 1 in
let f y = x + y in
let x = 2 in
f 2
```

Var	Value
x	1
f	fun y -> x + y

Environments with functions: first try

```
let x = 1 in
let f y = x + y in
let x = 2 in
f 2
```

Var	Value
x	1
f	fun y -> x + y
x	2

Environments with functions: first try

```
let x = 1 in
let f y = x + y in
let x = 2 in
f 2
```

Var	Value
x	1
f	fun y -> x + y
x	2
y	2

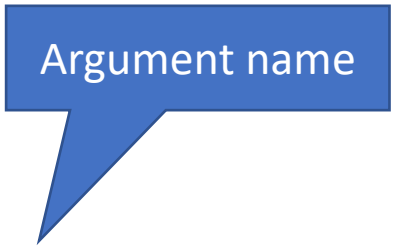
x should still be 1 in f!

Second try: use *closures*

- Closure: function code + environment
- This will be the value of a function

- in ML:

```
type value = …
           | VClos of var * expr * env
and env = (var * value option ref) list
```



With closures

```
let x = 1 in
let f y = x + y in
let x = 2 in
f 2
```

Var	Value

With closures

```
let x = 1 in
let f y = x + y in
let x = 2 in
f 2
```

Var	Value
x	1

With closures

```
let x = 1 in
let f y = x + y in
let x = 2 in
f 2
```

Var	Value
x	1
f	(fun y -> x + y, Var | Value x | 1)

With closures

```
let x = 1 in
let f y = x + y in
let x = 2 in
f 2
```

Var	Value
x	1
f	(fun y -> x + y, Var | Value x | 1)
x	2

With closures

```
let x = 1 in
let f y = x + y in
let x = 2 in
f 2
```

Var	Value
x	1
y	2

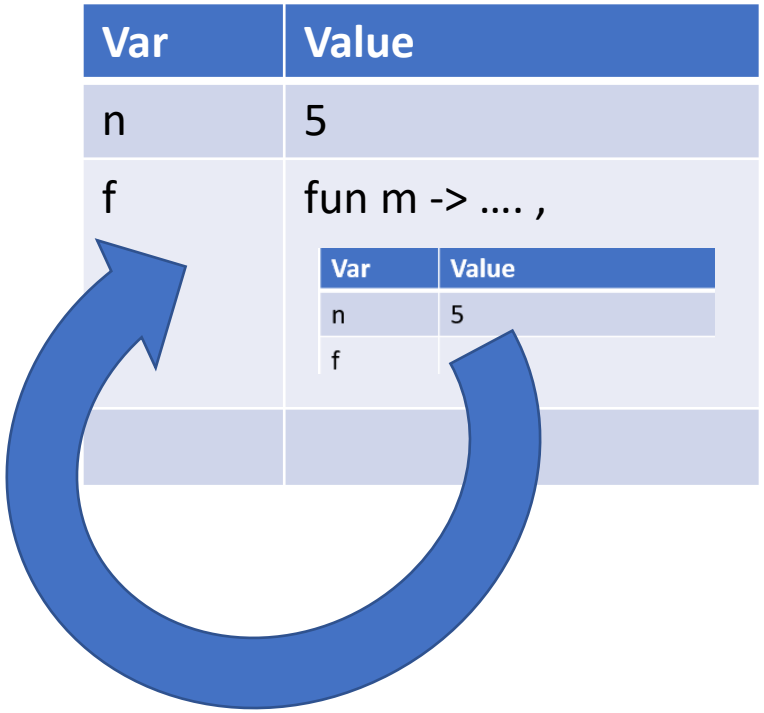
Call the function with the environment from the closure (+ arguments)

Interpreting with closures

- Interpreting a function: fun x -> e
  - Return a closure with variable x, expression e, current environment
- Interpreting an application e1 e2
  - Interpret e1 to closure (x, e, env)
  - Interpret e2 to arg value v
  - Add x -> v to env, interpret e with this env

Recursive closures

```
let n = 5 in
let rec f m =
  if m >= n then 1
  else m * f (m + 1)
in
f 0
```



Interpreting with recursive closures

- Interpreting a function def: let rec f x = e1 in e2
  - Let env’ = current env extended with placeholder for f
  - Let clos = Vclos(x, e1, env’)
  - Update env’ with f -> clos
  - Evaluate e2 with env’
- Interpreting an application e1 e2
  - Same as before: when you evaluate e1 to a closure, the function is already in the environment