CS443: Compiler Construction

Lecture 10: Closure Conversion

Stefan Muller

Based on material from Steve Chong, Steve Zdancewic, and Greg Morrisett

This ~~Next~~ time

- Suggests how to compile: closure now doesn't depend on environment
  - Add code to build closures (*closure conversion*)
  - Lift code parts of closures into top-level functions (*hoisting/lambda lifting*)

Add the environment as an extra parameter to functions

```
fun (y: int) : int -> x + y
```

```
int __fun (env env, int y) {
  env = __extend_env(env, “y”, y);
  return __lookup(env, “x”) + y;
}
```

Environment now includes y also.

Environment loses y when y goes out of scope

Can also just look y up in the environment

```
fun (y: int) : int -> x + y
```

Pro: uniform treatment of vars

Con: Less efficient

```
int __fun (env env, int y) {
  env = __extend_env(env, “y”, y);
  return __lookup(env, “x”) + __lookup(env, “y”);
}
```

We need to make sure the environment keeps up with ML variable scope

```
let x = (let x = 1 in x + x) + 1 in x

int x_1 = 1
env = __extend_env(env, “x”, x_1);
int temp_1 = x_1 + x_1;
env = __pop_env(env);
int x_2 = temp_1 + 1;
env = __extend_env(env, “x”, x_2);
int temp_3 = x_2;
env = __pop_env(env);
```

As suggested by “extend” and “pop”, environment follows a stack

```
let x = 1 in x + (let y = 2 in x + y) + x

int x_1 = 1;
env = __extend_env(env, “x”, x_1);
int y_1 = 2;
env = __extend_env(env, “y”, y_1);
temp_1 = x_1 + y_1;
env = __pop_env(env);
temp_2 = x_1 + temp_1 + x_1
env = __pop_env(env);
```

A closure is a pair of the function code and the current environment

```
let x = 1 in
let inc = fun y -> x + y in
inc 2

int x_1 = 1;
env = __extend_env(env, “x”, x_1);
closure inc_1 = __mk_clos( “fun y -> x + y”  , env);
env = __extend_env(env, “inc”, inc_1);
int temp_1 = __call_closure(inc_1, 2);
```

(But the function code needs to be lifted to the top level)

```
int inc1__body(env env, int y) {
  env = __extend_env(env, “y”, y);
  return __lookup(env, “x”) + y;
}

int x_1 = 1;
env = __extend_env(env, “x”, x_1);
closure inc_1 = __mk_clos(inc1__body       , env);
env = __extend_env(env, “inc”, inc_1);
int temp_1 = __call_closure(inc_1, 2);
```

Call a closure by calling the function with the closure's environment (NOT the current one)

```
int inc1__body(env env, int y) {
  env = __extend_env(env, “y”, y);
  return __lookup(env, “x”) + y;
}

int x_1 = 1;
env = __extend_env(env, “x”, x_1);
closure inc_1 = __mk_clos(inc1__body       , env);
env = __extend_env(env, “inc”, inc_1);
int temp_1 = inc_1.clos_fun(inc_1.clos_env, 2)
```

For recursive functions, the function itself needs to be in the environment

```
let rec fact n = if n <= 1 then n else n * (fact (n – 1))

int fact__body(env env, int n) {
  env = __extend_env(env, “n”, n);
  if (n <= 1) { return n; }
  else {
    return n * __lookup(env, “fact”).clos_fun(
               __lookup(env, “fact”).clos_env, n – 1);
  }
}
env = __extend_env(env, “fact”, __mk_clos(fact__body, env))
```

Gets a little tricky depending on how we define environments—we’ll revisit this later

Do closure conversion and hoisting in one pass

```
compile_exp : ML.Ast.t_exp ->
  C.Ast.p_stmt_list * C.Ast.p_exp * closure list
```

Any extra statements needed to compute the value of the expression

A C expression that computes the value of the ML expression (assuming the statements have run)

Closures collected while compiling the expression

Do closure conversion and hoisting in one pass

```
compile_exp : ML.Ast.t_exp ->
  C.Ast.p_stmt_list * C.Ast.p_exp * closure list
```

```
let x = 1 in
let inc = fun y -> x + y in
inc 2
```

```
int x_1 = 1;
env = __extend_env(env, “x”, x_1);
closure inc_1 = __mk_clos(inc1__body, env);
env = __extend_env(env, “inc”, inc_1);
int temp_1 = inc_1.clos_fun(inc_1.clos_env, 2)
```

```
temp_1
```

```
[int inc1__body(env env, int y) {
  env = __extend_env(env, “y”, y);
  return __lookup(env, “x”) + y;
}]
```

Do closure conversion and hoisting in one pass

```
compile_exp : ML.Ast.t_exp ->
  C.Ast.p_stmt_list * C.Ast.p_exp * closure list
```

```
let x = 1 in
let inc = fun y -> x + y in
inc 2
```

```
int x_1 = 1;
env = __extend_env(env, “x”, x_1);
closure inc_1 = __mk_clos(inc1__body, env);
env = __extend_env(env, “inc”, inc_1);
```

```
inc_1.clos_fun(inc_1.clos_env, 2)
```

```
[int inc1__body(env env, int y) {
  env = __extend_env(env, “y”, y);
  return __lookup(env, “x”) + y;
}]
```

A lambda just evaluates to a closure

```
compile_exp(fun (x: int) : int -> e)
```

```
closure temp_1 = __mk_clos(__fun, env);
```

temp_1

```
[int __fun(env env, int x) {
   env = __extend_env(env, “x”, x);
  … compilation of e …
}; plus any closures nested in e]
```

Funcception

```
let add = fun x -> fun y -> x + y

closure __fun1(env env, int x) {
  __env = __extend_env(env, “x”, x);
  return __mk_clos(__fun2, env);
}

int __fun2(env env, int y) {
  __env = __extend_env(env, “y”, y);
  return __lookup(env, “x”) + __lookup(env, “y”);
}
env = __extend_env(env, “add”, __mk_clos(__fun1, env));
```

Applications evaluate the two expressions, then apply

```
compile_exp(e1 e2)
```

```
statements for e1
statements for e2
```

exp_e1.clos_fun(exp_e1.clos_env, exp_e2)

(closures from e1) @ (closures from e2)

Applications evaluate the two expressions, then apply

Where do we push/pop?

```
compile_exp(let x = e1 in e2)
```

```
statements for e1
statements for e2
```

(closures from e1) @ (closures from e2)

Applications evaluate the two expressions, then apply

```
compile_exp(let x = e1 in e2)
```

```
statements for e1
env = __extend_env(env, “x”, e1_exp);
statements for e2
temp_1 = e2_exp;
env = __pop_env(env);
```

temp_1

(closures from e1) @ (closures from e2)

Think about what goes wrong if we did this

```
compile_exp(let x = e1 in e2)
```

```
statements for e1
env = __extend_env(env, “x”, e1_exp);
statements for e2
env = __pop_env(env);
```

e2_exp

(closures from e1) @ (closures from e2)

I guess we need to compile other things too

`compile_exp(if e1 then (e2: int) else (e3: int))`

statements for e1
`int temp_1`
`if (e1) {` statements for e2; `temp_1 = exp_e2; }`
`else {` statements for e3; `temp_1 = exp_e3; }`

temp_1

(closures from e1) @ (closures from e2) @ (closures from e3)

How to *actually* represent closures

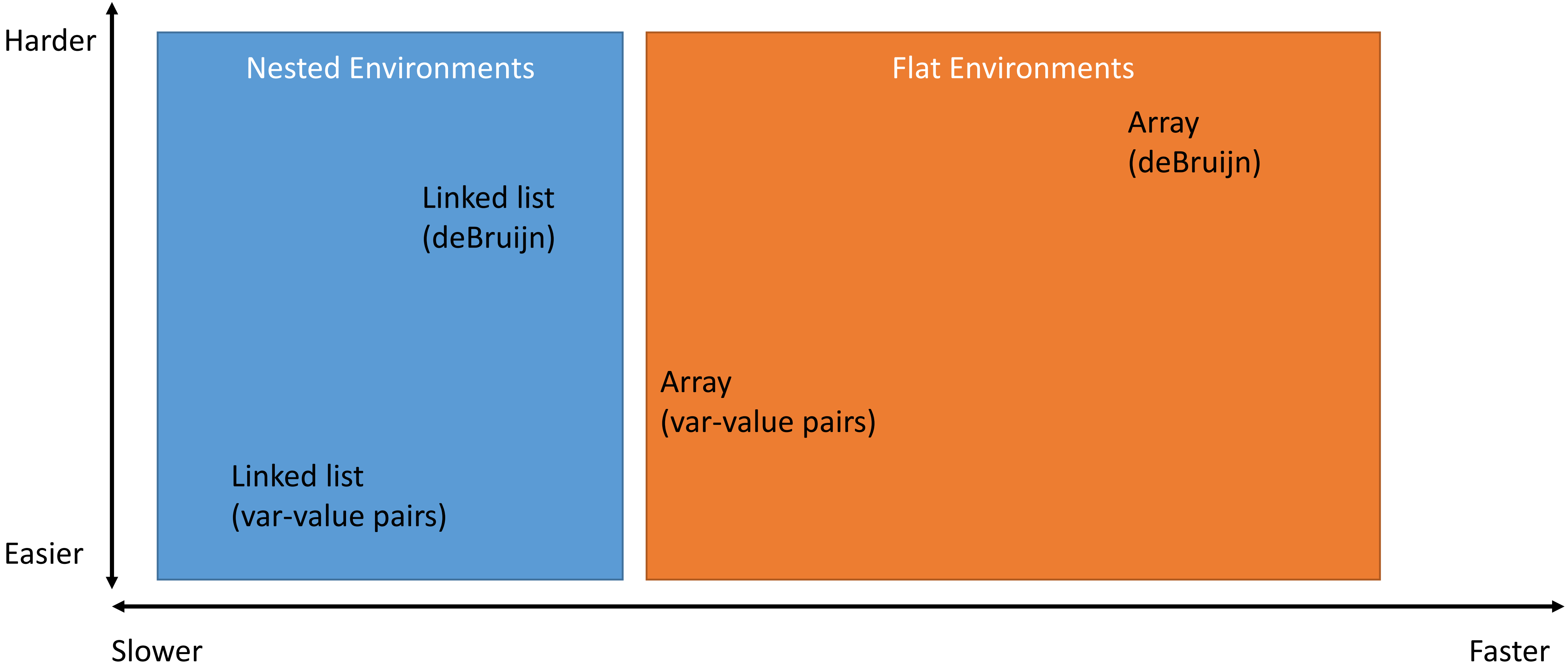
```
struct __clos {
  env clos_env;
  int clos_fun();
};
```

Stand-in function pointer type since C doesn't have parametric polymorphism. We'll need to cast it to whatever

How to represent environments

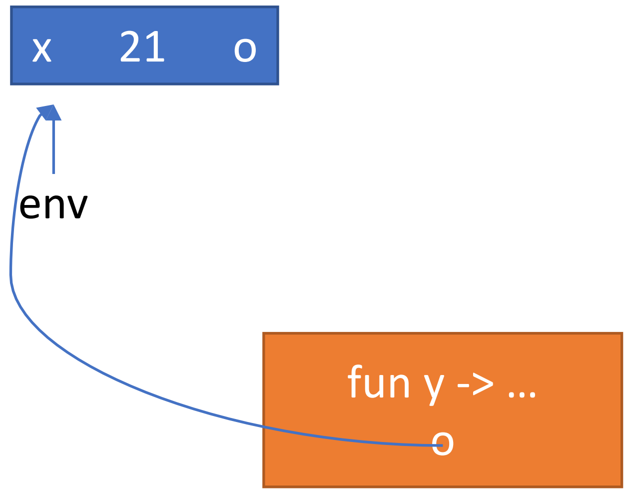
- Considerations:
  - Optimization: we don’t need to store all variables in the environment, just those that might “escape” (be used in nested functions)
  - Data structure: lookup should be fast (asymptotic and constant factors)

Data Structures for Environments



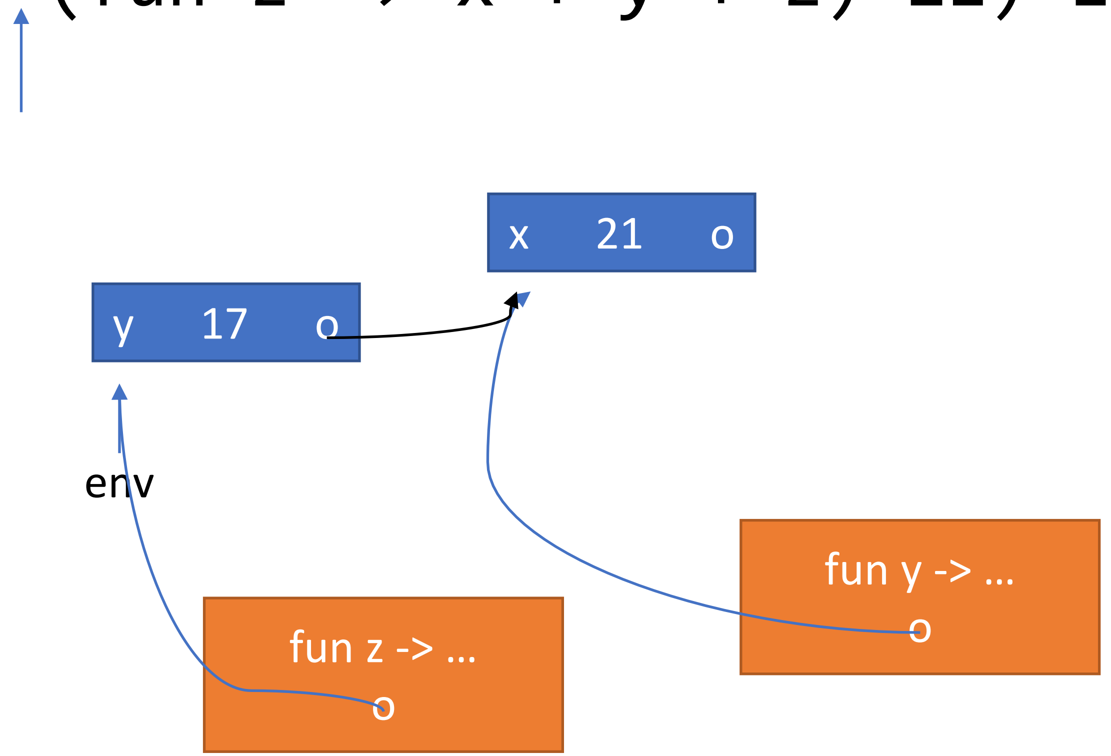
Nested Environments

```
(((fun x -> (fun y -> (fun z -> x + y + z) 21) 17) 4
```



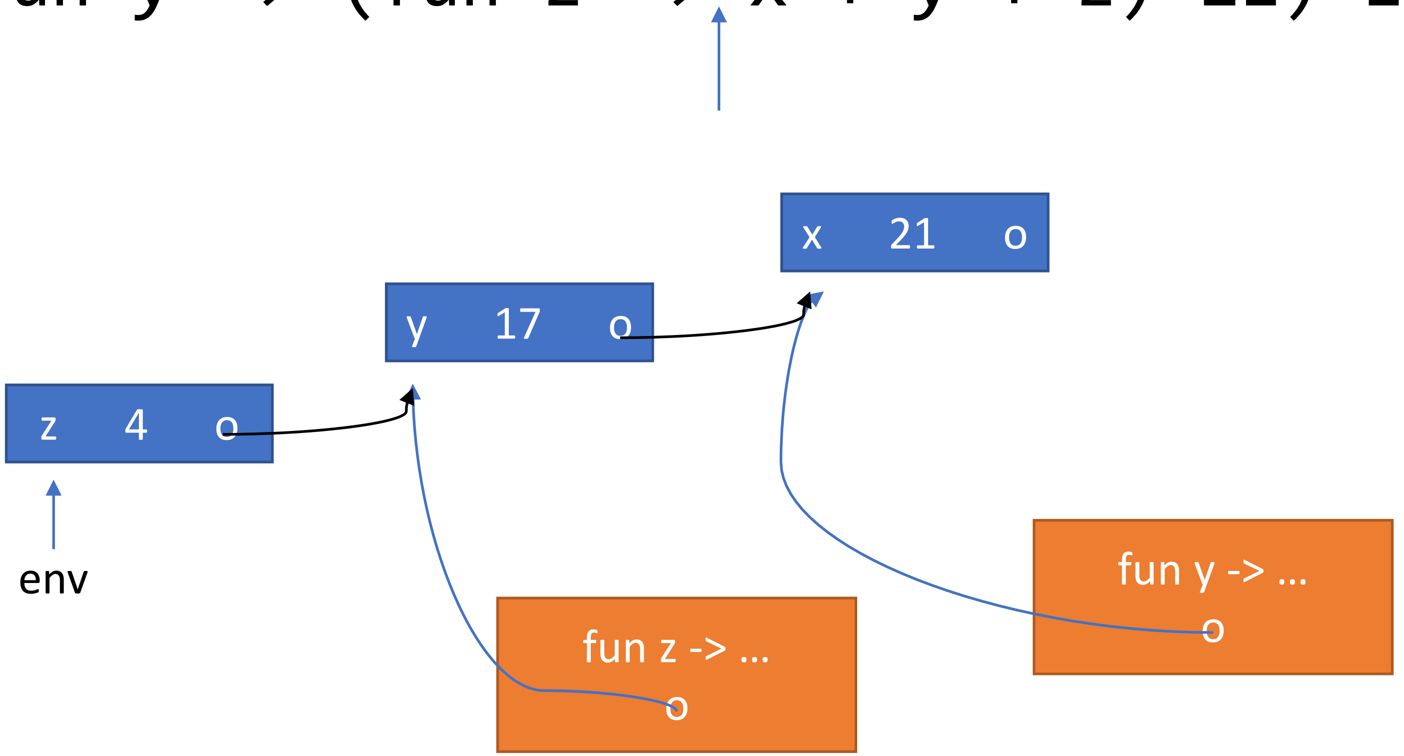
Nested Environments

```
(((fun x -> (fun y -> (fun z -> x + y + z) 21) 17) 4
```



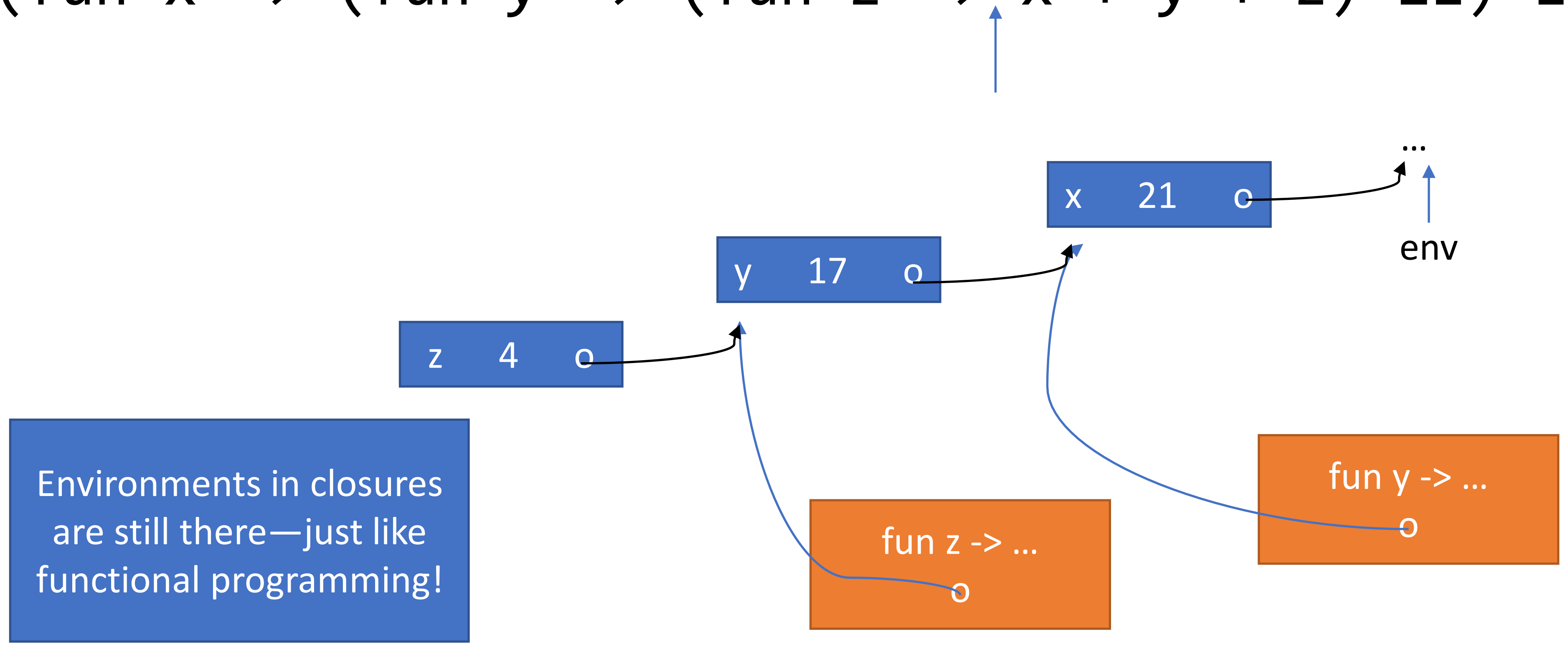
Nested Environments

```
(((fun x -> (fun y -> (fun z -> x + y + z) 21) 17) 4
```



Nested Environments

```
(((fun x -> (fun y -> (fun z -> x + y + z) 21) 17) 4
```



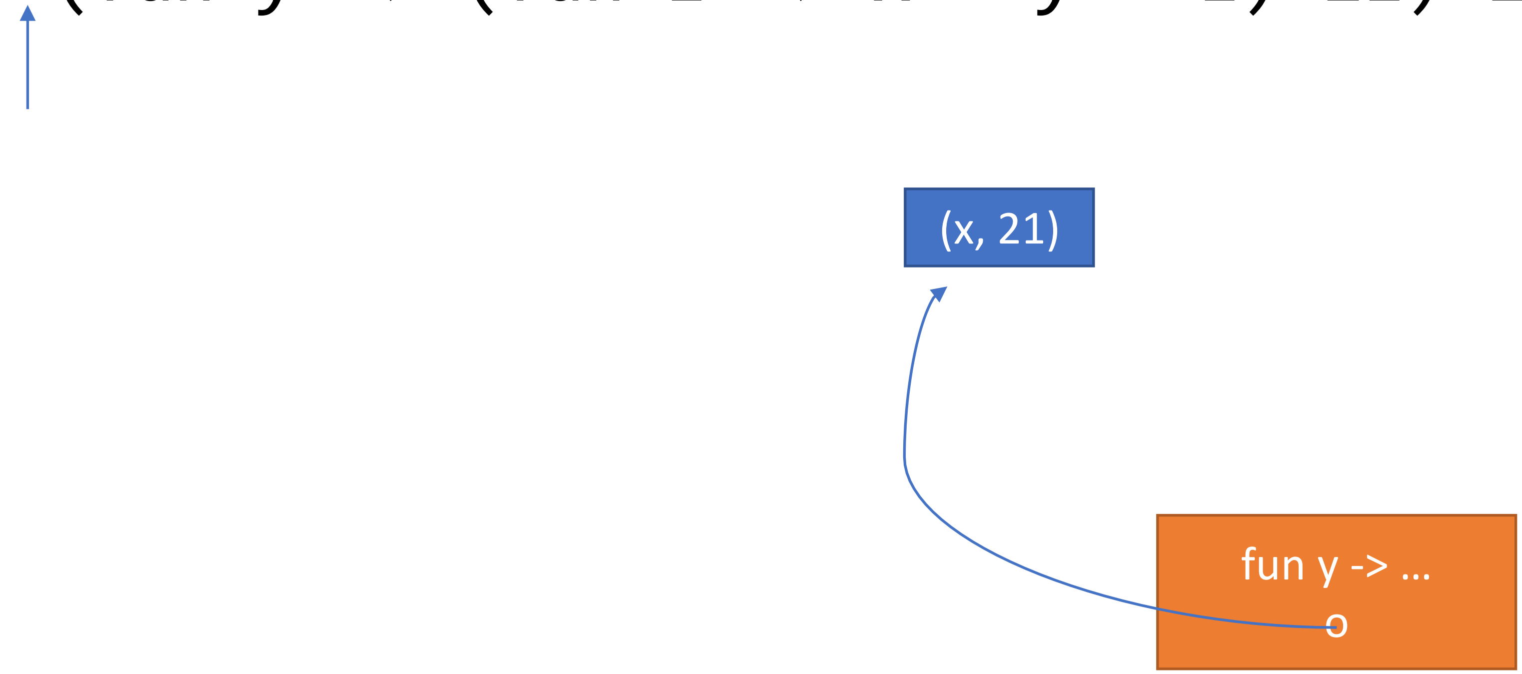
Extend and Lookup for Nested Envs

```
__extend_env(env, var, val):
  env new_node = new env(var, val, env)
  return new_node

__lookup(env, var):
  while(env.var != var && env != NULL):
    env = env.next
  return env.val
```

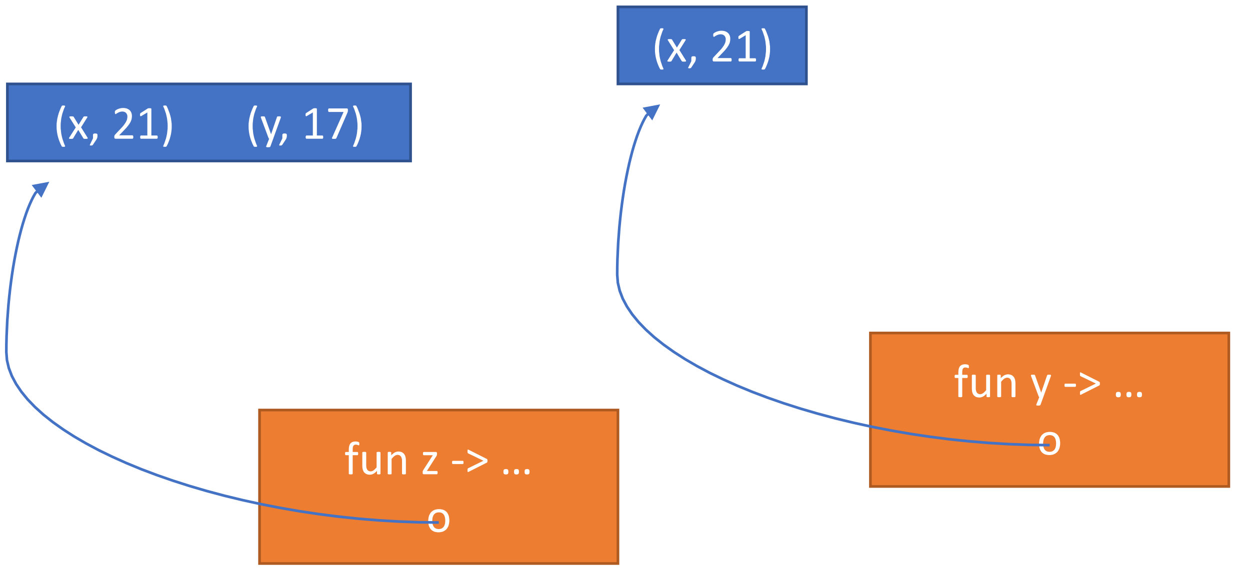
Flat Environments

```
(((fun x -> (fun y -> (fun z -> x + y + z) 21) 17) 4
```



Flat Environments

```
(((fun x -> (fun y -> (fun z -> x + y + z) 21) 17) 4
```



Pro: Faster lookup
Con: Slower construction

Extend and Lookup for Flat Envs

```
__extend_env(env, var, val):
  env new_env = new (env[env.length + 1])
  env[0] = (var, val)
  env[1:] = copy(env)
  return env

__lookup(env, var):
  i = 0
  while(env[i].var != var && i < env.length):
    i++
  return env[i].val
```

deBruijn Indices Track Number of Binders

```
(((fun x -> (fun y -> (fun z -> x + y + z) 21) 17) 4
```

deBruijn Indices Track Number of Binders

```
(((fun   -> (fun   -> (fun   ->   +   +   ) 21) 17) 4
```

deBruijn Indices Track Number of Binders

```
(((fun -> (fun -> (fun -> 2 + 1 + 0) 21) 17) 4
```

deBruijn Indices: Example

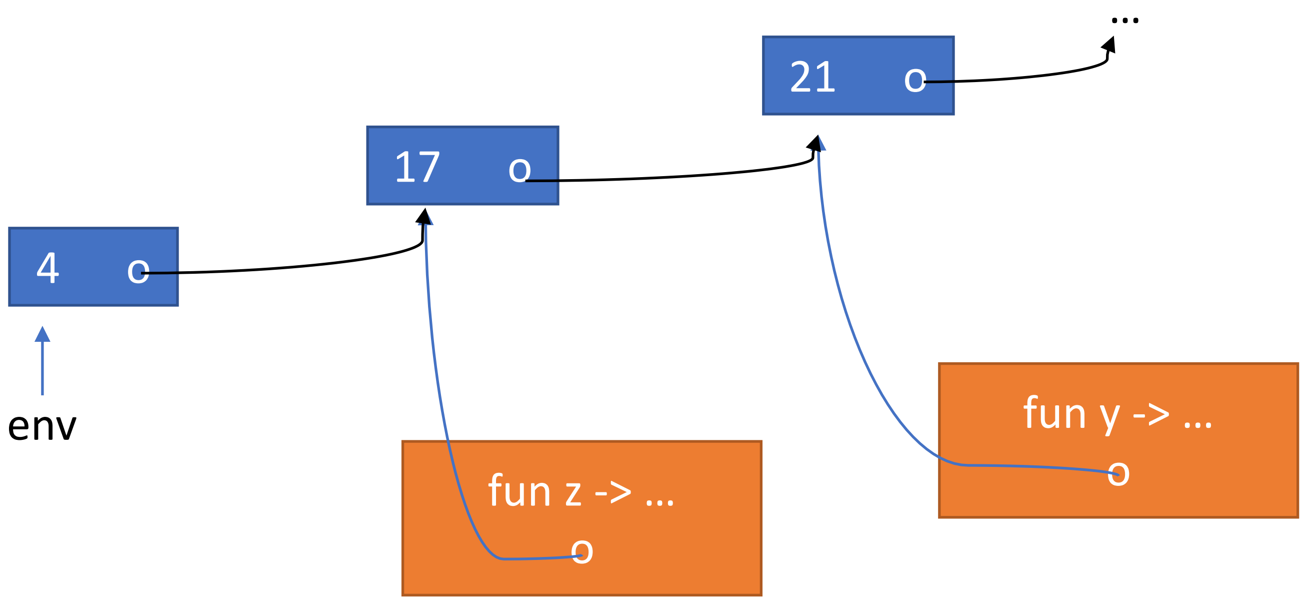
```
let x = 1 in x +
             (let y = 2 in
                (let x = 3 in x + y)
                + y)

let = 1 in 0 +
           (let = 2 in
              (let = 3 in 0 + 1)
              + 0)
```

> Note: Same binder can have different indices at different points in the program!

Nested Environments with deBruijn Indices

```
(((fun -> (fun -> (fun -> 2 + 1 + 0) 21) 17) 4
```



Extend and Lookup for Nested Envs (deBruijn)

```
__extend_env(env, val):
  env new_node = new env(val, env)
  return new_node

__lookup(env, ind):
  while(ind > 0):
    env = env.next
    ind--
  return env.val
```

Extend and Lookup for Flat Envs (deBruijn)

```
__extend_env(env, val):
 env new_env = new (env[env.length + 1])
  env[0] = val
  env[1:] = copy(env)
  return env

__lookup(env, ind):
  return env[ind]
```

Compromise: Keep variable names, but remember their deBruijn index while compiling

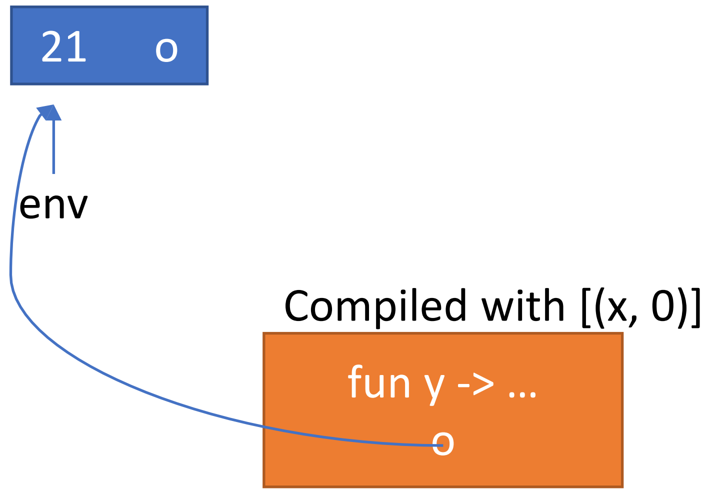
"Environment record"

```
compile_exp : (string * int) list -> ML.Ast.t_exp ->
  C.Ast.p_stmt_list * C.Ast.p_exp * closure list
```

- Con: Have to keep environment record in sync with environment
- Pro: Way easier to debug

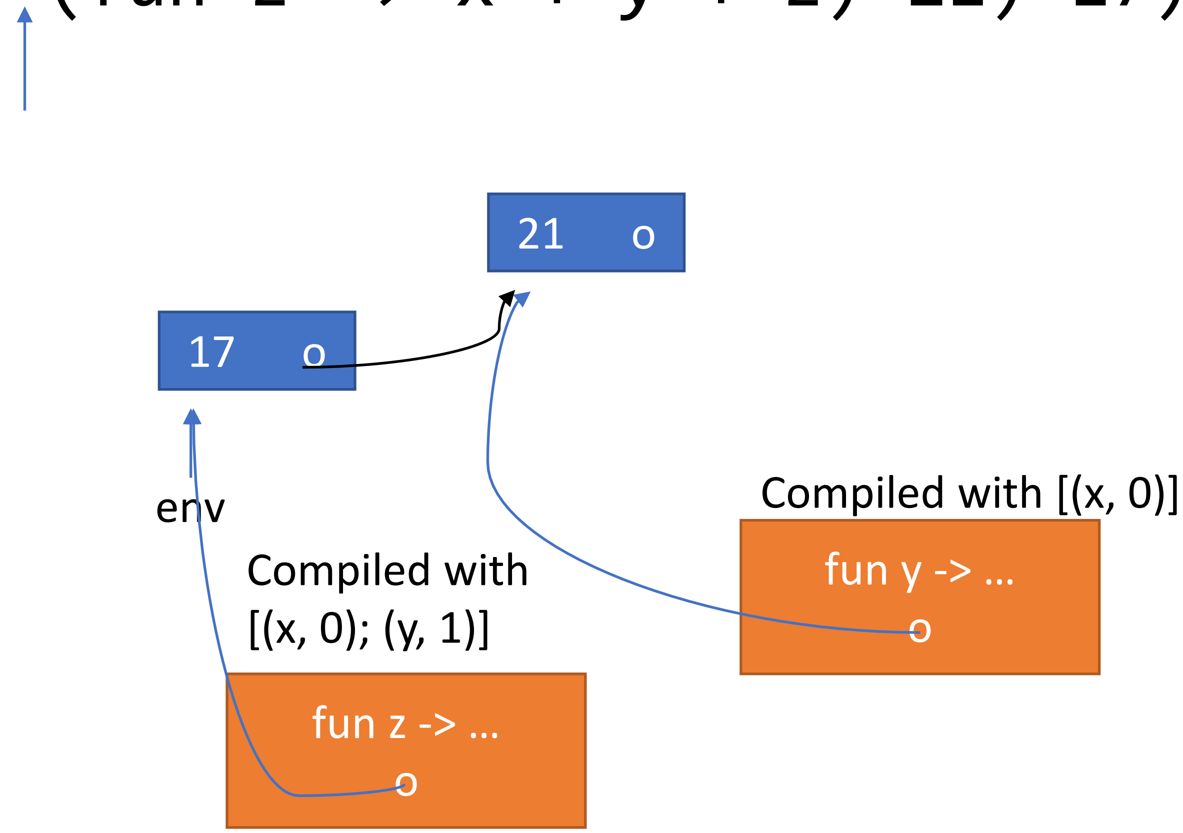
Nested Environments

```
(((fun x -> (fun y -> (fun z -> x + y + z) 21) 17) 4
```

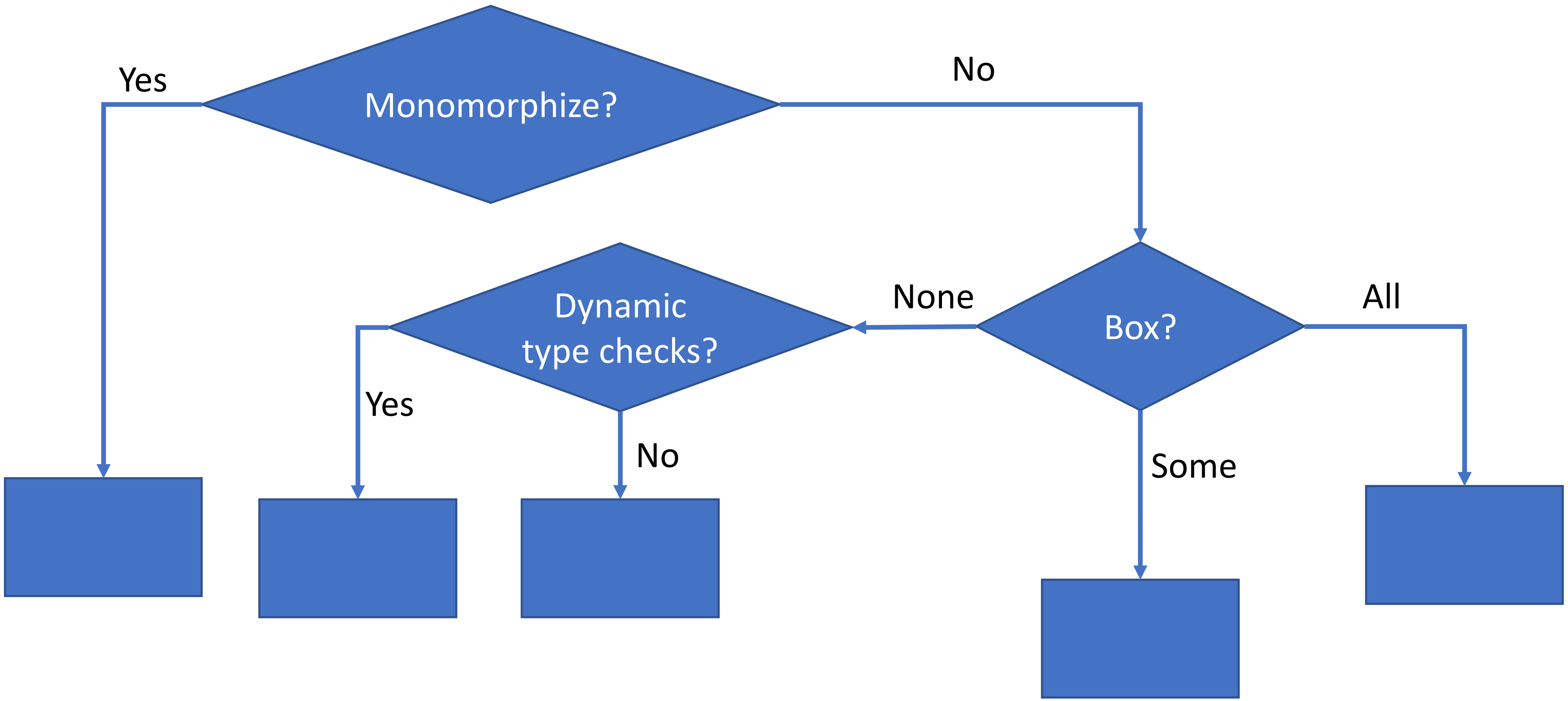


Nested Environments

```
(((fun x -> (fun y -> (fun z -> x + y + z) 21) 17) 4
```



There are a lot of ways to compile values



We (probably) want a uniform representation of values

```
‘a list
```

```
struct __list{
  value hd;
  __list tl;
};
```

Could pull the “pick a default type and cast as necessary” trick but still want values to all be the same size

First option: actually just have one type of values

```
enum Tag {INT, BOOLEAN, …} ;

struct Int { enum Tag t ; int value ; } ;

struct Boolean { enum Tag t ; unsigned int value ; } ;
…
union Value {
  enum Tag t ;
  struct Int z ;
  struct Boolean b ;
  …
} ;
```

Courtesy Matt Might: https://matt.might.net/articles/compiling-scheme-to-c/

Then we have to check the tag of an object when we use it…

```
Value neg(Value i) {
  switch (i.t) {
    case INT:
      Int ret;
      Int.t = INT;
      Int.value = -((Int) i).value;
      return ret;
    default:
      //Type Error!
      exit 1;
  }
}
```

Easy, Slow, Wasteful

…or do we?

- No (in a statically typed language without something like `instanceof`)

```
Value neg(Value i) {
  Int ret;
  Int.t = INT;
  Int.value = -((Int) i).value;
  return ret;
}
```

Easy, Fast, Wasteful

Second option: "Boxing" (use pointers for everything)

```
typedef void * Value

struct Int { int value; };
struct Boolean { bool value; };
struct List { Value hd; Value tl };
```

Key idea: we may not know a value's value at compile time, but we know its type!

Second option: "Boxing" (use pointers for everything)

```
let l: int list = 1::[]
in (hd l) + 2
```

↓

```
Value l = malloc(sizeof(List));
Value i = malloc(sizeof(Int));
((Int *)i)->value = 1;
((List *)l)->hd = i;
((List *)l)->tl = null;
Value i2 = malloc(sizeof(Int));
((Int *)i2)->value = 2;
return ((Int *) l->hd)->value + ((Int *) i2)->value
```

Harder, slower, still pretty wasteful

Compromise: “Unbox” ints, other small base types

```
let l: int list = 1::[]
in (hd l) + 2


Value l = malloc(sizeof(List));
((List *)l)->hd = (Value 1);
((List *)l)->tl = null;
return ((Int *) l->tl)->value + ((Int *) i2)->value
```

Harder, relatively fast, space-efficient

Have structs for different types

```
struct __list {
  int list_hd;
  __list list_tl;
};

struct __pair {
  int pair_fst;
  int pair_snd;
};
```

We need to pick a default type for values.
May as well use int (no void* in MiniC)

We still need dynamic tag checks for ADTs

```
type exp = EVar of string
         | EBinop of exp * exp
```

```
enum exp_tag { EVAR; EBINOP };
union exp;
struct EVar {
  exp_tag t;
  char[] arg1;
}
struct EBinop {
  exp_tag t;
  union exp *arg1;
  union exp *arg2;
}
union exp {
  struct EVar evar;
  struct EBinop ebinop;
}
```

A totally different option: get rid of polymorphism ("monomorphize")

```
struct int_list{
  int hd;
  __list tl;
};

struct bool_list{
  boolean hd;
  __list tl;
};
```

That means we need to make different versions of polymorphic functions

```
let pair (x: ‘a) : ‘a * ‘a = (x, x)



intpair pair_int(x: int) { … }
boolpair pair_bool(x: bool) { … }
…
```

(We’ll also need pair_intpair, pair_intboolpair, …)

To monomorphize functions, we need to know all the ways they can be used

- Check all call sites → Whole program compilation

Much harder
Slow, non-modular compilation
Blindingly fast at runtime
Space-efficient

There are a lot of ways to compile values