# CS 516: COMPILERS

## Lecture 14

*Topics*

- Introduction to Closures.
- Closure Conversion: Compiling lambda calculus to straight-line code.
- Static Analysis I: Scope Checking

*Materials*

- lec14.zip

Compiling lambda calculus to straight-line code.

Representing evaluation environments at runtime.

# CLOSURE CONVERSION

# Compiling First-class Functions

- We introduced high-level language, the Lambda Calculus
- Lambda calculus uses first-class functions.
- We looked at interpreting Lambda Calculus, but to implement first-class functions on a processor, there are two problems:
  1. We must implement substitution of free variables
  2. We must separate 'code' from 'data'

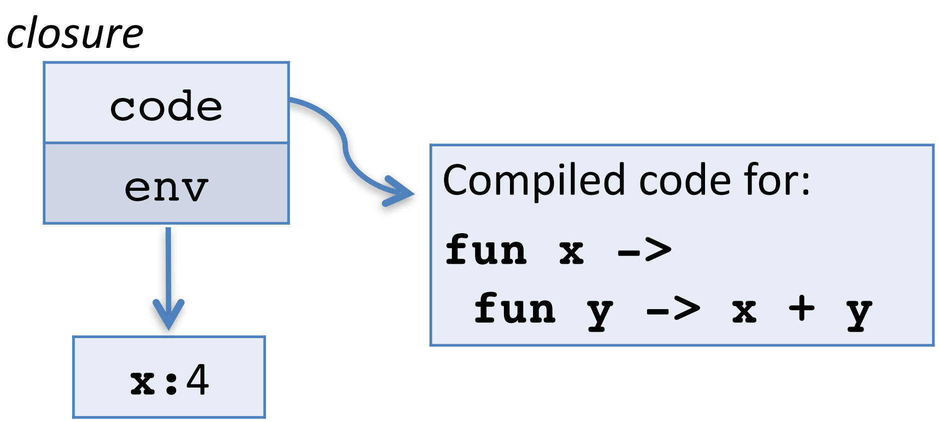
- Reify the substitution:
  - Move substitution from the meta language to the object language by making the data structure & lookup operation explicit
  - The environment-based interpreter is one step in this direction
- Closure Conversion:
  - Eliminates free variables by packaging up the needed environment in the data structure.
- Hoisting:
  - Separates code from data, pulling closed code to the top level.

# Example of closure creation

- Recall the "add" function:
  `let add = fun x -> fun y -> x + y`

- Consider the inner function: `fun y -> x + y`

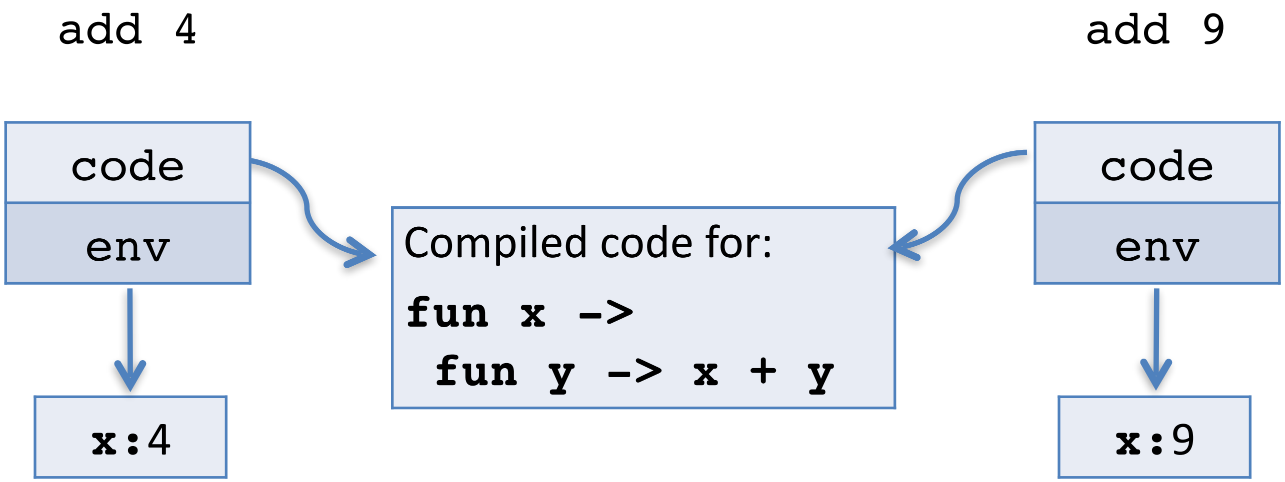
- When running the function application: `add 4`
  the program builds a closure and returns it.
  - The ***closure*** is a pair of the *environment* and a *code pointer*.



# Example of closure creation

- Can share closures:

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



# Representing Closures

- Using closures instead of function pointers to represent functions changes the way they are manipulated at **runtime**:
  - function abstraction (λ x. e) **builds and returns** a closure instead of a simple code pointer
  - function application (f v) **extracts the code pointer** from the closure, and **invokes it** with the environment as an additional argument.
    - Nothing is known about the closure being called – it can be any closure in the program.
    - Therefore, code pointer must be at a known and constant location.
    - Doesn't use the environment **env**.
    - Instead, the body of f may directly access **env**.

# Compiling Closures

- Compilers simplify closures via **closure conversion**.
- Transform a program
  - **from:** functions with nesting and free variables
  - **into**: equivalent program containing only top-level (and hence closed) functions
    - In this output, all functions can be represented as code pointers

- Two phases:
  1. **Close functions** by introducing environments (a dictionary for free vars)
  2. **Hoist** nested, closed functions to the top level

- Remember:
  - Outer (yellow) function has no free variables.
    It is closed like all top-level functions.
  - Inner (purple) function has a single free variable `x`.

```
fun x →
  fun y →
    x + y
```

# Compiling Closures

- Compilers simplify closures via **closure conversion**.
- Transform a program
  - **from:** functions with nesting and free variables
  - **into**: equivalent program containing only top-level (and hence closed) functions
    - In this output, all functions can be represented as code pointers

- Two phases:
  1. **Close functions** by introducing environments (a dictionary for free vars)
  2. **Hoist** nested, closed functions to the top level

- Remember:
  - Ou
    It i
  - Inn

- E.g., `fun x -> (fun y -> y+x)` becomes, in C-like code:

```
closure *f1(env *env, int x) {
  env *e1 = extend(env,"x",x);
  closure *c =
       malloc(sizeof(closure));
  c->env = e1; c->fn = &f2;
  return c;
}
```

```
int f2(env *env, int y) {
  env *e1 = extend(env,"y",y);
  return lookup(e1, "y")
         + lookup(e1, "x");
}
```

# Compiling Closures

- **Phase 1: Closing functions.**
  - Represent function values (closures) as a pair: function pointer+environ.
  - **Step A: Make environment explicit and use it for free vars.**
    - Add a parameter representing the environment
    - Use it in the function's body to access free variables.
    - Example: Close inner (purple) function, which has free variable **x**.

```
fun x →
  fun y →
    x + y
```

- It doesn't make sense yet - where do these env's come from?
  - Function abstraction and application must be adapted:
    - function abstraction must **create** and initialize the closure and its **env**
    - function application must **use** the **env** and pass it as an additional parameter

# Compiling Closures

- **Phase 1: Closing functions**.
  - Represent function values (closures) as a pair: function pointer+environ.
  - **Step A: Make environment explicit and use it for free vars**.
    - Add a parameter representing the environment
    - Use it in the function's body to access free variables.
    - Example: Close inner (purple) function, which has free variable **x**.

```
fun x →
  fun (env, y) →
    let x = lookup env 0 in
      x + y
```

- It doesn't make sense yet - where do these env's come from?
  - Function abstraction and application must be adapted:
    - function abstraction must **create** and initialize the closure and its **env**
    - function application must **use** the **env** and pass it as an additional parameter

# Compiling Closures

- **Phase 1: Closing functions.**
  - Represent function values (closures) as a pair: function pointer+environ.
  - **Step A: Make environment explicit and use it for free vars.**
    - Add a parameter representing the environment
    - Use it in the function's body to access free variables.
    - Example: Close inner (purple) function, which has free variable **x**.

```
fun x →
  fun (env, y) →
    let x = lookup env 0 in
      x + y
```

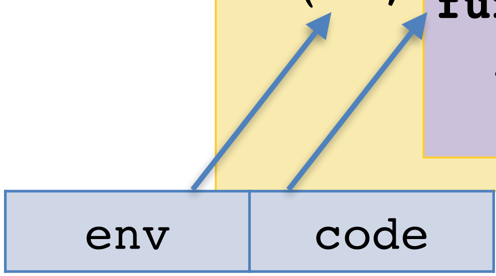
**x** is free, so get it from **env**

- It doesn't make sense yet - where do these env's come from?
  - Function abstraction and application must be adapted:
    - function abstraction must **create** and initialize the closure and its **env**
    - function application must **use** the **env** and pass it as an additional parameter

# Compiling Closures

- **Phase 1: Closing functions (continued).**
  - **Step A: Make environment explicit and use it for free vars.**
  - **Step B: Make all functions be pairs (env, lambda)**
  - **Step C: Make variable access via env**
- **Example**: `fun x → fun y → y+x` is converted to:

```
fun env x →
  let e' = extend env "x" x in
  (e', fun env y →
         let e' = extend env "y" y in
           (lookup e' "y") + (lookup e' "x")
  )
```




- Function abstraction and application are adapted:
  - function abstraction creates and initialize the closure and its env
  - function application extracts the environment and pass it as an additional parameter

# Compiling Closures

*Example: Let’s use it, by applying this to “4”*

```
( fun env x →
    let e’ = extend env “x” x in
    (e’, fun env y →
           let e’ = extend env “y” y in
             (lookup e’ “y”) + (lookup e’ “x”)
    )
  [] 4)
```

env | code

Reduces to:

```
([“x”:4], fun env y →
            let e’ = extend env “y” y in
              (lookup e’ “y”) + (lookup e’ “x”)
)
```

# Compiling Closures

- **Phase 2: Hoisting** (or "lambda lifting").
  - Move (now closed) nested anonymous functions to the top level
  - Give them an arbitrary (fresh) name
  - Replace original occurance with this new name
  - Now all functions are closed and top-level!
  - Can just represent them as code pointers (like in C)

```
fun env x →
  let e' = extend env "x" x in
  (e', fun env y →
         let e' = extend env "y" y in
           (lookup e' "y") + (lookup e' "x")
                                              )
```

# Compiling Closures

- **Phase 2: Hoisting** (or “lambda lifting”).
  - Move (now closed) nested anonymous functions to the top level
  - Give them an arbitrary (fresh) name
  - Replace original occurance with this new name
  - Now all functions are closed and top-level!
  - Can just represent them as code pointers (like in C)

```
fun env x →
  let e' = extend env "x" x in
  (e', fun env y →
         let e' = extend env "y" y in
           (lookup e' "y") + (lookup e' "x")
                                                )
```

```
fun env x →
  let e' = extend env "x" x in
  ( e', f02 )
```

# Compiling Closures

- **Phase 2: Hoisting** (or “lambda lifting”).
  - Move (now closed) nested anonymous functions to the top level
  - Give them an arbitrary (fresh) name
  - Replace original occurance with this new name
  - Now all functions are closed and top-level!
  - Can just represent them as code pointers (like in C)

```
fun env x →
  let e’ = extend env “x” x in
  (e’,
```

```
fun env y →
  let e’ = extend env “
    (lookup e’ “y”) + (
```

```
let f02 env y =
  let e’ = extend env “y” y in
 (lookup e’ “y”) + (lookup e’ “x”)
```

```
fun env x →
  let e’ = extend env “x” x in
  ( e’, f02 )
```

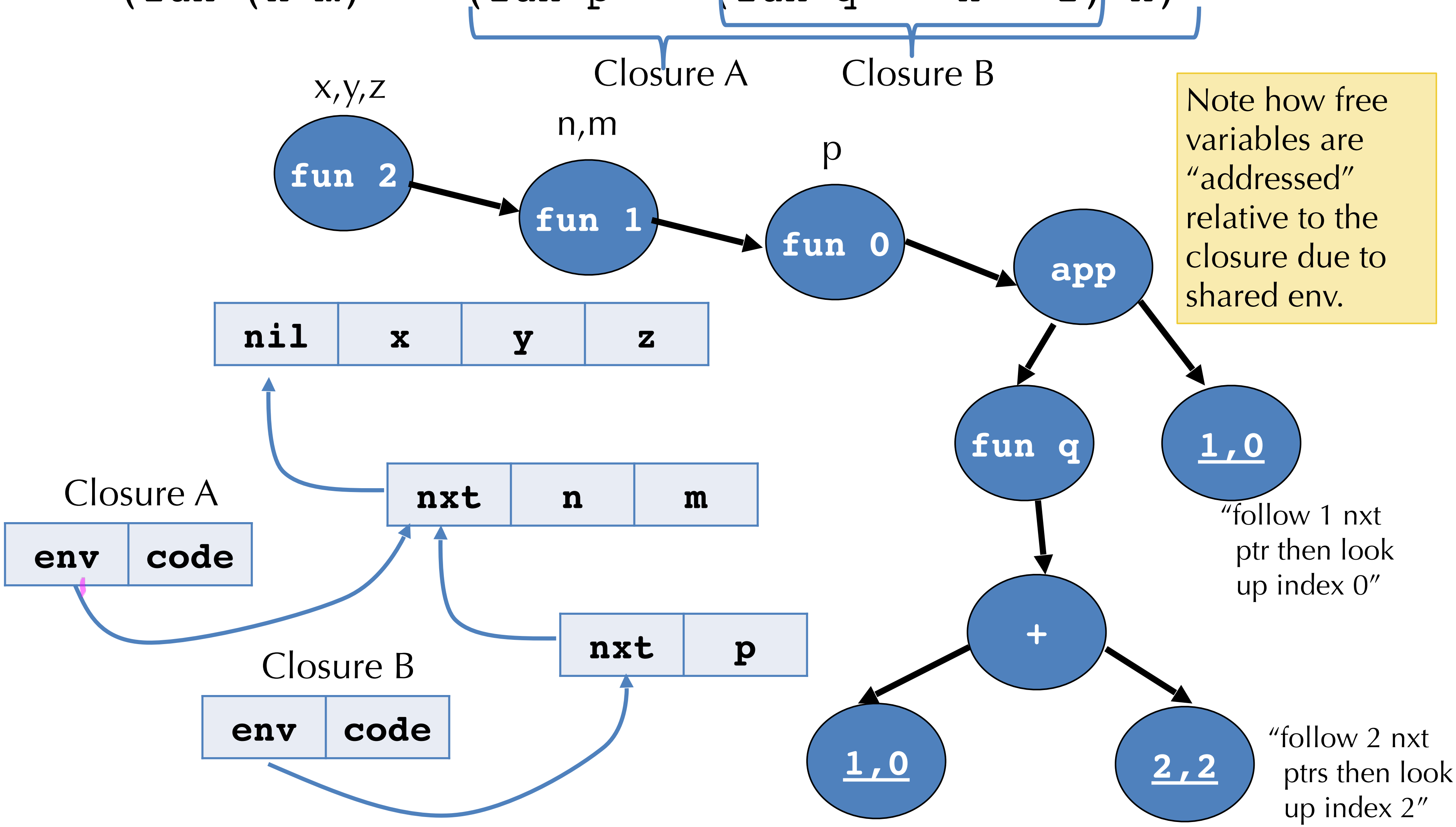
# Representing Closures

- Naïve closure conversion algorithm isn't very **efficient**:
  - It stores all the values for variables in the environment, even if they aren't needed by the function body.
  - It copies the environment values each time a nested closure is created.
  - It uses a linked-list datastructure for tuples.

- There are many options:
  - Store only the values for free variables in the body of the closure.
  - Share subcomponents of the environment to avoid copying
  - Use vectors or arrays rather than linked structures

# Array-based Closures with N-ary Functions

```
(fun (x y z) ->
   (fun (n m) -> (fun p -> (fun q -> n + z) x)
```



# CLOSURES

1. **Look at the closure-based Lambda Calculus interpreter in:**

```
open fun.ml
```

2. **Closure conversion (discussed in the next few slides):**

```
open cc.ml
```

# IR Support for Closures

- Need support for Tuples
  - Closure is a pair: (env, code)
  - Environment is a list of variables' values: (42, 9, 0)
    - Variables names, for example, (x, y, z)
    - Actually need **n-ary** tuples.
- Need global variables
  - Lifted lambdas are top-level

# cc.ml

```
module IR = struct
  type var = string
  type exp =
    | Val of value                   (* all values are expressions *)
    | Var of var                     (* local variables *)
    | Add of exp * exp               (* binary operations *)
    | App of exp * exp               (* application *)
    | Tuple of exp list
    | Nth of exp * int

    | Let of var * exp * exp         (* introduce local variables *)
    | Global of var                  (* global variables *)


  and value =
    | IntV of int
    | CodeV of var * var * exp         (* Environment name, arg name, body *)
    | TupleV of value list


  type environment = var list
```

# cc.ml

```
let rec convert env (e:Fun.exp) : exp =
  match e with
    | Fun.Int i -> Val (IntV i)
    | Fun.Add (e1, e2) -> Add (convert env e1, convert env e2)
    | Fun.Var x -> Var x
    | Fun.Fun (arg, body) ->
        let env_name = mk_tmp "ENV" in
        let body' = build_local_env env env_name (convert (arg::env) body) in
        let env' = build_closure_env env in
        Tuple [env'; Val(CodeV(env_name, arg, body'))]
    | Fun.App (e1, e2) -> App (convert env e1, convert env e2)

(* Top-level programs are closure-converted starting in an empty environment *)
let closure_convert = convert []
```

# cc.ml

Prologue

```
let rec convert env (e:Fun.exp) : exp =
  match e with
    | Fun.Int i -> Val (IntV i)
    | Fun.Add (e1, e2) -> Add (convert env e1, convert env e2)
    | Fun.Var x -> Var x
    | Fun.Fun (arg, body) ->
        let env_name = mk_tmp "ENV" in
        let body' = build_local_env env env_name (convert (arg::env) body) in
        let env' = build_closure_env env in
        Tuple [env'; Val(CodeV(env_name, arg, body'))]
    | Fun.App (e1, e2) -> App (convert env e1, convert env e2)

(* Top-level programs are closure-converted starting in an empty environment *)
let closure_convert = convert []
```

# cc.ml

```
fun (env, y) →
  let x = lookup env 0 in
    x + y
```

Prologue

```
let rec convert env (e:Fun.exp) : exp =
  match e with
    | Fun.Int i -> Val (IntV i)
    | Fun.Add (e1, e2) -> Add (convert env e1, convert env e2)
    | Fun.Var x -> Var x
    | Fun.Fun (arg, body) ->
        let env_name = mk_tmp "ENV" in
        let body' = build_local_env env env_name (convert (arg::env) body) in
        let env' = build_closure_env env in
        Tuple [env'; Val(CodeV(env_name, arg, body'))]
    | Fun.App (e1, e2) -> App (convert env e1, convert env e2)

(* Top-level programs are closure-converted starting in an empty environment *)
let closure_convert = convert []
```

# cc.ml

```
(* A function prologue that sets up the expected local variables. *)
let build_local_env env env_name body =
  let (_, code) =
    List.fold_left (fun (i, code) -> fun x ->
                      (i+1, Let(x, Nth(Var env_name, i), code)))
          (0, body) env
  in
    code
```

```
fun (env, y) →
  let x = lookup env 0 in
    x + y
```


```
let rec convert env (e:Fun.exp) : exp =
  match e with
    | Fun.Int i -> Val (IntV i)
    | Fun.Add (e1, e2) -> Add (convert env e1, convert env e2)
    | Fun.Var x -> Var x
    | Fun.Fun (arg, body) ->
        let env_name = mk_tmp "ENV" in
        let body' = build_local_env env env_name (convert (arg::env) body) in
        let env' = build_closure_env env in
        Tuple [env'; Val(CodeV(env_name, arg, body'))]
    | Fun.App (e1, e2) -> App (convert env e1, convert env e2)

(* Top-level programs are closure-converted starting in an empty environment *)
let closure_convert = convert []
```

# cc.ml

```
(* A function prologue that sets up the expected local variables. *)
let build_local_env env env_name body =
  let (_, code) =
    List.fold_left (fun (i, code) -> fun x ->
                      (i+1, Let(x, Nth(Var env_name, i), code)))
          (0, body) env
  in
    code
```

```
fun (env, y) →
  let x = lookup env 0 in
    x + y
```

Prologue

```
let rec convert env (e:Fun.exp) : exp =
  match e with
    | Fun.Int i -> Val (IntV i)
    | Fun.Add (e1, e2) -> Add (convert env e1, convert env e2)
    | Fun.Var x -> Var x
    | Fun.Fun (arg, body) ->
        let env_name = mk_tmp "ENV" in
        let body' = build_local_env env env_name (convert (arg::env) body) in
        let env' = build_closure_env env in
        Tuple [env'; Val(CodeV(env_name, arg, body'))]
    | Fun.App (e1, e2) -> App (convert env e1, convert env e2)

                     are closure-converted starting in an empty environment *)
                     convert []
```

**env'** you'll use if you want to invoke

# cc.ml

```
(* A function prologue that sets up the expected local variables. *)
let build_local_env env env_name body =
  let (_, code) =
    List.fold_left (fun (i, code) -> fun x ->
                      (i+1, Let(x, Nth(Var env_name, i), code)))
          (0, body) env
  in
    code
```

```
fun (env, y) →
  let x = lookup env 0 in
    x + y
```

Prologue

```
let rec convert env (e:Fun.exp) : ex =
  match e with
    | Fun.Int i -> Val (IntV i)
    | Fun.Add (e1, e2) -> Add (convert env e1, convert env e2)
    | Fun.Var x -> Var x
    | Fun.Fun (arg, body) ->
        let env_name = mk_tmp "ENV" in
        let body' = build_local_env env env_name (convert (arg::env) body) in
        let env' = build_closure_env env in
        Tuple [env'; Val(CodeV(env_name, arg, body'))]
    | Fun.App (e1, e2) -> App (convert env e1, convert env e2)

  are closure-converted starting in *)
  convert []
```

**env'** you'll use if you want to invoke

has Lets that consult **env_name** for values

341)

# cc.ml

```ocaml
(* A function prologue that sets up the expected local variables. *)
let build_local_env env env_name body =
  let (_, code) =
    List.fold_left (fun (i, code) -> fun x ->
                          (i+1, Let(x, Nth(Var env_name, i), code)))
          (0, body) env
  in
    code
```

```
fun (env, y) →
  let x = lookup env 0 in
    x + y
```

Prologue

```ocaml
let rec convert env (e:Fun.exp) : exp =
  match e with
    | Fun.Int i -> Val (IntV i)
    | Fun.Add (e1, e2) -> Add (convert env e1, convert env e2)
    | Fun.Var x -> Var x
    | Fun.Fun (arg, body) ->
        let env_name = mk_tmp "ENV" in
        let body' = build_local_env env env_name (convert (arg::env) body) in
        let env' = build_closure_env env in
        Tuple [env'; Val(CodeV(env_name, arg, body'))]
    | Fun.App (e1, e2) -> App (convert env e1, convert env e2)

  ... are clos ... ng in ... *)
  ... convert
```

**env'** you'll use if you want to invoke

fresh **env_name** for environment

has Lets that consult **env_name** for values

# cc.ml

```
(* A function prologue that sets up the expected local variables. *)
let build_local_env env env_name body =
  let (_, code) =
    List.fold_left (fun (i, code) -> fun x ->
                     (i+1, Let(x, Nth(Var env_name, i), code)))
          (0, body) env
  in
    code
```

```
let build_closure_env env =
  Tuple (List.map (fun x -> Var x) env)
```

```
let rec convert env (e:Fun.exp) : exp =
  match e with
    | Fun.Int i -> Val (IntV i)
    | Fun.Add (e1, e2) -> Add (convert env e1, convert env e2)
    | Fun.Var x -> Var x
    | Fun.Fun (arg, body) ->
        let env_name = mk_tmp "ENV" in
        let body' = build_local_env env env_name (convert (arg::env) body) in
        let env' = build_closure_env env in
        Tuple [env'; Val(CodeV(env_name, arg, body'))]
    | Fun.App (e1, e2) -> App (convert env e1, convert env e2)

(* Top-level programs are closure-converted starting in an empty environment *)
let closure_convert = convert []
```

# cc.ml

```
(* A function prologue that sets up the expected local variables. *)
let build_local_env env env_name body =
  let (_, code) =
    List.fold_left (fun (i, code) -> fun x ->
                          (i+1, Let(x, Nth(Var env_name, i), code)))
          (0, body) env
  in
    code
```

```
let build_closure_env env =
  Tuple (List.map (fun x -> Var x) env)
```

convert the "meta-level" environment to an "object-level" data structure. Here, an OCaml list is converted to an IL tuple.

```
let rec convert env (e:Fun.exp) : exp =
  match e with
    | Fun.Int i -> Val (IntV i)
    | Fun.Add (e1, e2) -> Add (convert env e1, convert env e2)
    | Fun.Var x -> Var x
    | Fun.Fun (arg, body) ->
        let env_name = mk_tmp "ENV" in
        let body' = build_local_env env env_name (convert (arg::env) body) in
        let env' = build_closure_env env in
        Tuple [env'; Val(CodeV(env_name, arg, body'))]
    | Fun.App (e1, e2) -> App (convert env e1, convert env e2)

(* Top-level programs are closure-converted starting in an empty environment *)
let closure_convert = convert []
```

```
let hoist e =
  let rec hoist_exp (e:exp):((var * value) list * exp) =
    match e with
      | Val(CodeV(env, x, body)) ->
          let (c1, r1) = hoist_exp body in
          let tmp = mk_tmp "CODE" in
            ((tmp, CodeV(env, x, r1))::c1, Global tmp)
      | Val(v) ->
          let (c1, r1) = hoist_val v in
            (c1, Val r1)
      | Var x -> ([], Var x)
      | Global x -> ([], Global x)
      | Add(e1, e2) ->
          let (c1, r1) = hoist_exp e1 in
          let (c2, r2) = hoist_exp e2 in
            (c1@c2, Add(r1, r2))
      | App(e1, e2) ->
          let (c1, r1) = hoist_exp e1 in
          let (c2, r2) = hoist_exp e2 in
            (c1@c2, App(r1, r2))
      | Let(x, e1, e2) ->
          let (c1, r1) = hoist_exp e1 in
          let (c2, r2) = hoist_exp e2 in
            (c1@c2, Let(x, r1, r2))
      | Tuple(elist) ->
          let (cs, rs) = List.split(List.map hoist_exp elist) in
            (List.concat cs, Tuple rs)
      | Nth(e1, i) ->
          let (c1, r1) = hoist_exp e1 in
            (c1, Nth(r1, i))

  and hoist_val v =
    match v with
      | IntV i -> ([], IntV i)
      | TupleV(vlist) ->
          let (cs, rs) = List.split(List.map hoist_val vlist) in
            (List.concat cs, TupleV rs)
      | _ -> failwith "impossible"
  in
    hoist_exp e
```

```
let hoist e =
  let rec hoist_exp (e:exp):((var * value) list * exp) =
    match e with
      | Val(CodeV(env, x, body)) ->
          let (c1, r1) = hoist_exp body in
          let tmp = mk_tmp "CODE" in
            ((tmp, CodeV(env, x, r1))::c1, Global tmp)
      | Val(v) ->
          let (c1, r1) = hoist_val v in
            (c1, Val r1)
      | Var x -> ([], Var x)
      | Global x -> ([], Global x)
      | Add(e1, e2) ->
          let (c1, r1) = hoist_exp e1 in
          let (c2, r2) = hoist_exp e2 in
            (c1@c2, Add(r1, r2))
      | App(e1, e2) ->
          let (c1, r1) = hoist_exp e1 in
          let (c2, r2) = hoist_exp e2 in
            (c1@c2, App(r1, r2))
      | Let(x, e1, e2) ->
          let (c1, r1) = hoist_exp e1 in
          let (c2, r2) = hoist_exp e2 in
            (c1@c2, Let(x, r1, r2))
      | Tuple(elist) ->
          let (cs, rs) = List.split(List.map hoist_exp elist) in
            (List.concat cs, Tuple rs)
      | Nth(e1, i) ->
          let (c1, r1) = hoist_exp e1 in
            (c1, Nth(r1, i))

  and hoist_val v =
    match v with
      | IntV i -> ([], IntV i)
      | TupleV(vlist) ->
          let (cs, rs) = List.split(List.map hoist_val vlist) in
            (List.concat cs, TupleV rs)
      | _ -> failwith "impossible"
  in
    hoist_exp e
```

returns a list of (fn001, closure)
as well as an revised expression

```
let hoist e =
  let rec hoist_exp (e:exp):((var * value) list * exp) =
    match e with
      | Val(CodeV(env, x, body)) ->
          let (c1, r1) = hoist_exp body in
          let tmp = mk_tmp "CODE" in
            ((tmp, CodeV(env, x, r1))::c1, Global tmp)
      | Val(v) ->
          let (c1, r1) = hoist_val v in
            (c1, Val r1)
      | Var x -> ([], Var x)
      | Global x -> ([], Global x)
      | Add(e1, e2) ->
          let (c1, r1) = hoist_exp e1 in
          let (c2, r2) = hoist_exp e2 in
            (c1@c2, Add(r1, r2))
      | App(e1, e2) ->
          let (c1, r1) = hoist_exp e1 in
          let (c2, r2) = hoist_exp e2 in
            (c1@c2, App(r1, r2))
      | Let(x, e1, e2) ->
          let (c1, r1) = hoist_exp e1 in
          let (c2, r2) = hoist_exp e2 in
            (c1@c2, Let(x, r1, r2))
      | Tuple(elist) ->
          let (cs, rs) = List.split(List.map hoist_exp elist) in
            (List.concat cs, Tuple rs)
      | Nth(e1, i) ->
          let (c1, r1) = hoist_exp e1 in
            (c1, Nth(r1, i))

  and hoist_val v =
    match v with
      | IntV i -> ([], IntV i)
      | TupleV(vlist) ->
          let (cs, rs) = List.split(List.map hoist_val vlist) in
            (List.concat cs, TupleV rs)
      | _ -> failwith "impossible"
  in
    hoist_exp e
```

returns a list of (fn001, closure) as well as an revised expression

interesting case: make a name, move the function to the toplevel list, resulting expression is just a Global

```
let hoist e =
  let rec hoist_exp (e:exp):((var * value) list * exp) =
    match e with
      | Val(CodeV(env, x, body)) ->
          let (c1, r1) = hoist_exp body in
          let tmp = mk_tmp "CODE" in
            ((tmp, CodeV(env, x, r1))::c1, Global tmp)
      | Val(v) ->
          let (c1, r1) = hoist_val v in
            (c1, Val r1)
      | Var x -> ([], Var x)
      | Global x -> ([], Global x)
      | Add(e1, e2) ->
          let (c1, r1) = hoist_exp e1 in
          let (c2, r2) = hoist_exp e2 in
            (c1@c2, Add(r1, r2))
      | App(e1, e2) ->
          let (c1, r1) = hoist_exp e1 in
          let (c2, r2) = hoist_exp e2 in
            (c1@c2, App(r1, r2))
      | Let(x, e1, e2) ->
          let (c1, r1) = hoist_exp e1 in
          let (c2, r2) = hoist_exp e2 in
            (c1@c2, Let(x, r1, r2))
      | Tuple(elist) ->
          let (cs, rs) = List.split(List.map hoist_exp elist) in
            (List.concat cs, Tuple rs)
      | Nth(e1, i) ->
          let (c1, r1) = hoist_exp e1 in
            (c1, Nth(r1, i))

  and hoist_val v =
    match v with
      | IntV i -> ([], IntV i)
      | TupleV(vlist) ->
          let (cs, rs) = List.split(List.map hoist_val vlist) in
            (List.concat cs, TupleV rs)
      | _ -> failwith "impossible"
  in
    hoist_exp e
```

returns a list of (fn001, closure) as well as an revised expression

interesting case: make a name, move the function to the toplevel list, resulting expression is just a Global

just recursively collect toplevels

# Compiling Closures to LLVM IR

- The "types" of the environment data structures are generic tuples
  - The tuples contain a mix of int and closure values
  - We know statically what the tuple-type of the environment should be
  - LLVM IR doesn't have generic types

- Type translations:
  - ⟦ - ⟧ for "intepretation" that retains type information
    ⟦int⟧ = i64
    ⟦(t1, …, tn)⟧ = {⟦t1⟧, …, ⟦tn⟧}*
    ⟦t1 → t2⟧ = $⟦t1 \to t2⟧_C$
  - $⟦t1 \to t2⟧_C$ = {i8*, ((i8*, ⟦t1⟧) → ⟦t2⟧)*}* "Closure Representation"

- Rough sketch:
  - Allocation & uses of objects us the "interpretation" translation
  - Anywhere an environment is passed or stored, use i8* and bitcast to/from the translation type.

# Currying

- We just saw a way for a function to take multiple arguments!
  - The function consumes one argument and returns a function that takes the rest
- This is called currying the function
  - Named after the logician Haskell B. Curry
  - But Schönfinkel and Frege discovered it
    - So it should probably be called Schönfinkelizing or Fregging

Scope, Types, and Context

# STATIC ANALYSIS I: SCOPE CHECKING

# Variable Scoping

- Consider the problem of determining whether a programmer-declared variable is in scope.

- Issues:
  - Which variables are available at a given point in the program?
  - Shadowing – is it permissible to re-use the same identifier, or is it an error?

- Example: The following program is syntactically correct but not well-formed. (y and q are used without being defined anywhere)

```
int fact(int x) {
  var acc = 1;
  while (x > 0) {
    acc = acc * y;
    x = q - 1;
  }
  return acc;
}
```

Q: Can we solve this problem by changing the parser to rule out such programs?

# Contexts and Inference Rules

- Need to keep track of contextual information
  - What variables are in scope?
  - What are their types?
- How do we describe this?
  - In the compiler, there's a mapping from *variables* ***to*** *information we know* about them.
- ***Inference rules***:

Premises

$$\frac{G \vdash e_1 \qquad G \vdash e_2}{G \vdash e_1 \; e_2}$$

Conclusion

# Scope-Checking Lambda Calculus

- Consider how to identify "well-scoped" lambda calculus terms
  - Recall the free variable calculation
  - Given: G, a set of variable identifiers, e, a term of the lambda calculus
  - *Judgment*: $G \vdash e$ means "the free variables of e are included in G": $fv(e) \subseteq G$

| | | | |
|---|---|---|---|
| $fv(x)$ | = | $\{x\}$ | |
| $fv(\text{fun } x \rightarrow exp)$ | = | $fv(exp) \setminus \{x\}$ | *('x' is a bound in exp)* |
| $fv(exp_1\ exp_2)$ | = | $fv(exp_1) \cup fv(exp_2)$ | |

$$\frac{x \in G}{G \vdash x}$$

"the variable x is free"

$$\frac{G \vdash e_1 \qquad G \vdash e_2}{G \vdash e_1\ e_2}$$

"G contains the free variables of $e_1$ and $e_2$"

$$\frac{G \cup \{x\} \vdash e}{G \vdash \text{fun } x \rightarrow e}$$

"x is available in the function body e"

# Scope-checking Code

- Compare the OCaml code to the inference rules:
  - structural recursion over syntax
  - the check either "succeeds" or "fails"

```
let rec scope_check (g:VarSet.t) (e:exp) : unit =
  begin match e with
    | Var x -> if VarSet.member x g then () else failwith (x ^ "not in scope")
    | App(e1, e2) -> ignore (scope_check g e1); scope_check g e2
    | Fun(x, e) -> scope_check (VarSet.union g (VarSet.singleton x)) e
  end
```

fun.ml - modules Fun and Eval1

$$\frac{x \in G}{G \vdash x} \qquad \frac{G \vdash e_1 \qquad G \vdash e_2}{G \vdash e_1\ e_2} \qquad \frac{G \cup \{x\} \vdash e}{G \vdash \text{fun}\ x \rightarrow e}$$