Closure Conversion

CIS400 (Compiler Construction)

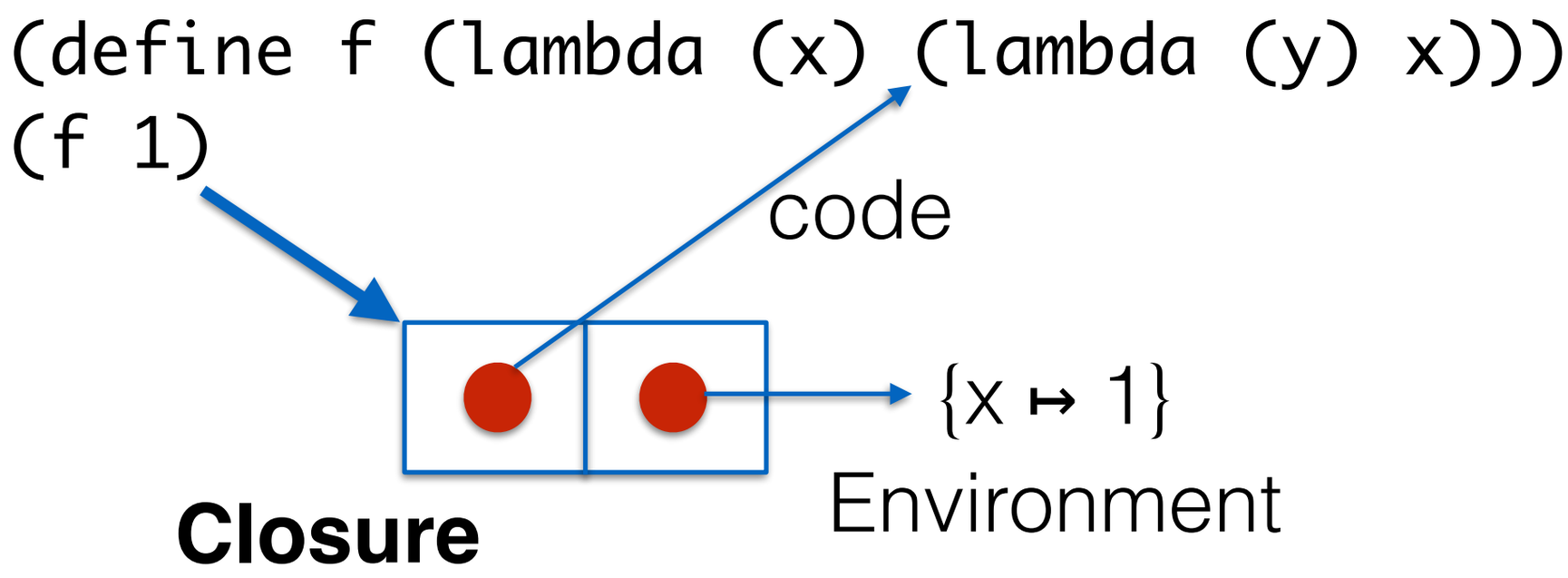
Kris Micinski, Fall 2021

Our output language from project p3

In this lecture, we will talk about removing **lambda**

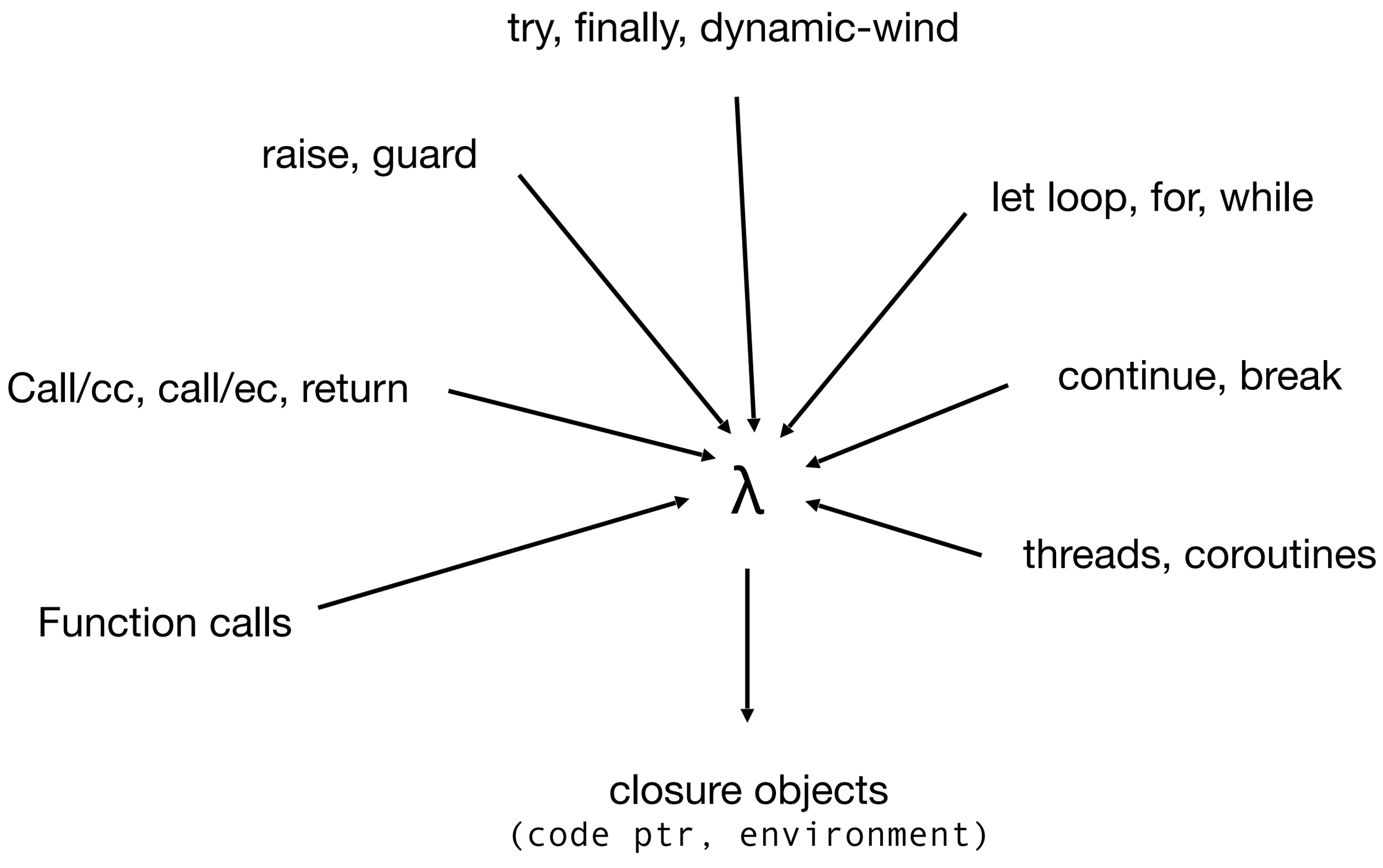
```
; e ::= (let ([x (apply-prim op ae)]) e)
;      | (let ([x (prim op ae ...)]) e)
;      | (let ([x (lambda x e)]) e)
;      | (let ([x (lambda (x ...) e)]) e)
;      | (let ([x (quote dat)]) e)
;      | (apply ae ae)
;      | (ae ae ...)
;      | (if ae e e)
; ae ::= (lambda (x ...) e)
;      | (lambda x e)
;      | x
;      | (quote dat)
```

As we have discussed previously, functional programming languages use **closures** to represent lambdas at runtime



Closures are **code + data**

In this course, we have compiled most serious language features to lambdas



Closure conversion

- In strict CPS, lambdas no longer return.
- Function calls are just first-class `goto`'s that take arguments and carry values for free variables in an environment.
- Closure conversion:
  - Hoists all functions to the top-level.
  - Allocates closures that save the current environment.
  - Function calls explicitly pass the closure's env.
  - Replaces references to free variables with env access.

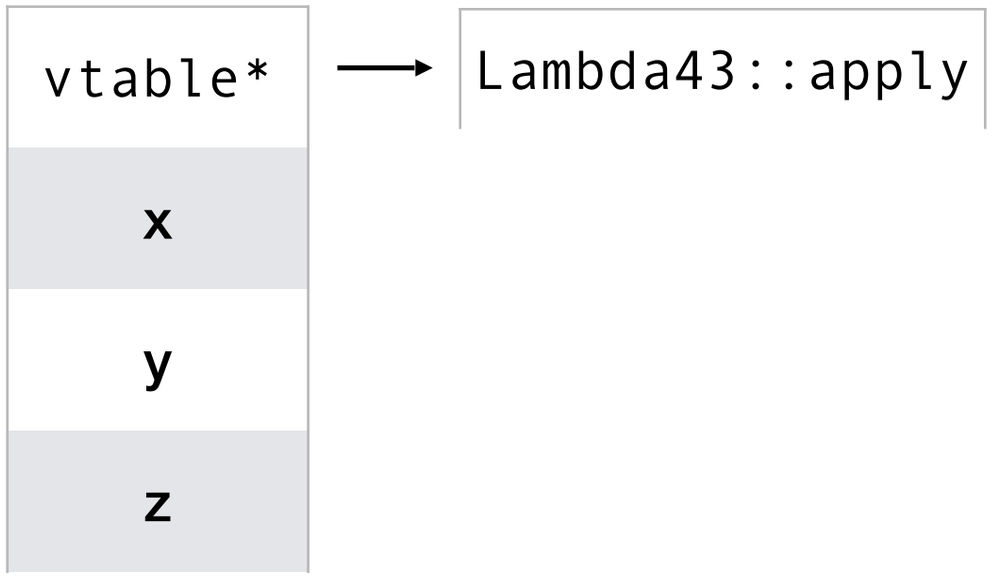
At a high level: to perform closure conversion, we calculate the free variables necessary to keep around, and turn lambdas into allocation sites for closures

`(λ (a b) …)` ⟶ `new Lambda43(x,y,z);`

$\mathcal{FV}$`((λ (a b) …))`
= `{x,y,z}`

```
class Lambda43 : public Lam
{
private:
  const u64 x,y,z;
public:
  Lambda43(u64 x, u64 y, u64 z)
    : x(x), y(y), z(z)
  {}

  u64 apply(u64 a, u64 b)
  {
    …
  }
};
```




```
class Lambda43 : public Lam
{
private:
  const u64 x,y,z;
public:
  Lambda43(u64 x, u64 y, u64 z)
    : x(x), y(y), z(z)
  {}

  u64 apply(u64 a, u64 b)
  {
    …
  }
};
```

Closure conversion:

two principal strategies

- Top-down closure conversion:
    - Traverses the AST and turns lambdas into closures on the way down (computing *environments* as it goes).
    - Produces ***linked environments/closures***.
        - Pros: compact, shared environments; Cons: generates slower code (lookup is O(n)).
- Bottom-up closure conversion:
    - Traverses the AST and turns lambdas into closures on the way back up (computing ***free vars*** as it goes).
    - Produces ***flat environments/closures***.
    - Pros: fast, computes free vars; Cons: more copying, closure allocation is slower

```
…
   (λ (a)
        …


                    (λ (b c)
                         …


                                   (λ (d)
                                        …


                                              (f a c)


…) …) …) …
```

Top-down closure conversion

- As the AST is traversed, a set of locally bound variables are computed.
- A map from non-locally bound variables to their access expressions is maintained so that variables can be looked up in an environment.
- At each lambda: the algorithm lifts the lambda to a first-order C-like procedure, allocates a new closure and its environment, then converts it’s body using an updated map for non-locally bound variables (with paths into this newly defined environment).
- Previously allocated environment is linked-to/shared.

…

(λ (a)

…

bound vars: x

(λ (b c)

…

(λ (d)

…

(f a c)

…) …) …) …

```
(proc (main)
 …
  (vector
     lam0
     (prim vector
       x))
 …)
```

env mapping:

```
x -> (vector-ref env0 ‘0)
```

```
(proc (lam0 env0 a)

  …
   (λ (b c)
     …
        (λ (d)
           …
              (f a c)
   …) …) …)
```

```
(proc (main)
 …
  (prim vector
     lam0
     (prim vector
       x))
 …)
```

bound vars: `env0,a,y`

env mapping:

```
x    -> (vector-ref env0 ‘0)
env0 -> (vector-ref env1 ‘0)
a    -> (vector-ref env1 ‘1)
y    -> (vector-ref env1 ‘2)
```

```
(proc (lam0 env0 a)

  …
   (prim vector
         lam1
         (prim vector
               env0
               a
               y))
     …)
```

```
(proc (lam1 env1 b c)

    …
      (λ (d)
        …
          (f a c)

    …) …)
```

Bottom-up closure conversion

- As the AST is traversed, free variables are computed.
- At each lambda: 1) the algorithm converts any lambdas under the lambda's body first (and also computes a set of free variables); then 2) it emits code to allocate the lambda's closure/environment and replaces free vars with env access.
- Converting the body of a lambda yields a set of free variables that can be *canonically ordered*.
- Closures are *flat* heap-allocated vectors containing a function pointer and then each free var *in order*.
- Accesses of free variables are turned into a `vector-ref` with the predetermined index.

…

(λ (a)

…

(λ (b c)

…

(λ (d)

…

free vars: a , c , f

(f a c)

…) …) …) …

```
…
   (λ (a)
      …
```

```
                (λ (b c)
                   …
```

```
(proc (lam14 env d)
  …
  (clo-app
   (prim vector-ref
         env ‘3)    ;f
   (prim vector-ref
         env ‘1)    ;a
   (prim vector-ref
         env ‘2))   ;c
   …)
```

adds first-order proc

```
(prim vector
      lam14
      a
      c
      f)
```

allocates flat closure

```
…) …) …) …
```

```
…
   (λ (a)
        …
```

```
(λ (b c)
     …
```

free vars: a f x y

```
(prim vector
      lam14
      a
      c
      f)
```

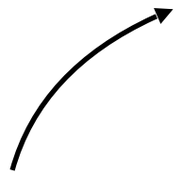
references at closure allocation can remain free

```
…) …) …) …
```

```
(clo-app
   (prim vector-ref
         env ‘3)    ;f
   (prim vector-ref
         env ‘1)    ;a
   (prim vector-ref
         env ‘2))   ;c
```



```
(let ([f-clo (prim vector-ref env ‘3)])
  (let ([f-ptr (prim vector-ref f-clo ‘0)])
    (let ([a (prim vector-ref env ‘1)])
      (let ([c (prim vector-ref env ‘2)])
        (C-style-call f-ptr f-clo a c)))))
```



application: 1) function pointer is accessed from closure

2) closure (`f-clo`) is passed to invoked function ptr

Let’s live code bottom-up closure conversion.

```
; Input Language (simplified cps):
; e ::= (let ([x 'dat]) e)
;      | (let ([x (prim op x ...)]) e)
;      | (let ([x (lambda (x ...) e)]) e)
;      | (if x e e)
;      | (x x ...)
```

Our output language is a list of C-style procedures

```
; Output Language (procedural):
; p ::= ((proc (x x ...) e) ...)
; e ::= (let ([x 'dat]) e)
;      | (let ([x (prim op x ...)]) e)
;      | (let ([x (make-closure x x ...)]) e)
;      | (let ([x (env-ref x nat)]) e)
;      | (if x e e)
;      | (clo-app x x ...)
```

```
(define (closure-convert simple-cps)
 ;; Exp x List[Proc] -> Exp x Set[Var] x List[Proc]
 (define (bottom-up e procs)
    (match e
       …))
;; returns the top expression, set of free variables
;; and list of sub-procs
(match-define `(,main-body ,free ,procs)
  (bottom-up simple-cps '()))
;; return the list of procs & main expression proc
`((proc (main) ,main-body) . ,procs))
```

```
(define (closure-convert simple-cps)
 ;; Exp x List[Proc] -> Exp x Set[Var] x List[Proc]
 (define (bottom-up e procs)
    (match e
      [`(let ([,x ',dat]) ,e0)
(match-define `(,e0+ ,free+ ,procs+)
              (bottom-up e0 procs))
`((let ([,x ',dat]) ,e0+)
  ,(set-remove free+ x)
  ,procs+)]
```

```
(define (closure-convert simple-cps)
 ;; Exp x List[Proc] -> Exp x Set[Var] x List[Proc]
 (define (bottom-up e procs)
    (match e
      …
    [`(let ([,x (prim ,op ,xs ...)]) ,e0)
     (match-define `(,e0+ ,free+ ,procs+)
                   (bottom-up e0 procs))
     `((let ([,x (prim ,op ,@xs)]) ,e0+)
       ,(set-remove (set-union free+ (list->set xs)) x)
       ,procs+)]
```

```
(define (closure-convert simple-cps)
 ;; Exp x List[Proc] -> Exp x Set[Var] x List[Proc]
 (define (bottom-up e procs)
    (match e
      …
      [`(let ([,x (lambda (,xs ...) ,body)]) ,e0)
       (match-define `(,e0+ ,free0+ ,procs0+)
                     (bottom-up e0 procs))
       (match-define `(,body+ ,freelam+ ,procs1+)
                     (bottom-up body procs0+))
       (define env-vars (foldl (lambda (x fr) (set-remove fr x))
                               freelam+
                               xs))
       (define ordered-env-vars (set->list env-vars))
       (define lamx (gensym 'lam))
       (define envx (gensym 'env))
       (define body++ (cdr (foldl (lambda (x count+body)
                                    (match-define (cons cnt bdy) count+body)
                                     (cons (+ 1 cnt)
                                           `(let ([,x (env-ref ,envx ,cnt)])
                                              ,bdy)))
                                  (cons 1 body+)
                                  ordered-env-vars)))
       `((let ([,x (make-closure ,lamx ,@ordered-env-vars)]) ,e0+)
         ,(set-remove (set-union free0+ env-vars) x)
         ((proc (,lamx ,envx ,@xs) ,body++) . ,procs1+))]
```

```
(define (closure-convert simple-cps)
 ;; Exp x List[Proc] -> Exp x Set[Var] x List[Proc]
 (define (bottom-up e procs)
    (match e
      …
      [`(if ,(? symbol? x) ,e0 ,e1)
       (match-define `(,e0+ ,free0+ ,procs0+)
                     (bottom-up e0 procs))
       (match-define `(,e1+ ,free1+ ,procs1+)
                     (bottom-up e1 procs0+))
       `((if ,x ,e0+ ,e1+)
         ,(set-union free1+ free0+ (set x))
         ,procs1+)]
```

```
(define (closure-convert simple-cps)
 ;; Exp x List[Proc] -> Exp x Set[Var] x List[Proc]
 (define (bottom-up e procs)
    (match e
    …
    [`(,(? symbol? xs) ...)
     `((clo-app ,@xs)
       ,(list->set xs)
       ,procs)])
```

```
(define (closure-convert simple-cps)
  ; Exp x List[Proc] -> Exp x Set[Var] x List[Proc]
  (define (bottom-up e procs)
    (match e
      [`(let ([,x ',dat]) ,e0)
       (match-define `(,e0+ ,free+ ,procs+)
                     (bottom-up e0 procs))
       `((let ([,x ',dat]) ,e0+)
         ,(set-remove free+ x)
         ,procs+)]
      [`(let ([,x (prim ,op ,xs ...)]) ,e0)
       (match-define `(,e0+ ,free+ ,procs+)
                     (bottom-up e0 procs))
       `((let ([,x (prim ,op ,@xs)]) ,e0+)
         ,(set-remove (set-union free+ (list->set xs)) x)
         ,procs+)]
      [`(let ([,x (lambda (,xs ...) ,body)]) ,e0)
       (match-define `(,e0+ ,free0+ ,procs0+)
                     (bottom-up e0 procs))
       (match-define `(,body+ ,freelam+ ,procs1+)
                     (bottom-up body procs0+))
       (define env-vars (foldl (lambda (x fr) (set-remove fr x))
                               freelam+
                               xs))
       (define ordered-env-vars (set->list env-vars))
       (define lamx (gensym 'lam))
       (define envx (gensym 'env))
       (define body++ (cdr (foldl (lambda (x count+body)
                                    (match-define (cons cnt bdy) count+body)
                                    (cons (+ 1 cnt)
                                          `(let ([,x (env-ref ,envx ,cnt)])
                                             ,bdy)))
                                  (cons 1 body+)
                                  ordered-env-vars)))
       `((let ([,x (make-closure ,lamx ,@ordered-env-vars)]) ,e0+)
         ,(set-remove (set-union free0+ env-vars) x)
         ((proc (,lamx ,envx ,@xs) ,body++) . ,procs1+))]
      [`(if ,(? symbol? x) ,e0 ,e1)
       (match-define `(,e0+ ,free0+ ,procs0+)
                     (bottom-up e0 procs))
       (match-define `(,e1+ ,free1+ ,procs1+)
                     (bottom-up e1 procs0+))
       `((if ,x ,e0+ ,e1+)
         ,(set-union free1+ free0+ (set x))
         ,procs1+)]
      [`(,(? symbol? xs) ...)
       `((clo-app ,@xs)
         ,(list->set xs)
         ,procs)]))
  (match-define `(,main-body ,free ,procs) (bottom-up simple-cps '()))
  `((proc (main) ,main-body) . ,procs))
```

```
(define test0 '(let ([a '3])
                 (let ([f (lambda (b) (let ([c (prim + a b)])
                                        (let ([_ (prim halt c)])
                                          (_ _))))])
                   (f a))))
```