SICP section 5.5

April 18, 2008 at 12:17 Tags SICP

It took me some effort to get the compiler code to work in conjunction with the explicit control evaluator, especially when it came to linking compiled and interpreted code. The prime problem was additions I've made to the register machine simulator in previous chapters, in answer to exercises, and that didn't work well with the needs of the compiler.

For example, in one of the exercises it was requested to test whether operations are being done on labels, and flag these cases as errors. However, in `compile-lambda`, the compiler creates this assignment:

```
`((assign ,target
    (op make-compiled-procedure)
    (label ,proc-entry)
    (reg env)))))
```

Note that it acts upon a label. So, I had to modify `make-operation-exp` and remove the error caused by operations on labels.

Another place where a change was needed is in `lookup-variable-value`, which, to answer an exercise, was modified to prepend `bound` or `unbound` to variables that were looked up. So I modified the code of `compile-variable` to:

```
(define (compile-variable exp target linkage)
  (end-with-linkage linkage
    (make-instruction-sequence '(env) (list target)
      `((assign ,target
                (op lookup-variable-value)
                (const ,exp)
                (reg env))
        (assign ,target
                (op var-val-extract-value)
                (reg ,target))))))
```

The addition is another `assign` which uses `var-val-extract-value` to extract the bound symbol values. As before the exercise, this code assumes that all the looked up variables are actually bound, and does not signal an error[1].

All the code required to run the compiler and execute the compiled code together with the explicit control evaluator on the register machine simulator can be downloaded in this bundle

In case you've missed it, consider what we have here. We have a compiler for Scheme, written in Scheme. This compiler spits out custom assembly code that executes on a register machine simulator, also written in Scheme. Moreover, this code can coexist with an interpreter for Scheme, written in the same assembly code. Isn't this just awesome ?

Anyway, since we now have all the infrastructure running, we're ready to tackle the exercises.

Exercise 5.31

- `(f 'x 'y)` – all those saves and restores are superfluous, because looking up the operator and operands does not modify any registers.
- `((f) 'x 'y)` – again, all the saves are superfluous. At first sight it might seem that `(f)` may modify the `env` register, so `env` must be saved. But note that `env` is not needed to evaluate the operands, since they are constants. Had the expression been `((f) x 'y)`, `env` should've been saved, because it's needed to evaluate the variable `x`.
- `(f (g 'x) y)` – `proc` must be saved around the evaluation of the first operand, because it involves an application, which modifies `proc`. The application also modifies `argl`, so that one has to be saved too. `env`,

however, needs not be saved. To understand why, recall that the compiler builds the argument list in reverse. So first it evaluates `y`, and only then `(g 'x)`. Evaluating `y` does not modify `env`, so saving it isn't required.

- `(f (g 'x) 'y)` – exactly similar to the previous case. Since the argument list is built in reverse `(g 'x)` is constructed last, and there's really no difference in the effects of looking up `y` or `'y` on the registers.

Exercise 5.32

a.

The section beginning with `ev-application` has to be modified to:

```
ev-application
  (save continue)
  (assign unev (op operands) (reg exp))
  (assign exp (op operator) (reg exp))
  (test (op symbol?) (reg exp))
  (branch (label ev-appl-operator-symbol))
  (save env)
  (save unev)
  (assign continue (label ev-appl-did-operator))
  (goto (label eval-dispatch))
ev-appl-operator-symbol
  (assign continue (label ev-appl-did-operator-no-restore))
  (goto (label eval-dispatch))
ev-appl-did-operator
  (restore unev)              ; the operands
  (restore env)
ev-appl-did-operator-no-restore
  (assign argl (op empty-arglist))
  (assign proc (reg val))     ; the operator
  (test (op no-operands?) (reg unev))
  (branch (label apply-dispatch))
  (save proc)
```

Note the test for `symbol?` that jumps over the saves, and assigns `continue` to a label that skips the restores.

b.

Alyssa is wrong for two reasons.

1. Note that the test and branch inserted in the code of part **a.** does not come for free – it has to be executed for each operator, so the performance increase is less than expected. The compiler does its optimizations at compile-time, investing the time once but causing the code to run faster every time it's executed.
2. Another major reason for the compiler's speed advantage over the interpreter is "parsing" of expressions. The evaluator has to take the expression `(+ 1 2)` apart each time, figuring out that `+` is the operator and 1 and 2 the operands. The compiler does this step just once, and generates code that already knows where everything is located.

Exercise 5.33

The difference in the compiled code of the alternative is that it handles the invocation of the recursive call before evaluating `n` in the multiplication, while the original code first evaluates `n`.

Since the amount of instruction executed by each version of the compiled code is the same, I would expect the runtimes to be close to identical.

Exercise 5.34

Here's the compiled code, annotated with comments:

```
;; Construct the procedure and skip over code for
;; the procedure body
;;
  (assign val (op make-compiled-procedure) (label entry1) (reg env))
  (goto (label after-lambda2))
;; Entry point for calls to factorial
;;
entry1
  (assign env (op compiled-procedure-env) (reg proc))
```

```
  (assign env (op compiled-procedure-env) (reg proc))
  (assign env (op extend-environment) (const (n)) (reg argl) (reg env))
;; Construct the internal procedure 'iter'
;;
  (assign val (op make-compiled-procedure) (label entry3) (reg env))
  (goto (label after-lambda4))
entry3
  (assign env (op compiled-procedure-env) (reg proc))
  (assign env (op extend-environment) (const (product counter)) (reg argl) (reg env))
  (save continue)
  (save env)
;; evaluate the 'if' predicate: (> counter n)
;;
  (assign proc (op lookup-variable-value) (const >) (reg env))
  (assign proc (op var-val-extract-value) (reg proc))
  (assign val (op lookup-variable-value) (const n) (reg env))
  (assign val (op var-val-extract-value) (reg val))
  (assign argl (op list) (reg val))
  (assign val (op lookup-variable-value) (const counter) (reg env))
  (assign val (op var-val-extract-value) (reg val))
  (assign argl (op cons) (reg val) (reg argl))
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch8))
compiled-branch9
  (assign continue (label after-call10))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch8
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
after-call10
  (restore env)
  (restore continue)
;; Test the 'if' predicate
;;
  (test (op false?) (reg val))
  (branch (label false-branch6))
;; In the true branch, evaluate and return 'product'
;;
true-branch5
  (assign val (op lookup-variable-value) (const product) (reg env))
  (assign val (op var-val-extract-value) (reg val))
  (goto (reg continue))
;; In the false branch, evaluate the application
;; of iter.
;;
false-branch6
  (assign proc (op lookup-variable-value) (const iter) (reg env))
  (assign proc (op var-val-extract-value) (reg proc))
  (save continue)
  (save proc)
  (save env)
;; Evaluate (+ counter 1)
;;
  (assign proc (op lookup-variable-value) (const +) (reg env))
  (assign proc (op var-val-extract-value) (reg proc))
  (assign val (const 1))
  (assign argl (op list) (reg val))
  (assign val (op lookup-variable-value) (const counter) (reg env))
  (assign val (op var-val-extract-value) (reg val))
  (assign argl (op cons) (reg val) (reg argl))
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch14))
compiled-branch15
  (assign continue (label after-call16))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
```

```
primitive-branch14
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
after-call16
  (assign argl (op list) (reg val))
  (restore env)
  (save argl)
;; Evaluate (* counter product)
;;
  (assign proc (op lookup-variable-value) (const *) (reg env))
  (assign proc (op var-val-extract-value) (reg proc))
  (assign val (op lookup-variable-value) (const product) (reg env))
  (assign val (op var-val-extract-value) (reg val))
  (assign argl (op list) (reg val))
  (assign val (op lookup-variable-value) (const counter) (reg env))
  (assign val (op var-val-extract-value) (reg val))
  (assign argl (op cons) (reg val) (reg argl))
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch11))
compiled-branch12
  (assign continue (label after-call13))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch11
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
after-call13
  (restore argl)
  (assign argl (op cons) (reg val) (reg argl))
  (restore proc)
  (restore continue)
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch17))
;; The "recursive" call is done here.
;;
compiled-branch18
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch17
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
  (goto (reg continue))
after-call19
after-if7
;; After the body of the internal procedure,
;; we can compile the rest of the body of
;; 'factorial'
;;
after-lambda4
;; Assign the variable 'iter' its procedure body
;;
  (perform (op define-variable!) (const iter) (reg val) (reg env))
  (assign val (const ok))
;; Execute (iter 1 1)
;;
  (assign proc (op lookup-variable-value) (const iter) (reg env))
  (assign proc (op var-val-extract-value) (reg proc))
  (assign val (const 1))
  (assign argl (op list) (reg val))
  (assign val (const 1))
  (assign argl (op cons) (reg val) (reg argl))
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch20))
compiled-branch21
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch20
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
  (goto (reg continue))
```

```
  (goto (reg continue))
after-call22
;; Now assign
;;
after-lambda2
  (perform (op define-variable!) (const factorial) (reg val) (reg env))
  (assign val (const ok))
```

Note that nothing is saved before the recursive call to `iter`, in order to be restored later. It just jumps back to `iter` using `(goto (reg val))`. Contrary to the recursive case, where `continue` was assigned to just after the recursive call before executing it, here `continue` is just restored from the stack to its pre-call value. This is the essence of tail-call optimization.

Exercise 5.35

Following the code carefully, and remembering that the argument list is being built from right to left, it's not difficult to see that the Scheme code that compiled into this was:

```
(define (f x)
  (+ x (g (+ x 2))))
```

Exercise 5.36

The compiler produces right-to-left order of evaluation for operands of a combinations. This can be easily seen in the compiled code for `(+ x y)`:

```
  (assign proc (op lookup-variable-value) (const +) (reg env))
  (assign proc (op var-val-extract-value) (reg proc))
  (assign val (op lookup-variable-value) (const y) (reg env))
  (assign val (op var-val-extract-value) (reg val))
  (assign argl (op list) (reg val))
  (assign val (op lookup-variable-value) (const x) (reg env))
  (assign val (op var-val-extract-value) (reg val))
  (assign argl (op cons) (reg val) (reg argl))
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch1))
compiled-branch2
  (assign continue (label after-call3))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch1
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
after-call3
```

Note the order of `lookup-variable-value` calls. `y` is first, `x` second.

The reason for this is also clear and was thoroughly explained in section 5.5.3 – "Compiling Combinations"

> The code to construct the argument list will evaluate each operand into val and then cons that value onto the argument list being accumulated in argl. Since we cons the arguments onto argl in sequence, we must start with the last argument and end with the first, so that the arguments will appear in order from first to last in the resulting list.

In the compiler code, the culprit is function `construct-arglist`, which calls `reverse` on the list of operand codes before generating the actual code for them.

In order to modify the order to left-to-right, we'll need to move the `reverse` from compile-time to run-time. The instructions will be generated in left-to-right order, and `reverse` will be called on `argl` afterwards. Here's the modified `construct-arglist` that does the job:

```
(define (construct-arglist operand-codes)
  (let ((operand-codes operand-codes))
    (if (null? operand-codes)
      (make-instruction-sequence '() '(argl)
        `((assign argl (const ())))
      (let ((code-to-get-last-arg
              (append-instruction-sequences
                (car operand-codes)
                (make-instruction-sequence '(val) '(argl)
                  `((assign argl (op list) (reg val)))))))
        (if (null? (cdr operand-codes))
          code-to-get-last-arg
          (tack-on-instruction-sequence
            (preserving '(env)
              code-to-get-last-arg
              (code-to-get-rest-args
                (cdr operand-codes)))
            (make-instruction-sequence '() '()
              '((assign argl (op reverse) (reg argl)))))))))
```

Now the compiled code for `(+ x y)` will be:

```
  (assign proc (op lookup-variable-value) (const +) (reg env))
  (assign proc (op var-val-extract-value) (reg proc))
  (assign val (op lookup-variable-value) (const x) (reg env))
  (assign val (op var-val-extract-value) (reg val))
  (assign argl (op list) (reg val))
  (assign val (op lookup-variable-value) (const y) (reg env))
  (assign val (op var-val-extract-value) (reg val))
  (assign argl (op cons) (reg val) (reg argl))
  (assign argl (op reverse) (reg argl))
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch1))
compiled-branch2
  (assign continue (label after-call3))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch1
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
after-call3
```

This version of the compiler generates slower code, because the `reverse` now has to be performed at run-time, something that was spared before.

Exercise 5.37

The function `preserving` generates a `save` and `restore` for a register only if the first statement modifies a register the second one needs. By removing this condition, we can make `preserving` generate the instructions always:

```
(define (preserving regs seq1 seq2)
  (if (null? regs)
    (append-instruction-sequences seq1 seq2)
    (let ((first-reg (car regs)))
      (preserving (cdr regs)
        (make-instruction-sequence
          (list-union (list first-reg) (registers-needed seq1))
          (list-difference
            (registers-modified seq1)
            (list first-reg))
          (append
            `((save ,first-reg))
            (statements seq1)
            `((restore ,first-reg))))
        seq2))))
```

Here's the compiled code for `(f 1 2)` with the original `preserving`:

```
  (assign proc (op lookup-variable-value) (const f) (reg env))
  (assign proc (op var-val-extract-value) (reg proc))
  (assign val (const 2))
  (assign argl (op list) (reg val))
  (assign val (const 1))
  (assign argl (op cons) (reg val) (reg argl))
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch1))
compiled-branch2
  (assign continue (label after-call3))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch1
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
after-call3
```

Note that no saves and restores are generated, because none are needed here (see solution for Exercise 5.31)

Now, the same expression with the modified`preserving` is compiled to:

```
  (save continue)
  (save env)
  (save continue)
  (assign proc (op lookup-variable-value) (const f) (reg env))
  (assign proc (op var-val-extract-value) (reg proc))
  (restore continue)
  (restore env)
  (restore continue)
  (save continue)
  (save proc)
  (save env)
  (save continue)
  (assign val (const 2))
  (restore continue)
  (assign argl (op list) (reg val))
  (restore env)
  (save argl)
  (save continue)
  (assign val (const 1))
  (restore continue)
  (restore argl)
  (assign argl (op cons) (reg val) (reg argl))
  (restore proc)
  (restore continue)
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch1))
compiled-branch2
  (assign continue (label after-call3))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch1
  (save continue)
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
  (restore continue)
after-call3
```

Exercise 5.38

a & b.

I'll add this new dispatch to `compile`:

```
((memq (car exp) '(+ - * / =))
  (compile-open-binary-op exp target linkage))
```

And the extra code is:

```
(define (spread-arguments a1 a2)
  (let ((ca1 (compile a1 'arg1 'next))
        (ca2 (compile a2 'arg2 'next)))
    (list ca1 ca2)))

(define (compile-open-binary-op exp target linkage)
  (if (= (length exp) 3)
    (let ((op (car exp))
          (args (spread-arguments (cadr exp) (caddr exp))))
      (end-with-linkage linkage
        (append-instruction-sequences
          (car args)
          (preserving '(arg1)
            (cadr args)
            (make-instruction-sequence '(arg1 arg2) (list target)
              `((assign ,target (op ,op) (reg arg1) (reg arg2)))))))))
    (error "Expected a 3-element list -- COMPILE-OPEN-BINARY-OP" exp)))
```

Note that `spread-arguments` accepts strictly two arguments, because it has to place them in `arg1` and `arg2`. Also, I made it return a normal list, rather than an instruction sequence. This is because the appropriate place what to preserve is `compile-open-binary-op`, that has 3 instruction sequences to append:

1. The evaluation of the first argument into `arg1`
2. The evaluation of the second argument into `arg2`, preserving `arg1` because it will be needed later
3. The application of the operator to `arg1` and `arg2`

Here are a couple of examples of code generated with these functions:

```
'(* 4 5)
=>
  (assign arg1 (const 4))
  (assign arg2 (const 5))
  (assign val (op *) (reg arg1) (reg arg2))
  (goto (reg continue))

(* 66 (+ 1 3))
=>
  (assign arg1 (const 66))
  (save arg1)
  (assign arg1 (const 1))
  (assign arg2 (const 3))
  (assign arg2 (op +) (reg arg1) (reg arg2))
  (restore arg1)
  (assign val (op *) (reg arg1) (reg arg2))
  (goto (reg continue))
```

Exercise 5.39

```
(define (make-lexaddr framenum displacement)
  (list framenum displacement))

(define (lexaddr-framenum lexaddr)
  (car lexaddr))

(define (lexaddr-displacement lexaddr)
  (cadr lexaddr))

; Note, the following two functions operate on the _runtime_
; environment, where each frame is a pair: list of variables
; with a list of their values.
;
(define (lexical-address-lookup lexaddr rt-env)
  (let ((addr-frame (list-ref rt-env (lexaddr-framenum lexaddr))))
    (let ((addr-val (list-ref (frame-values addr-frame) (lexaddr-displacement lexaddr))))
      (if (eq? addr-val '*unassigned*)
        (error "Var is unassigned"))
        (cons 'bound addr-val))))

(define (lexical-address-set! lexaddr rt-env newval)
  (let ((addr-frame (list-ref rt-env (lexaddr-framenum lexaddr))))
    (define (iter vals count)
      (cond ((null? vals)
              (error "Invalid lexical address - bad displacement"))
            ((= count 0)
              (set-car! vals newval))
            (else (iter (cdr vals) (+ 1 count)))))
    (iter (frame-values addr-frame) (lexaddr-displacement lexaddr))))
```

Exercise 5.40

The full code for this section, including the modified compiler, can be downloaded here

He're the code generator for `lambda`, which extends the compile-time environment:

```
(define (compile-lambda-body exp proc-entry ct-env)
  (let ((formals (lambda-parameters exp)))
    (append-instruction-sequences
      (make-instruction-sequence '(env proc argl) '(env)
        `(,proc-entry
          (assign env (op compiled-procedure-env) (reg proc))
          (assign env
                  (op extend-environment)
                  (const ,formals)
                  (reg argl)
                  (reg env))))
      (compile-sequence
        (lambda-body exp)
        'val 'return
        (extend-ct-env ct-env formals)))))
```

It uses this utility function:

```
(define (extend-ct-env ct-env frame)
  (cons frame ct-env))
```

Exercise 5.41

```
(define (find-variable var ct-env)
  (define (frame-iter var frames count)
    (if (null? frames)
      '()
      (let ((var-index (index-in-list
                        (lambda (i) (eq? i var))
                        (car frames))))
        (if (null? var-index)
          (frame-iter var (cdr frames) (+ count 1))
          (make-lexaddr count var-index)))))
  (frame-iter var ct-env 0))

; Returns the numerical index (0-based) of the first
; item from the list that satisfies the predicate.
; If no suitable item is found, returns '()
;
(define (index-in-list pred lst)
  (define (iter pred lst count)
    (cond
      ((null? lst) '())
      ((pred (car lst)) count)
      (else (iter pred (cdr lst) (+ count 1)))))
  (iter pred lst 0))
```

Exercise 5.42

To understand how this works, it's important to keep in mind that the structure of the compile-time environment follows the structure of the run-time environment. The compiler, by knowing where inside the stack of `lambda` statements it's located, can figure out which value is referenced in the code.

```
(define (compile-variable exp target linkage ct-env)
  (let ((lookup-instruction
         (let ((lexaddr-of-var (find-variable exp ct-env)))
           (if (null? lexaddr-of-var)
             `(assign ,target
                      (op lookup-variable-value)
                      (const ,exp)
                      (reg env))
             `(assign ,target
                      (op lexical-address-lookup)
                      (const ,lexaddr-of-var)
                      (reg env))))))
    (end-with-linkage linkage
      (make-instruction-sequence '(env) (list target)
        (list
          lookup-instruction
          `(assign ,target
                 (op var-val-extract-value)
                 (reg ,target)))))))

(define (compile-assignment exp target linkage ct-env)
  (let ((var (assignment-variable exp))
        (get-value-code
          (compile (assignment-value exp) 'val 'next ct-env)))
    (end-with-linkage linkage
      (preserving '(env)
        get-value-code
        (make-instruction-sequence '(env val) (list target)
          (let ((lexaddr-of-var (find-variable var ct-env)))
            (if (null? lexaddr-of-var)
              `((perform
                 (op set-variable-value!)
                 (const ,var)
                 (reg val)
                 (reg env))
                (assign ,target (const ok)))
              `((perform
                 (op lexical-address-set!)
                 (const ,lexaddr-of-var)
                 (reg env)
                 (reg val))
                (assign ,target (const ok)))))))))
```

Here's an example that demonstrates how these modifications work. This code:

```
(define code
  '((lambda (x y)
      (lambda (u v w)
        (set! x 60)
        (+ v w x))
      10 20 30)
    1 2)
)
```

Is compiled into:

```
  (assign proc (op make-compiled-procedure) (label entry1) (reg env))
  (goto (label after-lambda2))
entry1
  (assign env (op compiled-procedure-env) (reg proc))
  (assign env (op extend-environment) (const (x y)) (reg argl) (reg env))
  (assign val (op make-compiled-procedure) (label entry3) (reg env))
  (goto (label after-lambda4))
entry3
  (assign env (op compiled-procedure-env) (reg proc))
  (assign env (op extend-environment) (const (u v w)) (reg argl) (reg env))
  (assign val (const 60))
  (perform (op lexical-address-set!) (const (1 0)) (reg env) (reg val))
  (assign val (const ok))
  (assign proc (op lookup-variable-value) (const +) (reg env))
  (assign proc (op var-val-extract-value) (reg proc))
  (assign val (op lexical-address-lookup) (const (1 0)) (reg env))
  (assign val (op var-val-extract-value) (reg val))
  (assign argl (op list) (reg val))
  (assign val (op lexical-address-lookup) (const (0 2)) (reg env))
  (assign val (op var-val-extract-value) (reg val))
  (assign argl (op cons) (reg val) (reg argl))
  (assign val (op lexical-address-lookup) (const (0 1)) (reg env))
  (assign val (op var-val-extract-value) (reg val))
  (assign argl (op cons) (reg val) (reg argl))
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch5))
compiled-branch6
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch5
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
  (goto (reg continue))
after-call7
after-lambda4
  (assign val (const 10))
  (assign val (const 20))
  (assign val (const 30))
  (goto (reg continue))
after-lambda2
  (assign val (const 2))
  (assign argl (op list) (reg val))
  (assign val (const 1))
  (assign argl (op cons) (reg val) (reg argl))
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch8))
compiled-branch9
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch8
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
  (goto (reg continue))
after-call10
```

Note how the relevant variables are accessed using `lexical-address-lookup`, where applicable.

Exercise 5.43

The change required here is very simple. `compile-lambda-body` is modified to execute `scan-out-defines` on the lambda body before passing it on to compilation:

```
(define (compile-lambda-body exp proc-entry ct-env)
  (let ((formals (lambda-parameters exp)))
    (append-instruction-sequences
      (make-instruction-sequence '(env proc argl) '(env)
        `(,proc-entry
          (assign env (op compiled-procedure-env) (reg proc))
          (assign env
                  (op extend-environment)
                  (const ,formals)
                  (reg argl)
                  (reg env))))
      (compile-sequence
        (scan-out-defines (lambda-body exp))  ;; change!
        'val 'return
        (extend-ct-env ct-env formals)))))
```

The rest of section 5.5

I think I'll skip the rest of the exercises.

[1] Although such error signaling shouldn't be difficult to add.

Back to top