# 6.001 Section Notes
# A Simpler Compiler

Albert R. Meyer

May 3, 2000

By dropping most of the optimizations from the compiler in the text, we get a simpler compilation procedure, `mycompile`, which helps clarify the core ideas of how Scheme programs are translated into register machine code.

- The compilation procedures below take an expression as their only parameter; each expression compiles in only one way.
- Compiled expressions and compiled procedures each fulfill the *same* contract wrt register and stack use.
- Compilation preserves tail recursiveness: procedures compiled from tail recursive definitions use only a bounded depth of stack.
- It is fairly easy to see what the generated register-machine code will look like, because the bodies of the subprocedures of `mycompile` look very much like the code they generate; this is accomplished by making elementary use of `backquote`.
- `mycompile` uses the syntax and register machine procedures of 6.001 Spring 97, PS10.
- `mycompile` is independent of, and not compatible with, the `eceval` interpreter, because the register-code it generates obeys a different contract than expected by `eceval`. Compatibility could be achieved with a local change in the `eceval` apply-dispatch code to handle `mycompile`'d procedures as a separate case.
- The compiled code presumes that the following register ops are available:

```
extend-environment
lookup-variable-value
set-variable-value!
define-variable!

empty-arglist
adjoin-arg

make-compiled-procedure
compiled-procedure-entry
compiled-procedure-env

primitive-procedure?
apply-primitive-procedure

false?
```

## The Contract

`(mycompile exp)` yields register machine code such that

- if register ENV contains an environment, and
- register CONTINUE contains a label,
- then execution of the code evaluates `exp`, causing appropriate mutations of the environment and store, if any, and leaves the value in VAL.
- Execution preserves the stack.
- At the end of execution, control transfers to the label originally in CONTINUE.
- All labels explicitly in the code are “fresh” (do not appear in other code fragments).

## The Compiler Procedures

```
(define (mycompile exp)
  (cond ((self-evaluating? exp)
         (mycompile-self-evaluating exp))
        ((quoted? exp)
         (mycompile-quoted (text-of-quotation exp)))
        ((variable? exp)
         (mycompile-variable-access exp))
        ((assignment? exp)
         (mycompile-assignment
          (assignment-variable exp)
          (assignment-value exp)))
        ((definition? exp)
         (mycompile-definition
          (definition-variable exp)
          (definition-value exp)))
        ((if? exp)
         (mycompile-if
          (if-predicate exp)
          (if-consequent exp)
          (if-alternative exp)))
        ((begin? exp)
         (if (no-more-exps? (begin-actions exp))
             (error "Empty sequence of actions -- MYCOMPILE" exp)
             (mycompile-seq (begin-actions exp))))
        ((cond? exp)
         (mycompile (COND->IF exp)))
        ((lambda? exp)
         (let ((params (lambda-parameters exp))
               (body (lambda-body exp)))
           (if (no-more-exps? body)
               (error "Empty lambda body -- MYCOMPILE" exp))
           (mycompile-lambda params body)))
        ((let? exp)
         (mycompile (LET->combination exp)))
        ((application? exp)
         (mycompile-application
```

```
          (operator exp)
          (operands exp)))
        (else
         (error "Unknown expression type -- MYCOMPILE" exp))))

(define (mycompile-self-evaluating constant)
  `((assign val (const ,constant))
   (goto (reg continue))))

(define (mycompile-quoted text)
  `((assign val (const ,text))
    (goto (reg continue))))

(define (mycompile-variable-access var)
  `((assign val (op lookup-variable-value) (const ,var) (reg env))
    (goto (reg continue))))

(define (mycompile-assignment assignment-var assignment-valexp)
  (let ((after-valexp-label (make-label 'after-valexp)))
    (append
     `((save continue)
       (assign continue (label ,after-valexp-label))
       (save env))
     (mycompile assignment-valexp)
     `(,after-valexp-label
       (restore env)
       (perform (op set-variable-value!) (const ,assignment-var) (reg val) (reg env))
       (assign val (const ,the-unspecified-value))
       (restore continue)
       (goto (reg continue))))))

(define (mycompile-definition def-var def-valexp)
  (let ((after-def-value-label (make-label 'after-def-value)))
    (append
     `((save continue)
       (assign continue (label ,after-def-value-label))
       (save env))
     (mycompile def-valexp)
     `(,after-def-value-label
       (restore env)
       (perform (op define-variable!) (const ,def-var) (reg val) (reg env))
       (assign val (const ,the-unspecified-value))
       (restore continue)
       (goto (reg continue))))))

(define (mycompile-if pred true-brnch false-brnch)
  (let ((after-pred-label (make-label 'after-pred))
        (false-branch-label (make-label 'false-branch)))
    (append
     `((save continue)
       (assign continue (label ,after-pred-label))
       (save env))
     (mycompile pred)
```

```
     `(,after-pred-label
       (restore env)
       (restore continue)
       (test (op false?) (reg val))
       (branch (label ,false-branch-label)))
     (mycompile true-brnch)
     `(,false-branch-label)
     (mycompile false-brnch))))

                       ; SEQUENCE is the only case where maintaining
                       ; tail-recursion requires special attention.
                       ; In particular, using a simple APPEND-MAP on the
                       ; whole explist would not yield tail-recursion

(define (mycompile-seq explist)
  (append
   '((save continue))
   (append-map
    (lambda (exp)
      (let ((after-current-exp-label (make-label 'after-current-exp)))
        (append
         `((save env)
           (assign continue (label ,after-current-exp-label)))
         (mycompile exp)
         `(,after-current-exp-label
           (restore env)))))
    (all-but-last explist))
   '((restore continue))
   (mycompile (last-exp explist))))

                       ;;SEQUENCE SYNTAX
(define (all-but-last explist)
  (if (last-exp? explist)
      '()
      (cons (first-exp explist)
            (all-but-last (rest-exps explist)))))

(define (last-exp explist)
  (list-ref explist (-1+ (length explist))))


(define (mycompile-lambda params body)
  (let ((proc-entry-label (make-label 'proc-entry)))
    (append
     `((assign val (op make-compiled-procedure) (label ,proc-entry-label) (reg env))
       (goto (reg continue))
       ,proc-entry-label
               ;This code expects the procedure-object's environment in
               ;ENV, a list of arguments in ARGL, and a label in
               ;CONTINUE.  It evaluates body in the extended environment,
               ;leaving its value in VAL.  It preserves the stack.
               ;Finally, it goes to CONTINUE.
       (assign env
```

```
               (op extend-environment)
               (const ,params)
               (reg argl)
               (reg env)))
     (mycompile-seq body))))

(define (mycompile-application rator rands)
  (let ((after-rator-label (make-label 'after-rator))
        (primitive-apply-label (make-label 'primitive-apply)))
    (append
     `((save continue)
       (assign argl (op empty-arglist)))
     (mycompile-operands rands)
     `((assign continue (label ,after-rator-label))
       (save argl))
     (mycompile rator)
     `(,after-rator-label
       (restore argl)
       (restore continue)
       (test (op primitive-procedure?) (reg val))
       (branch (label ,primitive-apply-label))
                                           ;doesn't handle interpreted procedures.
       (assign env (op compiled-procedure-env) (reg val))
       (assign val (op compiled-procedure-entry) (reg val))
       (goto (reg val))
       ,primitive-apply-label
       (assign val (op apply-primitive-procedure) (reg val) (reg argl))
       (goto (reg continue))))))

               ;This code evaluates rands, adjoining their values to ARGL.
               ;It preserves STACK and ENV, returns to next instruction.
(define (mycompile-operands rands)
  (append-map
   (lambda (rand)
     (let ((after-rand-label (make-label 'after-rand)))
       (append
        `((assign continue (label ,after-rand-label))
          (save env)
          (save argl))
        (mycompile rand)
        `(,after-rand-label
          (restore argl)
          (assign argl (op adjoin-arg) (reg val) (reg argl))
          (restore env)))))
   rands))
```

## Generated Code Samples

### Constant

```
(mycompile-and-display 3)

  (assign val (const 3))
```

```
  (goto (reg continue))
;Value: #[useless-value]
```

### Quoted

```
(mycompile-and-display '(quote (some text)))

  (assign val (const (some text)))
  (goto (reg continue))
```

### Variable

```
(mycompile-and-display 'x)

  (assign val (op lookup-variable-value) (const x) (reg env))
  (goto (reg continue))
```

### Assignment

```
(mycompile-and-display '(set! x 3))

  (save continue)
  (assign continue (label after-valexp1))
  (save env)
  (assign val (const 3))
  (goto (reg continue))
after-valexp1
  (restore env)
  (perform (op set-variable-value!) (const x) (reg val) (reg env))
  (assign val (const (the-unspecified-value)))
  (restore continue)
  (goto (reg continue))
```

### Definition

```
(mycompile-and-display '(define x 3))

  (save continue)
  (assign continue (label after-def-value4))
  (save env)
  (assign val (const 3))
  (goto (reg continue))
after-def-value4
  (restore env)
  (perform (op define-variable!) (const x) (reg val) (reg env))
  (assign val (const (the-unspecified-value)))
  (restore continue)
  (goto (reg continue))
```

### if

```
(mycompile-and-display '(if x 3 4))

  (save continue)
```

```
  (assign continue (label after-pred12))
  (save env)
  (assign val (op lookup-variable-value) (const x) (reg env))
  (goto (reg continue))
after-pred12
  (restore env)
  (restore continue)
  (test (op false?) (reg val))
  (branch (label false-branch11))
  (assign val (const 3))
  (goto (reg continue))
false-branch11
  (assign val (const 4))
  (goto (reg continue))
```

### begin

```
(mycompile-and-display '(begin x 3 4))

  (save continue)
  (save env)
  (assign continue (label after-current-exp43))
  (assign val (op lookup-variable-value) (const x) (reg env))
  (goto (reg continue))
after-current-exp43
  (restore env)
  (save env)
  (assign continue (label after-current-exp42))
  (assign val (const 3))
  (goto (reg continue))
after-current-exp42
  (restore env)
  (restore continue)
  (assign val (const 4))
  (goto (reg continue))
```

### Conditional

```
(mycompile-and-display '(cond (1 2) (3 4) (else 5)))

  (save continue)
  (assign continue (label after-pred1199))
  (save env)
  (assign val (const 1))
  (goto (reg continue))
after-pred1199
  (restore env)
  (restore continue)
  (test (op false?) (reg val))
  (branch (label false-branch1198))
  (assign val (const 2))
  (goto (reg continue))
false-branch1198
  (save continue)
```

```
  (assign continue (label after-pred1201))
  (save env)
  (assign val (const 3))
  (goto (reg continue))
after-pred1201
  (restore env)
  (restore continue)
  (test (op false?) (reg val))
  (branch (label false-branch1200))
  (assign val (const 4))
  (goto (reg continue))
false-branch1200
  (assign val (const 5))
  (goto (reg continue))
```

### lambda

```
(mycompile-and-display '(lambda (y z) z))

  (assign val (op make-compiled-procedure) (label proc-entry1202) (reg env))
  (goto (reg continue))
proc-entry1202
  (assign env (op extend-environment) (const (y z)) (reg argl) (reg env))
  (save continue)
  (restore continue)
  (assign val (op lookup-variable-value) (const z) (reg env))
  (goto (reg continue))
```

### Application

```
(mycompile-and-display '(+ x 3))
  (save continue)
  (assign argl (op empty-arglist))
  (assign continue (label after-rand1206))
  (save env)
  (save argl)
  (assign val (op lookup-variable-value) (const x) (reg env))
  (goto (reg continue))
after-rand1206
  (restore argl)
  (assign argl (op adjoin-arg) (reg val) (reg argl))
  (restore env)
  (assign continue (label after-rand1205))
  (save env)
  (save argl)
  (assign val (const 3))
  (goto (reg continue))
after-rand1205
  (restore argl)
  (assign argl (op adjoin-arg) (reg val) (reg argl))
  (restore env)
  (assign continue (label after-rator1204))
  (save argl)
  (assign val (op lookup-variable-value) (const +) (reg env))
```

```
  (goto (reg continue))
after-rator1204
  (restore argl)
  (restore continue)
  (test (op primitive-procedure?) (reg val))
  (branch (label primitive-apply1203))
  (assign env (op compiled-procedure-env) (reg val))
  (assign val (op compiled-procedure-entry) (reg val))
  (goto (reg val))
primitive-apply1203
  (assign val (op apply-primitive-procedure) (reg val) (reg argl))
  (goto (reg continue))
```

### Nested Expression

```
(mycompile-and-display  '(+ 6 (- 5 4)))

  (save continue)
  (assign argl (op empty-arglist))
  (assign continue (label after-rand1214))
  (save env)
  (save argl)
  (assign val (const 6))
  (goto (reg continue))
after-rand1214
  (restore argl)
  (assign argl (op adjoin-arg) (reg val) (reg argl))
  (restore env)
  (assign continue (label after-rand1209))
  (save env)
  (save argl)
  (save continue)
  (assign argl (op empty-arglist))
  (assign continue (label after-rand1213))
  (save env)
  (save argl)
  (assign val (const 5))
  (goto (reg continue))
after-rand1213
  (restore argl)
  (assign argl (op adjoin-arg) (reg val) (reg argl))
  (restore env)
  (assign continue (label after-rand1212))
  (save env)
  (save argl)
  (assign val (const 4))
  (goto (reg continue))
after-rand1212
  (restore argl)
  (assign argl (op adjoin-arg) (reg val) (reg argl))
  (restore env)
  (assign continue (label after-rator1211))
  (save argl)
```

```
  (assign val (op lookup-variable-value) (const -) (reg env))
  (goto (reg continue))
after-rator1211
  (restore argl)
  (restore continue)
  (test (op primitive-procedure?) (reg val))
  (branch (label primitive-apply1210))
  (assign env (op compiled-procedure-env) (reg val))
  (assign val (op compiled-procedure-entry) (reg val))
  (goto (reg val))
primitive-apply1210
  (assign val (op apply-primitive-procedure) (reg val) (reg argl))
  (goto (reg continue))
after-rand1209
  (restore argl)
  (assign argl (op adjoin-arg) (reg val) (reg argl))
  (restore env)
  (assign continue (label after-rator1208))
  (save argl)
  (assign val (op lookup-variable-value) (const +) (reg env))
  (goto (reg continue))
after-rator1208
  (restore argl)
  (restore continue)
  (test (op primitive-procedure?) (reg val))
  (branch (label primitive-apply1207))
  (assign env (op compiled-procedure-env) (reg val))
  (assign val (op compiled-procedure-entry) (reg val))
  (goto (reg val))
primitive-apply1207
  (assign val (op apply-primitive-procedure) (reg val) (reg argl))
  (goto (reg continue))
```

Compare the above 60 instructions plus 8 labels to the 30 instructions plus 5 labels generated by the compiler of PS10:

**PS10 Compilation**

```
(compile-and-display  '(+ 6 (- 5 4)))

  (assign proc (op lookup-variable-value) (const +) (reg env))
  (save continue)
  (save proc)
  (assign proc (op lookup-variable-value) (const -) (reg env))
  (assign val (const 4))
  (assign argl (op list) (reg val))
  (assign val (const 5))
  (assign argl (op cons) (reg val) (reg argl))
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch23))
compiled-branch22
  (assign continue (label after-call21))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
```

```
primitive-branch23
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
after-call21
  (assign argl (op list) (reg val))
  (assign val (const 6))
  (assign argl (op cons) (reg val) (reg argl))
  (restore proc)
  (restore continue)
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch26))
compiled-branch25
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch26
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
  (goto (reg continue))
```

## Tests

To run `mycompile`'d code for an expression, we define some utility procedures:

```
(define (mycompile-and-display expression)
  (define (iter s)
    (cond ((not (null? s))
           (let ((state (car s)))
             (newline)
             (cond ((pair? state)
                    (display "  ")
                    (display state))
                   (else (pp state))))
           (iter (cdr s)))))
  (iter (mycompile expression)))

(define compiler-ops
  `((lookup-variable-value ,lookup-variable-value)
    (set-variable-value! ,set-variable-value!)
    (define-variable! ,define-variable!)
    (false? ,false?)
    (empty-arglist ,empty-arglist)
    (adjoin-arg ,adjoin-arg)
    (make-compiled-procedure ,make-compiled-procedure)
    (extend-environment ,extend-environment)
    (compiled-procedure-entry ,compiled-procedure-entry)
    (compiled-procedure-env ,compiled-procedure-env)
    (primitive-procedure? ,primitive-procedure?)
    (apply-primitive-procedure ,apply-primitive-procedure)))

(define access-entire-scheme-environment? #t)


(define (make-into-machine exp)
  (make-machine '(val env continue argl)
```

```
                compiler-ops
                (mycompile exp)))


(define (mycompile-and-go exp)
  (let ((my-machine (make-into-machine exp)))
    (set-register-contents! my-machine 'env (setup-environment))
    (set-register-contents! my-machine 'continue '())
    (newline)
    (display ";Resetting... ignore")
    (let ((the-stack (my-machine 'stack)))
      (the-stack 'initialize)
      (start my-machine)
      (newline)
      (display ";Run complete:")
      (the-stack 'print-statistics))
    (get-register-contents my-machine 'val)))


(define (fibexp n)
  `(begin (define (fib m)
            (if (< m 2)
                m
                (+ (fib (- m 1)) (fib (- m 2)))))
          (fib ,n)))

(define (tailexp n)
  `(let ((x ,n))
     (define (iter)
       (cond ((zero? x) x)
             (else (set! x (-1+ x))
                   (display '!)
                   (iter))))
     (newline)
     (iter)))

(define (not-tailexp n)
  `(let ((x ,n))
     (define (iter)
       (cond ((zero? x) x)
             (else (set! x (-1+ x))
                   (display '!)
                   (+ 0 (iter)))))  ;;the "(+ 0 ...)" breaks tail-recursion
     (newline)
     (iter)))
```

We first compare the somewhat optimized compiler of PS10 to the simple compiler above on a familiar recursive example:

```
(mycompile-and-go (fibexp 8))
(total-pushes = 1469 maximum-depth = 27)
(machine-ops = 6544)
;value of expression: 21
;Value: #[useless-value]
```

```
(compile-and-go (fibexp 8))
(total-pushes = 332 maximum-depth = 23)
(machine-ops = 2772 )
;;; EC-Eval value: 21
```

So in this case, the optimized code executes about 40% as many instructions and 85% of the stack space as the code generated by the simple compiler above.

Let's also check for tail-recursive behavior:

```
(mycompile-and-go (tailexp 50))
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
(total-pushes = 1121 maximum-depth = 7)
(machine-ops = 5049)
;value of expression: 0
;Value: #[useless-value]

(mycompile-and-go (not-tailexp 50))
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
(total-pushes = 1421 maximum-depth = 155)
(machine-ops = 6349)
;value of expression: 0
;Value: #[useless-value]
```

Sure enough, the stack depth stays bounded (by 7) for the tail-recursive version, but grows linearly (namely, to depth $3n + 5$) for the non-tail-recursive version.