MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1998

Lecture Notes – Nov. 24, 1998

Compilation

If we watched the evaluation of the combination `(f x)` in the explicit control evaluator we would see the following sequence of machine instructions between deciding that its an application and getting to `apply-dispatch`.

```
 1. (save continue)
 2. (save env)
 3. (assign unev (op operands) (reg exp))
 4. (save unev)
 5. (assign exp (op operator) (reg exp))
 6. (assign continue (label ev-appl-did-operator))
 7. (goto (label eval-dispatch))                          ;goto taken!
 8. (test (op self-evaluating?) (reg exp))
 9. (branch (label ev-self-eval))                         ;not taken
10. (test (op quoted?) (reg exp))
11. (branch (label ev-quoted))                            ;not taken
12. (test (op variable?) (reg exp))
13. (branch (label ev-variable))                          ;taken
14. (assign val (op lookup-variable-value) (reg exp) (reg env))
15. (goto (reg continue))                                 ;goto ev-appl-did-operator
16. (restore unev)
17. (restore env)
18. (assign argl (op empty-arglist))
19. (assign proc (reg val))
20. (test (op no-operands?) (reg unev))
21. (branch (label apply-dispatch))                       ;not taken
22. (save proc)
23. (save argl)
24. (assign exp (op first-operand) (reg unev))
25. (test (op last-operand?) (reg unev))
26. (branch (label ev-appl-last-arg))                     ;taken
27. (assign continue (label ev-appl-accum-last-arg))
28. (goto (label eval-dispatch))                          ;goto taken!
29. (test (op self-evaluating?) (reg exp))
30. (branch (label ev-self-eval))                         ;not taken
31. (test (op quoted?) (reg exp))
32. (branch (label ev-quoted))                            ;not taken
33. (test (op variable?) (reg exp))
34. (branch (label ev-variable))                          ;taken
35. (assign val (op lookup-variable-value) (reg exp) (reg env))
36. (goto (reg continue))                                 ;goto ev-appl-accum-last-arg
37. (restore argl)
38. (assign argl (op adjoin-arg) (reg val) (reg argl))
39. (restore proc)
;; computation proceeds at apply-dispatch
```

If we know we are evaluating `(f x)` however, then all the tests, branches, and gotos will work out the same way every time, so we can eliminate them, leaving just the register operations.

```
 1. (save continue)
 2. (save env)
 3. (assign unev (op operands) (reg exp))
 4. (save unev)
 5. (assign exp (op operator) (reg exp))
 6. (assign continue (label ev-appl-did-operator))
14. (assign val (op lookup-variable-value) (reg exp) (reg env))
16. (restore unev)
17. (restore env)
18. (assign argl (op empty-arglist))
19. (assign proc (reg val))
22. (save proc)
23. (save argl)
24. (assign exp (op first-operand) (reg unev))
27. (assign continue (label ev-appl-accum-last-arg))
35. (assign val (op lookup-variable-value) (reg exp) (reg env))
37. (restore argl)
38. (assign argl (op adjoin-arg) (reg val) (reg argl))
39. (restore proc)
;; computation proceeds at apply-dispatch
```

With a compiler, we can build the pieces of the expression directly into the register operations in lines 14 and 35. Thus, we do not need to worry about saving and restoring `exp` or `unev`. We can also ignore the continuations generated in lines 6 and 27, and thus do not need the initial save of continue in line 1. Thus, simply taking advantage of the fact that the form of the expression can be compiled into the flow of the evaluation gives the following alternative to the above code:

```
 2. (save env)
14. (assign val (op lookup-variable-value) (const f) (reg env))
17. (restore env)
18. (assign argl (op empty-arglist))
19. (assign proc (reg val))
22. (save proc)
23. (save argl)
35. (assign val (op lookup-variable-value) (const x) (reg env))
37. (restore argl)
38. (assign argl (op adjoin-arg) (reg val) (reg argl))
39. (restore proc)
;; computation proceeds at apply-dispatch
```

But we can do some more optimizations if we look carefully at the details of the code.

- Line 14 doesn't clobber `env` so no need to save/restore with 2 and 17.
- Line 14 may as well put the value directly in `proc`, so eliminate 19.
- Lines 18, 23, and 37 simply save and restore the empty list for 38.
- `proc` is not clobbered, so no need to save and restore it.

Thus a smarter compiler would generate:

```
14. (assign proc (op lookup-variable-value) (const f) (reg env))
35. (assign val (op lookup-variable-value) (const x) (reg env))
38. (assign argl (op adjoin-arg) (reg val) (const ()))
;; computation proceeds at apply-dispatch
```

In fact this is what our compiler we use in problem set 9 will generate (except that the stuff at `apply-dispatch` is also compiled:

```
1 ]=> (pretty-print (compile '(f x) 'val 'next))
((env)
 (env proc argl continue val)
 ((assign proc (op lookup-variable-value) (const f) (reg env))  ;our three lines
  (assign val (op lookup-variable-value) (const x) (reg env))   ;of optimized code
  (assign argl (op list) (reg val))                             ;just as above
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch9))
  compiled-branch8
  (assign continue (label after-call7))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
  primitive-branch9
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
  after-call7))
```

Backquote and comma

Just as `'` is a character that is treated specially by the reader (so that `'(a b)` gets read as though it was `(quote (a b))`) so are the characters ` (backquote) and `,` (comma). The details of how they are treated are complex, but the rough idea is that backquote says to make an expression which when evaluated treats the following thing to be read as an expression where things following a comma have their values substituted.

So if we have:

```
(define a 3)
(define b '(bar x))
```

then (using `==>` to mean "evaluates to") we get:

```
`(foo ,a ,b) ==> (foo 3 (bar x))        [(list 'foo a b)]
`(foo ,a b)  ==> (foo 3 b)              [(list 'foo a 'b)]
`(foo (+ ,a 2 ,4 ,'a) ,b) ==> (foo (+ 3 2 4 a) (bar x))
                                        [(list 'foo (list '+ a '2 4 'a) b)]
```

where the expressions in square brackets are what the reader actually reads.

The Compiler

The rest of the code is exactly as it appears in the file `compiler.scm` in problem set 9.

The big dispatcher:

```
(define (compile exp target linkage)
  (cond ((self-evaluating? exp)
         (compile-self-evaluating exp target linkage))
        ((quoted? exp) (compile-quoted exp target linkage))
        ((variable? exp)
         (compile-variable exp target linkage))
        ((assignment? exp)
         (compile-assignment exp target linkage))
        ((definition? exp)
         (compile-definition exp target linkage))
        ((if? exp) (compile-if exp target linkage))
        ((lambda? exp) (compile-lambda exp target linkage))
        ((begin? exp)
         (compile-sequence (begin-actions exp)
                           target
                           linkage))
        ((cond? exp) (compile (cond->if exp) target linkage))
        ((application? exp)
         (compile-application exp target linkage))
        (else
         (error "Unknown expression type -- COMPILE" exp))))
```

An abstraction for code sequences which also handles a singleton symbol as a label in an instruction sequence:

```
(define (make-instruction-sequence needs modifies statements)
  (list needs modifies statements))
(define (empty-instruction-sequence)
  (make-instruction-sequence '() '() '()))
(define (registers-needed s)
  (if (symbol? s) '() (car s)))
(define (registers-modified s)
  (if (symbol? s) '() (cadr s)))
(define (statements s)
  (if (symbol? s) (list s) (caddr s)))
(define (needs-register? seq reg)
  (memq reg (registers-needed seq)))
(define (modifies-register? seq reg)
  (memq reg (registers-modified seq)))
```

Appending code sequences assuming that they don't interfere with each other in anyway:

```
(define (append-instruction-sequences . seqs)
  (define (append-2-sequences seq1 seq2)
    (make-instruction-sequence
     (list-union (registers-needed seq1)
                 (list-difference (registers-needed seq2)
                                  (registers-modified seq1)))
     (list-union (registers-modified seq1)
                 (registers-modified seq2))
     (append (statements seq1) (statements seq2))))
  (define (append-seq-list seqs)
    (if (null? seqs)
        (empty-instruction-sequence)
        (append-2-sequences (car seqs)
                            (append-seq-list (cdr seqs)))))
  (append-seq-list seqs))
```

Appending two code sequences making sure that `seq1` does not clobber and registers in the list `regs`:

```
(define (preserving regs seq1 seq2)
  (if (null? regs)
      (append-instruction-sequences seq1 seq2)
      (let ((first-reg (car regs)))
        (if (and (needs-register? seq2 first-reg)
                 (modifies-register? seq1 first-reg))
            (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)
            (preserving (cdr regs) seq1 seq2)))))
```

Some simple handlers from the big compile dispatch:

```
(define (compile-self-evaluating exp target linkage)
  (end-with-linkage linkage
   (make-instruction-sequence '() (list target)
    `((assign ,target (const ,exp))))))
(define (compile-quoted exp target linkage)
  (end-with-linkage linkage
   (make-instruction-sequence '() (list target)
    `((assign ,target (const ,(text-of-quotation exp)))))))
```

But these need to handle the linkage at the end:

```
(define (compile-linkage linkage)
  (cond ((eq? linkage 'return)
         (make-instruction-sequence '(continue) '()
          '((goto (reg continue)))))
        ((eq? linkage 'next)
         (empty-instruction-sequence))
        (else
         (make-instruction-sequence '() '()
          `((goto (label ,linkage)))))))
(define (end-with-linkage linkage instruction-sequence)
  (preserving '(continue)
   instruction-sequence
   (compile-linkage linkage)))
```

One key to tail recursion is that we delay generating the linkage until each instance of the actual procedure call, rather than just calling `end-with-linkage` here and calling `compile-procedure-call` with a linkage of `next`. Also `construct-arglist` determines the order of evaluation of arguments.

```
(define (compile-application exp target linkage)
  (let ((proc-code (compile (operator exp) 'proc 'next))
        (operand-codes
         (map (lambda (operand) (compile operand 'val 'next))
              (operands exp))))
    (preserving '(env continue)
     proc-code
     (preserving '(proc continue)
      (construct-arglist operand-codes)
      (compile-procedure-call target linkage)))))
```

An interesting case is `if`:

```
(define (compile-if exp target linkage)
  (let ((t-branch (make-label 'true-branch))
        (f-branch (make-label 'false-branch))
        (after-if (make-label 'after-if)))
    (let ((consequent-linkage                          ;tail recursion
           (if (eq? linkage 'next) after-if linkage)))  ;sensitivity!
      (let ((p-code (compile (if-predicate exp) 'val 'next))
            (c-code
             (compile
              (if-consequent exp) target consequent-linkage))
            (a-code
             (compile (if-alternative exp) target linkage)))
        (preserving '(env continue)
         p-code
         (append-instruction-sequences
          (make-instruction-sequence '(val) '()
           `((test (op false?) (reg val))
             (branch (label ,f-branch))))
          (parallel-instruction-sequences
           (append-instruction-sequences t-branch c-code)
           (append-instruction-sequences f-branch a-code))
          after-if))))))
```

We need to have two instruction sequences and we don't know which one will be taken at compile time, so we must assume the worst case for all the registers, even though at run time it won't be so bad:

```
(define (parallel-instruction-sequences seq1 seq2)
  (make-instruction-sequence
   (list-union (registers-needed seq1)
               (registers-needed seq2))
   (list-union (registers-modified seq1)
               (registers-modified seq2))
   (append (statements seq1) (statements seq2))))
```