Watching EC-EVAL on (f x)

```
(save continue)
(save env)
(assign unev (op operands) (reg exp))
(save unev)
(assign exp (op operator) (reg exp))
(assign continue (label ev-appl-did-operator))
(goto (label eval-dispatch))
(test (op self-evaluating?) (reg exp))
(branch (label ev-self-eval))
(test (op quoted?) (reg exp))
(branch (label ev-quoted))
(test (op variable?) (reg exp))
(branch (label ev-variable))
(assign val (op lookup-variable-value) (reg exp) (reg env))
(goto (reg continue))
(restore unev)
(restore env)
(assign argl (op empty-arglist))
(assign proc (reg val))
(test (op no-operands?) (reg unev))
(branch (label apply-dispatch))
(save proc)
(save argl)
(assign exp (op first-operand) (reg unev))
(test (op last-operand?) (reg unev))
(branch (label ev-appl-last-arg))
(assign continue (label ev-appl-accum-last-arg))(goto (label eval-dispatch))
(test (op self-evaluating?) (reg exp))
(branch (label ev-self-eval))
(test (op quoted?) (reg exp))
(branch (label ev-quoted))
(test (op variable?) (reg exp))
(branch (label ev-variable))
(assign val (op lookup-variable-value) (reg exp) (reg env))
(goto (reg continue))
(restore argl)
(assign argl (op adjoin-arg) (reg val) (reg argl))
(restore proc)
```

Register Traffic

- 🟦 **save**
- 🟥 **assign**
- 🟪 **restore**
- ● 🟩 **reference**

argl	proc	unev	continue	val	env	exp	
			🟦				(save continue)
					🟦		(save env)
		🟥				🟩	(assign unev (op operands) (reg exp))
		🟦					(save unev)
						🟥●	(assign exp (op operator) (reg exp))
			🟥				(assign continue (label ev-appl-did-operator))
				🟥	🟩	🟩	(assign val (op lookup-variable-value) (reg exp) (reg env))
		🟪					(restore unev)
					🟪		(restore env)
🟥							(assign argl (op empty-arglist))
	🟥			🟩			(assign proc (reg val))
	🟦						(save proc)
🟦							(save argl)
		🟩				🟥	(assign exp (op first-operand) (reg unev))
			🟥				(assign continue (label ev-appl-accum-last-arg))
				🟥	🟩	🟩	(assign val (op lookup-variable-value) (reg exp) (reg env))
🟪							(restore argl)
🟥●				🟩			(assign argl (op adjoin-arg) (reg val) (reg argl))
	🟪						(restore proc)

Don’t Need Continuations

🟦 **save**

🟥 **assign**

▨ **restore**

● 🟩 **reference**

argl	proc	unev	continue	val	env	exp	
					🟦		(save env)
		🟥				🟩	(assign unev (op operands) (reg exp))
		🟦					(save unev)
						🟥●	(assign exp (op operator) (reg exp))
				🟥	🟩	🟩	(assign val (op lookup-variable-value) (reg exp) (reg env))
		▨					(restore unev)
					▨		(restore env)
🟥							(assign argl (op empty-arglist))
	🟥			🟩			(assign proc (reg val))
	🟦						(save proc)
🟦							(save argl)
		🟩				🟥	(assign exp (op first-operand) (reg unev))
				🟥	🟩	🟩	(assign val (op lookup-variable-value) (reg exp) (reg env))
▨							(restore argl)
🟥●				🟩			(assign argl (op adjoin-arg) (reg val) (reg argl))
	▨						(restore proc)

Taking Form (f x) Into Account

- 🟦 **save**
- 🟥 **assign**
- 🟪 **restore**
- ● 🟩 **reference**

argl	proc	unev	continue	val	env	exp	
					🟦		(save env)
				🟥	🟩	🟩	(assign val (op lookup-variable-value) (const f) (reg env))
					🟪		(restore env)
🟥							(assign argl (op empty-arglist))
	🟥			🟩			(assign proc (reg val))
	🟦						(save proc)
🟦							(save argl)
				🟥	🟩	🟩	(assign val (op lookup-variable-value) (const x) (reg env))
🟪							(restore argl)
🟥●				🟩			(assign argl (op adjoin-arg) (reg val) (reg argl))
	🟪						(restore proc)

Remove Extra Save/Restores

🟦 **save**
🟥 **assign**
🟪 **restore**
● 🟩 **reference**

argl	proc	unev	continue	val	env	exp	
				🟥	🟩	🟩	(assign val (op lookup-variable-value) (const f) (reg env))
🟥							(assign argl (op empty-arglist))
	🟥			🟩			(assign proc (reg val))
🟦							(save argl)
				🟥	🟩	🟩	(assign val (op lookup-variable-value) (const x) (reg env))
🟪							(restore argl)
🟥●				🟩			(assign argl (op adjoin-arg) (reg val) (reg argl))

Optimize Register Target

🟦 **save**
🟥 **assign**
🟪 **restore**
● 🟩 **reference**

argl	proc	unev	continue	val	env	exp	
	🟥				🟩	🟩	(assign proc (op lookup-variable-value) (const f) (reg env))
🟥							(assign argl (op empty-arglist))
🟦							(save argl)
				🟥	🟩	🟩	(assign val (op lookup-variable-value) (const x) (reg env))
🟪							(restore argl)
🟥●				🟩			(assign argl (op adjoin-arg) (reg val) (reg argl))

Trace Constant () Usage

- 🟦 **save**
- 🟥 **assign**
- 🟪 **restore**
- 🟩 **reference**

argl	proc	unev	continue	val	env	exp	
	🟥				🟩	🟩	(assign proc (op lookup-variable-value) (const f) (reg env))
				🟥	🟩	🟩	(assign val (op lookup-variable-value) (const x) (reg env))
🟥				🟩			(assign argl (op adjoin-arg) (reg val) (const ()))

Compiler Dispatch

```
(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))))
```

Sequence Abstraction

```
(define (make-instruction-sequence
          needs modifies statements)
  (list needs modifies statements))

(define (empty-instruction-sequence)
  (make-instruction-sequence
    '() '() '()))
```

Accessors (handling labels)

```
(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)))
```

Plus Predicates

```
(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)))
```

Simple Appending

```
(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 With Preservation

```
(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)))))
```

Simple Things To Compile

```
(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)))))))
```

Handling The Linkage

```
(define (end-with-linkage
          linkage instruction-sequence)
  (preserving '(continue)
   instruction-sequence
   (compile-linkage linkage)))

(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)))))))
```

Register Needs of Application

```
[(save continue)]
[(save env)]
<evaluate operator; result in proc>
[(restore env)]
[(save proc)]
<evaluate operands; result in argl>
[(restore proc)]
[(restore continue)]
<apply procedure in proc to arguments
   in argl, and link>
```

Compiling A Procedure Application

```
(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)))))
```

Register Needs of IF

```
[(save env)]
[(save continue)]
<evaluate predicate; result in val>
[(restore continue)]
[(restore env)]
(test (op false?) (reg val))
(branch (label <elselabel>))
<evaluate consequent; result in target;
                      special linkage>
<elselabel>
<evaluate alternate; result in target;
                     linkage>
```

An IF Destroying Tail Recursion

```
[(save env)]
[(save continue)]
<evaluate predicate; result in val>
[(restore continue)]
[(restore env)]
(test (op false?) (reg val))
(branch (label <elselabel>))
<evaluate consequent; result in target>
(goto (label <endlabel>)
<elselabel>
<evaluate alternate; result in target>
<endlabel>
<linkage>
```

Compiling an 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
            (if (eq? linkage 'next)
                after-if
                linkage)))
      (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))))))
```

Worst Case For Two Codes

```
(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))))
```

Compiled Code Still Cumbersome

```
(+ x y)
```

```
(assign proc (op lookup-variable-value)
             (const +)
             (reg env))
(assign val (op lookup-variable-value)
            (const y)
            (reg env))
(assign argl (op list) (reg val))
(assign val (op lookup-variable-value)
            (const x)
            (reg env))
(assign argl (op cons)
             (reg val)
             (reg argl))
;; proceed as if at apply-dispatch
```

Better?

```
(+ x y)
```

```
(assign exp (op lookup-variable-value)
            (const y)
            (reg env))
(assign val (op lookup-variable-value)
            (const x)
            (reg env))
(assign val (op +)
            (reg val)
            (reg exp))
;; computation proceeds
;; forget about apply-dispatch !!
```

But...

```
((lambda (+) (+ x y)) *)
```

```
(assign exp (op lookup-variable-value)
            (const y)
            (reg env))
(assign val (op lookup-variable-value)
            (const x)
            (reg env))
(assign val (op +)
            (reg val)
            (reg exp))
;; computation proceeds
;; forget about apply-dispatch !!
```