First version of compiler is a simple recursive tree walking translator, well atleast the code generator phase is.

format of compilation is 3 arguments , x is the expression , si is the stack index , env is the compile time environment.

```
(define (comp x si env) ...)
```

Every pointer to a heap- allocated object is tagged by a 3-bit tag (001 b for pairs, 010 b for vectors, 011 b for strings, 101 b for symbols, and 110 b for closures; 000 b , 100 b and 111 b were already used for fixnums and the other immediate objects).

Macro expansion .

Alpha conversion .

Tail call elimination .

Free variable analysis .

Closure conversion .

Simple compiler.

Development of these notes is in reverse order.

The spill phase - pass 1 identify free variables

Assuming really that these lambdas are just labels to jump to in code.

No higher order procedures here, no closures being created.

```
(define (ktak x y z k)
k x y z       : (k< y x
k x y z          :  (lambda ()   (k- x 1
k x y z        :   (lambda (x1) (ktak x1 y z
k x y z         :   (lambda (t1) (k- y 1
k x y z :        :   (lambda (y1) (ktak y1 z x
k t1 t2 x y z :  (lambda (t2) (k- z 1
x y t1 t2 k    :   (lambda (z1) (ktak z1 x y
t1 t2 k        :   (lambda (t3) (ktak t1 t2 t3 k)))))))))))))
k z            : (lambda () (k z))))
```

ktak is known , this is procedure being compiled. k is unknown as it will no doubt vary.

Exploring CPS conversion and compilation to x86 32 bit register machine code.

```
(define tak (lambda (x y z)
      (if (< y x)
  (tak (tak (- x 1) y z)
       (tak (- y 1) z x)
       (tak (- z 1) x y))
  z)))
(tak 18 12 6)

=> 7
```

CPS convert by hand , every call is a tail call

```
(define (k< a b k1 k2)  (if (< a b)  (k1)   (k2)))

(define (k- a b k)  (k (- a b)))

(define (ktak x y z k)
  (k< y x
      (lambda () ; (< y x)
(k- x 1
    (lambda (x1)
      (ktak x1 y z
   (lambda (tak1v)
(k- y 1
    (lambda (y1)
      (ktak y1 z x
   (lambda (tak2v)
(k- z 1
    (lambda (z1)
      (ktak z1 x y
    (lambda (tak3v)
      (ktak tak1v tak2v tak3v k)))))))))))))
      (lambda () (k z))))

;; identity continuation
(define kid (lambda (x) x))

(ktak 18 12 6 kid)
```

....

Lisp.scm

Minimal.scm

Minimal version is supposed to be absolute CORE of SPECIAL FORMS.

IF is a conditional . Cond can be built on if construct.

QUOTE allows arbitrary lisp expressions .

BEGIN allows sequencing of operations .

DEFINE allows mutate the environment , it also redefines the first found definition if it exists, or creates a new definition. the original environment is mutated in place.

SET! should alter existing binding in the environment.

LAMBDA allows abstraction .

There is another APPLICATION

....

so these 7 special forms QUOTE , IF , BEGIN , DEFINE , SET! , LAMBDA , APPLICATION .

so we notice there is no READER and no pretty PRINTER.

we can make the low level code FAST , if it does little or no checks .

....

Okay , do we need some PRIMITIVES .

....

'() : nil : the empty list

(null? x) : type predicate

....

Pairs

(cons x y) : constructor

(car x) : selector

(cdr x) : selector

(pair? x) : type predicate

....

Symbols

(make-symbol x) : constructor

(eq? x y) : are two symbols the same memory location ?

(symbol? x) : type predicate

(gensym) : special symbol constructor

....

Strings

(string-length x) : length of string

(string? x) : type predicate

....

Self evaluating items - #a characters , "a string" strings , 123 numbers , #T #F booleans.

Symbols are looked up in the environment.

also hash tables , vectors ...

....

Boolean

#T true : constructor

#F false : constructor

(boolean? x) : type predicate

....

Compiler

If compiler can see an expression that can be simplified it is probably a good idea to replace it with an equivalent but more efficient expression.

```
(if #f 1 2) => 2
```

Lets assume got debugged code and want it to run fast. Here is where the compiler can help.

Lets build a compiler.

Lets build a correct interpreter first then .

In the following pages are the basis of a scheme interpreter using 7 registers argl , expr , val , et al .

The interpreter does not include call-with-current-continuation , but this is a small addition with another primitive that wraps continuation much like a closure wraps lambda and environment. * TO DO * .

Condition system needs to be written in also , as we do not yet have any way to signal a condition, let alone handle it or pass it off.

Garbage collector needs to be written up , that is also missing.

Evaluation of arguments left to right means that building up argument list requires a reversal at end of ev-operands.

```
-----------------------------------------------------------
        *    repl   *
-----------------------------------------------------------
because in a repl we do not know when last expression will be ,
we treat it like an endless begin sequence.
thus we need to

MAYBE - avoid saving and restoring cont , have ev-read-3 do restore and continue
so save cont register before entering ev-read proper ,
-----------------------------------------------------------
(define (eval-repl exp env cont)
  (newline)
  (display ";; Ready > ")
  (let ((input (read)))
    (base-eval input env
     (lambda (result)
   (display ";; Value > ")
   (display result)
   (newline)
   (eval-repl #f env cont)))))
-----------------------------------------------------------
ev-read :
 save cont
 goto ev-read-2

ev-read-2 : display ";; Ready > "
 exp = read-primitive
 if eof-object?
     goto ev-read-4
 save env
 cont = ev-read-2
 goto eval-dispatch

ev-read-3 :
 restore env
 display ";; Value > " val
 goto ev-read

ev-read-4 :
         restore cont
```

```
 goto continue
----------------------------------------------------------------
```

in this version , shorter but makes 2 saves and restores , instead of just 1 for each round trip of the read eval print loop.

```
---------------------------------------------------------
        *    repl   *
---------------------------------------------------------
ev-read :
 display ";; Ready > "
 exp = read-primitive
 if eof-object?
     goto continue
 save cont
 save env
 cont = ev-read-2
 goto eval-dispatch

ev-read-2 :
 restore env
 restore cont
 display ";; Value > " val
 goto ev-read

---------------------------------------------------------
```

```
---------------------------------------------------------
        *    apply-dispatch   *
---------------------------------------------------------
inputs    : proc = evaluated procedure , argl = evaluated arguments
outputs   :
clobber   :
preserved :
---------------------------------------------------------
apply-dispatch :
 if primitive-procedure? proc
     goto primitive-apply
 if user-procedure? proc
     goto user-apply
 otherwise
     eval-error " unknown procedure type "
---------------------------------------------------------
        *    primitive-apply   *
---------------------------------------------------------
primitive-apply :
save continue ;; allows primitive to use all registers
argl = reverse argl
val = call the primitive proc with arguments argl
restore continue ;; restore
goto continue
---------------------------------------------------------
        *    user-apply   *
---------------------------------------------------------
user-apply :
argl = reverse argl
unev = user-lambda-params proc
env  = user-lambda-env proc
extend the environment [env] with [unev : params ] and [ argl : values ]
unev = user-lambda-body proc
goto ev-sequence ;; implicit sequence on body of lambda
---------------------------------------------------------
```

```
-----------------------------------------------------------
        *    eval-dispatch   *
-----------------------------------------------------------
example input  :    exp = < thing to be evaluated >
example output :    val = < what it evaluates to >
-----------------------------------------------------------
inputs    : exp
outputs   : val
clobber   : argl , exp , unev , val , proc
preserved : cont , env
-----------------------------------------------------------
eval-dispatch :
 if self-evaluating? exp
     val = exp
     goto continue
 if symbol? exp
     val = lookup symbol exp env
     goto continue
 if pair? exp and car [exp] = 'if
     goto ev-if
 if pair? exp and car [exp] = 'quote
     val = car [ cdr exp ]
     goto continue
 if pair? exp and car [exp] = 'begin
     goto ev-begin
 if pair? exp and car [exp] = 'lambda
     goto ev-lambda
 if pair? exp and car [exp] = 'define
     goto ev-definition
 if pair? exp and car [exp] = 'set!
     goto ev-assignment
 if pair? exp
     goto ev-application
 otherwise
     eval-error " what now ? "
-----------------------------------------------------------
```

```
---------------------------------------------------------
 self-evaluating ?
---------------------------------------------------------
 if boolean? exp
    string? exp
    vector? exp
    procedure? exp
  then return true
  else return false
---------------------------------------------------------
        *    ev-lambda   *
---------------------------------------------------------
ev-lambda :
 unev = car [ cdr exp ] ; params of lambda
 exp = cdr [ cdr exp ]  ; body of lambda
 val = #[ closure unev exp env ]
         goto cont
---------------------------------------------------------
(define (user-procedure? x)
  (and (vector? x)
       (eq? 'closure (vector-ref x 0))))
(define (user-lambda-params x) (vector-ref x 1))
(define (user-lambda-body x)   (vector-ref x 2))
(define (user-lambda-env x)    (vector-ref x 3))
---------------------------------------------------------
```

```
----------------------------------------------------------
        *    ev-begin   *
----------------------------------------------------------
example input  :    exp = '(begin 1 2 3)
example output :    val = 3
----------------------------------------------------------
inputs    : exp
outputs   : val
clobber   : argl , exp , unev , val , proc
preserved : cont , env
----------------------------------------------------------
ev-begin :
 unev = cdr [ exp ]
 goto ev-sequence

ev-sequence :
  if null? unev
       set val \# f
       goto continue
  exp = car [ unev ]
  if null? cdr [ unev ]
       ev-sequence-3
  save cont
  save env
  save unev
  cont = ev-sequence-2
  goto eval-dispatch
ev-sequence-2 :
  restore unev
  restore env
  restore cont
  unev = cdr unev
  goto ev-sequence
ev-sequence-3 :
  goto eval-dispatch
----------------------------------------------------------
```

```
---------------------------------------------------------
        *    ev-definition   *
---------------------------------------------------------
example input  :    exp = '(define a 5)
example output :    val = 'ok
---------------------------------------------------------
inputs    : exp
outputs   : val
clobber   : argl , exp , unev , val , proc
preserved : cont , env
---------------------------------------------------------
ev-definition:
 unev = car [ cdr exp ] ;  variable
 save unev
 save env  ; save environment define will need that
 save cont
 exp = car [cdr [ cdr exp ]]  ;; value to eval
 cont = ev-definition-2
 eval-dispatch

ev-definition-2:
restore cont
restore env
restore unev
... primitive ... (env-define! exp val env) ...
val = ok
goto cont
---------------------------------------------------------
```

```
---------------------------------------------------------
        *    ev-assignment   *
---------------------------------------------------------
example input  :    exp = '(set! a 5)
example output :    val = 'ok
---------------------------------------------------------
inputs    : exp
outputs   : val
clobber   : argl , exp , unev , val , proc
preserved : cont , env
---------------------------------------------------------
ev-assignment:
 save exp
 save env
 save cont
 exp = car [cdr [ cdr exp ]]  ;; y
 cont = ev-assignment-2
 eval-dispatch

ev-assignment-2:
restore cont
restore env
restore exp
exp = car [ cdr exp ] ;; x
... primitive operation (env-set! exp val env) ...
val = ok
goto cont
---------------------------------------------------------
```

```
----------------------------------------------------------
        *    ev-if   *
----------------------------------------------------------
example input  :    exp = '(if #f 1 2)
example output :    val = 2
----------------------------------------------------------
inputs    : exp
outputs   : val
clobber   : argl , exp , unev , val , proc
preserved : cont , env
----------------------------------------------------------
ev-if:
 unev = cdr [ cdr exp ]
 save unev
 save env
 save cont
 exp = car [cdr exp ]  ;; x
 cont = ev-if-2
 eval-dispatch
ev-if-2:
restore cont
restore env
restore unev
if val is true
   goto ev-if-3
   goto ev if-4
ev-if-3:
exp = car unev ;; y
goto eval-dispatch
ev-if-4:
unev = cdr unev
if null? unev
   val = \#f  ;; false
   goto continue
otherwise
   exp = car unev
   goto eval-dispatch
----------------------------------------------------------
```

```
------------------------------------------------------------
        *    ev-application   *
------------------------------------------------------------
example input  :    exp = '(factorial 5)
example output :    val = 120
------------------------------------------------------------
inputs    : exp
outputs   : val
clobber   : argl , exp , unev , val , proc
preserved : cont , env
------------------------------------------------------------
ev-application :
unev = cdr exp  ; operands
save unev
save env
save cont
exp = car exp   ; the operator
cont = ev-application-2
goto eval-dispatch

ev-application-2 :
restore cont
restore env
restore unev
proc = val ;
save proc
save env
save cont
cont = ev-application-3
argl = '() ; the empty list
goto ev-operands

ev-application-3 :
restore cont
restore env
restore proc
goto apply-dispatch
------------------------------------------------------------
```

```
---------------------------------------------------------
            *    ev-operands usage   *
---------------------------------------------------------
example input  :    unev = '((+ 10 20 ) (+ 30 40) (+ 50 60)
example output :    argl = '( 110  70  30 )
notes : argl result is REVERSEd .
---------------------------------------------------------
inputs    : unev
outputs   : argl , the arguments are in REVERSE order
clobber   : argl , exp , unev , val , maybe proc
preserved : cont , env
---------------------------------------------------------
ev-operands :
    argl = '()   ; empty list
    goto ev-operands-2

ev-operands-2 :
      if null? unev
          goto cont
      exp = car unev
      unev = cdr unev
      save env
      save unev
      save cont
      save argl
      cont = ev-operands-3
      goto eval-dispatch

ev-operands-3 :
      restore argl
      restore cont
      restore unev
      restore env
      argl = cons val argl  ; adjoin
      goto ev-operands-2
---------------------------------------------------------
```

When we call ev-operands , we want to be sure argl is properly initialised. We observe that the arguments are accumulated in reverse . ARGL values are reversed

The rest of this document contains old notes that may be useful

```
------------------------------------------------------------
         *    reverse  *
piggy back on ev-operands
------------------------------------------------------------
ev-reverse :
   unev = car [ cdr exp ]
   save-cont
   cont = ev-reverse-2
   goto ev-operands

ev-reverse-2 :
   restore-cont
   val = argl
   goto cont
------------------------------------------------------------
```

Ev-operands takes unevaluated arguments in unev , we want evaluated arguments in argl

if no operands are to be evaluated we are done

```
(define (ev-operands exp env cont)
  (if (null? exp)
      (cont '())
      (base-eval (car exp) env
EV-3 :   (lambda (val1)
     (ev-operands (cdr exp) env
EV-4 :     (lambda (val2)
  (cont (cons val1 val2))))))))
```

Motivation

We have a working continuation passing style lisp interpreter and have successfully written call-with-current-continuation.

since callcc is crucial measure of a scheme system.

Also written trace routine that works especially well in cps format , want to see this translated to SICP register machine version

want to prove the simple CPS to register machine translation that we devised as a simple and trustworthy mechanism that requires mental overhead.

eval-application

left to right evaluation of arguments

many ways to do this , here we evaluate the operator and then the operands then apply operator to operands , simple really.

```
(define (eval-application exp env cont)
   (let ((operator (car exp))
     (operands (cdr exp)))
     (base-eval operator env
CONT-2:        (lambda (eop)
    (eval-operands operands env
CONT-3:      (lambda (eargs)
        (base-apply eop eargs env cont)))))))
```

cont1 and cont2 need environment env and continuation cont

if we have unev be unevaluated arguments then exp can be operator in first call to base eval

eop ends up in proc register

eargs ends up in argl register

```
ev-application :
unev = cdr exp  ; operands
save unev
save env
save cont
exp = car exp   ; the operator
cont = ev-application-2
goto eval-dispatch

ev-application-2 :
restore cont
restore env
restore unev
proc = val ;
...
```

now we have the procedure evaluated into proc register , we need to preserve it over the operands evaluation , so we stash it on the stack.

```
save proc
save env
save cont
cont = ev-application-3
argl = '() ; the empty list
goto ev-operands
```

Now want operands to be evaluated and result placed in argl register

```
ev-application-3 :
restore cont
restore env
restore proc
goto apply-dispatch
```

Self evaluating forms just return themselves
evaluating quoted expression is just return the thing that is quoted

```
(define (eval-quote exp env cont)
   (let ((thing-quoted (car (cdr exp))))
     (cont the-thing-quoted)
```

```
1 (define (eval-if exp env cont)
2 (let ((condition (car (cdr exp)))
3       (consequent (car (cdr (cdr exp))))
4 (alternative (cdr (cdr (cdr exp)))))
5   (base-eval condition  env
6     (lambda (bool)
7       (if bool
8           (base-eval consequent env cont)
9     (if (null? alternative)
10         (cont #f)
11 (base-eval (car alternative) env cont)))))))
```

Splits if expression apart , evaluates the conditon , if the condition is true then go on to evaluate the consequent. If the condition is false and there is no alternative , continue with dummy #f false value. otherwise evaluate the alternative in current environment.

continuation at line 6 (lambda (bool) ....) need original env and continuation cont, also need original expression.

ev-if is not called from any other site other than eval-dispatch , good modularity.

```
;; (if x y [z]) - conditional where z is optional
ev-if:
 save exp
 save env
 save cont
 exp = car [cdr exp ]  ;; x
 cont = ev-if-2
 eval-dispatch
ev-if-2:
restore cont
restore env
restore exp
if val is true
   goto ev-if-3
exp = cdr [ cdr [cdr exp ]
if null? exp
   val = \#f  ;; false
   goto continue
otherwise
   exp = car [ exp ]
   goto eval-dispatch
ev-if-3:
exp = car [ cdr [cdr exp]] ;; y
goto eval-dispatch
```

```
1 (define (eval-set exp env cont)
2 (let ((var (car (cdr exp)))
3       (val (car (cdr (cdr exp)))))
4     (base-eval val env (lambda (data)
5                          (env-set! var data env)
6                          (cont 'ok)))))
```

Line 1 :

Line 2 : break up set! expression into variable and the value components

Line 3 :

Line 4 : evaluate the value that will become the result to be assigned to var in the environment

Line 5 : use a primitive procedure that alters the environment

Line 6 : passing a nominal result to show the assignment has been completed. Also useful to prevent code from depending on the result of assignment as assignment can return anything it likes

Looking at the routine we can see that continuation is (lambda (data) ...)

VAR : need that symbol

ENV : environment env is used in the env-set! primitive

CONT : original cont is required also

```
;; (set! x y)
ev-assignment:
 save exp
 save env
 save cont
 exp = car [cdr [ cdr exp ]]  ;; y
 cont = ev-assignment-2
 eval-dispatch

ev-assignment-2:
restore cont
restore env
restore exp
exp = car [ cdr exp ] ;; x
... primitive operation (env-set! exp val env) ...
val = ok
goto cont
```

SICP version uses unev

```
;; (set! x y)
ev-assignment:
 unev = car [ cdr exp ]
 save unev
 save env
 save cont
 exp = car [cdr [ cdr exp ]]  ;; y
 cont = ev-assignment-2
 eval-dispatch

ev-assignment-2:
restore cont
restore env
restore unev
;; unev = x , val = evaluated y
... primitive operation (env-set! exp val env) ...
val = ok
goto cont
```

```
1 (define (eval-define exp env cont)
2 (let ((var (car (cdr exp)))
3       (val (car (cdr (cdr exp)))))
4     (base-eval val env (lambda (data)
5                          (env-define! var data env)
6                          (cont 'ok)))))
```

Line 1 :

Line 2 : break up set! expression into variable and the value components

Line 3 :

Line 4 : evaluate the value that will become the result to be assigned to var in the environment

Line 5 : use a primitive procedure that alters the environment

Line 6 : passing a nominal result to show the assignment has been completed. Also useful to prevent code from depending on the result of assignment as assignment can return anything it likes

Looking at the routine we can see that continuation is (lambda (data) ...)

VAR : need that symbol

ENV : environment env is used in the env-set! primitive

CONT : original cont is required also

```
;; (define f y)
ev-definition:
 save exp
 save env
 save cont
 exp = car [cdr [ cdr exp ]]  ;; y
 cont = ev-definition-2
 eval-dispatch

ev-definition-2:
restore cont
restore env
restore exp
exp = car [ cdr exp ] ;; f
... primitive operation (env-define! exp val env) ...
val = ok
goto cont
```

this version is new tail recursive as last expression in a begin sequence is evaluated without any further added continuations also incorporated a safety valve for empty begin expressions

```
1 (define (eval-begin exp env cont)
2  (let ((body (cdr exp)))
3    (eval-sequence body env cont)))
4
5 (define (eval-sequence body env cont)
6  (cond
7   ;; safety valve - just means (begin) evaluates to false
8   ((null? body) (cont \#f))
9   ;; last expression has no need to return -- key to tail recursion
10   ((null? (cdr body))
11    (base-eval (car body) env cont))
12   (else
13    (base-eval (car body) env
14        (lambda (ignored)
15  ;; totally ignore the result
16  (eval-sequence (cdr body) env cont))))))
```

lines 1 to 3 going evaluate the expressions of the begin sequence so seems appropriate to set unev to be the body of the begin sequence Line 3 : simple jump to eval sequence

eval-begin : (set! unev (cdr exp)) goto eval-sequence

ev-sequence : Line 8 : if unev is null , meaning nothing to evaluate then the result of whole begin sequence is undefined. let it be false #f . (if (null? unev) (cont #f))

set! val #f goto continue

Line 10 : if this is the last expression , then all we need to do is pass this to base-eval , it never returns.

set! exp (car unev) goto eval-dispatch ; which is base-eval

Line 13 : otherwise evaluate the expression in the begin sequence and remember to come back to evaluate the rest

set! exp (car unev) we want to come back to somewhere where we can tell it to evaluate the rest , ev-sequence-2 seems appropriate set! cont ev-sequence-2

but we need original cont to carry on evaluating the sequence we also need original environment we also need the other unevaluated arguments also

save-cont save-env save-unev set! exp (car unev) set! cont ev-sequence-2 goto eval-dispatch ev-sequence-2 : line 14 : we observe see each lambda in continuation passing style becomes a new goto label location in register machine basic stack discipline , we restore the stack to its original state

restore-unev restore-env restore-cont

(cdr body) set! unev (cdr unev)

line 16 : environment and continuation are the same as they are now , so we jump to ev-sequence goto ev-sequence

Here is what the translation looks like

```
(define (ev-begin)
  (set! unev (cdr exp))
  (ev-sequence))

(define (ev-sequence)
  (set! exp (car unev))
  (cond
   ((null? unev)
    (set! val \#f)
    (cont))
   ((null? (cdr unev))
    (ev-sequence-last-exp))
   (else
S1:    (save-cont)      ;; ***
S2:    (save-env)       ;; ***
S3:    (save-unev)      ;; ***
    (set! cont ev-sequence-2)
    (eval-dispatch))))

(define (ev-sequence-2)
R3:  (restore-unev)     ;; ***
R2:  (restore-env)      ;; ***
R1:  (restore-cont)     ;; ***
  (set! unev (cdr unev))
  (ev-sequence))

(define (ev-sequence-last-exp)
  (eval-dispatch))
```

ev-sequence HOWEVER is called from other sites in addition to eval-dispatch. This is AWFUL modularity , means we cannot think about ev-sequence in isolation but must then look at who calls ev-sequence .

notice that the saves S1 S2 S3 and restores R3 R2 R1 must happen in reverse order to each other. it doesnt matter if we save any cont env or unev , but they must be preserved on restore to their original values – stack discipline.

Observed in SICP book version saves S1 and restores R1 have been hoisted out but this has far reaching ramifications to everything that uses ev-sequence - due to stack discipline.

```
(define (ev-begin)
  (set! unev (cdr exp))
S1:  (save-cont)        ;; *****
  (ev-sequence))

(define (ev-sequence)
  (set! exp (car unev))
  (cond
   ((null? unev)
    (set! val \#f)
    (cont))
   ((null? (cdr unev))
    (ev-sequence-last-exp))
   (else
S2:    (save-env)
S3:    (save-unev)
    (set! cont ev-sequence-2)
    (eval-dispatch))))

(define (ev-sequence-2)
R3:  (restore-unev)
R2:  (restore-env)
  (set! unev (cdr unev))
  (ev-sequence))

(define (ev-sequence-last-exp)
R1:    (restore-cont)   ;; *****
  (eval-dispatch))
```