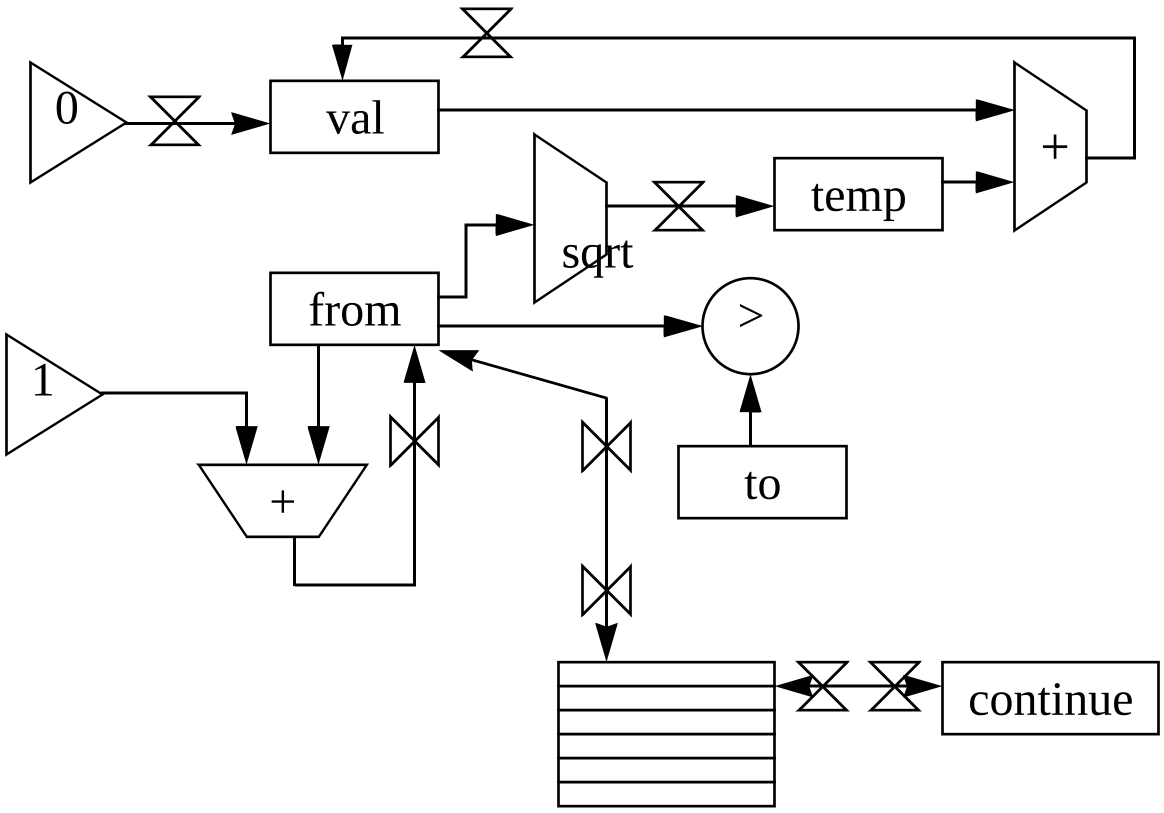
Special-Purpose Register Machine

```
; Recursive version
(define (sum-roots from to)
  (if (> from to)
      0                  ;base case
      (+ (sqrt from)         ;deferred operation
         (sum-roots (+ 1 from) to)))) ;recursion
```



Assembly Language

```
(assign <reg-name1> (reg <reg-name2>))
(assign <reg-name> (const <constant-value>))
(assign <reg-name> (op <op-name>) <input1> <input2> ...)
(assign <reg-name> (label <label-name>))

(perform (op <op-name>) <input1> <input2> ...)

(test (op <op-name>) <input1> <input2> ...)
(branch (label <label-name>))
(goto (label <label-name>))
(goto (reg <reg-name>))

(save <reg-name>)
(restore <reg-name>)
```

Notes:

```
<inputi> is either (const <constant-value>) or
                   (reg <reg-name>)
```

Calling Convention

- Save any registers that will be needed later on stack
    - Put label for continuation in the **`continue`** register
        - Goto subroutine entry point (label)
        - Do subroutine computations
        - Put result in the **`val`** register
    - Goto location specified by **`continue`** register
- Restore save registers from the stack

Recursive Sum-Roots (With Stack)

```
(define-machine sum-roots
 (registers continue from to temp)
 (operations + sqrt >)
 (controller
  ;; On entry  -- continue holds return label
  ;;           -- registers from, to hold input values
  ;; On return -- register val holds answer
  sum-roots
    (test (op >) (reg from) (reg to))
    (branch (label base-case))
    ;; Need to recurse, so remember what we'll need
    ;; for the deferred operation...
    (save from)
    (save continue)
    (assign continue (label do-deferred-operations))
    (assign from (op +) (const 1) (reg from))
    (goto (label sum-roots))  ; recurse
  base-case
    (assign val (const 0))
    (goto (reg continue))
  do-deferred-operations
    (restore continue)  ;; restore in reverse order!
    (restore from)
    (assign temp (op sqrt) (reg from))
    (assign val (op +) (reg val) (reg temp))
    (goto (reg continue))))
```

Implementing a Scheme Interpreter in Assembly Language

1. Registers

- exp	The expression to be evaluated
- env	The environment in which to find variable vals
- val	Where the value is stored when **`eval`** is done
- continue	The label to resume at when **`eval`** is done
- argl	The arguments to a procedure to be applied
- proc	The procedure that is to be applied
- unev	Either uneval'd subexpressions of a combination (which go into **`proc`** and **`argl`** after they are evaluated), or the subexpressions of a **`begin`** expression that is being evaluated

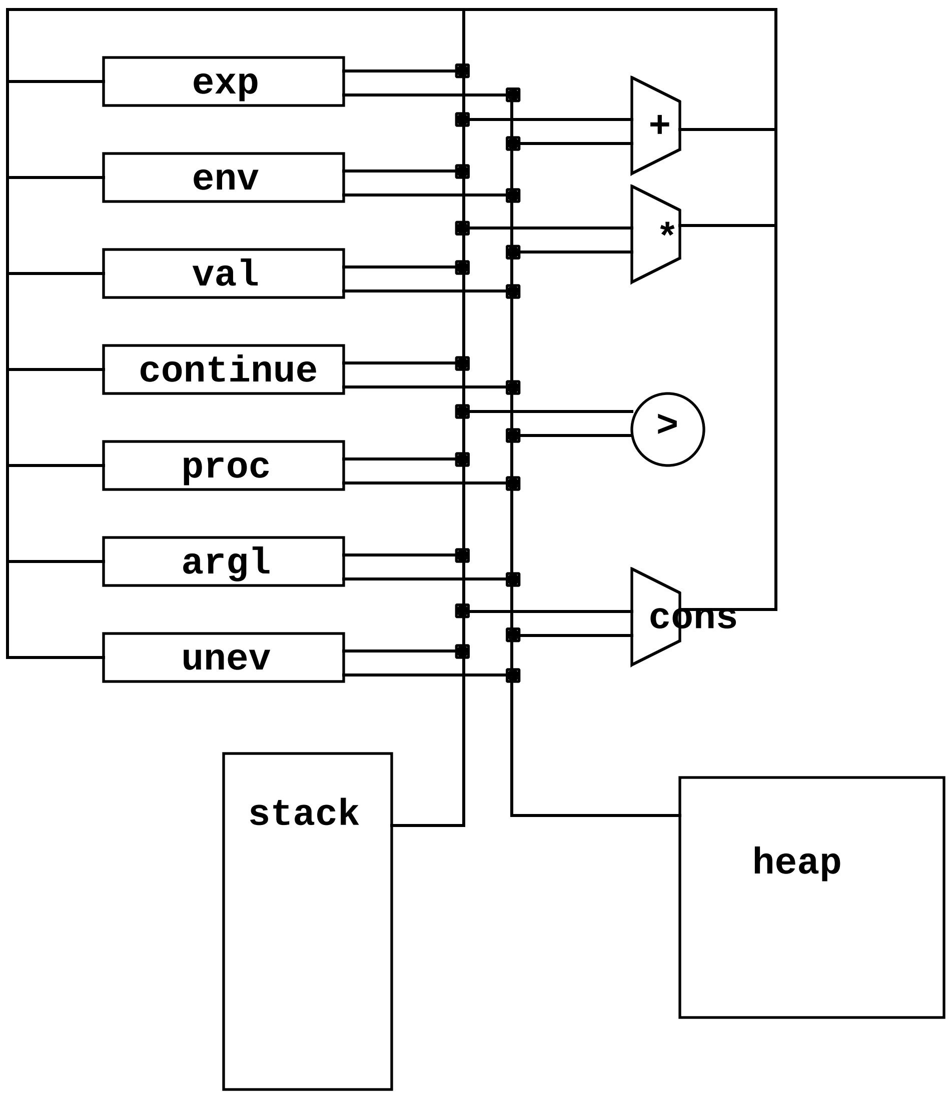
2. Primitive Operations:
   - Basic mathematical operations (e.g. **`+, -, *, /`**)
   - Predicates (e.g. **`=, >, <`**, etc.)
   - Compound data operations (e.g. **`cons, car, cdr, null?`**)
   - Environment manipulation (e.g. **`lookup-variable, extend-environment`**)
   - Expression syntax operations (e.g. **`if-predicate`**)

3. Stack

4. Heap - random access memory for storage of cons cells and other objects

Scheme Interpreter - Register Machine

Data Flow:



Controller: To be defined...

Meta-Circular Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Eval-dispatch Contract

1. `exp` has an expression to be evaluated
2. `env` has an environment to be used for the evaluation
3. `continue` has a label to be used to deliver the result
4. When the value has been computed, it will be placed in the `val` register and computing will continue at the label stored in the `continue` register
5. The stack will be returned to its state at the beginning of `eval-dispatch` (that is, anything added to the stack will have been restored back off the stack, and nothing originally on the stack will have been removed from the stack)
6. Trust No One - any other registers may be obliterated!

Eval-dispatch

```
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))
  (test (op assignment?) (reg exp))
  (branch (label ev-assignment))
  (test (op definition?) (reg exp))
  (branch (label ev-definition))
  (test (op if?) (reg exp))
  (branch (label ev-if))
  (test (op lambda?) (reg exp))
  (branch (label ev-lambda))
  (test (op begin?) (reg exp))
  (branch (label ev-begin))
  (test (op cond?) (reg exp))
  (branch (label ev-cond))
  (test (op let?) (reg exp))
  (branch (label ev-let))
  (test (op application?) (reg exp))
  (branch (label ev-application))
  (goto (label unknown-expression-type-error))
```

Evaluation of Simple Expressions

```
ev-self-eval
  (assign val (reg exp))
  (goto (reg continue))

ev-quoted
  (assign val (op text-of-quotation) (reg exp))
  (goto (reg continue))

ev-variable
  (assign val (op lookup-variable-value)
              (reg exp) (reg env))
  (goto (reg continue))

ev-lambda
  (assign unev (op lambda-parameters) (reg exp))
  (assign exp (op lambda-body) (reg exp))
  (assign val (op make-procedure)
              (reg unev) (reg exp) (reg env))
  (goto (reg continue))
```

Eval-if

Strategy:

1. Make a recursive call to evaluate the predicate
   -> Deferred operations: the consequent or the alternative

2. Turn control over to **`eval-dispatch`** to handle either the alternative or the consequent
   -> Do NOT make a recursive call -- no deferred ops!

```
ev-if
  (assign unev (reg exp))
  (save unev)          ;save expression for later
  (save env)
  (save continue)
  (assign continue (label ev-if-decide))
  (assign exp (op if-predicate) (reg unev))
  (goto (label eval-dispatch));evaluate the predicate

ev-if-decide
  (restore continue)
  (restore env)
  (restore unev)
  (test (reg val))
  (branch (label ev-if-consequent))
ev-if-alternative
  (assign exp (op if-alternative) (reg unev))
  (goto (label eval-dispatch))
ev-if-consequent
  (assign exp (op if-consequent) (reg unev))
  (goto (label eval-dispatch))
```

Apply-dispatch Contract

1. `proc` contains a procedure (primitive or compound) that is to be applied
2. `argl` contains the arguments to the procedure (in reverse order!)
3. The top of the stack contains a label to be used to deliver the result
4. When the value has been computed, it will be placed in the `val` register, and computation will continue at the location originally stored at the top of the stack
5. The stack will have had one item (the `continue` label) removed from it. Nothing on the stack when `apply-dispatch` started will have been changed
6. Trust No One - any other registers may be obliterated

Apply-dispatch

```
apply-dispatch ;expects continuation to be on the stack
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-apply))
  (test (op compound-procedure?) (reg proc))
  (branch (label compound-apply))
  (test (op compiled-procedure?) (reg proc))
  (branch (label compiled-apply))
  (goto (label unknown-procedure-type-error))

primitive-apply
  (assign val (op apply-primitive-procedure)
              (reg proc) (reg argl))
  (restore continue)
  (goto (reg continue))

compound-apply
  (assign unev (op procedure-parameters) (reg proc))
  (assign env (op procedure-environment) (reg proc))
  (assign env (op extend-environment)
              (reg unev) (reg argl) (reg env))
  (assign unev (op procedure-body) (reg proc))
  (goto (label ev-sequence))
```

Evaluating an Application

Strategy: "Evaluate all subexpressions, then apply the value of the first to the values of the rest."

1. Evaluate the operator (using a subroutine call to **eval-dispatch**). Remember our calling convention & **eval-dispatch** contract!

```
ev-application
  (save continue)     ;dest for after application
  (save env)
  (assign unev (op operands) (reg exp))
  (save unev)         ;the operands
  (assign exp (op operator) (reg exp))
  (assign continue (label ev-appl-did-operator))
  (goto (label eval-dispatch)) ;eval the operator
```

2. Once we have the operator, we will save it away (in **proc**) and start working on the operands. Notice that the operands (in **unev**) and the environment are restored, but not the continuation - which remains on the stack.

```
ev-appl-did-operator  ;deferred ops - handle operands
  (restore unev)                    ;the operands
  (restore env)
  (assign argl (op empty-arglist))  ;init argl
  (assign proc (reg val))           ;the operator
  (test (op no-operands?) (reg unev))
  (branch (label apply-dispatch))  ;ready to apply!
  (save proc)        ;else we need to eval operands
```

3. Save the arguments that have been computed so far (and are in **argl**) so we can add on to them when we have the value of the next operand. See if this is the last operand. If so, handle it specially (because after the last one we don't need to save **env** or **unev** any more).

```
ev-appl-operand-loop
  (save argl)
  (assign exp (op first-operand) (reg unev))
  (test (op last-operand?) (reg unev))
  (branch (label ev-appl-last-arg))
  (save env)  ;prepare for subroutine call to compute
  (save unev) ; the next operand value
  (assign continue (label ev-appl-accumulate-arg))
  (goto (label eval-dispatch))
```

4. We have one more argument value, so get back the list of arguments remaining to evaluate (**unev**), the environment for the next argument (**env**), and the list of argument values already computed (**argl**). Add the value just computed to the list of computed arguments, remove one argument from the list remaining to compute, and return to step 3.

```
ev-appl-accumulate-arg
  (restore unev)
  (restore env)
  (restore argl)
  (assign argl (op adjoin-arg) (reg val) (reg argl))
  (assign unev (op rest-operands) (reg unev))
  (goto (label ev-appl-operand-loop))
```

5. Special case for computing the value of the last argument: don’t save anything, and use a different continuation label. Remember that the original continuation for the combination is still on the stack from step 1!

```
ev-appl-last-arg
  (assign continue (label ev-appl-accum-last-arg))
  (goto (label eval-dispatch))
```

6. We’ve got the value of the last argument now. So get back all of the computed argument values (`argl`), add this last one to the list, get back the procedure to call (`proc`), and go do the application. Remember that the original continuation for the combination is still on the stack from step 1 - so we conform to the contract for `apply-dispatch`

```
ev-appl-accum-last-arg
  (restore argl)
  (assign argl (op adjoin-arg) (reg val) (reg argl))
  (restore proc)
  (goto (label apply-dispatch))
```

Eval-sequence

```
ev-sequence
  (assign exp (op first-exp) (reg unev))
  (test (op last-exp?) (reg unev))
  (branch (label ev-sequence-last-exp))
  (save unev)
  (save env)
  (assign continue (label ev-sequence-continue))
  (goto (label eval-dispatch))
ev-sequence-continue
  (restore env)
  (restore unev)
  (assign unev (op rest-exps) (reg unev))
  (goto (label ev-sequence))
ev-sequence-last-exp
  (restore continue)
  (goto (label eval-dispatch))
```

Ev-sequence - With tail recursion broken

```
ev-sequence
  (test (op no-more-exps?) (reg unev))
  (branch (label ev-sequence-end))
  (assign exp (op first-exp) (reg unev))
  (save unev)
  (save env)
  (assign continue (label ev-sequence-continue))
  (goto (label eval-dispatch))
ev-sequence-continue
  (restore env)
  (restore unev)
  (assign unev (op rest-exps) (reg unev))
  (goto (label ev-sequence))
ev-sequence-end
  (restore continue)
  (goto (reg continue))
```