Scheme

Scheme

- Lisp Dialect.
- Expression Oriented.
- Functional style (not pure).
- Interpreted, but compilers exist.
- Simple but expressive.

Expressions

A Scheme program is a sequence of expressions.
(Also called *S-Expressions* for symbolic)

S-Expressions are recursively Defined as either:

- An *atom*: a variable or value.
- A list of S-Expressions.

Primitive Values: Atoms

Atoms can be:

- Numbers: Integers, Reals, Rationals (e.g. `42`, `3.14`, `22/7`).
- Symbols (e.g. `fred`, `g42`, `*`).
- Booleans (e.g. `#t` or `#f`).
- Strings (e.g. `"hello world"`)
- Characters (e.g. `#\c`, `#\C` , `#\newline`)

Combinations

A combination is a function *application*.
(looks like a list of S-Expressions)

In general: (<f-expr> <expr-1> <expr-2> ...)

For example: (+ 5 6) or (display "hello world")

Read-Eval-Print Loop

Interactive interpreter's infinite loop:

1. *Read* an S-Expression.
2. *Evaluate* the S-Expression (in the *global environment*).
3. *Print* the expression's value.

Note: an *environment* is a lookup table for variable values.

Eval: Atoms

Variables: (if the expression is a symbol)

- Look up variables value in *current environment*.
- Return the value.

Values: (Any other primitive type)

- Return it.

Eval: *Combinations*

Given: (<f-expr> <expr-1> <expr-2> ...)

- Evaluate all expressions to get $f, e_1, e_2, \ldots$
- Apply f as a function to arguments $e_1, e_2, \ldots$
- Return the result of the function.

Note: Scheme operators are *prefix* not *infix*.

Evaluate?

- `1`
- `"hi there"`
- `(+ 5 4)`
- `(1 2 3)`
- `(+ (* 2 3) 4)`

Primitive Procedures

Builtin functions defined in global environment.

Arithmetic: `+`, `-`, `*`, `/`, `sqrt`, and `abs`
For example: `(+ 1 2 3 4)` or `(*)`
Relational (numbers): `=`, `<`, `>`, `<=`, and `>=`
For example: `(< 1 2)` or `(< 1 2 3)` or `(<)`
Predicates: `number?`, `string?`, `procedure?`, etc.
For example: `(procedure? +)` or `(number? +)`

... and many more.

Defining Variables

Just like this: `(define x 5)`

Or even: `(define answer (* 1 2 3 4))`

But how does the evaluator handle this?

Special Forms

Not every list of S-Expressions is a combination.

If first element of list is symbol naming a *special form*. Then evaluate special form in a special way.

code define is a special-form.

Why is this necessary?

Conditionals ...

conditionals: `(if <test> <consequent> <alternate>)`

For Example:

```
(if (> x 5)
    5
    x
)
```

boolean relations: `or` and `and`

For Example: `(or (< 4 3) #t (/ 1 0))`

What would happen if these were evaluated like *combinations*?

Defining Functions

Special form for defining functions:

```
(define (<fname> <formal-1> <formal-2> ...) <body>)
```

For example:

```
(define (double x) (* 2 x))
```

Note: `define` has two special forms. One for functions and one for other vraiables.

Temporary Variables

Special form: `let`

```
(let ((<var-1> <expr-1>)
      (<var-2> <expr-2>)
      ...
      (<var-n> <expr-n>)
     )
     <body>
)
```

For Example:

```
(let ((x 50)
      (y 10)
     )
     (+ x y)
)
```


Quoting

How do we get symbols for values? E.g. `(define plus-sym +)`

Answer: Use `quote` special form: `(define plus-sym (quote +))`

Or use a simpler special notation: `(define plus-sym '+)`

Quote tells evaluator not to evaluate. Just to return the symbolic value of the S-expression.

What is `(number? (quote 5))`?

Lists

Create the list `(1 "hi" bob)` with:

```
(list 1 "hi" 'bob)
```

Functions: `list`, `length`, `equal?`, `car`, `cdr`, `cons`, etc.
Predicates: `list?`, `pair?`, and `null?`.

Creating Lists:

```
(cons 1 (cons "hi" (cons 'bob '())))
'(1 "hi" bob)
(quote (1 "hi" bob))
```


`cons`, `car`, `cdr`

Who came up with these names?

`cons` stands for *construct*.

From original Lisp implementation on IBM 704 where "address" or "decrement" part of memory could be accessed.

`car` stands for "Contents of Address part of Register"
`cdr` stands for "Contents of Decrement part of Register"

Why do we still use these names? Because of contractions:

```
(define (cadr x) (car (cdr x)))     ;; second element of list
(define (cdar x) (cdr (car x)))     ;; ??
(define (cadaaar x) (car (cdr (car (car (car x))))))
```


List Dissecting

```
(define (sum lst)
   (if (null? lst)
       0                                ; base case
       (+ (car lst) (sum (cdr lst)))    ; recursive call
   )
)
```

and

```
(define (double-list lst)
   (if (null? lst)
       '()
       (cons (double (car lst)) (double-list (cdr lst)))
   )
)
```


Linear Recursive Process

```
guile> (sum '(1 2 3))
```

```
(sum '(1 2 3))
=> (+ 1 (sum '(2 3)))
=> (+ 1 (+ 2 (sum '(3))))
=> (+ 1 (+ 2 (+ 3 (sum '()))))
=> (+ 1 (+ 2 (+ 3 0)))
=> (+ 1 (+ 2 3))
=> (+ 1 5)
=> 6
```

Linear Recursion builds up *call stack* to depth of recursion.

Tail Recursion

If last thing a function does is make a recursive call, that recursive call's *stack frame* replaces the current functions stack frame.

```
;; return last element in list
(define (last lst)
   (if (null? (cdr lst))
       (car lst)
       (last (cdr lst))
   )
)
```

The function `last` is *tail-recursive*

Iterative Process

```
guile> (last '(1 2 3))
```

```
(last '(1 2 3))
=> (last '(2 3))
=> (last '(3))
=> 3
```


Tail Recursive `sum`

Can we make `sum` tail recursive? (hint: use a helper function)

Trees

Lists can contain lists.

```
'(1 (2 3) ((4) 5))
```

Tree Recursion:

```
(define (sum-tree t)
   (cond ((null? t)    0)
         ((number? t) t)
         ((pair? t)    (+ (sum-tree (car t)) (sum-tree (cdr t))))
         (else (error "non-number in tree"))
   )
)
```

Lists and Memory

Appending two lists:

```
(define (append first second)
   (if (null? first)
       second
       (cons (car first) (append (cdr first) second))
   )
)
```

What happens below?

```
(define x '(1 2 3))
(define y '(a b c))
(define z (cdr x))
(define w (append x y))
```


Numbers

Three types of numbers *integers*, *reals*, and *rationals*.

There are different internal representations for the same number.

Integers can be any size.

Ever wonder what `(factorial 1000)` is?

Equality

Testing for equality:

`eq?` tests pointer equality. Equal numbers may not be `eq?`

`eqv?` tests pointer and number equality.

`equal?` tests *deep* equality (lists, trees, numbers, not procedures).

`=` tests number equality only. Non numbers are errors.

Equality Examples

How about?

```
(eq? 5 5)
(eq? 'the-little-red-hen 'the-little-red-hen)
(eq? "hi there" "hi there")
(eq? (list 1 2) (list 1 2))
(eq? (factorial 200) (factorial 200))
(eqv? 5 'a)
(eqv? (factorial 200) (factorial 200))
(equal? (list 1 2) '(1 2))
(= 5 'a)
```