Scheme

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

Expressions

Also called S-Expressions (for symbolic).

Recursively Defined, either:

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

Primitive Values: Atoms

Atoms can be:

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

Combinations

(A list of S-Expressions)

A combination is a function application.

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.

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.

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.

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))`

Defining Functions

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

For example:

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


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`, `(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 atom.

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

Lists

```
(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))
```

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 Recursion

```
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
```


Tail Recursion

Linear Recursion builds up call stack to depth of 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))
   )
)
```


Linear Recursion

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

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

Tail Recursive `sum`

Can we make `sum` tail recursive?

Trees

Lists can contain lists.

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

Tree Recursion:

```
(define (sum-tree t)
   (if (null? t)
       0
       (+ (if (number? (car t))
              (car t)              ; it is a number
              (sum-tree (car t))   ; it must be a list
              )
          (sum-tree (cdr t))
       )
   )
)
```