Scheme SICP HtDP ;-)

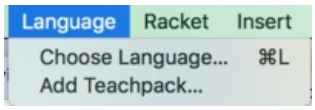
JavaScript Python

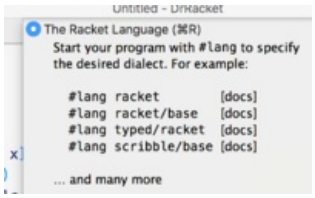
“R2”

Racket

Scheme Scheme “” Racket Racket pattern matching Scheme Racket Scheme

Racket macro DrRacket“” R5RS DrRacket “” Language “Racket”





Racket

```
(let ([x 1]
      [y 2])
  (+ x y))
```

“” `[x 1], [y 2]`“”“”

Racket `#lang racket` Racket

“” “”

`'(+ 1 2) 3`“”“”“”“”“”“”

“S ”S-expression `'(+ 1 2)` listsymbol+, `1 2"(+ 1 2)"`

S “”SchemeLisp Lisp Unix

“”“”“”

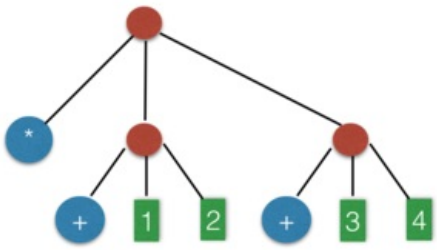
CPU CPU

Abstract Syntax Tree

S “”tree“”Abstract Syntax Tree AST“”“”

“”“”“”“”

```
'(* (+ 1 2) (+ 3 4))
```




```
*+1234 '(+ 1 2)'(+ 3 4)'(* (+ 1 2) (+ 3 4))
```

tree-sum“”(tree-sum '((1 2) (3 4))) 10 2 ((1 2) (3 4 5)) (1 2)(1 (2 3)), ((1 2) 3) ((1 2) (3 4))……

```
#lang racket

(define tree-sum
  (lambda (exp)
    (match exp                          ; exp
      [(? number? x) x]                 ; expxx
      [`(,e1 ,e2)                       ; exp
       (let ([v1 (tree-sum e1)]         ; tree-sume1
             [v2 (tree-sum e2)])        ; tree-sume2
         (+ v1 v2))])))                 ; v1v2


(tree-sum '(1 2))
;; => 3
(tree-sum '(1 (2 3)))
;; => 6
(tree-sum '((1 2) 3))
;; => 6
(tree-sum '((1 2) (3 4)))
;; => 10
```

1. exp
2. exp (,e1 ,e2) e1 e2 tree-sum v1 v2 v1 + v2

if cond Racket match if cond match match

Racket Racket Racket

```
(match x
  [ ]
  [ ]
   ...   ...
)
```

x match x

SchemeLisp cond Java if ... else if ... else ...“” match “”accessor foo.x attribute

match

```
(match exp
  [(? number? x) x]
  [`(,e1 ,e2)
   (let ([v1 (tree-sum e1)]
         [v2 (tree-sum e2)])
     (+ v1 v2))])
```

`'(,e1 ,e2)` pattern `exp` `exp` `'(1 2)` `'(,e1 ,e2)` `e1` `'1` `e2` `'2`

```
`(,e1 ,e2)
'(  1   2)
```

“” `e1` `e2` `'(,e1 ,e2)` `'(1 2)`

“”`(? number? x)` `(number? exp)` `exp` `x` `x` `exp“”`

ML OCamlHaskell MLMeta-LanguageRacket ML

`'(* (+ 1 2) (+ 3 4))` `21`

`'(* (+ 1 2) (+ 3 4))` `'((1 2) (3 4))` `*` `+`

“”

```
#lang racket                                ;  Racket

(define calc
  (lambda (exp)
    (match exp                              ;
      [(? number? x) x]                     ;
      [`(,op ,e1 ,e2)                       ; ope1,e2
       (let ([v1 (calc e1)]                 ;  calc  e1
             [v2 (calc e2)])                ;  calc  e2
         (match op                          ;  op  4
           ['+ (+ v1 v2)]                   ;  (+ v1 v2)
           ['- (- v1 v2)]                   ;
           ['* (* v1 v2)]
           ['/ (/ v1 v2)]))])))



(calc '(+ 1 2))
;; => 3
(calc '(* 2 3))
;; => 6
(calc '(* (+ 1 2) (+ 3 4)))
;; => 21
```

1. “”op`(,op ,e1 ,e2)`
2. e1 e2 `(+ v1 v2)` `op`

```
   (match op
     ['+ (+ v1 v2)]
     ['- (- v1 v2)]
     ['* (* v1 v2)]
     ['/ (/ v1 v2)])
```

R2

R2R2 5 Scheme 5

- x
- (lambda (x) e)
- (let ([x e1]) e2)
- (e1 e2)
- (• e2 e2)

- +, -, *, /

Scheme ”””“

“first-class function”

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

```
(((lambda (x)
    (lambda (y) (+ x y)))
  1)
 2)
;; => 3
```

PLcurrying

`(let ([x e1]) e2)` Scheme let Scheme `(let ([x 1] [y 2]) (+ x y))` let

```
(let ([x 1])
  (let ([y 2])
    (+ x y)))
```

R2

R2

```
#lang racket

;;;  env0, ext-env, lookup environment

;;
(define env0 '())

;;  env  x  v
(define ext-env
  (lambda (x v env)
    (cons `(,x . ,v) env)))

;;  env  x  #f
(define lookup
  (lambda (x env)
    (let ([p (assq x env)])
      (cond
       [(not p) #f]
       [else (cdr p)]))))

;;  f
(struct Closure (f env))

;;  exp  env
;;  5
(define interp
  (lambda (exp env)
    (match exp                                        ; exp
      [(? symbol? x)                                  ;
       (let ([v (lookup x env)])
         (cond
          [(not v)
           (error "undefined variable" x)]
          [else v]))]
      [(? number? x) x]                               ;
      [`(lambda (,x) ,e)                              ;
       (Closure exp env)]
```

```
      [`(let ([,x ,e1]) ,e2)                          ;
       (let ([v1 (interp e1 env)])
         (interp e2 (ext-env x v1 env)))]
      [`(,e1 ,e2)                                     ;
       (let ([v1 (interp e1 env)]
             [v2 (interp e2 env)])
         (match v1
           [(Closure `(lambda (,x) ,e) env-save)
            (interp e (ext-env x v2 env-save))]))]
      [`(,op ,e1 ,e2)                                 ;
       (let ([v1 (interp e1 env)]
             [v2 (interp e2 env)])
         (match op
           ['+ (+ v1 v2)]
           ['- (- v1 v2)]
           ['* (* v1 v2)]
           ['/ (/ v1 v2)]))]))))

;; “” interp  env0
(define r2
  (lambda (exp)
    (interp exp env0)))



(r2 '(+ 1 2))
;; => 3

(r2 '(* 2 3))
;; => 6

(r2 '(* 2 (+ 3 4)))
;; => 14

(r2 '(* (+ 1 2) (+ 3 4)))
;; => 21

(r2 '((lambda (x) (* 2 x)) 3))
;; => 6

(r2
'(let ([x 2])
   (let ([f (lambda (y) (* x y))])
     (f 3))))
;; => 6

(r2
'(let ([x 2])
   (let ([f (lambda (y) (* x y))])
     (let ([x 4])
       (f 3)))))
;; => 6

)
```

R2 `interp`

```
(match exp
  ... ...
  [`(,op ,e1 ,e2)
   (let ([v1 (interp e1 env)]             ;  interp  e1
         [v2 (interp e2 env)])            ;  interp  e2
     (match op                            ;  op  4
       ['+ (+ v1 v2)]                     ;  (+ v1 v2)
       ['- (- v1 v2)]                     ;
       ['* (* v1 v2)]
       ['/ (/ v1 v2)]))])
```

`interp env env “”`

```
[(? number? x) x]
```

variable `f(x) = x * 2` `x  x * 2`

“”binding“”evaluate `f(x)  f(1)  x` 1 `x * 2` 2 `f(2)  x` 2 `x * 2` 4 `f  x  x` 1 2

“” `f(x)  x` `x * 2  x`

```
f(1)
```

1. `x` 1
2. `f  x * 2`

1.
2. 2

`x` “”“”“”“”scope

```
;;
(define env0 '())

;;  env  x  v
(define ext-env
  (lambda (x v env)
    (cons `(,x . ,v) env)))

;;  env  x
(define lookup
  (lambda (x env)
    (let ([p (assq x env)])
      (cond
       [(not p) #f]
       [else (cdr p)]))))
```

Scheme association listAssociation list `((x . 1) (y . 2) (z . 5))`pair key value

```
((x . 1)
 (y . 2)
 (z . 5))
```

key pair

`ext-env env1 ((y . 2) (x . 1)) (ext-env x 3 env1) ((x . 3) (y . 2) (x . 1)) (x . 3)  env1`

let scope

stack“”

```
(let ([x 1])         ; env='()x1
  (let ([y 2])       ; env='((x . 1))y2
    (let ([x 3])     ; env='((y . 2) (x . 1))x3
      (+ x y))))     ; env='((x . 3) (y . 2) (x . 1))x3y2
;; => 5
```

5 `(x . 3)  x  (x . 3)`3 `(x . 1) (x . 3)`

`(x . 1)`

```
(let ([x 1])          ; env='()x1
  (+ (let ([x 2])     ; env='((x . 1))x2
       x)             ; env='((x . 2) (x . 1))x2
   x))                ; env='((x . 1))x1
;; => 3               ; x1+23
```

334 x 3 x let `((x . 2) (x . 1))` x 24 x let let `((x . 1))` x 1 x

`((y . 2) (x . 1))` “”

mutation“”“”immutable

“” `match`

```
[(? symbol? x) (lookup x env)]
```

“”

`(? symbol? x)` Scheme `symbol? x` x

let

```
[`(let ([,x ,e1]) ,e2)
 (let ([v1 (interp e1 env)])              ; e1v1
   (interp e2 (ext-env x v1 env)))]       ; (x . v1)e2
```

`e1` `v1` `(x . v1)` `(let ([x e1]) ...)` x let `e2` let

Lexical Scoping Dynamic Scoping

“”“”scoping

```
(let ([x 2])
  (let ([f (lambda (y) (* x y))])
    (let ([x 4])
      (f 3))))
```

`f (lambda (y) (* x y))` x“”x x

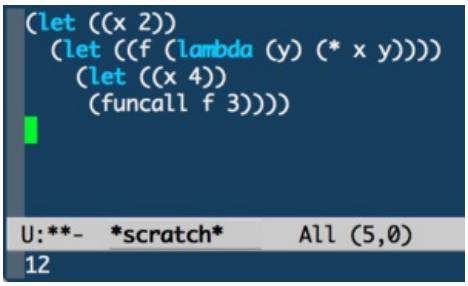
x 24 x `(f 3)` `f (* x y)` `y` 3 x 24

Scheme Racket6

```
;; Scheme
(let ([x 2])
  (let ([f (lambda (y) (* x y))])
    (let ([x 4])
      (f 3))))

;; => 6
```

Emacs Lisp Emacs Lisp Emacs `*scratch*` buffer C-x C-e Emacs minibuffer



12 x

Scheme Emacs Lisp “Lisp ”Scheme lexical scoping static scoping Emacs dynamic scoping

dynamic scoping bug dynamic scoping

`(let ((x 4)) …)` “” `x` `(let ((x 4)) (f 3))` let “x” “”`(let ((x 4)) …)` `x` `(f 3)`

dynamic scoping `f x (f 3) f f f x` dynamic scoping :)

lexical scoping `(let ((x 4)) (f 3)) x` 4 `f x f x x (let ([x 2]) ...)` 2 `(f 3)` 612

lexical scoping“”closure Racket struct

```
(struct Closure (f env))

 (lambda (x) e)

[`(lambda (,x) ,e)
 (Closure exp env)]
```

`exp` ``(lambda (,x) ,e)`

`(lambda (x) e)`Closure“”

```
[`(,e1 ,e2)
 (let ([v1 (interp e1 env)]             ;  e1
       [v2 (interp e2 env)])            ;  e2
   (match v1
     [(Closure `(lambda (,x) ,e) env-save)      ;
      (interp e (ext-env x v2 env-save))]))]    ; env-savexv2
```

`(e1 e2)` e1 e2 e1 e2

`(lambda (x) (* x 2))` 1 x 1 `(* x 2)`

`e1 v1 env-save` env-save env-save `(ext-env x v2 env-save)` env-save

`env env e1 e2 e1 e2` “”env `env-save v1 env-save`

`envenv-save` dynamic scoping `(interp e (ext-env x v2 env-save)) env-save env` 12dynamic scoping

```
(r2
'(let ([x 2])
   (let ([f (lambda (y) (* x y))])
     (let ([x 4])
       (f 3)))))

;; => 12
```

dynamic scopingdynamic scoping

```
(define interp
  (lambda (exp env)
    (match exp
```

```
      ... ...
      [`(lambda (,x) ,e)                          ;
       exp]
      ... ...
      [`(,e1 ,e2)
       (let ([v1 (interp e1 env)]
             [v2 (interp e2 env)])
         (match v1
           [`(lambda (,x) ,e)                     ;
            (interp e (ext-env x v2 env))]))]
      ... ...
)))
```

dynamic scoping

Lisp Emacs Lisp dynamic scoping dynamic scoping lexical scoping

“”“”

lambda calculus `(let ([x e1]) e2)` `((lambda (x) e2) e1)` lexical scoping dynamic scoping let lexical scoping dynamic scoping

1. ......
2. match
3. association list index
4. S S S S

   `(,op ,e1 ,e2)` `(+ 1 2)` let `(let ([x 1]) (* x 2))` `(let ([,x ,e1]) ,e2)` `(,op ,e1, e2)` S “parser” parser S Racket struct struct