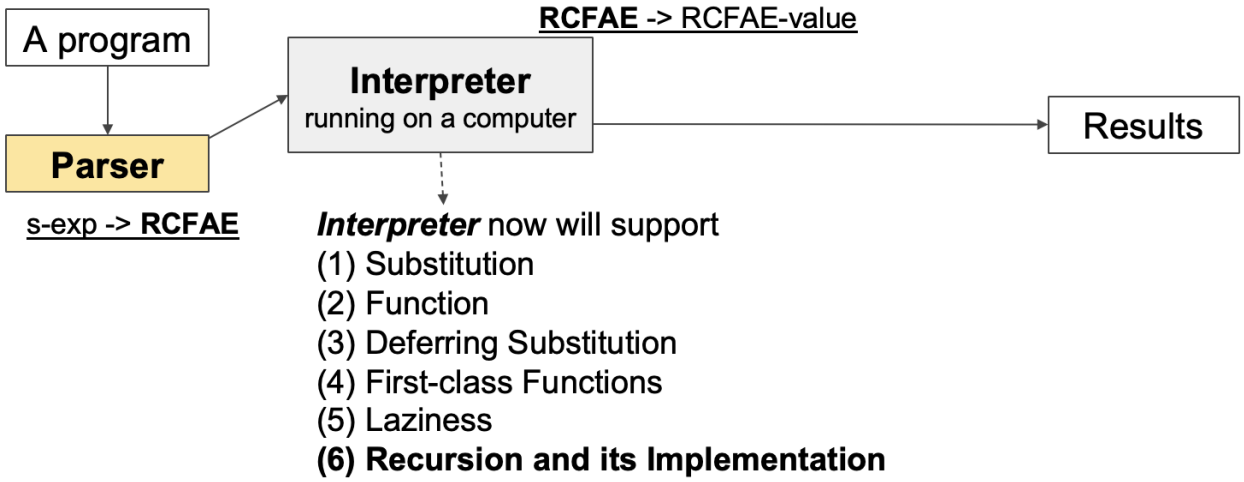
# L16. Recursion 2

| | |
|---|---|
| 유형 | 강의 |
| 복습 여부 | ☐ |
| Status | In progress |



**Using the existing syntax vs. Adding new syntax 'rec'**

```
{with {fac {with {facX {fun {facY}
                         {with {fac {fun {x}
                                     {{facY facY} x}}}
                               {fun {n}
                                    {if0 n
                                         1
                                         {* n {fac {- n 1}}}}}}}
                               {facX facX}}}
      {fac 10}}
```

```
{rec {fac {fun {n}
               {if0 n
                    1
                    {* n {fac {- n 1}}}}}}
     {fac 10}}
```

**Do not need to significantly update our interpreter.** vs. **Need to update our interpreter to support this syntax.**

**Code in concrete syntax is complicated.** vs. **Code is intuitive and simpler.**

## RCFAE: Concrete Syntax

```
<RCFAE> ::= <num>
          | {+ <RCFAE> <RCFAE>}
          | {- <RCFAE> <RCFAE>}
          | <id>
          | {fun {<id} <RCFAE>}
          | {<RCFAE> <RCFAE>}
          | {if0 <RCFAE> <RCFAE> <RCFAE>}
          | {rec {<id> <RCFAE>} <RCFAE>}
```

```
{rec {count {fun {n} {if0 n 0 {+ 1 {count {- n 1}}}}}} {count 8}}
```

▼ 과정

{count 8}

⇒ {+ 1 {count {- n 1}}}

⇒ {+ 1 {count 7}}

⇒ {+ 1 {+ 1 {count 6}}}

⇒ …

⇒ {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {count 0}}}}....}

⇒ {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 0}}}....}

⇒ …

⇒ 8

# RCFAE: Abstract Syntax

```
; RCFAE
(define-type RCFAE
  [num (n number?)]
  [add (lhs RCFAE?) (rhs RCFAE?)]
  [sub (lhs RCFAE?) (rhs RCFAE?)]
  [id (name symbol?)]
  [fun (param symbol?) (body RCFAE?)]
  [app (fun-expr RCFAE?) (arg-expr RCFAE?)]
  [if0 (test-expr RCFAE?) (then-expr RCFAE?) (else-expr RCFAE?)]
  [rec (name symbol?) (named-expr RCFAE?) (fst-call RCFAE?)])
```

```
; RCFAE-Value
(define-type RCFAE-Value
  [numV (n number?)]
  [closureV (param symbol?) (body RCFAE?) (ds DefrdSub?)]
  [exprV (expr RCFAE?) (ds DefrdSub?) (value (box/c (or/c RCFAE-Value?)))])

; DefrdSub
(define-type DefrdSub
  [mtSub]
  [aSub (name symbol?) (value RCFAE-Value?) (saved DefrdSub?)]
  [aRecSub (name symbol?) (value-box (box/c RCFAE-Value?)) (ds DefrdSub?)])
```

# Parser

```
; Parser
; sexp -> RCFAE
(define (parse sexp)
  (match sexp
    [(? number?) (num sexp)]
    [(list '+ l r) (add (parse l) (parse r))]
    [(list '- l r) (sub (parse l) (parse r))]
    [(list 'with (list i v) e) (app (fun i (parse e)) (parse v))]
    [(? symbol?) (id sexp)]
    [(list 'fun (list p) b) (fun p (parse b))]
    [(list f a) (app (parse f) (parse a))]
    [(list 'if0 ex1 ex2 ex3) (if0 (parse ex1) (parse ex2) (parse ex3))]
    [(list 'rec (list n ex) ft) (rec n (parse ex) (parse ft))]
    [else (error 'parse "bad syntax: ~a" sexp)]))
```

# Interpreter

```
; interp: RCFAE -> RCFAE
(define (interp rcfae ds)
  (type-case RCFAE rcfae
    [num (n) (numV n)]
    [add (l r) (num+ (interp l ds) (interp r ds))]
    [sub (l r) (num- (interp l ds) (interp r ds))]
    [id (s) (lookup s ds)]
    [fun (p b) (closureV p b ds)]
    [app (f a) (local [(define ftn (interp f ds))]
                 (interp (closureV-body ftn)
                         (aSub (closureV-param ftn)
                               (interp a ds)
                               (closureV-ds ftn))))]
    [if0 (test-expr then-expr else-expr) (if (numzero? (interp test-expr ds))
                                             (interp then-expr ds)
                                             (interp else-expr ds))]
    [rec (bound-id named-expr fst-call)
      (local [(define value-holder (box (numV 100)))
              (define new-ds (aRecSub bound-id value-holder ds))]
        ; new-ds
        ; new type of cache only for the recursion
        ; we don't know the value for bound-id yet. So we just use dummy value.
        ; dummy value needs to be replaced when we know the actual value later, so we use box for 'aRecSub' type.
        (begin (set-box! value-holder (interp named-expr new-ds))
               (interp fst-call new-ds)))
        ; if we just use 'ds' instead of 'new-ds' it does not contain any binding information for our recursive function.
      ]
    )
  )
```

## Supporting Functions for the interpreter

```
; num-op: (number number -> number) -> (RCFAE RCFAE -> RCFAE)
(define (num-op op)
  (lambda (x y)
    (numV (op (numV-n (strict x)) (numV-n (strict y))))
    )
  )

(define num+ (num-op +))
(define num- (num-op -))
(define num* (num-op *))

; strict: RCFAE-Value -> RCFAE-Value
(define (strict v)
  (type-case RCFAE-Value v
    [exprV (expr ds v-box)
           (if (not (unbox v-box))
               (local [(define v (strict (interp expr ds)))]
                 (begin (set-box! v-box v) v))
               (unbox))]
    [else v]))

; lookup: symbol DefrdSub -> RCFAE-Value
(define (lookup name ds)
  (type-case DefrdSub ds
    [mtSub() (error 'lookup "free variable")]
    [aSub (sub-name val rest-ds)
          (if (symbol=? sub-name name)
              val
```

```
               (lookup name rest-ds))]
    [aRecSub (sub-name val-box rest-ds)
             (if (symbol=? sub-name name)
                 (unbox val-box)
                 (lookup name rest-ds))]))

; numzero? RCFAE-Value -> boolean
(define (numzero? n)
  (zero? (numV-n n)))
```

```
(parse '{rec {count {fun {n} {if0 n 0 {+ 1 {count {- n 1}}}}}} {count 8}})
(run '{rec {count {fun {n} {if0 n 0 {+ 1 {count {- n 1}}}}}} {count 8}} (mtSub))
```

```
(rec 'count (fun 'n (if0 (id 'n) (num 0) (add (num 1) (app (id 'count) (sub (id 'n) (num 1)))))) (app (id 'count) (num 8)))
(numV 8)
```