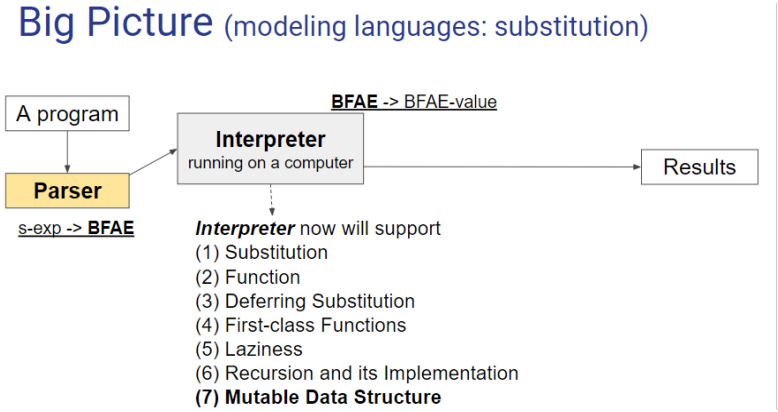
# Racket tutorials (L2,3)




good programming: the process of programming as **systematic**, The creation of software that relies on SYSTEMATIC thought, planning, and understanding.
step a program works: Design - write the program by PL - Interpret or compile - run
Elements for Systematic Program Design: Problem Analysis and Data Deflnitions / Contract (Signature), Purpose (Effect) statement / Header / Functional Examples / Function Definition / Testing
Basic PL elements: Numbers and Arithmetic / Variables and Functions / Conditional Expressions / Conditional Function / Symbols
Design recipe for functions: Contract - purpose - example - header - body - tests
Test-Driven Development : write test cases before writing programs.

- Keep design simple / make incremental progress(점점나아짐) / protect code

Interpreter takes a program and produces a result (bash, racket, search engine) execute code line by line.
Compiler takes a program and produces a (binary) program (gcc, javac, racket) converts program to executable (binary) program

- Interpreter is more platform independent but slower. Compiled code is faster but hard to debug because you need to compile the program first.

type-decomposition: deal abstract syntax semantically: implement semantic

Syntax: the structure of grammar of the language
Semantics: behaviors associated with each syntax. Most significant to learn PLT.

- Syntax is the grammatical structure of language. Semantic is the meaning of the sentence. Need syntax that computer understands and semantics so that computer actually do what we want
- Free identifier: Semantic error because the format is syntactically correct but the meaning of identifier is undefined. typically arises from issues related to a program's meaning or execution. (in some case, can occur during the syntax parsing stage)

Goal of sugaring-desugaring is to get a lighter/efficient interpreter . sugar is making additions to a language using existing features. Desugaring is writing the sugared syntax into core constructs.
List : (cons a (cons 2 empty)). (list 1 2 3), (append (list 1 2) (list 3 4)), (firts ()) (rest ()) (empty?

(list)), (cons? (list))
Symbol: two symbols that have the same contents are guaranteed to be the same object(string is not) // (eq? (sym1 ‘hello) (sym3 (string->symbol “hello”)) ;#t

# Modeling languages, Interpreting arithmetic (L4-5) : *AE*

Modeling syntax: Concrete Syntax(expression) -> abstract syntax(put this in parser)

- Abstract syntax is one data definition for the AE, essence in a tree form
- Use AE not num, to flexibility for nested expression / racket: right data definition

Parser: component in an interpreter or compiler. Identifies what kinds of program code it is examining. Convert concrete syntax into abstract.

- Using BNF(Backus-Naur Form) to specify the concrete syntax. Captures both the concrete syntax and a default abstract syntax

| | |
|---|---|
| <AE> ::= <num><br>\|{+ <expr> <expr>}<br>\|{- <expr> <expr>}<br><br>(define-type AE<br>[num (n number?)]<br>[add (lhs AE?) (rhs AE?)]<br>[sub (lhs AE?) (rhs AE?)] | - <expr>: non-terminal(대체됨), Meta-variable(define a set)<br>- ::= : Can be written as<br>- \| : one more choice<br>- Terminal: actual value , no change |

- Every PL has BNF; complex BNF -> detail logic expression, complex syntax
- (error ‘parse “bad syntax: ~a” sexp) | (test/exn (parse ‘{1 2 3}) “parse: bad syntax: (1 2 3)

# Substitution (L6-7) : *WAE*

Substitution: using identifiers(need to be replaced) to avoid redundancy. O(n^2) -> recursive
(with i v e) -> replace all bound instances of i and free instances of i in e with v.

- Free identifier: id not contained in the scope of any binding instance of its name
- binding instance of an identifier is the instance of the identifier that gives it its value. In WAE, the <id> position of a with is the only binding instance. identifier is bound if it is contained within the scope of a binding instance of its name.
- scope of a binding instance is the region of program text in which instances of the identifier refer to the value bound by the binding instance
- With expression is sugaring: it makes lighter/efficient interpreter

Patch note for WAE: now interp using helper function -> (subst wae id val)

| | |
|---|---|
| (define-type WAE<br>[num (n number?)]<br>[add (lhs WAE?) (rhs WAE?)]<br>[sub (lhs WAE?) (rhs WAE?)]<br>[with (name symbol?) (named-expr WAE?) (body WAE?)]<br>[id (name symbol?)]) | **;; parse : sexp -> WAE**<br>(define (parse sexp)<br>(match sexp<br>[(? number?) (num sexp)]<br>[(list '+ l r) (add (parse l) (parse r))]<br>[(list '- l r) (sub (parse l) (parse r))]<br>[(list 'with (list i v) e) (with i (parse v) (parse e))]<br>[(? symbol?) (id sexp)]<br>[else (error 'parse "bad syntax: ~a" sexp)])) |

```
;; subst : WAE symbol number→ WAE
(define (subst wae idtf val)
  (type-case WAE wae
    [num (n)   wae]
    [add (l r)   (add (subst l idtf val)
                  (subst r idtf val))]
    [sub (l r)   (sub (subst l idtf val)
                 (subst r idtf val))]
    [with (i v e)
        (if (symbol=? i idtf)
            e
           (subst e idtf val)))]
    [id (s) (if (symbol=? s idtf) (num val) wae)]))
```

```
;; interp : WAE → number
(define (interp wae)
  (type-case WAE wae
    [num (n) n]
    [add (l r) (+ (interp l) (interp r))]
    [sub (l r) (- (interp l) (interp r))]
    [with (i v e) (interp (subst e i (num (interp v))))]
    [id (v) (error 'interp "free identifier")]))
```

# Function (L8) : *F1WAE*

Function: reduce mistake and reduce the amount of code (reducing repetition)
Patch note for F1WAE: Using new non-terminal fundef type, new parse-fd, interp updated to consume the list of fundef, F1WAE and interp, subst updated to add (app) case.

```
(define-type F1WAE
  [num (n number?)]
  [add (lhs F1WAE?) (rhs F1WAE?)]
  [sub (lhs F1WAE?) (rhs F1WAE?)]
  [with (name symbol?) (named-expr F1WAE?) (body
F1WAE?)]
  [id (name symbol?)]
  [app (ftn symbol?) (arg F1WAE?)])
```

```
;; parse : sexp -> F1WAE
(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) (with i (parse v)(parse e))]
    [(? symbol?)          (id sexp)]
    [(list f a)           (app f (parse a))]
    [else                 (error 'parse "bad syntax: ~a" sexp)]))
```

```
(define-type FunDef
  [fundef   (fun-name symbol?) (arg-name symbol?)
            (body F1WAE?)])
```

```
;; parse-fd : sexp -> FunDef
(define (parse-fd sexp)
  (match sexp
    [(list 'deffun (list f x) b) (fundef f x (parse b))]))
```

**s**

```
;; subst : F1WAE symbol number → F1WAE
(define (subst f1wae idtf val)
  (type-case F1WAE f1wae
    [num (n)       f1wae]
    [add (l r)     (add (subst l idtf val) (subst r idtf val))]
    [sub (l r)     (sub (subst l idtf val) (subst r idtf val))]
    [with (i v e)   (with i (subst v idtf val)
                        (if (symbol? l idtf)
                            e
                           (subst e idtf val)))]
    [id (s)        (if (symbol=? s idtf) (num val) f1wae)]
    [app (f a)     (app f (subst a idtf val))]))
```

```
;; interp : F1WAE list-of-fundef → number
(define (interp f1wae fundefs)
  (type-case F1WAE f1wae
    [num (n) n]
    [add (l r) (+ (interp l fundefs) (interp r fundefs))]
    [with (i v e) (interp (subst e i (num (interp v))))]
    [id (v) (error 'interp "free identifier")]
    [app (f a)
        (local ([define the-fun-def (lookup-fundef f fun-defs)])
               (interp (subst (fundef-body the-fun-def)
                      (fundef-arg-name the-fun-def)
                      (interp a fundefs)))
```

| | |
|---|---|
| | fundefs))])) |

# Substitution (L9): *F1WAE with deferring*

Patch note for F1WAE-deferring: We added DefrdSub and interp consumes ds too. Now <id> is replaced from the ds list. <with> is using lookup function once and not using subst anymore.

- Lookup function time complexity: O(n), linear search

static scope: the scope of an identifier's binding is a syntactically delimited region.
dynamic scope: the scope of an identifier's binding is the entire remainder of the execution during which that binding is in effect.
environment is a repository of deferred substitutions.

| | |
|---|---|
| (define-type **F1WAE**<br>[num (n number?)]<br>[add (lhs F1WAE?) (rhs F1WAE?)]<br>[sub (lhs F1WAE?) (rhs F1WAE?)]<br>[with (name symbol?) (named-expr F1WAE?) (body F1WAE?)]<br>[id (name symbol?)]<br>[app (ftn symbol?) (arg F1WAE?)]) | parser… |
| (define-type **FunDef**<br>[fundef (fun-name symbol?)<br>(arg-name symbol?)<br>(body F1WAE?)]) | **;; lookup-fundef : symbol list-of-FunDef -→ FunDef**<br>(define (lookup-fundef name fundefs)<br>(cond<br>[(empty? fundefs) (error 'lookup-fundef "function not found")]<br>[else (if (symbol=? name (fundef-fun-name (first fundefs)))<br>(first fundefs)<br>(lookup-fundef name (rest fundefs)))])) |
| (define-type DefrdSub<br>[mtSub]<br>[aSub (name symbol?)<br>(value number?)<br>(saved DefrdSub?)]) | **;; lookup : symbol DefrdSub → F1WAE**<br>(define (lookup name ds)<br>(type-case DefrdSub ds<br>[mtSub () (error 'lookup "no binding for identifier")]<br>[aSub (bound-name bound-value rest-ds)<br>(if (symbol=? bound-name name)<br>bound-value<br>(lookup name rest-ds))])) |
| **;; interp : F1WAE list-of-fundef DefrdSub → number**<br>(define (interp f1wae fundefs ds)<br>(type-case F1WAE f1wae<br>[num (n) n]<br>[add (l r) (+ (interp l fundefs ds) (interp r fundefs ds))]<br>[sub (l r) (- (interp l fundefs ds) (interp r fundefs ds))]<br>[with (i v e) (interp e fundefs (aSub i (interp v findefs ds))]<br>[id (s) (lookup s ds)] | |

```
[app (f a) (local
            ([define the-fun-def (lookup-fundef f fundefs)])
            (interp (fundef-body the-fun-def)
                    fun-defs
                    (aSub (fundef-arg-name the-fun-def)
                          (interp a fundefs ds)
                          (mtSub))))]))
```

# First-class Functions (L10-L12): *FWAE, FAE*

First-class Function: now functions are values. (can be the value of arguments to function, return value of function, stored in data structure)
Lambda(anonymous) : good for code length(remove loop and reuse fundef), bad for speed, difficult to debug and understand the code. (Python: (lambda x, y: x + y)(3, 5))
Patch note for FWAE: Now fun syntax(anonymous) is available. In BNF, independent FunDef is merged with (fun) syntax. Parser updated to support fundef(fun) and function call(list f a) both. For interp, list-of-fundef is not used longer and return type becomes FWAE not number. (fun) returns itself and (app) uses subst() again. Using num+, num- (by using num-op) instead of + and - to consider the return type. Dynamic scope issue exist, but ignore it.(ds will deal with it later)

```
(define-type FWAE
  [num    (n numer?)]
  [add    (lhs FWAE?) (rhs FWAE?)]
  [sub    (lhs FWAE?) (rhs FWAE?)]
  [with   (name symbol?) (named-expr FWAE?)
          (body FWAE?)]
  [id     (name symbol?)]
  [fun    (param symbol?) (body FWAE?)]
  [app    (ftn FWAE?) (arg FWAE?)])
```

```
;; parse : sexp -> : FWAE
(define (parse sexp)
(match sexp
   [(? number?)          (num sexp)]
   [(list '+ l r)        (add (parse l)(parse r))]
   [(list '- l r)        (sub (parse l)(parse r))]
   [(lsit 'with (list i v) e) (with i (parse v)(parse e))]
   [(? symbol?)          (id sexp)]
   [(list 'fun (list p) b)  (fun p (parse b))]
   [(list f a)           (app (parse f)(parse a))]
   [else            (error 'parse "bad syntax: ~a" sexp)]))
```

```
;; num-op : (number number -> number) -> (FWAE
FWAE -> FWAE)
(define (num-op op)
    (lambda (x y)
        (num (op (num-n x)(num-n y)))))
```

```
(define num+ (num-op +))
(define num- (num-op -))
```

```
;; subst : FWAE symbol FWAE -> FWAE
(define (subst exp idtf val)
  (type-case FWAE exp
    [num (n) f1wae]
    [add (l r) (add (subst l sub-id val)
                    (subst r sub-id val))]
    [sub (l r) (sub (subst l sub-id val)
                    (subst r sub-id val))]
    [id (name) (cond [(equal? name idtf) val]
                     [else exp])]
    [app (f arg) (app (subst f idtf val) (subst arg idtf val))]
    [fun (id body) (if (equal? idtf id)
```

```
;; interp : FWAE -> FWAE
(define (interp fwae)
  (type-case FWAE fwae
    [num (n)      fwae]
    [add (l r)    (num+ (interp l)(interp r))]
    [sub (l r)    (num- (interp l)(interp r))]
    [with (i v e) (interp (subst e i (interp v)))]
    [id (s)       (error 'interp "free identifier")]
    [fun (p b)    fwae]
    [app (f a)    (local
                    [(define ftn (interp f))]
                    (interp (subst (fun-body ftn)
```

| | |
|---|---|
| exp<br>(fun id (subst body idtf val)))]) | (fun-param ftn)<br>(interp a))))])) |

Patch note for FAE: remove (with). Ds is available now. New define-type FAE-Value added for return value. It contains closureV that has captured-valid-ds-list as third param. Now (with) is removed and (fun) returns closure with current ds. (app) support new ds (current ds + param)

- (removing (with) is kind of sugaring for interp). keep (with) in BNF cus users still want it. Just remove it in abstract syntax. Parser will do desugaing for interp. [(list 'with (list i v) e) (app (fun i (parse e)) (parse v))] also, remove (with) in interp and update (app)
- Closure: closes a function's environment. Can contain the function's environment until its execution. We can use function as return value by using closure. It deferred substitution to handle variables when executed.

```
(define-type FAE
  [num (n number?)]
  [add (lhs FAE?) (rhs FAE?)]
  [sub (lhs FAE?) (rhs FAE?)]
  [id (name symbol?)]
  [fun (param symbol?) (body FAE?)]
  [app (fun-expr FAE?) (arg-expr FAE?)])
```

```
; parse: sexp -> FAE
; purpose: to convert sexp to FAE
(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))]
     [else                     (error 'parse "bad syntax: ~a"
sexp)]))
```

```
(define-type FAE-Value
  [numV     (n number?)]
  [closureV (param symbol?)
            (body FAE?)
            (ds DefrdSub?)])
```

```
(define-type DefrdSub
  [mtSub]
  [aSub   (name symbol?) (value FAE-Value?)
          (ds DefrdSub?)])
```

```
;; lookup : symbol DefrdSub → FAE-Value
(define (lookup name ds)
  (type-case DefrdSub ds
    [mtSub () (error 'lookup "no binding for identifier")]
    [aSub (bound-name bound-value rest-ds)
        (if (symbol=? bound-name name)
            bound-value
            (lookup name rest-ds))]))

[ [x 10], [y 12], ]

;; num+ : numV numV -→ numV
(define (num+ n1 n2)
  (numV (+ (numV-n n1) (numV-n n2))))

;; num- : numV numV -→ numV
(define (num- n1 n2)
  (numV (- (numV-n n1) (numV-n n2))))
```

```
;; interp : FAE DefrdSub → FAE-Value
(define (interp fae ds)
  (type-case FAE fae
    [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 fun-val (interp f ds))
                       (define arg-val (interp a ds))]
                 (interp (closureV-body fun-val)
                         (aSub (closureV-param fun-val)
                               arg-val
                               (closureV-ds fun-val))))]))
```

# Laziness (L13,14) *LFAE*

Lazy: avoid unnecessary work, evaluate only if its result is needed. Efficient!
Laziness: not evaluate the argument express until its value is needed. Close it over its environment to preserve static scope.
DefrdSub is substitution delayed, Laziness is evaluation delayed. Both make interpreters efficient.
Short-circuiting stops right after you know the result(cut off unnecessary computation), Laziness evaluates only when it is needed. Just delay the whole computation until its result is required.
Box: single value container. We use it cuz we needed data type which can store any type
Memoization: caches function's result and checks the cache when the function is invoked next time

- This LFAE is not memoization: It just reduces redundant evaluation of the function, but if scope is changed, the function will be evaluated again

```
(define-type LFAE
  [num (n number?)]
  [add (lhs LFAE?) (rhs LFAE?)]
  [id (name symbol?)]
  [fun (param symbol?) (body LFAE?)]
  [app (fun-expr LFAE?) (arg-expr LFAE?)])
```

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

```
(define-type DefrdSub
  [mtSub]
  [aSub   (name symbol?) (value LFAE-Value?)
          (ds DefrdSub?)])
```

```
;; num-op :
(define (num-op op x y)
    (numV (op (numV-n (strict x))(numV-n (strict y))))))
(define (num+ x y) (num-op + x y))
(define (num- x y)  (num-op - x y))
```

```
;; strict : LFAE-Value → LFAE-Value
(define (strict v)
  (type-case LFAE-Value v
    [exprV (expr ds v-box)
        (if (not (unbox v-box)) ;box containing #f
            (local
                [(define the-value (strict (interp expr ds)))]
              (begin (set-box! v-box the-value)
                     the-value))
              (unbox v-box))]
    [else v]))
```

*parser is the same with FAE

```
;; interp : LFAE DefrdSub → LFAE -Value
(define (interp lfae ds)
  (type-case LFAE lfae
    [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 fun-val (strict (interp f ds)))
                       (define arg-val (exprV a ds (box #f)))]
                 (interp (closureV-body fun-val)
                         (aSub (closureV-param fun-val)
                               arg-val
                               (closureV-ds fun-val))))]))
```

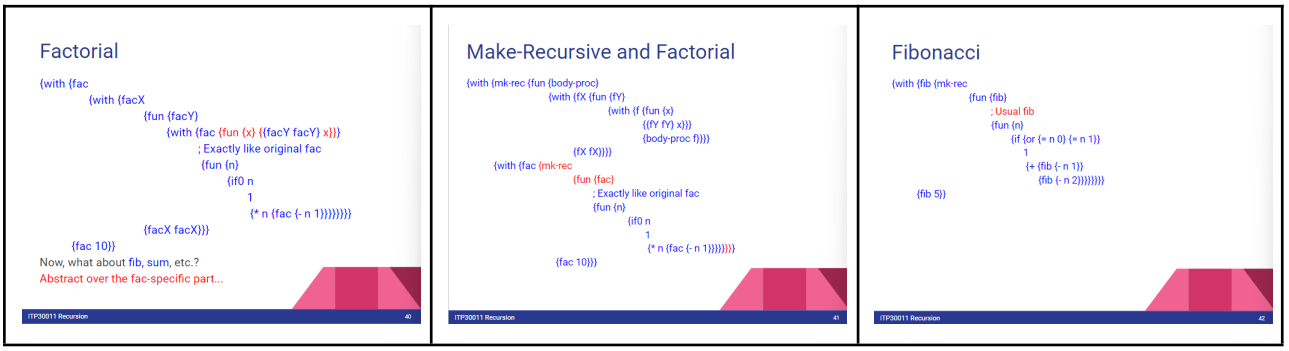
# Recursion (L15, 16) *LCFAE*

```
<RCFAE> ::= <num>
| {+ <RCFAE> <RCFAE>}
| {- <RCFAE> <RCFAE>}
| {* <RCFAE> <RCFAE>}
| <id>
```

```
| {fun {<id>} <RCFAE>}
| {<RCFAE> <RCFAE>}
| {if0 <RCFAE> <RCFAE> <RCFAE>}
| {rec {<id> <RCFAE>} <RCFAE>}
```

Patch note for LCFAE: some of new things added, * and if0, rec. rec syntax defining a recursive function and its call. Also, rec binds both in the body expression and in the binding expression.

- 'With' does not support a recursive definition. (free identifier, need to pass {fac 10} as arg)
- η reduction (eta reduction): If two functions lead to the same result, they are the same fun.
  - {fun {n} {with {f …} {{f f} n}}} -> {with {f …} {f f}}
  - η reduction is not possible as n is free in {fun {x} n}




## RCFAE: Concrete Syntax

**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.**

```
(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?)]
```

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

(define-type RCFAE-Value
```

| | |
|---|---|
| [rec (name symbol?) (named-expr RCFAE? (fst-call RCFAE?)]) | [numV (n number?)]<br>[closureV (param Symbol?) (body RCFAE?) (ds DefrdSub?)]) |
| ; interp : RCFAE DefrdSub -> RCFAE-Value<br>(define (interp rcfae ds)<br>(type-case RCFAE rcfae<br>[num (n) (numV n)]<br>[add (l r) (num+ (interp l ds) (interp r ds))]<br>[sub (l r) (num- (interp l ds) (interp r ds))]<br>[id (name) (lookup name ds)]<br>[fun (param body-expr) (closureV param body-expr ds)]<br>[app (f a) (local [(define ftn (interp f ds))]<br>(interp (closureV-body ftn)<br>(aSub (closureV-param ftn)<br>(interp a ds)<br>(closureV-ds ftn)))))]<br>[if0 (test-expr then-expr else-expr)<br>(if (numzero? (interp test-expr ds)(<br>(interp then-expr ds)<br>(interp else-expr ds))]<br>[rec (bound-id named-expr fst-call)<br>(local [(define value-holder (box (numV 198)))<br>(define new-ds (aRecSub bound-id<br>value-holder ds))]<br>(begin<br>(set-box! value-holder<br>(interp named-expr new-ds))<br>(interp fst-call new-ds)))])) | ; numzero? : RCFAE-Value -> boolean<br>(define (**numzero**? n)<br>(zero? (numV-n n)))<br><br>; lookup : symbol DefrdSub -> RCFAE-Value<br>(define (**lookup** name ds)<br>(type-case DefrdSub ds<br>[mtSub () (error 'lookup "free variable")]<br>[aSub (sub-name val rest-ds)<br>(if (symbol=? sub-name name)<br>Val<br>(lookup name rest-ds))]<br>[aRecSub (sub-name val-box rest-ds)<br>(if (symbol=? sub-name name)<br>(unbox val-box)<br>(lookup name rest-ds))])) |

A box is like a single-element vector, normally used as minimal mutable storage.
box: (define value-holder (box (numV 198)))
set-box! (set-box! value-holder (interp named-expr new-ds))
unbox: (unbox val-box)
box/c: (value-box (box/c RCFAE-Value?))

# Mutable Data Structure(L17, 18, 19, 20)

TODO

- Mutation involves changing or varying data.

Memoization: caches function's result and checks the cache when the function is invoked next time

This LFAE is not memoization: It just reduces redundant evaluation of the function, but if scope is changed, the function will be evaluated again

eta deduction can simplify functions by removing unnecessary lambda abstractions. It helps in creating more concise and potentially more optimized code in functional programming languages.

One for keeping a memory address value of a box for static

scope

Another for tracking dynamic changes of boxes.

# Variables(L21, 22)

Store-Passing Interpreters

Our BFAE interpreter explains state by representing the store as a value.

-Every step in computation produces a new store.

-The interpreter itself is purely functional.

Call-by-value call-by-reference

Call-by-value

When a function is called, malloc generates a new address for the function parameter.

Call-by-reference

When a function is called, the value of the existing address of 'a' is stored with 5, which is mutated in the function.

Continuation

Rest of computation to be evaluated from one point.

Rest of work that has to happen to finish the evaluation of a program

Abstract representation of the control state of a program.

ContinuationPassingStyle(CPS)

Easy to transform your representation of stacks from the actual stack to heap

So, if you have a deep recursion, you wouldn't be run out of stack memory

Can simulate control flow like operators, exceptions, loops,...

# Continuation (L23, 24, 25, 26)

Commonalities

Control Flow Management: Both concepts deal with the control state of a program. They are involved in deciding what the next step of the computation should be after a certain point in the program is reached.

Handling Deep Recursion: Both can be used to manage deep recursion without running out of stack memory. This is because they can shift the representation of stack frames onto the heap, which is typically much larger.

Abstraction of Execution: They abstract the flow of execution in the program, allowing for the capture of the current state of computation and its continuation at a later point.

Differences

Concept vs. Technique:

Continuation: It is an abstract concept that represents the remainder of the program after a certain point has been reached. It is a way of thinking about the state of the program execution.

CPS: This is a programming technique where continuations are explicitly passed as arguments to functions. It is a concrete implementation strategy that uses the concept of continuations.

Representation of Control State:

Continuation: Acts as an abstract representation of the control state that could potentially be resumed.

CPS: Transforms the control state into a series of function calls, effectively managing the continuation of the program's execution on the heap instead of the stack.

Memory Management:

Continuation: While it doesn't specify how memory is managed, it is closely related to the idea of stack versus heap management.

CPS: Specifically allows the transformation of stack memory representation to heap memory, which is beneficial for programming languages that don't automatically handle stack overflow issues.

Simulation of Control Flow Constructs:

Continuation: The concept itself doesn't inherently simulate control flow constructs such as operators, exceptions, and loops.

CPS: The technique is particularly adept at simulating various control flow constructs and can be used to implement features like exceptions and loops in a structured way.

(1) Discuss if the C programming language supports call by reference. Write your arguments for your answer.

The C programming language does not inherently support call by reference. In C, when you pass an argument to a function, what you're actually passing is a copy of the argument's value, known as call by value. However, you can achieve a behavior similar to call by reference by using pointers. When you pass the address of a variable (a pointer) to a function, the function can

dereference the pointer and modify the variable it points to. This mimics call by reference, although it's technically still call by value, where the value being passed is the address.

(2) Think about call by reference in Java. Does Java support call by reference? Write your arguments for your answer.

Java does not support call by reference; it is strictly a call by value language. This can be a bit confusing because when you pass an object to a method in Java, you're passing the reference to the object by value. This means that while you can modify the object's state within the method, you cannot change the reference itself to point to a different object in a way that is reflected outside of the method.

RFAE and RCFAE
RFAE is implemented by using current FAE syntax while RCFAE is implemented by changing the interpreter.  Is there a reason why using the current FAE syntax called as syntatic sugaring? Isnt updating the interpreter closer to the actual definition of syntatic sugaring?
When adding new language feature, there could be several ways. RFAE and RCFAE show such examples. Using a syntatic sugar is a lighter way to update the langauge. Deciding a better way for the updating the language need to consider many factors. Usually updating an interpreter is more tricky. Updaging an interpreter is dealing with new semantic of the langauge while updating the langauge by using the syntactic sugar is the matter of adding new syntax.

대예은's test case for HW3-task2(updated by 갓은혁)

| Concrete Syntax | Parser | Interpreter |
|---|---|---|
| '{+ 1 {+ 1 {+ 1 1}}}' | (add (num 1) (add (num 1) (add (num 1) (num 1)))) | (numV 4) |
| '{fun {x} {+ x x}}' | (fun x (add (id x) (id x))) | (closureV x (add (id x) (id x)) (mtSub)) |
| '{{fun {x} {+ 1 x}} 10}' | (app (fun x (add (num 1) (id x))) (num 10)) | (numV 11) |
| '{with {y 10} {fun {x} {+ x x}}}' | (app (fun y (fun x (add (id x) (id x)))) (num 10)) | (closureV x (add (id x) (id x)) (aSub 'y (numV 10) (mtSub))) |
| '{with {x 3} {+ x x}}' | (app (fun x (add (id x) (id x))) (num 3)) | (numV 6) |
| '{with {x {+ 1 1}} {with {y 3} {+ x y}}}' | (app (fun x (app (fun y (add (id x) (id y))) (num 3))) (add (num 1) (num 1))) | (numV 5) |
| '{f 3}' | (app (id f) (num 3)) | error in lookup: no binding for identifier |
| '{fun {x} 3}' | (fun x (num 3)) | (closureV x (num 3) (mtSub)) |
| '{fun x 3}' | parse: bad syntax: {fun x 3} | parse: bad syntax: {fun x 3} |
| '{1}' | parse: bad syntax: (1) | parse: bad syntax: (1) |
| '1' | (num 1) | (numV 1) |
| '{1 2}' | (app (num 1) (num 2)) | *(contract violation error)* |
| '{with {x {with {y 5} x}} y}' | (app (fun x (id y)) (app (fun y (id x)) (num 5))) | error in lookup: no binding for identifier |
| '{with {x 3 4} {+ x x}}' | parse: bad syntax: (fun (x y) 3) | parse: bad syntax: (fun (x y) 3) |
| "{with {fib {fun {n} {if {or {= n 0} {= n 1}} 1 {+ {fib {- n 1}} {fib {- n 2}}}}}} {fib 10}}" | (app (fun mk-rec (app (fun fib (app (id fib) (num 10))) (app (id mk-rec) (fun fib (fun n (if (or (= (id n) (num 0)) (= (id n) (num 1))) (num 1) (add (app (id fib) (sub (id n) (num 1))) (app (id fib) (sub (id n) (num 2)))))))))) (fun body-proc (app (fun fX (app (id fX) (id fX))) (fun fY (app (fun f (app (id body-proc) (id f))) (fun x (app (app (id fY) (id fY)) (id x)))))))) | |
| "{with {fac {fun {n} {if {= n 0} 1 {* n {fac {- n 1}}}}}} {fac 10}}" | (app (fun mk-rec (app (fun fac (app (id fac) (num 10))) (app (id mk-rec) (fun fac (fun n (if (= (id n) | |

| | | |
|---|---|---|
| | (num 0)) (num 1) (mul (id n) (app (id fac) (sub (id n) (num 1)))))))))) (fun body-proc (app (fun fX (app (id fX) (id fX))) (fun fY (app (fun f (app (id body-proc) (id f))) (fun x (app (app (id fY) (id fY)) (id x)))))))) | |