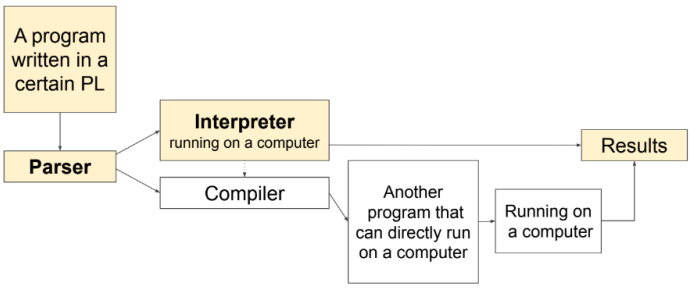
Racket tutorials (L2,3)

Big Picture



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> |{+ <expr> <expr>} |{- <expr> <expr>} (define-type AE [num (n number?)] [add (lhs AE?) (rhs AE?)] [sub (lhs AE?) (rhs AE?)]	- <expr>: non-terminal(대체됨), Meta-variable(define a set) - ::= : Can be written as - | : one more choice - 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
  [num (n number?)]
  [add (lhs WAE?) (rhs WAE?)]
  [sub (lhs WAE?) (rhs WAE?)]
  [with (name symbol?) (named-expr WAE?) (body
WAE?)]
  [id (name symbol?)])
```

```
;; parse : sexp -> WAE
(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)]
    [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))]
```

```
;; 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")]))
```

```
  [with (i v e)
      (if (symbol=? i idtf)
          e
          (subst e idtf val)))]
  [id (s) (if (symbol=? s idtf) (num val) wae)]))
```

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))]))
```

```
;; lookup-fundef : symbol list-of-FunDef -→ FunDef
(define (lookup-fundef name fundefs)
  (cond
    [(empty? fundefs) (error ‘lookup-fundef ”function not found”)]
    [else                 (if (symbol=? name (fundef-fun-name (first fundefs)))
                              (first fundefs)
                              (lookup-fundef name (rest fundefs)))]))
```

```
;; 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
  [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?)])
```

parser…

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

```
;; lookup-fundef : symbol list-of-FunDef -→ FunDef
(define (lookup-fundef name fundefs)
  (cond
    [(empty? fundefs) (error 'lookup-fundef "function not
found")]
    [else  (if (symbol=? name (fundef-fun-name (first fundefs)))
               (first fundefs)
               (lookup-fundef name (rest fundefs)))]))
```

```
(define-type DefrdSub
  [mtSub]
  [aSub    (name symbol?)
           (value number?)
           (saved DefrdSub?)])
```

```
;; lookup : symbol DefrdSub → F1WAE
(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))]))
```

```
;; interp : F1WAE list-of-fundef DefrdSub → number
(define (interp f1wae fundefs ds)
  (type-case F1WAE f1wae
    [num (n) n]
    [add (l r) (+ (interp l fundefs ds) (interp r fundefs ds))]
    [sub (l r) (- (interp l fundefs ds) (interp r fundefs ds))]
    [with (i v e) (interp e fundefs (aSub i (interp v findefs ds))]
    [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)
                       exp
                       (fun id (subst body idtf val)))]))
```

```
;; 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)
                                   (fun-param ftn)
                                   (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 variable when execution.

```
(define-type FAE
  [num (n number?)]
  [add (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))]))

;; 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 (LI 3,14) *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 isi 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))))]))
```