```
--<<<FLANG-Substitution>>>-----------------------------------------------
  ;; The Flang interpreter (substitution model)

  #lang pl

  #|  The grammar:
    <FLANG> ::= <num>
              | { + <FLANG> <FLANG> }
              | { - <FLANG> <FLANG> }
              | { * <FLANG> <FLANG> }
              | { / <FLANG> <FLANG> }
              | { with { <id> <FLANG> } <FLANG> }
              | <id>
              | { fun { <id> } <FLANG> }
              | { call <FLANG> <FLANG> }

Evaluation rules:
  subst:
    N[v/x]                 = N
    {+ E1 E2}[v/x]         = {+ E1[v/x] E2[v/x]}
    {- E1 E2}[v/x]         = {- E1[v/x] E2[v/x]}
    {* E1 E2}[v/x]         = {* E1[v/x] E2[v/x]}
    {/ E1 E2}[v/x]         = {/ E1[v/x] E2[v/x]}
    y[v/x]                 = y
    x[v/x]                 = x
    {with {y E1} E2}[v/x] = {with {y E1[v/x]} E2[v/x]} ; if y!=x
    {with {x E1} E2}[v/x] = {with {y E1[v/x]} E2}
    {call E1 E2}[v/x]      = {call E1[v/x] E2[v/x]}
    {fun {y} E}[v/x]       = {fun {y} E[v/x]}           ; if y!=x
    {fun {x} E}[v/x]       = {fun {x} E}

  eval:
    eval(N)                 = N
    eval({+ E1 E2})         = eval(E1) + eval(E2)
    eval({- E1 E2})         = eval(E1) - eval(E2)
    eval({* E1 E2})         = eval(E1) * eval(E2)
    eval({/ E1 E2})         = eval(E1) / eval(E2)
    eval(id)                 = error!
    eval({with {x E1} E2}) = eval(E2[eval(E1)/x])
    eval(FUN)       = FUN ; assuming Fun is a function expression
    eval({call E1 E2}) = eval(Ef[eval(E2)/x]) if eval(E1)={fun {x} Ef}
                       = error!           otherwise
|#

  (define-type FLANG
    [Num  Number]
    [Add  FLANG FLANG]
    [Sub  FLANG FLANG]
    [Mul  FLANG FLANG]
    [Div  FLANG FLANG]
    [Id   Symbol]
    [With Symbol FLANG FLANG]
    [Fun  Symbol FLANG]
    [Call FLANG FLANG])
```

```
(: parse-sexpr : Sexpr -> FLANG)
  ;; to convert s-expressions into FLANGs
  (define (parse-sexpr sexpr)
    (match sexpr
      [(number: n)     (Num n)]
      [(symbol: name) (Id name)]
      [(cons 'with more)
       (match sexpr
         [(list 'with (list (symbol: name) named) body)
          (With name (parse-sexpr named) (parse-sexpr body))]
         [else (error 'parse-sexpr "bad `with' syntax in ~s" sexpr)])]
      [(cons 'fun more)
       (match sexpr
         [(list 'fun (list (symbol: name)) body)
          (Fun name (parse-sexpr body))]
         [else (error 'parse-sexpr "bad `fun' syntax in ~s" sexpr)])]
      [(list '+ lhs rhs) (Add (parse-sexpr lhs) (parse-sexpr rhs))]
      [(list '- lhs rhs) (Sub (parse-sexpr lhs) (parse-sexpr rhs))]
      [(list '* lhs rhs) (Mul (parse-sexpr lhs) (parse-sexpr rhs))]
      [(list '/ lhs rhs) (Div (parse-sexpr lhs) (parse-sexpr rhs))]
      [(list 'call fun arg) (Call (parse-sexpr fun) (parse-sexpr arg))]
      [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))

  (: parse : String -> FLANG)
  ;; parses a string containing a FLANG expression to a FLANG AST
  (define (parse str)
    (parse-sexpr (string->sexpr str)))
```

הפרוצדורות eval **ו-** subst **מתוך האינטרפרטר של מודל ההחלפות (מה שבאמת שונה):**

```
(: subst : FLANG Symbol FLANG -> FLANG)
  ;; substitutes the second argument with the third argument in the
  ;; first argument, as per the rules of substitution; the resulting
  ;; expression contains no free instances of the second argument
  (define (subst expr from to)
    (cases expr
      [(Num n) expr]
      [(Add l r) (Add (subst l from to) (subst r from to))]
      [(Sub l r) (Sub (subst l from to) (subst r from to))]
      [(Mul l r) (Mul (subst l from to) (subst r from to))]
      [(Div l r) (Div (subst l from to) (subst r from to))]
      [(Id name) (if (eq? name from) to expr)]
      [(With bound-id named-expr bound-body)
       (With bound-id
             (subst named-expr from to)
             (if (eq? bound-id from)
               bound-body
               (subst bound-body from to)))]
      [(Call l r) (Call (subst l from to) (subst r from to))]
      [(Fun bound-id bound-body)
       (if (eq? bound-id from)
         expr
         (Fun bound-id (subst bound-body from to)))]))
```

```
(: arith-op : (Number Number -> Number) FLANG FLANG -> FLANG)
  ;; gets a Racket numeric binary operator, and uses it within a FLANG
  ;; `Num' wrapper
  (define (arith-op op expr1 expr2)
    (: Num->number : FLANG -> Number)
    (define (Num->number e)
      (cases e
        [(Num n) n]
        [else (error 'arith-op "expects a number, got: ~s" e)]))
    (Num (op (Num->number expr1) (Num->number expr2))))

(: eval : FLANG -> FLANG)
  ;; evaluates FLANG expressions by reducing them to *expressions*
  (define (eval expr)
    (cases expr
      [(Num n) expr]
      [(Add l r) (arith-op + (eval l) (eval r))]
      [(Sub l r) (arith-op - (eval l) (eval r))]
      [(Mul l r) (arith-op * (eval l) (eval r))]
      [(Div l r) (arith-op / (eval l) (eval r))]
      [(With bound-id named-expr bound-body)
       (eval (subst bound-body
                    bound-id
                    (eval named-expr)))]
      [(Id name) (error 'eval "free identifier: ~s" name)]
      [(Fun bound-id bound-body) expr]
      [(Call fun-expr arg-expr)
       (let ([fval (eval fun-expr)])
         (cases fval
           [(Fun bound-id bound-body)
            (eval (subst bound-body bound-id (eval arg-expr)))]
           [else (error 'eval "`call' expects a function, got: ~s"
                                fval)]))])))
```

במודל זה קוראים להערכה ללא מחסנית

```
(: run : String -> Number)
  ;; evaluate a FLANG program contained in a string
  (define (run str)
    (let ([result (eval (parse str))])
      (cases result
        [(Num n) n]
        [else (error 'run
                     "evaluation returned a non-number: ~s" result)])))
```

```
--<<<FLANG-Substitution-cache>>>----------------------------------------
  ;; The Flang interpreter, using Substitution-cache

  #lang pl

  #|  The grammar:
    <FLANG> ::= <num>
              | { + <FLANG> <FLANG> }
              | { - <FLANG> <FLANG> }
              | { * <FLANG> <FLANG> }
              | { / <FLANG> <FLANG> }
              | { with { <id> <FLANG> } <FLANG> }
              | <id>
              | { fun { <id> } <FLANG> }
              | { call <FLANG> <FLANG> }

  Evaluation rules:
    eval(N,sc)                 = N
    eval({+ E1 E2},sc)         = eval(E1,sc) + eval(E2,sc)
    eval({- E1 E2},sc)         = eval(E1,sc) - eval(E2,sc)
    eval({* E1 E2},sc)         = eval(E1,sc) * eval(E2,sc)
    eval({/ E1 E2},sc)         = eval(E1,sc) / eval(E2,sc)
    eval(x,sc)                 = lookup(x,sc)
    eval({with {x E1} E2},sc)  = eval(E2,extend(x,eval(E1,sc),sc))
    eval({fun {x} E},sc)       = {fun {x} E}
    eval({call E1 E2},sc)
             = eval(Ef,extend(x,eval(E2,sc),sc))
                                 if eval(E1,sc) = {fun {x} Ef}
             = error!            otherwise
  |#

  (define-type FLANG
    [Num  Number]
    [Add  FLANG FLANG]
    [Sub  FLANG FLANG]
    [Mul  FLANG FLANG]
    [Div  FLANG FLANG]
    [Id   Symbol]
    [With Symbol FLANG FLANG]
    [Fun  Symbol FLANG]
    [Call FLANG FLANG])
```

החלק של ה-**parser** זהה למודל הסביבות

```
(: parse-sexpr : Sexpr -> FLANG)
;; to convert s-expressions into FLANGs
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n)     (Num n)]
    [(symbol: name) (Id name)]
    [(cons 'with more)
     (match sexpr
       [(list 'with (list (symbol: name) named) body)
        (With name (parse-sexpr named) (parse-sexpr body))]
       [else (error 'parse-sexpr "bad `with' syntax in ~s" sexpr)])]
    [(cons 'fun more)
     (match sexpr
       [(list 'fun (list (symbol: name)) body)
        (Fun name (parse-sexpr body))]
       [else (error 'parse-sexpr "bad `fun' syntax in ~s" sexpr)])]
    [(list '+ lhs rhs) (Add (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '- lhs rhs) (Sub (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '* lhs rhs) (Mul (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '/ lhs rhs) (Div (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list 'call fun arg) (Call (parse-sexpr fun) (parse-sexpr arg))]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))

(: parse : String -> FLANG)
;; parses a string containing a FLANG expression to a FLANG AST
(define (parse str)
  (parse-sexpr (string->sexpr str)))
```

עד כאן החלק של ה-**parser** (כאמור, זהה למודל הסביבות)

```
;; a type for substitution caches:
(define-type SubstCache = (Listof (List Symbol FLANG)))

(: empty-subst : SubstCache)
(define empty-subst null)

(: extend : Symbol FLANG SubstCache -> SubstCache)
(define (extend name val sc)
  (cons (list name val) sc))

(: lookup : Symbol SubstCache -> FLANG)
(define (lookup name sc)
  (let ([cell (assq name sc)])
    (if cell
      (second cell)
      (error 'lookup "no binding for ~s" name))))
```

```
(: arith-op : (Number Number -> Number) FLANG FLANG -> FLANG)
;; gets a Racket numeric binary operator, and uses it within a FLANG
;; `Num' wrapper
(define (arith-op op expr1 expr2)
  (: Num->number : FLANG -> Number)
  (define (Num->number e)
    (cases e
      [(Num n) n]
      [else (error 'arith-op "expects a number, got: ~s" e)]))
  (Num (op (Num->number expr1) (Num->number expr2))))

(: eval : FLANG SubstCache -> FLANG)
;; evaluates FLANG expressions by reducing them to expressions
(define (eval expr sc)
  (cases expr
    [(Num n) expr]
    [(Add l r) (arith-op + (eval l sc) (eval r sc))]
    [(Sub l r) (arith-op - (eval l sc) (eval r sc))]
    [(Mul l r) (arith-op * (eval l sc) (eval r sc))]
    [(Div l r) (arith-op / (eval l sc) (eval r sc))]
    [(With bound-id named-expr bound-body)
     (eval bound-body
           (extend bound-id (eval named-expr sc) sc))]
    [(Id name) (lookup name sc)]
    [(Fun bound-id bound-body) expr]
    [(Call fun-expr arg-expr)
     (let ([fval (eval fun-expr sc)])
       (cases fval
         [(Fun bound-id bound-body)
          (eval bound-body
                (extend bound-id (eval arg-expr sc) sc))]
         [else (error 'eval "`call' expects a function, got: ~s"
                      fval)]))]))

(: run : String -> Number)
;; evaluate a FLANG program contained in a string
(define (run str)
  (let ([result (eval (parse str) empty-subst)])
    (cases result
      [(Num n) n]
      [else (error 'run
                   "evaluation returned a non-number: ~s" result)])))
```

```
--<<<FLANG-ENV>>>-----------------------------------------------------
  ;; The Flang interpreter, using environments

  #lang pl

  #|  The grammar:
    <FLANG> ::= <num>
              | { + <FLANG> <FLANG> }
              | { - <FLANG> <FLANG> }
              | { * <FLANG> <FLANG> }
              | { / <FLANG> <FLANG> }
              | { with { <id> <FLANG> } <FLANG> }
              | <id>
              | { fun { <id> } <FLANG> }
              | { call <FLANG> <FLANG> }

  Evaluation rules:
    eval(N,env)                 = N
    eval({+ E1 E2},env)         = eval(E1,env) + eval(E2,env)
    eval({- E1 E2},env)         = eval(E1,env) - eval(E2,env)
    eval({* E1 E2},env)         = eval(E1,env) * eval(E2,env)
    eval({/ E1 E2},env)         = eval(E1,env) / eval(E2,env)
    eval(x,env)                 = lookup(x,env)
    eval({with {x E1} E2},env) = eval(E2,extend(x,eval(E1,env),env))
    eval({fun {x} E},env)       = <{fun {x} E}, env>
    eval({call E1 E2},env1)
             = eval(Ef,extend(x,eval(E2,env1),env2))
                                if eval(E1,env1) = <{fun {x} Ef}, env2>
             = error!           otherwise
  |#

  (define-type FLANG
    [Num  Number]
    [Add  FLANG FLANG]
    [Sub  FLANG FLANG]
    [Mul  FLANG FLANG]
    [Div  FLANG FLANG]
    [Id   Symbol]
    [With Symbol FLANG FLANG]
    [Fun  Symbol FLANG]
    [Call FLANG FLANG])
```

```
(: parse-sexpr : Sexpr -> FLANG)
;; to convert s-expressions into FLANGs
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n)     (Num n)]
    [(symbol: name) (Id name)]
    [(cons 'with more)
     (match sexpr
       [(list 'with (list (symbol: name) named) body)
        (With name (parse-sexpr named) (parse-sexpr body))]
       [else (error 'parse-sexpr "bad `with' syntax in ~s" sexpr)])]
    [(cons 'fun more)
     (match sexpr
       [(list 'fun (list (symbol: name)) body)
        (Fun name (parse-sexpr body))]
       [else (error 'parse-sexpr "bad `fun' syntax in ~s" sexpr)])]
    [(list '+ lhs rhs) (Add (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '- lhs rhs) (Sub (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '* lhs rhs) (Mul (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '/ lhs rhs) (Div (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list 'call fun arg) (Call (parse-sexpr fun) (parse-sexpr arg))]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))

(: parse : String -> FLANG)
;; parses a string containing a FLANG expression to a FLANG AST
(define (parse str)
  (parse-sexpr (string->sexpr str)))

;; Types for environments, values, and a lookup function

(define-type ENV
  [EmptyEnv]
  [Extend Symbol VAL ENV])

(define-type VAL
  [NumV Number]
  [FunV Symbol FLANG ENV])

(: lookup : Symbol ENV -> VAL)
(define (lookup name env)
  (cases env
    [(EmptyEnv) (error 'lookup "no binding for ~s" name)]
    [(Extend id val rest-env)
     (if (eq? id name) val (lookup name rest-env))]))

(: arith-op : (Number Number -> Number) VAL VAL -> VAL)
;; gets a Racket numeric binary operator, and uses it within a NumV
;; wrapper
(define (arith-op op val1 val2)
  (: NumV->number : VAL -> Number)
  (define (NumV->number v)
    (cases v
      [(NumV n) n]
      [else (error 'arith-op "expects a number, got: ~s" v)]))
  (NumV (op (NumV->number val1) (NumV->number val2))))
```

```
(: eval : FLANG ENV -> VAL)
;; evaluates FLANG expressions by reducing them to values
(define (eval expr env)
  (cases expr
    [(Num n) (NumV n)]
    [(Add l r) (arith-op + (eval l env) (eval r env))]
    [(Sub l r) (arith-op - (eval l env) (eval r env))]
    [(Mul l r) (arith-op * (eval l env) (eval r env))]
    [(Div l r) (arith-op / (eval l env) (eval r env))]
    [(With bound-id named-expr bound-body)
     (eval bound-body
           (Extend bound-id (eval named-expr env) env))]
    [(Id name) (lookup name env)]
    [(Fun bound-id bound-body)
     (FunV bound-id bound-body env)]
    [(Call fun-expr arg-expr)
     (let ([fval (eval fun-expr env)])
       (cases fval
         [(FunV bound-id bound-body f-env)
          (eval bound-body
                (Extend bound-id (eval arg-expr env) f-env))]
         [else (error 'eval "`call' expects a function, got: ~s"
                      fval)]))]))

(: run : String -> Number)
;; evaluate a FLANG program contained in a string
(define (run str)
  (let ([result (eval (parse str) (EmptyEnv))])
    (cases result
      [(NumV n) n]
      [else (error 'run
                   "evaluation returned a non-number: ~s" result)])))
```