Part 1

Arithmetic+Function Language

```
; An expr-S-exp is either
;  - number
;  - symbol
;  - (list '+ expr-S-expr expr-S-expr)
;  - (list '* expr-S-expr expr-S-expr)
;  - (list symbol expr-S-expr)
```

```
<Expr> ::= <Num>
         | <Sym>
         | {+ <Expr> <Expr>}
         | {* <Expr> <Expr>}
         | {<Sym> <Expr>}
```

Arithmetic+Function Language

```
<Expr> ::= <Num>
         | <Sym>
         | {+ <Expr> <Expr>}
         | {* <Expr> <Expr>}
         | {<Sym> <Expr>}
```

```
(define-type ExprC
  [numC (n : number)]
  [idC (s : symbol)]
  [plusC (l : ExprC) (r : ExprC)]
  [multC (l : ExprC) (r : ExprC)]
  [appC (s : symbol) (arg : ExprC)])
```

Arithmetic+Function Language

```
<Expr> ::= <Num>
        |  <Sym>
        |  {+ <Expr> <Expr>}
        |  {* <Expr> <Expr>}
        |  {<Sym> <Expr>}
```

Arithmetic+Function+Local Language

```
<Expr> ::= <Num>
         | <Sym>
         | {+ <Expr> <Expr>}
         | {* <Expr> <Expr>}
         | {<Sym> <Expr>}
         | {let {[<Sym> <Expr>]}      NEW
             <Expr>}
```

```
{let {[x {+ 1 2}]}
  {+ x x}}                 ⇒   6
```

Arithmetic+Function+Local Language

```
<Expr>  ::=  <Num>
          |  <Sym>
          |  {+ <Expr> <Expr>}
          |  {* <Expr> <Expr>}
          |  {<Sym> <Expr>}
          |  {let {[<Sym> <Expr>]}      NEW
               <Expr>}
```

```
{+ {let {[x {+ 1 2}]}
        {+ x x}}
   1}                          ⇒   7
```

Arithmetic+Function+Local Language

```
<Expr> ::= <Num>
         | <Sym>
         | {+ <Expr> <Expr>}
         | {* <Expr> <Expr>}
         | {<Sym> <Expr>}
         | {let {[<Sym> <Expr>]}        NEW
             <Expr>}
```

```
{+ {let {[x {+ 1 2}]}
      {+ x x}}
   {let {[x {- 4 3}]}
      {+ x x}}}                   ⇒   8
```

Arithmetic+Function+Local Language

```
<Expr> ::= <Num>
        |  <Sym>
        |  {+ <Expr> <Expr>}
        |  {* <Expr> <Expr>}
        |  {<Sym> <Expr>}
        |  {let {[<Sym> <Expr>]}     NEW
             <Expr>}
```

```
{+ {let {[x {+ 1 2}]}
     {+ x x}}
   {let {[y {- 4 3}]}
     {+ y y}}}              ⇒   8
```

Arithmetic+Function+Local Language

```
<Expr>  ::=  <Num>
          |  <Sym>
          |  {+ <Expr> <Expr>}
          |  {* <Expr> <Expr>}
          |  {<Sym> <Expr>}
          |  {let {[<Sym> <Expr>]}      NEW
               <Expr>}
```

```
{let {[x {+ 1 2}]}
  {let {[x {- 4 3}]}
    {+ x x}}}              ⇒   2
```

Arithmetic+Function+Local Language

```
<Expr> ::= <Num>
         | <Sym>
         | {+ <Expr> <Expr>}
         | {* <Expr> <Expr>}
         | {<Sym> <Expr>}
         | {let {[<Sym> <Expr>]}
             <Expr>}
```

NEW

```
{let {[x {+ 1 2}]}
  {let {[y {- 4 3}]}
    {+ x x}}}                ⇒    6
```

Arithmetic+Function+Local Language

```
<Expr>  ::=  <Num>
          |  <Sym>
          |  {+ <Expr> <Expr>}
          |  {* <Expr> <Expr>}
          |  {<Sym> <Expr>}
          |  {let {[<Sym> <Expr>]}     NEW
               <Expr>}
```

```
{let {[x {+ 1 2}]}
  {let {[x {- 4 x}]}
    {+ x x}}}             ⇒   2
```

Arithmetic+Function+Local Language

```
<Expr>  ::=  <Num>
          |  <Sym>
          |  {+ <Expr> <Expr>}
          |  {* <Expr> <Expr>}
          |  {<Sym> <Expr>}
          |  {let {[<Sym> <Expr>]}    NEW
               <Expr>}
```

Arithmetic+Function+Local Language

```
<Expr>  ::=  <Num>
          |  <Sym>
          |  {+ <Expr> <Expr>}
          |  {* <Expr> <Expr>}
          |  {<Sym> <Expr>}
          |  {let {[<Sym> <Expr>]}     NEW
               <Expr>}
```

```
(define-type ExprC
  [numC (n : number)]
  [idC (s : symbol)]
  [plusC (l : ExprC) (r : ExprC)]
  [multC (l : ExprC) (r : ExprC)]
  [appC (s : symbol) (arg : ExprC)]
  [letC (n : symbol) (rhs : ExprC)
        (body : ExprC)])
```

Part 2

Substitution

```
(interp (parse '{let {[x 8]}
                  {+ x x}})
        ....)

⇒  (interp (subst (parse '8)
                  'x
                  (parse '{+ x x}))
           ....)
```

Sustitutions and Local Binding

```
; 10 for x in  {let {[y 17]} x} ⇒ {let {[y 17]} 10}
(test (subst (numC 10) 'x (letC 'y (numC 17) (idC 'x)))
      (letC 'y (numC 17) (numC 10)))
```

```
; 10 for x in  {let {[y x]} y} ⇒ {let {[y 10]} y}
(test (subst (numC 10) 'x (letC 'y (idC 'x) (idC 'y)))
      (letC 'y (numC 10) (idC 'y)))
```

```
; 10 for x in  {let {[x y]} x} ⇒ {let {[x y]} x}
(test (subst (numC 10) 'x (letC 'x (idC 'y) (idC 'x)))
      (letC 'x (idC 'y) (idC 'x)))
```

```
; 10 for x in  {let {[x x]} x} ⇒ {let {[x 10]} x}
(test (subst (numC 10) 'x (letC 'x (idC 'x) (idC 'x)))
      (letC 'x (numC 10) (idC 'x)))
```

Sustitutions and Local Binding

```
(define (subst [what : ExprC] [for : symbol] [in : ExprC])
  (type-case ExprC in
    ....
    [letC (n rhs body)
          ....]))
```

Sustitutions and Local Binding

```
(define (subst [what : ExprC] [for : symbol] [in : ExprC])
  (type-case ExprC in
    ....
    [letC (n rhs body)
          (letC ....)]))
```

Sustitutions and Local Binding

```
(define (subst [what : ExprC] [for : symbol] [in : ExprC])
  (type-case ExprC in
    ....
    [letC (n rhs body)
          (letC n
                ....)]))
```

Sustitutions and Local Binding

```
(define (subst [what : ExprC] [for : symbol] [in : ExprC])
  (type-case ExprC in
    ....
    [letC (n rhs body)
          (letC n
                (subst what for rhs)
                ....)]))
```

Sustitutions and Local Binding

```
(define (subst [what : ExprC] [for : symbol] [in : ExprC])
  (type-case ExprC in
    ....
    [letC (n rhs body)
          (letC n
                (subst what for rhs)
                (if (symbol=? n for)
                    ....
                    ....))]))
```

Sustitutions and Local Binding

```
(define (subst [what : ExprC] [for : symbol] [in : ExprC])
  (type-case ExprC in
    ....
    [letC (n rhs body)
          (letC n
                (subst what for rhs)
                (if (symbol=? n for)
                    body
                    ....))]))
```

Sustitutions and Local Binding

```
(define (subst [what : ExprC] [for : symbol] [in : ExprC])
  (type-case ExprC in
    ....
    [letC (n rhs body)
          (letC n
                (subst what for rhs)
                (if (symbol=? n for)
                    body
                    (subst what for body)))]))
```

Part 3

Parsing `let`

```
; An expr-S-exp is either ...
; - (list 'let (list (list symbol expr-S-expr))
;              expr-S-expr)
```

```
(and (s-exp-list? s)
     (= 3 (length (s-exp->list s)))
     (s-exp-symbol? (first (s-exp->list s)))
     (eq? 'let (s-exp->symbol (first (s-exp->list s))))
     (s-exp-list? (second (s-exp->list s)))
     (= 1 (length (s-exp->list (second (s-exp->list s)))))
     (s-exp-list? (first (s-exp->list (second (s-exp->list s)))))
     (= 2 (length (s-exp->list
                   (first (s-exp->list
                           (second (s-exp->list s)))))))
     (s-exp-symbol? (first (s-exp->list
                             (first (s-exp->list
                                     (second (s-exp->list s))))))))
```

Parsing `let`

```
; An expr-S-exp is either ...
; - (list 'let (list (list symbol expr-S-expr))
;             expr-S-expr)
```

```
(and (s-exp-list? s)
     (let ([sl (s-exp->list s)])
       (and
        (= 3 (length sl))
        (s-exp-symbol? (first sl))
        (eq? 'let (s-exp->symbol (first sl)))
        (let ([bs (second sl)])
          (and (s-exp-list? bs)
               (= 1 (length (s-exp->list bs)))
               (let ([b (first (s-exp->list bs))])
                 (and (s-exp-list? b)
                      (= 2 (length (s-exp->list b)))
                      (s-exp-symbol?
                       (first (s-exp->list b))))))))))
```

Parsing `let`

```
; An expr-S-exp is either ...
; - (list 'let (list (list symbol expr-S-expr))
;             expr-S-expr)
```

```
(s-exp-match? '{let {[SYMBOL ANY]} ANY} s)
```

Parser with S-Expression Matching

```
(define (parse [s : s-expression]) : ExprC
  (cond
    [(s-exp-match? `NUMBER s) (numC (s-exp->number s))]
    [(s-exp-match? `SYMBOL s) (idC (s-exp->symbol s))]
    [(s-exp-match? '{+ ANY ANY} s)
     (plusC (parse (second (s-exp->list s)))
            (parse (third (s-exp->list s))))]
    [(s-exp-match? '{* ANY ANY} s)
     (multC (parse (second (s-exp->list s)))
            (parse (third (s-exp->list s))))]
    [(s-exp-match? '{SYMBOL ANY} s)
     (appC (s-exp->symbol (first (s-exp->list s)))
           (parse (second (s-exp->list s))))]
    [(s-exp-match? '{let {[SYMBOL ANY]} ANY} s)
     (let ([bs (s-exp->list (first
                              (s-exp->list (second
                                             (s-exp->list s)))))])
       (letC (s-exp->symbol (first bs))
             (parse (second bs))
             (parse (third (s-exp->list s)))))]
    [else (error 'parse "invalid input")]))
```

Getting `s-exp-match?`

- Install the **`plai-typed-s-exp-match`** package
- Add

  ```
  (require plai-typed/s-exp-match)
  ```

  to your program

Part 4

Cost of Substitution

```
(interp {let {[x 1]}
          {let {[y 2]}
            {+ 100 {+ 99 {+ 98 ... {+ y x}}}}}})
```

⇒

```
(interp {let {[y 2]}
          {+ 100 {+ 99 {+ 98 ... {+ y 1}}}}})
```

⇒

```
(interp {+ 100 {+ 99 {+ 98 ... {+ 2 1}}}})
```

With **n** variables, evaluation will take O(**n**^{2}) time!

Deferring Substitution

```
(interp {let {[x 1]}
          {let {[y 2]}
            {+ 100 {+ 99 {+ 98 ... {+ y x}}}}}})
```

⇒

x = 1

```
(interp {let {[y 2]}
          {+ 100 {+ 99 {+ 98 ... {+ y x}}}}})
```

⇒

y = 2   x = 1

```
(interp {+ 100 {+ 99 {+ 98 ... {+ y x}}}})
```

⇒ ... ⇒

y = 2   x = 1

```
(interp y)
```

Deferring Substitution with the Same Identifier

```
(interp {let {[x 1]}
          {let {[x 2]}
            x}})
```

⇒

x = 1

```
(interp {let {[x 2]}
          x})
```

⇒

x = 2   x = 1

```
(interp x)
```

Always add to start, then always check from start

Representing Deferred Substitution: Environments

Change

```
interp : (ExprC (listof FunDef) -> number)
```

to

```
interp : (ExprC Env (listof FunDef) -> number)
```

```
mt-env : Env
extend-env : (Binding Env -> Env)
bind : (symbol number -> Binding)
lookup : (symbol Env -> number)
```

```
mt-env
```

Representing Deferred Substitution: Environments

Change

```
interp : (ExprC (listof FunDef) -> number)
```

to

```
interp : (ExprC Env (listof FunDef) -> number)
```

```
mt-env : Env
extend-env : (Binding Env -> Env)
bind : (symbol number -> Binding)
lookup : (symbol Env -> number)
```

x = 1

```
(extend-env (bind 'x 1)
            mt-env)
```

Representing Deferred Substitution: Environments

Change

```
interp : (ExprC (listof FunDef) -> number)
```

to

```
interp : (ExprC Env (listof FunDef) -> number)
```

```
mt-env : Env
extend-env : (Binding Env -> Env)
bind : (symbol number -> Binding)
lookup : (symbol Env -> number)
```

y = 2 x = 1

```
(extend-env (bind 'y 2)
            (extend-env (bind 'x 1)
                        mt-env))
```

Environments

```
(define-type Binding
  [bind (name : symbol)
        (val : number)])

(define-type-alias Env (listof Binding))

(define mt-env empty)
(define extend-env cons)
```

Environment Lookup

```
(define (lookup [n : symbol] [env : Env]) : number
  (cond
   [(empty? env) (error 'lookup "free variable")]
   [else (cond
           [(symbol=? n (bind-name (first env)))
            (bind-val (first env))]
           [else (lookup n (rest env))])]))
```

Part 5

Interp with Environments

```
(interp {let {[x 1]}
          {let {[y 2]}
            {+ 100 {+ 99 {+ 98 ... {+ y x}}}}}}
        mt-env)

⇒ (interp {let {[y 2]}
            {+ 100 {+ 99 {+ 98 ... {+ y x}}}}}
          (extend-env (bind 'x 1) mt-env))

⇒ (interp {+ 100 {+ 99 {+ 98 ... {+ y x}}}}
          (extend-env (bind 'y 2)
                      (extend-env (bind 'x 1)
                                  mt-env)))

⇒ ...

⇒ (interp y (extend-env (bind 'y 2)
                        (extend-env (bind 'x 1)
                                    mt-env)))
```

Interp with Environments

```
(define (interp [a : ExprC] [env : Env] [fds : (listof FunDefC)])
  (type-case ExprC a
    [numC (n) n]
    [idC (s) ...]
    [plusC (l r) (+ (interp l env fds) (interp r env fds))]
    [multC (l r) (* (interp l env fds) (interp r env fds))]
    [appC (s arg) (local [(define fd (get-fundef s fds))]
                    ...)]
    [letC (n rhs body) ...]))
```

Interp with Environments

```
(define (interp [a : ExprC] [env : Env] [fds : (listof FunDefC)])
  (type-case ExprC a
    [numC (n) n]
    [idC (s) (lookup s env)]
    [plusC (l r) (+ (interp l env fds) (interp r env fds))]
    [multC (l r) (* (interp l env fds) (interp r env fds))]
    [appC (s arg) (local [(define fd (get-fundef s fds))]
                    ...)]
    [letC (n rhs body) ...]))
```

Interp with Environments

```
(define (interp [a : ExprC] [env : Env] [fds : (listof FunDefC)])
  (type-case ExprC a
    [numC (n) n]
    [idC (s) (lookup s env)]
    [plusC (l r) (+ (interp l env fds) (interp r env fds))]
    [multC (l r) (* (interp l env fds) (interp r env fds))]
    [appC (s arg) (local [(define fd (get-fundef s fds))]
                   ...)]
    [letC (n rhs body) ...
                                 ...
                                  ...     (interp rhs env fds)
                                 ...
                      ...          ]))
```

Interp with Environments

```
(define (interp [a : ExprC] [env : Env] [fds : (listof FunDefC)])
  (type-case ExprC a
    [numC (n) n]
    [idC (s) (lookup s env)]
    [plusC (l r) (+ (interp l env fds) (interp r env fds))]
    [multC (l r) (* (interp l env fds) (interp r env fds))]
    [appC (s arg) (local [(define fd (get-fundef s fds))]
                    ...)]
    [letC (n rhs body) ...
                                 ...
                                  (bind n (interp rhs env fds))
                                 ...
                         ...        ]))
```

Interp with Environments

```
(define (interp [a : ExprC] [env : Env] [fds : (listof FunDefC)])
  (type-case ExprC a
    [numC (n) n]
    [idC (s) (lookup s env)]
    [plusC (l r) (+ (interp l env fds) (interp r env fds))]
    [multC (l r) (* (interp l env fds) (interp r env fds))]
    [appC (s arg) (local [(define fd (get-fundef s fds))]
                   ...)]
    [letC (n rhs body) ...
                                (extend-env
                                 (bind n (interp rhs env fds))
                                 env)
                        ...         ]))
```

Interp with Environments

```
(define (interp [a : ExprC] [env : Env] [fds : (listof FunDefC)])
  (type-case ExprC a
    [numC (n) n]
    [idC (s) (lookup s env)]
    [plusC (l r) (+ (interp l env fds) (interp r env fds))]
    [multC (l r) (* (interp l env fds) (interp r env fds))]
    [appC (s arg) (local [(define fd (get-fundef s fds))]
                    ...)]
    [letC (n rhs body) (interp body
                               (extend-env
                                (bind n (interp rhs env fds))
                                env)
                               fds)]))
```

Interp with Environments

```
(define (interp [a : ExprC] [env : Env] [fds : (listof FunDefC)])
  (type-case ExprC a
    [numC (n) n]
    [idC (s) (lookup s env)]
    [plusC (l r) (+ (interp l env fds) (interp r env fds))]
    [multC (l r) (* (interp l env fds) (interp r env fds))]
    [appC (s arg) (local [(define fd (get-fundef s fds))]
                    ...
                            ...
                    ...      ...
                            ...    (interp arg env fds)

                                )]
    [letC (n rhs body) (interp body
                               (extend-env
                                (bind n (interp rhs env fds))
                                env)
                               fds)]))
```

Interp with Environments

```
(define (interp [a : ExprC] [env : Env] [fds : (listof FunDefC)])
  (type-case ExprC a
    [numC (n) n]
    [idC (s) (lookup s env)]
    [plusC (l r) (+ (interp l env fds) (interp r env fds))]
    [multC (l r) (* (interp l env fds) (interp r env fds))]
    [appC (s arg) (local [(define fd (get-fundef s fds))]
                    ...
                            ...
                    ...      (bind (fdC-arg fd)
                            ...    (interp arg env fds))

                                 )]
    [letC (n rhs body) (interp body
                               (extend-env
                                (bind n (interp rhs env fds))
                                env)
                               fds)]))
```

Interp with Environments

```
(define (interp [a : ExprC] [env : Env] [fds : (listof FunDefC)])
  (type-case ExprC a
    [numC (n) n]
    [idC (s) (lookup s env)]
    [plusC (l r) (+ (interp l env fds) (interp r env fds))]
    [multC (l r) (* (interp l env fds) (interp r env fds))]
    [appC (s arg) (local [(define fd (get-fundef s fds))]
                    (interp (fdC-body fd)
                            ...
                             (bind (fdC-arg fd)
                            ...      (interp arg env fds))

                            fds))]
    [letC (n rhs body) (interp body
                               (extend-env
                                (bind n (interp rhs env fds))
                                env)
                               fds)]))
```

Function Calls

```
{define {bad x} {+ x y}}
```

```
(interp {let {[y 2]}
          {bad 10}} )
```

⇒

y = 2

```
(interp {bad 10})
```

⇒

...

```
(interp {+ x y})
```

Function Calls

```
{define {bad x} {+ x y}}

(interp {let {[y 2]}
          {bad 10}} )
```

⇒

```
                    y = 2
(interp {bad 10})
```

⇒

```
                   x = 10
(interp {+ x y})
```

Interpreting function body starts with only one substitution

Interp with Environments

```
(define (interp [a : ExprC] [env : Env] [fds : (listof FunDefC)])
  (type-case ExprC a
    [numC (n) n]
    [idC (s) (lookup s env)]
    [plusC (l r) (+ (interp l env fds) (interp r env fds))]
    [multC (l r) (* (interp l env fds) (interp r env fds))]
    [appC (s arg) (local [(define fd (get-fundef s fds))]
                     (interp (fdC-body fd)
                             ...
                              (bind (fdC-arg fd)
                             ...    (interp arg env fds))

                             fds))]
    [letC (n rhs body) (interp body
                                (extend-env
                                 (bind n (interp rhs env fds))
                                 env)
                                fds)]))
```

Interp with Environments

```
(define (interp [a : ExprC] [env : Env] [fds : (listof FunDefC)])
  (type-case ExprC a
    [numC (n) n]
    [idC (s) (lookup s env)]
    [plusC (l r) (+ (interp l env fds) (interp r env fds))]
    [multC (l r) (* (interp l env fds) (interp r env fds))]
    [appC (s arg) (local [(define fd (get-fundef s fds))]
                (interp (fdC-body fd)
                        (extend-env
                         (bind (fdC-arg fd)
                               (interp arg env fds))
                         mt-env)
                        fds))]
    [letC (n rhs body) (interp body
                           (extend-env
                            (bind n (interp rhs env fds))
                            env)
                           fds)]))
```

Part 6

Binding Terminology

binding — where an identifier gets its meaning

```
{let {[x 5]} ....}
{define {f x} ....}
```

bound — refers to a binding

```
{let {[x 5]} .... x ....}
{define {f x} .... x ....}
```

free — does not have a binding

```
{let {[x 5]} .... y ....}
{define {f x} .... y ....}
```

Free and Bound

```
{let {[x 5]}
  {let {[y x]}
    {+ y {+ z x}}}}
```

Free and Bound

```
{let {[x 5]}
  {let {[y x]}
    {+ y {+ z x}}}}
```

Free and Bound

```
{let {[x 5]}
  {let {[y x]}
    {+ y {+ z x}}}}
```

Free and Bound

```
{let {[x 5]}
  {let {[x x]}
    {+ y {+ z x}}}}
```

Free and Bound

```
{let {[x 5]}
  {let {[x x]}
    {+ y {+ z x}}}}
```

Free and Bound

```
{let {[x 5]}
  {let {[x x]}
    {+ y {+ z x}}}}
```

Free and Bound

```
{define {double x} {+ x x}}

{double 3}
```

Free and Bound

```
{define {double x} {+ x x}}
{double 3}
```