Part I

Languages and Sugar

```
<Expr>  ::=  <Num>
          |  {+ <Expr> <Expr>}
          |  {* <Expr> <Expr>}
          |  <Sym>
          |  {lambda {<Sym>} <Expr>}
          |  {<Expr> <Expr>}
          |  {if0 <Expr> <Expr> <Expr>}
```

Languages and Sugar

```
<Expr>  ::=  <Num>
          |  {+ <Expr> <Expr>}
          |  {* <Expr> <Expr>}
          |  <Sym>
          |  {lambda {<Sym>} <Expr>}
          |  {<Expr> <Expr>}
          |  {if0 <Expr> <Expr> <Expr>}
          |  {let {[<Sym> <Expr>]} <Expr>}
```

Languages and Sugar

```
<Expr>  ::=  <Num>
          |  {+ <Expr> <Expr>}
          |  {* <Expr> <Expr>}
          |  <Sym>
          |  {lambda {<Sym>} <Expr>}
          |  {<Expr> <Expr>}
          |  {if0 <Expr> <Expr> <Expr>}
          |  {let {[<Sym> <Expr>]} <Expr>}
          |  {add1 <Expr>}
```

Languages and Sugar

```
<Expr>  ::=  <Num>
          |  {+ <Expr> <Expr>}
          |  {* <Expr> <Expr>}
          |  <Sym>
          |  {lambda {<Sym>} <Expr>}
          |  {<Expr> <Expr>}
          |  {if0 <Expr> <Expr> <Expr>}
          |  {let {[<Sym> <Expr>]} <Expr>}
```

```
{let {[add1 {lambda {n}
                    {+ n 1}}]}
 .... {add1 x} ....}
```

Languages and Sugar

Potential sugar:

```
{- <Expr> <Expr>}

{case <Expr>
 [{<Num>} <Expr>]
 ...
 [else <Expr>]}

{thunk <Expr>}

{force <Expr>}
```

Part 2

Replacing a Parser

We can add sugar to a language by replacing the **`parse`** function while reusing **`interp`**

```
(require plai-typed/s-exp-match
         "core.rkt")

(define (parse2 [s : s-expression]) : ExprC
  (cond
   [(s-exp-match? `NUMBER s) (numC (s-exp->number s))]
   ....
   [(s-exp-match? '{let {[SYMBOL ANY]} ANY} s)
    (let ([bs ....])
      (parse2 `{{lambda {,(first bs)}
                        ,(third (s-exp->list s))}
                ,(second bs)}))]
   [(s-exp-match? '{case ANY ....} s)
    ....]
   ....))
```

Extending a Parser

Better: Set up an ***extensible*** parser to support new forms:

```
(define (parse* [s : s-expression]
                [expand : (s-expression
                           -> (optionof s-expression))])
  (type-case (optionof s-expression) (expand s)
    [some (e) (parse* e expand)]
    [none ()
     (cond
       [(s-exp-match? `NUMBER s) (numC (s-exp->number s))]
       ....)]))

(define (parse [s : s-expression]) : ExprC
  (parse* s (lambda (s) (none))))
```

Extending a Parser

Better: Set up an ***extensible*** parser to support new forms:

```
(define (parse* [s : s-expre
                [expand : (s
                                -> (optionof s-expression))])
  (type-case (optionof s-expression) (expand s)
    [some (e) (parse* e expand)]
    [none ()
     (cond
       [(s-exp-match? `NUMBER s) (numC (s-exp->number s))]
       ....)]))

(define (parse [s : s-expression]) : ExprC
  (parse* s (lambda (s) (none))))
```

expand gets first shot at rewriting each S-expression

Extending a Parser

Better: Set up an ***extensible*** parser to support new forms:

```
(define (parse2 [s : s-expression]) : ExprC
  (parse* s try-let+case))

(define (try-let+case [s : s-expression])
  (cond
   [(s-exp-match? '{let {[SYMBOL ANY]} ANY} s)
    (let ([bs ....])
      (some `{{lambda {,(first bs)}
                      ,(third (s-exp->list s))}
              ,(second bs)}))]
   [(s-exp-match? '{case ....} s)
    (some ....)]
   [else (none)]))
```

Part 3

Extensible Languages

An ***extensible parser*** provides hooks for a language *implementer* to add sugar

- Programs that use the new sugar must be run with the extended implementation
- Different sets of extensions don't work together

An ***extensible language*** provides hooks for a language *user* to add sugar

- Programs can use new sugar with the existing implementation
- Different sets of extensions can work together

Language Extension via Macros

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

Language Extension via Macros

```
{let-macro {[let ....]}
  {let {[x 5]}
    {+ x x}}}
```

Language Extension via Macros

a function that acts like **`expand`** for forms that start with **`let`**

```
{let-macro {[let ....]}
  {let {[x 5]}
    {+ x x}}}
```

Our extensible language needs

- **`let-macro`**
- S-expression values and operations

S-expression Type versus Values

`plai-typed`'s **`s-expressions`** is a *type*, not a representation

In our untyped language, an S-expression is one of

- number
- symbol
- list of S-expressions

So, we need to add symbols and lists

Part 4

Adding `let-macro`

```
(define (parse [s : s-expression]) : ExprC
  (cond
   [(s-exp-match? `{let-macro {[SYMBOL ANY]} ANY} s)
                                                  ]
    ....))
```

Adding `let-macro`

```
(define (parse [s : s-expression]) : ExprC
  (cond
   [(s-exp-match? `{let-macro {[SYMBOL ANY]} ANY} s)


      (parse (third (s-exp->list s)))


                                    ]
   ....))
```

Adding `let-macro`

```
(define (parse [s : s-expression]) : ExprC
  (cond
   [(s-exp-match? `{let-macro {[SYMBOL ANY]} ANY} s)
    (let ([bs (s-exp->list (first
                            (s-exp->list (second
                                          (s-exp->list s)))))])
      (parse (third (s-exp->list s)))


                                    )]
   ....))
```

Adding `let-macro`

```
(define (parse [s : s-expression]) : ExprC
  (cond
   [(s-exp-match? `{let-macro {[SYMBOL ANY]} ANY} s)
    (let ([bs (s-exp->list (first
                            (s-exp->list (second
                                          (s-exp->list s)))))])
      (parse (third (s-exp->list s)))
                                (s-exp->symbol (first bs))


                               )]
    ....))
```

Adding `let-macro`

```
(define (parse* [s : s-expression] [env : Env]) : ExprC
  (cond
   [(s-exp-match? `{let-macro {[SYMBOL ANY]} ANY} s)
    (let ([bs (s-exp->list (first
                            (s-exp->list (second
                                          (s-exp->list s)))))])
      (parse* (third (s-exp->list s))
              (extend-env (bind (s-exp->symbol (first bs))

                                )
                          env)))]
   ....))
```

Adding `let-macro`

```
(define (parse* [s : s-expression] [env : Env]) : ExprC
  (cond
   [(s-exp-match? `{let-macro {[SYMBOL ANY]} ANY} s)
    (let ([bs (s-exp->list (first
                            (s-exp->list (second
                                          (s-exp->list s)))))])
      (parse* (third (s-exp->list s))
              (extend-env (bind (s-exp->symbol (first bs))
                                (parse (second bs))
                                )
                          env)))]
   ....))
```

Adding `let-macro`

```
(define (parse* [s : s-expression] [env : Env]) : ExprC
  (cond
   [(s-exp-match? `{let-macro {[SYMBOL ANY]} ANY} s)
    (let ([bs (s-exp->list (first
                            (s-exp->list (second
                                          (s-exp->list s)))))])
      (parse* (third (s-exp->list s))
              (extend-env (bind (s-exp->symbol (first bs))
                                (interp (parse (second bs))
                                        mt-env))
                          env)))]
   ....))
```

Adding `let-macro`

```
(define (parse* [s : s-expression] [env : Env]) : ExprC
  (cond
   [(s-exp-match? `{let-macro {[SYMBOL ANY]} ANY} s)
    (let ([bs (s-exp->list (first
                            (s-exp->list (second
                                          (s-exp->list s)))))])
      (parse* (third (s-exp->list s))
              (extend-env (bind (s-exp->symbol (first bs))
                                (interp (parse (second bs))
                                        mt-env))
                          env)))]
   ....))
```

parse calls **interp**!

Adding `let-macro`

```
(define (parse* [s : s-expression] [env : Env]) : ExprC
  (cond
   [(s-exp-match? `{let-macro {[SYMBOL ANY]} ANY} s)
    (let ([bs (s-exp->list (first
                            (s-exp->list (second
                                          (s-exp->list s)))))])
      (parse* (third (s-exp->list s))
              (extend-env (bind (s-exp->symbol (first bs))
                                (interp (parse (second bs))
                                        mt-env))
                          env)))]
   ....))
```

Parse-time **`interp`** cannot see run-time variables

Adding `let-macro`

```
(define (parse* [s : s-expression] [env : Env]) : ExprC
  (cond
   ....
   [(s-exp-match? '{ANY ANY ...} s)
     ]))
```

Adding `let-macro`

```
(define (parse* [s                                    ExprC
  (cond
   ....
   [(s-exp-match? '{ANY ANY ...} s)
     ]))
```

Maybe a function call, maybe a macro use

Adding `let-macro`

```
(define (parse* [s : s-expression] [env : Env]) : ExprC
  (cond
   ....
   [(s-exp-match? '{ANY ANY ...} s)
    (let ([rator (first (s-exp->list s))])
      )]))
```

Adding `let-macro`

```
(define (parse* [s : s-expression] [env : Env]) : ExprC
  (cond
   ....
   [(s-exp-match? '{ANY ANY ...} s)
    (let ([rator (first (s-exp->list s))])
      (try



       (lambda ()
         (if (= (length (s-exp->list s)) 2)
             (appC (parse* rator env)
                   (parse* (second (s-exp->list s)) env))
             (error 'parse "invalid input"))))))]))
```

Adding `let-macro`

```
(define (parse* [s : s-expression] [env : Env]) : ExprC
  (cond
   ....
   [(s-exp-match? '{ANY ANY ...} s)
    (let ([rator (first (s-exp->list s))])
      (try
                              (lookup (s-exp->symbol rator) env)


       (lambda ()
         (if (= (length (s-exp->list s)) 2)
             (appC (parse* rator env)
                   (parse* (second (s-exp->list s)) env))
             (error 'parse "invalid input"))))))]))
```

Adding `let-macro`

```
(define (parse* [s : s-expression] [env : Env]) : ExprC
  (cond
   ....
   [(s-exp-match? '{ANY ANY ...} s)
    (let ([rator (first (s-exp->list s))])
      (try
               (apply-macro (lookup (s-exp->symbol rator) env)
                            s)

       (lambda ()
         (if (= (length (s-exp->list s)) 2)
             (appC (parse* rator env)
                   (parse* (second (s-exp->list s)) env))
             (error 'parse "invalid input"))))))]))
```

Adding `let-macro`

```
(define (parse* [s : s-expression] [env : Env]) : ExprC
  (cond
   ....
   [(s-exp-match? '{ANY ANY ...} s)
    (let ([rator (first (s-exp->list s))])
      (try
       (parse* (apply-macro (lookup (s-exp->symbol rator) env)
                            s)
               env)
       (lambda ()
         (if (= (length (s-exp->list s)) 2)
             (appC (parse* rator env)
                   (parse* (second (s-exp->list s)) env))
             (error 'parse "invalid input"))))))]))
```

Adding `let-macro`

```
(define (apply-macro [transformer : Value] [s : s-expression])
  (type-case Value transformer
    [closV (arg body env)
           (value->s-exp
            (interp body (extend-env
                          (bind arg (s-exp->value s))
                          env)))]
    [else (error 'apply-macro "not a function")]))
```

Example Use of `let-macro`

```
(parse '{let-macro {[delay {lambda {s}
                             {cons 'lambda
                                   {cons '{dummy}
                                         {cons {first {rest s}}
                                               '{}}}}}]}
         {let-macro {[force {lambda {s}
                              {cons {first {rest s}}
                                    '{0}}}]}
           {force {delay 7}}}})
```

Part 5

Accidental Capture

```
(parse '{let-macro {[delay {lambda {s}
                             {cons 'lambda
                                   {cons '{dummy}
                                         {cons {first {rest s}}
                                               '{}}}}}]}
         {let-macro {[force {lambda {s}
                              {cons {first {rest s}}
                                    '{0}}}]}
           {let {[dummy 8]}
             {force {delay dummy}}}}})
```

Macros must be careful to invent names for new variables

The **`gensym`** function makes a fresh symbol