Part I

Languages and Sugar

```
<Exp>  ::=  <Number>
         |  {+ <Exp> <Exp>}
         |  {* <Exp> <Exp>}
         |  <Symbol>
         |  {lambda {<Symbol>} <Exp>}
         |  {<Exp> <Exp>}
         |  {if0 <Exp> <Exp> <Exp>}
```

Languages and Sugar

```
<Exp>  ::=  <Number>
         |  {+ <Exp> <Exp>}
         |  {* <Exp> <Exp>}
         |  <Symbol>
         |  {lambda {<Symbol>} <Exp>}
         |  {<Exp> <Exp>}
         |  {if0 <Exp> <Exp> <Exp>}
         |  {let {[<Symbol> <Exp>]} <Exp>}
```

Languages and Sugar

```
<Exp>  ::=  <Number>
         |  {+ <Exp> <Exp>}
         |  {* <Exp> <Exp>}
         |  <Symbol>
         |  {lambda {<Symbol>} <Exp>}
         |  {<Exp> <Exp>}
         |  {if0 <Exp> <Exp> <Exp>}
         |  {let {[<Symbol> <Exp>]} <Exp>}
         |  {add1 <Exp>}
```

Languages and Sugar

```
<Exp>  ::=  <Number>
         |  {+ <Exp> <Exp>}
         |  {* <Exp> <Exp>}
         |  <Symbol>
         |  {lambda {<Symbol>} <Exp>}
         |  {<Exp> <Exp>}
         |  {if0 <Exp> <Exp> <Exp>}
         |  {let {[<Symbol> <Exp>]} <Exp>}
```

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

Languages and Sugar

Potential sugar:

```
{- <Exp> <Exp>}
```

```
{case <Exp>
  [{<Number>} <Exp>]
  ...
  [else <Exp>]}
```

```
{delay <Exp>}
```

```
{force <Exp>}
```

Part 2

S-Expressions with and without Types

Plait:

```
(s-exp->number (first (s-exp->list `{1 2})))
        `{1 ,(number->s-exp (+ 2 3))}
```

Racket:

```
(first `{1 2})
`{1 ,(+ 2 3)}
```

Part 3

S-Expressions

In Plait:

```
(define (expand-delay [s : S-Exp])
  `{lambda {dummy} ,(second (s-exp->list s))})

(test (expand-delay `{delay {+ 1 2}})
      `{lambda {dummy} {+ 1 2}})
```

In Racket:

```
(define (expand-delay s)
  `{lambda {dummy} ,(second s)})

(require (only-in plai test))
(test (expand-delay `{delay {+ 1 2}})
      `{lambda {dummy} {+ 1 2}})
```

Primitive S-Expression Construction

```
(define (expand-delay s)
  `{lambda {dummy} ,(second s)})
```

=

```
(define (expand-delay s)
  (list 'lambda
        `{dummy}
        (second s)))
```

=

```
(define (expand-delay s)
  (cons 'lambda
        (cons `{dummy}
              (cons (second s)
                    '{}))))
```

Primitive S-Expression Construction

```
(define (expand-delay s)
  `{lambda {dummy} ,(second s)})
```

=

```
(define (expand-delay s)
  (list 'lambda
        `{dummy}
        (second s)))
```

=

```
(define (expand-delay s)
  (cons 'lambda
        (cons (list 'dummy)
              (cons (second s)
                    '{}))))
```

Primitive S-Expression Construction

```
(define (expand-delay s)
  `{lambda {dummy} ,(second s)})
```

=

```
(define (expand-delay s)
  (list 'lambda
        `{dummy}
        (second s)))
```

=

```
(define (expand-delay s)
  (cons 'lambda
        (cons '{dummy}
              (cons (second s)
                    '{}))))
```

Primitive S-Expression Construction

```
(define (expand-delay s)
  `{lambda {dummy} ,(second s)})
```

=

```
(define (expand-delay s)
  (list 'lambda
        `{dummy}
        (second s)))
```

=

```
(define (expand-delay s)
  (cons 'lambda
        (cons '{dummy}
              (cons (first (rest s))
                    '{}))))
```

Expansion Function

```
(define (expand-delay s)
        (cons 'lambda
              (cons '{dummy}
                    (cons (first (rest s))
                          '{}))))
(expand-delay `{delay {+ 1 2}})
```

Expansion Function

```
(define expand-delay
     (lambda (s)
       (cons 'lambda
             (cons '{dummy}
                   (cons (first (rest s))
                         '{})))))
(expand-delay `{delay {+ 1 2}})
```

Expansion Function

```
(let ([expand-delay
       (lambda (s)
         (cons 'lambda
               (cons '{dummy}
                     (cons (first (rest s))
                           '{}))))])
  (expand-delay `{delay {+ 1 2}}))
```

Expansion Function

```
(let ([delay
       (lambda (s)
         (cons 'lambda
               (cons '{dummy}
                     (cons (first (rest s))
                           '{}))))])
  (delay `{delay {+ 1 2}}))
```

Part 4

Recursive Expansion

```
{case <Exp>
  [{<Number>} <Exp>]
  ...
  [else <Exp>]}
```

```
{case {+ 1 2}
  [{3} 'three]
  [{4} 'four]
  [else 'no]}
```

⇒

```
{let {[tmp {+ 1 2}]}
  {if0 {- tmp 3}
       'three
       {if0 {- tmp 4}
            'four
            'no}}}
```

Recursive Expansion

```
{case <Exp>
  [{<Number>} <Exp>]
  ...
  [else <Exp>]}
```

```
{case {+ 1 2}
  [{3} 'three]
  [{4} 'four]
  [else 'no]}
```

⇒

```
{let {[x {+ 1 2}]}
  {if0 {- x 3}
       'three
       {case x
         [{4} 'four]
         [else 'no]}}}
```

Recursive Expansion

```
{case <Exp>
  [{<Number>} <Exp>]
  ...
  [else <Exp>]}
```

```
{case {+ 1 2}
  [{3} 'three]
  [{4} 'four]
  [else 'no]}
```

⇒

```
{let {[x {+ 1 2}]}
  {if0 {- x 3}
       'three
       {case x
         [{4} 'four]
         [else 'no]}}}
```

Recursive Expansion

```
{case <Exp>
  [{<Number>} <Exp>]
  ...
  [else <Exp>]}
```

```
{case x                ⇒   {let {[tmp x]}
  [else 'no]}                 'no}
```

`case` Expander

```
(let ([case (lambda (s)
              (if (and (symbol? (first (third s)))
                       (symbol=? 'else (first (third s))))
                  `{let {[tmp ,(second s)]}
                     ,(second (third s))}
                  `{let {[tmp ,(second s)]}
                     {if0 {- tmp ,(first (first (third s)))}
                          ,(second (third s))
                          ,(cons 'case
                                 (cons 'tmp
                                       (rest (rest (rest s))))))}}))])
  (test (case `{case x
                 [else 'no]})
        `{let {[tmp x]}
           'no}))
```

`case` Expander

```
(let ([case (lambda (s)
              (if (and (symbol? (first (third s)))
                       (symbol=? 'else (first (third s))))
                  `{let {[tmp ,(second s)]}
                     ,(second (third s))}
                  `{let {[tmp ,(second s)]}
                     {if0 {- tmp ,(first (first (third s)))}
                          ,(second (third s))
                          ,(cons 'case
                                 (cons 'tmp
                                       (rest (rest (rest s))))))}}))])
  (test (case `{case {+ 1 2}
                 [{3} 'three]
                 [{4} 'four]
                 [else 'no]})
        `{let {[tmp (+ 1 2)]}
           {if0 {- tmp 3}
                'three
                {case tmp
                  [{4} 'four]
                  [else 'no]}}}))
```

Part 5

Replacing a Parser

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

```
(require "core.rkt")

(define (parse2 [s : S-Exp]) : Exp
  (cond
   [(s-exp-match? `NUMBER s) (numE (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-Exp]
                [expand : (S-Exp -> (Optionof S-Exp))])
  (type-case (Optionof S-Exp) (expand s)
    [(some e) (parse* e expand)]
    [(none)
     (cond
       [(s-exp-match? `NUMBER s) (numE (s-exp->number s))]
       ....)]))

(define (parse [s : S-Exp]) : Exp
  (parse* s (lambda (s) (none))))
```

Extending a Parser

Better: Set up an ***extensi*** ms:

expand gets first shot at rewriting each S-expression

```
(define (parse* [s : S-Exp]
                [expand : (S-Exp -> (Optionof S-Exp))])
  (type-case (Optionof S-Exp) (expand s)
    [(some e) (parse* e expand)]
    [(none)
     (cond
       [(s-exp-match? `NUMBER s) (numE (s-exp->number s))]
       ....)]))

(define (parse [s : S-Exp]) : Exp
  (parse* s (lambda (s) (none))))
```

Extending a Parser

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

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

(define (try-let+case [s : S-Exp])
  (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 6

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 E

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
- **`quote`**

S-Expression Values

Plait S-expressions are distinct from lists and symbols

In untyped Racket or Curly, an S-expression is one of

- number
- symbol
- list of S-expressions

So, we need to add symbols and lists

Part 7

Adding `let-macro`

```
<Exp>  ::=  <Number>
         |  {+ <Exp> <Exp>}
         |  {* <Exp> <Exp>}
         |  <Symbol>
         |  {lambda {<Symbol>} <Exp>}
         |  {<Exp> <Exp>}
         |  {if0 <Exp> <Exp> <Exp>}
         |  {cons <Exp> <Exp>}
         |  …
         |  {let-macro {[<Symbol> <Exp>]} <Exp>}
```

```
{let-macro {[delay {lambda {s}
                     {cons 'lambda ....}}]}
  {delay 7}}
```

Adding `let-macro`

```
<Exp>  ::=  <Number>
         |  {+ <Exp> <Exp>}
         |  {* <Exp> <Exp>}
         |  <Symbol>
         |  {lambda {<Symbol>} <Exp>}
         |  {<Exp> <Exp>}
         |  {if0 <Exp> <Exp> <Exp>}
         |  {cons <Exp> <Exp>}
         |  …
         |  {let-macro {[<Symbol> <Exp>]} <Exp>}
```

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

Adding `let-macro`

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

```
{let-macro {[delay {lambda {s}
                     {cons 'lambda ....}}]}
  {delay 7}}
```

Adding `let-macro`

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


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


                                          ]
     ....))
```

```
{let-macro {[delay {lambda {s}
                     {cons 'lambda ....}}]}
  {delay 7}}
```

Adding `let-macro`

```
(define (parse [s : S-Exp]) : Exp
  (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)))


                                        )]
     ....))
```

```
{let-macro {[delay {lambda {s}
                     {cons 'lambda ....}}]}
  {delay 7}}
```

Adding `let-macro`

```
(define (parse [s : S-Exp]) : Exp
  (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))


                                  )]
     ....))
```

```
{let-macro {[delay {lambda {s}
                     {cons 'lambda ....}}]}
  {delay 7}}
```

Adding `let-macro`

```
(define (parse* [s : S-Exp] [env : Env]) : Exp
  (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)))]
    ....))
```

```
{let-macro {[delay {lambda {s}
                     {cons 'lambda ....}}]}
  {delay 7}}
```

Adding `let-macro`

```
(define (parse* [s : S-Exp] [env : Env]) : Exp
  (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)))]
    ....))
```

```
{let-macro {[delay {lambda {s}
                     {cons 'lambda ....}}]}
  {delay 7}}
```

Adding `let-macro`

```
(define (parse* [s : S-Exp] [env : Env]) : Exp
  (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)))]
    ....))
```

```
{let-macro {[delay {lambda {s}
                     {cons 'lambda ....}}]}
  {delay 7}}
```

Adding `let-macro`

```
(define (parse* [s : S-Exp] [env : Env]) : Exp
  (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!

```
{let-macro {[delay {lambda {s}
                     {cons 'lambda ....}}]}
  {delay 7}}
```

Adding `let-macro`

```
(define (parse* [s : S-Exp] [env : Env]) : Exp
  (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

```
{let-macro {[delay {lambda {s}
                     {cons 'lambda ....}}]}
  {delay 7}}
```

Adding `let-macro`

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

```
{delay 7}
```

Adding `let-macro`

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

Maybe a function call, maybe a macro use

```
{delay 7}
```

Adding `let-macro`

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

```
{delay 7}
```

Adding `let-macro`

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



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

`{delay 7}`

Adding `let-macro`

```
(define (parse* [s : S-Exp] [env : Env]) : Exp
  (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)
             (appE (parse* rator env)
                   (parse* (second (s-exp->list s)) env))
             (error 'parse "invalid input"))))))]))
```

`{delay 7}`

Adding `let-macro`

```
(define (parse* [s : S-Exp] [env : Env]) : Exp
  (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)
             (appE (parse* rator env)
                   (parse* (second (s-exp->list s)) env))
             (error 'parse "invalid input"))))))]))
```

`{delay 7}`

Adding `let-macro`

```
(define (parse* [s : S-Exp] [env : Env]) : Exp
  (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)
             (appE (parse* rator env)
                   (parse* (second (s-exp->list s)) env))
             (error 'parse "invalid input"))))))]))
```

```
{delay 7}
```

Adding `let-macro`

```
(define (apply-macro [transformer : Value] [s : S-Exp])
  (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 8

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