Part I

Languages and Sugar

```
<Exp>  ::=  <Int>
         |  <Exp> + <Exp>
         |  <Exp> * <Exp>
         |  <Symbol>
         |  fun (<Symbol>): <Exp>
         |  <Exp>(<Exp>)
         |  if <Exp> == 0 | <Exp> | <Exp>
```

Languages and Sugar

```
<Exp>  ::=  <Int>
         |  <Exp> + <Exp>
         |  <Exp> * <Exp>
         |  <Symbol>
         |  fun (<Symbol>): <Exp>
         |  <Exp>(<Exp>)
         |  if <Exp> == 0 | <Exp> | <Exp>
         |  let <Symbol> = <Exp>: <Exp>
```

Languages and Sugar

```
<Exp>  ::=  <Int>
         |  <Exp> + <Exp>
         |  <Exp> * <Exp>
         |  <Symbol>
         |  fun (<Symbol>): <Exp>
         |  <Exp>(<Exp>)
         |  if <Exp> == 0 | <Exp> | <Exp>
         |  let <Symbol> = <Exp>: <Exp>
         |  add1(<Exp>)
```

Languages and Sugar

```
<Exp>  ::=  <Int>
         |  <Exp> + <Exp>
         |  <Exp> * <Exp>
         |  <Symbol>
         |  fun (<Symbol>): <Exp>
         |  <Exp>(<Exp>)
         |  if <Exp> == 0 | <Exp> | <Exp>
         |  let <Symbol> = <Exp>: <Exp>
```

```
let add1 = (fun (n):
               n + 1):
  .... add1(x) ....
```

Languages and Sugar

```
<Exp>  ::=  <Int>
         |  <Exp> + <Exp>
         |  <Exp> * <Exp>
         |  <Symbol>
         |  fun (<Symbol>): <Exp>
         |  <Exp>(<Exp>)
         |  if <Exp> == 0 | <Exp> | <Exp>
         |  let <Symbol> = <Exp>: <Exp>
```

```
let add1 = (fun (n):
               n + 1):
  .... add1(x) ....
```

implemented in Moe instead of changing `parse`

Languages and Sugar

Potential sugar:

```
<Exp> - <Exp>
```

```
cond
| <Exp> == <Int>: <Exp>
| ...
| ~else: <Exp>
```

```
delay: <Exp>
```

```
force(<Exp>)
```

Languages and Sugar

Potential sugar:

```
<Exp> - <Exp>
```

```
cond
| <Exp> == <Int>: <Exp>
| ...
| ~else: <Exp>
```

What if we could implement new forms in Moe, too?

```
delay: <Exp>
```

```
force(<Exp>)
```

Languages and Sugar

Potential sugar:

```
<Exp> - <Exp>
```

```
cond
| <Exp> == <Int>: <Exp>
| ...
| ~else: <Exp>
```

What if we could implement new forms in Moe, too?

```
delay: <Exp>
```

```
force(<Exp>)
```

An ***extensible language*** supports new forms via ***macros***

Macros in Shplait

```
macro 'reslet ($v_id, $sto_id) = $call:
         $body':
  'match $call
   | res($v_id, $sto_id):
       $body'
```

```
reslet (v_r, sto_r) = interp(r, env, sto_l):
  res(num_plus(v_l, v_r), sto_r)
```

Macros in Shplait

```
macro 'reslet ($v_id, $sto_id) = $call:
         $body':
  'match $call
   | res($v_id, $sto_id):
       $body'
```

```
reslet (v_r, sto_r) = interp(r, env, sto_l):
  res(num_plus(v_l, v_r), sto_r)
⇒
match interp(r, env, sto_l)
| res(v_r, sto_r):
    res(num_plus(v_l, v_r), sto_r)
```

Macros in Shplait

```
macro 'reslet ($v_id, $sto_id) = $call:
         $body':
  'match $call
   | res($v_id, $sto_id):
       $body'
```

Resembles a pattern and template within `parse`

```
reslet (v_r, sto_r) = interp(r, env, sto_l):
  res(num_plus(v_l, v_r), sto_r)
⇒
match interp(r, env, sto_l)
| res(v_r, sto_r):
    res(num_plus(v_l, v_r), sto_r)
```

Macros in Shplait

```
macro 'reslet ($v_id, $sto_id) = $call:
         $body':
  'match $call
   | res($v_id, $sto_id):
       $body'
```

More primitive variant:

```
def_macro reslet:
  fun (stx):
    match stx
    | 'reslet ($v_id, $sto_id) = $call:
         $body':
       'match $call
        | res($v_id, $sto_id):
            $body'
```

Macros in Shplait

```
macro 'reslet ($v_id, $sto_id) = $call:
         $body':
  'match $call
   | res($v_id, $sto_
       $body'
```

Tells Shplait's own `parse` to call this function when an expression starts with `reslet`, then recur to `parse` on the result

More primitive variant:

```
def_macro reslet:
  fun (stx):
    match stx
     | 'reslet ($v_id, $sto_id) = $call:
          $body':
        'match $call
         | res($v_id, $sto_id):
             $body'
```

Part 2

Syntax Objects in Moe

Add patterns and templates to Moe in a simplified `match` form:

```
match s
| 'let %id = %rhs: %body':
    '(fun (%id): %body)(%rhs)'
| ~else: 'fail'
```

Using `%` makes it easier to quote Mode programs in Shplait

	Moe with `$`	*Moe with* `%`
Moe program	`(1 + $x)`	`(1 + %x)`
Shplait expression	`'(1 + $('$')x)'`	`'(1 + %x)'`

Syntax Objects in Moe

Add patterns and templates to Moe in a simplified **`match`** form:

```
match s
| 'let %id = %rhs: %body':
    '(fun (%id): %body)(%rhs)'
| ~else: 'fail'
```

Using `''` is also a hassle, but not as bad

Moe program `'1 + %x'`

Shplait expression `'« '1 + %x' »'`

while `'  '1 + %x'  '` would be `'' 1 + %x ''`

Syntax Objects in Moe

Add patterns and templates to Moe in a simplified **`match`** form:

```
match s
| 'let %id = %rhs: %body':
    '(fun (%id): %body)(%rhs)'
| ~else: 'fail'
```

Using `''` is also a hassle, but not as bad

Moe program `('1 + %x')`

Shplait expression `'('1 + %x')'`

Syntax Objects in Moe

Add patterns and templates to Moe in a simplified **`match`** form:

```
match s
| 'let %id = %rhs: %body':
    '(fun (%id): %body)(%rhs)'
| ~else: 'fail'
```

To understand how matching works, we'll implement it ourselves, instead of using Shplait's matching

Anything inside `''` conforms to **Shrubbery** notation

Part 3

Shrubbery Grammar

```
<Group>  ::=  <Term> <Term> ...

<Term>   ::=  <Int>
           |  <Boolean>
           |  <String>
           |  <Symbol>
           |  <Operator>
           |  ( <Group> , ... )
           |  [ <Group> , ... ]
           |  { <Group> , ... }
           |  ' <Group> ; ... '
           |  : <Block>
           |  <Alts>

<Block>  ::=  <Group> ; ...

<Alts>   ::=  | <Block> | ...
```

Shrubbery Grammar

```
<Group>  ::=  <Term> <Term> ...

<Term>   ::=  <Int>
           |  <Boolean>
           |  <String>
           |  <Symbol>
           |  <Operator>
           |  ( <Group> , ... )
           |  [ <Group> , ... ]
           |  { <Group> , ... }
           |  ' <Group> ; ... '
           |  : <Block>
           |  <Alts>

<Block>  ::=  <Group> ; ...

<Alts>   ::=  | <Block> | ...
```

```
fun (x, y):
  println("add")
  x + y
```

Shrubbery Grammar

```
<Group>  ::=  <Term> <Term> ...

<Term>   ::=  <Int>
           |  <Boolean>
           |  <String>
           |  <Symbol>
           |  <Operator>
           |  ( <Group> , ... )
           |  [ <Group> , ... ]
           |  { <Group> , ... }
           |  ' <Group> ; ... '
           |  : <Block>
           |  <Alts>

<Block>  ::=  <Group> ; ...

<Alts>   ::=  | <Block> | ...
```

```
'add1(x + 2)
 sub1(w)'
```

Shrubbery Grammar

```
<Group>  ::=  <Term> <Term> ...

<Term>   ::=  <Int>
           |  <Boolean>
           |  <String>
           |  <Symbol>
           |  <Operator>
           |  ( <Group> , ... )
           |  [ <Group> , ... ]
           |  { <Group> , ... }
           |  ' <Group> ; ... '
           |  : <Block>          at end or before alts
           |  <Alts>             at end

<Block>  ::=  <Group> ; ...

<Alts>   ::=  | <Block> | ...
```

```
'add1(x + 2)
 sub1(w)'
```

Shrubbery Grammar

```
<Group>  ::=  <Term> <Term> ...

<Term>   ::=  <Int>                 syntax_is_integer
           |  <Boolean>             syntax_is_boolean
           |  <String>              syntax_is_string
           |  <Symbol>              syntax_is_symbol
           |  <Operator>            syntax_is_operator
           |  ( <Group> , ... )     syntax_is_parens
           |  [ <Group> , ... ]     syntax_is_list
           |  { <Group> , ... }     syntax_is_brackets
           |  ' <Group> ; ... '     syntax_is_quotes
           |  : <Block>             syntax_is_block
           |  <Alts>                syntax_is_alts

<Block>  ::=  <Group> ; ...

<Alts>   ::=  | <Block> | ...
```

Shrubbery Grammar

```
<Group>  ::=  <Term> <Term> ...   syntax_group_to_list

<Term>   ::=  <Int>               syntax_to_integer
           |  <Boolean>           syntax_to_boolean
           |  <String>            syntax_to_string
           |  <Symbol>            syntax_to_symbol
           |  <Operator>          syntax_operator_to_symbol
           |  ( <Group> , ... )   syntax_parens_to_list
           |  [ <Group> , ... ]   syntax_to_list
           |  { <Group> , ... }   syntax_brackets_to_list
           |  ' <Group> ; ... '   syntax_quotes_to_list
           |  : <Block>           syntax_block_to_list
           |  <Alts>              syntax_alts_to_list

<Block>  ::=  <Group> ; ...

<Alts>   ::=  | <Block> | ...
```

Shrubbery Grammar

```
<Group>  ::=  <Term> <Term> ...   list_to_group_syntax

<Term>   ::=  <Int>               integer_to_syntax
           |  <Boolean>           boolean_to_syntax
           |  <String>            string_to_syntax
           |  <Symbol>            symbol_to_syntax
           |  <Operator>          symbol_to_operator_syntax
           |  ( <Group> , ... )   list_to_parens_syntax
           |  [ <Group> , ... ]   list_to_syntax
           |  { <Group> , ... }   list_to_brackets_syntax
           |  ' <Group> ; ... '   list_to_quotes_syntax
           |  : <Block>           list_to_block_syntax
           |  <Alts>              list_to_alts_syntax

<Block>  ::=  <Group> ; ...

<Alts>   ::=  | <Block> | ...
```

Parsing without Patterns

```
fun parse(s :: Syntax):
  cond
  | syntax_is_integer(s):
      intE(syntax_to_integer(s))
  | syntax_is_symbol(s):
      idE(syntax_to_symbol(s))
  | syntax_starts(s, 'let'):
      def name = syntax_before(syntax_after(s, 'let'), '=')
      def rhs = syntax_except_last(syntax_after(s, '='))
      def body = syntax_block_inside(syntax_last(s))
      appE(funE(syntax_to_symbol(name),
                parse(body)),
           parse(rhs))
  | ....
```

See **lambda_matchless.rhm**

Parsing without Patterns

```
fun syntax_starts(s, term):
  match syntax_group_to_list(s)
  | []: #false
  | cons(f, r): f == term

fun syntax_contains(s, term):
  member(term, syntax_group_to_list(s))

fun syntax_block_inside(s):
  cond
  | syntax_is_block(s):
      def lst = syntax_block_to_list(s)
      if length(lst) == 1
      | first(lst)
      | error(#'parse, "expected one group in block")
  | ~else: error(#'parse, "expected block")

....
```

Part 4

Moe with Patterns and Templates

```
<Exp>  ::=  <Int>
         |  <Exp> + <Exp>
         |  <Exp> * <Exp>
         |  <Symbol>
         |  fun (<Symbol>): <Exp>
         |  <Exp>(<Exp>)
         |  if <Exp> == 0 | <Exp> | <Exp>
         |  match <Exp>                          NEW
            | '<Shrub>': <Exp>
            | ~else: <Exp>
         |  '<Shrub>'                            NEW
```

Moe with Patterns and Templates

```
<Exp>  ::=  <Int>
         |  <Exp> + <Exp>
         |  <Exp> * <Exp>
         |  <Symbol>
         |  fun (<Symbol>): <Exp>
         |  <Exp>(<Exp>)
         |  if <Exp> ==
         |  match <Exp>
            | '<Shrub>': <Exp>
            | ~else: <Exp>
         |  '<Shrub>'
```

should produce a syntax value

NEW

NEW

Moe with Patterns and Templates

```
<Exp>  ::=  <Int>
         |  <Exp> + <Exp>
         |  <Exp> * <Exp>
         |  <Symbol>
         |  fun (<Symbol>): <Exp>
         |  <Exp>(<Exp>)
         |  if <Exp> == 0 | <Exp> | <Exp>
         |  match <Exp>
              '<Shrub>': <Exp>
            | ~else: <Exp>
         |  '<Shrub>'
```

NEW

NEW

pattern to match against

Moe with Patterns and Templates

```
<Exp>  ::=  <Int>
         |  <Exp> + <Exp>
         |  <Exp> * <Exp>
         |  <Symbol>
         |  fun (<Symbol>): <Exp>
         |  <Exp>(<Exp>)
         |  if <Exp> == 0 | <Exp> | <Exp>
         |  match <Exp>                          NEW
            | '<Shrub>': <Exp>
            | ~else: <Exp>
            '<Shrub>'                            NEW
```

template to produce a syntax object

Syntax in Moe Expressions and Values

```
type Exp
....
| matchE(tst :: Exp,
          pat :: Syntax,
          thn :: Exp,
          els :: Exp)
| quoteE(tmpl :: Syntax)


type Value
....
| syntaxV(stx :: Syntax)
```

Pattern Matching

To implement the `match` form via `matchE`

```
                                                    ?
def group_match :: (Syntax, Syntax) -> Boolean:
  fun (pat, arg):
    ....
```

Pattern Matching

To implement the `match` form via `matchE`

```
                                                          ?
def group_match :: (Syntax, Syntax) -> Boolean:
  fun (pat, arg):
    ....

check: group_match('1 + 2', '1 + 2')
       ~is #true
```

Pattern Matching

To implement the `match` form via `matchE`

```
                                                          ?
def group_match :: (Syntax, Syntax) -> Boolean:
  fun (pat, arg):
    ....

check: group_match('1 + 2', '1 * 2')
       ~is #false
```

Pattern Matching

To implement the `match` form via `matchE`

```
                                                          ?
def group_match :: (Syntax, Syntax) -> Boolean:
  fun (pat, arg):
    ....

check: group_match('%x + 2', '1 + 2')
       ~is #true
```

Pattern Matching

To implement the `match` form via `matchE`

```
                                                    ?
def group_match :: (Syntax, Syntax) -> Env:
  fun (pat, arg):
    ....

check: group_match('%x + 2', '1 + 2')
       ~is [bind(#'x, syntaxV('1'))]
```

Pattern Matching

To implement the `match` form via `matchE`

```
                                                        ?
def group_match :: (Syntax, Syntax) -> Env:
  fun (pat, arg):
    ....

check: group_match('%x + %y', '1 + 2')
       ~is [bind(#'x, syntaxV('1')),
            bind(#'y, syntaxV('2'))]
```

Pattern Matching

To implement the `match` form via `matchE`

```
                                                        ?
def group_match :: (Syntax, Syntax) -> Env:
  fun (pat, arg):
    ....

check: group_match('%x + %y', '3 * 4 + 2')
       ~is [bind(#'x, syntaxV('3 * 4')),
            bind(#'y, syntaxV('2'))]
```

Pattern Matching

To implement the `match` form via `matchE`

```
                                                    ?
def group_match :: (Syntax, Syntax) -> Env:
  fun (pat, arg):
    ....

check: group_match('1 + 2', '1 * 2')
       ~is #false
```

Pattern Matching

To implement the `match` form via `matchE`

```
def group_match :: (Syntax, Syntax) -> Optionof(Env):
  fun (pat, arg):
    ....

check: group_match('1 + 2', '1 * 2')
       ~is none()
```

Pattern Matching

To implement the `match` form via `matchE`

```
def group_match :: (Syntax, Syntax) -> Optionof(Env):
  fun (pat, arg):
    ....

check: group_match('1 + 2', '1 + 2')
       ~is some(mt_env)
```

Pattern Matching

To implement the `match` form via `matchE`

```
def group_match :: (Syntax, Syntax) -> Optionof(Env):
  fun (pat, arg):
    ....

check: group_match('%x + 2', '1 + 2')
       ~is some([bind(#'x, syntaxV('1'))])
```

Template Substitution

To implement `'<Shrub>'` results via **`quoteE`**

```
def group_subst :: (Syntax, Env) -> Syntax:
  fun (tmpl, env):
    ....
```

Template Substitution

To implement `'<Shrub>'` results via **`quoteE`**

```
def group_subst :: (Syntax, Env) -> Syntax:
  fun (tmpl, env):
    ....

check: group_subst('1 + 2', mt_env)
       ~is '1 + 2'
```

Template Substitution

To implement `'<Shrub>'` results via **`quoteE`**

```
def group_subst :: (Syntax, Env) -> Syntax:
  fun (tmpl, env):
    ....

check: group_subst('%x + %y',
                   [bind(#'x, syntaxV('3 * 4')),
                    bind(#'y, syntaxV('2'))])
       ~is '3 * 4 + 2'
```

Part 5

Implementing Pattern Matching

```
def group_match :: (Syntax, Syntax) -> Optionof(Env):
  fun (pat, arg):
    ....

check: group_match('%x + 2', '1 + 2')
       ~is some([bind(#'x, syntaxV('1'))])
```

Implementing Pattern Matching

```
fun group_match(pat :: Syntax, arg :: Syntax) :: Optionof(Env):
  term_list_match(syntax_group_to_list(pat),
                  syntax_group_to_list(arg))

fun term_list_match(pat :: Listof(Syntax),
                    arg :: Listof(Syntax)) :: Optionof(Env):
  cond
  | ....
```

Implementing Pattern Matching

```
fun term_list_match(pat :: Listof(Syntax),
                    arg :: Listof(Syntax)) :: Optionof(Env):
  cond
  | ....
```

Implementing Pattern Matching

```
fun term_list_match(pat :: Listof(Syntax),
                    arg :: Listof(Syntax)) :: Optionof(Env):
  cond
  | pat == []:
      if arg == []
      | some(mt_env)
      | none()
  | ....
```

Implementing Pattern Matching

```
fun term_list_match(pat :: Listof(Syntax),
                    arg :: Listof(Syntax)) :: Optionof(Env):
  cond
  | ....
  | starts_escape(pat):
      ....
  | ....
```

Implementing Pattern Matching

```
fun term_list_match(pat :: Listof(Syntax),
                    arg :: Listof(Syntax)) :: Optionof(Env):
  cond
  | ....
  | ....
  | ~else:
      // match first of `pat` to first of `arg`
      ....
```

Implementing Pattern Matching

```
fun term_list_match(pat :: Listof(Syntax),
                    arg :: Listof(Syntax)) :: Optionof(Env):
  cond
  | ....
  | ....
  | ~else:
      if arg == []
      | none()
      |
```

Implementing Pattern Matching

```
fun term_list_match(pat :: Listof(Syntax),
                    arg :: Listof(Syntax)) :: Optionof(Env):
  cond
  | ....
  | ....
  | ~else:
      if arg == []
      | none()
      |       term_match(first(pat), first(arg))
```

Implementing Pattern Matching

```
fun term_list_match(pat :: Listof(Syntax),
                    arg :: Listof(Syntax)) :: Optionof(Env):
  cond
  | ....
  | ....
  | ~else:
      if arg == []
      | none()
      | match term_match(first(pat), first(arg))
        | some(f_env):




        | none():
            none()
```

Implementing Pattern Matching

```
fun term_list_match(pat :: Listof(Syntax),
                    arg :: Listof(Syntax)) :: Optionof(Env):
  cond
  | ....
  | ....
  | ~else:
      if arg == []
      | none()
      | match term_match(first(pat), first(arg))
        | some(f_env):
                  term_list_match(rest(pat), rest(arg))



        | none():
            none()
```

Implementing Pattern Matching

```
fun term_list_match(pat :: Listof(Syntax),
                    arg :: Listof(Syntax)) :: Optionof(Env):
  cond
  | ....
  | ....
  | ~else:
      if arg == []
      | none()
      | match term_match(first(pat), first(arg))
        | some(f_env):
            match term_list_match(rest(pat), rest(arg))
            | some(r_env):
                some(append(f_env, r_env))
            | none():
                none()
        | none():
            none()
```

Implementing Pattern Matching

```
fun term_list_match(pat :: Listof(Syntax),
                    arg :: Listof(Syntax)) :: Optionof(Env):
  cond
  | ....
  | starts_escape(pat):
      ....
  | ....
```

Implementing Pattern Matching

```
fun term_list_match(pat :: Listof(Syntax),
                    arg :: Listof(Syntax)) :: Optionof(Env):
  cond
  | ....
  | starts_escape(pat):
      def name = syntax_to_symbol(first(rest(pat)))
      def pat_rest = rest(rest(pat))
      // consume 1 or more of `arg` as match to `name`
      ....
  | ....
```

Implementing Pattern Matching

```
fun term_list_match(pat :: Listof(Syntax),
                    arg :: Listof(Syntax)) :: Optionof(Env):
  cond
  | ....
  | starts_escape(pat):
      ....
      fun match_pat_first(arg, accum):
        ....
      ....
      match_pat_first(arg, [])
```

Implementing Pattern Matching

```
fun term_list_match(pat :: Listof(Syntax),
                    arg :: Listof(Syntax)) :: Optionof(Env):
  cond
  | ....
  | starts_escape(pat):
      ....
      fun match_pat_first(arg, accum):
        match arg
        | []:
            match_pat_rest([], accum)
        | ....
      ....
      match_pat_first(arg, [])
```

Implementing Pattern Matching

```
fun term_list_match(pat :: Listof(Syntax),
                    arg :: Listof(Syntax)) :: Optionof(Env):
  cond
  | ....
  | starts_escape(pat):
      ....
      fun match_pat_rest(arg, accum):
        match accum
        | []: none()
        | ~else:
            ...
      ....
```

Implementing Pattern Matching

```
fun term_list_match(pat :: Listof(Syntax),
                    arg :: Listof(Syntax)) :: Optionof(Env):
  cond
  | ....
  | starts_escape(pat):
      ....
      fun match_pat_rest(arg, accum):
        match accum
        | []: none()
        | ~else:
            match term_list_match(pat_rest, arg)
            | some(r_env):
                combine_results(accum, r_env)
            | none(): none()
      ....
```

Implementing Pattern Matching

```
fun term_list_match(pat :: Listof(Syntax),
                    arg :: Listof(Syntax)) :: Optionof(Env):
  cond
  | ....
  | starts_escape(pat):
      ....
      fun combine_results(accum, r_env):
        def g = list_to_group_syntax(reverse(accum))
        some(extend_env(bind(name, syntaxV(g)),
                        r_env))
      ....
```

Implementing Pattern Matching

```
fun term_list_match(pat :: Listof(Syntax),
                    arg :: Listof(Syntax)) :: Optionof(Env):
  cond
  | ....
  | starts_escape(pat):
      ....
      fun match_pat_first(arg, accum):
        match arg
        | ....
        | cons(arg_f, arg_r):
            ....
      ....
```

Implementing Pattern Matching

```
fun term_list_match(pat :: Listof(Syntax),
                    arg :: Listof(Syntax)) :: Optionof(Env):
  cond
  | ....
  | starts_escape(pat):
      ....
      fun match_pat_first(arg, accum):
        match arg
        | ....
        | cons(arg_f, arg_r):
            // try consuming `arg_f`
            match match_pat_first(arg_r, cons(arg_f, accum))
            | some(env): some(env)
            | none(): ....
      ....
```

Implementing Pattern Matching

```
fun term_list_match(pat :: Listof(Syntax),
                    arg :: Listof(Syntax)) :: Optionof(Env):
  cond
  | ....
  | starts_escape(pat):
      ....
      fun match_pat_first(arg, accum):
        match arg
        | ....
        | cons(arg_f, arg_r):
            // try consuming `arg_f`
            match match_pat_first(arg_r, cons(arg_f, accum))
            | some(env): some(env)
            | none():
                // didn't work; try not consuming `arg_f`
                match_pat_rest(arg, accum)
      ....
```

Implementing Pattern Matching

```
fun term_match(pat :: Syntax, arg :: Syntax) :: Optionof(Env):
  cond
  | syntax_is_list(pat) && syntax_is_list(arg):
      group_list_match(syntax_to_list(pat),
                       syntax_to_list(arg))
  | syntax_is_parens(pat) && syntax_is_parens(arg):
      group_list_match(syntax_parens_to_list(pat),
                       syntax_parens_to_list(arg))
  | ...
  | ~else:
      // numbers, symbols, booleans, etc.
      if pat == arg
      | some(mt_env)
      | none()
```

Implementing Pattern Matching

```
fun interp(a :: Exp, env :: Env) :: Value:
  match a
  | ....
  | matchE(tst, pat, thn, els):
      match interp(tst, env)
      | syntaxV(arg):
          match group_match(pat, arg)
          | some(m_env): interp(thn, append(m_env, env))
          | none(): interp(els, env)
      | ~else: error(#'interp, "not a syntax object")
  | ....
```

Part 6

Implementing Template Substitution

```
def group_subst :: (Syntax, Env) -> Syntax:
  fun (tmpl, env):
    ....

check: group_subst('%x + %y',
                   [bind(#'x, syntaxV('3 * 4')),
                    bind(#'y, syntaxV('2'))])
       ~is '3 * 4 + 2'
```

Implementing Template Substitution

```
fun term_list_subst(tmpl :: Listof(Syntax), env :: Env):
  cond
  | tmpl == []: []
  | starts_escape(tmpl):
      ....
  | ~else:
      ....
```

Implementing Template Substitution

```
fun term_list_subst(tmpl :: Listof(Syntax), env :: Env):
  cond
  | tmpl == []: []
  | starts_escape(tmpl):
      ....
  | ~else:
      cons(term_subst(first(tmpl), env),
           term_list_subst(rest(tmpl), env))
```

Implementing Template Substitution

```
fun term_list_subst(tmpl :: Listof(Syntax), env :: Env):
  cond
  | tmpl == []: []
  | starts_escape(tmpl):
      def name = syntax_to_symbol(first(rest(tmpl)))
      ....
  | ~else:
      cons(term_subst(first(tmpl), env),
           term_list_subst(rest(tmpl), env))
```

Implementing Template Substitution

```
fun term_list_subst(tmpl :: Listof(Syntax), env :: Env):
  cond
  | tmpl == []: []
  | starts_escape(tmpl):
      def name = syntax_to_symbol(first(rest(tmpl)))
      match lookup(name, env)
      | syntaxV(s):
          ....
      | ~else:
          error(#'interp, "not a syntax object")
  | ~else:
      cons(term_subst(first(tmpl), env),
           term_list_subst(rest(tmpl), env))
```

Implementing Template Substitution

```
fun term_list_subst(tmpl :: Listof(Syntax), env :: Env):
  cond
  | tmpl == []: []
  | starts_escape(tmpl):
      def name = syntax_to_symbol(first(rest(tmpl)))
      match lookup(name, env)
      | syntaxV(s):
          append(syntax_group_to_list(s),
                 term_list_subst(rest(rest(tmpl)), env))
      | ~else:
          error(#'interp, "not a syntax object")
  | ~else:
      cons(term_subst(first(tmpl), env),
           term_list_subst(rest(tmpl), env))
```

Implementing Template Substitution

```
fun interp(a :: Exp, env :: Env) :: Value:
  match a
  | ....
  | quoteE(tmpl):
      syntaxV(group_subst(tmpl, env))
  | ....
```

Part 7

Expansion

```
delay:
  <Exp>
```

```
delay:
  1 + 2
```

⇒

```
fun (dummy):
  1 + 2
```

`delay` Expander

```
let delay_m = (fun (s):
                 match s
                 | 'delay: $expr':
                     'fun (dummy): $expr'
                 | ~else:
                     error(#'delay, "syntax")):
  check: delay_m('delay: 1 + 2')
         ~is 'fun (dummy): 1 + 2'
```

`delay` Expander

```
let delay_m = (fun (s):
                  match s
                  | 'delay: $expr':
                      'fun (dummy): $expr'
                  | ~else:
                      error(#'delay, "syntax")):
  delay_m('delay: 1 + 2')
```

`delay` Expander

```
let delay_m = (fun (s):
                  match s
                  | 'delay: %expr':
                      'fun (dummy): %expr'
                  | ~else:
                      'bad_delay'):
  delay_m('delay: 1 + 2')
```

`delay` Expander

```
let_macro delay = (fun (s):
                     match s
                     | 'delay: %expr':
                         'fun (dummy): %expr'
                     | ~else:
                         'bad_delay'):
  delay: 1 + 2
```

Recursive Expansion

```
cond
| <Exp> == <Int>: <Exp>
| ...
| ~else: <Exp>
```

```
cond
| x == 3: 'three'
| x == 4: 'four'
| ~else: 'no'
```

⇒

```
if x + -1 * 3 == 0
| 'three'
| if x + -1 * 4 == 0
  | 'four'
  | 'no'
```

Recursive Expansion

```
cond
| <Exp> == <Int>: <Exp>
| ...
| ~else: <Exp>
```

```
cond
| x == 3: 'three'
| x == 4: 'four'
| ~else: 'no'
```

⇒

```
if x + -1 * 3 == 0
| 'three'
| cond
  | x == 4: 'four'
  | ~else: 'no'
```

Recursive Expansion

```
cond
| <Exp> == <Int>: <Exp>
| ...
| ~else: <Exp>
```

```
cond
| x == 3: 'three'
| x == 4: 'four'
| ~else: 'no'
```

⇒

```
if x + -1 * 3 == 0
| 'three'
| cond
  | x == 4: 'four'
  | ~else: 'no'
```

Recursive Expansion

```
cond
| <Exp> == <Int>: <Exp>
| ...
| ~else: <Exp>
```

```
cond
| ~else: 'no'
```

⇒ `'no'`

`cond` Expander

```
let cond_m = (fun (s):
                match s
                | 'cond | ~else: $expr':
                    expr
                | 'cond | $x == $n: $thn | $more | ...':
                    'if $x + -1 * $n == 0
                     | $thn
                     | cond
                       | $more | ...'
                | ~else: error(#'cond, "syntax")):
  check: cond_m('« cond
                   | ~else: 'no' »')
         ~is '« 'no' »'
```

cond Expander

```
let cond_m = (fun (s):
                match s
                | 'cond | ~else: $expr':
                    expr
                | 'cond | $x == $n: $thn | $more | ...':
                    'if $x + -1 * $n == 0
                     | $thn
                     | cond
                       | $more | ...'
                | ~else: error(#'cond, "syntax")):
  check: cond_m('« cond
                   | x == 3: 'three'
                   | x == 4: 'four'
                   | ~else: 'no' »')
         ~is '« if x + -1 * 3 == 0
                | 'three'
                | cond
                  | x == 4: 'four'
                  | ~else: 'no' »'
```

`cond` Expander

```
let cond_m = (fun (s):
                match s
                | 'cond | ~else: $expr':
                    expr
                | 'cond | $x == $n: $thn | $more | ...':
                    'if $x + -1 * $n == 0
                     | $thn
                     | cond
                       | $more | ...'
                | ~else: error(#'cond, "syntax")):
  cond_m('« cond
            | x == 3: 'three'
            | x == 4: 'four'
            | ~else: 'no' »')
```

cond Expander

```
let cond_m = (fun (s):
                match s
                | 'cond | ~else: %expr':
                    expr
                | ~else:
                    match s
                    | 'cond | %x == %n: %thn | %more | ...':
                        'if %x + -1 * %n == 0
                         | %thn
                         | (cond | %more | ...)'
                    | ~else: 'bad_match'):
  cond_m('« cond
            | x == 3: 'three'
            | x == 4: 'four'
            | ~else: 'no' »')
```

cond Expander

```
let cond_m = (fun (s):
                match s
                | 'cond | ~else: %expr':
                    expr
                | ~else:
                    match s
                    | 'cond | %x == n: %thn | %more':
                        'if %x + -1 * %n == 0
                         | %thn
                         | cond | %more'
                    | ~else:
                        match s
                        | 'cond | %x == %n: %thn | %two | %three':
                            'if %x + -1 * %n == 0
                             | %thn
                             | (cond | %two | %three)'
                        | ~else: 'bad_match'):
  cond_m('« cond
            | x == 3: 'three'
            | x == 4: 'four'
            | ~else: 'no' »')
```

cond Expander

```
let_macro cond = (fun (s):
                    match s
                    | 'cond | ~else: %expr':
                        expr
                    | ~else:
                        match s
                        | 'cond | %x == n: %thn | %more':
                            'if %x + -1 * %n == 0
                             | %thn
                             | cond | %more'
                        | ~else:
                            match s
                            | 'cond | %x == %n: %thn | %two | %three':
                                'if %x + -1 * %n == 0
                                 | %thn
                                 | (cond | %two | %three)'
                            | ~else: 'bad_match'):
  cond
  | x == 3: 'three'
  | x == 4: 'four'
  | ~else: 'no'
```

Moe with Macros

```
<Exp>  ::=  <Int>
         |  <Exp> + <Exp>
         |  <Exp> * <Exp>
         |  <Symbol>
         |  fun (<Symbol>): <Exp>
         |  <Exp>(<Exp>)
         |  if <Exp> == 0 | <Exp> | <Exp>
         |  match <Exp>
            | '<Shrub>': <Exp>
            | ~else: <Exp>
         |  '<Shrub>'
         |  let_macro <Symbol> = <Exp>:
              <Exp>
```

Part 8

Adding `let_macro`

```
fun parse(s :: Syntax) :: Exp:
  cond
  | ....
  | ~else:
      match s
       | 'let_macro $name = $rhs: $body':




       | ....
```

```
let_macro delay = (fun (s):
                     match s
                     | 'delay: %expr':
                         'fun (dummy): %expr'
                     | ~else:
                         'bad_delay'):
  delay: 1 + 2
```

Adding `let_macro`

```
fun parse(s :: Syntax) :: Exp:
  cond
  | ....
  | ~else:
      match s
      | 'let_macro $name = $rhs: $body':
          parse(body)



      | ....
```

```
let_macro delay = (fun (s):
                     match s
                     | 'delay: %expr':
                         'fun (dummy): %expr'
                     | ~else:
                         'bad_delay'):
  delay: 1 + 2
```

Adding `let_macro`

```
fun parse(s :: Syntax) :: Exp:
  cond
  | ....
  | ~else:
      match s
      | 'let_macro $name = $rhs: $body':
          parse(body)
                                          name



      | ....
```

```
let_macro delay = (fun (s):
                     match s
                     | 'delay: %expr':
                         'fun (dummy): %expr'
                     | ~else:
                         'bad_delay'):
  delay: 1 + 2
```

Adding `let_macro`

```
fun parse_in_env(s :: Syntax, env :: Env) :: Exp:
  cond
  | ....
  | ~else:
      match s
      | 'let_macro $name = $rhs: $body':
          parse_in_env(body,
                       extend_env(bind(name,
                                       ),
                                  env))
      | ....
```

```
let_macro delay = (fun (s):
                     match s
                     | 'delay: %expr':
                         'fun (dummy): %expr'
                     | ~else:
                         'bad_delay'):
  delay: 1 + 2
```

Adding `let_macro`

```
fun parse_in_env(s :: Syntax, env :: Env) :: Exp:
  cond
  | ....
  | ~else:
      match s
      | 'let_macro $name = $rhs: $body':
          parse_in_env(body,
                       extend_env(bind(name,
                                       parse(rhs)
                                       ),
                                  env))
      | ....
```

```
let_macro delay = (fun (s):
                     match s
                     | 'delay: %expr':
                         'fun (dummy): %expr'
                     | ~else:
                         'bad_delay'):
  delay: 1 + 2
```

Adding `let_macro`

```
fun parse_in_env(s :: Syntax, env :: Env) :: Exp:
  cond
  | ....
  | ~else:
      match s
      | 'let_macro $name = $rhs: $body':
          parse_in_env(body,
                       extend_env(bind(name,
                                       interp(parse(rhs),
                                              mt_env)),
                                  env))
      | ....
```

```
let_macro delay = (fun (s):
                     match s
                     | 'delay: %expr':
                         'fun (dummy): %expr'
                     | ~else:
                         'bad_delay'):
  delay: 1 + 2
```

Adding `let_macro`

```
fun parse_in_env(s :: Syntax, env :: Env) :: Exp:
  cond
  | ....
  | ~else:
      match s
      | 'let_macro $name = $rhs: $body':
          parse_in_env(body,
                       extend_env(bind(name,
                                       interp(parse(rhs),
                                              mt_env)),
                                  env))
      | ....
```

`parse` calls `interp`!

```
let_macro delay = (fun (s):
                     match s
                     | 'delay: %expr':
                         'fun (dummy): %expr'
                     | ~else:
                         'bad_delay'):
  delay: 1 + 2
```

Adding `let_macro`

```
fun parse_in_env(s :: Syntax, env :: Env) :: Exp:
  cond
  | ....
  | ~else:
      match s
      | 'let_macro $name = $rhs: $body':
          parse_in_env(body,
                       extend_env(bind(name,
                                       interp(parse(rhs),
                                              mt_env)),
                                  env))
      | ....
```

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

```
let_macro delay = (fun (s):
                     match s
                     | 'delay: %expr':
                         'fun (dummy): %expr'
                     | ~else:
                         'bad_delay'):
  delay: 1 + 2
```

Adding `let_macro`

```
fun parse_in_env(s :: Syntax, env :: Env) :: Exp:
  cond
  | starts_with_bound_identifier(s, env):

  | ....
```

```
delay: 1 + 2
```

Adding `let_macro`

```
fun parse_in_env(s :: Syntax, env :: Env) :: Exp:
  cond
  | starts_with_bound_identifier(s, env):
      match s
      | '$(id :: Identifier) $arg ...':


  | ....
```

```
delay: 1 + 2
```

Adding `let_macro`

```
fun parse_in_env(s :: Syntax, env :: Env) :: Exp:
  cond
  | starts_with_bound_identifier(s, env):
      match s
      | '$(id :: Identifier) $arg ...':
          def macro = lookup(syntax_to_symbol(id), env)

  | ....
```

```
delay: 1 + 2
```

Adding `let_macro`

```
fun parse_in_env(s :: Syntax, env :: Env) :: Exp:
  cond
  | starts_with_bound_identifier(s, env):
      match s
      | '$(id :: Identifier) $arg ...':
          def macro = lookup(syntax_to_symbol(id), env)
                apply_macro(macro, s)
  | ....
```

```
delay: 1 + 2
```

Adding `let_macro`

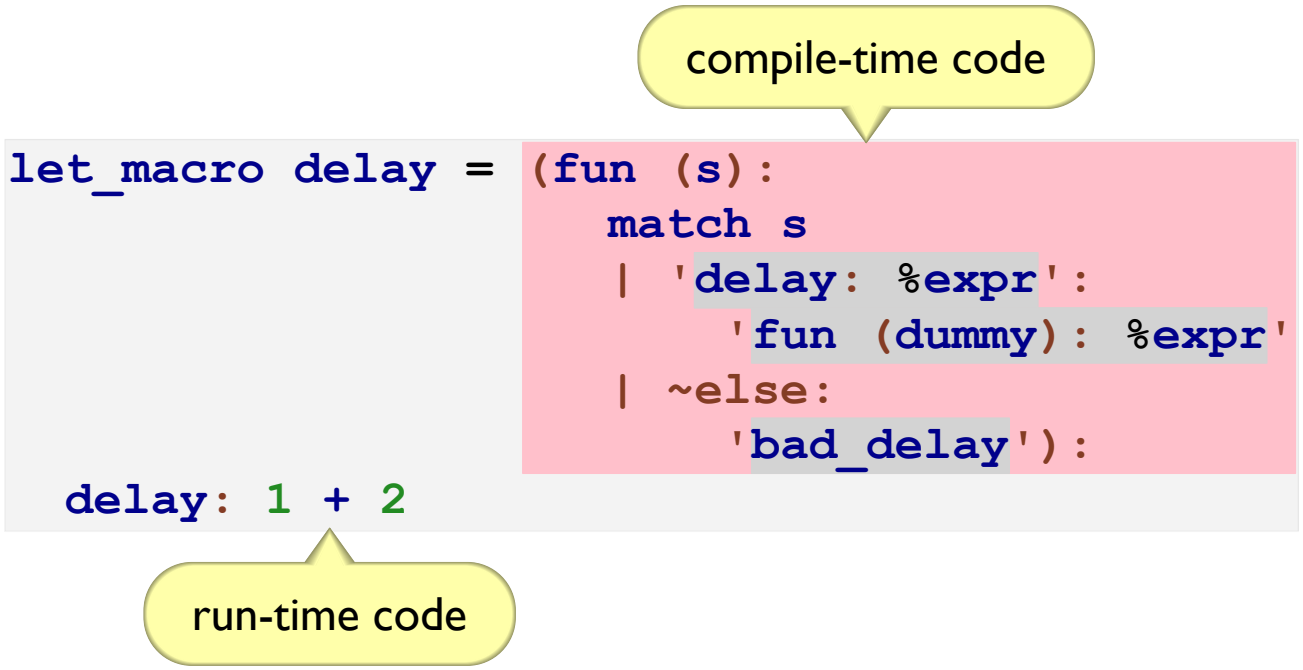
```
fun parse_in_env(s :: Syntax, env :: Env) :: Exp:
  cond
  | starts_with_bound_identifier(s, env):
      match s
      | '$(id :: Identifier) $arg ...':
          def macro = lookup(syntax_to_symbol(id), env)
          parse(apply_macro(macro, s))
  | ....
```

```
delay: 1 + 2
```

Adding `let_macro`

```
fun apply_macro(macro, s):
  match macro
  | closV(arg, body, c_env):
      match interp(body,
                   extend_env(bind(arg, syntaxV(s)),
                              c_env))
      | syntaxV(s): s
      | ~else:
          error(#'parse, "macro result is not syntax")
  | ~else:
      error(#'parse, "let_macro bound a non-function")
```

Compile Time vs. Run Time



Example Use of `let_macro`

```
parse('let_macro delay = (fun (s):
                            match s
                            | 'delay: %expr':
                                'fun (dummy): %expr'
                            | ~else:
                                'bad_delay'):
         let force = (fun (d): d(0)):
           force(delay: 1 + 2)')
```

Part 9

Accidental Capture

```
parse('let_macro delay = (fun (s):
                            match s
                            | 'delay: %expr':
                                'fun (dummy): %expr'
                            | ~else:
                                'bad_delay'):
         let force = (fun (d): d(0)):
           let dummy = 8:
             force(delay: 1 + dummy)')
// ⇒ 1
```

Moe macros must be careful to invent names for new variables

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

Hygienic Macros

A ***hygienic macro system*** avoids capture automatically

Shplait macros are hygienic

```
macro 'delay: $body':
  'fun (dummy): $body'

let dummy = 8:
  let d = (delay: 1 + dummy):
    d(0)
// ⇒ 9
```