Core forms:

Const:

Constants (type `expr`, type-constructor `Const`)

- Type: `Const of sexpr`
- The type-constructor `Const` is used both for self-evaluating and non-self-evaluating constants

A self-evaluating constant is one you can type at the Scheme prompt and see it printed back at you: Numbers, chars, Booleans, strings

$$[\![\langle \textit{self-evaluating sexpr}\rangle]\!] \quad = \quad \texttt{Const}(\langle \textit{self-evaluating sexpr}\rangle)$$

Example:

```
let tag_parse = function
  ...
  | Number(x) -> Const(Sexpr(Number(x)))
  ... ;;
```

Qoute:

$$\llbracket\texttt{(quote }\langle sexpr\rangle\texttt{)}\rrbracket = \llbracket\texttt{Pair(Symbol("quote"), Pair(}\langle sexpr\rangle\texttt{, Nil))}\rrbracket = \texttt{Const(}\langle sexpr\rangle\texttt{)}$$

⚠ A tricky example:

$$\llbracket\texttt{(quote (quote }\langle sexpr\rangle\texttt{))}\rrbracket$$
$$= \llbracket\texttt{Pair(Symbol("quote"), Pair(Pair(Symbol("quote"), Pair(}\langle sexpr\rangle\texttt{, Nil)), Nil))}\rrbracket$$
$$= \texttt{Const(Pair(Symbol("quote"), Pair(}\langle sexpr\rangle\texttt{, Nil)))}$$

Example:

```
let tag_parse = function
  ...
  | Pair(Symbol("quote"), Pair(x, Nil)) -> Const(Sexpr(x))evaluate
```

Variable:

Variables (type expr, type-constructor Var)

- Type: `Var of string`
- Variables are literal symbols that are not reserved words
  - The latest version of Scheme (R^{6}RS) does not have many reserved words
  - Not having reserved words makes the parser more complex
  - We're going to ignore this, and assume that words that are used for syntax are reserved words. These include:
    - `and`, `begin`, `cond`, `define`, `else`, `if`, `lambda`, `let`, `let*` `letrec`, `or`, `quasiquote`, `quote`, `set!` `unquote`, `unquote-splicing`
    - There are additional reserved words, but we'll ignore those

If:

Conditionals (type `expr`, type-constructor `If`)

- Type: `If of expr * expr * expr`
- Scheme supports if-then variant without an `else`-clause
  - These are used when the `then`-clause contains side-effects
  - The "missing"/implicit `else`-clause is defined to be `Const(Void)`
  - We shall support the if-then variant, and tacitly add the implicit `else`-clause
- This is your first recursive case of the `expr` datatype: An `expr` that contains sub-`expr`s.
  - Obviously, the tag-parser will have to be recursive!

Example:

```
let tag_parse = function
  ...
  |Pair(Symbol("if"), Pair(test, Pair(dit, Pair(dif, Nil)))) ->
    If(tag_parse test, tag_parse dif, tag_parse dif)
```

Lambda:

Lambdas (type `expr`, type-constructor `LambdaSimple`, `LambdaOpt`)

- Types:
  - `LambdaSimple of string list * expr`
  - `LambdaOpt of string list * string * expr`
- Scheme has three `lambda`-forms, and we're going to represent these three forms using the two AST nodes `LambdaSimple` & `LambdaOpt`.

Lambdas (type expr, type-constructor LambdaSimple, LambdaOpt)

- The general form of lambda-expressions is
  (lambda $\langle arglist \rangle$. $(\langle expr \rangle^{+})$):
  1. If $\langle arglist \rangle$ is a proper list of unique variable names, then the lambda-expression is said to be simple, and we represent it using the AST node LambdaSimple

Lambda simple:

Lambdas (type expr, type-constructor LambdaSimple, LambdaOpt)

- The general form of lambda-expressions is
  (lambda $\langle arglist \rangle$. $(\langle expr \rangle^{+})$):
  1. If $\langle arglist \rangle$ is a proper list of unique variable names, then the lambda-expression is said to be simple, and we represent it using the AST node LambdaSimple

example:

(lambda (x y) . (x))

Lambda opt:

Lambdas (type expr, type-constructor LambdaSimple, LambdaOpt)

- The general form of lambda-expressions is
  (lambda $\langle arglist \rangle$. $(\langle expr \rangle^{+})$):
  2. If $\langle arglist \rangle$ is the improper list $(v_1 \cdots v_n \,.\, vs)$, then the lambda-expression is said to take at least n arguments:

Example:

(lambda (x y . z) . (x))

Lambda variadic: (translated to lambda opt)

Lambdas (type expr, type-constructor LambdaSimple, LambdaOpt)

- The general form of lambda-expressions is (lambda ⟨*arglist*⟩ . (⟨*expr*⟩$^+$)):
  - ③ If ⟨*arglist*⟩ is the symbol *vs*, then the lambda-expression is said to be variadic, and may be applied to any number of arguments:

Example:

(lambda x . (x))

Sequences:

Sequences (type expr, type-constructor Seq)

- Type: `Seq of expr list`
- There are two types of sequences:
  - Explicit sequences (begin-expressions)
  - Implicit sequences
    - Body of lambda
    - In the Ribs of cond
    - In the body of let, let*, letrec
    - Other syntactic forms we shall not support
- Both implicit & explicit sequences are encoded as single expressions using the type-constructor Seq

Set:

Assignments (type expr, type-constructor Set)

- Type: `Set of expr * expr`
- The AST node for set! (pronounced "set-bang") expressions

Define:

Definitions (type `expr`, type-constructor `Def`)

- Type: `Def of expr * expr`
- The AST node for `define`-expressions
- Two syntaxes for `define`:
  - `(define` $\langle var \rangle$ $\langle expr \rangle$`)`
    - Example:
      `(define pi (* 4 (atan 1)))`

Or:

Disjunctions (type `expr`, type-constructor `Or`)

- Type: `Or of expr list`
- $[\![$`(or)`$]\!] = [\![$`#f`$]\!]$ (by definition)
- $[\![$`(or` $\langle expr \rangle$`)`$]\!] = [\![\langle expr \rangle]\!]$ (`#f` is the unit element of `or`)
- The real work is done here:

$$
\begin{aligned}
&[\![\texttt{(or } \langle expr_1 \rangle \ \cdots \ \langle expr_n \rangle\texttt{)}]\!] \\
&\quad = \texttt{Or([}[\![\langle expr_1 \rangle]\!]\texttt{; } \cdots \texttt{; } [\![\langle expr_n \rangle]\!]\texttt{])}
\end{aligned}
$$

Application:

Applications (type `expr`, type-constructor `Applic`)

- Type: `Applic of expr * (expr list)`
- The AST node separates the expression in the procedure position from the list of arguments
- The tag-parser recurses over the procedure & the list of arguments:
  $[\![\texttt{(}\langle expr \rangle \ \langle expr \rangle_1 \cdots \langle expr \rangle_n\texttt{)}]\!] =$
  $\texttt{Applic(}[\![\langle expr \rangle]\!]\texttt{, [}[\![\langle expr \rangle_1]\!]\texttt{; } \cdots \texttt{ ; } [\![\langle expr \rangle_n]\!]\texttt{])}$

Macro Expansions:

Let:

form:

- The syntax looks like this:

$$\begin{array}{l}
\texttt{(let ((v}_1\ \langle Expr_1\rangle\texttt{)} \\
\qquad\quad \cdots \\
\qquad\quad \texttt{(v}_n\ \langle Expr_n\rangle\texttt{))} \\
\quad \langle expr_1\rangle\ \cdots\ \langle expr_m\rangle\texttt{)}
\end{array}$$

Expands to:

Putting it all together, we get the following macro-expansion:

$$\begin{array}{l}
[\![\texttt{(let ((v}_1\ \langle Expr_1\rangle\texttt{)} \\
\qquad\quad \cdots \\
\qquad\quad \texttt{(v}_n\ \langle Expr_n\rangle\texttt{))} \\
\quad \langle expr_1\rangle\ \cdots\ \langle expr_m\rangle\texttt{)}]\!] \\
\quad = \ [\![\texttt{((lambda (v}_1\ \cdots\ \texttt{v}_n\texttt{)} \\
\qquad\qquad \langle expr_1\rangle\ \cdots\ \langle expr_m\rangle\texttt{)} \\
\qquad\quad \langle Expr_1\rangle\ \cdots\ \langle Expr_n\rangle\texttt{)}]\!]
\end{array}$$

Let*:

① This is the first of the two base cases:

$$
\begin{aligned}
&[\![\texttt{(let* ()}\ \langle \mathit{expr}_1\rangle\ \cdots\ \langle \mathit{expr}_m\rangle\texttt{)}]\!]\\
&\quad = [\![\texttt{(let ()}\ \langle \mathit{expr}_1\rangle\ \cdots\ \langle \mathit{expr}_m\rangle\texttt{)}]\!]
\end{aligned}
$$

② This is the second base case:

$$
\begin{aligned}
&[\![\texttt{(let* ((v}\ \langle \mathit{Expr}\rangle\texttt{))}\ \langle \mathit{expr}_1\rangle\ \cdots\ \langle \mathit{expr}_m\rangle\texttt{)}]\!]\\
&\quad = [\![\texttt{(let ((v}\ \mathit{Expr}\texttt{))}\ \langle \mathit{expr}_1\rangle\ \cdots\ \langle \mathit{expr}_m\rangle\texttt{)}]\!]
\end{aligned}
$$

③ This is the inductive case:

$$
\begin{aligned}
&[\![\texttt{(let* ((v}_1\ \langle \mathit{Expr}_1\rangle\texttt{)}\ \texttt{(v}_2\langle \mathit{Expr}_2\rangle\texttt{)}\ \cdots\ \texttt{(v}_n\ \langle \mathit{Expr}_n\rangle\texttt{))}\\
&\quad \langle \mathit{expr}_1\rangle\ \cdots\ \langle \mathit{expr}_m\rangle\texttt{)}]\!]\\
&\quad = [\![\texttt{(let ((v}_1\ \langle \mathit{Expr}_1\rangle\texttt{)))}\\
&\qquad\qquad \texttt{(let* ((v}_2\ \langle \mathit{Expr}_2\rangle\texttt{)}\ \cdots\ \texttt{(v}_n\ \langle \mathit{Expr}_n\rangle\texttt{))}\\
&\qquad\qquad\quad \langle \mathit{expr}_1\rangle\ \cdots\ \langle \mathit{expr}_m\rangle\texttt{))}]\!]
\end{aligned}
$$

Letrec:

$$
\begin{aligned}
&[\![\texttt{(letrec ((f}_1\ \langle \mathit{Expr}_1\rangle\texttt{)} && = \texttt{(let ((f}_1\ \texttt{'whatever)}\\
&\qquad\qquad \texttt{(f}_2\ \langle \mathit{Expr}_2\rangle\texttt{)} && \qquad\quad \texttt{(f}_2\ \texttt{'whatever)}\\
&\qquad\qquad \cdots && \qquad\quad \cdots\\
&\qquad\qquad \texttt{(f}_n\ \langle \mathit{Expr}_n\rangle\texttt{))} && \qquad\quad \texttt{(f}_n\ \texttt{'whatever))}\\
&\quad \langle \mathit{expr}_1\rangle\ \cdots\ \langle \mathit{expr}_m\rangle\texttt{)}]\!] && \quad \texttt{(set! f}_1\ \langle \mathit{Expr}_1\rangle\texttt{)}\\
& && \quad \texttt{(set! f}_2\ \langle \mathit{Expr}_2\rangle\texttt{)}\\
& && \quad \cdots\\
& && \quad \texttt{(set! f}_n\ \langle \mathit{Expr}_n\rangle\texttt{)}\\
& && \quad \langle \mathit{expr}_1\rangle\ \cdots\ \langle \mathit{expr}_m\rangle\texttt{)}
\end{aligned}
$$

Cond:

Option 1:

Form:

The cond form has the general form:

$$\begin{aligned}(\texttt{cond}\ &\langle rib_1\rangle\\ &\cdots\\ &\langle rib_n\rangle)\end{aligned}$$

There are 3 kinds of cond-ribs:

1. The common form $(\langle expr\rangle\ \langle expr_1\rangle \cdots \langle expr_n\rangle)$, where $\langle expr\rangle$ is the test-expression: It is evaluated, and if not false, the rib is satisfied, all subsequent ribs are ignored, the corresponding implicit sequence is evaluated, and its final expression is returned.

Expands to:

The cond form macro-expands into nested if-expressions:

1. The general form of the rib converts into an if-expression with a condition and an explicit sequence for the then-clause. The else-clause of the if-expression continues the expansion of the cond:

Option 2:

Form:

② The arrow form ($\langle expr \rangle$ => $\langle expr_f \rangle$), where $\langle expr \rangle$ is evaluated: If non-false, the rib is satisfied, and the return value is the application of $\langle expr_f \rangle$ to the value of $\langle expr \rangle$.

Expands to:

② The arrow-form of the rib converts into a `let` that captures the value of the test, and if not false, passes it onto the function. For test-expression $\langle expr \rangle$, and function-expression $\langle expr_f \rangle$, the following expansion would do:

```
(let ((value [[⟨expr⟩]])
      (f (lambda () [[⟨expr_f⟩]])))
   (if value
       ((f) value)
       ;;; Continue with cond-ribs))
```

Option 3:

Form:

③ The `else`-rib has the form (`else` $\langle expr1 \rangle \cdots \langle expr_n \rangle$). It is satisfied immediately, and all subsequent ribs are ignored. The implicit sequence is evaluated, and the value of its final expression is returned.

Expands to:

③ The `else`-form of the rib converts into a `begin`-expression, and subsequent ribs are ignored

Quasiquote:

1. Upon receiving the expression `(unquote` $\langle sexpr \rangle$`)`, we return $\langle sexpr \rangle$
2. Upon receiving the expression `(unquote-splicing` $\langle sexpr \rangle$`)`, we generate an error message, and quit
3. Given either the empty list or a symbol, we wrap `(quote ···)` around it
4. Given a vector, we apply to it map the quasiquote-expander over the elements of the list, and apply the procedure `vector` to the elements of the resulting list

This is the heart of the algorithm:

5. Given a pair, let A be the `car`, and let B be the `cdr` respectively.
   - If $A =$`(unquote-splicing` $\langle sexpr \rangle$`)`, then return `(append` $sexpr$ $[\![B]\!]$`)`
   - If $B =$`(unquote-splicing` $\langle sexpr \rangle$`)`, then return `(cons` $[\![A]\!]$ $sexpr$`)`
   - Otherwise, return `(cons` $[\![A]\!]$ $[\![B]\!]$`)`

Examples:

Some examples:

sexpr	expansion
`,x`	`x`
`,@x`	*error*
`(a b)`	`(cons 'a (cons 'b '()))`
`(,a b)`	`(cons a (cons 'b '()))`
`(a ,b)`	`(cons 'a (cons b '()))`
`(,@a b)`	`(append a (cons 'b '()))`
`(a ,@b)`	`(cons 'a (append b '()))`

Some examples:

sexpr	expansion
`(,a ,@b)`	`(cons a (append b '()))`
`(,@a ,@b)`	`(append a (append b '()))`
`(,@a . ,b)`	`(append a b)`
`(,a . ,@b)`	`(cons a b)`
`(((,@a)))`	`(cons (cons (append a '()) '()) '())`
`#(a ,b c ,d)`	`(vector 'a b 'c d)`

And:

and

- Conjunctions are easily expanded into nested `if`-expressions:
  - $[\![$`(and)`$]\!]$ $=$ $[\![$`#t`$]\!]$ (by definition)
  - $[\![$`(and` $\langle \mathit{expr} \rangle$`)`$]\!]$ $=$ $[\![\langle \mathit{expr} \rangle]\!]$ (`#t` is the unit element of `and`)
  - $[\![$`(and` $\langle \mathit{expr}_1 \rangle$ $\langle \mathit{expr}_2 \rangle$ $\cdots$ $\langle \mathit{expr}_n \rangle$`)`$]\!]$ $=$ `(if` $[\![\langle \mathit{expr}_1 \rangle]\!]$ $[\![$`(and` $\langle \mathit{expr}_2 \rangle$ $\cdots$ $\langle \mathit{expr}_n \rangle$`)`$]\!]$ $[\![$`#f`$]\!]$`)`

MIT define:

- `(define (`$\langle \mathit{var} \rangle$ `.` $\langle \mathit{arglist} \rangle$`) . (`$\langle \mathit{expr} \rangle^+$`))`
  - This form is macro-expanded into `(define` $\langle \mathit{var} \rangle$ `(lambda` $\langle \mathit{arglist} \rangle$ `. (`$\langle \mathit{expr} \rangle^+$`)))`
  - Used to define functions without specifying the λ: This is almost always a bad idea!
  - ☞ Note the implicit sequences!
  - Example: `(define (square x) (* x x))`