Assignment 2 (Due: December 7th end of day (23:59))

Mayer Goldberg

December 1, 2021

Contents

1 General

- You may work on this assignment alone, or with a single partner. You may not join a group of two or more students to work on the assignment. If you are unable to find or maintain a partner with whom to work on the assignment, then you will work on it alone.
- You should be very careful to test your work before you submit. Testing means all your files are pushed to the git repository, you've implemented the required signatures in tag-parser.ml and you tested their behavior matchs the requirements in this document available and usable and are included in your submission.

- If you develop on other platforms and fail to verify that your work runs correctly under Chez Scheme, Ocaml, and your your work happens to fail under the graders' testing environment, then we shall not accept such failure as grouds for appeal or resubmission.
- Your work should run in Ocaml version on the departmental Linux image or the VM we provided. We will not test your work on other platforms. **Test, test, and test again:** Make sure your work runs correctly under Chez Scheme and Ocaml. We will not allow for re-submissions or corrections after the fact, so please be responsible and test!
- Make sure your code doesn't generate any unnecessary output: Forgotten debug statements, unnecessary print statements, etc., will result in your output being considered incorrect, and you will lose points. You will not get back these points by appealing or writing emails and complaints, so **please** be careful and make sure your code runs properly
- Please read this document completely, from start to finish, before beginning to write code for the assignment.

2 A tag-parser for Scheme

For this assignment, you shall need to implement a tag-parser in Scheme. You shall implement the procedure `tag_parse_expression` that takes an sexpr as an argument, and returns a tag-parsed sexpr of type `expr`. Parsing in this sense means annotating the sexpr with tags, so that the type of expression can be known easily and the various sub-expressions can be accessed with confidence (that is, without having to check each time to make sure that they syntactically correct and legal), and of course, parsing the sub-expressions as well.

Your tag parser should include the macro-expansion code according to the specifications below.

2.1 The Core Forms

The parser will recognize the following Scheme syntax:

2.1.1 Constants

You need to package constants with `ScmConst` tags: If an expression `<expr>` is a constant sexpr, then the parsed expression is `(ScmConst <expr>)`. There is one special case here that has to do with quoted expressions. For more information, see below.

Constant expressions can be formed from the following types, according to the examples below:

- Nil: `(ScmConst ScmNil)`
- Booleans: `ScmConst (ScmBoolean false)`
- Characters: `ScmConst (ScmChar 'a')`
- Numbers: `ScmConst (ScmNumber (ScmReal 3.1415))`
- Strings: `ScmConst (ScmString "abc")`
- Quoted sexprs: Remember that one level of quote needs to be removed. Consider the following examples:

```
# (* 'a *)
tag_parse_expression ((ScmPair (ScmSymbol "quote", (ScmPair (ScmSymbol "a", ScmNil)))));;
- : expr = ScmConst (ScmSymbol "a")
# (* ''a *)
tag_parse_expression (
ScmPair
 (ScmSymbol "quote",
  ScmPair
   (ScmPair
     (ScmSymbol "quote",
      ScmPair (ScmSymbol "a", ScmNil)),
    ScmNil)));;
- : expr =
ScmConst
 (ScmPair
   (ScmSymbol "quote", ScmPair (ScmSymbol "a", ScmNil)))
# (* '(a b c) *)
tag_parse_expression (ScmPair
 (ScmSymbol "quote",
  ScmPair
   (ScmPair
     (ScmSymbol "a", ScmPair (ScmSymbol "b", ScmPair (ScmSymbol "c", ScmNil))),
    ScmNil)));;
- : expr =
ScmConst
 (ScmPair
   (ScmSymbol "a", ScmPair (ScmSymbol "b", ScmPair (ScmSymbol "c", ScmNil))))
# (* (quote '(a b c)) *)
tag_parse_expression (
ScmPair
 (ScmSymbol "quote",
  ScmPair
   (ScmPair
     (ScmSymbol "quote",
      ScmPair
       (ScmPair
         (ScmSymbol "a",
          ScmPair (ScmSymbol "b", ScmPair (ScmSymbol "c", ScmNil))),
        ScmNil)),
    ScmNil)));;
- : expr =
ScmConst
 (ScmPair
   (ScmSymbol "quote",
    ScmPair
     (ScmPair
       (ScmSymbol "a",
        ScmPair (ScmSymbol "b", ScmPair (ScmSymbol "c", ScmNil))),
      ScmNil)))
# (* '#(a b c) *)
tag_parse_expression (
ScmPair
 (ScmSymbol "quote",
  ScmPair (ScmVector [ScmSymbol "a"; ScmSymbol "b"; ScmSymbol "c"], ScmNil)));;
- : expr = ScmConst (ScmVector [ScmSymbol "a"; ScmSymbol "b"; ScmSymbol "c"])
```

2.1.2 Variables

Package variables with the `ScmVar` tag: If an expression consists of the non-reserved symbol named `<M>`, then the parsed expression is `(ScmVar <M>)`. Consider the following examples:

```
# tag_parse_expression (ScmSymbol "abc");;
- : expr = ScmVar "abc"
# tag_parse_expression (ScmSymbol "if");;
Exception: X_reserved_word "if".
```

Here is the list of symbols that are reserved words:

```
# reserved_word_list;;
- : string list =
["and"; "begin"; "cond"; "define"; "else"; "if"; "lambda"; "let"; "let*";
 "letrec"; "or"; "quasiquote"; "quote"; "set!"; "unquote";
 "unquote-splicing"]
```

2.1.3 Conditionals

Expressions of the form `(if <test> <dit> <dif>)` are parsed with the Tag `ScmIf` which contains 3 expr sub-trees
`(tag_parse_expression <test>, tag_parse_expression <dit>, tag_parse_expression <dif>)`.

```
# (* (if a b c) *)
tag_parse_expression (
ScmPair
 (ScmSymbol "if",
  ScmPair
   (ScmSymbol "a", ScmPair (ScmSymbol "b", ScmPair (ScmSymbol "c", ScmNil)))));;
- : expr = ScmIf (ScmVar "a", ScmVar "b", ScmVar "c")
```

Expressions of the form `(if <test> <dit>)` are also parsed using the same `ScmIf` tag, where the missing `<dif>` clause is parsed to `ScmConst ScmVoid`

```
# tag_parse_expression (
ScmPair
 (ScmSymbol "if", ScmPair (ScmSymbol "a", ScmPair (ScmSymbol "b", ScmNil))));;
- : expr = ScmIf (ScmVar "a", ScmVar "b", ScmConst ScmVoid)
```

The *void* object was not introduced in the lecture on *S-expressions* because it really isn't an S-expression: There is no way to type a constant literal void. However, the *void* object is a constant, and should be parsed accordingly, i.e., packaged in the `ScmConst` structure.

2.1.4 Disjunctions

An expression of the form `(or expr1 expr2 expr3 ... )` is parsed using the `ScmOr` tag which holds a list of subtrees for the disjunction expressions `[tag_parse_expression expr1; tag_parse_expression expr2; ...]`

```
# (* (or a b) *)
tag_parse_expression (
ScmPair
 (ScmSymbol "or", ScmPair (ScmSymbol "a", ScmPair (ScmSymbol "b", ScmNil))));;
- : expr = ScmOr [ScmVar "a"; ScmVar "b"]
# (* (or a) *)
tag_parse_expression (
ScmPair (ScmSymbol "or", ScmPair (ScmSymbol "a", ScmNil)));;
- : expr = ScmVar "a"
# (* (or) *)
tag_parse_expression (
ScmPair (ScmSymbol "or", ScmNil));;
- : expr = ScmConst (ScmBoolean false)
```

Note the two base case for `or` expressions. By definition a disjunction of zero expressions is `false`, and a disjunction of a single expression `<expr>` is just `<expr>`.

2.1.5 Lambda forms

Three types of lambda-expressions are supported by the parser, using two AST nodes:

1. Simple lambda

   The most common lambda-expression is of the form

   ```
   (lambda (<v1> ... <vn>) <e1> ... )
   ```

   where `(<v1> ... <vn>)` is a list of zero or more variables, and `<e1>`, ... denotes one or more expressions. The simple lambda expression `(lambda (<x1> ... <xn>) <e1> ... <em>)` is parsed using the `ScmLambdaSimple` tag, where the parameters list `(<x1> ... <xn>)` is parsed as a `string list` and the body is parsed as `tag_parse_expression <e1>` if it's a single expression or, if its multiple expression (i.e. an implicit sequence), it's parsed as `ScmSeq [tag_parse_expression e1; tag_parse_expression e2;...]` (**See more about sequences in section 2.1.9**)

   ```
   # (* (lambda (a b) b) *)
   tag_parse_expression (
   ScmPair
    (ScmSymbol "lambda",
     ScmPair
      (ScmPair (ScmSymbol "a", ScmPair (ScmSymbol "b", ScmNil)),
       ScmPair (ScmSymbol "b", ScmNil))));;
   - : expr = ScmLambdaSimple (["a"; "b"], ScmVar "b")

   # (* (lambda (a b) b a) *)
   tag_parse_expression (
   ```

```
ScmPair
 (ScmSymbol "lambda",
  ScmPair
   (ScmPair (ScmSymbol "a", ScmPair (ScmSymbol "b", ScmNil)),
    ScmPair (ScmSymbol "b", ScmPair (ScmSymbol "a", ScmNil)))));;
- : expr = ScmLambdaSimple (["a"; "b"], ScmSeq [ScmVar "b"; ScmVar "a"])
```

2. Lambda with optional arguments

   The lambda-expression with optional arguments is of the form

   ```
   (lambda (<v1> ... <vn> . <v-rest>) <e1> ... <em>)
   ```

   where `(<v1> ... <vn> . <v-rest>)` is an improper list of variables, and `<e1>` … `<em>` denotes one or more expressions. The lambda-expression with optional arguments `(lambda (<x1> ... <xn> . <x>) <e1> ... <em>)` is parsed using the `ScmLambdaOpt` tag, where the parameters list `(<x1> ... <xn>)` is parsed as a `string list` and the, `x` is parsed as a `string` and the body is parsed the same as in a simple lambda

   ```
   # (* (lambda (a . b) b) *)
   tag_parse_expression (
   ScmPair
    (ScmSymbol "lambda",
     ScmPair
      (ScmPair (ScmSymbol "a", ScmSymbol "b"),
       ScmPair (ScmSymbol "b", ScmNil))));;
   - : expr = ScmLambdaOpt (["a"], "b", ScmVar "b")

   # (* (lambda (a . b) b a) *)
   tag_parse_expression (
   ScmPair
    (ScmSymbol "lambda",
     ScmPair
      (ScmPair (ScmSymbol "a", ScmSymbol "b"),
       ScmPair (ScmSymbol "b", ScmPair (ScmSymbol "a", ScmNil)))));;
   - : expr = ScmLambdaOpt (["a"], "b", ScmSeq [ScmVar "b"; ScmVar "a"])
   ```

3. Variadic lambda

   ```
   (lambda v e1 ... em)
   ```

   The variadic lambda is of the form `(lambda <args> <e1> ... <em>)` where `<args>` is a symbol, and `<e1>` … `<em>` denotes one or more expressions. The variadic lambda binds the list of its arguments to the single identifier `<args>`. The variadic lambda expression `(lambda <args> <e1> ... <em>)` is parsed as a special case of the `ScmLambdaOpt` form, where the list of required parameters is empty.

   ```
   # (* (lambda a a) *)
   tag_parse_expression (
   ScmPair
    (ScmSymbol "lambda",
     ScmPair
   ```

```
    (ScmSymbol "a",
     ScmPair (ScmSymbol "a", ScmNil))));;
- : expr = ScmLambdaOpt ([], "a", ScmVar "a")

# (* (lambda a a #\b) *)
tag_parse_expression (
ScmPair
 (ScmSymbol "lambda",
  ScmPair
    (ScmSymbol "a",
     ScmPair (ScmSymbol "a", ScmPair (ScmChar 'b', ScmNil)))));;
- : expr = ScmLambdaOpt ([], "a", ScmSeq [ScmVar "a"; ScmConst (ScmChar 'b')])
```

2.1.6 Define

Two types of `define`-expressions are supported by the parser: Simple `define`-expressions and the MIT-style `define`-expressions used for defining procedures. They are both be parsed using the single define-record, so the MIT-style define will need some processing so that it fits the correct format.

1. Simple define

   An expression of the form `(define <var> <val>)` is parsed using the `ScmDef` tag which contains two subexpressions, an `expr`, `<var>`, who's the variable to define, and another `expr`, `<val>`, who's value will be the new value for `<var>`.

   The subexpression `<var>` has to be a Scheme variable (that is, it must be tagged with `ScmVar`). If it's not a variable, this is a syntax error.

```
# (* (define x 5) *)
tag_parse_expression (
ScmPair
 (ScmSymbol "define",
  ScmPair (ScmSymbol "x", ScmPair (ScmNumber (ScmRational (5, 1)), ScmNil))));;
- : expr = ScmDef (ScmVar "x", ScmConst (ScmNumber (ScmRational (5, 1))))

# (* (define define 5) *)
tag_parse_expression (
ScmPair
 (ScmSymbol "define",
  ScmPair (ScmSymbol "define", ScmPair (ScmNumber (ScmRational (5, 1)), ScmNil))));;
Exception:
X_syntax_error
 (ScmPair
    (ScmSymbol "define",
     ScmPair
       (ScmSymbol "define", ScmPair (ScmNumber (ScmRational (5, 1)), ScmNil))),
 "Expected variable on LHS of define").
```

2. MIT-style define (syntactic sugar for defining procedures)

An expression of the form `(define (<var> . <args>) <e1> ... <em>)` is parsed using the `ScmDef` tag by tag-parsing the expression: `(define <var> <lambdaExpr>)` where `<lambdaExpr>` is `(lambda <args> <e1> ... <em>)`.

Note that `<args>` may be a proper list (denoting a simple lambda), an improper list (denoting an optional lambda) or a symbol (denoting a variadic lambda).

```
# (* (define (id x) x) *)
tag_parse_expression (
ScmPair
 (ScmSymbol "define",
  ScmPair
   (ScmPair (ScmSymbol "id", ScmPair (ScmSymbol "x", ScmNil)),
    ScmPair (ScmSymbol "x", ScmNil))));;
- : expr = ScmDef (ScmVar "id", ScmLambdaSimple (["x"], ScmVar "x"))

(* (define (foo var . vars) (if var (display vars))) *)
tag_parse_expression (ScmPair
 (ScmSymbol "define",
  ScmPair
   (ScmPair (ScmSymbol "foo", ScmPair (ScmSymbol "var", ScmSymbol "vars")),
    ScmPair
     (ScmPair
       (ScmSymbol "if",
        ScmPair
         (ScmSymbol "var",
          ScmPair
           (ScmPair (ScmSymbol "display", ScmPair (ScmSymbol "vars", ScmNil)),
            ScmNil))),
      ScmNil))));;
- : expr =
ScmDef (ScmVar "foo",
 ScmLambdaOpt (["var"], "vars",
  ScmIf (ScmVar "var", ScmApplic (ScmVar "display", [ScmVar "vars"]),
   ScmConst ScmVoid)))
```

2.1.7 Assignments

Assignments are written using `set!`-expressions. The general form of `set!`-expressions is `(set! <var> <val>)`. This should be parsed using the `ScmSet` tag. This expression contains two subexpressions, an `expr`, `<var>`, which is the varaible being assigned (which must be an `ScmVar`), and another `expr`, `<val>`, that is evaluated to create the new value for `var`

```
# (* (set! x "value") *)
tag_parse_expression (
ScmPair
 (ScmSymbol "set!",
  ScmPair (ScmSymbol "x", ScmPair (ScmString "value", ScmNil))));;
- : expr = ScmSet (ScmVar "x", ScmConst (ScmString "value"))
```

```
# (* (set! "x" "value") *)
tag_parse_expression (
ScmPair
 (ScmSymbol "set!",
  ScmPair (ScmString "x", ScmPair (ScmString "value", ScmNil))));;
Exception:
X_syntax_error
 (ScmPair
   (ScmSymbol "set!",
    ScmPair (ScmString "x", ScmPair (ScmString "value", ScmNil))),
 "Expected variable on LHS of set!").
```

2.1.8 Applications

An expression of the form `(<e1> ... <em>)` is an application if `<e1>` isn't a reserved symbol, and is parsed using an `ScmApplic` tag who's subexpressions are `[tag_parse_expression <e2>; tag_parse_expression <e3>; ...]`

```
# (* (a b #\c) *)
tag_parse_expression (
ScmPair (ScmSymbol "a", ScmPair (ScmSymbol "b", ScmPair (ScmChar 'c', ScmNil))));;
- : expr = ScmApplic (ScmVar "a", [ScmVar "b"; ScmConst (ScmChar 'c')])

# (* (a) *)
tag_parse_expression (
ScmPair (ScmSymbol "a", ScmNil));;
- : expr = ScmApplic (ScmVar "a", [])

# (* ((lambda v (apply + v)) 2.570750 0.570750) *)
tag_parse_expression (ScmPair
 (ScmPair
   (ScmSymbol "lambda",
    ScmPair
     (ScmSymbol "v",
      ScmPair
       (ScmPair
         (ScmSymbol "apply",
          ScmPair (ScmSymbol "+", ScmPair (ScmSymbol "v", ScmNil))),
        ScmNil))),
  ScmPair
   (ScmNumber (ScmReal 2.57075), ScmPair (ScmNumber (ScmReal 0.57075), ScmNil))));;
- : expr =
ScmApplic
 (ScmLambdaOpt ([], "v", ScmApplic (ScmVar "apply", [ScmVar "+"; ScmVar "v"])),
 [ScmConst (ScmNumber (ScmReal 2.57075));
  ScmConst (ScmNumber (ScmReal 0.57075))])
```

2.1.9 Sequences

There are two kinds of sequences of expressions: Explicit, and implicit. Explicit sequences are `begin`-expressions. Implicit sequences of expressions appear in various special forms, such as `cond`, `lambda`, `let`, etc. Both kinds of sequences are parsed using the `ScmSeq` tag with the list of subexpressions `[tag_parse_expression <e1>; tag_parse_expression <e2>2; ...]`

```
# (* (begin "e1" e2) *)
tag_parse_expression (
ScmPair
 (ScmSymbol "begin",
  ScmPair (ScmString "e1", ScmPair (ScmSymbol "e2", ScmNil))));;
- : expr = ScmSeq [ScmConst (ScmString "e1"); ScmVar "e2"]

# (* (lambda (v1 v2) (display v1) v2) *)
tag_parse_expression (
ScmPair
 (ScmSymbol "lambda",
  ScmPair
   (ScmPair (ScmSymbol "v1", ScmPair (ScmSymbol "v2", ScmNil)),
    ScmPair
     (ScmPair (ScmSymbol "display", ScmPair (ScmSymbol "v1", ScmNil)),
      ScmPair (ScmSymbol "v2", ScmNil)))));;
- : expr =
ScmLambdaSimple (["v1"; "v2"],
 ScmSeq [ScmApplic (ScmVar "display", [ScmVar "v1"]); ScmVar "v2"])
```

2.2 Macro-Expanding Special Forms in Scheme

For this problem, you will need to recognize some of the special forms in Scheme and remove them by macro expansion.

The forms you will support are:

- `let`, `let*`, `letrec`
- `and`
- `cond`, MIT-style `define`

Please keep in mind that there are many things to check, and that your code will be ttested for accuracy and thoroughness: The variables on the left hand side of the ribs in a `let` and `letrec` expressions must all be different. This needn't be the case in `let*` expressions. Refer to the lectures to learn about the behavior of the `and` and `or` special forms; It's not as trivial as it may appear at first. There are many such small points that will need to be checked to make sure that what you have is a valid expression.

Your macro-expanders *may not use* Scheme's `gensym` procedure or any other mechanism for inventing new names (even if you write one yourself!).

Any variables or constant symbols your parser generates during macro-expansion, must follow the name formats as shown in the lectures.

2.3 Expanding `cond` statements

You should follow the expansion instructions show in the lectures for expanding `cond`-expressions. Your parser should support:

- Simple `cond`-ribs, where the first expression in the rib is the condition for the rib and the rest are an implicit sequence of expressions
- `else`-rib, where the first expression in the rib is the symbol `begin` and the rest are an implicit sequence of expressions. An empty `else`-rib is invalid syntax. Any subsequent rib after the `else`-rib is ignored.
- arrow-rib, where the first expression is the condition for the rib and the second expression evaluates to a procedure that takes the condition's value as an argument. You need not (and in fact cannot) verify that the second expression is in fact a procedure.

Note that an empty `cond`-expression is invalid syntax. A `cond`-expression without an `else`-rib where the conditions for all rib's are not met evaluates to the `void`-object. You should test this behavior in your macro-expander

2.4 Expanding `let`, `let*`, `letrec`

- The order of the ribs in the `let`-expression and `letrec`-expressions determines the order of the parameters in the corresponding `lambda`-expressions.
- The `let*`-expression should be expanded into nested `let`-expressions, where the order of ribs in the `let*`-expression describe the order of nesting for the expanded `let`-expressions.
- The `letrec`-expression should be expanded to use side-effects in the way described in class. The expansion you're required to implement is the one that does not/ support nested define expressions.

2.5 Handling quasiquote-expressions

Your tag-parser should handle *quasiquote*expressions. Expressions of the form `(quasiquote <sexpr>)` will need to be expanded into combinations of static and dynamic expressions. The results of dynamic expressions are combined with constant data via applications of procedures for constructing lists. The specific requirements for this expansion appear in the slides and were presented in the lecture

Here are a few examples of expanding quasiquote:

```
# (* `(a b c) *)
macro_expand (
  ScmPair
   (ScmSymbol "quasiquote",
    ScmPair
     (ScmPair
       (ScmSymbol "a", ScmPair (ScmSymbol "b", ScmPair (ScmSymbol "c", ScmNil))),
      ScmNil)));;
- : sexpr =
ScmPair
 (ScmSymbol "cons",
  ScmPair
   (ScmPair (ScmSymbol "quote", ScmPair (ScmSymbol "a", ScmNil)),
    ScmPair
     (ScmPair
       (ScmSymbol "cons",
        ScmPair
         (ScmPair (ScmSymbol "quote", ScmPair (ScmSymbol "b", ScmNil)),
          ScmPair
           (ScmPair
             (ScmSymbol "cons",
              ScmPair
               (ScmPair (ScmSymbol "quote", ScmPair (ScmSymbol "c", ScmNil)),
                ScmPair
                 (ScmPair (ScmSymbol "quote", ScmPair (ScmNil, ScmNil)),
                  ScmNil))),
            ScmNil))),
      ScmNil)))
(* (cons 'a (cons 'b (cons 'c '()))) *)

# (* `(a ,b c) *)
macro_expand (
  ScmPair
   (ScmSymbol "quasiquote",
    ScmPair
     (ScmPair
       (ScmSymbol "a",
        ScmPair
         (ScmPair (ScmSymbol "unquote", ScmPair (ScmSymbol "b", ScmNil)),
          ScmPair (ScmSymbol "c", ScmNil))),
      ScmNil)));;
- : sexpr =
  ScmPair
   (ScmSymbol "cons",
    ScmPair
     (ScmPair (ScmSymbol "quote", ScmPair (ScmSymbol "a", ScmNil)),
      ScmPair
       (ScmPair
         (ScmSymbol "cons",
          ScmPair
           (ScmSymbol "b",
            ScmPair
             (ScmPair
               (ScmSymbol "cons",
                ScmPair
                 (ScmPair (ScmSymbol "quote", ScmPair (ScmSymbol "c", ScmNil)),
                  ScmPair
                   (ScmPair (ScmSymbol "quote", ScmPair (ScmNil, ScmNil)),
                    ScmNil))),
              ScmNil))),
        ScmNil)))
(* (cons 'a (cons b (cons 'c '()))) *)

# (* `(a ,b ,@c) *)
macro_expand (
  ScmPair
   (ScmSymbol "quasiquote",
    ScmPair
     (ScmPair
       (ScmSymbol "a",
        ScmPair
         (ScmPair (ScmSymbol "unquote", ScmPair (ScmSymbol "b", ScmNil)),
          ScmPair
           (ScmPair
             (ScmSymbol "unquote-splicing", ScmPair (ScmSymbol "c", ScmNil)),
            ScmNil))),
      ScmNil)));;
- : sexpr =
ScmPair
 (ScmSymbol "cons",
  ScmPair
   (ScmPair (ScmSymbol "quote", ScmPair (ScmSymbol "a", ScmNil)),
    ScmPair
     (ScmPair
       (ScmSymbol "cons",
        ScmPair
         (ScmSymbol "b",
          ScmPair
           (ScmPair
             (ScmSymbol "append",
              ScmPair
               (ScmSymbol "c",
                ScmPair
                 (ScmPair (ScmSymbol "quote", ScmPair (ScmNil, ScmNil)),
                  ScmNil))),
            ScmNil))),
      ScmNil)))
(* (cons 'a (cons b (append c '()))) *)

# (* `(,@a ,@b c) *)
macro_expand (
  ScmPair
   (ScmSymbol "quasiquote",
    ScmPair
     (ScmPair
       (ScmPair (ScmSymbol "unquote-splicing", ScmPair (ScmSymbol "a", ScmNil)),
        ScmPair
```

```
          (ScmPair
            (ScmSymbol "unquote-splicing", ScmPair (ScmSymbol "b", ScmNil)),
          ScmPair (ScmSymbol "c", ScmNil))),
      ScmNil)));;
- : sexpr =
ScmPair
 (ScmSymbol "append",
  ScmPair
   (ScmSymbol "a",
    ScmPair
     (ScmPair
       (ScmSymbol "append",
        ScmPair
         (ScmSymbol "b",
          ScmPair
           (ScmPair
             (ScmSymbol "cons",
              ScmPair
               (ScmPair (ScmSymbol "quote", ScmPair (ScmSymbol "c", ScmNil)),
                ScmPair
                 (ScmPair (ScmSymbol "quote", ScmPair (ScmNil, ScmNil)),
                  ScmNil))),
            ScmNil))),
      ScmNil)))
(* (append a (append b (cons 'c '()))) *)
```

2.6 Summary

There is plenty of Scheme syntax that isn't covered by the parser or the macro system described here. We will only test your code against Scheme syntax specified in this document.

Your parser should then be able to return *parse-trees* for all the core forms. For the forms that are handled by your macro expander, your parser should expand and then parse these forms, returning *parse-trees* containing the core forms only. This means, for example, that when you parse a `let`-expression, your tag parser should return a parse-tree of an application of a `lambda`-expression to a list of arguments.

3 Submission Guidelines

Submission guidelines are as detailed in the `README.md` file in your git repositories.

Note that work on this assignment should be done on the same repository as you worked on during the first assignment. If, for some reason, your old repository can't be updated with the skeleton for this assignment, you should create a fresh repository and copy over all the files from the previous one manually.

4 A word of advice

The class is very large. We do not have the human resources to handle late submissions or late corrections from people who do not follow instructions. By contrast, it should take you very little effort to make sure your submission conforms to what we ask. If you fail to follow the instructions to the letter, you will not have another chance to submit the assignment:

4.1 A final checklist

1. You copmleted the `parse` function in the `tag-parser.scm` file
2. You did not break the interface
3. Your tag-parser supports all the core forms
4. Your tag-parser supports all the special forms
5. Your tag-parser supports quasiquoted expressions
6. Your tag-parser runs correctly under Ocaml and Chez Scheme on the

departmental Linux image or the VM image included in the repository

1. You commited you work in its entirety
2. You pushed your work to your git repository
3. You created and submitted the `submission` file to the submission system