Assignment 2 (Due: Monday (12:00pm (צהריים), December 12, 2016)

Mayer Goldberg

November 27, 2016

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 your work means that all the files you require are available and usable and are included in your submission.

- Your work should run in Chez Scheme. We will not test your work on other platforms. **Test, test, and test again:** Make sure your work runs under Chez Scheme the same way you had it running under Racket. We will not allow for re-submissions or corrections after the deadline, 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

2 Changes to this document

- *<2016-11-27 Sun>*
  - Gave additional examples of constants (with *nil* and *vectors*).
  - Added the core form `or` to the tag-parser.
  - Added the section on changes to this document.

3 A tag-parser for Scheme

For this assignment, you need to implement a tag-parser in Scheme. You will implement the procedure `parse` which will take an sexpr as an argument, and return a parsed sexpr. 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 gotten at 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.

3.1 The Core Forms

The parser will recognize the following Scheme syntax:

3.1.1 Constants

You need to package the constant within a `const`-record: If an expression `M` is a constant, then the parsed expression is `(const M)`. 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: `(const ())`
- Vectors: `(const #(1 (2 3 4) 2 3))`
- Booleans: `(const #f)`
- Characters: `(const #\a)`
- Numbers: `(const 34)`
- Strings: `(const "abc")`

- Quotes sexprs: Remember that the quote needs to be removed. Consider the following examples:

```
> (parse '(quote a))
(const a)
> (parse '(quote (a b c)))
(const (a b c))
> (parse '(quote (quote a b c)))
(const (quote a b c))
```

Note that this last one is *not* an error; Think why!

3.1.2 Variables

You will need to package variables within a variable-record: If an expression consists of the non-reserved symbol `M`, then the parsed expression is `(var M)`. Consider the following examples:

```
> (parse 'abc)
(var abc)
> (parse '123x)
(var 123x)
```

This last example demonstrates the extension you were asked to support of the Scheme syntax: Remember that *officially* a variable name isn't supposed to begin with a digit, but we aren't going by this.

Here is the list of reserved words that I use:

```
(define *reserved-words*
  '(and begin cond define do else if lambda
    let let* letrec or quasiquote unquote
    unquote-splicing quote set!))
```

3.1.3 Conditionals

Expressions of the form `(if <test> <dit> <dif>)` are parsed as follows: `` `(if3 ,(parse <test>) ,(parse <dit>) ,(parse <dif>)) ``:

```
> (parse '(if a b c))
(if3 (var a) (var b) (var c))
> (parse '(if (if a b c)
      'x
      '(x y z)))
(if3 (if3 (var a)
  (var b)
  (var c))
  (const x)
  (const (x y z)))
```

Expressions of the form `(if <test> <dit>)` are parsed as follows: `` `(if3 ,(parse <test>) ,(parse <dit>) ,(parse (void))) ``:

```
> (parse '(if a b))
(if3 (var a) (var b) (const #<void>))
> (parse '(if a 4))
(if3 (var a) (const 4) (const #<void>))
> (parse '(if #t 'abc))
(if3 (const #t)
  (const abc)
  (const #<void>))
```

3.1.4 Disjunctions

An expression of the form `(or expr1 expr2 expr3 ... )` will be parsed using an or-record as follows `(or (peExpr1 peExpr2 peExpr3 ... ))`, where `peExpr1`, `peExpr2`, `peExpr3`, etc, are the the parsed expressions corresponding to `expr1`, `expr2`, `expr3`, etc. Consider the following examples:

```
> (parse '(or (zero? x) (zero? y) (zero? z)))
(or ((applic (var zero?) ((var x)))
      (applic (var zero?) ((var y)))
      (applic (var zero?) ((var z)))))
> (parse '(or (or (f1 x) (f2 y))
      (or (f3 z) (f4 w) (f5 r))
      (and (f6 u) (f7 t))))
(or ((or ((applic (var f1) ((var x)))
   (applic (var f2) ((var y)))))
      (or ((applic (var f3) ((var z)))
    (applic (var f4) ((var w)))
    (applic (var f5) ((var r)))))
      (if3 (applic (var f6) ((var u)))
   (applic (var f7) ((var t)))
   (const #f))))
```

3.1.5 Lambda forms

Three types of lambda-expressions are supported by the parser:

1. Regular 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 regular lambda expression `(lambda (x1 ... xn) e1 ... em)` is parsed using the lambda-simple record, as `(lambda-simple (x1 ... xn) <Body>)` where `<Body>` is the parsed expression corresponding to the sequence `e1` … `em`. Consider the follwing examples:

   ```
   > (parse
      '(lambda (x y z)
   ```

```
      (if x y z)))
(lambda-simple (x y z)
       (if3 (var x)
    (var y)
    (var z)))
> (parse '(lambda () a))
(lambda-simple ()
       (var 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 intended use of the variable `v-rest` is that upon application it will be bound to the list of optional arguments to the procedure. The lambda expression with optional arguments `(lambda (x1 ... xn . x) e1 ... em)` is parsed using the `lambda-opt` record, as `` `(lambda-opt (x1 ... xn) x ,(parse `(begin e1 ... em))) ``. Consider the following examples:

   ```
   > (parse
      '(lambda (x y z . rest)
         (if x y z)))
   (lambda-opt (x y z) rest
       (if3 (var x)
    (var y)
    (var z)))
   > (parse
      '(lambda (x . rest)
         rest))
   (lambda-opt (x) rest
       (var rest))
   ```

3. Variadic lambda

   The variadic lambda is of the form `(lambda args e1 ... em)` where `args` is a variable, 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 using the `lambda-var` record, as `` `(lambda-var args ,(parse `(begin e1 ... em))) ``. Consider the

   ```
   > (parse
      '(lambda args
         (if x y z)))
   (lambda-var args
     (if3 (var x)
          (var y)
   ```

```
       (var z)))
> (parse '(lambda args args))
(lambda-var args
  (var args))
```

You *must* use the `begin` special form in order to expand *all forms of* `lambda` that have a sequence of more than one expression in the body. *This is very different from what you did in PPL,* where *thunks* were used to macro-expand `begin`-expressions. See the item on sequences below.

3.1.6 Define

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

1. Regular define

   An expression of the form `(define var expr)` will be parsed using the define-record as follows: `(define (var var) peExpr)` where `peExpr` is the parsed expression corresponding to `expr`. Consider the following examples:

   ```
   > (parse '(define x 5))
   (define (var x) (const 5))
   > (parse '(define x (lambda (x) x)))
   (define (var x)
     (lambda-simple (x)
       (var x)))
   ```

2. MIT-style define (for procedures)

   An expression of the form `(define (var v1 ... vn) e1 ... em)` will be parsed using the define record as follows: `(define var peExpr)` where `peExpr` is the parsed expression corresponding to the expression `(lambda (v1 ... vn) e1 ... em)`. Consider the following examples:

   ```
   > (parse '(define (id x) x))
   (define (var id)
     (lambda-simple (x)
    (var x)))
   > (parse '(define (foo x y z)
       (if x y z)))
   (define (var foo)
     (lambda-simple (x y z)
    (if3 (var x)
         (var y)
         (var z))))
   > (parse '(define (foo x y . z)
   ```

```
      (if x y z)))
    (define (var foo)
      (lambda-opt (x y) z
          (if3 (var x)
       (var y)
       (var z))))
    > (parse '(define (list . args)
      args))
    (define (var list)
      (lambda-var args
        (var args)))
```

3.1.7 Applications

An expression of the form `(E1 ... Em)` is an application if e1 isn't a reserved symbol, and will be parsed using the `applic`-record as follows: `` `(applic ,(parse E1) (,(parse E2) ... ,(parse Em))) ``. If `E1` is called with no arguments then the list of parsed arguments given by `(,(parse E2) ... ,(parse Em))` is empty. The rationale behind the structure of the `applic`-record is that we would like to separate the procedure from the arguments it takes. This will make life a bit simpler later on. Consider the following examples:

```
> (parse '(a))
(applic (var a) ())
> (parse '(a b c))
(applic (var a)
  ((var b) (var c)))
> (parse '((a b) (a c)))
(applic
  (applic (var a)
    ((var b)))
  ((applic (var a)
   ((var c)))))
```

3.1.8 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 will be parsed using the seq record: The sequence `E1`, … , `En` of expressions will be parsed as

```
`(seq (,(parse E1) ... ,(parse En)))
```

3.1.9 Summary

There is plenty of Scheme syntax that isn't covered by the parser or the macro system described here. Within this course, we will assume that this extra syntax simply doesn't exist, and in any case you need not support it in your parser: The parser does not need to recognize any additional syntax – neither additional special forms nor variations on supported forms. Just what's been specified above.

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, the parser should return the parse tree for an application of the appropriate parsed lambda and the parsed arguments.

3.2 Macro-Expanding Special Forms in Scheme

For this problem you need to make sure you're accurate and complete. For this problem, you will need to recognize some of Scheme's special forms and remove them by macro expansion.

The forms you will need to be able to handle are as follows:

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

Please keep in mind that there are many things to check, and that because this is a relatively mindless exercise, your code will be tested 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. Read over the Revised Report on Scheme and learn 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 carefully to make sure that what you have is a valid expression.

Your macro-expanders *may not use* Scheme's `gensym` procedure (if you're not familiar with the `gensym` procedure in Scheme – good for you!) or any other mechanism for inventing new names at run-time (even if you write one yourself!).

3.3 Handling *quasiquoted* expressions

Your tag parser will need to be able to handle *quasiquoted* expressions. Expressions of the form `(quasiquote <sexpr>)` will need to be expanded into either constants or applications, based on what they contain. When `<sexpr>` contains sub-expressions of the form `(unquote <sexpr>)` or `(unquote-splicing <sexpr>)`, you will need to be able to convert such expressions into semantically-equivalent applications using `cons` and `append` respectively.

The latest version of Scheme supports *nested quasiquotes*. You do not need to support these, as their implementation is non-trivial. One-level quasiquotation has been present in LISP and Scheme for decades and is quite straightforward to implement. Here are examples from my own *quasiquote expander* (the expander takes the quasiquoted expression without the `quasiquote` itself):

```
> (expand-qq '(a b c))
(cons 'a (cons 'b (cons 'c '())))
> (expand-qq '(a ,b c))
(cons 'a (cons b (cons 'c '())))
> (expand-qq '(a ,b ,@c))
(cons 'a (cons b (append c '())))
> (expand-qq '(,@a ,b c))
(append a (cons b (cons 'c '())))
```

As you can see, the code it generates is pretty horrible, but the expander itself is short & sweet (14 lines of Scheme source code). I could have spent more efforts trying to write a smarter expander, but instead, I used the pattern-matching package to write a post-expansion optimizer:

```
> (optimize-qq-expansion (expand-qq '(a b c)))
'(a b c)
> (optimize-qq-expansion (expand-qq '(a ,b c)))
(cons 'a (cons b '(c)))
> (optimize-qq-expansion (expand-qq '(a ,b ,@c)))
(cons 'a (cons b c))
```

There's still plenty of room for further optimization on the expressions I generate, but already they generate considerably nicer code. Post-expansion optimization's seem like a far easier route to generating good code for `quasiquote`-expressions than to try and figure out how to generate perfect code recursively. Some problems just don't lend themselves to simple recursive solutions.

You do not need to worry about optimizing your `quasiquote` expansion code. Just get it to work as quickly and as correctly as you can.

3.4 What to hand in

Add all your code for this problem in the file `compiler.scm`.

4 Compiler Optimization: Common Sub-Expression Elimination

For this assignment you will implement the `cse` procedure that implements *common sub-expression elimination* for the language of applications. This means that you will need to factor out commonly recurring sub-expressions using the special form `let*`.

When you create your own `let*`-expressions, you'll need to create variable names for the left-hand side of the `let`-rib. If you use your own mechanism for generating symbols, you risk name collisions with the variable names in the expressions you send to `cse` (in other words, you may be using the same variable names!). To avoid this, you can use the built-in procedure `gensym`. You invoke `gensym` with no arguments, and the return value is a new symbol that hasn't been previously used in the system. There are no assumptions as to how the symbol should look, and each Scheme system creates its own bizarre `gensym`-symbol names. Chez, the Scheme system I used to write the solution to this problem, creates symbols that look like `#:g45`, `#:g46`, `#:g47`, etc, generally numbering the symbols consecutively. Other Scheme systems might have a different printed representation for `gensym`-generated symbols.

Only applications are factored out. Below are some annotated examples. In these examples, nothing can be factored out, so we return the original expressions:

```
> (cse '(+ 2 3))
(+ 2 3)

> (cse '(f (f (f (f x)))))
(f (f (f (f x))))
```

In the following examples, the `cse` procedure identifies a recurring sub-expression, and factors it out, returning the *text* of a `let*`-expression:

```
> (cse '(* (+ 2 3 4) (+ 2 3 4)))
(let* ((#:g1408 (+ 2 3 4))) (* #:g1408 #:g1408))

> (cse '(f (g x y) (f (g x y) z)))
(let* ((#:g1410 (g x y))) (f #:g1410 (f #:g1410 z)))
```

The problem becomes interesting when some recurring sub-expressions contain other recurring sub-expressions. Also notice that applications can occur in the *procedure* position of an application:

```
> (cse '(+ (* (- x y) (* x x))
   (* x x)
   (foo (- x y))
   (goo (* (- x y) (* x x)))))
(let* ((#:g1414 (- x y))
       (#:g1415 (* x x))
       (#:g1413 (* #:g1414 #:g1415)))
  (+ #:g1413 #:g1415 (foo #:g1414) (goo #:g1413)))

> (cse '(f (g x)
   (g (g x))
   (h (g (g x)) (g x))
   ((g x) (g x))))
(let* ((#:g1421 (g x)) (#:g1422 (g #:g1421)))
  (f #:g1421 #:g1422 (h #:g1422 #:g1421) (#:g1421 #:g1421)))
```

You need to be able to handle `quote` properly. Here are some examples:

```
> (cse '(list (cons 'a 'b)
      (cons 'a 'b)
      (list (cons 'a 'b)
    (cons 'a 'b))
      (list (list (cons 'a 'b)
  (cons 'a 'b)))))
(let* ((#:g1429 (cons 'a 'b))
       (#:g1430 (list #:g1429 #:g1429)))
  (list #:g1429 #:g1429 #:g1430 (list #:g1430)))

> (cse '(list '(a b)
      (list '(a b) '(c d))
      (list '(a b) '(c d))))
(let* ((#:g1436 (list '(a b) '(c d))))
  (list '(a b) #:g1436 #:g1436))
```

As this last example demonstrates, you are *not* supposed to factor constants, and quoted lists are just that — constants.

Your code for this problem should be placed in a file called `cse.scm`.

5 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:

If you fail to submit `zip` file, if files are missing, if functions don't work as they are supposed to, if the statement asserting authenticity of your work is missing, if your work generates output that is not called for (e.g., because of leftover debug statements), etc., then you're going to get a grade of zero. The graders are instructed not to accept any late corrections or re-submissions under any circumstances.

5.1 A final checklist

1. You placed all the source code for your tag-parser in a file named `compiler.scm`
2. Your tag-parser supports all the core forms
3. Your tag-parser supports all the special forms
4. Your tag-parser supports quasiquoted expressions
5. Your tag-parser runs correctly under Chez Scheme
6. Place your solution for the *common sub-expression elimination* problem in a file named `cse.scm`
7. All files related to this assignment were placed in a folder called `hw2`.
8. You compressed the file `hw2` uzing *zip* compression, and you verified that it uncompresses correctly, building the entire `hw2/` directory tree.
9. You included within the `hw2` directory the `readme.txt` file that contains the following information:
   (a) Your name and ID
   (b) The name and ID of your partner for this assignment, assuming you worked with a partner.
   (c) A statement asserting that the code you are submitting is your own work, that you did not use code found on the internet or given to you by someone other than the teaching staff or your partner for the assignment.
10. You submitted the file `hw2.zip`.