Assignment 3 (Due: Sunday (12:00pm (צהריים), January 8, 2017)

Mayer Goldberg

December 31, 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

- *<2016-12-31 Sat>* Fixed the formatting of one example. Only the formatting has changed, so that the example would look better in the pdf file. The contents is the same.
- *<2016-12-26 Mon>* Changed an example regarding `tc-applic`: An application that does not appear within the body of a `lambda`-expression cannot be in tail-position. In the past, we did otherwise, but this keeps things simpler.

<2016-12-26 Mon> Changed a few `if-3` into `if3` — this is a leftover from previous versions of the tag-parser.

- *<2016-12-21 Wed>* Changed the order of composing the parts of the assignment: `remove-applic-lambda-ni` should now be applied *before* `box-set`.

3 Eliminating nested `define`-expressions

As discussed in class, the `define`-expression can appear in two places in the code: *within* the body of a `lambda`-expression or *outside* it. A `define`-expression that appears within the body of a `lambda`-expression is a *local define*. It's effect is to *extend* the lexical environment. By extension, local `define`-expressions are possible also in syntactic forms that *expand* into `lambda`-expressions: `let`, `let*`, `letrec`. When appearing outside the body of any `lambda`-expression, `define`-expressions are to the *global* environment. Recall that the global environment is going to be implemented very differently from the lexical environment.

Nested `define`-expressions complicate our analysis of lexical addressing, and so, and we would like to replace them with corresponding `letrec`-expressions that do the same thing. This transformation should take place in the *tag parser*.

Nested `define`-expressions can occur **only** at the beginning of the body of `lambda`-expressions. They can occur either as part of an explicit or implicit sequence, i.e., within any number of `begin`-expressions. Here are some examples:

```
(lambda (a b c)
  (define foo (lambda (x) ... ))
  (define goo (lambda (y) ... ))
  (* (foo a)
     (goo (* b c))))

(lambda (a b c)
  (begin
    (define foo (lambda (x) ... ))
    (define goo (lambda (y) ... )))
  (* (foo a)
     (goo (* b c))))
```

```
(lambda (a b c)
  (begin
    (begin
      (define foo (lambda (x) ... ))
      (define goo (lambda (y) ... )))
    (begin
      (define moo (lambda (x) ... ))
      (define poo (lambda (y) ... ))))
  (* (foo (poo (moo a)))
     (goo (* b c))))

(lambda (a b c)
  (begin
    (begin
      (begin
        (define foo (lambda (x) ... ))
        (define goo (lambda (y) ... )))
      (begin
        (define moo (lambda (x) ... ))
        (define poo (lambda (y) ... ))))
    (set! a (* a b c))
    (* (foo (poo (moo a)))
       (goo (* b c)))))
```

As you can see from the examples, the rest of the body of the `lambda`-expression may appear as the rest of the sequences of some `begin`-expression… The only complicated thing in replacing nested `define`-expressions with corresponding `letrec`-expressions is really just this: To separate the body of each `lambda`-expression that contains nested `define`-expressions into two parts:

1. The list of nested definitions, and
2. All the rest.

Your support for nested `define`-expressions will introduce `letrec`-expressions which will be expanded further. Here are some examples of the code you should generate:

```
> (define *test-expr*
    '(define my-even?
       (lambda (e)
 (define even? (lambda (n) (or (zero? n) (odd? (- n 1)))))
 (define odd?
   (lambda (n) (and (positive? n) (even? (- n 1)))))
 (even? e))))

> (parse *test-expr*)
(def (var my-even?)
     (lambda-simple
      (e)
```

```
      (seq ((def (var even?)
 (lambda-simple
  (n)
  (or ((applic (var zero?) ((var n)))
       (applic
(var odd?)
((applic (var -) ((var n) (const 1)))))))))
    (def (var odd?)
 (lambda-simple
  (n)
  (if3 (applic (var positive?) ((var n)))
       (applic
(var even?)
((applic (var -) ((var n) (const 1)))))
       (const #f))))
    (applic (var even?) ((var e)))))))
> (eliminate-nested-defines
   (parse *test-expr*))
(def (var my-even?)
     (lambda-simple
      (e)
      (applic
       (lambda-simple
(even? odd?)
(seq ((set (var even?)
   (lambda-simple
    (n)
    (or ((applic (var zero?) ((var n)))
 (applic
  (var odd?)
  ((applic (var -) ((var n) (const 1)))))))))
      (set (var odd?)
   (lambda-simple
    (n)
    (if3 (applic (var positive?) ((var n)))
 (applic
  (var even?)
  ((applic (var -) ((var n) (const 1)))))
 (const #f))))
      (applic (var even?) ((var e))))))
       ((const #f) (const #f)))))
```

After the change, the remaining `def` records in the AST should be to *free variables* only.

3.1 What to hand in

Starting with the file `compiler.scm` from **hw2**, add this code to the file.

4 Removing redundant applications

If you recall, correctly expanding `letrec`-expressions required us to wrap the body of the `letrec` with a `(let () ... )`. This results in many redundant applications of the form `((lambda () ... ))`. For this part, you shall write the Scheme procedure `remove-applic-lambda-nil`, which removes all such redundant applications of `lambda`-expressions with no arguments. Expressions of the form `((lambda () Expr1))` should be replaced with `Expr2`, where `Expr2` is the result of applying `remove-applic-lambda-nil` to `Expr1`.

Here is an example processing the exapansion of a `letrec`-expression that defines a local, recursive procedure `fact`, and calls it as `(fact 5)`:

```
> (remove-applic-lambda-nil
   '(applic
     (lambda-simple
      (fact)
      (seq ((set (var fact) (box (var fact)))
    (box-set
     (var fact)
     (lambda-simple
      (n)
      (if3 (applic (var zero?) ((var n)))
   (const 1)
   (applic
    (var *)
    ((var n)
     (applic
      (box-get (var fact))
      ((applic (var -) ((var n) (const 1))))))))))
    (applic
     (lambda-simple () (applic (box-get (var fact)) ((const 5))))
     ()))))
     ((const #f))))
(applic
  (lambda-simple
    (fact)
    (seq ((set (var fact) (box (var fact)))
   (box-set
     (var fact)
     (lambda-simple
       (n)
       (if3 (applic (var zero?) ((var n)))
    (const 1)
    (applic
      (var *)
      ((var n)
(applic
  (box-get (var fact))
  ((applic (var -) ((var n) (const 1))))))))))
```

```
   (applic (box-get (var fact)) ((const 5))))))
  ((const #f)))
```

4.1 What to hand in

Add the code to your file `compiler.scm`.

5 Boxing of variables

For this assignment, we are going to box any procedure parameter that meets the following criteria:

- It has a *bound* occurrence in the body of the procedure.
- It is set (via a `set!`-expression) somewhere in the body of the procedure.
- It has a *get*-occurrence somewhere in the body of the procedure. A *get* means that it either appears as a `(pvar name minor)` or `(bvar name major minor)`.

To box a variable `(pvar name minor)`, we must do 3 things:

1. Add the expression `(set (pvar name minor) (box (pvar name minor)))` right under the `lambda`-expression in which it is defined.
2. Replace any *get*-occurances of `v` with `box-get` records. These occurrences can be either as parameters or as bound variables, depending on where the variable occurrence was found.
3. Replace any *set*-occurrences of `v` with `box-set` records. These occurrences can be either as parameters or as bound variables, depending on where the variable occurrence was found.

Consider the following example:

```
> (define *example*
    '(let ((a 0))
       (list
(lambda () a)
(lambda () (set! a (+ a 1)))
(lambda (b) (set! a b)))))
> (parse *example*)
(applic
 (lambda-simple
  (a)
  (applic
   (var list)
   ((lambda-simple () (var a))
    (lambda-simple
     ()
     (set (var a) (applic (var +) ((var a) (const 1)))))
    (lambda-simple (b) (set (var a) (var b))))))
 ((const 0)))
> (eliminate-nested-defines
   (parse *example*))
```

```
(applic
 (lambda-simple
  (a)
  (applic
   (var list)
   ((lambda-simple () (var a))
    (lambda-simple
     ()
     (set (var a) (applic (var +) ((var a) (const 1)))))
    (lambda-simple (b) (set (var a) (var b))))))
 ((const 0)))
> (box-set
   (eliminate-nested-defines
    (parse *example*))))))
(applic
 (lambda-simple
  (a)
  (seq ((set (var a) (box (var a)))
(applic
 (var list)
 ((lambda-simple () (box-get (var a)))
  (lambda-simple
   ()
   (box-set
    (var a)
    (applic (var +) ((box-get (var a)) (const 1)))))
  (lambda-simple (b) (box-set (var a) (var b))))))))
 ((const 0)))
```

Notice that there are some cases where the above recipe will box a variable unnecessarily. It is easy to improve our analysis of which variables to box using some extra data-flow analysis, but we shall refrain from doing this for the sake of simplicity.

5.1 What to hand in

Add the code to your file `compiler.scm`.

6 Annotating Variables with their Lexical address

Write a procedure `pe->lex-pe`, which takes a parsed expression `pe`, and traverses it [deeply], replacing all `var` records with the corresponding `(fvar name)`, `(pvar name minor)`, `(bvar name major minor)`. The algorithm for computing the lexical address was discussed in class. Here are a few examples:

```
> (pe->lex-pe (parse '(x (lambda (x) (x (lambda () (x (lambda () (x x)))))))))
(applic
  (fvar x)
  ((lambda-simple
```

```
     (x)
     (applic
       (pvar x 0)
       ((lambda-simple
  ()
  (applic
    (bvar x 0 0)
    ((lambda-simple
       ()
       (applic (bvar x 1 0) ((bvar x 1 0)))))))))))
> (pe->lex-pe (parse '(lambda (a b) (lambda (c) (+ a b c)))))
(lambda-simple
  (a b)
  (lambda-simple
    (c)
    (applic (fvar +) ((bvar a 0 0) (bvar b 0 1) (pvar c 0)))))
> (pe->lex-pe (parse '(define fact (lambda (n) (if (zero? n) 1 (* n (fact (- n 1))))))))
(def (fvar fact)
     (lambda-simple
       (n)
       (if3 (applic (fvar zero?) ((pvar n 0)))
    (const 1)
    (applic
      (fvar *)
      ((pvar n 0)
(applic
  (fvar fact)
  ((applic (fvar -) ((pvar n 0) (const 1)))))))))
```

6.1 What to hand in

Add the code to your file `compiler.scm`.

7 Annotating tail calls

Write a procedure `annotate-tc`, which takes a parsed expression `pe`, and traverses it [deeply], replacing all `applic` records with the corresponding `tc-applic` records. The algorithm was discussed in class. Examples will appear on the website. In the meantime, here are some to keep you going:

```
> (annotate-tc (parse '(lambda (x) (x x))))
(lambda-simple (x) (tc-applic (var x) ((var x))))
> (annotate-tc (parse '(define fact (lambda (n) (if (zero? n) 1 (* n (fact (- n 1))))))))
(def (var fact)
     (lambda-simple
       (n)
       (if3 (applic (var zero?) ((var n)))
    (const 1)
    (tc-applic
```

```
      (var *)
      ((var n)
(applic
  (var fact)
  ((applic (var -) ((var n) (const 1)))))))))
> (annotate-tc (parse '(x (lambda (x) (x (lambda () (x (lambda () (x x)))))))))
(applic
  (var x)
  ((lambda-simple
     (x)
     (tc-applic
       (var x)
       ((lambda-simple
  ()
  (tc-applic
    (var x)
    ((lambda-simple () (tc-applic (var x) ((var x))))))))))))
> (annotate-tc
    (parse
      '(lambda (f)
 ((lambda (x) (f (lambda s (apply (x x) s))))
  (lambda (x) (f (lambda s (apply (x x) s)))))))
(lambda-simple
  (f)
  (tc-applic
    (lambda-simple
      (x)
      (tc-applic
(var f)
((lambda-var
   s
   (tc-applic
     (var apply)
     ((applic (var x) ((var x))) (var s)))))))
    ((lambda-simple
       (x)
       (tc-applic
 (var f)
 ((lambda-var
    s
    (tc-applic
      (var apply)
      ((applic (var x) ((var x))) (var s)))))))))
```

7.1 What to hand in

Add the code to your file `compiler.scm`.

8 How we shall test your assignment

When components of the assignment can be tested separately, we will try to have at least some of the tests run on individual components. Most components, however, must be run in *cascade*, so we shall need to compose individual components to run them. The order of the components is important. We will be composing the components in the order they appear in this document:

1. `eliminate-nested-defines`
2. `remove-applic-lambda-nil`
3. `box-set`
4. `pe->lex-pe`
5. `annotate-tc`

with `eliminate-nested-defines` being called *first* to process parsed expressions, and `annotate-tc` being called *last*. It is possible to write a perfect compiler while following a different order, however for the purpose of this assignment we ask you to follow the order we specify, lest you fail the automatic testing.

9 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.

9.1 A final checklist

1. You placed all the source code for your tag-parser in a file named `compiler.scm`
2. All files related to this assignment were placed in a folder called `hw3`.
3. You compressed the file `hw3` uzing *zip* compression, and you verified that it uncompresses correctly, building the entire `hw3/` directory tree.
4. You included within the `hw3` 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.

5. You submitted the file `hw3.zip`.