Assignment 3 (Due: Wednesday (12:00pm (צהריים), December 27th, 2017)

Mayer Goldberg

December 13, 2017

Contents

1 General 1
2 Removing redundant applications 2
3 Boxing of variables 3
4 Annotating Variables with their Lexical address 4
5 Annotating tail calls 5
6 How we shall test your assignment 6
7 Submission Guidelines 7
  7.1 Creating a patch file 8
8 A word of advice 8
  8.1 A final checklist 9

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.
• Please read this document completely, from start to finish, before beginning work on the assignment.

2 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 expansion 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)))))
                      (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)
                              )))))
            (const #f))))
  ))}
```
3 Boxing of variables

For this assignment, we are going to box any procedure parameter that meets all of 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-occurrences 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.

4 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)
5 Annotating tail calls

Write a procedure annotate-tc, which takes a parsed expression pe, and traverses it [deeply], re-placing all applic records with the corresponding tc-applic records. The algorithm was discussed in class. Here are some examples:

> (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))))))))
(define (var fact)
  (lambda-simple
    (n)
    (if3 (applic (var zero?) ((var n)))
       (const 1)
       (aplic
         (var *)
         ((var n) 0))
     (applic
      (var fact)
      ((applic (var -) ((var n) 0) (const 1)))))))

> (annotate-tc (parse '(x (lambda (x) (x (lambda () (x (lambda () (x x))))))))
(applic
 (var fact)
 ((applic (var -) ((var n) (const 1))))))

> (annotate-tc (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))))))

> (annotate-tc (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))))))

> (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))))))))
(define (var fact)
  (lambda-simple
    (n)
    (if3 (applic (var zero?) ((var n)))
       (const 1)
       (aplic
         (var *)
         ((var n) 0))
     (applic
      (var fact)
      ((applic (var -) ((var n) 0) (const 1)))))))

> (annotate-tc (parse '(x (lambda (x) (x (lambda () (x (lambda () (x x))))))))
(applic
 (var fact)
 ((applic (var -) ((var n) (const 1)))))))
(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))))))))))

6 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. remove-applic-lambda-nil
2. box-set
3. pe->lex-pe
4. annotate-tc

with remove-applic-lambda-nil 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.

7 Submission Guidelines

In this course, we use the git DVCS for assignment publishing and submission. You can find more information on git at [https://git-scm.com/](https://git-scm.com/).

To begin your work, clone the assignment template from the course website:

git clone https://www.cs.bgu.ac.il/~comp181/assignments/3

This will create a copy of the assignment template folder, named 3, in your local directory. The template contains two (2) files:

- semantic-analyzer.scm (the assignment interface)
- readme.txt

The file semantic-analyzer.scm is the interface file for your assignment. The definitions in this file will be used to test your code. If you make breaking changes to these definitions, we will be unable to test and grade your assignment. Do not break the interface. Operations which are considered interface-breaking:

- Modifying the 1st line: (define remove-applic-lambda-nil
- Modifying the 5th line: (define box-set
- Modifying the 9th line: (define pe->lex-pe
- Modifying the 13th line: (define annotate-tc

Other than breaking the interface, you are allowed to add any code and/or files you like. Among the files you are required to edit is the file readme.txt. The file readme.txt should contain:

1. The names and IDs of all the people who worked on this assignment. There should be either your own name, or your name and that of your partner. You may only have one partner for this assignment.

2. The following statement:

I (We) assert that the work we submitted is 100% our own. We have not received any part from any other student in the class, nor have we give parts of it for use to others. Nor have we used code from other sources: Courses taught previously at this university, courses taught at other universities, various bits of code found on the internet, etc. We realize that should our code be found to contain code from other sources, that a formal case shall be opened against us with ,in pursuit of disciplinary action.
Submissions are only allowed through the submission system. You are required to submit a patch file of the changes you made to the assignment template. See instructions on how to create a patch file below.

Please be careful to check your work multiple times. Because of the size of the class, we cannot handle appeals to recheck your work in case you forget or fail to follow any instructions precisely. Specifically, before you submit your final version, please take the time to make sure your code loads and runs properly in a fresh Scheme session.

### 7.1 Creating a patch file

Before creating the patch review the change set and make sure it contains all the changes that you applied and noting more. Modified files are automatically detected by git but new files must be added explicitly with the ‘git add’ command:

```
    git add -Av .; git commit -m "write a commit message"
```

At this point you may review all the changes you made (the patch):

```
    git diff origin
```

Once you are ready to create a patch for submission, simply make sure the output is redirected to the patch file:

```
    git diff origin > assignment3.patch
```

After submission (but before the deadline), it is strongly recommended that you download, apply and test your submitted patch file. Assuming you download `assignment3.patch` to your home directory, this can be done in the following manner:

```
    cd ~
    git clone https://www.cs.bgu.ac.il/~comp181/assignments/3 fresh_assignment3
    cd fresh_assignment3
    git apply ~/assignment3.patch
```

Then test the result in the directory `fresh_assignment3`.

Finally, remember that your work will be tested on lab computers only! We advise you to test your code on lab computers prior to submission!

### 8 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 a patch file, if files your work depends on 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.
8.1 A final checklist

1. You completed the four (4) required functions in the `semantic-analyzer.scm` file.

2. You did not break the interface.

3. Your semantic analyzer runs correctly under Chez Scheme on the departmental Linux image.

4. You completed 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 committed your work in its entirety.

6. You created and submitted a patch file containing your work in the submission system.