CSSE 304 Assignment 10

No input error-checking is required. You may assume that all arguments have the correct form.
Abbreviations for the textbook: EoPL - *Essentials of Programming Languages*, 3rd Edition.
Section 1.2.4 is especially relevant to this assignment.

Mutation is not allowed for this assignment.

#1 (15 points) `free-vars`, `bound-vars`. LCExp is defined by a grammar on page 9 of EoPL. Given a LcExp *e*, `(free-vars e)` returns the set of all variables that occur free in e. `bound-vars` is similar. Write these procedures directly; do not use `occurs-free` or `occurs-bound` in your definitions. Your code only needs to process the simple lambda-calculus expressions from the grammar from EoPL, not the extended expressions from problem 3 and 4 of this assignment. By *set*, we mean a list of symbols with no duplicates, as in previous assignments. The order of the symbols in the return value does not matter.

```
> (free-vars   '((lambda (x) (x y)) (z (lambda (y) (z y)))))
(y z)
> (bound-vars '((lambda (x) (x y)) (z (lambda (y) (z z)))))
(x)
```

#2 (40 points) Expand `occurs-free?` and `occurs-bound?` (written in class and in the textbook for basic lambda-calculus expressions) to incorporate the following language features into your code. You can find the original `occurs-free?` and `occurs-bound?` from the textbook at

http://www.rose-hulman.edu/class/csse/csse304/202110/Resources/Code-from-Textbook/1.scm

a) Scheme `lambda` expressions (abstractions) may now have more than one (or zero) parameters, and Scheme procedure calls (applications) may have more than one (or zero) arguments. Modify the formal definitions of `occurs-free?` and `occurs-bound?` to allow `lambda` expressions with any number of parameters and procedure calls with any number of arguments. Then modify the procedures `occurs-free?` and `occurs-bound?` to include these new definitions.

b) Extend the formal definitions of `occurs-free?` and `occurs-bound?` to include `if` expressions, and implement these in your code. You are only required to handle "two-armed" `if` expressions that have both a "then" part and an "else" part.

c) Extend the formal definitions of `occurs-free?` and `occurs-bound?` to include Scheme `let` and `let*` expressions (you are not required to do "named let"), and implement these in your code.

d) Extend the formal definitions of `occurs-free?` and `occurs-bound?` to include Scheme `set!` expressions, and implement these in your code. Note that `set!` does not bind any variables.

```
(occurs-bound? 'x '(lambda (y) (set! x y)))                ➔ #f
(occurs-free?  'y '(lambda (x a b) y))                     ➔ #t
(occurs-free? 'b '(let* ((y a) (x b)) ((x y) z)))          ➔ #t
(occurs-free? 'set! '(lambda (x) (set! x y)))              ➔ #f   ; set! is Scheme syntax, not a variable
(occurs-bound? 'z '(lambda () (let* ((x a) (y x)) (if (y z) (lambda () x) (lambda () y))))) ➔ #f
```

See the test cases for additional examples.

Assignment continues on the next page

#3 (30 points). `lexical-address`. Write a procedure `lexical-address` that takes an expression like those from the previous problem (except that you are not required to do `let*` expressions for this problem) and returns a copy of the expression with every bound occurrence of a variable `v` replaced by a list `(: d p)`. The two numbers `d` and `p` are the lexical depth and position of that variable occurrence. If the variable occurrence `v` is free, produce the following list instead: `(: free xyz)` To produce the symbols `:` and `free`, use the code `':` and `'free`.

Hint: It may be easiest to do this with a recursive helper procedure that keeps track of bound variables and their levels as it descends into various levels of the expression. Note that this is very similar to the `depth` variable that we used in writing the `notate-depth` procedure during the live coding on Day 8.

Examples:

```
(lexical-address '(lambda (a b c)
                    (if (eq? b c)
                        ((lambda (c)
                           (cons a c))
                         a)
                        b)))                    ➔
(lambda (a b c)
  (if ((: free eq?) (: 0 1) (: 0 2))
      ((lambda (c) ((: free cons) (: 1 0) (: 0 0)))
       (: 0 0))
      (: 0 1)))
```

```
(lexical-address
 '((lambda (x y)
     (((lambda (z)
         (lambda (w y)
           (+ x z w y)))
       (list w x y z))
      (+ x y z)))
   (y z)))              ➔

((lambda (x y)
   (((lambda (z)
       (lambda (w y)
         ((: free +) (: 2 0) (: 1 0) (: 0 0) (: 0 1))))
     ((: free list) (: free w) (: 0 0) (: 0 1) (: free z)))
    ((: free +) (: 0 0) (: 0 1) (: free z))))
 ((: free y) (: free z)))
```

```
(lexical-address
 '(lambda (a b c)
    (if (eq? b c)
        ((lambda (c) (cons a c))
         a)
        b)))        ➔

(lambda (a b c)
  (if ((: free eq?)(: 0 1) (: 0 2))
      ((lambda (c) ((: free cons) (: 1 0) (: 0 0)))
       (: 0 0))
      (: 0 1)))
```

```
(lexical-address
  '(let ([a 3] [b 4])
    (let ([a (+ b 2)] [c a])
      (+ a b c))))      ➔

(let ((a 3) (b 4))
  (let ((a ((: free +) (: 0 1) 2))
        (c (: 0 0)))
    ((: free +) (: 0 0) (: 1 1) (: 0 1))))
```

#4 (30 points). `un-lexical-address`. Its input will be in the form of the output from `lexical-address`, as described in the previous problem. In the test-cases, we will evaluate

`(un-lexical-address (lexical-address <`*some-expression*`>))`

and test whether this returns something that is `equal?` to the original expression. You cannot get any credit for this problem unless you also get a significant number of the points for `lexical-address`. [For example, someone who defines both

`lexical-address` and `un-lexical-address` to be the identity procedure will trick the grading program into giving them full credit for `un-lexical-address`, but we will assign zero points as their actual grade for both problems after we look at the code by hand.]

Note: `lexical-address` is harder than `un-lexical-address`; if there are errors in your `lexical-address` code, they will most likely be discovered when you test `un-lexical-address`.

Hint Copied from Piazza

A10 lexical-address hint

I gave this hint verbally in class on both days when we discussed `lexical-address`, but that was a long time ago, so I am reminding you now and giving you a little bit more detail.

`lexical-address` and `un-lexical-address` will each need to have a recursive helper procedure. Each of these procedures will have a parameter that is the current "scope-list". It will be a list of lists of variables, the variables bound by the `lambda`s and `let`s that the current expression is inside of.

For example, consider `(lambda (x y) (lambda (y z) (y (+ x z))))`, When your lexical-address code does the recursive call for the expression `(+ x z)`, the scope-list might be `((y z) (x y))`. A separate "lookup" procedure can use the scope-list to find the lexical depth and position for each local variable. It can also determine that `+` is a free variable, because `+` is not in the scope-list.

When the recursive call is for a non-binding expression (such as `if` or a procedure application), it passes the scope-list unchanged. When it is the body of a `let` or `lambda`, it passes in an expanded scope-list that includes the new bound variables.

`trace` and `trace-lambda` are your friends!

A10-#2

When it says update the formal definition for occurs-free? and occurs-bound?, do you want us to write these new definition in as comments or are these sort of just do on your own parts?

hw10
edit · good question 0

Updated 5 months ago by Ben Brubaker

the instructors' answer,
where instructors collectively construct a single answer
You do not need to submit the grammar. Just use it as a guideline as you write your code.

A10- #2 ifs?

I'm a bit confused on how an identifier is free or bound in an if. I assumed it would be whatever the output of the if statement would be, but that doesn't make sense as you can't know the output of it given random variables as the check. Is an identifier free/bound in an if statement as long as it is free/bound in either the check/then/else?

hw10
edit · good question 0

Updated 5 months ago by Ben Brubaker

the instructors' answer,
where instructors collectively construct a single answer
Yes. If is not a binding construct. Neither are cond, and, or, set!. But let, let*, and letrec are.

A#10-3 Square brackets?

I have my lexical-address procedure mostly working, except I can't figure out how to get it to make square brackets for processing the let cases, and from testing in repl (eq? '(a (: free b)) '[a (: free b)]) returns #f, so I would assume we can't just replace them with parentheses.

EDIT: I ended up getting it fully implemented anyway, and for some reason my code generated the square brackets. Not really sure why as I was just using list... But it put them there.

hw10
edit · good question 0

Updated 5 months ago by Ben Brubaker

the students' answer,
where students collectively construct a single answer
Did you test if it passed the tests? I think the test cases don't distinguish between brackets and parentheses, so if you just keep the list the same way it was passed into your function, it should work.

I'm just guessing, haven't actually looked at #3 yet.
edit · good answer 1

Updated 5 months ago by Joseph Brown

the instructors' answer,
where instructors collectively construct a single answer
Square brackets are merely an alternative for input into Scheme. When you use them, they produce the same code/data internally as if you had used round brackets (parentheses). Scheme never prints square brackets when it outputs data.

Joe is right; if this one passes the tests, you should be fine.

A10 #4. Can I use a stack if it says no mutation for the assignment?

I'm working on the 4th problem of assignment 10 and trying to implement it with a stack but then I realize there's mutation in stack. So are we allowed to use stack on this one?

the instructors' answer,
You are not allowed to use a stack **object** for that problem. Using a stack object would count as mutation. But you can pass a list to your recursive helper procedure that is treated as a stack, but without mutation.

A10 Test Case Occurs-Free?

I'm not understanding why the test case (occurs-free? (quote y) (quote (let ((y ((lambda (x) (+ x y)) z))) (+ y y)))) returns true? Isn't y being bound to the lambda expression? So then wouldn't it not be free?

the instructors' answer,
where instructors collectively construct a single answer

y occurs free in the expression `(+ x y)`. Since this is a normal let, that y is not bound to anything shown here.

HW 10: Question about set! in occurs-free.

A student wrote:
For problem 2, occurs-free? on the grading server, there is a test for the 'set!' component of occurs-free? which I believe has an error.
In the assignment directions, it is noted, that set! does not bind any variables. However, there is a test case such that:

(occurs-free? 'x '(set! x y))

which expects
#f

Considering that set! does not bind any variables, this test is supposed to be expecting #t.

My answer:
Your statement is correct, `set!` does not bind anything.
But x does not occur free in this expression because x does not occur at all.

Does occurs-bound? need defined behavior for set?

I've got my occurs-free? and occurs-bound? procedures passing all of the test cases, but I haven't actually written anything related to set! in the occurs-bound? procedure. It just treats (set! x y) like an application with 3 arguments, causing it to correctly determine that nothing is bound in that expression. Do I need to specifically define occurs-bound? behavior for a set! expression, and if so, is there a test case that would demonstrate why my current approach is incorrect?

the instructors' answer,
where instructors collectively construct a single answer
Actions
There are tests for this in occurs-free?
I may not have tested for it for occurs-bound?

Here's how it should work:
In `(set! z (lambda (x) y))`
y occurs free. x and z do not occur.
In `(set! x (+ 1 x))`
x occurs free.

Set! Test Clarification

Is the following test asking if the variable (quote set!) is free?
The test asserts the answer is false, so I must not be understanding what this test means when it has (quote set!) as the variable.

(occurs-free? (quote set!) (quote (lambda (x) (set! x y))))

the instructors' answer,
where instructors collectively construct a single answer
Actions
In the context (set! x y), set! is syntax, not a variable occurrence.

letrec

In the specifications it says we do not have to worry about named lets, but it never specified in either direction if we need to worry about letrec, and there are no tests for it, so can we safely ignore it?

the instructors' answer,
where instructors collectively construct a single answer

If there are no tests, you can ignore it for now

If statements, free, and bound variables

Are variables in an IF statement bound?

For example, in the following code, are y and z bound variables?

```
(if (y z)
   (lambda () x)
   (lambda () y))
```

the students' answer,

Since IF isn't a binding construct, y and z should be free variables.

~ An instructor (Claude Anderson) endorsed this answer ~

Why does x not occur free in (set! x y)?

The second-to-last test case in "test-occurs-free?" asserts that (occurs-free? 'x '(set! x y)) should return false. Why is it so?

the students' answer,

set! sets the value of the first arg to the value of the second arg. So x is a bound variable whose value is being changed.

the instructors' answer,

In this example, y occurs free, and x does not occur. Recall that an "occurrence" of a variable in the syntax we are considering is like a "use" of the variable in Scheme.

Understanding bound/free defaults

Why does y occur free in the following? Isn't y bound in the let?

```
 (occurs-free? (quote y)
               (quote (let ((y ((lambda (x) (+ x y)) z))) (+ y y))))
```

the students' answer,

where students collectively construct a single answer

The y in (lambda (x) (+ x y)) is free.

the instructors' answer,

where instructors collectively construct a single answer

That occurrence of y is free because in a let, the expressions that provide the initial values for the let variables are evaluated in the outer scope, outside the let scope.

followup discussions

for lingering questions and comments

In this case, x occurs bound in the (x a), but free in the (y x). Is the overall value false because the (y x) does not count as free because in let* the declarations "fall down" and x has already been bound in the previous (x a)?

(occurs-free? (quote x) (quote (lambda () (let* ((x a) (y x)) (if (y z) (lambda () x) (lambda () y))))))

That was poorly worded, but what I'm getting at is, occurs-free doesn't get "over ridden" if the variable also occurs bound?

A student answer

In your original example, the variable y both occurs bound and occurs free, because in some instances of y its value is defined outside of the code snippet, and in other instances it's defined inside the code snippet. These two variables ("global" y and "local" y) happen to have the same name, but they can stand for totally different things.

Occurs free? let with multiple expressions.

Do we have to account for let expressions with multiple expressions?

i.e (let ([x a]) (+ x x) (- x x))?

hw10

edit · good question 0

Updated 2 months ago by

Eric Tu

the instructors' answer,

where instructors collectively construct a single answer

Actions

It appears that none of the A10 test cases account for the multiple bodies case of let or lambda, so technically the answer is "no". But you'll definitely need that case for later assignments, so it wouldn't be a bad idea to include it now.