EECS 490 Midterm Exam, Fall 2017

Time: 75 minutes

- This exam is closed book, closed notebook.
- You may not provide your own scratch paper, but the last two pages are intentionally left blank for scratch work. Please do **not** detach them.
- Laptops, cell phones, calculators, and smartwatches are not allowed.
- Cell phones must be turned off.
- You are allowed one 8.5x11" sheet of notes as a reference.
- Any deviation from these rules will constitute an Honor Code violation.

- The exam consists of 5 questions, each with multiple parts.
- For multiple-choice and true/false questions, indicate your choice by filling in the appropriate square or bubble. Fill in the square or bubble completeley, and make sure to erase completely if you change your answer.
- For short-answer and coding questions, write your answers clearly in the space provided.
- Do **not** write in the margins of a page (i.e. within an inch of the border), as anything written there will not show up when your exam is scanned.
- Assume all Python code is Python 3.6, all C++ code is C++14, and all Scheme code is R5RS. Adhere to these standards in the code you write as well.
- Your exam should have 12 pages, including this page and the two blank pages at the end.

Include your signature to indicate that you acknowledge the Honor Pledge, printed below.

I have neither given nor received aid on this exam, nor have I concealed any violations of the Honor Code.

Name Solution

Uniqname ______________________

UMID ______________________

Signature ______________________

Name of person to your left ______________________

Name of person to your right ______________________

Question 1 (20 points)

Select the correct answers for each problem below from choices A-E by filling in the square next to each choice you select. There may be more than one correct answer for each problem; select all that are correct.

a) Which of the following statements are true?

A. ■ The grammar of a language determines which phrases have the correct structure.

B. □ The lexical structure defines how tokens of a programing language can be combined to form phrases that have the correct structure.

C. □ The pragmatics of a programming language determine how the actions specified by a meaningful phrase are accomplished.

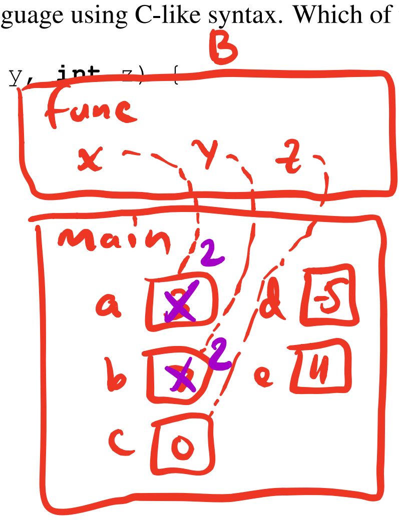
D. ■ Semantics specify the meaning of a correct phrase in a language.

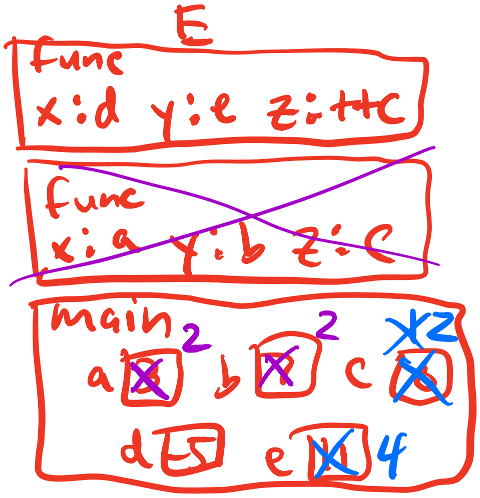
E. ■ An interpreter performs lexical analysis on a program before it analyzes the program's syntax.

b) Consider the following code in a language using C-like syntax. Which of the following statements are true?

```
void func(int x, int y, int z) {
  x += (z - 1);
  y = (z + x);
}

int main() {
  int a = 3;
  int b = 7;
  int c = 0;
  int d = -5;
  int e = 11;
  func(a, b, c);
  func(d, e, ++c);
}
```





A. ■ Under the call-by-value convention, a would have the value 3 and b the value 7 after the first call to func() completes.

B. □ Under call by reference, a would have the value 2 and b the value 3 after the first call to func() completes.

C. □ Under call by value, the second call to func() would result in a compilation error.

D. □ Under call by name, the second call to func() would result in a compilation error.

E. ■ Under call by name, c would have the value 2 after the second call to func() completes.

↑

We gave everyone credit for E., since whether or not a call-by-name argument is evaluated more than once is language dependent.

c) Which of the following regular expressions match exactly the strings that start with any number of `0`'s, followed by zero or one `1`, followed by one or more `2`'s?

A. □ `[012]*`

B. □ `0*1+2*`

C. ☒ `0*(2|12)2*`

D. □ `0*(02|012)2*`

E. ☒ `0*(12+|2+)2*`

d) Consider the following Scheme code:

```
(let* ((x 3)
       (y (+ x 1)))
  (cons x y)
)
```

For each of the following code fragments, fill in the bubble if the fragment has the same behavior as the code above when run in an environment with no user-created bindings:

A. □

```
(let ((x 3)
      (y (+ x 1)))
  (cons x y)
)
```

B. ☒

```
(let ((x 3))
  (let ((y (+ x 1)))
    (cons x y)
  )
)
```

C. □

```
((lambda (x y)
    (cons x y)
 )
 (+ x 1)
 3
)
```

D. ☒

```
((lambda (x)
   ((lambda (y)
      (cons x y)
    )
    (+ x 1)
   )
 )
 3
)
```

E. ☒

```
(let ((x 3))
  (define y (+ x 1))
  (cons x y)
)
```

Question 2 (20 points)

For each statement below, fill in the bubble to the left of **True** or **False**, or leave them both blank. Unanswered statements will receive 50% credit.

a) ●**True** / ○**False** First-class continuations can be used to build an error-handling facility that provides the same capabilities as exceptions in C++.

b) ○**True** / ●**False** Capturing non-local variables by reference in C++ is always an error when defining an anonymous function.

c) ○**True** / ●**False** In lambda calculus, when a function is applied to an argument, the argument must be evaluated before it is substituted into the body of the function.

d) ○**True** / ●**False** In R5RS Scheme, the non-local environment of a function call is the environment that was active when the function was invoked.

e) ●**True** / ○**False** In a derivation tree for a string from a context-free grammar, the in-order traversal of the leaves of the tree recovers the input string.

f) ○**True** / ●**False** The output of a parser is a plain-text representation of the input code.

g) ●**True** / ○**False** A tracing garbage collector traverses an object graph starting at some root set of live objects.

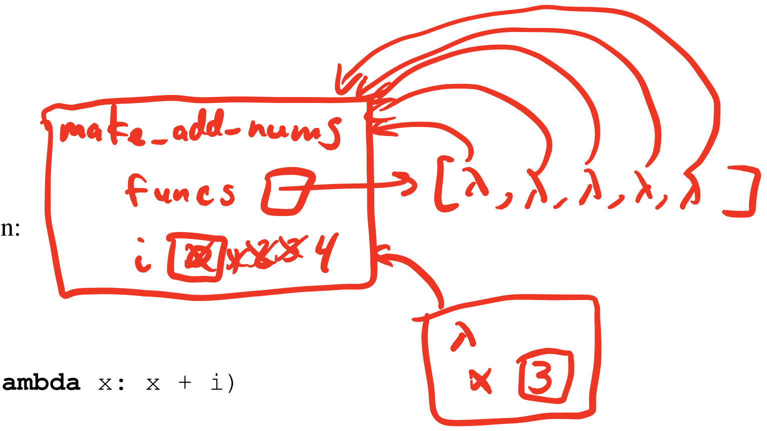
h) ●**True** / ○**False** In Scheme, it is legal for a `define` expression to appear at the beginning of the body of a `lambda` function.

i) ●**True** / ○**False** In Python, types are first-class entities that can be created at runtime.

j) ○**True** / ●**False** In Python, the body of a loop is an inline block that has its own scope that is separate from that of the function containing the loop.

Question 3 (30 points)

a) Consider the following Python function:



```
def make_add_nums(n):
    funcs = []
    for i in range(n):
        funcs.append(lambda x: x + i)
    return funcs
```

For each of the following function calls, write the value that is returned by the call in the box below the call:

```
>>> funcs = make_add_nums(5)
>>> funcs[0](3)
```

7

```
>>> funcs[1](3)
```

7

```
>>> funcs[4](3)
```

7

b) Evaluate the λ-calculus term below until it is in normal form, using the normal-order evaluation strategy. Show each α-reduction and β-reduction step, as in the following:

$$\begin{aligned} & (\lambda x.\, x)\ (\lambda x.\, x) \\ \to_\alpha\ & (\lambda x.\, x)\ (\lambda y.\, y) \\ \to_\beta\ & \lambda y.\, y \end{aligned}$$

Term to evaluate:

$$(\lambda w.\, \lambda x.\, x\ w)\ (\lambda y.\, y)\ \lambda z.\, z\ z$$

$\to_\beta\ (\lambda x.\, x\ \lambda y.y)\ \lambda z.\, z\ z$

$\to_\beta\ (\lambda z.\, z\ z)\ \lambda y.\, y$

$\to_\beta\ (\lambda y.\, y)\ \lambda y.\, y$

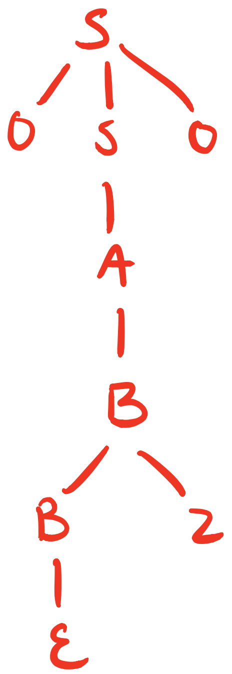
$\to_\alpha\ (\lambda y.\, y)\ \lambda x.\, x$

$\to_\beta\ \lambda x.\, x$

c) Consider the following context-free grammar, with start symbol S:

$$\begin{aligned} S &\rightarrow 0\,S\,0 \mid A \\ A &\rightarrow 1\,A\,1 \mid B \\ B &\rightarrow B\,2 \mid \varepsilon \end{aligned}$$

i. Draw the derivation tree for the string: 020



ii. Describe in no more than two sentences the set of strings that are matched by the grammar.

Strings of the form $0^k 1^m 2^n 1^m 0^k$, $k, m, n \geq 0$. In other words, strings with any number of 2's in the middle, surrounded by an equal number of 1's to the left and right, in turn surrounded by an equal number of 0's.

d) Consider the following implementation of a parser in Scheme. Assume that the `error` procedure is defined as in the distribution code for Project 2:

```
(define (parse)
  (list->string (parse-A))
)

(define (parse-A)
  (let ((next-char (peek-char)))
    (cond ((char=? next-char #\a)
           (read-char)
           (cons #\a (parse-A)))
          (else (parse-B))
    )
  )
)

(define (parse-B)
  (let ((first-char (read-char)))
    (if (char=? first-char #\b)
        (cons #\b (parse-C))
        (error "illegal character")
    )
  )
)

(define (parse-C)
  (let ((next-char (peek-char)))
    (cond ((char=? next-char #\c)
           (read-char)
           (cons #\c (parse-C)))
          (else '())
    )
  )
)
```

i. For each of the following inputs on standard input, determine what will be the return value of the call `(parse)`. If parsing results in an error, write `error`.

Input	Result
abc	"abc"
b	"b"
ac	error
bcd	"bc"
abcd	"abc"
aabbcc	"aab"
aabcdd	"aabc"

ii. Describe in no more than two sentences the set of inputs that are accepted by the parser.

Strings with any number of a's, followed by one b, followed by any number of c's.

Question 4 (15 points)

Recall the definition of Church numerals in λ-calculus:

$$\begin{aligned} zero &= \lambda f.\, \lambda x.\, x \\ one &= \lambda f.\, \lambda x.\, f\, x \\ two &= \lambda f.\, \lambda x.\, f\,(f\, x) \\ &\dots \end{aligned}$$

A number n is defined as a higher-order function that, given another function f, produces a new function that applies f to its argument n times. We can similarly define the Church numerals in Python:

```
zero = lambda f: lambda x: x
one = lambda f: lambda x: f(x)
two = lambda f: lambda x: f(f(x))
...
```

a) Define a function `print_arg()` such that `zero(print_arg)(x)` prints nothing, `one(print_arg)(x)` prints the value of `x` once on its own line, `two(print_arg)(x)` prints the value of `x` twice, each on its own line, and so on.

```
def print_arg(x):
    """Prints the value of x on its own line.

    >>> one = lambda f: lambda x: f(x)
    >>> two = lambda f: lambda x: f(f(x))
    >>> _ = one(print_arg)(3)  # assignment to _ suppresses return value
    3
    >>> _ = two(print_arg)(3)  # assignment to _ suppresses return value
    3
    3
    """
    # your code below
    print(x)
    return x
```

b) Define the `incr()` function, which given a Church numeral as the argument, returns the numeral that represents one more than the argument.

```
def incr(n):
    """Returns the Church numeral that represents one more than the
    Church numeral n.

    >>> one = lambda f: lambda x: f(x)
    >>> _ = incr(one)(print_arg)(3)
    3
    3
    """
    # your code below
    return lambda f: lambda x: f(n(f)(x))
```

c) Write the `deep_iterate()` generator below, which given a nested structure of lists, produces each of the non-list items in the structure in order. You may use `instanceof(x, list)` to determine if `x` is a list. Your solution may **not** construct any new lists, such as by slicing an existing list.

```
def deep_iterate(nested_items):
    """Produces a generator that iterates over the items
    contained in the given nested-list structure, in
    left-to-right order. If an item is a list, its elements are
    themselves iterated over. Otherwise, the item is yielded by
    the generator.

    >>> nested_lists = [[3, [4]], [2], [], 'a']
    >>> iter = deep_iterate(nested_lists)
    >>> next(iter)
    3
    >>> next(iter)
    4
    >>> next(iter)
    2
    >>> next(iter)
    'a'
    """
    # your code below
    for elem in nested_items:
        if isinstance(elem, list):
            for item in deep_iterate(elem):
                yield item
        else:
            yield elem
```

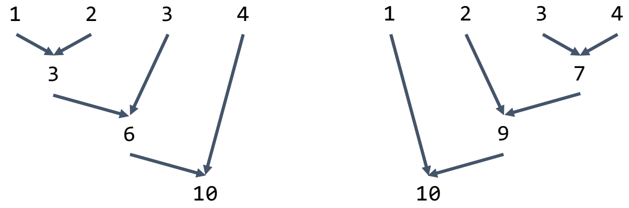
Can replace this with:

```
yield from deep_iterate(elem)
```

Question 5 (15 points)

For this question, you must write your code in R5RS Scheme. You may only use the following special forms and primitive procedures: `define` at global scope, `let`, `let*`, `lambda`, `if`, `cond`, `and`, `or`, `quote`, `null?`, `not`, `cons`, `car`, `cdr`, `list`, `append`. Your functions do **not** have to be tail recursive.

A *reduction* combines elements of an input sequence using a binary function. It proceeds by combining two elements of the sequence, then combining the results of that with another element, and so on, until all elements have been combined into a single result. A reduction can be left or right associative, as in the following for addition:



Write a definition of the `reduce-left` procedure below to implement a left-associative reduction. The following are examples comparing `reduce-right` and `reduce-left`, where `expt` is the built-in exponentiation procedure:

```
> (reduce-right - '(1 2 3 4))     # 1-(2-(3-4))
-2
> (reduce-left - '(1 2 3 4))      # ((1-2)-3)-4
-8
> (reduce-right expt '(2 3 2))    # 2^(3^2)
512
> (reduce-left expt '(2 3 2))     # (2^3)^2
64
```

You may assume that the input sequence is a null-terminated, non-empty list of items of the same type. You may assume that the input function takes in two arguments of that type and returns a result of the same type. You may define a helper function if you wish. Write your code below:

```
(define (reduce-left fn items)
  (reduce-left-helper fn (car items) (cdr items))
)

(define (reduce-left-helper fn result items)
    (if (null? items)
        result
        (reduce-left-helper
            fn
            (fn result (car items))
            (cdr items)
        )
    )
)
```

This page intentionally left blank. We will not grade any work here.

Alternate Solution :

```
(define (reduce-left fn items)
    (if (null? (cdr items))
        (car items)
        (reduce-left
            fn
            (cons (fn (car items) (car (cdr items)))
                  (cdr (cdr items)))
            )
        )
    )
)
```

This page intentionally left blank. We will not grade any work here.