
This is a 75 minute, CLOSED notes, books, etc. exam.
ASK if anything is not clear.
WORK INDIVIDUALLY.
Strategy: Scan the entire exam first. Work on the easier ones before the harder ones. Don't waste too much time on any one problem. Show all work on the space provided. Write your name on each page. Check to make sure you have 6 pages.

You may not use `set!` in your Racket programs unless otherwise indicated.

Question	Points	Score
1	5	
2	5	
3	5	
4	5	
5	5	
6	10	
7	15	
8	10	
9	10	
10	10	
11	10	
12	10	
Total:	100	

1. (5 points) Select **all** of the following true statements about macros.
   A. Text substitution macros are powerful, but may result in inadvertent variable capture.
   B. Syntactic macros work by restructuring abstract syntax trees.
   C. Macros in C are hygienic.
   D. Syntactic macros work at the "preprocessor" stage, before tokenization.
   E. Macros are not used in Racket, since it is easy to restructure lists without relying on macros.

2. (5 points) Select **all** of the following true statements about Louden & Lambert's language design criteria.
   A. Macros in Racket are an example of *extensibility*.
   B. A language with poor *regularity* might either make similar things look different, or make different things look similar.
   C. The design goal of *efficiency* refers only to making the programmer more efficient, not to making more efficient compiled code.
   D. C is a good example of a language that provides strong *security*, since it catches errors like writing past the end of an array.
   E. C may be considered to be an efficient language from the perspective of the compiler, since it produces efficient executables, but less efficient from the perspective of the programmer, since fewer details are handled automatically.

3. (5 points) Select **all** of the following true statements about parsers and tokenizers.
   A. The job of a parser is to take in tokens and produce an abstract syntax tree (AST).
   B. Abstract syntax trees are sometimes called "lexemes".
   C. Tokens may be thought of as the "words" of a language.
   D. Once an abstract syntax tree is produced, it is only useful for evaluation by an interpreter; for efficiency, compilers typically work directly from the stream of tokens.
   E. A *transpiler* is a compiler that translates one high-level language to another.

4. (5 points) Select **all** of the following true statements about programming contracts.
   A. Programming contracts are useful for establishing *blame* when things go wrong.
   B. If a precondition is not satisfied, then the library writer is at fault for any errors.
   C. Contracts may only be specified on modules.
   D. Specifying a contract for a function is more efficient than specifying one for a module, since it avoids repeated checks on the contract for a recursive call.
   E. In Racket, the `->i` operator is used to specify contracts on functions where the validity of the arguments or result depend on each other.

5. (5 points) Select **all** of the following true statements about scoping and operational semantics.
   A. Side effects cannot be expressed in operational semantics.
   B. With lexical scoping, free variables are determined by the path of execution.
   C. Operational semantics are intended to represent a language's behavior concisely and unambiguously, but they are not meant to be read by humans.
   D. Big-step operational semantics evaluate an expression to a value in one step, possibly by recursively evaluating sub-expressions.
   E. To represent state in operational semantics, you can use a "store" or "environment", which is a mapping of variables to values.

6. (10 points) Consider the following Racket code:

```
(struct account (balance))

(define new-account (account 0))

(define (balance acc)
  (account-balance acc))

(define (deposit acc amt)
  (account (+ (account-balance acc) amt)))
```

Write a `provide` statement that makes the following guarantees:

- The `balance` function takes in an `account` and returns a number.
- The `deposit` function takes in an `account` and a positive number, and returns an `account`.

7. (15 points) Write a `largest-elem` function in Racket that takes a list of numbers and returns the largest element in the list. (You may **not** use the `max` function). If the list is empty, raise an error.

```
;; Sample usage
(largest-elem '(9 0 42 1 6)) ;; evaluates to 42
(largest-elem '(-2 -8 -865)) ;; evaluates to -2
```

8. (10 points) Consider a `with-prob` construct that executes an expression with a given probability. You may find the built-in `random` function handy, which returns a number between 0 and 1. Sample usage:

```
(with-prob 0.99 (displayln "99% chance of printing"))
(with-prob 0.5 (displayln "even money"))
(with-prob 0.0001 (displayln "very rarely prints"))
```

Using macros, create the `with-prob` construct

```
(define-syntax-rule
```

9. (10 points) **In a tail-recursive style**, write a `mult-all` function that takes in a list of numbers and returns the product. (You will need to create a helper function in order to solve this problem).

```
;; Sample usage
(mult-all '(2 3 5)) ;; evaluates to 30
```

10. (10 points) Write an `odd-elements` function in Racket that takes a list of numbers and returns a list containing only the odd numbers.

```
;; Sample usage
(odd-elements '(9 0 42 1 6)) ;; evaluates to '(9 1)
(odd-elements '(-3 8 865)) ;; evaluates to '(-3 865)
```

11. (10 points) Consider the following code from a new programming language (specifically Lua, if you are curious).

```
x=42

function foo()
  print(x)
end

function bar()
  local x=666
  foo()
end

bar()
```

What would you expect this code to print if this language uses lexical scoping? What if it uses dynamic scoping?

12. (10 points) Consider the following language and big-step operational semantics:

$e ::=$		*Expressions*
	v	value
	`ton` e	ton expression
	`ro` e e	ro expression

$v ::=$		*Values*
	`yes`	yes
	`no`	no

$$[\text{VAL}]\ \frac{}{v \Downarrow v} \qquad [\text{RO-NO}]\ \frac{e_1 \Downarrow \texttt{no} \quad e_2 \Downarrow \texttt{no}}{\texttt{ro}\ e_1\ e_2 \Downarrow \texttt{no}}$$

$$[\text{TON-YES}]\ \frac{e \Downarrow \texttt{yes}}{\texttt{ton}\ e \Downarrow \texttt{no}} \qquad [\text{RO-YES1}]\ \frac{e_1 \Downarrow \texttt{yes}}{\texttt{ro}\ e_1\ e_2 \Downarrow \texttt{yes}}$$

$$[\text{TON-NO}]\ \frac{e \Downarrow \texttt{no}}{\texttt{ton}\ e \Downarrow \texttt{yes}} \qquad [\text{RO-YES2}]\ \frac{e_1 \Downarrow \texttt{no} \quad e_2 \Downarrow \texttt{yes}}{\texttt{ro}\ e_1\ e_2 \Downarrow \texttt{yes}}$$

Write out the derivation of

`ton (ro (ton no) (ro no yes))`

You might find it useful to specify the name of the rule you are using at each step.