Name:

Student Number:

Please read the following guidelines carefully!

- This examination has 8 questions. There are a total of 12 pages, DOUBLE-SIDED.
- You may use helper functions unless explicitly told not to.
- Any question you leave blank or clearly cross out your work and write “I don’t know” is worth 10% of the marks.
- You must earn a grade of at least 40% to pass this course.

<table>
<thead>
<tr>
<th></th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
<th>Q7</th>
<th>Q8</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grade</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Out Of</td>
<td>8</td>
<td>5</td>
<td>13</td>
<td>7</td>
<td>8</td>
<td>10</td>
<td>6</td>
<td>9</td>
<td>66</td>
</tr>
</tbody>
</table>

Take a deep breath.

This is your chance to show us

How much you’ve learned.

We WANT to give you the credit

That you’ve earned.

A number does not define you.

It’s been a real pleasure teaching you this term.

Good luck!
1. (a) [2] Consider the following nested function call expression in Racket. Write down the order in which the function call expressions will be evaluated, using what you know about eager evaluation in Racket. Write full function call expressions, e.g. “(c a)”, instead of just naming the function, e.g. “c”.

(f (g a b (c a) (h)) (a (b c) c))

(b) [2] Consider the following function definition:

(define (f a b)
    (let ([a 10] [c 20])
        (+ a b c d)))

State which identifiers are **bound** in this function, and which identifiers are **free**. No explanation necessary.

(c) [2] Define a **closure**. No code examples necessary for this question.

(d) [2] The following Racket code snippet produces two values when run. State the output; no explanation necessary. (Note: Racket uses lexical scope, not dynamic scope!)

(define x 10)
(define (f x) (* x x))
(define (g f) (f x))

(g f)
(let ([x 1]
       [f (lambda (x) (* x 3))])
  (g f))

Output:
2. [5] Below, we show the first few rows of Pascal’s triangle:

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
...```

The numbers in Pascal’s triangle are generated using two rules:

- The \( n \)th row has \( n + 1 \) entries, and its first and last entries are 1.
- All internal entries are computed by taking the sum of the two entries directly above it.

For example, the 5th row would be 1 5 10 10 5 1, where the 5’s are coming from the sum of 1 and 4 in the fifth row, and the 10’s are the sum of 4 and 6.

More suggestively:

\[
[1, 5, 10, 10, 5, 1] = [1, 1 + 4, 4 + 6, 6 + 4, 4 + 1, 1]
\]

Write a Haskell definition for the infinite list of lists \( \text{pascalTriangle ::} \ [\ [\text{Integer}] \] \), where the \( n \)th list is the \( n \)th row of Pascal’s triangle. Example usage:

```
> pascalTriangle !! 3
[1,3,3,1]
```

You may \textbf{not} use any formulas you might know for the entries in Pascal’s triangle. You may not use any built-in functions other than the ones found on the aid sheet; you may not use infinite list range expressions like \([0..]\), nor \texttt{nats} from the aid sheet. Instead, you must use the fact that you can generate the \( n \)th row of Pascal’s triangle from the \((n - 1)\)-th row.

\textbf{Hint}: define helper functions!
3. (a) [2] Explain the difference between dynamic typing (used in Racket) and static typing (used in Haskell).

(b) [6] For each of the following Haskell definitions, give the type that Haskell would infer for it in the space provided.

```haskell
f ::

f x y z = if x == (head y) then head z else "Hi"
```

```haskell
g ::

g m n = \p -> not (m n p)
```

(c) [2] Even though Haskell’s static type system offers many safety benefits, it also restricts the valid programs you can express. Describe one thing you can do in Racket that you cannot do in Haskell because of Haskell’s stronger restrictions on types. (So, do not mention things like macros, continuations, or mutation).

You should give a brief English description as well as a code example.

(d) [3] Something you may not have realized in your study of Haskell is just how powerful type signatures can be. If we are given a function type signature `f :: a -> a`, the only function `f` could be is the identity! In a similar vein, implement the following function `g`, based on its type signature. (Hint: a function does not necessarily use all of its arguments.)

```haskell
\[ g :: (a -> b) -> (b -> d) -> [c] -> a -> d \]```
4. Recall the definition of the `Maybe` data type in Haskell – used to safely represent the result of a failed computation – as well as one function we developed in lecture:

```haskell
data Maybe a = Nothing | Just a

lift :: (a -> b) -> Maybe a -> Maybe b
lift _ Nothing = Nothing
lift f (Just x) = Just (f x)
```

(a) [1] What is the type of `lift head (Just [True,True,False])`?

(b) [2] What is the type of `lift (++) (Just [True,True,False])`? Recall that `++` is list concatenation:

```haskell
> (++) [True,False] [True]
[True,False,True]
```

(c) [4] Consider the function `applyMaybe`, which takes two Maybe values, the first of which stores a function, and the second an argument to that function, and applies that function to the argument, “failing” if either argument is a failure. Here is an example of how `applyMaybe` could be used:

```haskell
> let f x = x + 10
> applyMaybe (Just f) (Just 2)
Just 12
> applyMaybe (Just f) Nothing
Nothing
> applyMaybe Nothing (Just 10)
Nothing
> applyMaybe Nothing Nothing
Nothing
```

Give both the (most general possible) **type signature** and **implementation** of `applyMaybe`.
5. Here are the English definitions of some predicates about movies. You may assume for this question that movie titles and person names are unique.

% movie(T, Y): 'There is a movie with title T made in year Y.'
% actor(X, T, C): 'Actor X played a character named C in the movie with title T.'
% rating(X, T, R): 'Person X rated the movie with title T a score R out of 5.'
% happy(X): 'Person X is happy.'

Using these predicates, write some facts and/or rules to represent the following English statements. You may write more than one line of code per question.

Use underscores appropriately.

(a) [2] The movie _cats_ was made in 1971. The actor _brad_ played a character named _rengar_ in this movie.

(b) [2] The actor _julia_ acted in every movie made in 1995. (Hint: rewrite this in the form “if ... then ...”)

(c) [4] A person is happy if they have acted in at least one movie with the actor _meryl_, or if they have rated at least one movie 5 (out of 5).
6. Consider the following Prolog code:

child(edgar, terra).
child(locke, ceres).
child(terra, ceres).
descendant(X, Y) :- child(X, Z), descendant(Z, Y).
descendant(X, Y) :- child(X, Y).

(a) [3] State the output of the following query, and carefully explain (in English) why this occurs.

?- descendant(X, ceres).

(b) [3] Suppose we replaced the current two lines defining descendant with the following:

descendant(X, Y) :- child(X, Y), !.
descendant(X, Y) :- child(X, Z), descendant(Z, Y).

State the output of the following query, and carefully explain (in English) why this occurs. Be sure to explain how the cut influences the output.

?- descendant(X, Y).
We have repeated the facts from the previous page, with a third definition of *descendant*.

- \text{child}(edgar, terra).
- \text{child}(locke, ceres).
- \text{child}(terra, ceres).
- \text{descendant}(X, Y) :- \text{descendant}(Z, Y), \text{child}(X, Z).
- \text{descendant}(X, Y) :- \text{child}(X, Y).

(c) [4] Explain what happens when we run the following query:

?- \text{descendant}(X, ceres).
7. [6] Define the predicate `subsetSum(L)`, which succeeds if and only if L is a list of integers, and a non-empty subset of the numbers in L have a sum of 0. Note that the elements used do not have to appear consecutively in L, and that L is allowed to have duplicates. For example, `subsetSum([1,2,-4,3,10])` succeeds because of the subset \{1,-4,3\}. You may assume that L is instantiated.

**Hint:** this question is quite hard to do without making a helper predicate. Try generalizing the problem to take in an arbitrary target sum instead of just 0.
8. (a) [4] Write a Racket function `concatMap`, which takes a function \( f \) which always returns a list, and a list \( \text{lst} \), and does the following:

- Apply \( f \) to each item in \( \text{lst} \)
- Append all of the resulting lists together into a single list and return the result

Note that this is a description of what this function does, but not its implementation, which is up to you. The only built-in functions you may use are the ones found on the cheat sheet.

```racket
> (define (f x) (list (+ x 1) (+ x 10)))
> (concatMap f '(1 2 3))
'(2 11 3 12 4 13)
```
(b) [5] On the aid sheet, we have defined a Racket list comprehension macro. Your job is to write a new macro called **list-comp-2** which represents a nested list comprehension:

```
(list-comp-2 <expr> for <var1> in <lst1> for <var2> in <lst2>)
```

This expression has the following semantics (imperative-style pseudocode):

```
for <var1> in <lst1>:
    for <var2> in <lst2>:
        evaluate <expr>
return all evaluations of <expr> in a list
```

For example:

```
> (list-comp-2 (+ x y) for x in '(1 2 3) for y in (list 10 x))
'(11 2 12 4 13 6)
```

Note that the second list can contain references to the first identifier, and the expression can contain references to zero, one, or both identifiers. In the space below, implement this macro. You do not need to handle the single-variable case, nor do you need to worry about more than two variables.

**Hint:** use `concatMap` from the previous part.
Use this page for rough work.