WrittenAssignment3 Solution Set

Don't forget to tell me with whom you collaborated (or that you didn't).

Grammars and Interpreters

1. In class, we discussed a lisp syntax for a subset of scheme. Write a grammar that corresponds to the following fragments of scheme:

- non-negative integers
- alphabetic names
- operations + - * / and predicates > < = <= >=
- if expressions (the most common form only), e.g. (if a b c))
- lambda expressions, e.g. (lambda (x) x)
- define expressions (unsugared form only), e.g., (define x 3)
- applications (i.e., applying an appropriate thing to an appropriate set of arguments)

There was enough ambiguity in this question that there are several correct answers. Some specific incorrect items are indicated in this solution.

We will use S as the start symbol of our grammar. Note that this grammar is for a single S-expression. A scheme program might better be characterized as S^+, i.e., one or more S-expressions in sequence.

Boldface indicates terminals. [..], |, and * are used for ranges, alternatives, and Kleene star, respectively. ;; is used to mark comments.

```
S -> Number
S -> Name
S -> Built-in                      ;; Technically, these are just names in real Scheme.
S -> ( if S S S )                  ;; Note NOT (if S*); only 3 Ss (or 2 for real Scheme)
S -> ( lambda ( Name* ) S )        ;; Note NOT (lambda (S*) S)
                                   ;;   Also, real Scheme lambdas are even more general.
S -> ( define Name S )             ;; Note NOT (define S S) nor (define Name S*)
S -> ( S S* )                      ;; This allows (Number S*), which is syntactically legal
                                   ;;   but doesn't run well.[1]

Number -> 0
Number  -> [1..9][0..9]*

Name -> [a..zA..Z][a..zA..Z]*      ;; Full scheme also includes symbols and numerals

Built-in -> + | - | * | / | > | < | = | <= | >= | <>    ;; oops, forgot the last in the pset
```

[1] It also allows a define as its first S-expression, which is technically not syntactically legal, but I didn't expect you to know that. Some lisps allow that, syntactically.

There are oodles of other correct grammars. Important points:

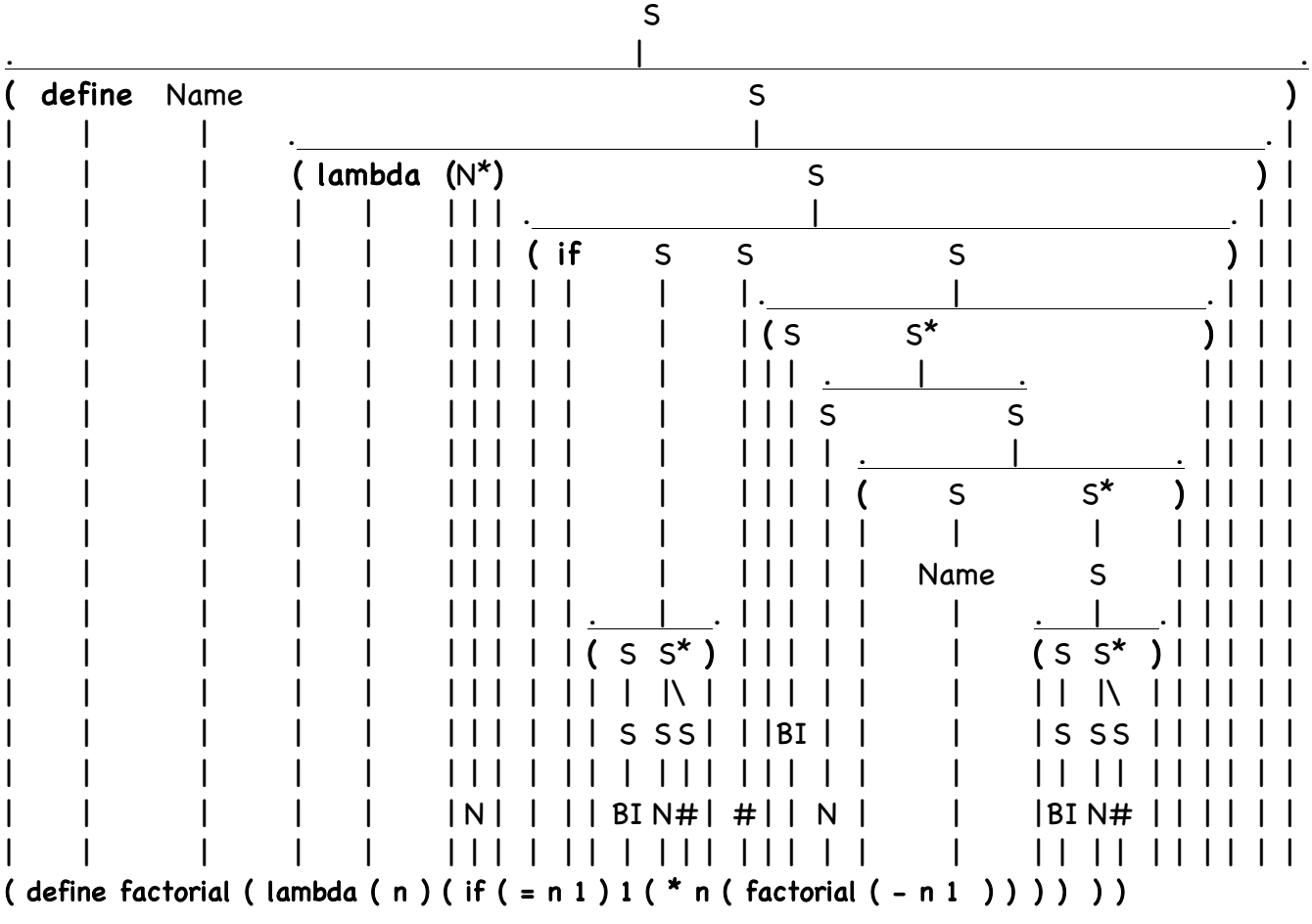
1. **if** should allow only 2 or 3 expressions, not 0 or 1 or 4 or more.
2. **if** should allow *any* S-expression as its test, consequent, and alternative.
3. **lambda** should allow only names as parameters, not S-expressions
4. **define** requires exactly a Name and an S-expression, *not* two S-expressions and not more than one S-expression after the Name.
5. applications must not be restricted to Built-ins; in particular, lambda-expressions, if-expressions, and names must be legal in some application form.

2. Write a factorial program in your subset of lisp.

Here's a basic one. There are lots of variants:

```
(define factorial
  (lambda (n)
    (if (= n 1)
        1
        (* n (factorial (- n 1))))))
```

3. Draw a parse tree for your lisp factorial program according to your grammar. Use a representation in which all terminal symbols appear in the leaves of the parse tree; each interior node should be labeled with the head of the production that it (and its children) represent.



Note that in this parse tree BI, N, and # are abbreviations for Built-in, Name, and Number respectively. (I couldn't make it pretty and verbose at the same time.)

There are many correct ways to draw a parse tree, and of course each parse tree presumes a particular grammar, so yours will probably look different from mine. A different parse tree representation (using the same factorial program and the same grammar) is included with the solution to question 7. The keys to a correct parse tree are:

- Every terminal must be represented explicitly.
- It must be possible to determine what derivation (what grammar rules) produced each terminal.
- The parse tree must have the start symbol at its root and every derivation should correspond to a production of the grammar.

4. Consider a simplified block-structured programming language in which the following statement types exist:

- conditionals of the form if *expr* then *stmt* else *stmt*
- loops of the form while *expr* do *stmt*
- blocks of the form begin *stmt* *end
- assignments of the form *var* := *expr*

Expressions should include the same operations and predicates as in scheme; names should be alphabetic only; numbers should be non-negative integers. Write a grammar for this language.

This language distinguishes expressions from statements, so our grammar has two sections for these. (Stmt is our start symbol)

```
Stmt -> if Expr then Stmt else Stmt      ;;
Stmt -> while Expr do Stmt               ;;
Stmt -> begin Stmt* end                  ;;
Stmt -> Name := Expr                     ;;

Expr -> Expr Op Expr
Expr -> Number
Expr -> Name
```

We can reuse several of the productions from our lisp grammar to complete this one:

```
Number -> 0
Number  -> [1..9][0..9]*

Name -> [a..zA..Z][a..zA..Z]*

Op -> + | - | * | / | > | < | = | <= | >=     ;; changed nonterminal name here to reflect role
```

Important considerations:

- Expressions and statements are different.
- A while loop only has a single statement in its body.

5. Write a factorial program in this language. Because this language does not include the definition of procedures, functions, or subroutines, you may assume that there is a variable n that contains the number to be factorial'd and that your result should wind up in a variable named ans .

There are of course lots of ways to do this. Here's one:

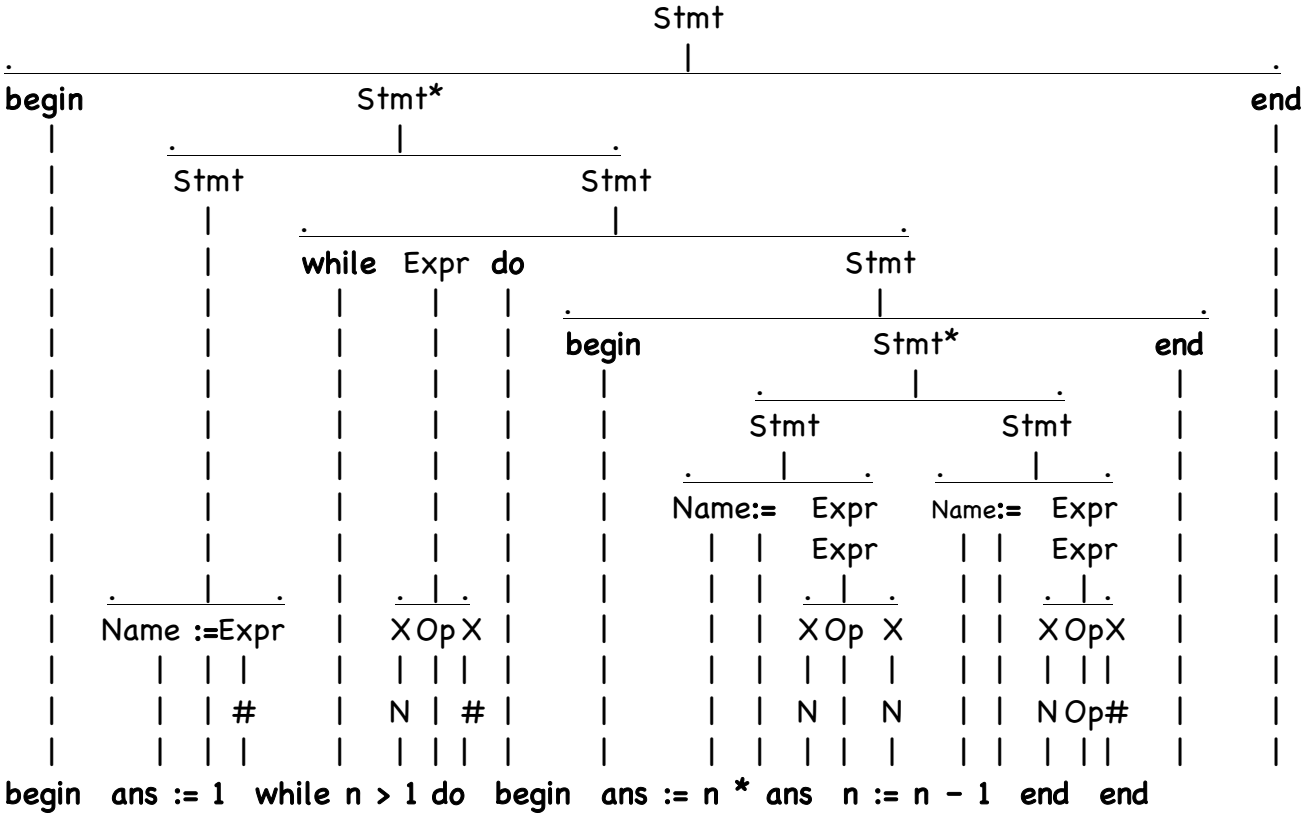
```
begin
    ans = 1
    while n > 1 do
        begin
            ans = n * ans
            n = n - 1
        end
end
```

I've used white space to make this easier to read, but note that the program would be legal if formatted differently. For example,

```
begin ans = 1 while n > 1 do begin ans = n * ans n = n - 1 end end
```

should have the same parse tree, etc. (and it does!)

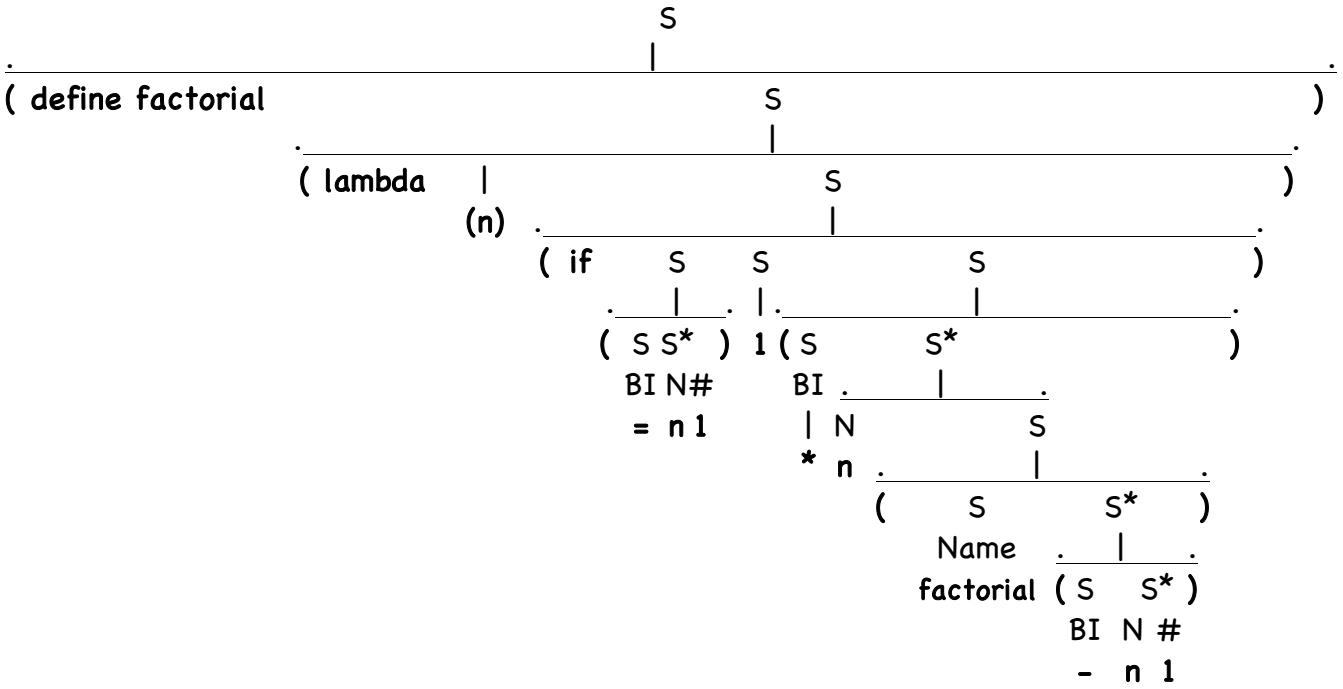
6. Draw a parse tree for your factorial program according to your grammar. Use a representation in which all terminal symbols appear in the leaves of the parse tree; each interior node should be labeled with the head of the production that it (and its children) represent.



7. Observing the parse trees for your two programs, try to explain why lisp is generally viewed as an easier language for which to write an interpreter.

Interestingly, many students said that they found the parse tree for the block structured language easier. This wasn't actually what I was getting at; the grammar for scheme may be either easier or harder, depending on how you think about it. (It has really only one major nonterminal – S-expressions – and so its productions all look sort-of the same. The added variety of the block structured language creates more complexity but also more structure.)

The reason that lisp is easier to interpret is that the lisp program structure – the list structured representation of the lisp program – *is* the parse tree. I've redrawn the parse tree in an attempt to show this:



The list structure for the lisp program is just a lisp representation of this tree (without the labels on the internal nodes…but note that these labels are almost exclusively S in any case!)

```
(define factorial
                (lambda
                        (n)    (if
                                    (=          1      (*
                                       n 1)                (factorial
                                                                  (-
                                                                     n 1) ) ) ) ) )
```

Automata

1. *[warmup]* Consider two regular languages, L 1and L 2, each with a corresponding finite state automaton (F 1and F 2, respectively). Explain how to construct a finite state state automaton that recognizes L 1union L 2, i.e., the language that includes all strings that are *either* in L 1or in L 2. Demonstrate this for the languages 0*1* and (01)*.

The intuition is that we can run either automata on the string and if it accepts, we're done. We just need to guess correctly which one to run, which we can do by constructing a nondeterministic FSA that makes the guess for us. (Remember that an NFA accepts a

string whenever there's *some* path through it that accepts that string.)

Create a new FSM that contains the union of the states of F1 and F2 (renamed, if necessary, so there's no confusion). Add a new start state, S, with epsilon-transitions to the start states of F1 and F2. The rest of the transition table is the transition tables of F1 and F2 (and, since their states are separate, there are no transitions between states of F1 and states of F2 or vice versa). The set of accepting states of this new automata is the union of the accepting states of the F1 and F2.

Even more formally:

If F1 = < Q, Sigma, q0, Delta, F > and F2 = < P, Sigma', p0, Delta', G > (we assume Q and P are disjoint and can rename states to make this so) then L1 U L2 is recognized by the automaton

< Q U P,
Sigma U Sigma',
S,
Delta U Delta' U {(S, epsilon, q0), (S, epsilon, p0)},
F U G >

2. *[warmup]* For regular languages L 1and L 2, explain how to construct a finite state automaton that recognizes L 1intersection L 2, i.e., the language that includes all strings that are in *both* L1and in L 2. Demonstrate this for the languages 0*1* and ((0U1)(0U1))*.

This time, the intuition is that we want to run *both* automata simultaneously. If they both accept the string, then we should accept it. We can't use the same trick, though, because an NFA accepts whenever *some* computation accepts and we need to be sure that *both* automata accept. So we'll make a new FSA that keeps track of both automata simultaneously, sort-of like running them both and keeping a finger on which state each one's in. (There's a superficial resemblance to the NFA->DFA transformation, though the construction is slightly different.)

As before, F1 = < Q, Sigma, q0, Delta, F > and F2 = < P, Sigma', p0, Delta', G >

Create a new FSM whose states will be named (qi, pj), where qi is a state of F1 and pj is a state of F2 (i.e., qi in Q, pj in P). Start in the state (q0, p0), where q0 and p0 are the start states of F1 and F2 respectively. For each symbol x in Sigma, Sigma' (actually we only care about their intersection), add all transitions ((qi, pj), x, (qk, pl)) for which (qi, x, qk) is a transition of F1 and (pj, x, pl) is a transition of F2; no other transitions. A state (qi, pj) is an accepting state of this new automaton exactly when qi is an accepting state of F1 AND pj is an accepting state of F2.

In other words, L1 intersect L2 is recognized by the automaton

< Q X P,
Sigma intersect Sigma',
(q0, p0),

{ ((qi, pj), x, (qk, pl)) | qi, qk in Q; pj, pl in P;
x in ((Sigma intersect Sigma') U epsilon);
(qi, x, qk) in Delta, (pj, x, pl) in Delta'}
{(qi, pj) | qi in F and pj in G} >[2]

3. Let C be a context free language and R be a regular language. Prove that the language C intersect R is context free (*Sipser*). Hint: Show how to construct a PDA that recognizes this language.

This is really the same problem as the previous one except that now one of our FSMs is a PDA. PDAs can behave like FSMs (by ignoring the stack), but an FSM can't behave like a PDA. This means that we'll need a PDA to simulate running both machines at the same time.

The only hard part of the construction (beyond #2, which is really the key to this construction, too) concerns how to handle the stack transitions. In the previous problem, we simulated each FSM separately, with the first part of the state keeping track of the first FSM and the second part of the state keeping track of the second FSM. This is exactly what we do here. The first part of the state tracks the FSM, which doesn't care about the stack. The second part of the state tracks the PDA, which does, So a transition can happen exactly when the FSM-part of the state can take its transition and the PDA-part can take *its* transition. That is, we only care about the stack-state for the PDA-part of the (new, combined) transition.

Formally, let the FSM for R be

F = < $States_F$,
$Alphabet_F$,
$startState_F$,
$Transitions_F$,
$AcceptingStates_F$ >

and let the PDA for C be

P = < $States_P$,
$Alphabet_P$,
$StackSymbols_P$,
$Transitions_P$,
$startState_P$,

[2] Subtle nit: This construction has a minor bug with respect to epsilon transitions. Right now, the new FSM can take an epsilon transition only if *both* original FSMs explicitly include the relevant epsilon transition. We can fix this by adding to each original FSM (before combining them) all transitions (q, epsilon, q) for every state q of that FSM. (These transitions say it's OK to stay in the same state if you don't consume any input.)

StackStartSymbol$_P$,

AcceptingStates$_P$ >

Then we can construct a new PDA that accepts exactly if *both* of the other machines would accept:

solution = < { (q, r) | q in States$_F$ and r in States$_P$ },

Alphabet$_F$ intersect Alphabet$_P$,

StackSymbols$_P$,

{ ((fState, pState), input, stackTop) -> ((newF, newP), newStack)

| fState, newF in States$_F$;

pState, newP in States$_P$;

input in ((Alphabet$_F$ intersect Alphabet$_P$) U epsilon);

stackTop, newStack in (StackSymbols$_P$ U epsilon);

(fState, input) -> newF in Transitions$_F$,
(pState, input, stackTop) -> (newP, newStack) in

Transitions$_P$ }

(startState$_F$, startState$_P$),

StackStartSymbol$_P$,

{ (q, r) | q in AcceptingStates$_F$ and r in AcceptingStates$_P$ } >

4. Using the previous proof, show that the language A = { w | w in {a, b, c}* and contains an equal number of a's, b's, and c's} is not context free. (*Sipser*) Note that this language is *not* $a^nb^nc^n$.

First, note that the reason why A is not $a^nb^nc^n$ is that the letters in a string in A may appear in any order. For example, cabbac and abbacc are both in A, but not in $a^nb^nc^n$.

Now assume by way of contradition that the specified language – A = { w | w in {a, b, c}* and contains an equal number of a's, b's, and c's} – *is* context free. (If we do get a contradiction, we will have proved that A is not context free, which is what we want.)

If A is context free, then there is a PDA that recognizes it. Call this machine P.

We also know that there's an FSM that recognizes the regular language a*b*c*, i.e, the language consisting of some number of a's followed by some (possibly different) number of b's followed by some (possibly different) number of c's. Call this FSM F.

Then F intersect P is the intersection of a regular language and a context free language, so by #3 the resulting language is context free.

But this leads to a problem. Specifically,

a*b*c* intersect A =

a*b*c* intersect { w | w in {a, b, c}* and contains an equal number of a's, b's, and c's}

is the language containing equal numbers of a, b, and c in that order. This is just

$a^n b^n c^n$

which is *not* context free. So assuming that A is context free leads to a contradiction, meaning that A must not be context free.

Karnaugh Maps

1. Draw the Karnaugh maps for the following functions:

- Parity, i.e., the function (on four inputs) that is true if the number of true inputs is even

Parity		q0, q1			
		00	01	11	10
q2, q3	00	0	1	0	1
	01	1	0	1	0
	11	0	1	0	1
	10	1	0	1	0

- (q0 v q1) ^ (q2 v q3)

(q0 v q1) ^ (q2 v q3)		q0, q1			
		00	01	11	10
q2, q3	00	0	0	0	0
	01	0	1	1	1
	11	0	1	1	1
	10	0	1	1	1

- q0 -> ((q1 ^ q2) v q3)

We begin by observing that this is not(q0) or ((q1 and q2) or q3. (Why?)

q0 -> ((q1 ^ q2) v q3)		q0, q1			
		00	01	11	10
q2, q3	00	1	1	0	0
	01	1	1	1	1
	11	1	1	1	1
	10	1	1	1	0

2. Populate a four-variable Karnaugh map with the formula made true by the truth assignment corresponding to that square. For example, if q0 is false, q1 is true, q2 is true, and q3 is false, the truth assignment might be written as as not(q0), q1, q2, not(q3) (but that renders extremely poorly on the wiki; you can use the overbar notation that makes it prettier!).

Truth values		q0, q1			
		00	01	11	10
q2, q3	00	$\overline{q_0}\overline{q_1}\overline{q_2}\overline{q_3}$	$\overline{q_0}q_1\overline{q_2}\overline{q_3}$	$q_0q_1\overline{q_2}\overline{q_3}$	$q_0\overline{q_1}\overline{q_2}\overline{q_3}$
	01	$\overline{q_0}\overline{q_1}\overline{q_2}q_3$	$\overline{q_0}q_1\overline{q_2}q_3$	$q_0q_1\overline{q_2}q_3$	$q_0\overline{q_1}\overline{q_2}q_3$
	11	$\overline{q_0}\overline{q_1}q_2q_3$	$\overline{q_0}q_1q_2q_3$	$q_0q_1q_2q_3$	$q_0\overline{q_1}q_2q_3$
	10	$\overline{q_0}\overline{q_1}q_2\overline{q_3}$	$\overline{q_0}q_1q_2\overline{q_3}$	$q_0q_1q_2\overline{q_3}$	$q_0\overline{q_1}q_2\overline{q_3}$

3. There is an important observation to be made about the formulae in adjacent squares. What simple property can be stated about every pair of adjacent squares (two squares sharing a top, bottom, or side) in a Karnaugh map?

The truth assignments in any two adjacent squares differ in the truth value assigned to exactly one variable.

4. (Hopefully) observe that this property holds for some non-adjacent squares as well. However, these squares can be made adjacent by wrapping the right edge of the Karnaugh map around to meet its left edge and, simultaneously, wrapping the top to meet the bottom. Explain.

The property in #3 holds because moving horizontally or vertically changes the bit pattern by one bit. This is true at the edges as well; 10 wraps around to differ from 00 by only one bit.

5. One can use a Karnaugh map to read off formulae for boolean operations in disjunctive normal form. For example, OR can be expressed as *not(q0)q1* v*q0 not(q1)* v*q0 q1* simply by reading the formulae corresponding to the squares with 1s. Using this simple-minded method, how many booleans would appear in the formula corresponding to a *k*variable Karnaugh map with *n*1s? (Hint: for the 2-variable function OR, with 3 1s, the answer is 6.)

For a k-variable Karnaugh map with n 1s, the naïve DNF formulation of the function is k variables per clause and n clauses, i.e., nk.

6. Fortunately, the formula for OR can be simplified by combining terms. (Shockingly enough, it reduces to *q0* v*q1*) In a traditional Karnaugh map -- such as AND, OR, or XOR, above, but including the larger variants -- suppose that an *m*x*n*rectangular region contains exclusively 1s. (For example, in the Karnaugh map for OR, the second column is a 2x1 region of 1s and the second row is a 1x2 region of 1s.) What does this tell you about the formula corresponding to this Karnaugh map? In particular, if you have a k-variable Karnaugh map with an *m*x*n*rectangular region of 1s, how many variables would that region require to represent in disjunctive normal form under the naive method of the previous question? How many does it in fact require?

For a k-variable Karnaugh map, an mxn rectangular region can be represented using a single conjunction of $k - \log_2 mn$ variables. For example, each of the two rectangles in the Karnaugh map for OR requires $2 - (\log_2 2) = 2 - 1 = 1$ variable to express. Thus, the simplest representation of OR in DNF is 2 variables long, a considerable simplification over the naïve representation (from problem 5), which is 6 variables long.