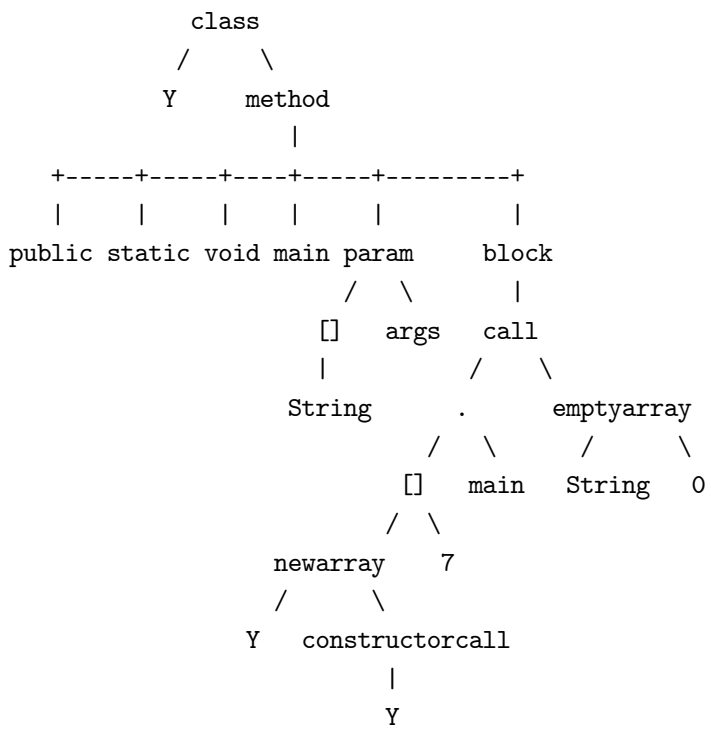
C M S I 4 8 8

Final Exam Answers

1.



2. a. `0[0-7]*`
   b. `0{1-4} | [01]*0000`
   c. `[A-Fa-f0-9]{3}[08]`
   d. `\d+\.\d+([Ee][+-]?\d{1,3})?`
   e. `([^f]|f(?!(ile|inal)))*` (*with a case-insensitive flag*)

3. Microsyntax:

```
NUMLIT → \d(\.\d*([Ee][+-]\d+)?)?
BOOLLIT → 'true' | 'false'
STRINGLIT → \"[^\p{Cc}*]\"
ID → [A-Za-z]([A-Za-z0-9_])*
```

   Macrosyntax

```
OBJECT  → '{' (BINDING (',' BINDING)*)? '}'
BINDING → ID ':' VALUE()
VALUE   → NUMLIT | STRINGLIT | BOOLLIT | ID | ARRAY | OBJECT
ARRAY   → '[' (VALUE (',' VALUE)*)? ']'
```

   The syntax-checking part of the parser (not the AST generation):

```
function parseObject() {
  match('{')
```

```
  parseBinding();
  while (at(',')) {
    match()
    parseBinding()
  }
  match('}')
}
function parseBinding() {
  match('ID')
  match(':')
  parseValue()
}
function parseValue() {
  if (at(['NUMLIT','STRINGLIT','BOOLLIT','ID'])) match()
  else if (at('[')) parseArray()
  else if (at('{')) parseObject()
  else error('Illegal value', tokens[0])
}
```

4. **Long-winded answer**: Because Java is statically typed, the compiler has to be able to assign a type to every expression at compile time. The way things are now, the type is specified directly in the source code; *that's* the type you are going to assign to the whole expression! Piece of cake. And all you have to do is go through the expressions in the array expression and just *check* them. If we left the type off, the compiler would have to run through all the expressions and *infer*, via a not-so-cheap and very involved recursive process, the *least general type* that encompasses all expressions. This can be pretty hard if your aggregate contains other aggregates! The designers figured the cost of doing is outweighed the benefits of terser code.

   Short Answer: Including the type means that type checking need only check compatibility with a given type, rather than a horribly expensive computation with type-inference to *compute* the type of the array.

5. a. OK, it's just indexing an array
   b. OK, Java string literals are pretty tolerant
   c. Static semantic, booleans connot be compared. By the way, Java, unlike Iki, permits the chaining of relational operators in its syntax.
   d. OK as long as you have two dots, or a space between the literal and the dot
   e. Lexical
   f. Static semantic
   g. OK, it would be sad if this were not allowed
   h. OK, because Python is cool
   i. OK, it actually produces the value undefined when reading, and doesn't complain at all when writing

6. Only the first two lines need to be touched.

```
SCRIPT → (FORM {FORM.level := 0})+
FORM   → '(' 'define' ID PARAMS (FORM' {FORM'.level := FORM.level+1})+ ')'
       | '(' OP OPERAND+ ')'
```

7. I hope you remembered our little discussion here....
   a. Terminating end (a la Ruby), Indentation (a la Python), Nested parens (a la Lisp).

   b. He probably thought that without the `elsif`, the requirement for bracing all blocks wout require braces to stack up, like

```
if (e1) {
  s1
```

```
  } else {if (e2) {
    s2
  }}
```

c. Yep, my way works. No stacked up braces. Note that it's LL(2), not LL(1). This is actually no big deal here, because we can still do recursive descent parsing, in case that's your concern.

d. Well, yeah, so here's the issue with the terminating-end languages. On the surface it looks okay and it's not left-recursive, but the grammar

```
IFSTMT -> 'if' EXP 'then' STMT+
          ('else' 'if'  EXP 'then' STMT+)*
          ('else' STMT+)?
          'end'
```

has a choice point at the beginning of line 2 — should we dive in to the (...)* or not? Both of these choices begin with an 'else' so it's not LL(1). How about LL(2)? Well, one starts with 'if' and the other with STMT. No help there, as STMT can begin with 'if'. So how far should we lookahead to choose between the choices? There's no upper bound! Each can expand to

```
if EXP then STMT else if EXP then STMT else if EXP then STMT else if EXP then ...
```

It is not LL(k) for any k.

8. One possible answer:

```
PROGRAM → INST+
INST    → deg | rad
        | down | up | left NUM | right NUM
        | forward NUM | backward NUM
        | color NUM NUM NUM | "[" INST+ "]"
```

Note that NUM is a primitive token, and note that the value constraint on color arguments is not specified in the syntax; we're leaving that to the static semantic description. The grammar is LL(1) and non-ambiguous.

9. Ah, now Iki is cool....
   a. In the macrosyntax, just change the `Stmt` clause `Id '=' Exp` to `Id (',' ID)* '=' Exp (',' Exp)*`

   b. No, we just need to enhance `AssignmentStatement`

   c.
```
function parseAssignmentStatement() {
  var targets = [match('ID')]
  while (at(',')) {
    match()
    targets.push(match('ID'))
  }
  match('=')
  var sources = [parseExpression()]
  while (at(',')) {
    match()
    sources.push(parseExpression())
  }
  return new AssignmentStatement(targets, sources)
}
```

d. We have to check that the number of clauses on both sides are the same, and check type compatibility for each pair

e. `var _old_a = a; a = b; b = _old_a + b;`

10. Here's my attempt:

```
// Fast string repeat from http://stackoverflow.com/a/5450113/831878
function _repeat(s, count) {
  if (count < 1) return '';
  var result = '', pattern = s;
  while (count > 0) {
    if (count & 1) result += pattern;
    count >>= 1, pattern += pattern;
  };
  return result;
}
var _x_1 = _readint('Enter a number');
var _temp_2 = ((_x_1 * 3) - 7);        // evaluate the common subexpression only once
var y_3 = _temp_2 + _temp_2;
var _b_4 = ['c'];
var _temp_5 = _repeat("rats", y + y);
for (var _x_6 of _temp_5.split("")) {  // using the new 'of' of ES6.
  _b_4.push(_x_6, _x_6);
}
console.log(_b_4);
```