

# CSE 413 Winter 2001 Final Exam

Name \_\_\_\_\_ ID # \_\_\_\_\_ Score \_\_\_\_\_

\_\_\_\_\_

1	2	3	4	5	6	7
---	---	---	---	---	---	---

There are **7** questions worth a total of **90** points. Please budget your time so you get to all of the questions. ***Keep your answers brief and to the point.***

You may refer to the textbook (Budd's *Understanding Object-Oriented Programming using Java*) and clean copies of the course handouts *only*. No other books or notes are allowed.

Please wait to turn the page until you are told to begin.

## CSE 413 Winter 2001 Final Exam

**Question 1.** (12 points, 4 each) Regular expressions.

(a) Describe the set of strings generated by the regular expression

$$((xy^*x) \mid (yx^*y))^*$$

(b) Write a regular expression that generates all non-empty strings of a's, b's, and c's where the first c (if any) appears after the first a (if there are any a's in the string).

(c) Write a regular expression that generates all non-empty strings of a's and b's that *don't* contain the contiguous substring baa.

## CSE 413 Winter 2001 Final Exam

**Question 2.** (12 points) Context-free grammars

Consider the following grammar.

$$\begin{aligned} S &::= SA \mid Ba \\ A &::= Ab \mid B \\ B &::= aA \mid c \end{aligned}$$

(a) (3 points) List the terminal, nonterminal, and start symbols for this grammar.

Terminals:

Non-terminals:

Start symbol:

(b) (6 points) Draw the parse tree for the sentence  $acac$ .

(c) (3 points) Give a leftmost derivation of the sentence  $acac$ .

## CSE 413 Winter 2001 Final Exam

### Question 3. (8 points) Stack machine hacking.

The Java virtual machine (JVM) defines an instruction set for a stack machine, and it is instructions for this machine that are contained in Java .class files. This question is about generic stack machines, as discussed in class.

Recall that most of the operations in a stack machine take their operands off the top of the expression stack and push the result onto the stack in their place. Typical operations are:

pushval <i>offset</i>	push contents of variable at given offset in the stack frame
pushaddr <i>offset</i>	push memory address of variable with given stack frame offset
pushconst <i>val</i>	push constant integer <i>val</i> onto stack
assign	store top of stack value at location whose address is underneath it on the stack; pop both
pop	pop unneeded value from stack
dup	push duplicate copy of value currently on top of the stack
swap	interchange top two values on the stack
add	addition
mul	multiplication

Suppose we're generating code in a method with the following symbol table.

Variable	Offset
p	12
q	4
z	8

Write the stack machine code corresponding to the following statements. For full credit, the operands of expressions should be pushed from left to right (i.e., push the 12 first when evaluating the last expression).

```
q = 17;
p = 42;
z = 12 * ((p*q) + 4*q+ p);
```

## CSE 413 Winter 2001 Final Exam

**Question 4.** (14 points) Java hacking.

Write a complete Java program that makes a copy of a text file. The file name should be given on the command line (or equivalent, depending on your development environment), and can be accessed as the first element of the `String` array parameter of method `main`. The copy of the file should be named `copy of originalfilename`, where *originalfilename* is the file name taken from the command line.

For full credit, you must copy the file a line at a time (`readLine`), not character by character, and you should use the appropriate reader and writer classes.

## CSE 413 Winter 2001 Final Exam

**Question 5.** (10 points) Language hacking (context-free grammars again).

One of your colleagues proposes to extend D by adding logical and (&&) to the grammar as follows:

$$bool\text{-}exp ::= rel\text{-}exp \mid ! ( rel\text{-}exp ) \mid bool\text{-}exp \ \&\& \ bool\text{-}exp$$

(a) (6 points) Show that this extension makes the grammar ambiguous.

(b) (4 points) Fix the grammar so it contains logical and, but is not ambiguous.

## CSE 413 Winter 2001 Final Exam

**Question 6.** (20 points) x86 hacking.

Here is yet another version of factorial written in D.

```
int fact(int n) { return factaux(1,2,n); }

int factaux(int result, int next, int n) {
    int answer;
    if (next > n)
        answer = result;
    else
        answer = factaux(result*next, next+1, n);
    return answer;
}
```

The goal of this problem is to write an x86 assembly language version of function `factaux` **only**. Use the standard Win32 conventions for function calls, **except** push the function arguments from *left to right*, as you did in your D compiler.

(a) (6 points) Draw a picture showing the local stack frame layout during execution of function `factaux`. This is the layout after the function prologue code has been executed, just before execution of the `if` statement. Be sure to show where registers `ebp` and `esp` point in the stack frame, the location of each parameter and local variable, and their numeric offsets from `ebp`.

## CSE 413 Winter 2001 Final Exam

### Question 6 (cont).

(b) (14 points) Translate `factaux` to x86 assembly language. You don't have to imitate your compiler precisely and generate really bad code; straightforward x86 code is fine, as long as it uses the registers properly and obeys the x86 conventions used by the D compiler for stack layout, function calls, etc. Also, don't omit any statements – be sure to actually store a value in variable `answer`, for instance. It will help us read your answer if you include the source code as comments near the corresponding x86 code. The D code is repeated here for reference.

```
int factaux(int result; int next; int n) {
    int answer;
    if (next > n)
        answer = result;
    else
        answer = factaux(result*next, next+1, n);
    return answer;
}
```



## CSE 413 Winter 2001 Final Exam

**Question 7.** (14 points) Compiler hacking.

C, C++, and Java all contain a *conditional expression* that has the syntax

$$\textit{condition} \ ? \ \textit{expression1} \ : \ \textit{expression2}$$

This is very much like the `if` expression in Scheme. The Boolean expression *condition* is evaluated. If it is true, then *expression1* is evaluated and its value is the value of the entire conditional expression. If *condition* evaluates to false, then *expression2* is evaluated and its value is the value of the conditional expression. **Only one** of *expression1* and *expression2* is evaluated; the other is not evaluated.

Example: This statement compares the values in `x` and `y` and stores the larger value in `max`.

```
max = x > y ? x : y;
```

For this problem, we will add a rule to the D grammar for conditional expressions, and your job is to write a method to compile it. The new grammar rule is

$$\textit{conditional-exp} ::= \textit{bool-exp} \ ? \ \textit{exp} \ : \ \textit{exp}$$

Complete the definition of method `conditionalExp` in the parser class on the next page so it compiles *conditional-exps*. The parser class contains an instance variable that holds the next unprocessed token from the input program, and a function that calls the scanner to read the next token. Use these to access the tokens from the source program - don't call the scanner directly. It also contains a method to write generated code to the output file. Use this to generate code; don't use `writeln` directly. The parameter to function `boolExp` is the label to which control should be transferred in the condition evaluates to false. Finally, function `newLabel` is available to create unique strings that can be used as labels in the generated code as needed.

Hint: You may find it helpful to sketch out the parsing and code generation separately before you write your solution.

Reference information: You should assume that classes representing tokens, `EchoIO`, and the scanner are available for your use, as described in the compiler project handouts.

```
// description of a single lexical token
class Token {
    public int kind;    // kind of token (see constants below)
    public int val;    // if kind=INT, then val=value of the integer

    // lexical classes:
    public static int INT        = 0;    // integer constant
    public static int QUESTION  = 1;    // ? symbol
    public static int COLON     = 2;    // : symbol
    ...                          // ... etc. ...
    public static int EOF       = 99;   // end of input
    // (returned by scanner
    // when no more tokens
    // are available)
}
```

## CSE 413 Winter 2001 Final Exam

```
// parser
class Parser {
    EchoIO eio;          // I/O interface [ initialized elsewhere ]
    Scanner scan;       // Scanner [ initialized elsewhere ]
    ...
    Token tok;          // next unprocessed token from source program

    // update tok by advancing to the next token in the source program
    void nextTok() { tok = scan.nextToken(); }

    // write generated code s to output
    void gen(String s) { eio.println(s); }

    // return unique string that can be used as an assembly lang. label
    String newLabel() { ... }

    // compile bool-exp and generate a conditional jump to falseTarget
    // if the bool-exp evaluates to false
    void boolExp(String falseTarget) { ... }

    // compile bool-exp ? exp : exp
    void conditionalExp( ) {
```

```
}
```