

## **CSE P 501 Exam 12/1/11**

Name \_\_\_\_\_

There are 7 questions worth a total of 100 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 following references:

- Your primary compiler textbook(s)
- Course lecture slides and notes
- Exam topic list
- x86-64 Instructions and ABI (Univ. of Chicago handout)
- MiniJava Grammar (included on the next page)

No other books or other materials, including old exams or homework. You may use electronic copies of the materials but may not access any other information on the internet, or other resources on your computer.

Please wait to turn the page until everyone is told to begin.

**CSE P 501 Exam 12/1/11**

Score \_\_\_\_\_

1 \_\_\_\_\_ / 10

2 \_\_\_\_\_ /10

3 \_\_\_\_\_ /18

4 \_\_\_\_\_ /22

5 \_\_\_\_\_ /17

6 \_\_\_\_\_ /13

7 \_\_\_\_\_ /10

## CSE P 501 Exam 12/1/11

**Question 1.** (10 points, 5 each) Regular expressions. Give regular expressions that generate the following sets of strings. You may only use the basic operations of concatenation, choice (`()`), and repetition (`*`) plus the derived operators `?` and `+`, and simple character classes like `[abc0-9]` and `[^a-z]`. You may use abbreviations like `vowels = [aeiou]`. You may not use more complex operators found in various software tools that handle extended regular expressions.

(a) All sequences of 0's and 1's that contain an odd number of 1's and contain at least one or more 1's.

(b) All strings containing a's, b's, and c's with at least one a and at least one b.

## CSE P 501 Exam 12/1/11

**Question 2.** (10 points) Context-free grammars. Consider the following syntax for expressions involving addition and field selection:

$$\begin{aligned} \text{expr} &::= \text{expr} + \text{field} \\ \text{expr} &::= \text{field} \\ \text{field} &::= \text{expr} . \text{id} \\ \text{field} &::= \text{id} \end{aligned}$$

(a) (6 points) Show that this grammar is ambiguous.

(b) (4 points) Give an unambiguous context-free grammar that fixes the problem(s) with the grammar in part (a) and generates expressions with `id`, field selection and addition. As in Java, field selection should have higher precedence than addition and both field selection and addition should be left-associative (i.e., `a+b+c` means `(a+b)+c`).

## CSE P 501 Exam 12/1/11

**Question 3.** (18 points) The (almost) obligatory LR-parsing question. Consider the following grammar that is meant to capture expressions involving addition (+) and exponentiation (\*\*), where addition is left associative and exponentiation is right associative.

0.  $e' ::= e \$$
1.  $e ::= e + t$
2.  $e ::= t$
3.  $t ::= x ** t$
4.  $t ::= x$

(a) (10 points) Draw the LR(0) state machine for this grammar. (You do not need to include the table with shift/reduce and goto actions, although you can write that out later if you find it useful to answer other parts of the question.)

(continued next page)

## CSE P 501 Exam 12/1/11

**Question 3 (cont.)** Grammar repeated for reference

0.  $e' ::= e \$$
1.  $e ::= e + t$
2.  $e ::= t$
3.  $t ::= x ** t$
4.  $t ::= x$

(b) (2 points) Is this grammar LR(0)? Why or why not?

(c) (4 points) Compute First, Follow, and Nullable for each of the non-terminals in this grammar.

(d) (2 points) Is this grammar SLR? Why or why not?

## CSE P 501 Exam 12/1/11

**Question 4.** (22 points) We would like to add a new loop to MiniJava that is similar to a while loop except that it can have multiple conditions and statement sequences in a loop.

The simple case has one condition and a sequence of statements. For example, the loop

```
do i<n => sum=sum+a[i]; i++; od
```

is equivalent to the traditional loop

```
while (i<n) { sum=sum+a[i]; i++; }
```

The difference is that parentheses are not required around the condition, the terminal symbol => separates the condition from the loop body, the end of the loop is indicated by the keyword od with no semicolon or other punctuation after it, and the loop body can have several statements without requiring curly braces because the => and od symbols indicate the extent of the loop body.

These loops can be nested in the obvious way since the new loop is just another kind of statement. For example, if we had 2-D arrays, a  $n \times n$  array could be initialized with

```
i=0;
do i<n => j=0;
    do j<n => a[i][j]=0; j++; od
    i++;
od
```

More interesting, a loop may contain more than one condition plus statement sequence group. The loop below changes positive integers  $x$  and  $y$  so that on termination both of them contain the greatest common divisor of their original values.

```
do x>y => x=x-y;
[] y>x => y=y-x;
od
```

If a loop contains more than one condition/statement sequence group, they are separated by a box made up of left and right brackets with no space between ([ ]). To execute the loop the conditions are evaluated in order from the beginning. If a condition evaluates to true, the corresponding statement(s) following the => symbol up to the next [] or od are executed, and then that iteration of the loop is finished. Execution of the next iteration of the loop begins back at the top after do. If no condition is true, then execution of the loop terminates. In other words, on each iteration of the loop, at most one sequence of statements is executed – the statements following the first true condition in the list – and later conditions and statements in the loop are not executed on that iteration.

(You may detach this page from the exam while working, but must turn it in with the rest of the exam.)

## CSE P 501 Exam 12/1/11

**Question 4 (cont.)** Answer the following questions about the new loop statement.

(a) (6 points) Give an unambiguous context free grammar rule or rules to add this new kind of loop to the MiniJava grammar for *Statement*. Your answer should include additional new terminals and non-terminals as needed and can include additional grammar rules for non-terminals if appropriate. You only need to give the additions and changes to the MiniJava grammar; you do not need to write CUP or other parser-generator source code.

(b) (3 points) What changes need to be made to the MiniJava scanner to add this new loop statement to the language? Again, just describe the changes. You do not need to write JFlex/CUP or other source code.

## CSE P 501 Exam 12/1/11

**Question 4 (cont.)** (c) (4 points) What changes or additions need to be made to the MiniJava Abstract Syntax Tree classes or node definitions to include this new loop statement in the MiniJava abstract grammar? Your answer should give a description of the kinds of nodes that need to be added or changed and their contents. It does not need to be detailed Java, C#, or other code.

(d) (3 points) What checks need to be added to the static semantics/type checking phase of the compiler for this new loop statement?

## CSE P 501 Exam 12/1/11

**Question 4. (cont.)** (e) (6 points) Outline the code shape (essentially the pseudo-assembly language) needed to implement this new loop statement. Your answer should show the code shape needed for a loop like the following one with two condition/statement sequence pairs:

```
do cond1 => stmt1
  [ ] cond2 => stmt2
od
```

The answer should be similar to the examples given in class for other control constructs showing where labels and jumps would appear and where the code for the conditions and statements inside the loop would be placed.

## CSE P 501 Exam 12/1/11

**Question 5.** (17 points) Consider the following C data structure definition and function. (This code uses the gcc convention that long is a 64-bit integer type):

```
struct node {      // node with 64-bit int and pointer
    long val;
    struct node * next;
};
long sum(struct node * p) {
    long first, rest;
    if (p == NULL) {
        return 0;
    } else {
        first = p->val;
        rest = sum(p->next);
        return first+rest;
    }
}
```

This question involves translating this function to x86-64 assembler code. Ground rules:

- You may use either Linux/gcc or Microsoft/masm assembly language, and must follow the corresponding register linkage and stack frame conventions.
  - Linux argument registers: rdi, rsi, rdx, rcx, r8, r9
  - Linux: called function must save/restore rbx, rbp, r12-r15 if used.
  - Microsoft argument registers: rcx, rdx, r8, r9; caller must provide a save area for these four parameter registers at the bottom of the caller's stack frame that the called function can use if desired.
  - Microsoft: called function must save/restore rbx, rsi, rdi, rbp, r12-r15 if used.
  - Both: function result returned in rax.
  - Both: rsp must be aligned on a 16-byte boundary when a call instruction is executed to call the function.
- Pointers and ints are 64 bits (8 bytes) each.
- Your code should implement all of the statements in the original function. In particular, it should include the recursive function call and include store instructions for assignments to local variables, and may not rewrite the function into something different like a loop that produces the same result. Other than that, you can use any reasonable x86-64 code that follows the standard function call and register conventions. (In particular, if a previously computed value is still in a register when needed later in the code, you don't need to include an additional instruction to reload it from memory.)
- Assume the representation of a NULL pointer is an 8-byte binary zero (0) value.
- A C struct is a simple record type. It is not an object with a method table pointer or other hidden data fields.

(You may detach this page from the exam while working, but must turn it in with the rest of the exam.)

## CSE P 501 Exam 12/1/11

**Question 5 (cont.)** Code repeated for reference.

```
long sum(struct node * p) {
    long first, rest;
    if (p == NULL) {
        return 0;
    } else {
        first = p->val;
        rest = sum(p->next);
        return first+rest;
    }
}
```

```
struct node {
    long val;
    struct node * next;
};
```

(a) (0 points) Which conventions are you using (circle)

Linux/gcc

Microsoft/masm

(b) (5 points) Draw the stack frame for function `sum` as it would appear in an x86-64 program using the function call conventions you indicated above. Your picture should show the locations of function parameters (if they occupy storage), local variables, temporaries, and the stack pointer and frame pointer registers as they exist after the function prologue has executed and has allocated the stack frame, but before any of the statements in the body of the function have been executed. Be sure to show the numeric offsets from the frame pointer register to each local variable and other assigned storage locations.

(continued next page)

## CSE P 501 Exam 12/1/11

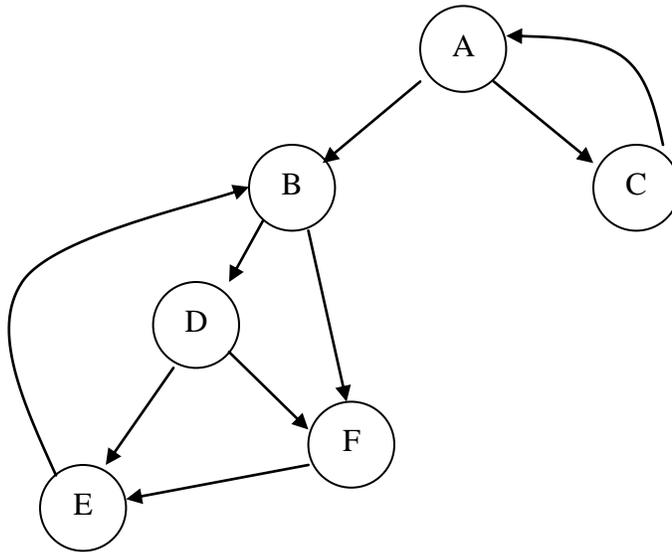
**Question 5 (cont.)** (c) (12 points) Translate function `sum` to x86-64 assembly language. Be sure to follow the rules given on the first page of the question (hint: read the rules again after you've finished your answer.) Code repeated for reference.

```
long sum(struct node * p) {
    long first, rest;
    if (p == NULL) {
        return 0;
    } else {
        first = p->val;
        rest = sum(p->next);
        return first+rest;
    }
}

struct node {
    long val;
    struct node * next;
};
```

**CSE P 501 Exam 12/1/11**

**Question 6.** (13 points) Dominators and loops. Consider the following flow graph.



(a) (9 points) For each node, list the nodes that are its dominators and its immediate dominator. A is the initial node in the flow graph.

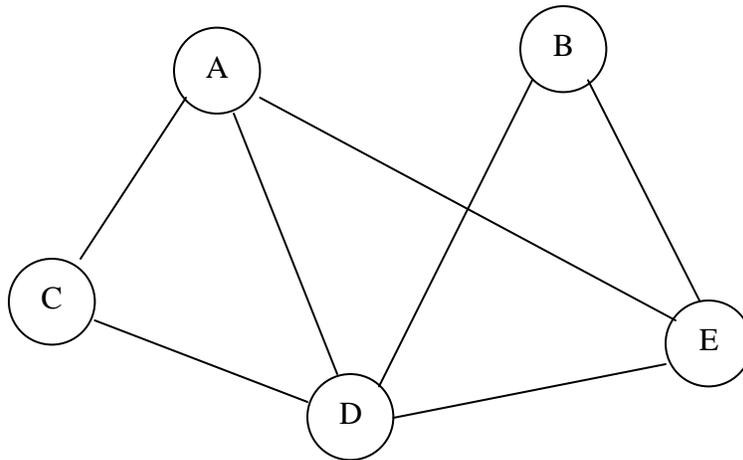
Node	Dominators	IDOM
A		
B		
C		
D		
E		
F		

(b) (4 points) List the *back edges* in the flow graph (i.e.,  $x \rightarrow y$  where the edge  $x \rightarrow y$  is a back edge). For each back edge, list the set of nodes that form the *natural loop* associated with that back edge. (There are likely more rows in the table below than you need, but add more if you need them.)

Back edge	Nodes in the associated natural loop

## CSE P 501 Exam 12/1/11

**Question 7.** (10 points) Register coloring. Suppose we have the following interference graph involving five live ranges A-E.



We would like to use the graph coloring register allocation algorithm to discover if there is a way to successfully allocate these five live ranges to three registers R1, R2, and R3.

(a) (5 points) List the nodes in the order they would be removed from the graph and placed on the stack during the simplify phase of the graph coloring algorithm. If there is more than one possible ordering you should list any one of them. If the simplify algorithm stops because it is not possible to remove any of the remaining nodes from the graph, indicate which nodes remain in the graph when the algorithm terminates.

(b) (5 points) Is it possible to assign the live ranges to three registers R1, R2, and R3 without interference? If so give an assignment based on your answer to part (a). If it is not possible, explain why not.