

Friday 2/29

**ANSWER ALL QUESTIONS – USE EXTRA PAPER AS NECESSARY**

This exam is open book, open notes; use of any electronic devices is prohibited. Further, work on this exam must be your own. Cheating will not be tolerated.

1. [10 POINTS] Consider the following C program skeleton:

```
/** CSCI-4430/CSCI-6969 Programming Languages **/  
/** Date: 2/29/2008 **/  
  
#include <stdio.h>  
  
void doSomething(int t, int u) /* DECLARATION A */  
{  
    int u, v; /* DECLARATION B */  
    ...  
    for ( ... )  
    {  
        int t, u, w; /* DECLARATION C */  
        ... ← POINT ONE  
        while ( ... )  
        {  
            int t, w, c; /* DECLARATION D */  
            ... ← POINT TWO  
        }  
        ... ← POINT THREE  
    }  
    ... ← POINT FOUR  
}
```

For each of the four marked points in this function, list each visible variable, along with the declaration (A, B, C, or D) that declares each. For example, at POINT ONE, variable w is visible based on declaration C, etc.

Write your answers in the space below:

POINT ONE: t (C), u (C), w (C), v (B)

POINT TWO: t (D), w (D), c (D), u (C), v (B)

POINT THREE: t (C), u (C), w (C), v (B)

POINT FOUR: u (B), v (B), t (A)

Each one worth 2  
points (for a total of 8)

So 2 free points.

go on to the next page →

2. [10 POINTS] Using dynamic-scoping in the given C program, what is the exact output for the call sequences listed below?

```

/** CSCI-4430/CSCI-6969 Programming Languages  **/
/** Date: 2/29/2008                               **/

#include <stdio.h>

void f1();
void f2();

main()
{
    int a = 30, b = 40;
    printf("%d ==> %d\n", a, b);
    /** always call a function here **/
    printf("%d ==> %d\n", a, b);
}

void f1()
{
    int a = 40, b = 30;
    printf("%d ==> %d\n", a, b);
    /** sometimes call a function here **/
    printf("%d ==> %d\n", a, b);
}

void f2()
{
    int a = 50;
    printf("%d ==> %d\n", a, b);
    /** sometimes call a function here **/
    printf("%d ==> %d\n", a, b);
}

```

- a. Call sequence: `main()` calls `f2()`.

30 ==> 40 \n 50 ==> 40 \n 50 ==> 40 \n 30 ==> 40 \n

- b. Call sequence: `main()` calls `f1()`, which calls `f2()`.

30 ==> 40 \n 40 ==> 30 \n 50 ==> 30 \n  
50 ==> 30 \n 40 ==> 30 \n 30 ==> 40 \n

- c. Call sequence: `main()` calls `f2()`, which calls `f1()`.

30 ==> 40 \n 50 ==> 40 \n 40 ==> 30 \n  
40 ==> 30 \n 50 ==> 40 \n 30 ==> 40 \n

Each one worth 0.5  
points (for a total of 8)

So 2 more free points.

go on to the next page →

3. [12 POINTS] Consider top-down parsing and the following BNF grammar:

```

<a> --> <id> := <e> ;
<id> --> a | b | c | d | ... | x | y | z
<e> --> <e> + <t> | <t>
<t> --> <t> * <f> | <f>
<f> --> ( <e> ) | <id> | <u> <id> | <id> <u>
<u> --> ++ | --

```

a. Show a leftmost derivation to obtain the following valid sentence:

s := m \* (++o + p);

```

<a> => <id> := <e> ;
=> s := <e> ;
=> s := <t> ;
=> s := <t> * <f> ;
=> s := <f> * <f> ;
=> s := <id> * <f> ;
=> s := m * <f> ;
=> s := m * ( <e> ) ;
=> s := m * ( <e> + <t> ) ;
=> s := m * ( <t> + <t> ) ;
=> s := m * ( <f> + <t> ) ;
=> s := m * ( <u> <id> + <t> ) ;
=> s := m * ( ++ <id> + <t> ) ;
=> s := m * ( ++ o + <t> ) ;
=> s := m * ( ++ o + <f> ) ;
=> s := m * ( ++ o + <id> ) ;
=> s := m * ( ++ o + p ) ;

```

Each part worth 4 points

Part (a) order matters, as does replacement of only one leftmost abstraction at each step

Part (b) solution not shown

Part (c) is broken down into 2 points for addressing direct left recursion of <e> and <t>; 2 points for <f> and <id>

b. Draw a parse tree corresponding to the leftmost derivation from (a).

c. Rewrite the grammar using only BNF and  $\epsilon$  such that all rules pass the pairwise disjointness test and all direct left recursion is factored out.

```

<a> --> <id> := <e> ;
<id> --> a | b | c | d | ... | x | y | z
<e> --> <t> <e'>
<e'> --> + <t> <e'> |  $\epsilon$ 
<t> --> <f> <t'>
<t'> --> * <f> <t'> |  $\epsilon$ 
<f> --> ( <e> ) | <u> <id> | <id> <v>
<v> --> <u> |  $\epsilon$ 
<u> --> ++ | --

```

Also for <e> and <t>:

```

<e> --> <t> {+ <t>}
<t> --> <f> {* <f>}

```

go on to the next page →

4. [10 POINTS] Given the following grammar:

```
S --> aAB | bC | aCb
A --> ab | bC | b
B --> b | bbBbb
C --> c | Ac | b
```

Part (a) is worth up to 8 points:  
 +3+3+2 points for each  
 correct one circled  
 -1 point for each incorrectly  
 circled answer  
 Part (b) is worth 2 points

a. Of the sentences below, **circle** all sentences generated by this grammar:

**abbccb**

**babc**

**bbbccc**

bbbacc

aabbbbbbb

cbabbbbb

abc

ababc

acbbca

b. Does this grammar pass the pairwise disjointness test? Why or why not?

**NO! Two rules from s lead to terminal symbol a.**

5. [10 POINTS] Write preprocessor directives using C for the following cases:

a. Write C preprocessor directive `MIN(A,B)`, which determines the minimum value of arguments `A` and `B`.

```
#define MIN(A,B) ((A) < (B) ? (A) : (B))
```

Parts (a) and (b) worth 3 points each;  
 Part (c) worth 4 points

Part (a): -2 if no parens around A & B

b. Write C preprocessor directive `SORT_AND_PRINTLN(A,B)`, which displays integer variables `A` and `B` in sorted order, followed by a newline character.

```
#define MAX(A,B) ((A) > (B) ? (A) : (B))
#define SORT_AND_PRINTLN(A,B)
    printf("%d %d\n", MIN(A,B), MAX(A,B))
```

Part (b) can be ascending or descending

c. Write C preprocessor directives `CASE(EXPR, CODE)` and `DEFAULT(CODE)`, which elaborate to cases in a C-style case statement, as shown below:

```
#define CASE(EXPR, CODE) case (EXPR) : CODE break;
```

```
/** example usage */
...
switch (x) {
    CASE(1, printf("ONE\n"));
    CASE(2, printf("TWO\n"));
    ...
    DEFAULT(printf("N/A\n"));
}
...
```

Part (c) must include `break`; (-2 if not)

Part (c): parens around `EXPR` are optional

go on to the next page →

6. [12 POINTS] Consider a Fortran DO loop, as shown in the example below in which variable `I` counts from 1 to 99 with a `STEP` of 2:

```

                DO 2000 I = 1, 99, 2
                ...
2000           CONTINUE

```

Each part worth 4 points; no EBNF!

Part (b) must show each step

- a. Write a grammar using strict BNF and  $\epsilon$  for the Fortran DO loop (note that the `STEP` is optional and Fortran statements are line-based).

```

<loop> --> DO <label> id = int-lit , int-lit <step> \n
                <stmts> or ... \n
                <label> CONTINUE \n
<step> --> , int-lit |  $\epsilon$ 

```

- b. Show a derivation for the example Fortran loop code above.

```

<loop> => DO <label> id = int-lit , int-lit <step> \n
                <stmts> \n <label> CONTINUE \n
=> DO 2000 id = int-lit , int-lit <step> \n
                <stmts> \n <label> CONTINUE \n
=> DO 2000 I = int-lit , int-lit <step> \n
                <stmts> \n <label> CONTINUE \n
=> DO 2000 I = 1 , int-lit <step> \n
                <stmts> \n <label> CONTINUE \n
=> DO 2000 I = 1 , 99 <step> \n
                <stmts> \n <label> CONTINUE \n
=> DO 2000 I = 1 , 99 , int-lit \n
                <stmts> \n <label> CONTINUE \n
=> DO 2000 I = 1 , 99 , 2 \n
                <stmts> \n <label> CONTINUE \n
=> DO 2000 I = 1 , 99 , 2 \n
                ... \n <label> CONTINUE \n
=> DO 2000 I = 1 , 99 , 2 \n
                ... \n 1000 CONTINUE \n

```

- c. Incorporate semantics information into your grammar to form an attribute grammar that ensures the `LABEL` (e.g. 2000) matches at the beginning and end of the loop.

Syntax rule:

```

<loop> --> DO <label>[1] id = int-lit , int-lit <step> \n
                <stmts> or ... \n
                <label>[2] CONTINUE \n

```

Predicate:

```

<label>[1].string == <label>[2].string

```

go on to the next page →

7. [12 POINTS] Consider the Perl code below:

```
my $j = 19;
for ( my $j = 20 ; $j > 13 ; $j-- ) {
    print $j;
}

$j *= 2;
print "\n$j\n";
```

Each part worth 4 points; exact output required, though okay if spaced improperly. Newlines required; for Parts (a) and (b), -2 if last output line incorrect

a. What is the exact output of the given Perl code?

```
20191817161514
38
```

b. What is the exact output if you remove the reserved word `my` from the `for` loop initializer?

```
20191817161514
26      because my creates new variable $j in loop scope
```

c. Using only the operational semantics language shown below, write operational semantics code for the Perl code from part (a) above.

```
integer identifier
identifier = integer-literal
identifier = identifier + integer-literal
identifier = identifier * integer-literal
if identifier <> identifier goto label
if identifier = identifier goto label
print identifier
println
```

Note that *identifier* is a string of one or more lowercase letters. Further, `print` outputs the specified *identifier*, whereas `println` simply outputs a newline character.

```
integer j
j = 19
integer jlocal
integer k
k = 13
jlocal = 20
loop: print jlocal
jlocal = jlocal + -1
if jlocal <> k goto loop
j = j * 2
println
print j
println
```

go on to the next page →

8. [12 POINTS] Consider the given grammar and its corresponding LR parsing table below:

1.  $E \rightarrow E + T$
2.  $E \rightarrow T$
3.  $T \rightarrow T * F$
4.  $T \rightarrow F$
5.  $F \rightarrow ( E )$
6.  $F \rightarrow id$

Each part is worth 6 points; partial credit where possible.

State	Action						Goto		
	id	+	*	(	)	\$	E	T	F
0	S5		S4				1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

a. Show a complete parse, including the parse stack contents, the input string, and the actions for valid sentence (id + id).

<i>Stack</i>	<i>Input</i>	<i>Action</i>
0	( id + id ) \$	Shift 4
0(4	id + id ) \$	Shift 5
0(4id5	+ id ) \$	Reduce 6 (GOTO [4,F])
0(4F3	+ id ) \$	Reduce 4 (GOTO [4,T])
0(4T2	+ id ) \$	Reduce 2 (GOTO [4,E])
0(4E8	+ id ) \$	Shift 6
0(4E8+6	id ) \$	Shift 5
0(4E8+6id5	) \$	Reduce 6 (GOTO [6,F])
0(4E8+6F3	) \$	Reduce 4 (GOTO [6,T])
0(4E8+6T9	) \$	Reduce 1 (GOTO [4,E])
0(4E8	) \$	Shift 11
0(4E8)11	\$	Reduce 5 (GOTO [0,F])
0F3	\$	Reduce 4 (GOTO [0,T])
0T2	\$	Reduce 2 (GOTO [0,E])
0E1	\$	Accept

b. Show a parse, including the parse stack contents, the input string, and the actions for invalid sentence id + + id. Stop when the error is detected.

<i>Stack</i>	<i>Input</i>	<i>Action</i>
0	id + + id \$	Shift 5
0id5	+ + id \$	Reduce 6 (GOTO [0,F])
0F3	+ + id \$	Reduce 4 (GOTO [0,T])
0T2	+ + id \$	Reduce 2 (GOTO [0,E])
0E1	+ + id \$	Shift 6
0E1+6	+ id \$	ERROR!

go on to the next page →

9. [12 POINTS] Fill in the missing C code below (where indicated) to implement a lexical analyzer that accepts Perl identifiers for scalars, arrays, and hashes, including the \$, @, and % prefixes. A valid Perl identifier consists of alphanumeric characters and the underscore character, but must begin with either a letter or underscore character. Remember that Perl is case-sensitive.

```

/** CSCI-4430/CSCI-6969 Programming Languages **/
/** Date: 2/29/2008 **/

/** assume charClass, addChar(), getChar() are available **/
...

/** character classes: **/
#define LETTER 1 /** [A-Za-z] **/
#define DIGIT 2 /** [0-9] **/
#define DOLLAR_SIGN 3
#define AT_SIGN 4
#define PERCENT_SIGN 5
#define UNDERSCORE 6

/** tokens: **/
#define SCALAR_IDENTIFIER 1 /** $ident **/
#define ARRAY_IDENTIFIER 2 /** @ident **/
#define HASH_IDENTIFIER 3 /** %ident **/
#define UNKNOWN_TOKEN 4
int t;

int lex()
{
    static int first = 1;
    if (first) { getChar(); first = 0; }
    lexemeLength = 0;

    switch (charClass)
    {
        case DOLLAR_SIGN:
        case AT_SIGN:
        case PERCENT_SIGN:
            if (charClass == DOLLAR_SIGN) { t = SCALAR_IDENTIFIER; }
            if (charClass == AT_SIGN) { t = ARRAY_IDENTIFIER; }
            if (charClass == PERCENT_SIGN) { t = HASH_IDENTIFIER; }
            addChar(); getChar();
            if ( charClass == LETTER || charClass == UNDERSCORE ) {
                addChar(); getChar();
                while ( charClass == LETTER || charClass == UNDERSCORE ||
                    charClass == DIGIT ) {
                    addChar(); getChar();
                }
                return t;
            } else {
                return UNKNOWN_TOKEN; /** or MOD_OPERATOR, etc. **/
            }
            break;
    }
    ...
}
}

```

The first two parts of the code (adding constants) is worth 4 points; lex() code is worth 8 points; certainly other variations are possible (e.g. combined cases for \$, @, %)

all done! ☺