Fall, 2023 Name: ______________________________

(Please *don't* write your id number!)

COP 3402 — Systems Software

Final Exam

Directions for this Test

This test has 10 questions and pages numbered 1 through 7. This test will be for the entire time allowed in class and is closed book. However, you may use one (1) page of notes on one (1) side of a standard 8.5 by 11 inch sheet of paper. These notes can either be hand-written or printed, but if printed, then the font must be a 9-point or larger font. These notes must be turned in with the exam.

For multiple choice questions, the directions in the problem will say either:

- "(Circle the **one** correct answer's letter.)", indicating that you should circle exactly one (1) answer, or
- "(Circle **each** answer letter that is correct.)", indicating that you should circle all the right answers, so you will only get full credit for choosing all of them and no wrong answers. We will take some points off for wrong answers.

For Grading

Question:	1	2	3	4	5	6	7	8	9	10	Total
Points:	5	10	10	15	10	10	10	10	10	10	100
Score:											

All questions on this exam are related to the course outcome [UseConcepts] and to some extent to the course outcome [Build].

Reference Material

There is some reference material that you can use during the exam on the next page.

Reference for SRM Assembly Language

Table 1 shows some SRM assembly language instructions that are used in this exam. Sign-extension, which is written as "sgnExt" extends a 16-bit immediate value i to a 32-bit integer. For example, if i is -1, which is FFFF in hexadecimal; then sgnExt(i) is FFFFFFFF in hexadecimal, which also represents -1. For branch instructions, the immediate value, o is first shifted left 2 bits (multiplied by 4) and then sign-extended, which is written as "formOffset" in the table. (Thus formOffset(o) $=$ sgnExt$(4 \times o)$.) Note that the resulting address is added to the address of the instruction following the currently executing instruction, not the address of the instruction itself, since the *PC* has already been advanced.

PL/0 Context-Free Syntax

The context-free syntax of PL/0 is defined by the context-free grammar shown in Figure 1.

Table 1: Some immediate format instructions used in the exam.

Name	op	rs	rt	immed	Comment (Explanation)
ADDI	9	s	t	i	Add immediate: GPR[t] $\leftarrow$ GPR[s] + sgnExt(i)
BEQ	4	s	t	o	Branch on Equal: **if** GPR[s] = GPR[t] **then** $PC \leftarrow PC$ + formOffset(o)
BNE	5	s	t	o	Branch Not Equal: **if** GPR[s] $\neq$ GPR[t] **then** $PC \leftarrow PC$ + formOffset(o)
LW	35	b	t	o	Load Word (4 bytes): GPR[t] $\leftarrow$ memory[GPR[b] + formOffset(o)]
SW	43	b	t	o	Store Word (4 bytes): memory[GPR[b] + formOffset(o)] $\leftarrow$ GPR[t]

⟨program⟩ ::= ⟨block⟩ `.`
⟨block⟩ ::= ⟨const-decls⟩ ⟨var-decls⟩ ⟨proc-decls⟩ ⟨stmt⟩
⟨const-decls⟩ ::= **{**⟨const-decl⟩**}**
⟨const-decl⟩ ::= **`const`** ⟨const-defs⟩ `;`
⟨const-defs⟩ ::= ⟨const-def⟩ | ⟨const-defs⟩ `,` ⟨const-def⟩
⟨const-def⟩ ::= ⟨ident⟩ `=` ⟨number⟩
⟨var-decls⟩ ::= **{**⟨var-decl⟩**}**
⟨var-decl⟩ ::= **`var`** ⟨idents⟩ `;`
⟨idents⟩ ::= ⟨ident⟩ | ⟨idents⟩ `,` ⟨ident⟩
⟨proc-decls⟩ ::= **{**⟨proc-decl⟩**}**
⟨proc-decl⟩ ::= **`procedure`** ⟨ident⟩ `;` ⟨block⟩ `;`
⟨stmt⟩ ::= ⟨assign-stmt⟩ | ⟨call-stmt⟩ | ⟨begin-stmt⟩ | ⟨if-stmt⟩
  | ⟨while-stmt⟩ | ⟨read-stmt⟩ | ⟨write-stmt⟩ | ⟨skip-stmt⟩
⟨assign-stmt⟩ ::= ⟨ident⟩ `:=` ⟨expr⟩
⟨call-stmt⟩ ::= **`call`** ⟨ident⟩
⟨begin-stmt⟩ ::= **`begin`** ⟨stmts⟩ **`end`**
⟨if-stmt⟩ ::= **`if`** ⟨condition⟩ **`then`** ⟨stmt⟩ **`else`** ⟨stmt⟩
⟨while-stmt⟩ ::= **`while`** ⟨condition⟩ **`do`** ⟨stmt⟩
⟨read-stmt⟩ ::= **`read`** ⟨ident⟩
⟨write-stmt⟩ ::= **`write`** ⟨expr⟩
⟨skip-stmt⟩ ::= **`skip`**
⟨stmts⟩ ::= ⟨stmt⟩ `|` ⟨stmts⟩ `;` ⟨stmt⟩
⟨condition⟩ ::= ⟨odd-condition⟩ | ⟨rel-op-condition⟩
⟨odd-condition⟩ ::= **`odd`** ⟨expr⟩
⟨rel-op-condition⟩ ::= ⟨expr⟩ ⟨rel-op⟩ ⟨expr⟩
⟨rel-op⟩ ::= `=` | `<>` | `<` | `<=` | `>` | `>=`
⟨expr⟩ ::= ⟨term⟩ | ⟨expr⟩ ⟨plus⟩ ⟨term⟩ | ⟨expr⟩ ⟨minus⟩ ⟨term⟩
⟨term⟩ ::= ⟨factor⟩ | ⟨term⟩ ⟨mult⟩ ⟨factor⟩ | ⟨term⟩ ⟨div⟩ ⟨factor⟩
⟨factor⟩ ::= ⟨ident⟩ | ⟨minus⟩ ⟨number⟩ | ⟨pos-sign⟩ ⟨number⟩ | `(` ⟨expr⟩ `)`
⟨pos-sign⟩ ::= ⟨plus⟩ | ⟨empty⟩
⟨empty⟩ ::=

Figure 1: Context-free grammar for the concrete syntax of PL/0.

1. (5 points) Consider the following PL/0 program:

```
const one = 1, two = 2;
var i, j, k;
var k;
begin
  i := one;
  k := one * two;
  j := k+k
end.
```

What, if anything, would be flagged by a PL/0 compiler as an error in the above program? (Circle the **one** correct answer's letter.)

   A. The identifier `k` is declared twice.

   B. The identifier `j` is undeclared.

   C. Nothing, the program is fine as it is.

   D. The program has a syntax error, as it has an extra comma (`,`).

   E. The constant declaration of `two` is silly.

2. (10 points) Consider the following C function.

```
int test(int a, int b)
{
    int c;
    c = 3;
    {
        int d = a + b;
        int c;
        c = d + 10;
        d = c - 10;
    }
    return c;
}
```

What, if anything, would be the value of the expression `test(1,5)` in a C program? (Circle the **one** correct answer's letter.)

   A. Nothing, the function `test` has multiple declarations of the variable `c`, so it will not compile.

   B. Nothing, the variables `a`, and `b` are not declared in the function `test`, so it will not compile.

   C. 16, because the last assignment to `c` is `d+10` and `d` has the value 6

   D. 3, because the return statement returns the value of the variable `c` declared in the outer block.

   E. 19, because 3+6+10 is 19.

3. (10 points) In the Simplified RISC Machine (SRM), there are 32 registers. Which of the following places can a code generator always put the value of an expression to be sure that the expression's value will be available without being overwritten during the computation of some other expression's value? (Circle the **one** correct answer's letter.)

   A. On top of the runtime stack.

   B. In some register; although the register used may vary, there are always enough registers to store the value of each expression.

   C. In the $v0 register.

   D. In the location declared with the name `ret` that is in the current activation record.

4. (15 points) This is a question about the code needed for if-statements in PL/0. Consider the following PL/0 program.

```
var x;
begin
  read x;
  if x > 0 then write 1 else write -1
end.
```

Assume that each of the write statements in the above program is compiled into a sequence of instructions of size 6, and that the code for evaluating the condition puts the condition's truth value on the top of the runtime stack.

What sequence of SRM assembly language instructions would correctly test the result of the condition and jump over the compiled code for the statement **`write`** `1` when the condition is false, to start executing the statement **`write`** `-1`? Note that the compiled code must *not* execute both write statements. (Circle the **one** correct answer's letter.)

A.
```
LW $sp, $v0, 0      # offset is +0 bytes
ADDI $sp, $sp, -1
BNE $v0, $0, 7      # offset is +28 bytes
```

B.
```
LW $sp, $v0, 0      # offset is +0 bytes
ADDI $sp, $sp, 4
BEQ $v0, $0, 7      # offset is +28 bytes
```

C.
```
LW $sp, $v0, 0      # offset is +0 bytes
ADDI $sp, $sp, 1
BNE $0, $v0, 6      # offset is +24 bytes
```

D.
```
LW $sp, $v0, 0      # offset is +0 bytes
ADDI $sp, $sp, 7
BNE $0, $v0, 28     # offset is +112 bytes
```

5. (10 points) This is a question about the compilation of variable declarations in PL/0. Which of the following describes the recommended code sequence that should be generated for a variable declaration in PL/0? (Circle the **one** correct answer's letter.)

   A. Initialize the next word on the runtime stack to 0, without changing the stack pointer (`$sp`) register's value.

   B. Branch over the next two instructions using `BEQ $0, $0, 2`.

   C. Load the variable's initial value from the data section into a register, and then push that register on the runtime stack.

   D. Allocate a word on the runtime stack and initialize that to 0, thereby pushing 0 on the runtime stack.

   E. Load a 0 into register `$a0` and then execute the `PINT` instruction.

6. (10 points) Which of the following regular expressions could be used by flex to recognize an integer literal with a possible leading minus sign, such as `3402`, `-0`, `55`, or `-34021895`? (Circle the **one** correct answer's letter.)

   A. `-?[0-9A-F]*`

   B. `[1-0]*`

   C. `-?[0-9]+`

   D. `[1-0]+`

   E. `[1-0][1-0]*`

7. (10 points) Which of the following are inputs or outputs of the bison parser generator? (Circle **each** answer letter that is correct.)

   A. A regular grammar is part of an input.

   B. A context-free grammar is part of an input.

   C. Actions to take when recognizing each production in the grammar are part of an input.

   D. A code for a parser is an output.

   E. Directions on how to build a parser module is an output.

8. (10 points) How is a symbol table used in a compiler? (Circle **each** answer letter that is correct.)

   A. It tracks the offset of each declared identifier in each scope.

   B. In each scope it tells the meaning of each symbol.

   C. It tracks each identifier's distance from the beginning of the program.

   D. It remembers the lexical address that the parser tells it for each identifier.

   E. In each scope it maps declared identifiers to their attributes, which can be queried to see if an identifier was declared in that scope.

9. (10 points) This is a question about generating code for assignment statements in PL/0. Which of the following best describes the code generated for an assignment statement? (Circle the **one** correct answer's letter.)

   A. The generated code evaluates the assignment statement's expression, which determines the variable's lexical address, then a value is stored into the variable by guessing its offset from the $fp register's value.

   B. The generated code evaluates the assignment statement's expression, that expression's value is put in a register, and then that register is stored into the variable using the variable's offset and the frame pointer for the AR in which it was declared.

   C. The generated code evaluates the assignment statement's variable, puts its value into a register, then puts the statement's expression's value into that same register.

   D. The generated code evaluates the assignment statement's expression, then it searches back through the runtime stack to find the most recent declaration of the statement's value, and stores the expression's value into that location.

10. (10 points) Which of the following is a context-free grammar that recognizes a language that is parentheses-balanced (i.e., in which each left parenthesis in the input is match by a right-parentheses in the input). Examples include: `((x))` and `((((y))))`, assuming that `x` and `y` are ⟨identifier⟩s, but not `(((x` or `y)`. (Circle the **one** correct answer's letter.)

    A. ⟨expr⟩ ::= `(` ⟨ending⟩
    ⟨ending⟩ ::= ⟨identifier⟩ | ⟨ending⟩ `)`

    B. ⟨expr⟩ ::= ⟨lefts⟩ ⟨identifier⟩ ⟨rights⟩
    ⟨lefts⟩ ::= `(` ⟨lefts⟩ | ⟨empty⟩
    ⟨empty⟩ ::=
    ⟨rights⟩ ::= ⟨empty⟩ `|` ⟨rights⟩ `)`

    C. ⟨expr⟩ ::= `(` ⟨expr⟩ `)` | ⟨identifier⟩

    D. ⟨expr⟩ ::= ⟨tokens⟩
    ⟨tokens⟩ ::= `(` | ⟨identifier⟩ | `)` | ⟨tokens⟩ ⟨empty⟩
    ⟨empty⟩ ::=