COS 217, Fall 2023
Midterm Exam

This exam consists of 6 questions, and you have 50 minutes – budget your time wisely. ***Do all of your work on these pages (using the back for scratch space), and give the answer in the space provided.*** Note that the exams will be scanned and graded online, so ***ONLY ANSWERS IN THE BOXES WILL BE GRADED.*** Assume the ArmLab/Linux/C/gcc217 environment unless otherwise stated. This is a closed-book, closed-note exam, and only 1 page of notes is allowed. Please place items that you will not need out of view in your bag or under your working space at this time. Electronic devices such as cell phones, laptops, tablets, etc. may not be used during this exam.

Name:		**NetID:**		**Precept:**	

P01	MW	1:30	Christopher Moretti
P02	MW	3:30	Christopher Moretti
P03	TTh	12:30	Guðni Nathan Gunnarsson
P04	TTh	12:30	Sam Ginzburg
P05	TTh	1:30	Indu Panigrahi
P06	TTh	1:30	Gongqi Huang
P07	TTh	2:30	Nanqinqin Li
P09	TTh	3:30	Jianan Lu
P10	TTh	7:30	Dwaha Daud

This examination is administered under the Princeton University Honor Code. Students should sit one seat apart from each other, and refrain from talking to other students during the exam. All suspected violations of the Honor Code must be reported to honor@princeton.edu.

Write out and sign the Honor Code pledge before turning in the test:

"I pledge my honor that I have not violated the Honor Code during this examination."

Pledge, written out exactly as above:

Sample Solutions

Signature:

Exam Statistics:
Max: 100/100 | Median: 79 | Mean: 77.74 | Standard Deviation:13.17

1. Build Process

For each item (a–g below), ***write the letter of the build stage*** from the list at the top that performs the action. ***Write your answers in the boxes at right.*** (2 pts ea)

P.	Preprocessor
C.	Compiler
A.	Assembler
L.	Linker

	Action	**Build stage (P, C, A, or L)**
(a)	Removes comments	**P** (recall from A1 that it maintains lines' numbering in its removal, however)
(b)	Generates assembly language from C source code	**C** (assembly language code is C's product and A's source)
(c)	Resolves references to `scanf`	**L** (fetches the machine code for standard library functions to add to the program)
(d)	Checks the declaration of `printf`	**C** (checks the declaration of functions for syntax and that calls to it match its types)
(e)	Generates machine language from assembly code	**A** (assembly language code is A's source, machine language code is its product)
(f)	Produces an executable	**L** (the final stage in the build process yields the final product)
(g)	Handles `#include <stdio.h>`	**P** (along with other preprocessor directives like #define)

2. Two's Complement

(a) Write the decimal number 14 as a ***5-bit unsigned binary number***, one bit per box: (2 pts)

0	**1** (8) +	**1** (4) +	**1** (2)	**0**

(b) Interpreting your answer in part (a) as a ***5-bit two's complement signed number***, what is its ***decimal*** value? (2 pts)

14
leftmost bit 0:
non-negative

(c) Write the decimal number 23 as a ***5-bit unsigned binary number***: (2 pts)

1 (16) +	**0**	**1** (4) +	**1** (2) +	**1** (1)

(d) Interpreting your answer in part (c) as a ***5-bit two's complement signed number***, what is its ***decimal*** value? (2 pts)

–9
(–16+4+2+1
or –(01001))

2. Two's Complement (cont.)

Sign extension is a procedure to increase the number of bits in a binary number. To perform it, take the ***leftmost*** bit of the original binary number, and add copies of the bit to the ***left*** of the binary number, until you reach the desired number of bits.

(e) Sign-extend your binary number from ***part (c) – not part (a)!*** – so that it is 6 bits long, and write down the resulting binary number. (1 pt)

1	1	0	1	1	1

(f) Interpreting your answer in part (e) as a ***6-bit two's complement signed number***, what is its ***decimal*** value? (2 pts)

–9
(–32+16+4+2+1
or –(001001))

(g) Based on your results in (d) and (f), what is the ***effect of sign extension on the two's complement signed value*** of a binary number? Write one phrase or sentence. (3 pts)

Sign extension preserves the value (including the sign) for any binary number, when interpreted as a two's complement signed number.

3. Pointers

Consider the following code:

```
#include <stdio.h>
void foo(int **ppi1, int **ppi2)
{
      int *piTemp;
      piTemp = *ppi2;
      *ppi2 = *ppi1;   /* THIS LINE WILL BE REPLACED IN PART (b) */
      *ppi1 = piTemp;
}
int main()
{
      int i1 = 10;
      int i2 = 20;
      int *pi1 = &i1;
      int *pi2 = &i2;

      foo(&pi1, &pi2);

      printf("%d %d %d %d\n", i1, i2, *pi1, *pi2);

      return 0;
}
```

(a) What are the values of `i1`, `i2`, `*pi1`, and `*pi2` printed in `main()`? (2 pts ea)

`i1:`	10 (foo swaps ptr not int vals)	`i2:`	20	`*pi1:`	20 (dereference of swapped ptr)	`*pi2:`	10 (dereference of swapped ptr)

(b) For each of the following, would ***replacing*** the line `*ppi2 = *ppi1;` with the proposed code result in ***different values*** being printed in `main()`? Write ***yes*** or ***no*** in the boxes at right (2 pts ea)

Replacement for `*ppi2 = *ppi1;`	**Different values printed? (yes/no)**
`*ppi1 = *ppi2;`	**Yes**: p1, p2 end up pointing to the same spot
`ppi2 = ppi1;`	**Yes**: changes foo's ppi2, not main's pi2
`ppi2[0] = ppi1[0];`	**No**: this is array syntax for the same operation
`*&*ppi2 = *&*ppi1;`	**No**: * and & are inverse ops, thus cancel out

4. DFA

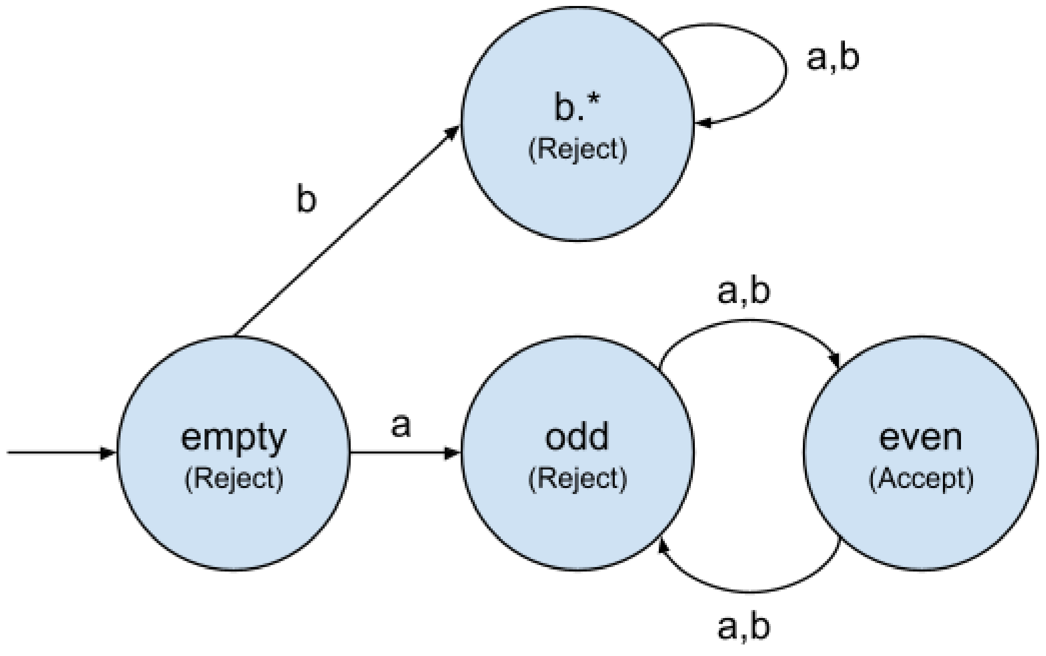
Construct a DFA that accepts strings consisting only of the letters **a** and **b**, satisfying both the following conditions:

- The string begins with an **a**
- The string has even length

Draw a DFA satisfying the above requirements ***in the box below***, as follows:

- Each ***state*** should be drawn as a circle. You do not have to give the states names, though doing so may aid in your design process.
- Each ***transition*** should be an arrow ***labeled with one or both*** of the letters **a** and **b**. You do not need to worry about any inputs other than the letters **a** and **b**.
- Clearly indicate the ***start*** state.
- Clearly indicate the ***accept*** state(s).

Hint: the minimal DFA for this task consists of 4 states. (14 pts)



5. Operators and operations

For each of the following code snippets, what output is printed? Please write your answers ***in the boxes to the right*** of the code snippets. Write “BAD” if the code results in a compiler error, an infinite loop, or an uninitialized value being printed. On the next page is a C operator table, for reference. (2 pts ea)

(a)	`int i = 260;` `unsigned char c = i;` `printf("%d\n", c);`	**4**: 260 > 255 (max unsigned char): overflow. 260 % 256 == 4
(b)	`int i, j = 0;` `for (i = 0; i < 10; i += 2)` `    j += i % 2;` `printf("%d\n", j);`	**0**: loop executes at i values {0, 2, 4, 6, 8}. i%2 == 0 for all those, so j is never changed.
(c)	`int i = 4, j = 0;` `while (i -= 2)` `    j += ++i;` `printf("%d\n", j);`	**5**: i←2 (-=), non-zero∴do body: i←3, j←3. i←1 (-=)∴do body: i←2, j←5. i←0 (-=)∴end
(d)	`int select = 2;` `switch (select) {` `    case 1: printf(“one\n”);` `    case 2: printf(“two\n”);` `    case 3: printf(“three\n”);` `    default: printf(“other\n”);` `}`	**two** **three** **other** there are no break statements, so case fallthrough occurs
(e)	`printf("%d\n", 2 == 2 + 2);`	**0**: per table on page 8, this is equivalent to 2 == (2+2)
(f)	`printf("%d\n", 2 == 2 | 2);`	**3**: equivalent to (2 == 2) | 2, 1 | 2 is: 0..01 | 0..10 = 0..11
(g)	`printf("%d\n", ~3 & 5);`	**4**: 3 is 0..011, so ~3 is 1..100 1..100 & 0..101 = 0..100 = 4.

Reference for Question 5:
C operators, in order of precedence (highest to lowest), with their associativity

`()` *(function call)* `[] . ->` `++ --` *(postfix)*	Left-to-right
`++ --` *(prefix)* `& *` *(address-of and pointer dereference)* `+ -` *(unary plus and minus)* `~ ! sizeof`	Right-to-left
`()` *(cast)*	Right-to-left
`* / %` *(multiplication, division, remainder)*	Left-to-right
`+ -` *(addition, subtraction)*	Left-to-right
`<< >>`	Left-to-right
`< > <= >=`	Left-to-right
`== !=`	Left-to-right
`&` *(bitwise and)*	Left-to-right
`^` *(bitwise xor)*	Left-to-right
`|` *(bitwise or)*	Left-to-right
`&&`	Left-to-right
`||`	Left-to-right
`?:`	Right-to-left
`= *= /= %= += -= <<= >>= &= ^= |=`	Right-to-left
`,`	Left-to-right

Reference for Question 6:
Information on standard library string functions

`size_t strlen(const char *s);`
Returns the length of the string pointed to by `s`, *excluding* the terminating null byte (`'\0'`).

`char *strcpy(char *dest, const char *src);`
Copies the string from `src`, *including* the terminating null byte, to `dest`. Returns `dest`.

6. Strings

In this problem you will consider multiple versions of this function:

```
/*
Exchanges the contents of the two string arguments, s1 and s2.
Precondition: strlen(s1) == strlen(s2).
*/
void strswap(char *s1, char *s2);
```

For example, if the caller's strings' contents are initially:

```
s1: {'h','o','u','s','e','\0'}
s2: {'g','u','e','s','t','\0'}
```

then after `strswap(s1, s2);` returns to the caller, they should be:

```
s1: {'g','u','e','s','t','\0'}
s2: {'h','o','u','s','e','\0'}
```

(a) Write a `strswap()` implementation in the box below. Your code ***must not*** call any functions, but ***must*** include two `assert`s checking the parameters (though not the precondition). (20 pts)

```
void strswap(char *s1, char *s2)
{

   assert( s1 != NULL);
   assert( s2 != NULL);

   while( *s1 != '\0') { /* precondition allows checking only 1 string */
      char cTemp = *s1;
      *s1 = *s2;
      *s2 = cTemp;
      s1++;
      s2++;
   }
   /* no need to swap '\0' with '\0' after loop */
}
```

6. Strings (cont.)

Now consider the following incorrect implementations of `strswap()`. Refer to page 8 for information on `strlen` and `strcpy`. When passed two strings ***with different contents*** satisfying the precondition, indicate in the box at right whether each:

> **A.** results in a compiler error or warning
>
> **B.** builds cleanly, but ***never*** results in successfully swapped strings
>
> **C.** results in successfully swapped strings ***sometimes but not always***

Write exactly one of the letters A, B, or C in the box at right. ***Hint: pay attention to the bolded-and-italicized text above. It matters.*** (2 pts ea)

	Code	**Result (A – C)**

(b)	`char temp[strlen(s1)+1];`
	`strcpy(temp, s1);`
	`strcpy(s1, s2);`
	`strcpy(s2, temp);`
		A: temp's declaration causes a compiler warning "ISO C90 forbids variable length array", since its length is not known at compile time.

(c)	`char *temp = s1;`
	`s1 = s2;`
	`s2 = temp;`
		B: this swaps strswap's pointer parameters, but has no impact on the strings they reference in the caller.

(d)	`char temp = *s1;`
	`*s1 = *s2;`
	`*s2 = temp;`
		C: this swaps the 0th char of the strings, so it works if and only if their only difference is that character.

(e)	`while ((*s1++ = *s2++) != '\0') {`
	`    char temp = *s1;`
	`    *s1 = *s2;`
	`    *s2 = temp;`
	`}`
		B: this causes both strings to end up with identical contents. Because they initially have different contents, this means they cannot have been swapped.