

Problem 1 *Free's Company*

(15 points)

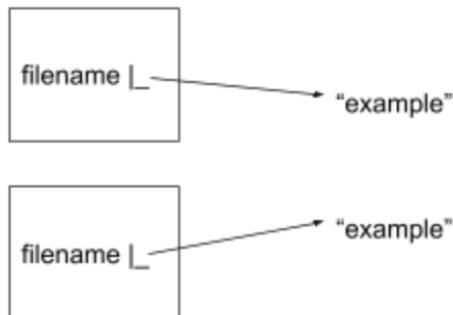
In project 1-2 we asked you to allocate and free memory for an AST struct. Consider the following simplified struct.

```
typedef struct simple_ast {
    char *filename;
    int type;
    struct simple_ast **children;
    int size;
} simple_AST;
```

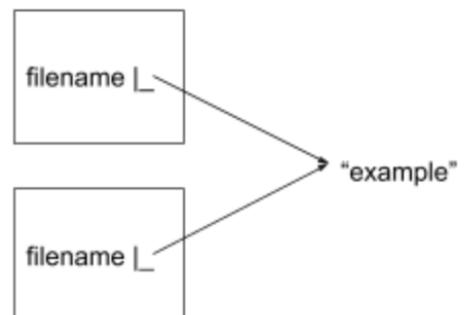
When Nick was testing the project, he found that he ran out of memory sooner than expected on large inputs. He attributed this to redundant malloc's for filenames.

To address this, Nick decides to make one malloc for filenames that are shared between nodes. The left image below shows what was assumed in the project, and the right image shows the new, single-malloc scheme.

OLD



NEW



(a) Nick decides to use the following function to free his ASTs.

```
void FreeAST (simple_AST *tree) {
    if (tree != NULL) {
        int i;
        for (i = 0; i < tree->size; i++) {
            FreeAST (tree->children [ i ]);
        }
        free (tree->children);
        free (tree->filename);
        free (tree);
    }
}
```

In 1 sentence explain why Nicks free function will cause problems on the following input. You may assume all calls to malloc succeed.

```
char *filename = malloc (sizeof (char) * (strlen("ex") + 1));
strcpy (filename, "ex");
simple_AST *tree = MakeAST (...., filename);
simple_AST *child = MakeAST (...., filename);
AppendAST (tree, child);
FreeAST (tree);
```

- (b) To fix this problem, Nick decides to take the following approach: he will keep any address he intends to free in a structure of unique elements. Then, once he has iterated through the whole tree, he will iterate through the structure and free each address. Nick creates the following structure for holding the addresses:

```
typedef struct shared_string {
    int size;
    int capacity;
    char **arr;
} shared_string_t;
```

Fill in the following function to add addresses to the lst. The function should *not* copy the strings themselves. Assume `contains` returns nonzero if the address is already in the list and zero otherwise.

```
void append_address (shared_string_t *lst, char *address) {
    if (! contains (lst, address)) {
        if (lst->size == lst->capacity) {
            _____;
            _____;
        }
        _____;
        _____;
    }
}
```

- (c) Finally we will implement `FreeAST`. This function should free all memory associated with the given `simple_AST` and should *not* exhibit the problem you described in part (a). You may assume the existence of the following helper functions:

```
/* This function allocates memory for a shared_string_t and initialises
it. It returns the pointer to the malloc'd memory. You may assume all
calls to malloc succeed. */
```

```
shared_string_t *create_list ();
```

```
/* This function frees the given simple_AST node and its children. It
also adds each filename to the given shared_string_t struct so it can be
deleted later. */
```

```
void FreeASTHelper (simple_AST *tree, shared_string_t *addr_list);
```

Fill in the remaining sections of `FreeAST`. Note the function should not leak any memory.

```
void FreeAST (simple_AST *tree) {
    if (tree != NULL) {
        shared_string_t *lst = create_list ();
        FreeASTHelper (tree, lst);
        int i;
        for (i = 0; i < lst->size; i++) {
            _____;
        }
        _____;
        _____;
    }
}
```

Problem 2 Remember-y Management**(18 points)**

For this problem, assume all pointers are four bytes and all characters are one byte. Consider the following C code (all the necessary #includes are omitted). C structs are properly aligned in memory and all calls to malloc succeed.

```
int size = 0;
struct map_entry {
    char *key;
    char *value;
};

void add_entry(struct map_entry *m, char *k, char *v) {
    int *zero = NULL;
    m[size].key = k;
    m[size].value = v;
    size++;
}

void main(void) {
    struct map_entry *map = malloc(sizeof(struct map_entry) * 10);
    char *key = malloc(sizeof(char) * 10);
    char value[20];
    add_entry(map, "k", "v");
    add_entry(map, key, value);
}
```

(a) For each of the following, bubble the option that best describes where in the memory layout each variable is stored **You should select one answer per variable.**

(a) zero

Stack

Static

Heap

Code

(b) *map[0].key

Stack

Static

Heap

Code

(c) map[1].value

Stack

Static

Heap

Code

(d) `add_entry`

Stack

Static

Heap

Code

(e) Bubble the comparators that would make the following expressions evaluate to true. If there is not enough information in the problem to answer conclusively, select the last option. Assume malloced memory grows upward, allocating the first available address.

(a) `map -- zero`

>

==

<

Not enough information

(b) `key -- &size`

>

==

<

Not enough information

(c) `map -- key`

>

==

<

Not enough information

(d) `value -- &zero`

>

==

<

Not enough information

(e) How many bytes of memory are leaked by this program?

Solution:

Problem 3 *AST: Another Stupid Tree***(20 points)**

In this problem, we will revisit a simplified version of the AST from Project 1-2. Our version is shown below. We are interested in implementing a function that searches our AST for a given piece of data and, when found, replaces the data with a value returned by `(*f)(data)`. We will search for and replace every occurrence of the data inside our tree.

```
Struct AST_Simple {
    Struct AST_Simple **children;
    int size;
    int data;
}

void SearchAST (struct AST_Simple *ast, int data, int (*f)(int)) {
    int i = 0;
    /* If the AST is NULL, no match */
    if (ast == NULL) {
        return;
    }

    /* If the head node contains our data, we found a match */
    if (ast->data == data){
        ast->data = (*f)(data);
    }

    /* Search for the data within the children nodes */
    for (; i < ast->size; i++){
        searchAST(ast->children[i], data, f);
    }
}
```

```
# Arguments follow the RISC-V calling convention
SearchAST:
```

```
    # Prologue
    addi sp sp _____
    sw  _____
    sw  _____
    sw  _____
    sw  _____
    sw  _____
    # Preserve and set arguments
    add s3 x0 x0
    mv s0 a0
    mv s1 a1
    mv s2 a2
    # Start computing
IfOne:
    bne _____
    j  _____
IfTwo: # Have we found what we're looking for?
    lw t0 _____
    bne _____
    _____
    _____
    _____
Loop:   # Check our children
    lw t0 _____
    bge _____
    lw t0 _____
    slli t1 _____
    add t2 _____
    # Prepare for recursive call
    lw a0 _____
    _____
    _____
    jal _____
    _____
    j  _____
Done:   # Epilogue
    lw  _____
    lw  _____
    lw  _____
    lw  _____
    lw  _____
    addi sp sp _____
    jr ra
```

Problem 4 *Do you see what IEC?*

(15 points)

(a) IEC Prefixes

1. The human genome is approximately 3,200 Mega-base pairs. How many base pairs is 3,200 Mega-base pairs? For credit you must format your answer using powers of 10 (ie $_ * 10 \^{_}$).

2. Ram wants to sell you a floppy disk with 8 (marketing) KIBIbytes memory for 16 dollars. Sruthi also wants to sell you a floppy disk, with 8 (marketing) KILObytes memory for 10 dollars.

- (a) For Rams floppy disk, how many bytes are on the floppy disk? Do not simplify.

_____ Bytes

(b) For Rams floppy disk, how many bytes per dollar?

_____ Bytes per dollar

(c) For Sruthis floppy disk, how many bytes are on the floppy disk? Do not simplify.

_____ Bytes

(d) For Sruthis floppy disk, how many bytes per dollar?

_____ Bytes per dollar

(b) Number Representation

1. Because Moores Law is dead, its time to try storing data in something alive. A DNA strand is composed of a linear sequence of chemical base pairs, and we can consider each of its 4 base pairs (A, C, T, G) as a "digit" How many unique values can be stored in an X-base pair DNA strand? Your answer should contain the variable X.

_____ unique values

2. Assume we use DNA to store data in base-4, unsigned. A = 0, C = 1, T = 2, and G = 3, with the most significant base pair first. *Use the number-equivalent of each base to format your answers below.*

- (a) How do we store the decimal (base-10) number 36 in DNA?

- (b) How do we store the binary (base-2) number *0b01111100* in DNA?

- (c) How do we store the hexadecimal (base-16) number *0x7AB* in DNA?

- (d) Convert GATTACA to binary

*0b*_____

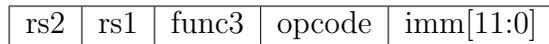
Problem 5 *More Poorly Written Instructions (Project 1-3)* (15 points)

As much as we love working in RISC-V, TAs and students everywhere are tired of having to transcribe 32-bits for every instruction. We'd like to invent a new machine language based on RISC-V that uses fewer bits overall.

To make this change possible, our reduced instruction set will contain *only* the following instructions:

1. add
2. addi
3. beq
4. jal

Additionally, we will group together all immediate bits and place them at the end of the instruction. If we rearrange the *standard RISC-V SB-type* to match our immediate adjustment, it would look like the following:



The instructions beq and jal will encode the entire immediate necessary for control flow and do not append a trailing zero. We would like to continue to support the use of all 32 registers.

1. Lets say we decide to remove the funct3 and funct7 fields so that our R-type format looks like the following:



How many bits do we need to represent each of the following fields?

Opcode: _____ bits

rd: _____ bits

rs1: _____ bits

rs2: _____ bits

2. Using the values in your previous answer (and assuming all instructions must be the same total size), how many bits would we have for the immediate field in an I type instruction?

Imm: _____ bits

3. Because we only have 4 instructions, we can't represent all of our instruction types. Which types are missing from our language? Mark all that apply.

- | | |
|-------------------------|--------------------------|
| <input type="radio"/> R | <input type="radio"/> SB |
| <input type="radio"/> I | <input type="radio"/> U |
| <input type="radio"/> S | <input type="radio"/> UJ |

4. For the following question, mark the statement as true or false and give a tweet-length justification. (if true, mention which two formats). Assume we consider all register fields (rs1, rs2, rd) the same.

Two of our instruction formats have the exact same field ordering

- True False

5. Now assume our instructions match the format above but are 8 bits in length. If we have 2 bits of the opcode and 2 bits for registers (which may or may not reflect answers in previous parts), convert the following instructions.

Opcodes	
0b00	add
0b01	addi
0b10	beq
0b11	jal
Registers	
0b00	zero
0b01	ra
0b10	s0
0b11	t0
Labels	
One	One byte forward
Two	Two bytes forward
BOne	One byte backward
BTwo	Two bytes backward

(a) `addi zero, ra, -1`

(b) `0b10101110`

Problem 6 *CALL: Crammed At the Last Lecture*

(7 points)

(a) Mark the following as either True or False

1. The linker reads in one or more object files and generates an executable or library.

True False

2. The assembler may remove pseudo instructions before passing the file to the linker, though this step is optional because pseudo instructions are understood by machines.

True False

3. The 61Ccc compiler you wrote in projects 1-1 and 1-2 is considered a complete compiler in that it converts c-like files to assembly code.

True False

4. Executable files generated by CALL are machine-dependent.

True False

(b) Consider an assembler that analyses code top-to-bottom in a single pass. Which of the following isolated blocks of code exhibit the forward reference problem? Bubble all that apply.

LOOP:
 beq t0 t1 END
 addi t0 t0 1
 sw t0 0(s0)
 j LOOP
END:

CALL:
 jal ra F
 la a0 CALL
F:
 addi sp sp -4
 sw s0 0(sp)

 add t0 t0 x0
 addi t1 x0 3
REDO:
 addi t0 t0 1
 slli s1 t0 2
LOOP:
 bne t0 t1 REDO
 add s0 s1 x1
 lw t0 0(s0)
 j LOOP

F_X:
 mv t0 a0
 addi t0 t0 4
 lw a1 0(t0)
 jr ra
G_X:
 mv a0 s0
 jal ra F_X



Figure 1: This Space Deliberately Left Blank