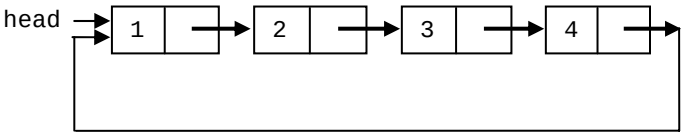
Fall 2004 MT (Paul)

Question 1: C and Circular Lists (22 points – 30 min.)

We're writing a circular linked list to keep numbers. The idea is very similar to a single-linked list, but the last element points to the first. Our circular linked list is made up of elements of type `pair` (a data type from CS61A and project 1). Assume when the list is empty we initialize the *global variable* `head` to `NULL`. Here's an example on the left, with the `pair` definition on the right:



The `pair` structure is defined as follows:

```
struct pair {
    int car;          // "number"
    struct pair *cdr; // next "pair"
} *head;
```

In the figure above, we can see 4 elements linked. When we insert an element, it goes *after the first element.* E.g., if we represent the distinct elements list in the example above *before* insertion as `{1 2 3 4}`, then *after* a call to `insert(5)` it would be `{1 5 2 3 4}`.

a) Help us to write the `insert` function by adding only 3 statements.

```
void insert(int d) {
      /* create the new node */
      [                                                    ]
      tmp->car = d;

      /* Insert the new node in the right place */
      if ( head == NULL ) {
            /* The struct was empty… link the itself and we're done */
            tmp->cdr  = tmp;
            head      = tmp;
      } else {
            /* There were already elements in the linked list.
                Link the new node after the first element. */
            [                                              ]
            [                                              ]
      }
}
```

b) Instead we'd like you to link it after the "second" element. (If we only have 1 element, do the same as before. Can it be done by only modifying the 2nd and 3rd statements? If so, do it below. If not, explain why.

c)

```
/* Link the new node after the "second" element */
[                                                    ]
[                                                    ]
```

Or

```
[                                                    ]
```

Question 1 (continued): C and Circular Lists (22 points – 30 min.)

d) Now, we want to be able to delete the full structure. Assume that the OS immediately fills any freed space with garbage, so you cannot access freed heap contents. Finish the recursive `delete_recursively` function. We want the tightest, cleanest code possible (measured by the number of statements which terminate in semicolons). If you use only *2 semicolons*, full credit. If you use 3, you'll lose 1 point. If you use more, you'll lose 2 points.

```
void delete_full_structure()
{
      if (head == NULL )
            return;

      delete_recursively(head);
}
```

You saw how inserting a fifth element numbered 5 into our list messed up our numbering. We'd like to write `reset_numbers` that *clobbers* the node numbers to restore the nice `1, 2, 3,…` numbering. Note: A pointer to a `struct` stored in memory is just a pointer to memory we treat as broken up into the fields.

```
      void reset_numbers(pair* p, int i)
      {
            if (p != NULL ) {
                  p->car = i;
                  reset_numbers(->cdr, i++);
            }
      }
```

e) Convert `reset_numbers` to MIPS **keeping its structure recursive**. I.e., don't hand-optimize.

prologue
body
epilogue

f) In one sentence, what happens *on an actual MIPS machine* if we call `reset_numbers(head, 1)` on the lists as described in this problem? (Assume our list is not empty).

Name: _____ Login: cs61c-____

Question 4: `fun` with MIPS ... more naughty bits! (15 pts, 36 min)

What follows is an inefficient MIPS function. Read it carefully, and answer the questions below. The definition of `div` can be found in your green sheet, column ↑ (`div a,b` ∏ `lo=a/b, hi=a%b`).

```
fun:  mov   $v0, $0
      li    $s0, 1
loop: beq   $a1, $0, end
      addiu $a1, $a1, -1
      sll   $s0, $s0, 1
      div   $a0, $s0
      mfhi  $s1
      or    $v0, $v0, $s1
      j     loop
end:  jr    $ra
```

```
// Precondition: y < 31

unsigned int fun(unsigned int x,
                 unsigned int y)
{




}
```

a) Briefly, explain what `fun` returns (assuming `y < 31`).
***Don't** describe the algorithm; explain how the return value relates to `x` and `y`.*

b) Write optimized C code for `fun` in the box (make it as compact and efficient as possible). That is, think of all the C tricks you know and try to author it in the fewest characters possible.

c) Uh oh, we've broken some calling conventions! What should we add to the beginning (before `mov $v0, $0`) and end (before `jr $ra`) of `fun` to correct this? Help!

BEGIN	END

Fall 2005 MT (Sung Roa)

Question 2: Bit fields and bit fields of wheat…(7 pts, 15 min.)

You believe the encoding of MIPS opcodes, functions and bit field widths should be modified. What we currently use is on the left and what you propose is on the right – note you have not changed the code for the R and I cases, just reordered them. We've left out the code (replacing it with "??") for determining *which* R and *which* I instruction it is, assume that can be worked out later. How many R-type, I-type and J-type instructions did we have and will we have and what is one big pro and two big cons of this proposal? Consider how this subtle change could change the bit fields for the instructions for better or worse. Yes, we already know we'd probably have to reprint things like the green sheet. When counting instructions, only count the number of different operators. E.g., you should count `jr $v0` and `jr $a1` as the same instruction (same `jr` operator). We've filled one in for you already.

Current	Proposed (changes in **bold**)

`if ((bit[31]…bit[26] == 0) {`
 	
`    // Look elsewhere for which R it is…`
`    inst_type = Rl inst = ??;`
 	
`} else if ((bit[31]…bit[26] == 2) {`
 	
`    inst_type = J; inst = jump;`
 	
`} else if ((bit[31]…bit[26] == 3) {`
 	
`    inst_type = Jl inst = jal;`
 	
`} else {`
 	
`    // Remaining ops are I`
`    // Ignore Floating Pt formats FR,FI`
`    inst_type = I; inst = ??;`
`}`
	`if (bit[31] == 1) {// Most-signif. Bit`
	
	`    inst_type = J`
	`    if (bit[30] == 1)`
	`        inst = jump;`
	`    else`
	`        inst = jal;`
	
	`} else if ((bit[31]…bit[26] == 0) {`
	
	`    // Look elsewhere for which R it is…`
	`    inst_type = Rl inst = ??;`
	
	`} else {`
	
	`    // Remaining ops are I`
	`    // Ignore Floating Pt formats FR,FI`
	`    inst_type = I; inst = ??;`
	`}`

# of R-type	# of I-type	# of J-type	# of R-type	# of I-type	# of J-type
		2			

The pro is…	1.
The cons are…	1. 2.

Name: _____ Login: cs61c-____

Question 4: Float, float on… (10 pts, 25 min)

The staff of CS61C is constantly investigating new approaches and solutions to old problems. (Either that or we just don't know when to leave well enough alone!) Well, we've come up with a new floating point format that **obeys the rules of the IEEE 754 Floating Point Standard** (denorms, NaNs, ±∞) but is *only 13 bits long*. It has 1 sign bit, 3 exponent bits, and 9 bits in the fraction (significand):

S	E	E	E	F	F	F	F	F	F	F	F	F

Answer the following questions about this new float format. As a sanity-check, if you calculate the bias of this format, you should get 3.

a) What is the largest non-infinite positive number that can be represented? Leave your answer as a base 10 mixed fraction (i.e. K ± (N/D)) where either K or N can be 0.

b) What is the smallest positive number that can be stored in this format?

Convert the following floating point values in the format above (expressed in hexadecimal) to their numerical (base 10) equivalents, if appropriate.

c) `0x1000`

d) `0x1F00`

What is the representation for the following floating point numbers? (Write your answer in hex).

e) `-∞`

f) `-4.5`