Assignment 9

Exercise 10.1

To understand how the abstract machine and the garbage collector work and how they collaborate, answer these questions:

(i)

Write 3-10 line descriptions of how the abstract machine executes each of the following instructions:

Instruction	Description
ADD	Retrieve the integer in the stack which is at sp-1 and untag it such that it is the original integer. The same is done for the integer at sp. The two integers are then added and tagged again. The stack pointer is decremented.
CSTI i	Takes the next instruction and tags it. Increments the stack pointer since you pushed the value onto the stack.
NIL	Pushes 0 as an address onto the stack. This integer value is not tagged so it is an address.
IFZERO	First take the value at sp-1 then it will check if this is 0 or nill. This is done by either untagging the integer or just checking if its nil. If it is zero, it will set the program counter and goto the address given. Otherwise the program counter is incremented
CONS	First allocate memory for the cons cell. Then it takes the two top words on the stack and put them into a cons cell.
CAR	Takes the first element from the cons cell at sp
SETCAR	Set first element of the cons cell

(ii)

Describe the result of applying each C macro Length, Color and Paint from Sect. 10.7.4 to a block header:

ttttttttnnnnnnnnnnnnnnnnnnnnngg,

that is, a 32-bit word, as described in the source code comments.

tttttttt is the block tagging nnnnnnnnnnnnnnnnnnnnn is the block length gg is the garbage collection color

(iii)

When does the abstract machine, or more precisely, its instruction interpretation loop, call the `allocate(...)` function? Is there any other interaction between the abstract machine (also called the mutator) and the garbage collector?

`allocate(...)` is only used in CSTI i which pushes the integer constant i The other garbage collection methods aren't used in the instruction interpretation loop.

(iv)

In what situation will the garbage collector's `collect(...)` function be called? When allocating memory, the `collect()` method will be called if there is no available memory.

Exercise 10.2

Add a simple mark-sweep garbage collector to listmachine.c, like this:

```
void collect(int s[], int sp) {
  markPhase(s, sp);
  sweepPhase();
}
```

Your markPhase function should scan the abstract machine stack `s[0..sp]` and call an auxiliary function `mark(word* block)` on each non-nil heap reference in the stack, to mark live blocks in the heap. Function `mark(word* block)` should recursively mark everything reachable from the block. The sweepPhase function should scan the entire heap, put white blocks on the freelist, and paint black blocks white. It should ignore blue blocks; they are either already on the freelist or they are orphan blocks which are neither used for data nor on the freelist, because they consist only of a block header, so there is no way to link them into the freelist. This may sound complicated, but the complete solution takes less than 30 lines of C code. Running `listmachine ex30.out 1000` should now work, also for arguments that are much larger than 1000. Remember that the listmachine has a tracing mode `listmachine -trace ex30.out 4` so you can see the stack state just before your garbage collector crashes. Also, calling the `heapStatistics()` function in listmachine.c performs some checking of the heap's consistency and reports some statistics on the number of used and free blocks and so on. It may be informative to call it before and after garbage collection, and between the mark and sweep phases. When your garbage collector works, use it to run the list-C programs ex35.lc and ex36.lc and check that they produce the expected output (described in their source files). These programs build shared and cyclic data structures in the heap, and this may reveal flaws in your garbage collector

markphase

```
//Go through each non-nil heap reference in the stack and mark it
//Your markPhase function should scan the abstract machine stack
s[0..sp] and
//call an auxiliary function mark(word* block) on each non-nil heap
reference in the stack,
//to mark live blocks in the heap.
void markPhase(word s[], word sp) {
     printf("mark phase starting...\n");

     //The mark phase goes through the entire heap to find all
references.
     //When we encounter a heap reference to a white block, it is
painted blackand
     //we recursively mark the block's words. After the mark phase,
every block in
     //the heap is either black because it is reachable, white
because it isn't
     //reachable or blue because it is in the free list.
     int i;

     for (i = 0; i <= sp; i++) {
          if (!IsNull(s[i]) && IsReference(s[i])) {
            word* p = (word*)s[i];
            p[0] = Paint(p[0], Grey);
          }
     }

     for (i = 0; i <= sp; i++) {
          if (!IsNull(s[i]) && IsReference(s[i])) {
            word* p = (word*)s[i];
            mark(p);
          }
     }
}
```

mark

```
void mark(word* block) {
    printf("marking ...\n");

    //If the block color is white, paint it black.
    if (Color(block[0]) == Grey)
    {
        block[0] = Paint(block[0], Black);

        //Check if car and cdr is a pointer
        //Call mark if car and cdr is pointer
        if (BlockTag(block[0]) == CONSTAG)
        {

            if (!IsNull(block[1]) && IsReference(block[1]))
            {
                mark((word*) block[1]);
            }

            if (!IsNull(block[2]) && IsReference(block[2]))
            {
                mark((word*) block[2]);
            }
        }
    }
}
```

sweepPhase

```
void sweepPhase() {
     printf("sweep phase starting ...\n");
```

```
	//Sweep phase visits all blocks in the heap.
	//After the sweep phase, the freelist contains all (and only) blocks that are
	//not reachable from the stack. Moreover, all blocks in the heap are white,
	//except those on the freelist, which are blue.

	word* heaper = heap;

	while (heaper < afterHeap) {
		word header = *heaper;
		int color = Color(header);

		switch (color)
		{
			case Black:

				//If a block is black: reset color to white.
				heaper[0] = Paint(header, White);
				break;

			case White:

				//If a block is white: it is painted blue and added to the freelist
				heaper[0] = Paint(header, Blue);
				heaper[1] = (word)freelist;
				freelist = heaper;

				break;

			case Blue:

				//If a block is blue: skip
				break;
		}
		heaper += 1 + Length(header);
	}
}
```

Exercise 10.3

Improve the sweep phase so that it joins adjacent dead blocks into a single dead block. More precisely, when sweep finds a white (dead) block of length n at address p, it checks whether there is also a white block at address p + 1 + n, and if so joins them into one block. Don’t forget to run the list-C programs ex35.lc and ex36.lc as in Exercise 10.2.