Section Handout

Problem 1: Meet The Flintstones

Consider the following C-style **`struct`** definitions:

```
typedef struct rubble { // need tag name for self-reference
   int betty;
   char barney[4];
   struct rubble *bammbamm;
} rubble;

typedef struct {
   short *wilma[2];
   short fred[2];
   rubble dino;
} flintstone;
```

Accurately diagram what computer memory looks like after the following seven lines of code have executed:

```
rubble *simpsons;
flintstone jetsons[4];

simpsons = &jetsons[0].dino;
jetsons[1].wilma[3] = (short *) &simpsons;
strcpy(simpsons[2].barney, "Bugs Bunny");
((flintstone *)(jetsons->fred))->dino.bammbamm = simpsons;
*(char **)jetson[4].fred = simpsons->barney + 4;
```

Problem 2: Scheme

Scheme is a language whose primary built-in data structure is the linked list. Unlike any of the lists you've dealt with in C, Scheme lists are fully heterogeneous—that is, the entries needn't all be the same type.

Some example lists are:

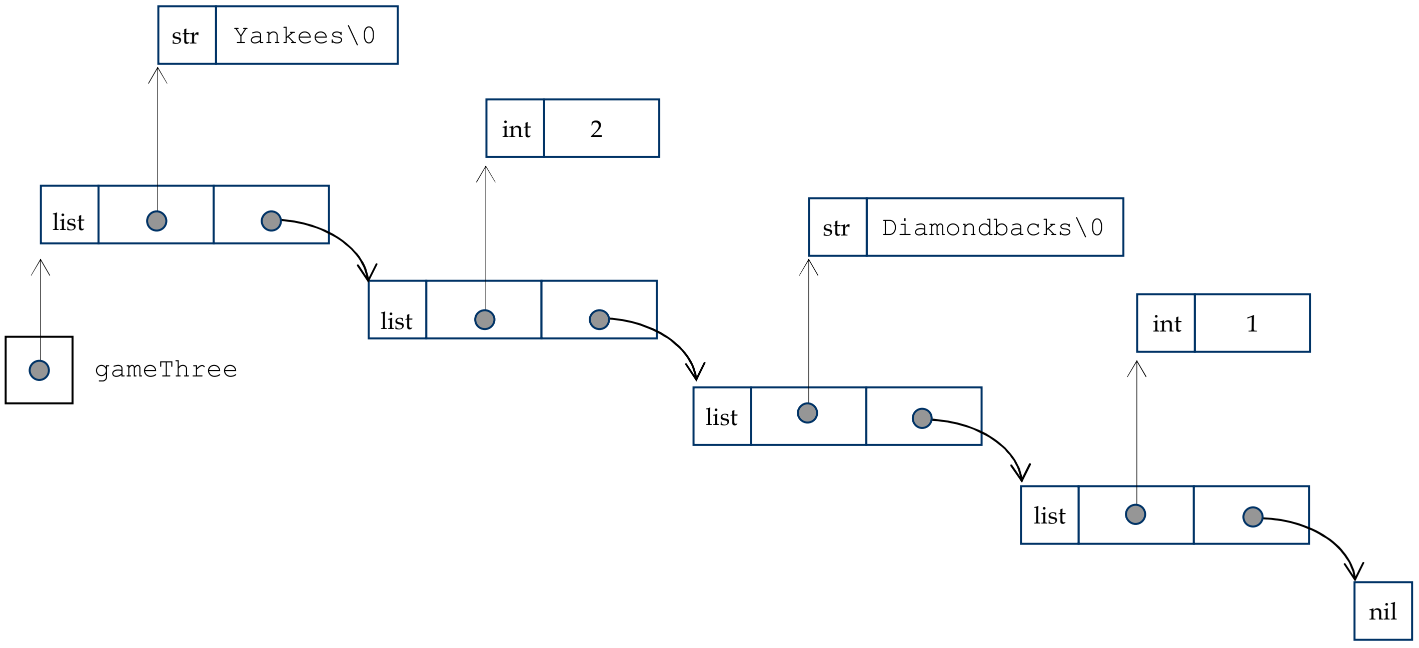
```
i.     (2 3 5 7)
ii.    (House at Pooh Corner)
iii.   (Yankees 2 Diamondbacks 1)
iv.    (4 calling birds 3 French hens 2 turtle doves 1 partridge)
```

These linked lists are so flexible, individual elements might themselves be lists. If that's the case, then lists can be nested to any depth.

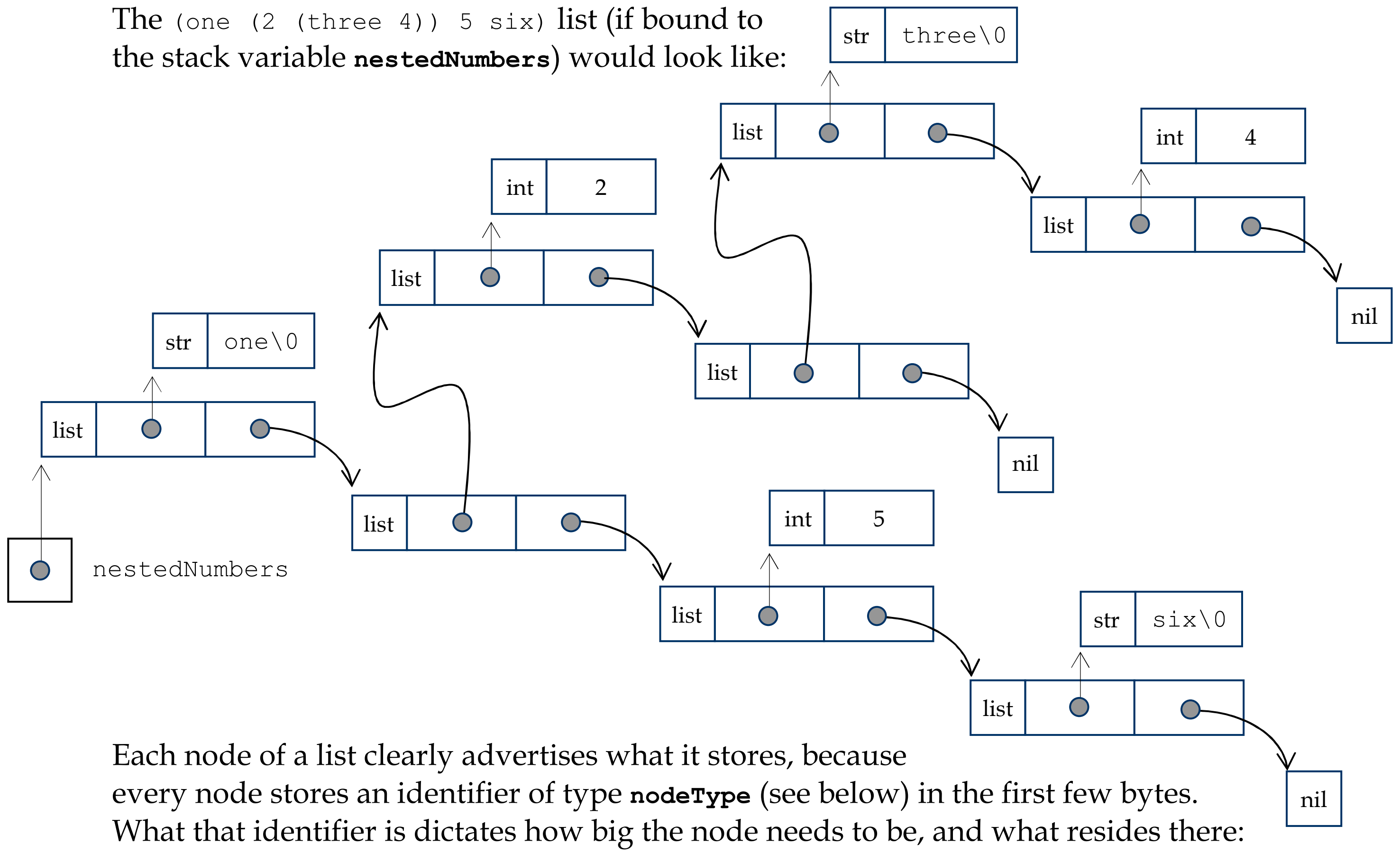
```
v.     ((1 2) (buckle my shoe))
vi.    (one (2 (three 4)) 5 six)
vii.   (how (nested (can (u (go)))) how (nested (can (u (go)))))
```

We can provide heterogeneous lists in C, but they don't come easy. In order for them to work, the individual elements of the list must carry their own type information. The idea is to tag each list node with some enumerated type that tells us what the rest of the node contains.

We'll just pretend that integers and strings are the only atomic types of interest. The third list above (if bound to the stack variable **`gameThree`**) would be structured as follows:



The `(one (2 (three 4)) 5 six)` list (if bound to the stack variable **`nestedNumbers`**) would look like:

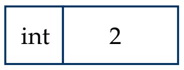


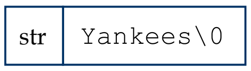
Each node of a list clearly advertises what it stores, because every node stores an identifier of type **`nodeType`** (see below) in the first few bytes. What that identifier is dictates how big the node needs to be, and what resides there:

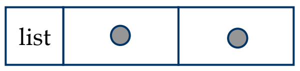
We'll officially allow only strings and integers; therefore, the following enumerated type suits our needs:

```
typedef enum {
   Integer, String, List, Nil
} nodeType;
```

When handed a list, we need to pull the **`nodeType`** value from the first **`sizeof(nodeType)`** bytes. From that, we know whether the rest of the node…

stores an **`int`**, as with: 

stores a null-terminated character array, as with: 

stores two addresses, as with:  , or

stores nothing—the end of some list has been reached.

Note the characters of a **`String`** node are inside the node.

The **ConcatAll** function takes a well-formed list and returns the ordered concatenation of all of the list's strings (including those in nested sublists.) Integers should just be skipped, and shouldn't contribute to the return value at all. Your implementation shouldn't orphan any memory.

```
/**
 * Traverses a properly structured list, and returns the ordered
 * concatenation of all strings, including those in nested sublists.
 *
 * When applied to the two lists drawn above, the following strings
 * would be returned:
 *
 *    ConcatAll(gameThree) would return "YankeesDiamondbacks"
 *    ConcatAll(nestedNumbers) would return "onethreesix"
 */

typedef enum {
   Integer, String, List, Nil
} nodeType;

char *ConcatAll(nodeType *list)
{
```