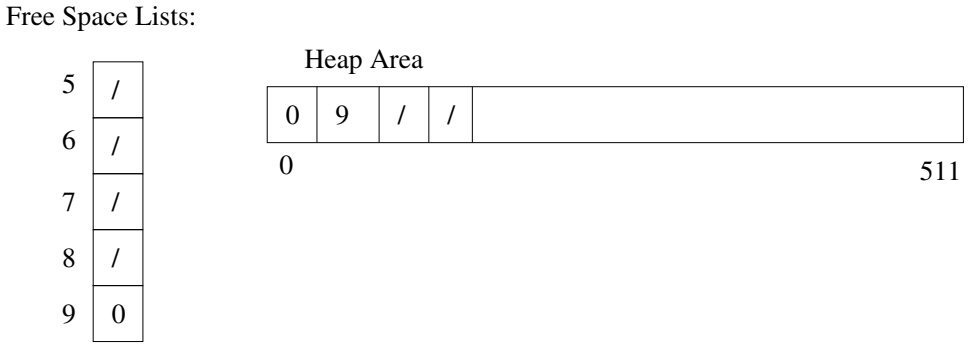
**CSC231 Programming Languages Spring 2017**
**Final Exam Practice Problems**

1. Buddy system memory management: Consider the following initial configuration of a block of 512 bytes of memory managed using the buddy-system. Assume the minimum block size is 32 bytes.




Draw a sequence of diagrams showing how the free space lists and the heap area change with the following requests:

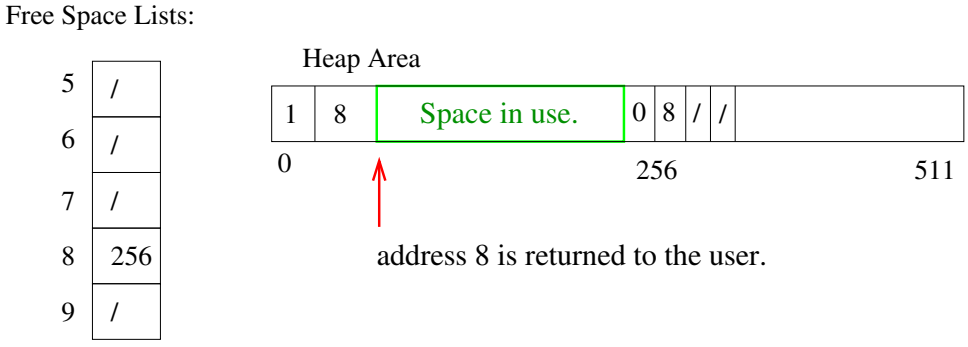
(a) A request for 200 bytes.

**Answer:** We start by finding the smallest $k$ such that $2^k \geq (r + v)$ where $r$ is the request size and $v$ is the overhead for a used block. We can reasonably suppose that the overhead for a used is 8 bytes. For this request, the total space needed is 208 bytes. We can satisfy that with a block size $k = 8$, i.e. $2^8 = 256 > 208$.

We then consult the collection of free space lists for a block size 8. There are none. We then search for the next larger size and find $k' = 9$.

We take the block size $k' = 9$ off of the available space list (of blocks size $2^9$ and split it into left and right "buddies" size $2^8$. One of the buddies is used to satisfy the memory request, and the other is placed of the free space lists in position 8.

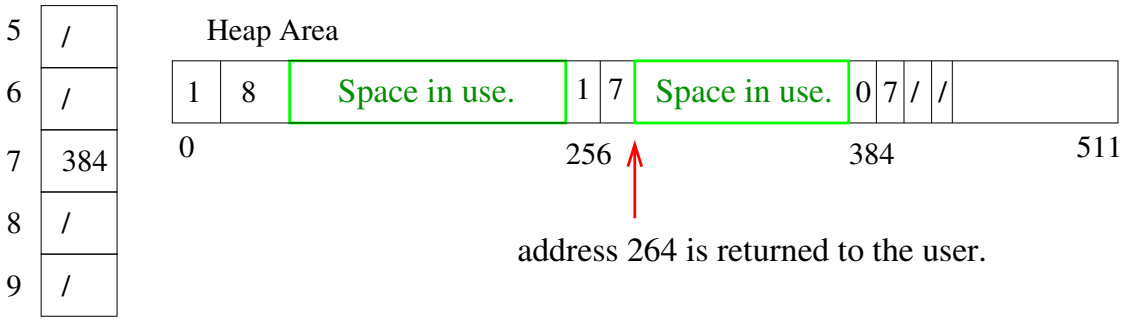
The resulting memory configuration is shown in the diagram below:




(b) A request for 100 bytes.

**Answer:** We seek the smallest $k$ such that $2^k \geq 108$ and find $k = 7$. There are no free blocks size $2^7$, so we look for a larger free block and find a block size $2^8$. We detach that block from the free space list, and split it. We put the right buddy on the list of free blocks "size 7" and use the left buddy to satisfy the request. The resulting memory configuration is shown in the diagram below:

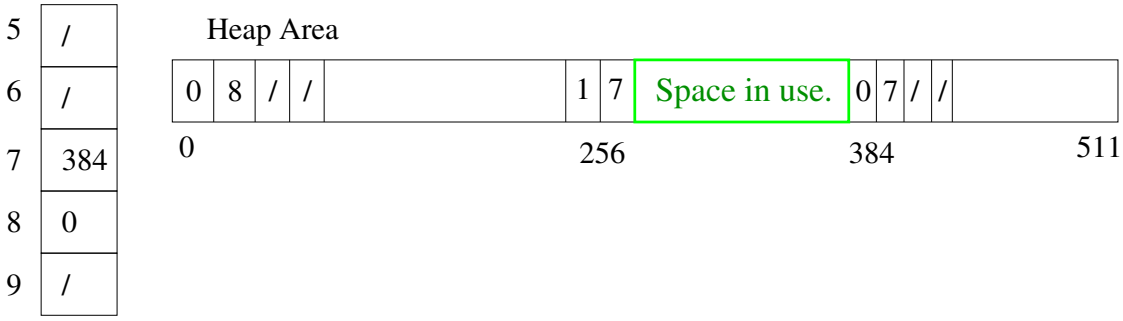
Free Space Lists:




(c) The previously allocated 200 bytes are released.

**Answer:** The system receives address 8 which was previous allocated. We subtract the overhead (also 8) to get address 0. From address zero, we use the known offsets to find the size of the block. The size is $2^8$. We divide the address by the block size. If that is an even number, we know this block was originally a left buddy. We have $0/(2^8) = 0$, an even number. Therefore, we compute the address of the right buddy: $0 + 2^8 = 256$. We look at the block starting at address 256. If it is free and the same size, we can merge the blocks. In this case, we find the block at address 256 is neither free nor the same size as the block starting at 0. We can not merge the blocks. We return the block starting at address 0 to the free space lists of blocks "size 8". The resulting memory configuration is shown in the diagram below:

Free Space Lists:




2. The following is a simplified grammar for C++ declarations.

$$\begin{aligned} \text{Decl} &\rightarrow \text{Type List ;} \\ \text{Type} &\rightarrow \text{int} \mid \text{char} \\ \text{List} &\rightarrow \text{id MoreList} \\ \text{MoreList} &\rightarrow \text{, id MoreList} \mid \epsilon \end{aligned}$$

For example, `int x, y, z ;`

Design a syntax directed definition to record the given type information in the symbol table record for each variable. Use pseudocode. Assume a symbol table object `HT` and assume any needed methods such as `insert()`, `search()`, etc. The following steps may help

(a) Decide on the attributes you wish to associate with each variable in the grammar.

(b) Decide which attributes are synthesized attributes and which are inherited attributes.

(c) Assume the terminal `id` provides the name of the identifier in the attribute `id.name`.

(d) Write semantic actions to move the information through the parse tree as needed. *Hint:* draw a parse tree for the sentence `int x, y, z ;`

**Answer:** reasonable set of attributes is given below:

**Type.whattype** Attribute **whattype** is records the data type for this declaration. **whattype** is a synthesized attribute.

**List.whatlist** Attribute **whatlist** transfers the information contained in `Type.whattype` across and down the parse tree. **whatlist** is an inherited attribute.

**Morelist.whatmore** Attribute **whatmore** is a similar to `whatlist` ; it propagates the type information across and down the parse tree. **whatmore** is an inherited attribute.

**id.name** The name of an identifier.

A syntax directed definition is shown below:

| **Production** | **Semantic Action** |
|---|---|
| Decl → Type List | List.whatlist = Type.whattype ; |
| Type → int | Type.whattype = "int" ; |
| Type → char | Type.whattype = "char" ; |
| List → id MoreList | HT.set_type( id.name, List.whatlist ) ;<br>MoreList.whatmore = List.whatlist ; |
| $\text{MoreList}_1$ → , id $\text{MoreList}_2$ | HT.set_type( id.name, $\text{MoreList}_1$.whatmore ) ;<br>$\text{MoreList}_2$.whatmore = $\text{MoreList}_1$.whatmore ; |
| MoreList → $\epsilon$ | /* Nothing to do here. */ |

3. The following is the first sets, the follow sets, and a predictive parsing table for the grammar in problem 2. Write the pseudocode for a recursive descent parser.

```
 First sets:
      FIRST(Decl) = { int char  }
      FIRST(Type) = { int char  }
      FIRST(List) = { id  }
      FIRST(MoreList) = { epsilon ,  }

Follow sets:
     FOLLOW(Decl) = { eof  }
     FOLLOW(Type) = { id  }
     FOLLOW(List) = { ;  }
     FOLLOW(MoreList) = { ;  }


Parse Table:

   Variable Decl :
      Input 'int':  Decl ->  Type List ;
      Input 'char':  Decl ->  Type List ;

   Variable Type :
      Input 'int':  Type ->  int
      Input 'char':  Type ->  char

   Variable List :
      Input 'id':  List ->  id MoreList

   Variable MoreList :
      Input ';':  MoreList ->  epsilon
      Input ',':  MoreList ->  , id MoreList
```

**Answer:**

```
// ------------------------------------------------
void Decl( token & current, hashtable & ht )
{
   attribute whattype ;

   if  ( current == 'int' ) or ( current == 'char' ) {
       Type( current, ht, whattype ) ;
       List( current, ht, whattype ) ;
   }
   else {
       syntax_error() ;
   }
}
```

```
// ------------------------------------------------
void Type( token & current, hashtable & ht, attribute & whattype )
{
   if  ( current == 'int' ) {
       match( current, 'int' ) ;
       whattype = 'int'        ;
   }
   else if  ( current == 'char' ) {
       match( current, 'char' ) ;
       whattype = 'char'        ;
   }
   else {
       syntax_error() ;
   }
}

// ------------------------------------------------
void List( token & current, hashtable & ht, attribute & whatlist )
{
   if ( current == id ) {
       ht.set_type( current.name, whatlist ) ;
       match( current, id ) ;
       Morelist( current, ht, whatlist ) ;
   }
   else {
       syntax_error() ;
   }
}

// ------------------------------------------------
void MoreList( token & current, hashtable & ht, attribute & whatmore )
{
   if ( current == comma ) {
       match( current, comma ) ;
       if ( current == id ) {
           ht.set_type( current.name, whatmore ) ;
           match( current, id ) ;
       }
       else {
           error( "Expected id." ) ;   boom() ;
       }
       MoreList( current, ht, whatmore ) ;
   }
   else if ( current == semicolon ) {
       /*  Do nothing:  MoreList --> epsilon   */
   }
   else {
       syntax_error() ;
   }
}
```

4. Define the following terms:

   (a) Grammar symbol

   **Answer:** A context free grammar is a 4-tuple $G = (V, T, S, R)$ where:

   - $V$ is a set of variable symbols
   - $T$ is a set of terminal symbols, $V \cap T = \phi$.
   - $S$ is a start symbol, $S \in V$.
   - $R$ is a set of rules of the form $A \rightarrow \alpha$.

   The symbols in the set $V \cup T$ are considered grammar symbols.

   (b) Production (as part of a grammar)

   **Answer:** A production is a rule in $R$.

   (c) Attribute

   **Answer:** An attribute is any type of information we wish to associate with a grammar symbol. Every instance of a grammar symbol in a parse tree often has different values for the attribute.

   (d) Synthesized attribute

   **Answer:** A synthesized attribute moves information up the parse tree, from child nodes to parent nodes.

   (e) Inherited attribute

   **Answer:** An inherited attribute moves information across the parse tree (from node to sibling), or down the parse tree (from parent to child).

   (f) Syntax directed definition

   **Answer:** A syntax directed definition is a collection of semantic actions. Each semantic action computes the value of an attribute in terms of other attributes.