CPSC 126 SAMPLE FINAL EXAMINATION
July, 2002

Name: ____________________
Student ID: ____________________
Lab Section: ____________________

- You have 180 minutes to write the 9 questions on this examination. A total of 130 marks are available.
- You are allowed **one** 8.5 X 11 two-sided sheet of handwritten notes. No other notes, books, calculators, computers, CD players, walkmans, boom boxes, robots, walkie-talkies, carrier pidgeons, nulcear weapons, or cellular phones are allowed.
- The number in square brackets to the left of the question number indicates the number of marks allocated for that question. Use these to help you determine how much time you should spend on each question.
- **Justify all your answers**
- Use the back of the pages for your rough work.
- **Good luck!**

Question	Marks
1	
2	
3	
4	
5	
6	
7	
8	
9	
Total	

UNIVERSITY REGULATIONS:

- No candidate shall be permitted to enter the examination room after the expiration of one half hour, or to leave during the first half hour of the examination.
- CAUTION: candidates guilty of any of the following, or similar, dishonest practices shall be immediately dismissed from the examination and shall be liable to disciplinary action.
  1. Making use of any books, papers or memoranda, electronic equipment, or other memory aid devices, other than those authorised by the examiners.
  2. Speaking or communicating with other candidates.
  3. Purposely exposing written papers to the view of other candidates. The plea of accident or forgetfulness shall not be received.
- Smoking is not permitted during examinations.

[10] 1. Short Answers

[2] a. *Briefly* describe the difference between a vector and an array:

```
A vector is an object which encapsulates an array and a size (at least).
An array is just a pointer
```

[2] b. *Briefly* describe the difference between a class and an object.

```
A class defines a type. An object is a particular instance of a class.
```

[6] c. Here is a function prototype:

```
void foo(double *x, string & y, list<Robot> a);
```

Describe *in words* the types of each input argument, and what kind of values will each argument contain.

```
double *x
A pointer to double.
x will contain the memory address of a double

string & y
A reference to string
y is an alias of a string variable

list<Robot> a
A list of Robot object
a is an instance of the STL list class, with Robot as the type of
variable in the list
```

[10] 2. Representing values inside the computer

[2] a. Convert the (decimal) number `58` to an 8-bit binary number. **HINT**: $58/2 = 29$ and $29/2 = 14 + 1/2$

```
00111010
```

[3] b. Calculate the 8-bit binary representation of `61` by adding 3 in binary (`00000011`) to your answer for (a). Show your work. Do not simply convert 61 to binary like you did for 58 in the part (a).

```
00111010
00000011
--------
00111101
```

[2] c. Give the 8-bit binary two's complement representation of `-61`. Show your work

```
62        : 00111101
flip bits: 11000010
add one   : 11000011
```

[3] d. Give the hexadecimal representation for 61, and for 58

```
61: 3d
58: 3a
```

[8] 3. The R2D2 Robot Company Inc. has a new line of coffee delivery robots. These robots understand a simple programming language as specified by this EBNF:

```
robot-program: motion-commands buy-command motion-commands "(deliver-coffee)"
motion-commands: motion-command { motion-command }
motion-command: "(" (move-command | turn-command) digit { digit } ")"
move-command: "move" ("forward" | "backward")
turn-command: "turn" ("right" | "left")
buy-command: "(buy-coffee" coffee-type size [ "cream" digit ] [ "sugar" digit ] ")"
coffee-type: "dark" | "light"
size: "S" | "L"
digit: "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

For each of the following programs, write either `valid` if its a valid program *according to the EBNF given above*, or `invalid` if it is not. If you write `invalid`, also give a reason. Don't worry about spaces or newlines. Remember that `[a]` means *optional*, { ... } means *zero or more repetitions*, `|` means *or* and `"..."` means *as-is*.

———————————————— (a) ————————————————

```
(move forward 10)
(turn left 30)
(buy-coffee dark L)
(turn right 20)
(move backward 300)
(deliver-coffee)
(jump-out-of-window)   <--- invalid because of this
```

———————————————— (b) ————————————————

```
(turn right 200)
(turn left 200)
(buy-coffee dark S cream 2)  valid
(move forward 20)
(deliver-coffee)
```

———————————————— (c) ————————————————

```
(move forward 20)
(buy-coffee light L sugar 12) <-- invalid cannot have 12 sugars
(move backward 20)
(deliver-coffee)
```

———————————————— (d) ————————————————

```
(turn left 20)
(move forward 300)
(turn right 30)
(move forward 10)
(buy-coffee dark S 2)  <-- invalid must have sugar or cream keyword
(deliver-coffee)       <-- invalid must have a motion-command
```

[15] 4. A program for a day at the horse races at the Blue Bonnets race track might look like the following

```
Race 1 9:30 am Fillies 3/4mi 4
   14 32:1 H: C rules          O: G. Lee  J: R. St-Aubin
   34 16:1 H: Assembly so cool O: B. Dow
...
Race 2 1:40 pm 1 yr olds 1 1/4mi $10,000
   14 100:1 H: Schemers Delight O: P. Belleville
   34 14:1  H: Prolog Clause    O: D. Poole      J: M. Horsch
...
```

A program consists of 1 to 5 races. Each race starts with the word "Race" and the race number. Then comes the time of the race (in 12-hour format: hh:mm where hh ranges from 1-12 and mm from 0-59 and the morning/afternoon indicator). This is followed by the type of horse and the length of the race, which is in miles (is followed by "mi"), and can be a number (like 1) or a fraction (like 3/4) or a combination of both (like 1 3/4). The longest race possible is 9 miles, and fractions are always in 1/4 mile increments. At the end of the line comes either the purse (the amount the winner gets), or the number of a race for which this is the qualifying race. Purses can be any sum greater than 0 preceded by a "$", but must be in the notation with sets of three digits separated by commas, and must not include any leading zeros. Thus, `$10,900` and `$9,999,999` are valid purses, but `10,990` and `$01,393` are not.

Following the header comes a list of 2 or more horses, each of which is a number, the odds (number ":" number), followed by the horse name (preceded by "H:"), the owner name (preceded by "O:") and the jockey name (preceded by "J:"). The jockey name is left out if the owner is the jockey. Write the EBNF for a horse race program at Blue Bonnets. Some of the categories you might want to use have been defined already.

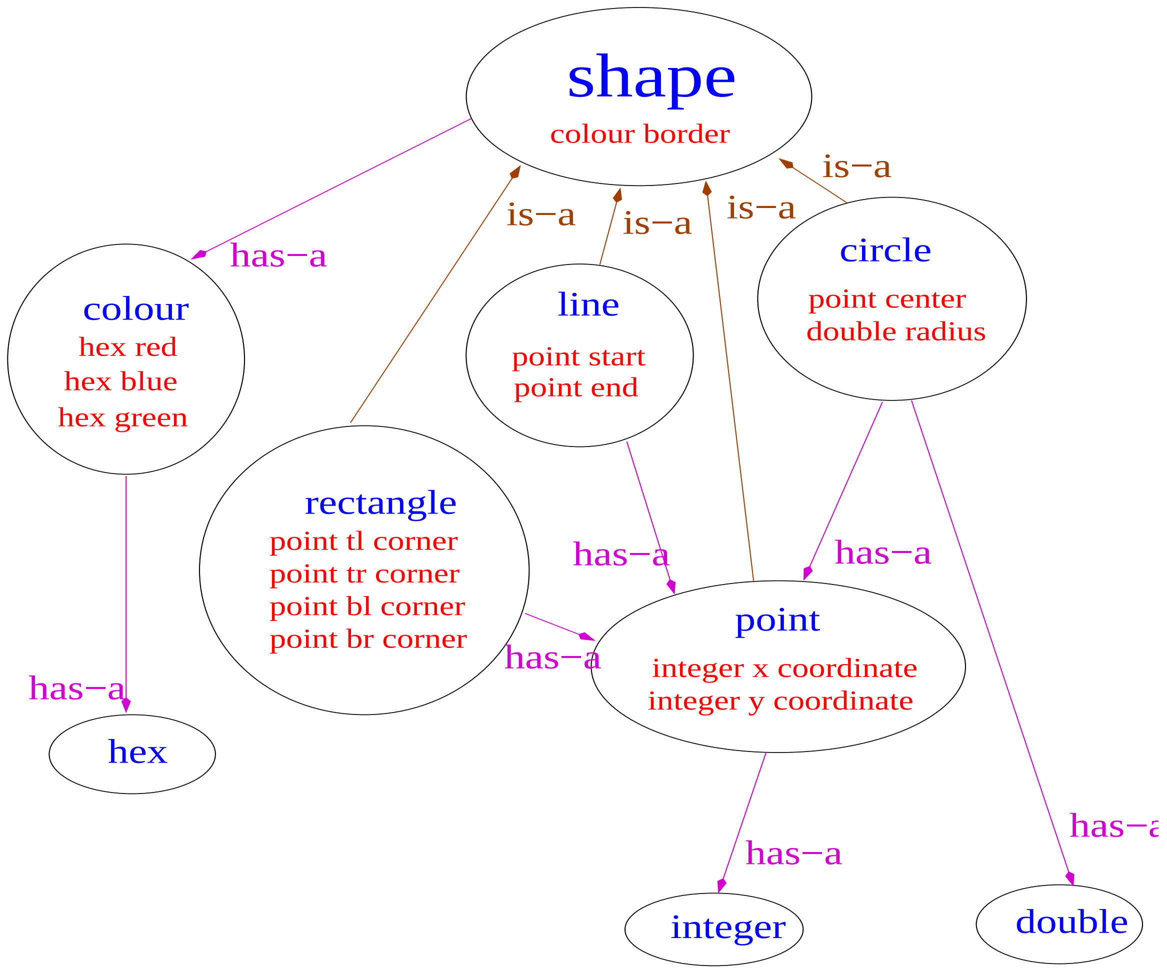
```
program: Race [Race] [Race] [Race] [Race]
Race: ``Race'' one5 time horse-type length (``$'' purse | less5) horselist
horselist: horse horse {horse}
time: one12``:'' one5 digit (``am'' | ``pm'')
length: one9 | fraclen | (one9 fraclen)
fraclen: one3 ``/4''
purse: ((one9 | one9 digit | one9 digit digit) {``,'' threeDig})
threeDig: digit digit digit

horse: number odds ``H:'' horse-name ``O:'' owner-name [``J:'' jockey-name]
odds: number ``:'' number

horse-type: ``Fillies'' | ``1 yr olds'' | ``2 yr old mares''
horse-name: ``C rules'' | ``Assembly so cool'' | ``Schemers Delight'' | ``Prolog Clause''
owner-name: ``G. Lee'' | ``B. Dow'' | ``P. Belleville'' | ``D.Poole''
jockey-name: ``R. St-Aubin'' | ``M. Horsch'' | ``R. Dearden''
one3: ``1'' | ``2'' | ``3''
one5: one3 | ``4'' | ``5''
six9: ``6'' | ``7'' | ``8'' | ``9''
one9: one5 | six9
one12: one9 | ``10'' | ``11'' | ``12''
digit: ``0'' | one9
number: one9 {digit}
```

[12] 5. A drawing program can display a variety of shapes including points, lines, circles and rectangles. A point represents the X and Y coordinates of a pixel (picture element) on the screen (coordinates are integer valued). A line is represented by a starting point and an ending point. A circle is represented by the center point, and the radius in pixels (which may be a fractional value). A rectangle is represented by four points, one for each corner. All shapes have a border colour. Colours are made up from a red, a green and a blue 2-digit hexadecimal number.

Use the Object Oriented Design technique discussed in class to design the classes for this program For each class, draw a circle labeled with the name of the class. Then, link the circles up with arrows labeled with the type of relationship between the classes (has-a or is-a). Finally, look at each has-a relationship, and write inside each circle what the member variables and their types will be for each object of that class. If you use a predefined C++ class, you do not need to specify its member variables or their types.



[20] 6. A square matrix is represented by the following class

```
class Matrix {
private:
  vector< vector<double> > mat;
  int size
  // calculates (-1)^d
  int pmin1(int d);
public:
  // constructs a new square sz by sz matrix
  Matrix(int sz);
  // returns the size of the matrix
  int getsize() { return size; }
  // sets the matrix element at (row,col) to be x
  void setTo(int row, int col, double x);
  // returns the sub-matrix of this matrix
  // obtained by removing the ith row and the jth column
  Matrix subMatrix(int i, int j);
  // calculates the determinant of a matrix
  double determinant(int i=0);
};
```

The member variable `mat` represents the matrix itself. The only thing you need to know about this collection of doubles is that the element at the `i`th row, `j`th column can be accessed using `mat[i][j]`. The **determinant** of a matrix A, $|A|$, is very useful in the analysis and solution of systems of linear equations. The determinant can be calculated using recursive *Laplacian expansion*, in which you first pick any row i of the matrix A, and expand along that row using

$$|A| = \sum_j (-1)^{(i+j)} A(i,j) |C^{ij}|$$

where $A(i,j)$ is the element of A at row i, column j, and C^{ij} is the sub-matrix of A obtained by removing the i^{th} row and j^{th} column from A. see over for questions

[12] a. Implement the function `subMatrix` declared above which returns C^{ij} for the matrix it is being called on.

```
Matrix Matrix::subMatrix(int row, int col) {
  Matrix smaller(size-1);
  for (int i=0; i<row; i++) {
    for (int j=0;  j<col; j++)
      smaller.mat[i][j] = mat[i][j];
    for (int j=col+1;  j<size; j++)
      smaller.mat[i][j-1] = mat[i][j];
  }
  for (int i=row+1; i<size; i++) {
    for (int j=0;  j<col; j++)
      smaller.mat[i-1][j] = mat[i][j];
    for (int j=col+1;  j<size; j++)
      smaller.mat[i-1][j-1] = mat[i][j];
  }
  return smaller;
}
```

[8] b. Implement the function `determinant` declared above which calculates the determinant of the matrix it is called on. This member function should be recursive, and should (probably) make use of your `subMatrix`. You can use the private member function `pmin1` in your solution.

```
double Matrix::determinant(int i) {
  if (size == 1)
    return mat[0][0];
  double det = 0;
  Matrix newmat(size-1);
  int sign=1;
  for (int j=0; j<size; j++) {
    sign = pmin1(i+j);
    newmat = subMatrix(i,j);
    det += sign*mat[i][j]*newmat.determinant();
  }
  return det;
}
```

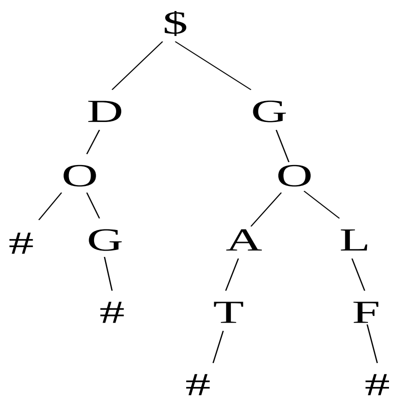
[10] 7. A large whole number can be represented as a string `"93828938747382"`, or as an array of integers (which are only 0-9) Write a function, `convertBigNum`, which converts from the string representation to the array representation. Your function should take a constant reference to a string as an argument, and should return a pointer to a dynamically allocated array of integers (of the correct size) which contains the string. Thus,

```
string x("489338389291");
int *y = convertBigNum(x);
// y is an array of \verb+x.size()=12+ integers:
// {4,8,9,3,3,8,3,8,9,2,9,1}.
```

HINT the character '0' is ASCII number 48.

```
int * convertBigNum(const string & x)
{
  int * tmp = new int[x.size()];
  for (int i=0;i <x.size(); i++) {
    tmp[i] = x.at(i)-48;
  }
  return tmp;
}
```

[15] 8. A **Trie** is a data structure for storing a list of words. A **Trie** is a graph which is structured like a tree, and in which every node is a character. Every path from the root of the Trie to a leaf is a word preceded by a `$` and followed by a `#`. A word will only appear in a Trie once. For example, here is a Trie which stores the list of words `"DOG"`,`"GOAT"`,`"GOLF"` and `"DO"`:



Tries are described by the following class:

```
class Trie {
private:
  // omitted

public:
  // returns the character at the root of the Trie
  char root();

  // returns a list of Tries which are the
  // subTries of this Trie. If called on a leaf Trie,
  // this returns an empty list.
  list<Trie> subtrees();
};
```

Write a function `trie_search` which takes a reference to a Trie and a string as arguments, and returns the boolean result of checking if the string is in the Trie. Thus, if `t` is the Trie shown above, `trie_search(t,"DO")` will return `true`, while `trie_search(t,"GOLFER")` will return `false`. **HINT:** Your function can be recursive. Note that the empty string is in any Trie with a '#' at the root of one of its subtree, and that a string s is in a trie if (1) the first character of the string is at the root of one of the subtrees and (2) the rest of the string is in that subtree whose root is the first character.

```
bool trie_search(Trie & t, string & s) {
  list<Trie> st = t.subtrees();
  list<Trie>::iterator ist = st.begin();

  char what('#');
  if (s.size() > 0)
    what = s.at(0);

  while (ist != st.end() && (*ist).root() != what)
    ist++;
  if ((*ist) == st.end())
    return false;
  else if ((*ist) == '#')
    return true;
  else
    return trie_search(*ist,substr(s,1,s.size()-1));
}
```

[30] 9. This question deals with vectors of integers.

[5] a. It is easy to remove the last element from a STL `vector` using the `pop_back()` method. If we don't care about the order of the elements in the vecor, we can remove an element at an arbitrary position, `pos`, as follows: (1) move the value at the end of the `vector` into position `pos` (2) remove the last element of the vector. Write a function `removeFromMiddle` which takes a reference to a `vector<int>`, and an integer position, `pos`, and removes **and returns** the element of the vector at position `pos` using this method.

```
int removeFromMiddle(vector<int> & x, int pos) {
  int r = x[pos];
  x[pos] = x[x.size()-1];
  x.pop_back();
  return r;
}
```

[5] b. Write a function, `shuffle`, which takes a reference to a vector of int, and rearranges the elements of the vector in a random order. You **should not** do this by simply swapping random values in the vector some number of times, as this will not guarantee a good shuffle. Use the function `int random(int mx)`, which generates a random integer between `0` and `mx` (including `0` and `mx`). **HINT:** Use your function `removeFromMiddle` from the first part of this question (even if you haven't written it).

```
void shuffle(vector<int> & x) {
  int r,i(0);
  vector<int> y(x);
  while (!y.empty())
    x[i++] = removeFromMiddle(y,rand(y.size()-1));
}
```

[8] c. Write a function `sorted`, which takes a reference to a vector of int, and returns true if the vector is sorted in ascending order, false otherwise.

```
bool sorted(vector<int> & x) {
  int i(0);
  bool sorted(true);
  while (sorted && i < x.size()-1) {
    if (x[i] < x[i+1])
      i++;
    else
      sorted = false;
  }
  return sorted;
}
```

[4] d. “Rediculous Sort”, invented by Yoshe J. Reeds, is a rediculous method of sorting a vector of integers which works as follows:

(1) shuffle the vector,

(2) check if its sorted. If it is, then stop, otherwise go back to (1).

Yoshe cannot program in C++, and so has asked you to implement his sorting method. Write a function `rediculousSort`, which takes a reference to a vector of int, and sorts the vector using Yoshe's method. Your function should return the number of shuffles that were used to sort the vector.

```
int rediculousSort(vector<int> & x) {
  int n(0);
  while (!sorted(x)) {
    shuffle(x); n++;
  }
  return n;
}
```

[3] e. What is the worst-case time complexity of `rediculousSort`?

```
with some infinitely small non-zero probability,
it will take infinite time
```