The 2008 Iverson Computing Science Competition
May 27, 2008
V3.2

Name: ____

School: ____

City: ____

Grade: _______

Are you taking AP or IB Computer Science (Yes/No): ______________

Have you completed Programming 5 (Yes/No/Taking it this term): ____________

Question	Question Name	Difficulty Level	Mark
1	Find the Gold	1	
2	Finite-State Machines	1	
3	Number Properties	1	
4	Playing the Slots	1	
5	Language Grammars	2	
6	The X-Machine	2	
7	Poster Covering	2	
8	Compressed Text	2	
TOTAL			

Overview:

This is a traditional 2 hour paper and pencil exam. It is divided into two sections: *required*, and *selected*. All answers, including rough work, are to be written in this booklet.

Question 1 and 2 are required. You should attempt the two required questions. The required questions do not require any particular programming language or advanced knowledge. They test your ability to read and understand computation. Please ask for assistance if you do not understand some aspect of the question, or the language.

Questions 3 to 8 are selected. You should attempt two of the selected questions. The selected questions constitute the main part of the competition. Each question is marked with a difficulty level of 1 or 2, where 1 is less difficult than 2. The level is based on the opinion of our question reviewers, and may not be that accurate. All questions will be marked out of ten but the questions will have different weights. A question of difficulty level 2 is worth 1.5 times a question of difficulty level 1. We'll be interested in getting your feedback about the questions after the competition.

If you have the time, feel free to do more than two questions from this section. We feel that the two required questions and two selected questions will use up most of your exam time but we are interested in seeing what you can do. Don't feel pressured, but if you do finish the two required questions and two selected questions feel free to do more. All questions will be marked.

Programming Language:

The questions that require programming can be answered using any programming language you wish (for example, VB, C/C++, Java, or Perl). If you wish you can even use pseudo-code. However, if you do so, be sure to provide an adequate amount of detail so the markers can determine if your solution is correct. Your pseudo-code should be detailed enough to allow for near direct translation into an appropriate programming language.

Our primary interest is in your higher order thinking skills rather than on your code wizardry. So a demonstration of logical thinking and systematic problem solving approaches will count for more than a mastery of one particular language's syntax. Be this as it may, you will still have to use some type of coding language to demonstrate what you can do. However minor syntactical errors will be ignored and we encourage you to put in comments where needed to clarify your code.

Suggestions:

1. Read the problem descriptions *carefully*. To get full marks *all* problem specifications have to be met. You can assume that only valid input values will be entered by the user. You do not need to include out-of-range or data type error checks in the input section of your programs.

2. Where feasible, sample executions of the desired program have been included for each problem. Review them carefully to make sure you haven't missed any specifications and to get hints as to how to proceed. In these sample executions the sample data entered by the user are underlined.

3. We recommend that you think through your program prior to writing any code. Use pseudo-code, diagrams, screen displays, tables or any other aid to help you plan your code. We will be looking at your rough work. If that work is present you won't have to rely completely on your coded solution to get marks. Remember we are looking for the key computing ideas, not specific coding details (which really are impossible to remember over a long term period). In particular you can introduce your own "built-in" functions for sub tasks such as reading the next number, or the next character in a string, or loading an array. Just make sure you specify the pre and post conditions of your "built-in" sub-programs.

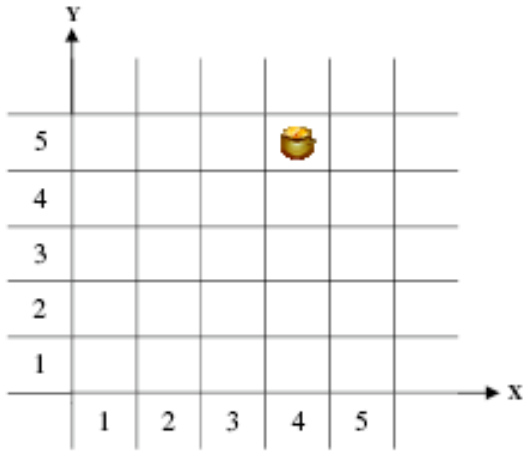
4. Take a look at *all* of the questions before deciding on the ones to attempt as you will not likely be able to finish all of them.

5. Make sure to include English language comments to explain non-obvious or clever parts of your solution.

Required Questions

Question 1 (Required) – Difficulty Level 1 – Find the Gold

Imagine a two-dimensional grid of square cells. There is a pot of gold that is *guaranteed* to be located somewhere in the grid. The gold occupies exactly one cell in the grid. For instance, the gold in the map below is located at position (4, 5).



You can check for the existence of gold in a grid cell using the function *has_gold*. The funtion *has_gold* takes two positive integer coordinates, *x* and *y*, and returns a 1 if there is gold in that cell, and 0 otherwise. In the above example, *has_gold(4, 5)* would return a value of 1, whereas *has_gold(2, 2)* would return a value of 0. Although the values of *x* and *y* are greater than 0, they have no set maximum (that is, no assumptions can be made about the size of the grid). *You should design and write your program as if the grid was infinite.*

Your task is to complete the following program. The program should search the grid for the gold, and when found should set the values of *gx* and *gy* to the coordinates of the gold. Since the gold is guaranteed to be present, you can assume that one of the squares of the grid contains gold. This means that even though the grid is of indeterminate size a properly written program will terminate.

Possible Hint: Approach the problem much the same way as a search and rescue *team* might look for a lost hiker.

```
int main() {
   int gx;
   int gy;
  /* your code would go here */


  return 0;
}
```

Code for Question 1:

Question 2 (Required) - Difficulty Level 1 - Finite-State Machines

A finite-state machine (FSM) is another kind of computation device. Although you don't usually encounter them directly, finite-state machines exist inside almost any device that does something interesting: kitchen appliances, remote control toys, simple phones (the complex ones have full computers), and so on.

There are three primitive concepts associated with a FSM, each associated with part of a FSM diagram:

State – a circle that indicates the current settings of the parts of the machine. Typically the state has a label that tells you what it means.

Transition - a potential change from one state to another (possibly the same) state. The transition is labeled with the events that cause the transition to occur. There is often a transition named DEFAULT. This is the transition that is taken when the event that occurs does not match a specific transition from the current state.

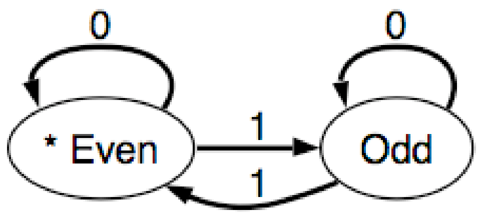
Event - something that causes a transition. For example, pushing a button, or reading a character from the input, or the ticking of a clock are events.

A FSM does a *computation* by starting in a specific start state (marked with a *), and then waiting for events to occur. Each event causes the machine to change state by following the transitions that matches the event. The computation stops when there are no more events; or an event occurs that does not have a matching transition and there is no DEFAULT transition from the current state.

A good physical intuition for these notions is: A state is a possible location that you can be at on a map. When you are at one location a transition is a road that can take you to another location. An event causes you to choose a particular road to travel. A computation is a road-trip, or path, between a start location and an end location.

So when you trace a FSM computation, think of putting a marker (like a pebble) on the current state, and when an event occurs, moving the marker to the next state along the transition that matches the incoming event.

Example: Here is a FSM that processes two possible kinds of events: 0 and 1



This FSM starts in state **Even** (the * means start here). Every time a 1 event occurs the FSM makes a transition to the other state. But every time a 0 event occurs the FSM makes a transition back to the same state. Thus, the name of the current state tells you whether the number of 1's that have arrived so far is odd (**Odd**) or even (**Even**).

Your Problem: A vending machine contains the following state machine to compute whether a customer has put in the correct amount. Items in the vending machine cost 20 cents, and can be paid for in any combination of 5 and 10 cent coins. You must put in exact change, and once you start putting in coins you must continue until you reach a total of 20 cents, at which point the machine enters the **20** state, and it will then allow you to make your choice of item. If you ever insert something other than a 5 or 10 cent coin, like a 25 or 50 cent coin; or if you enter in too much money (the machine cannot make change) then the machine enters the **Error** state. Note that the coin events are named 5c, 10c, 25c, and 50c.

Note the extensive use of DEFAULT transitions. For example in state **0** inserting a 25 or 50 cent coin will cause the DEFAULT transition to be taken to the **Error** state.

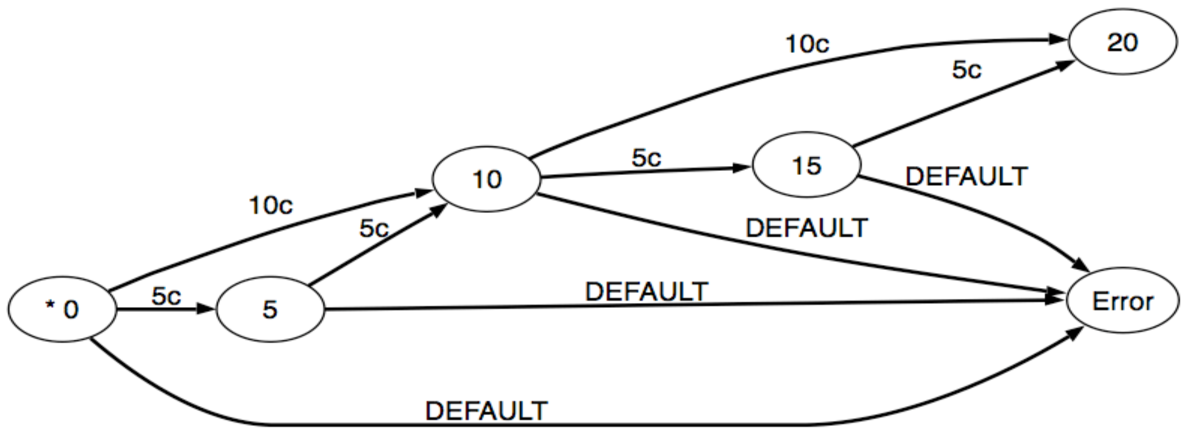
Customers have been complaining that under certain circumstances they have the correct change but that the machine enters the error state when they are inserting their coins into the vending machine.

Meanwhile, the price of items in the vending machine has increased to 25 cents.

You have been sent out to service the machine. *You are to modify the machine so that the problem the customers have been complaining about is fixed* and *that the price is increased to 25 cents.* You do this by altering existing transitions and adding new ones to the state machine below, and possibly adding some new states.

Explain what the original error was in the state machine.

Write your modifications directly on the state machine diagram below.



Selected Questions

Question 3 (Selected) – Difficulty Level 1 – Number Properties

A divisor of a positive integer is another positive integer that evenly divides into the first. For example, 3 and 4 are divisors of 12. The complete set of divisors of 24 is: 1, 2, 3, 4, 6, 8, 12, 24. Note that y is a divisor of x if x % y is 0, where % is the mod function in most programming languages.

A positive integer is called *prime* if its only divisors are itself and 1. By convention, 1 is not itself a prime. Thus, the sequence of primes begins:
2, 3, 5, 7, 11, 13, 17...

A positive integer is called *perfect* if it is the sum of all of its divisors less than itself. For example, 28 is a perfect number because 28 is divisible by 1, 2, 4, 7, 14; and 28 = 1+2+4+7+14. Again by convention, 1 is not regarded as a perfect number.

Your assignment is to write a program that will input one integer at a time, analyze it to determine if it is prime or perfect, and then print out an assessment. If the number is:

neither prime nor perfect, output "Dull".
is prime, output "Prime".
is perfect, output "Perfect".

You can assume that only integers are ever entered by the user. Entering a 0 or negative number should cause the program to quit.

Note: In the sample run below, the values entered by the user are underlined.

Sample Run:

```
Enter an integer: 28
28 is perfect
Enter an integer: 17
17 is prime
Enter an integer: 140
140 is dull
Enter an integer: 1
1 is dull
Enter an integer: 7
7 is prime
Enter an integer: 0
Bye!
```

Code for Question 3:

Question 4 (Selected) – Difficulty Level 1 – Playing the Slots

Gerry, Jennie and Jessie Gullible (a set of fraternal triplets) have just been given a cheque from one of their relatives for their 18th birthday. To celebrate, they decide to visit the local casino and parlay this money into a bigger sum. They cash their cheque and clutching a small bag of loonies they move to the slot machines to make their fortune.

They notice that one row 18 of the casino has a bank of three different slot machines. As there are three of them and three machines and as the row number matches their age they take this as a good omen and decide to play these three machines. They decide that each triplet will take a handful of loonies from the bag and that they will play all three machines simultaneously either until they go broke or triple their money. Each machine takes one loonie per play.

However, you have been observing the machines, and unbeknownst to the triplets, you know that the machines follow a predictable pattern. The first machine pays always pays 8 loonies every 12th time it is played; the second machine always pays 5 loonies every 8th time it is played; the third always pays 7 loonies every 10th time it is played.

You decide to write a program that will predict how long the players will last before going broke. Broke means that they no longer have enough money to play all of the machines in a round, that is, having fewer than 3 loonies left to bet.

Your program should take as input the total number of loonies the players have to start with, plus your observations of the number of times each machine has been played since it last paid out. Your program should output the number of rounds of betting that occurs before the players go broke. Each round consists of placing a 1 loonie bet on each machine.

Note: The values entered by the user are underlined. Assume that only valid input values will be entered.

A sample run of the program…

```
How many loonies are you starting with? 25
How many times has the first machine been played since paying out? 3
How many times has the second machine been played since paying out? 5
How many times has the third machine been played since paying out? 4
The players play 20 rounds before going broke.
```

Another sample run of the program…

```
How many loonies are you starting with? 3
How many times has the first machine been played since paying out? 11
How many times has the second machine been played since paying out? 7
How many times has the third machine been played since paying out? 9
The players play 7 rounds before going broke.
```

Code for Question 4:

Question 5 (Selected) – Difficulty Level 2 – Language Grammars

Communication between people, between machines, and between people and machines is often done by using a language. For example, people might use English, programs are written in Java, and machines often talk between each other using XML.

A language is defined by a grammar which specifies how the sentences in the language are constructed. Natural languages, like English, don't have a full grammar that specifies exactly what you are allowed to say. But the artificial languages that we use with computers can have complete descriptions. In this case, a grammar for the language defines all the grammatically correct sentences that you can write. The grammar defines the form, or *syntax* of the language. It is possible to have grammatically correct sentences that don't make sense, such as "My fish is a bicycle". The study of the meaning of sentences is called *semantics*.

Since sentences are typically composed of words, a grammar can be though of as a way of generating all the syntactically correct combinations of words that make up sentences. The grammar itself is specified as rewrite rules that generate strings of words. The rewrite rules are very simple, and are written in a form that means *rewrite the symbol on the left hand side according to the right hand side of the rule*. For example, here is a grammar that describes how to construct simple sentences. The rules have been numbered in order to explain the example, the numbering is not necessary:

Rule 1:	Start => Sentence
Rule 2:	Sentence => Subject Verb Object
Rule 3:	Subject => 'Fred'
Rule 4:	Subject => 'Barney'
Rule 5:	Verb => 'eats'
Rule 6:	Verb => 'walks'
Rule 7:	Verb => 'says'
Rule 8:	Object => 'ribs'
Rule 9:	Object => 'I am hungry'
Rule 10:	Object => 'Dino'

The quotes indicate actual strings on which no rule substitution is allowed.

We generate various possible sentences using the following process:
1. Begin with the symbol Start.
2. Pick an unquoted symbol in the string we are generating, and find a rule that has that symbol on the left hand side. Replace the symbol on the left hand side by the list of symbols on the right hand side. Then keep repeating this process until all of the unquoted symbols have been processed or until you come to a symbol for which no rule exists.

The resulting string of symbols is a grammatically correct sentence.

For example, here is the sequence of replacements beginning with Start that we get when we apply the grammar rules in this order: 1, 2, 5, 3, and 10.

```
Start
Sentence
Subject Verb Object
Subject  'eats'  Object
'Fred'  'eats'  Object
'Fred'  'eats'  'Dino'
```

Note that each of the sentences generated by Rules 1 to 10 must consist of exactly three words. If we add additional rules to the grammar, we can make more complex sentences.

```
Rule 11: Sentence => Subject  Verbphrase
Rule 12: Verbphrase => Verb  Object
Rule 13: Verbphrase => Verbphrase  'and'  Verbphrase
```

So now we can make longer sentences. Here is what happens when we apply rules:
1, 11, 3, 13, 12, 7, 9, 12, 5, 10

```
Start
Sentence
Subject  Verbphrase
'Fred'  Verbphrase
'Fred'  Verbphrase  'and'  Verbphrase
'Fred'  Verb  Object  'and'  Verbphrase
'Fred'  'Says'  Object  'and'  Verbphrase
'Fred'  'Says'  'I am hungry'  'and'  Verbphrase
'Fred'  'Says'  'I am hungry'  'and'  Verb  Object
'Fred'  'Says'  'I am hungry'  'and'  'eats'  Object
'Fred'  'Says'  'I am hungry'  'and'  'eats'  'Dino'
```

Your Problem:

Grammars are particularly good for things like arithmetic expressions. Here is one for expressions involving addition and subtraction of the numbers 0, 1, 2:

```
Rule 1:  Start => Expr
Rule 2:  Expr => Number
Rule 3:  Number => '0'
Rule 4:  Number => '1'
Rule 5:  Number => '2'
Rule 6:  Expr => '-' Number
Rule 7:  Expr => '('  Expr  ')'  Oper  '('  Expr  ')'
Rule 8:  Oper => '+'
Rule 9:  Oper => '-'
```

Here is how we can derive the expression (1) + (- 2)

Start	Apply rule 1
Expr	Apply rule 7
'(' Expr ')' Oper '(' Expr ')'	Apply rule 8
'(' Expr ')' '+' '(' Expr ')'	Apply rule 2
'(' Number ')' '+' '(' Expr ')'	Apply rule 4
'(' '1' ')' '+' '(' Expr ')'	Apply rule 6
'(' '1’ ')' '+' '(' '-' Number ')'	Apply rule 5
'(' '1’ ')' '+' '(' '-' '2' ')'	done, no rules apply

Dropping all the quotes we get: (1) + (- 2)

Your Problem:

Part 1: Using the format in the preceding example to explain your steps, derive the expression:
(1) + (-2 + ((0) - (1)))

Part 2: Is it possible to put together a sequence of rules that generates the expression (- - 1) ? Explain.

Part 3 Bonus Challenge: Since the expressions generated by the above grammar are proper arithmetic formulas, each of them can be evaluated to an integer value. Some of these expressions will have an even value (eg. -4, 0, 22) and others will have an odd value (-1, 1, 99). Develop a new grammar that generates all of the even-valued expressions of the original grammar. For example, these even-valued expressions would be generated by your new grammar:

```
0
( 0 ) + ( 2 )
( 0 ) + ( ( 1 ) + ( 1 ) )
( 1 ) + ( 1 )
( ( 2 ) + ( 1 ) ) + ( ( 1 ) + ( 2 ) )
```

But these odd-valued expressions would not

```
1
( 0 ) + ( 1 )
( 1 ) + ( ( 1 ) + ( 1 ) )
( 2 ) + ( 1 )
( ( 2 ) + ( 2 ) ) + ( ( 1 ) + ( 2 ) )
```

Hint: keep track of whether an expression is odd or even, and how odd and even combine with addition and subtraction.

Grammar for Question 5 Part 3:

Question 6 (Selected) – Difficulty Level 2 - The X-Machine

Modern computers are very complex. But they were not always so. Early computers consisted of memory, a processing unit that fetched instructions from memory, and limited input/output capabilities.

The X-machine is an example of a simple, yet interesting computer. The basic machine consists of a memory of M cells, with cells numbered 0, 1, 2, ..., M-1. The maximum that M can be is 10000. Each cell can hold an integer in the range -10000 to +10000.

Since cells do not have names, we need to have a notation for talking about the contents of a cell: The notation [x] means the contents of memory location x. Suppose cell 1 contains the number 10. Then the contents of cell 1, that is [1], equals 10. The notation [[1]] means the contents of the contents of cell 1, which is in this case the same as [10].

One of the memory locations has special significance. Location 0 contains the number of the cell where the next instruction to be executed begins. That is, the first part of the instruction is stored in cell number [0]. The value of that part of the instruction is called the *opcode* (operation code). Thus the opcode of the next instruction to be executed is [[0]], which is the contents of the location given by the contents of cell 0.

There is only one format of instruction for the X machine. Each instruction consists of 4 adjacent memory cells. As we said above, the beginning of the next instruction to be executed is located in cell [0]. The contents of this location and the next 3 define the instruction to be executed. In other words, the next instruction to be executed consists of these 4 integers:

[[0]]	Opcode: 1001 (SUB), or 1002 (CMP)
[[0]+1]	X
[[0]+2]	Y
[[0]+3]	Z

There are only two possible instructions for this machine. Any opcode other than 1001 (SUB) or 1002 (CMP) causes the machine to halt.

SUB X Y Z means subtract the contents of cell Y from the contents of cell X and store the result in cell Z. Then move to the next instruction, which is 4 cells after the first cell of this instruction.

Using the [ ] notation, the effect of this instruction is as follows:

```
[Z] := [X] - [Y]
[0] := [0] + 4
```

The notation [Z] := [0] means replace the left hand side by the right, or in this case, replace the contents of cell Z by the contents of cell 0.

CMP X Y Z means compare the contents of cell X to the contents of cell Y, and if less, use location Z as the next instruction to execute. Otherwise, continue with the next instruction.

Using the [ ] notation, the effect of this instruction is as follows:

```
if [X] < [Y] then
   [0] := Z
else
   [0] := [0] + 4
```

Note: the contents of cell [0] is being replaced by Z, not by the contents of cell Z.

Running a Program: The machine runs a program by first loading the memory with the values of the program (don't worry about how this is done) and setting the contents of cell 0 to 1. Then it fetches the instruction that begins in location [0], executes it, and keeps doing this fetch-execute cycle until it encounters an invalid opcode and halts.

We can describe exactly how the X machine executes instructions with this pseudo-code program:

```
[0] := 1
running := true
While running {

   # SUB X Y Z
   if [[0]] is 1001 then {

      [[[0]+3]] := [[[0]+1]] - [[[0]+2]]
      [0] := [0] + 4

      }

   # CMP X Y Z
   else if [[0]] is 1002 then {

      if [[[0]+1]] < [[[0]+2]] then {
         [0] := [[0] + 3]
         }
      else {
         [0] := [0] + 4
         }

      }

   # oops, bad opcode, so halt
   else  {
      running = false
      }

   }
```

Example: To illustrate the X-machine, here is a simple X-machine program that puts the minimum of cells 100 and 101 into cell 50. The CONST instructions are just a way of specifying the initial contents of cells that do not contain instructions. The number to the left of each instruction is the cell number that the instruction starts at:

```
       # program to put the minimum of [100] and [101] into [50]
       # clear cells 50, 51
1:      SUB 50 50 50
5:      SUB 51 51 51
9:      CMP 100 101 21
        # [101] is smaller, copy negation into cell 50
13:     SUB 50 101 50
        # skip the next instruction
17:     CMP 98 99 25
        # [100] is smaller, copy negation into cell 50
21:     SUB 50 100 50
        # change sign of cell 50
25:     SUB 51 50 50
        # now stop

 98:    CONST 0
 99:    CONST 1
100:    CONST 34
101:    CONST -50
```

The program looks like this in memory just before running:

0: 0	17: 1002
1: 1001	18: 98
2: 50	19: 99
3: 50	20: 25
4: 50	21: 1001
5: 1001	22: 50
6: 51	23: 100
7: 51	24: 50
8: 51	25: 1001
9: 1002	26: 51
10: 100	27: 50
11: 101	28: 50
12: 21	…
13: 1001	98: 0
14: 50	99: 1
15: 101	100: 34
16: 50	101: -50

And when it runs is does this:

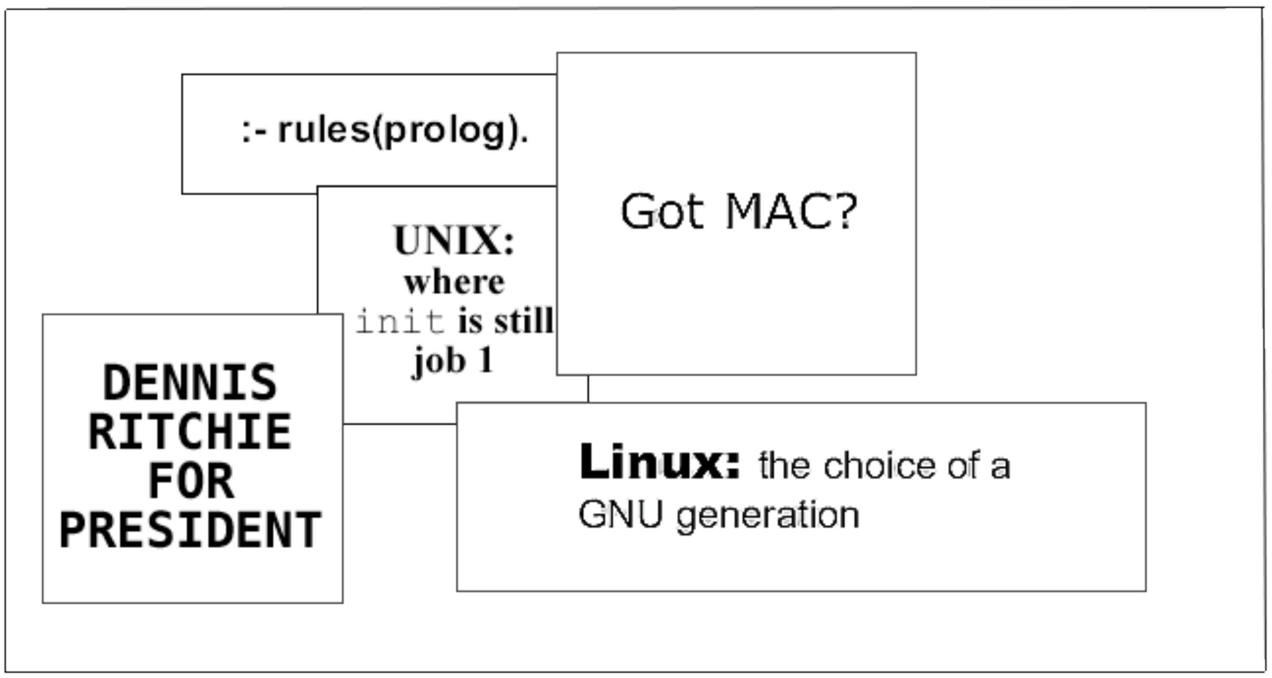
```
Exec  1: SUB 50 50 50
Subtracting [50]=0 - [50]=0
Writing [50]=0
Exec  5: SUB 51 51 51
Subtracting [51]=0 - [51]=0
Writing [51]=0
Exec  9: CMP 100 101 21
Comparing cell [100]=34 with [101]=-50
Exec 13: SUB 50 101 50
Subtracting [50]=0 - [101]=-50
Writing [50]=50
Exec 17: CMP 98 99 25
Comparing cell [98]=0 with [99]=1
Exec 25: SUB 51 50 50
Subtracting [51]=0 - [50]=50
Writing [50]=-50
Halt at 29
```

Your Problem:
Write an X-machine program to sum up the integers 1 ... n, where n is initially stored in cell 33, and the sum is finally stored in cell 34. Write your program using the CMP, SUB, and CONST commands and put it in the table below. If the first column put the cell number where the instruction starts, and the instruction in the second column.

Cell #	
0	Reserved for cell number of next instruction
1	
5	
9	
13	
17	
21	
25	
29	
33	
37	

Question 7 (Selected) – Difficulty Level 2 – Poster Covering

Imagine a dorm room occupied by a number of computer science students. Each student has a set of posters that they would like to place on the wall, however, poster placement is done hastily, and some of the posters overlap each other. The following figure shows an example.


:- rules(prolog).
UNIX: where init is still job 1
Got MAC?
DENNIS RITCHIE FOR PRESIDENT
Linux: the choice of a GNU generation

Your task will be to write a program that computes the total area of the wall being covered by the posters. You may assume the following:

1. The size of the wall is 500 cm wide by 250 cm high.
2. The size of the posters will always be measured in cm, and the dimensions will always be whole numbers (i.e. the size of a poster might be 60cm x 80cm, but never 40.5cm x 84.4 cm).
3. The corners of the poster will always be whole numbers. That is, the bottom left corner of a poster might be at location (60cm, 80cm), measured from the bottom left corner of the wall, but never at (40.5cm, 84.4cm).
4. The number of posters will be given by the integer *n*. Each poster will be indexed from 1 to n.
5. There are four available functions:
   a. *x(index)* takes the number of a poster, and returns the x-coordinate of its bottom left corner (relative to the left edge of the wall). For instance, calling *x(2)* will return the *x-coordinate* of the bottom left corner of the second poster.
   b. *y(index)* takes the number of a poster, and returns the y-coordinate of its bottom left corner (relative to the bottom edge of the wall).
   c. *height(index)* takes the number of a poster, and returns its height.
   d. *width(index)* takes the number of a poster, and returns its width.

Code for Question 7:

Question 8 (Selected) – Difficulty Level 2 – Searching Compressed Text

Data compression is the task of representing a sequence of data with fewer bits than its original representation. One way to do this is with *run-length* encoding. In run-length encoding, consecutive sequences of the same character are stored as a single version of that character, along with that character's count. For instance, the text:

```
   AAAABBCCCDEEEEEEEEEE
```

could be stored as:

```
   4A2B3C1D10E
```

which requires only 11 characters instead of 20.

Your task is to write a program that reads in two character strings. The first string will be some text that has been compressed using run-length encoding. The second string will be a pattern in that text to find (which is not encoded). If the pattern is found, the program should print out "Pattern found!". If the pattern cannot be found, the program should print "Pattern not found!".

For simplicity, you may assume that the counts and the characters are separated by spaces. You may also assume that the number of character-count pairs in the text is less than or equal to 100, and that the size of the pattern is less than 50 characters.

Here are some examples of the program in action:

```
Enter text: 4 A 2 B 3 C 1 D 10 E
Enter a pattern: AA
Pattern found!


Enter text: 4 A 2 B 3 C 1 D 10 E
Enter a pattern: BBCCC
Pattern found!


Enter text: 4 A 2 B 3 C 1 D 10 E
Enter a pattern: ABBC
Pattern found!


Enter text: 4 A 2 B 3 C 1 D 10 E
Enter a pattern: ABBBC
Pattern not found!
```

Code for Question 8: