American Computer Science League

Theoretical Problem Conceptual Description

Organized by. Yeseong Lee

Contents

Computer Number System

All digital computers, from supercomputers to your smartphone, are electronic devices and ultimately can do one thing: detect whether an electrical signal is on or off. That basic information, called a *bit* (**bi**nary digi**t**), has two values: a 1 (or *true*) when the signal is on, and a 0 (of *false*) when the signal is off.

Larger values can be stored by a group of bits. For example, there are 4 different values stored by 2 bits (00, 01, 10, and 11), 8 values for 3 bits (000, 001, 010, 011, 100, 101, 110, and 111), 16 values for 4 bits (0000, 0001, ..., 1111), and so on. However, large numbers, using 0s and 1s only, are quite unwieldy for humans. For example, a computer would need 19 bits to store the numbers up to 500,000! We use of different number systems to work with large binary strings that represent numbers within computers.

<Different Number System>

A *number system* is the way we name and represent numbers. The number system we use in our daily life is known as **decimal number system** or **base 10** because it is based on 10 different digits: 0, 1, 2, ..., 8, and 9. The base of number is written as subscript to a number. For example, the decimal number "twelve thousand three hundred and forty-five" is written as 1234510. Without a base or explicit context, it's assumed that a number is in base 10.

In computer science, apart from the decimal system, three additional number systems are commonly used: **binary** (base-2), **octal** (base-8), and **hexadecimal** or just **hex** (base-16). Binary numbers are important because that is how numbers are stored in the computer. Octal and hexadecimal are used to represent binary numbers in a user-friendly way. Each octal symbol represents 3 binary bits and each hex digit represents 4 binary bits. For example, the decimal number 53201, stored as in the computer as the binary number 1000000111100010100, is represented by the octal number 2017054, and the hex number 81E2C. The table below compares the number systems:

Number System	Base	Digits Used	Examples
Binary	2	0,1	10110_2, 10110011_2
Octal	8	0,1,2,3,4,5,6,7	75021_8, 231_8, 60012_8
Decimal	10	0,1,2,3,4,5,6,7,8,9	97425_{10} or simply 97425
Hexadecimal	16	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F	$54A2DD0F_{16}$

In general, N bits have 2^N different values. The computers aboard the Apollo spacecraft had 8-bit words of memory. Each word of memory could store 256 different values, and the contents were displayed using 2 hex characters.

The following table shows the first 20 numbers in decimal, binary, octal, and hexadecimal:

Decimal	Binary	Octal	Hexadecimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
17	10001	21	11
18	10010	22	12
19	10011	23	13
20	10100	24	14

ACSL Theoretical Problem Conceptual Description

<Converting to Decimal>

As we learned in the first or second grade, the decimal value of a decimal number is simply sum of each digit multiplied by its place value:

$12345=1\times10^4+2\times10^3+3\times10^2+4\times10^1+5\times10^0=10000+2000+300+40+5=12345$
$3079=3\times10^3+0\times10^2+7\times10^1+9\times10^0=3000+70+9=3079$

Analogously, the decimal value of a number in an arbitrary base is the sum of each digit multiplied by its place value. Here are some examples of converting from one base into base 10:

$11012=1\times2^3+1\times2^2+0\times2^1+1\times2^0=8+4+0+1=13_{10}$
$1758=1\times8^2+7\times8^1+5\times8^0=1\times64+7\times8+5\times1=64+56+5=125_{10}$
$A5E16=10\times16^2+5\times16^1+14\times16^0=10\times256+5\times16+14\times1=2560+80+14=2654_{10}$

<Converting from Decimal>

The algorithm to convert a number from an arbitrary base requires finding how many times successive powers of the base go into the number, starting with the largest power of the base less than the starting number. For example, converting 3306 from base 10 to octal proceeds as follows:

1) The largest power of 8 that is less than or equal to 3306 is 512 (8^3). Divide 3306 by 512:
   $3306=6*8^3+234$
2) The next power of 8 is 64 (8^2). Divide 234 by 64:
   $234=3*8^2+42$
3) The next smaller power of 8 is 8 (8^1). Divide 42 by 8:
   $42=5*8^1+2$
4) Finally, the next smaller power of 8 is 1 (8^0). Divide 2 by 1:
   $2=2*8^0$

The answer is 6352_8.

<Converting between Binary, Octal, and Hexadecimal>

Converting from octal to binary is simple: replace each octal digit by its corresponding 3 binary bits. For example:

$375_8=$ 011 111 $101_2=11111101_2$

Converting from hex to binary is also simple: replace each hex digit by its corresponding 4 binary bits. For example:

$FD_{16}=$ 1111 $1101_2=11111101_2$

Converting from binary to either octal or hex is pretty simple as well: group the bits by 3s or 4s (starting at the right), and convert each group:

$1000000111100010110 0=10\ 000\ 001\ 111\ 000\ 101\ 100=2017054_8$
$1000000111100010110 0=1000\ 0001\ 1110\ 0010\ 1100=81E2C_{16}$

Converting between base 8 and 16 is easy by expressing the number in base 2 (easy to do!) and then converting that number from base 2 (another easy operation)! This is shown below in Sample Problem #1.

ACSL Theoretical Problem Conceptual Description

<Using Hexadecimal Numbers to Represent Colors>

Computers use hexadecimal numbers to represent various colors in computer graphics because all computer screens use combinations of red, green, and blue light or RGB to represent thousands of different colors. Two digits are used for each so the hexadecimal number "#FF0000" represents the color red, "#00FF00" represents green, and "#0000FF" represents blue. The color black is "#000000" and white is "#FFFFFF".

The hash tag or number sign is used to denote a hexadecimal number. $FF_{16}=F(15)\times16+F(15)\times1=240+15=255_{10}$ so there are 0 to 255 or 256 different shades of each color or 256^3=16,777,216 different colors.

For example "salmon" is "#FA8072" which represents the decimal numbers 250 (hex FA), 128 (hex 80), and 114 (hex 72).

<Format of ACSL Problems>

The problems in this category will focus on converting between binary, octal, decimal, and hexadecimal, basic arithmetic of numbers in those bases, and, occasionally, fractions in those bases.

To be successful in this category, you must know the following facts cold:

1. The binary value of each octal digit 0, 1, ..., 7
2. The binary value of each hex digit 0, 1, ..., 9, A, B, C, D, E, F
3. The decimal value of each hex digit 0, 1, ..., F
4. Powers of 2, up to 4096
5. Powers of 8, up to 4096
6. Powers of 16, up to 65,536

<Sample>

Solve for x where $x_{16}=3676_8$.

Solution: One method of solution is to convert 3676_8 into base 10, and then convert that number into base 16 to yield the value of x.

An easier solution, less prone to arithmetic mistakes, is to convert from octal (base 8) to hexadecimal (base 16) through the binary (base 2) representation of the number:

3676_8	$=011\ 110\ 111\ 110_2$	convert each octal digit into base 2
	$=0111\ 1011\ 1110_2$	group by 4 bits, from right-to-left
	$=7\ B\ E_{16}$	convert each group of 4 bits into a hex digit

Recursive Functions

A definition that defines an object in terms of itself is said to be *recursive*. In computer science, recursion refers to a function or subroutine that calls itself, and it is a fundamental paradigm in programming. A recursive program is used for solving problems that can be broken down into sub-problems of the same type, doing so until the problem is easy enough to solve directly.

<Examples>

Fibonacci Numbers

A common recursive function that you've probably encountered is the Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, and so on. That is, you get the next Fibonacci number by adding together the previous two. Mathematically, this is written as

$$f(N)=f(N-1)+f(N-2)$$

Try finding f(10). No doubt, you have the correct answer, because you intuitively stopped when you reach f(1) and f(0). To be formal about this, we need to define when the recursion stops, called the *base cases*. The base cases for the Fibonacci function is f(0)=0, and f(1)=1. The typical way to write this function is as follows:

$$f(N)=\{Nf(N-1)+f(N-2) \text{ if } N\leq 1 \text{ if } N>1$$

Here is a Python implementation of the Fibonacci function:

```
def Fibonacci(x):
  if (x <= 1) return x
  return Fibonacci(x-1) + Fibonacci(x-2)
```

(As a challenge to the reader: How could you implement the Fibonacci function without using recursion?)

Factorial Function

Consider the factorial function, n!=n*(n−1)*...*1, with 0! defined as having a value of 1. We can define this recursively as follows:

$$f(x)=\{1x*f(x-1) \text{ if } x=0 \text{ if } x>0$$

With this definition, the factorial of every non-negative integer can be found.

Here is a Python implementation of the factorial function:

```
def Factorial(x):
  if (x == 0) return 1
  return x * Factorial(x-1)
```

Some Definitions

A few definitions: *Indirection recursion* is when a function calls another function which eventually calls the original function. For example, A calls B, and then, before function B exits, function A is called (either by B or by a function that B calls). *Single recursion* is recursion with a single reference to itself, such as the factorial example above. *Multiple recursion*, illustrated by the Fibonacci number function, is when a function has multiple self references. *Infinite recursion* is a recursive function that never returns because it keeps calling itself. The program will eventually crash with an *out of memory* error message of some sort.

ACSL Theoretical Problem Conceptual Description

This ACSL category focuses on mathematical recursive functions rather than programming algorithms. While many mathematical functions can be done iteratively more efficiently than they can be done recursively, many algorithms in computer science must be written recursively. The Tower of Hanoi which is referred to in one of the videos is a good example of this. Another example is given in the sample problems below.

<Sample>

1. Problem: Find g(11) given the following

$$g(x)=\{g(x-3)+13x \text{ if } x>0 \text{ otherwise}$$

Solution:

The evaluation proceeds top-down as follows:

$$g(11)g(8)g(5)g(2)g(-1)=g(8)+1=g(5)+1=g(2)+1=g(-1)+1=-3$$

We now use what we know about g(−1) to learn more values, working our way back "up" the recursion:

We now know the value of g(2):$g(2)=-3+1=-2$

We now know the value of g(5):$g(5)=-2+1=-1$

We now know the value of g(8):$g(8)=-1+1=0$

And finally, we now have the value of g(11):$g(11)=0+1=1$

2. Problem: Find the value of h(13) given the following definition of h:

$$h(x)=\{h(x-7)+1xh(x+3) \text{ when } x>5 \text{ when } 0\le x\le 5 \text{ when } x<0$$

Solution:

$$h(13)h(6)h(-1)h(2)=h(6)+1=h(-1)+1=h(-1+3)=h(2)=2\text{top rule, since } x>5\text{top rule, since } x>5\text{bottom rule, since } x<0\text{middle rule, since } 0\le x\le 5$$

We now work our way back up the recursion, filling in values that have been computed.

$h(-1)=h(2)=2$

$h(6)=h(-1)+1=2+1=3$

$h(13)=h(6)+1=3+1=4$

3. Problem: Consider the following recursive algorithm for painting a square:

1. Given a square. 2. If the length of a side is less than 2 feet, then stop. 3. Divide the square into 4 equal size squares (i.e., draw a "plus" sign inside the square). 4. Paint one of these 4 small squares. 5. Repeat this procedure (start at step 1) for each of the 3 unpainted squares.

If this algorithm is applied to a square with a side of 16 feet (having a total area of 256 sq. feet), how many square feet will be painted?

Solution:

In the first pass, we get four squares of side 8. One is painted, three are unpainted. Next, we have 3*4 squares of side 4. Three are painted (area=$3*4^2$), nine are not. Next, we have 9*4 squares of side 2. Nine are painted (area = $9*2^2$), 27 are not. Finally, we have 27*4 squares of side 1. Twenty-seven are painted. Therefore, the total painted is $1*8^2 + 3*4^2 + 9*2^2 + 27*1^2 = 175$.

What Does This Program Do?

Frequently, one must use or modify sections of another programmer's code. Since the original author is often unavailable to explain his/her code, and documentation is, unfortunately, not always available or sufficient, it is essential to be able to read and understand an arbitrary program.

This category presents a program and asks the student to determine that the program does. The programs are written using a pseudocode that should be readily understandable by all programmers familiar with a high-level programming language, such as Python, Java, or C.

<Description of the ACSL Pseudo-code>

We will use the following constructs in writing this code for this topic in ACSL:

Construct	Code Segment
Operators	! (not) , ^ or ↑(exponent), *, / (real division), % (modulus), +, -, >, <, >=, <=, !=, ==, && (and), || (or) in that order of precedence
Functions	abs(x) - absolute value, sqrt(x) - square root, int(x) - greatest integer <= x
Variables	Start with a letter, only letters and digits
Sequential statements	INPUT variable variable = expression (assignment) OUTPUT variable

Decision statements	IF boolean expression THEN
	Statement(s)
	ELSE (optional)
	Statement(s)
	END IF

Indefinite Loop statements	WHILE Boolean expression Statement(s) END WHILE
Definite Loop statements	FOR variable = start TO end STEP increment Statement(s) NEXT
Arrays:	1 dimensional arrays use a single subscript such as A(5). 2 dimensional arrays use (row, col) order such as A(2,3). Arrays can start at location 0 for 1 dimensional arrays and location (0,0) for 2 dimensional arrays. Most ACSL past problems start with either A(1) or A(1,1). The size of the array will usually be specified in the problem statement.
Strings:	Strings can contain 0 or more characters and the indexed position starts with 0 at the first character. An empty string has a length of 0. Errors occur if accessing a character that is in a negative position or equal to the length of the string or larger. The len(A) function will find the length of the string which is the total number of characters. Strings are identified with surrounding double quotes. Use [ ] for identifying the characters in a substring of a given string as follows: S = "ACSL WDTPD" (S has a length of 10 and D is at location 9) S[:3] = "ACS" (take the first 3 characters starting on the left) S[4:] = "DTPD" (take the last 4 characters starting on the right) S[2:6] = "SL WD" (take the characters starting at location 2 and ending at location 6) S[0] = "A" (position 0 only). String concatenation is accomplished using the + symbol

The questions in this topic will cover any of the above constructs in the Intermediate and Senior Division. In the Junior Division, loops will not be included in contest 1; loops will be used in contest 2; strings will be used in contest 3; and arrays will be included in contest 4.

<Sample>

After this program is executed, what is the value of B that is printed if the input values are 50 and 10?

```
input H, R
B = 0
if H>48 then
    B = B + (H - 48) * 2 * R
    H = 48
end if
if H>40 then
   B = B + (H - 40) * (3/2) * R
   H = 40
end if
B = B + H * R
output B
```

Solution:

This program computes an employee's weekly salary, given the hourly rate (R) and the number of hours worked in the week (H). The employee is paid an hourly rate for the number of hours worked, up to 40, time and a half for the overtime hours, up to 48 hours, and double for all hours over 48. The table monitors variables B and H:

B	H
0	50
40	48
160	40
560	40

Therefore, the final value of B is 2*2*10 + 8*3/2*10 + 40*10 = 40 + 120 + 400 = 560.

Prefix/Infix/Postfix Notation

The order of precedence is often given the mnemonic of Please excuse my dear Aunt Sue or PEMDAS: **p**arentheses, **e**xponentiation, **m**ultiplication/**d**ivision, and **a**ddition/'**s**ubtraction. Multiplication and division have the same level of precedence; addition and subtraction also have the same level of precedence. Terms with equal precedence are evaluated from left to right

The algorithm to evaluate an *infix* expression is complex, as it must address the order of precedence. Two alternative notations have been developed which lend themselves to simple computer algorithms for evaluating expressions.

In *prefix* notation, each operator is placed before its operands.
In *postfix* notation, each operator is placed after its operand.
In *prefix* and *postfix* notations, there is no notion of order of precedence, nor are there any parentheses.
The evaluation is the same regardless of the operators.

<Converting Expression>

An algorithm for converting from infix to prefix (postfix) is as follows:

- Fully parenthesize the infix expression. It should now consist solely of "terms": a binary operator sandwiched between two operands.
- Write down the operands in the same order that they appear in the infix expression.
- Look at each term in the infix expression in the order that one would evaluate them, i.e., inner-most parenthesis to outer-most and left to right among terms of the same depth.
- For each term, write down the operator before (after) the operands.

One way to convert from prefix (postfix) to infix is to make repeated scans through the expression. Each scan, find an operator with two adjacent operands and replace it with a parenthesized infix expression. This is not the most efficient algorithm, but works well for a human.
A quick check for determining whether a conversion is correct is to convert the result back into the original format.

Examples

$X = \left(AB - \frac{C}{D}\right)^E$ $(3 * 4 + \frac{8}{2})^{(7-5)}$

Infix to Prefix

(X = (((A * B) - (C / D)) ↑ E))
X A B C D E
X * A B C D E
X * A B / C D E
X - * A B / C D E
X ↑ - * A B / C D E
= X ↑ - * A B / C D E

Infix to Postfix

(X = (((A * B) - (C / D)) ↑ E))
X A B C D E
X A B * C D E
X A B * C D / E
X A B * C D / - E
X A B * C D / - E ↑
X A B * C D / - E ↑ =

Prefix to Infix

↑ + * 3 4 / 8 2 – 7 5
↑ + (3*4) / 8 2 – 7 5
↑ + (3*4) (8/2) – 7 5
↑ (3*4)+(8/2) - 7 5
↑ ((3*4)+(8/2)) (7-5)
(((3*4)+(8/2))↑(7-5))

Postfix to Infix

3 4 * 8 2 / + 7 5 -↑
(3*4) 8 2 / + 7 5 - ↑
(3*4) (8/2) + 7 5 - ↑
((3*4)+(8/2)) 7 5 -↑
((3*4)+(8/2)) (7-5) ↑
(((3*4)+(8/2))↑(7-5))

Bit-String Flicking

Bit strings (strings of binary digits) are frequently manipulated bit-by-bit using the logical operators **NOT**, **AND**, **OR**, and **XOR**. Bits strings are manipulated as a unit using **SHIFT** and **CIRCULATE** operators. The bits on the left are called the *most significant bits* and those on the right are the *least significant bits*.

Most high-level languages (e.g., Python, Java, C++), support bit-string operations. Programmers typically use bit strings to maintain a set of flags. Suppose that a program supports 8 options, each of which can be either "on" or "off". One could maintain this information using an array of size 8, or one could use a single variable (if it is internally stored using at least 8 bits or 1 byte, which is usually the case) and represent each option with a single bit. In addition to saving space, the program is often cleaner if a single variable is involved rather than an array. Bits strings are often used to maintain a set where values are either in the set or not. Shifting of bits is also used to multiply or divide by powers of 2.

Mastering this topic is essential for systems programming, programming in assembly language, optimizing code, and hardware design.

<Operators>

Bitwise Operators

The logical operators are **NOT** (~ or ¬), **AND** (&), **OR** (|), and **XOR** (⊕). These operators should be familiar to ACSL students from the Boolean Algebra and Digital Electronics categories.

- **NOT** is a unary operator that performs logical negation on each bit. Bits that are 0 become 1, and those that are 1 become 0. For example: ~101110 has a value of 010001.
- **AND** is a binary operator that performs the logical **AND** of each bit in each of its operands. The **AND** of two values is 1 only if both values are 1. For example, **1011011 and 0011001** has a value of **0011001**. The **AND** function is often used to isolate the value of a bit in a bit-string or to clear the value of a bit in a bit-string.
- **OR** is a binary operator that performs the logical **OR** of each bit in each of its operands. The **OR** of two values is 1 only if one or both values are 1. For example, **1011011 or 0011001** has a value of **1011011**. The **OR** function is often used to force the value of a bit in a bit-string to be 1, if it isn't already.
- **XOR** is a binary operator that performs the logical **XOR** of each bit in each of its operands. The **XOR** of two values is 1 if the values are different and 0 if they are the same. For example, **1011011 xor 011001 = 110010**. The **XOR** function is often used to change the value of a particular bit.

All binary operators (AND, OR, or XOR) must operate on bit-strings that are of the same length. If the operands are not the same length, the shorter one is padded with 0's on the left as needed. For example, **11010 and 1110** would have value of **11010 and 01110 = 01010**.

The following table summarizes the operators:

x	y	not x	x and y	x or y	x xor y
0	**0**	1	0	0	0
0	**1**	1	0	1	1
1	**0**	0	0	1	1
1	**1**	0	1	1	0

Shift Operators

Logical shifts (LSHIFT-x and RSHIFT-x) "ripple" the bit-string x positions in the indicated direction, either to the left or to the right. Bits shifted out are lost; zeros are shifted in at the other end.

Circulates (RCIRC-x and LCIRC-x) "ripple" the bit string x positions in the indicated direction. As each bit is shifted out one end, it is shifted in at the other end. The effect of this is that the bits remain in the same order on the other side of the string.

The size of a bit-string does not change with shifts, or circulates. If any bit strings are initially of different lengths, all shorter ones are padded with zeros in the left bits until all strings are of the same length.

The following table gives some examples of these operations:

x	(LSHIFT-2 x)	(RSHIFT-3 x)	(LCIRC-3 x)	(RCIRC-1 x)
01101	10100	00001	01011	10110
10	00	00	01	01
1110	1000	0001	0111	0111
1011011	1101100	0001011	1011101	1101101

Order of Precedence

The order of precedence (from highest to lowest) is: NOT; SHIFT and CIRC; AND; XOR; and finally, OR. In other words, all unary operators are performed on a single operator first. Operators with equal precedence are evaluated left to right; all unary operators bind from right to left.

<Example>

Evaluate the following expression:

(101110 AND NOT 110110 OR (LSHIFT-3 101010))

Solution: The expression evaluates as follows:

(101110 AND **001001** OR (LSHIFT-3 101010))
(**001000** OR (LSHIFT-3 101010))
(001000 OR **010000**)
011000

LISP

LISP is one of the simplest computer languages in terms of syntax and semantics, and also one of the most powerful. It was developed in the mid-1950s by John McCarthy at M.I.T. as a "List Processing language." It has been historically used for virtually all Artificial Intelligence programs and is often the environment of choice for applications that require a powerful interactive working environment. LISP presents a very different way to think about programming from the "algorithmic" languages, such as Python, C++, and Java.

<Syntax>

As its name implies, the basis of LISP is a list. One constructs a list by enumerating elements inside a pair of parentheses. For example, here is a list with four elements (the second element is also a list):

(23 (this is easy) hello 821)

The elements in the list, which are not lists, are called "atoms." For example, the atoms in the list above are 23, 'this, 'hello, 821, 'easy, and 'is. Literals are identified with a single leading quote. Everything in LISP is either an atom or a list (but not both). The only exception is "NIL," which is both an atom and a list. It can also be written as "()" – a pair of parentheses with nothing inside.

All statements in LISP are function calls with the following syntax: (function arg1 arg2 arg3 ... argn). To evaluate a LISP statement, each of the arguments (possibly functions themselves) are evaluated, and then the function is invoked with the arguments. For example, (MULT (ADD 2 3) (ADD 1 4 2)) has a value of 35, since (ADD 2 3) has a value of 5, (ADD 1 4 2) has a value of 7, and (MULT 5 7) has a value of 35. Some functions have an arbitrary number of arguments; others require a fixed number. All statements return a value, which is either an atom or a list.

<Basic Functions (SET, SETQ, EVAL, ATOM)>

We may assign values to variables using the function SET. For example, the statement (SET 'test 6) would have a value of a 6, and (more importantly, however) would also cause the atom 'test to be bound to the atom 6. The function SETQ is the same as SET, but it causes LISP to act as if the first argument was quoted. Observe the following examples:

Statement	Value	Comment
(SET 'a (MULT 2 3))	6	a is an atom with a value of 6
(SET 'a '(MULT 2 3))	(MULT 2 3)	a is a list with 3 elements
(SET 'b 'a)	a	b is an atom with a value of the character a
(SET 'c a)	(MULT 2 3)	c is a list with 3 elements
(SETQ EX (ADD 3 (MULT 2 5)))	13	The variable EX has a value of 13
(SETQ VOWELS '(A E I O U))	(A E I O U)	VOWELS is a list of 5 elements

The function EVAL returns the value of its argument after it has been evaluated. For example, (SETQ z '(ADD 2 3)) has a value of the list (ADD 2 3); the function (EVAL 'z) has a value of (ADD 2 3); the function (EVAL z) has a value of 5 (but the binding of the atom z has not changed). In this last example, you can think of z being "resolved" twice: once because it is an argument to a function and LISP evaluates all arguments to functions before the function is invoked, and once when the function EVAL is invoked to resolve arguments.

The function ATOM can be used to determine whether an item is an atom or a list. It returns either "true", or "NIL" for false. Consider the following examples:

Statement	Value	Comment
(SETQ p '(ADD 1 2 3 4))	(ADD 1 2 3 4)	p is a list with 5 elements
(ATOM 'p)	true	The argument to ATOM is the atom p
(ATOM p)	NIL	Because p is not quoted, it is evaluated to the 5-element list.
(EVAL p)	10	The argument to EVAL is the value of p; the value of p is 10.

<List Functions (CAR, CDR, CONS, REVERSE)>

The two most famous LISP functions are CAR and CDR (pronounced: could-er), named after registers of a now long-forgotten IBM machine on which LISP was first developed. The function (CAR x) returns the first item of the list x (and x must be a list or an error will occur); (CDR x) returns the list without its first element (again, x must be a list). The function CONS takes two arguments, of which the second must be a list. It returns a list which is composed by placing the first argument as the first element in the second argument's list. The function REVERSE returns a list which is its arguments in reverse order.

The CAR and CDR functions are used extensively to grab specific elements of a list or sublist, that there's a shorthand for this: (CADR x) is the same as (CAR (CDR x)), which retrieves the second element of the list x; (CAADDAR x) is a shorthand for (CAR (CAR (CDR (CDR (CAR x))))), and so on.

The following examples illustrate the use of CAR, CDR, CONS, and REVERSE:

Statement	Value
(CAR '(This is a list))	This
(CDR '(This is a list))	(is a list)
(CONS 'red '(white blue))	(red white blue)
(SETQ z (CONS '(red white blue) (CDR '(This is a list))))	((red white blue) is a list)
(REVERSE z)	(list a is (red white blue))
(CDDAR z)	(blue)

<Arithmetic Functions (CAR, CDR, CONS, REVERSE)>

As you have probably already figured out, the function ADD simply summed its arguments. We'll also be using the following arithmetic functions:

Function	Result
(ADD x1 x2 ...)	sum of all arguments
(SUB a b)	a-b
(MULT x1 x2 ...)	product of all arguments
(DIV a b)	a/b
(SQUARE a)	a*a
(EXP a n)	a^n
(EQ a b)	true if a and b are equal, NIL otherwise
(POS a)	true if a is positive, NIL otherwise
(NEG a)	true if a is negative, NIL otherwise

Functions ADD, SUB, MULT, and DIV can be written as their common mathematical symbols, +, -, *, and /. Here are some examples of these functions:

Statement	Value
(ADD (EXP 2 3) (SUB 4 1) (DIV 54 4))	24.5
(- (* 3 2) (- 12 (+ 1 2 1)))	-2
(ADD (SQUARE 3) (SQUARE 4))	25

<User-defined Functions>

LISP also allows us to create our functions using the DEF function. (We will sometimes use DEFUN rather than DEF, as it is a bit more standard terminology.) For example,

```
(DEF SECOND (args) (CAR (CDR args)))
```

defines a new function called SECOND which operates on a single parameter named "args". SECOND will take the CDR of the parameter and then the CAR of that result. So, for example: (SECOND '(a b c d e)) would first CDR the list to give (b c d e), and then CAR that values returning the single character "b". Consider the following program fragment:

```
(SETQ X '(a c s l))
(DEF WHAT(args) (CONS args (REVERSE (CDR args))))
(DEF SECOND(args) (CONS (CAR (CDR args)) NIL))
```

The following chart illustrates the use of the user-defined functions WHAT and SECOND:

Statement	Value
(WHAT X)	((a c s l) l s c)
(SECOND X)	(c)
(SECOND (WHAT X))	(l)
(WHAT (SECOND X))	((c))

<Outline Interpreters>

There are many online LISP interpreters available on the Internet. The one that ACSL uses for testing its programs is CLISP which is accessible from JDoodle. This interpreter is nice because it is quite peppy in running programs, and functions are not case-sensitive. So, `(CAR (CDR x))` is legal as is `(car (cdr x))` One drawback of this interpreter is the print function changes lowercase input into uppercase.

<Example>

Evaluate the following expression. `(MULT (ADD 6 5 0) (MULT 5 1 2 2) (DIV 6 (SUB 2 5)))`

Solution:

```
(MULT (ADD 6 5 0) (MULT 5 1 2 2) (DIV 6 (SUB 2 5)))
(MULT 11 20   (DIV 6 -3))
(MULT 11 20 -2)
-440
```

Boolean Algebra

Boolean algebra is the branch of algebra in which the values of the variables and constants have exactly two values: *true* and *false*, usually denoted 1 and 0 respectively.

<Operators>

The basic operators in Boolean algebra are **AND**, **OR**, and **NOT**. The secondary operators are *eXclusive OR* (often called **XOR**) and *eXclusive NOR* (**XNOR**, sometimes called **equivalence**). They are secondary in the sense that they can be composed from the basic operators.

The **AND** of two values is true only whenever both values are true. It is written as xy or $x \cdot y$. The values of *and* for all possible inputs is shown in the following truth table:

x	y	xy
0	**0**	0
0	**1**	0
1	**0**	0
1	**1**	1

The **OR** of two values is true whenever either or both values are true. It is written as x+y. The values of *or* for all possible inputs is shown in the following truth table:

x	y	$x+y$
0	**0**	0
0	**1**	1
1	**0**	1
1	**1**	1

The **NOT** of a value is its opposite; that is, the *not* of a true value is false whereas the *not* of a false value is true. It is written as x or ¬ x. The values of *not* for all possible inputs is shown in the following truth table:

x	$\overline{x}$
0	1
1	0

The **XOR** of two values is true whenever the values are different. It uses the ⊕ operator, and can be built from the basic operators: x⊕y=xy+xy The values of *xor* for all possible inputs is shown in the following truth table:

x	y	$x \oplus y$
0	**0**	0
0	**1**	1
1	**0**	1
1	**1**	0

The **XNOR** of two values is true whenever the values are the same. It is the **NOT** of the **XOR** function. It uses the ⊙ operator: x⊙y=x⊕y. The *xnor* can be built from basic operators: x⊙y=xy+xy. The values of *xnor* for all possible inputs is shown in the following truth table:

x	y	$x \odot y$
0	**0**	1
0	**1**	0
1	**0**	0
1	**1**	1

Just as algebra has basic rules for simplifying and evaluating expressions, so does Boolean algebra.

<Importance of Boolean Algebra>

Boolean algebra is important to programmers, computer scientists, and the general population.

- For programmers, Boolean expressions are used for conditionals and loops. For example, the following snippet of code sums the even numbers that are not also multiples of 3, stopping when the sum hits 100:

```
s = 0
x = 1
while (s < 100):
    if (x % 2 == 0) and (x % 3 != 0)
        then s = s + x
    x = x + 1
```

Both the conditional statement `s < 100` and the Boolean expression with 2 conditional statements `(x % 2 == 0) and (x % 3 != 0)` evaluate to *true* or *false*.

- For computer scientists, Boolean algebra is the basis for digital circuits that make up a computer's hardware. The Digital Electronics category concerns a graphical representation of a circuit. That circuit is typically easiest to understand and evaluate by converting it to its Boolean algebra representation.

- The general population uses Boolean algebra, probably without knowing that they are doing so, when they enter search terms in Internet search engines. For example, the search expression *jaguar speed -car* is parsed by the search engine as the Boolean expression `"jaguar" and "car" and not "speed"`; it returns pages about the speed of the jaguar animal, not the Jaguar car.

<Laws>

A **law** of Boolean algebra is an identity such as x+(y+z)=(x+y)+z between two Boolean terms, where a **Boolean term** is defined as an expression built up from variables, the constants 0 and 1, and operations *and*, *or*, *not*, *xor*, and *xnor*.
Like ordinary algebra, parentheses are used to group terms. When a *not* is represented with an overhead horizontal line, there is an implicit grouping of the terms under the line. That is, x·y+z is evaluated as if it were written x·(~~y+z~~).

Order of Precedence

The order of operator precedence is *not*, then *and*, then *xor* and *xnor*, and finally *or*. Operators with the same level of precedence are evaluated from left-to-right.

Fundamental Identities

Commutative Law – The order of application of two separate terms is not important.	$x + y = y + x$	$x \cdot y = y \cdot x$
Associative Law – Regrouping of the terms in an expression doesn't change the value of the expression.	$(x + y) + z = x + (y + z)$	$x \cdot (y \cdot z) = (x \cdot y) \cdot z$
Idempotent Law – A term that is *or*'ed or *and*'ed with itself is equal to that term.	$x + x = x$	$x \cdot x = x$
Annihilator Law – A term that is *or*'ed with 1 is 1; a term *and*'ed with 0 is 0.	$x + 1 = 1$	$x \cdot 0 = 0$
Identity Law – A term *or*'ed 0 or *and*'ed with a 1 will always equal that term.	$x + 0 = x$	$x \cdot 1 = x$
Complement Law – A term *or*'ed with its complement equals 1 and a term *and*'ed with its complement equals 0.	$x + \overline{x} = 1$	$x \cdot \overline{x} = 0$
Absorptive Law – Complex expressions can be reduced to a simpler ones by absorbing like terms.	$x + xy = x$ $x + \overline{x}y = x + y$ $x(x + y) = x$	
Distributive Law – It's OK to multiply or factor-out an expression.	$x \cdot (y + z) = xy + xz$ $(x + y) \cdot (p + q) = xp + xq + yp + yq$ $(x + y)(x + z) = x + yz$	
DeMorgan's Law – An *or* (*and*) expression that is negated is equal to the *and* (*or*) of the negation of each term.	$\overline{x + y} = \overline{x} \cdot \overline{y}$	$\overline{x \cdot y} = \overline{x} + \overline{y}$
Double Negation – A term that is inverted twice is equal to the original term.	$\overline{\overline{x}} = x$	
Relationship between XOR and XNOR	$x \odot y = \overline{x \oplus y} = x \oplus \overline{y} = \overline{x} \oplus y$	

<Example>

Problems in this category are typically of the form "Given a Boolean expression, simplify it as much as possible" or "Given a Boolean expression, find the values of all possible inputs that make the expression *true*." Simplify means writing an equivalent expression using the fewest number of operators.

Problem: Simplify the following expression as much as possible: $\overline{\overline{A(A+B)}+B\overline{A}}$

Solution:

The simplification proceeds as follows:

$$\begin{aligned}
&\overline{\overline{A(A+B)}+B\overline{A}} \\
&= \left(\overline{\overline{A(A+B)}}\right)\cdot\left(\overline{B\overline{A}}\right) && \text{(DeMorgan's Law)} \\
&= (A(A+B))\cdot\left(\overline{B}+\overline{\overline{A}}\right) && \text{(Double Negation; DeMorgan's Law)} \\
&= A\cdot\left(\overline{B}+A\right) && \text{(Absorption; Double Negation)} \\
&= A && \text{(Absorption)}
\end{aligned}$$

Data Structures

At the heart of virtually every computer program are its algorithms and its data structures. It is hard to separate these two items, for data structures are meaningless without algorithms to create and manipulate them, and algorithms are usually trivial unless there are data structures on which to operate. The bigger the data sets, the more important data structures are in various algorithms.

This category concentrates on four of the most basic structures: **stacks**, **queues**, **binary search trees**, and **priority queues**. Questions will cover these data structures and implicit algorithms, not specific to implementation language details.

A *stack* is usually used to save information that will need to be processed later. Items are processed in a "last-in, first-out" (LIFO) order. A *queue* is usually used to process items in the order in which requests are generated; a new item is not processed until all items currently on the queue are processed. This is also known as "first-in, first-out" (FIFO) order. A *binary search tree* is used when one is storing items and needs to be able to efficiently process the operations of insertion, deletion, and query (i.e. find out if a particular item is found in the list of items and if not, which item is close to the item in question). A *priority queue* is used like a binary search tree, except one cannot delete an arbitrary item, nor can one make an arbitrary query. One can only find out or delete the smallest element of the list.

There are many online resources covering these basic data structures; indeed there are many books and entire courses devoted to fundamental data structures. Implementation varies by computer language, but for our purposes, they can all be represented as a list of items that might contain duplicates. The rest of this page is an overview of these structures.

<Stacks and Queues>

A *stack* supports two operations: PUSH and POP. A command of the form PUSH("A") puts the key "A" at the top of the stack; the command "X = POP()" removes the top item from the stack and stores its value into variable X. If the stack was empty (because nothing had ever been pushed on it, or if all elements have been popped off of it), then X is given the special value of NIL. An analogy to this is a stack of books on a desk: a new book is placed on the top of the stack (pushed) and a book is removed from the top also (popped). Some textbooks call this data structure a "push-down stack" or a "LIFO stack".

Queues operate just like stacks, except items are removed from the bottom instead of the top. A command of the form PUSH("A") puts the key "A" at the top of the queue; the command "X = POP()" removes the item from the bottom of the queue and stores its value into variable X. If the queue was empty (because nothing had ever been pushed on it, or if all elements have been popped off of it), then X is given the special value of NIL. A good physical analogy of this is the way a train conductor or newspaper boy uses a coin machine to give change: new coins are added to the tops of the piles, and change is given from the bottom of each. Sometimes the top and bottom of a queue are referred to as the rear and the front respectively. Therefore, items are pushed/enqueued at the rear of the queue and popped/dequeued at the front of the queue. There is a similarity to the Britsh "queueing up". Some textbooks refer to this data structure as a "FIFO stack".

Consider the following sequence of 14 operations:

```
PUSH("A")
PUSH("M")
PUSH("E")
X = POP()
PUSH("R")
X = POP()
PUSH("I")
X = POP()
X = POP()
X = POP()
X = POP()
PUSH("C")
PUSH("A")
PUSH("N")
```

If these operations are applied to a stack, then the values of the pops are: E, R, I, M, A and NIL. After all of the operations, there are three items still on the stack: the N is at the top (it will be the next to be popped, if nothing else is pushed before the pop command), and C is at the bottom. If, instead of using a stack we used a queue, then the values popped would be: A, M, E, R, I and NIL. There would be three items still on the queue: N at the top and C on the bottom. Since items are removed from the bottom of a queue, C would be the next item to be popped regardless of any additional pushes.

Pre-2018 Syntax

ACSL Contests pre-2018 used a slightly different, and more liberal, syntax. Consider the following sequence on a stack:

```
PUSH(A)
PUSH(B)
PUSH(C)
POP(X)
POP(Z)
```

After the 5 operations, the stack would be left with A as the only element on the stack, the value of C in variable X, and the value of B in variable Z.

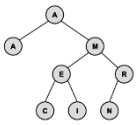
Were the above operations applied to a queue, the queue would be left with C; variable X would contain A; and variable Z would contain B.

<Trees>

Trees, in general, use the following terminology: the *root* is the top node in the tree; *children* are the nodes that are immediately below a *parent* node; *leaves* are the bottom-most nodes on every branch of the tree; and *siblings* are nodes that have the same immediate parent.

A *binary search tree* is composed of nodes having three parts: information (or a key), a pointer to a left child, and a pointer to a right child. It has the property that the key at every node is always greater than or equal to the key of its left child, and less than the key of its right child.

The following tree is built from the keys A, M, E, R, I, C, A, N in that order:



The *root* of the resulting tree is the node containing the key A. Our ACSL convention places duplicate keys into the tree as if they were less than their equal key. (In some textbooks and software libraries, duplicate keys may be considered larger than their equal key.) The tree has a *depth* (sometimes called height) of 3 because the deepest node is 3 nodes below the root. The root node has a depth of 0. Nodes with no children are called leaf nodes; there are four of them in the tree: A, C, I and N. Our ACSL convention is that an *external node* is the name given to a place where a new node could be attached to the tree. (In some textbooks, an external node is synonymous with a leaf node.) In the final tree above, there are 9 external nodes; these are not drawn. The tree has an *internal path length* of 15 which is the sum of the depths of all nodes. It has an *external path length* of 31 which is the sum of the depths of all external nodes. To insert the N (the last key inserted), 3 *comparisons* were needed against the root A (>), the M (>), and the R (≤).

To perform an *inorder* traversal of the tree, recursively traverse the tree by first visiting the left child, then the root, then the right child. In the tree above, the nodes are visited in the following order: A, A, C, E, I, M, N, and R. A *preorder* travel (root, left, right) visits in the following order: A, A, M, E, C, I, R, and N. A *postorder* traversal (left, right, root) is: A, C, I, E, N, R, M, A. Inorder traversals are typically used to list the contents of the tree in sorted order.

ACSL Theoretical Problem Conceptual Description

A binary search tree can support the operations insert, delete, and search. Moreover, it handles the operations efficiently for *balanced* trees. In a tree with 1 million items, one can search for a particular value in about $\log_2 1{,}000{,}000 \approx 20$ steps. Items can be inserted or deleted in about as many steps, too. However, consider the binary search tree resulting from inserting the keys A, E, I, O, U, Y which places all of the other letters on the right side of the root "A". This is very unbalanced; therefore, sophisticated algorithms can be used to maintain balanced trees. Binary search trees are "dynamic" data structures that can support many operations in any order and introduces or removes nodes as needed.

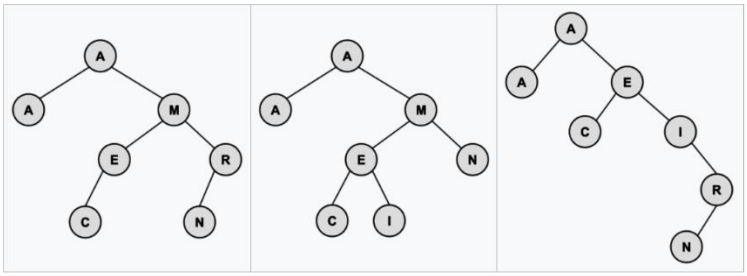
To search for a node in a binary tree, the following algorithm (in pseudo-code) is used:

```
p = root
found = FALSE
while (p ≠ NIL) and (not found)
  if (x < p's key)
    p = p's left child
  else if (x > p's key)
    p = p's right child
  else
    found = TRUE
  end if
end while
```

Deleting from a binary search tree is a bit more complicated. The algorithm we'll use is as follows:

```
p = node to delete
f = father of p
if (p has no children)
  delete p
else if (p has one child)
  make p's child become f's child
  delete p
else if (p has two children)
  l = p's left child (it might also have children)
  r = p's right child (it might also have children)
  make l become f's child instead of p
  stick r onto the l tree
  delete p
end if
```

These diagrams illustrate the algorithm using the tree above. At the left, we delete I (0 children); in the middle, we delete the R (1 child); and at the right, we delete the M (2 children).



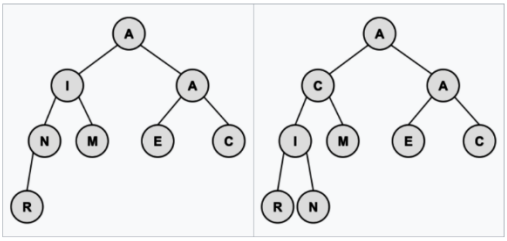
There are also general trees that use the same terminology, but they have 0 or more *subnodes* which can be accessed with an array or linked list of pointers. *Pre-order* and *post-order* traversals are possible with these trees, but the other algorithms do not work in the same way. Applications of general trees include game theory, organizational charts, family trees, etc.

Balanced trees minimize searching time when every leaf node has a depth of within 1 of every other leaf node. *Complete* trees are filled in at every level and are always balanced. *Strictly binary* trees ensure that every node has either 0 or 2 subnodes. You may want to consider how there are exactly 5 strictly binary trees with 7 nodes.

<Priority Queues>

A *priority queue* is quite similar to a binary search tree, but one can only delete the smallest item and retrieve the smallest item only. These insert and delete operations can be done in a guaranteed time proportional to the log (base 2) of the number of items; the retrieve-the-smallest can be done in constant time.

The standard way to implement a priority queue is using a *heap* data structure. A heap uses a binary tree (that is, a tree with two children) and maintains the following two properties: every node is less than or equal to both of its two children (nothing is said about the relative magnitude of the two children), and the resulting tree contains no "holes". That is, all levels of the tree are completely filled, except the bottom level, which is filled in from the left to the right.



The algorithm for insertion is not too difficult: put the new node at the bottom of the tree and then go up the tree, making exchanges with its parent, until the tree is valid. The heap at the left was building from the letters A, M, E, R, I, C, A, N (in that order); the heap at the right is after a C has been added.

The smallest value is always the root. To delete it (and one can only delete the smallest value), one replaces it with the bottom-most and right-most element, and then walks down the tree making exchanges with the smaller child in order to ensure that the tree is valid. The following pseudo-code formalizes this notion:

```
b = bottom-most and right-most element
p = root of tree
p's key = b's key
delete b
while (p is larger than either child)
  exchange p with smaller child
  p = smaller child
end while
```

When the smallest item is at the root of the heap, the heap is called a *min-heap*. Of course, A *max-heap* is also possible and is common in practice. An efficient implementation of a heap uses an array that can be understood conceptually by using a tree data structure.

<Sample>

Consider an initially empty stack. After the following operations are performed, what is the value of Z?

```
PUSH(3)
PUSH(6)
PUSH(8)
Y = POP()
X = POP()
PUSH(X-Y)
Z = POP()
```

Solution: The first POP stores 8 in Y. The second POP stores 6 in X. Then, 6-8=-2 is pushed onto the stack. Finally, the last POP removes the -2 and stores it in Z.

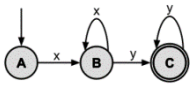
FSAs and Regular Expressions

A Finite State Automaton (FSA) is a mathematical model of computation comprising all 4 of the following: 1) a finite number of *states*, of which exactly one is *active* at any given time; 2) *transition rules* to change the active state; 3) an *initial state*; and 4) one or more *final states*. We can draw an FSA by representing each state as a circle, the final state as a double circle, the start state as the only state with an incoming arrow, and the transition rules as labeled-edges connecting the states. When labels are assigned to states, they appear inside the circle representing the state.

In this category, FSAs will be limited to parsing strings. That is, determining if a string is valid or not.

<Basics>

Here is a drawing of an FSA that is used to parse strings consisting of x's and y's:



In the above FSA, there are three states: A, B, and C. The initial state is A; the final state is C. The only way to go from state A to B is by *seeing* the letter x. Once in state B, there are two transition rules: seeing the letter y will cause the FSA to make C the active state, and seeing an x will keep B as the active state. State C is a final state so if the string being parsed is completed and the FSA is in State C, the input string is said to be *accepted* by the FSA. In State C, seeing any additional letter y will keep the machine in state C. The FSA above will accept strings composed of one or more x's followed by one or more y's (e.g., xy, xxy, xxxyy, xyyy, xxyyyy).

A Regular Expression (RE) is an algebraic representation of an FSA. For example, the regular expression corresponding to the first FSA given above is xx*yy*.

The rules for forming a Regular Expression (RE) are as follows:

1. The null string (λ) is a RE.
2. If the string a is in the input alphabet, then it is a RE.
3. if a and b are both REs, then so are the strings built up using the following rules:

a. CONCATENATION. "ab" (a followed by b).
b. UNION. "aUb" or "a|b" (a or b).
c. CLOSURE. "a*" (a repeated zero or more times). This is known as the Kleene Star.

The order of precedence for Regular Expression operators is: Kleene Star, concatenation, and then union. Similar to standard Algebra, parentheses can be used to group sub-expressions. For example, "dca*b" generates strings dcb, dcab, dcaab, and so on, whereas "d(ca)*b" generates strings db, dcab, dcacab, dcacacab, and so on.

If we have a Regular Expression, then we can mechanically build an FSA to accept the strings which are generated by the Regular Expression. Conversely, if we have an FSA, we can mechanically develop a Regular Expression which will describe the strings which can be parsed by the FSA. For a given FSA or Regular Expression, there are many others which are equivalent to it. A "most simplified" Regular Expression or FSA is not always well defined.

<Regular Expression Identities>

1. (a*)* = a*
2. aa* = a*a
3. aa* U λ = a*
4. a(b U c) = ab U ac
5. a(ba)* = (ab)*a
6. (a U b)* = (a* U b*)*
7. (a U b)* = (a*b*)*
8. (a U b)* = a*(ba*)*

<RegEX in Practice>

Programmers use Regular Expressions (usually referred to as **regex**) extensively for expressing patterns to search for. All modern programming languages have regular expression libraries.

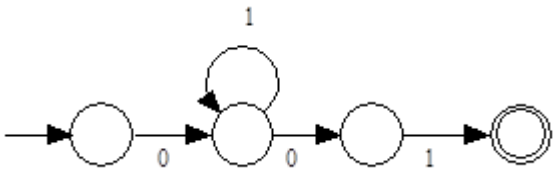
Unfortunately, the specific syntax rules vary depending on the specific implementation, programming language, or library in use. Interactive websites for testing regexes are a useful resource for learning regexes by experimentation. An excellent online tool is https://regex101.com/. A very nice exposition is Pattern Matching with Regular Expressions from the Automate the Boring Stuff book and online course.

Here are the additional syntax rules that we will use. They are pretty universal across all regex packages.

Pattern	Description
|	As described above, a vertical bar separates alternatives. For example, gray|grey can match "gray" or "grey".
*	As described above, the asterisk indicates zero or more occurrences of the preceding element. For example, ab*c matches "ac", "abc", "abbc", "abbbc", and so on.
?	The question mark indicates zero or one occurrences of the preceding element. For example, colou?r matches both "color" and "colour".
+	The plus sign indicates one or more occurrences of the preceding element. For example, ab+c matches "abc", "abbc", "abbbc", and so on, but not "ac".
.	The wildcard . matches any character. For example, a.b matches any string that contains an "a", then any other character, and then a "b" such as "a7b", "a&b", or "arb", but not "abbb". Therefore, a.*b matches any string that contains an "a" and a "b" with 0 or more characters in between. This includes "ab", "acb", or "a123456789b".
[]	A bracket expression matches a single character that is contained within the brackets. For example, [abc] matches "a", "b", or "c". [a-z] specifies a range which matches any lowercase letter from "a" to "z". These forms can be mixed: [abcx-z] matches "a", "b", "c", "x", "y", or "z", as does [a-cx-z].
[^]	Matches a single character that is not contained within the brackets. For example, [^abc] matches any character other than "a", "b", or "c". [^a-z] matches any single character that is not a lowercase letter from "a" to "z". Likewise, literal characters and ranges can be mixed.
()	As described above, parentheses define a sub-expression. For example, the pattern H(ä|ae?)ndel matches "Handel", "Händel", and "Haendel".

<Sample>

1. Find a simplified Regular Expression for the following FSA:



Solution:

The expression 01*01 is read directly from the FSA. It is in its most simplified form.

2. Which of the following strings are accepted by the following Regular Expression "00*1*1U11*0*0" ?

A. 0000001111111
B. 1010101010
C. 1111111
D. 0110
E. 10

Solution:

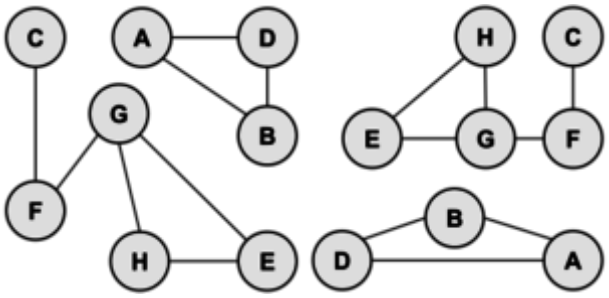
This Regular Expression parses strings described by the union of 00*1*1 and 11*0*0. The RE 00*1*1 matches strings starting with one or more 0s followed by one or more 1s: 01, 001, 0001111, and so on. The RE 11*0*0 matches strings with one or more 1s followed by one or more 0s: 10, 1110, 1111100, and so on. In other words, strings of the form: 0s followed by some 1s; or 1s followed by some 0s. Choice A and E following this pattern.

Graph Theory

Many problems are naturally formulated in terms of points and connections between them. For example, a computer network has PCs connected by cables, an airline map has cities connected by routes, and a school has rooms connected by hallways. A graph is a mathematical object which models such situations.

<Overview>

A *graph* is a collection of vertices and edges. An *edge* is a connection between two *vertices* (sometimes referred to as *nodes*). One can draw a graph by marking points for the vertices and drawing lines connecting them for the edges, but the graph is defined independently of the visual representation. For example, the following two drawings represent the same graph:



The precise way to represent this graph is to identify its set of vertices {A, B, C, D, E, F, G}, and its set of edges between these vertices {AB, AD, BD, CF, FG, GH, GE, HE}.

<Terminology>

The edges of the above graph have no directions meaning that the edge from one vertex A to another vertex B is the same as from vertex B to vertex A. Such a graph is called an *undirected graph*. Similarly, a graph having a direction associated with each edge is known as a *directed graph*.

A *path* from vertex x to y in a graph is a list of vertices, in which successive vertices are connected by edges in the graph. For example, FGHE is path from F to E in the graph above. A *simple path* is a path with no vertex repeated. For example, FGHEG is not a simple path.

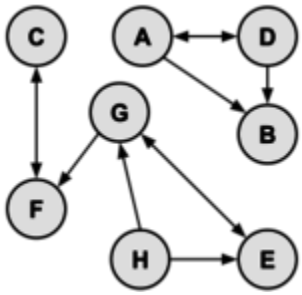
A graph is *connected* if there is a path from every vertex to every other vertex in the graph. Intuitively, if the vertices were physical objects and the edges were strings connecting them, a connected graph would stay in one piece if picked up by any vertex. A graph which is not connected is made up of *connected components*. For example, the graph above has two connected components: {A, B, D} and {C, E, F, G, H}.

A *cycle* is a path, which is simple except that the first and last vertex are the same (a path from a point back to itself). For example, the path HEGH is a cycle in our example. Vertices must be listed in the order that they are traveled to make the path; any of the vertices may be listed first. Thus, HEGH and EHGE are different ways to identify the same cycle. For clarity, we list the start / end vertex twice: once at the start of the cycle and once at the end.

We'll denote the number of vertices in a given graph by V and the number of edges by E. Note that E can range anywhere from V to V2 (or V(V−1)/2 in an undirected graph). Graphs with all edges present are called *complete graphs*; graphs with relatively few edges present (say less than Vlog(V)) are called *sparse*; graphs with relatively few edges missing are called *dense*.

<Directed Graphs>

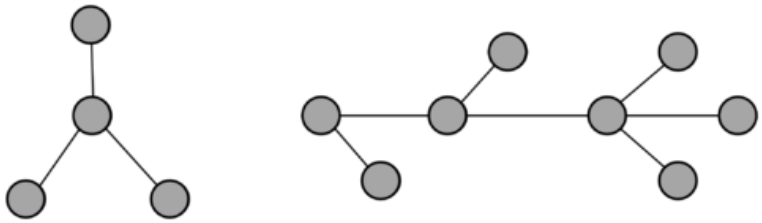
Directed graphs are graphs which have a direction associated with each edge. An edge xy in a directed graph can be used in a path that goes from x to y but not necessarily from y to x. For example, a directed graph similar to our example graph is drawn below:



This graph is defined as the set of vertices V = {A,B,C,D,E,F,G,H} and the set of edges {AB,AD,DA,DB,EG,GE,HG,HE,GF,CF,FC}. There is one directed path from G to C (namely, GFC); however, there are no directed paths from C to G. Note that a few of the edges have arrows on both ends, such as the edge between A and D. These dual arrows indicate that there is an edge in each direction, which essentially makes an undirected edge.
An *undirected graph* can be thought of as a directed graph with all edges occurring in pairs in this way. A directed graph with no cycles is called a *dag* (directed acyclic graph).

<Tree & Forests>

A graph with no cycles is called a *tree*. There is only one path between any two nodes in a tree. A tree with N vertices contains exactly N−1 edges.



The two graphs shown above are trees because neither has any cycles and all vertices are connected. The graph on the left has 4 vertices and 3 edges; the graph on the right has 8 vertices and 7 edges. Note that in both cases, because they are trees, the number of edges is one less than the number of vertices.

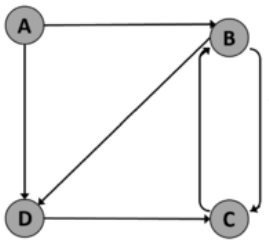
A group of disconnected trees is called a *forest*.

A *weighted graph* is a graph that has a weight (also referred to as a cost) associated with each edge. For example, in a graph used by airlines where cities are vertices and edges are cities with direct flights connecting them, the weight for each edge might be the distance between the cities.

A *spanning tree* of a graph is a subgraph that contains all the vertices and forms a tree. A *minimal spanning tree* can be found for weighted graphs in order to minimize the cost across an entire network.

<Adjacency Matrices>

It is frequently convenient to represent a graph by a matrix known as an *adjacency matrix.* Consider the following directed graph:



To draw the adjacency matrix, we create an N by N grid and label the rows and columns for each vertex (diagram at the left). Then, place a 1 for each edge in the cell whose row and column correspond to the starting and ending vertices of the edge (diagram in the middle). Finally, place a 0 in all other cells (diagram at the right).

M =

	A	B	C	D
A				
B				
C				
D				

=

	A	B	C	D
A		1		1
B			1	1
C		1		
D			1	

=

	A	B	C	D
A	0	1	0	1
B	0	0	1	1
C	0	1	0	0
D	0	0	1	0

By construction, cell (i,j) in the matrix with a value of 1 indicates a direct path from vertex i to vertex j. If we square the matrix, the value in cell (i,j) indicates the number of paths of length 2 from vertex i to vertex j.

M^2 =

	A	B	C	D
A	0	0	2	1
B	0	1	1	0
C	0	0	1	1
D	0	1	0	0

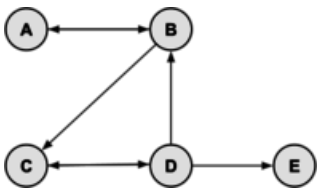
For example, the M2 says that there 2 paths of length 2 from A to C (A → B → C and A → D → C). This also says that there is exactly 1 path of length 2 from A to D (A → B → D), exactly 1 path of length 2 from B to B (B → C → B), and so on.

In general, if we raise M to the *pth* power, the resulting matrix indicates which paths of length p exist in the graph. The value in $M^p(i,j)$ is the number of paths from vertex i to vertex j.

<Sample>

Find the number of different cycles contained in the directed graph with vertices {A, B, C, D, E} and edges {AB, BA, BC, CD, DC, DB, DE}.

Solution: The graph is as follows:

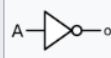
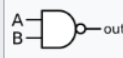
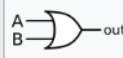
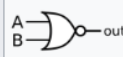


By inspection, the cycles are: ABA, BCDB, and CDC. Thus, there are 3 cycles in the graph.

Digital Electronics

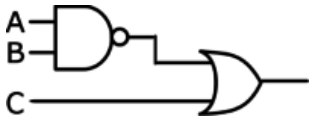
A digital circuit is constructed from logic gates. Each logic gate performs a function of boolean logic based on its inputs, such as AND or OR. Each circuit can be represented as a Boolean Algebra expression; this topic is an extension of the topic of Boolean Algebra, which includes a thorough description of truth tables and simplifying expressions.

<Definitions>

NAME	GRAPHICAL SYMBOL	ALGEBRAIC EXPRESSION	TRUTH TABLE
BUFFER		X = A	INPUT: A; OUTPUT: X 0 0 1 1
NOT		$X = \overline{A}$ or $\neg A$	INPUT: A; OUTPUT: X 0 1 1 0
AND		$X = AB$ or $A \cdot B$	INPUT: A B; OUTPUT: X 0 0 0 0 1 0 1 0 0 1 1 1
NAND		$X = \overline{AB}$ or $\overline{A \cdot B}$	INPUT: A B; OUTPUT: X 0 0 1 0 1 1 1 0 1 1 1 0
OR		$X = A + B$	INPUT: A B; OUTPUT: X 0 0 0 0 1 1 1 0 1 1 1 1
NOR		$X = \overline{A + B}$	INPUT: A B; OUTPUT: X 0 0 1 0 1 0 1 0 0 1 1 0
XOR		$X = A \oplus B$	INPUT: A B; OUTPUT: X 0 0 0 0 1 1 1 0 1 1 1 0
XNOR		$X = \overline{A \oplus B}$ or $A \odot B$	INPUT: A B; OUTPUT: X 0 0 1 0 1 0 1 0 0 1 1 1

<Sample>

Find all ordered triplets (A, B, C) which make the following circuit FALSE:



Solution:

One approach to solving this problem is to reason about that inputs and outputs are necessary at each gate. For the circuit to be FALSE, both inputs to the file OR gate must be false. Thus, input C must be FALSE, and the output of the NAND gate must also be false. The NAND gate is false only when both of its inputs are TRUE; thus, inputs A and B must both be TRUE. The final answer is (TRUE, TRUE, FALSE), or (1, 1, 0).

Another approach to solving this problem is to translate the circuit into a Boolean Algebra expression and simplify the expression using the laws of Boolean Algebra. This circuit translates to the Boolean expression $AB^{\overline{\quad\quad\quad}}+C$. To find when this is FALSE we can equivalently find when the $AB^{\overline{\quad\quad\quad}}+C^{\overline{\quad\quad\quad\quad\quad\quad\quad}}$ is TRUE. The expression becomes $AB^{\overline{\quad\quad\quad\quad\quad\quad\quad}} \cdot C^{\overline{\quad\quad\quad}}$ after applying DeMorgan's Law. The double NOT over the AB expression cancels out, to become $ABC^{\overline{\quad\quad\quad}}$. The AND of 3 terms is TRUE when each term is TRUE, or A=1, B=1 and C=0.

Assembly Language Programming

Programs written in high-level languages are traditionally converted by compilers into assembly language, which is turned into machine language programs – sequences of 1's and 0's – by an assembler. Even today, with very good quality compilers available, there is the need for programmers to understand assembly language. First, it provides programmers with a better understanding of the compiler and its constraints. Second, on occasion, programmers find themselves needing to program directly in assembly language in order to meet constraints in execution speed or space.

ACSL chose to define its own assembly language rather than use a "real" one in order to eliminate the many sticky details associated with real languages. The basic concepts of our ACSL topic description are common to all assembly languages.

<Reference Manual>

Execution starts at the first line of the program and continues sequentially, except for branch instructions (BG, BE, BL, BU), until the end instruction (END) is encountered. The result of each operation is stored in a special word of memory, called the "accumulator" (ACC). The initial value of the ACC is 0. Each line of an assembly language program has the following fields:

```
LABEL OPCODE LOC
```

The LABEL field, if present, is an alphanumeric character string beginning in the first column. A label must begin with an alphabetic character(A through Z, or a through z), and labels are case-sensitive. Valid OPCODEs are listed in the chart below; they are also case-sensitive and uppercase. Opcodes are reserved words of the language and (the uppercase version) many not be used a label. The LOC field is either a reference to a label or *immediate data*. For example, "LOAD A" would put the contents referenced by the label "A" into the ACC; "LOAD =123" would store the value 123 in the ACC. Only those instructions that do not modify the LOC field can use the "immediate data" format. In the following chart, they are indicated by an asterisk in the first column.

OP CODE	DESCRIPTION
*LOAD	The contents of LOC are placed in the ACC. LOC is unchanged.
STORE	The contents of the ACC are placed in the LOC. ACC is unchanged.
*ADD	The contents of LOC are added to the contents of the ACC. The sum is stored in the ACC. LOC is unchanged. Addition is modulo 1,000,000.
*SUB	The contents of LOC are subtracted from the contents of the ACC. The difference is stored in the ACC. LOC is unchanged. Subtraction is modulo 1,000,000.
*MULT	The contents of LOC are multiplied by the contents of the ACC. The product is stored in the ACC. LOC is unchanged. Multiplication is modulo 1,000,000.
*DIV	The contents of LOC are divided into the contents of the ACC. The signed integer part of the quotient is stored in the ACC. LOC is unchanged. .
BG	Branch to the instruction labeled with LOC if ACC>0.
BE	Branch to the instruction labeled with LOC if ACC=0.
BL	Branch to the instruction labeled with LOC if ACC<0.
BU	Branch to the instruction labeled with LOC.
READ	Read a signed integer (modulo 1,000,000) into LOC.
PRINT	Print the contents of LOC.
DC	The value of the memory word defined by the LABEL field is defined to contain the specified constant. The LABEL field is mandatory for this opcode. The ACC is not modified.
END	Program terminates. LOC field is ignored and must be empty.

<Sample>

1. After the following program is executed, what value is in location TEMP?

TEMP	DC	0
A	DC	8
B	DC	-2
C	DC	3
	LOAD	B
	MULT	C
	ADD	A
	DIV	B
	SUB	A
	STORE	TEMP
	END	

Solution: The ACC takes on values -2, -6, 2, -1, and -9 in that order. The last value, -9, is stored in location TEMP.

2. If the following program has an input value of N, what is the final value of X which is computed? Express X as an algebraic expression in terms of N.

	READ	X
	LOAD	X
TOP	SUB	=1
	BE	DONE
	STORE	A
	MULT	X
	STORE	X
	LOAD	A
	BU	TOP
DONE	END	

Solution: This program loops between labels TOP and DONE for A times. A has an initial value of X and subsequent terms of N, then values of A-1, A-2, ..., 1. Each time through the loop, X is multiplied by the the current value of A. Thus, X = A * (A-1) * (A-2) * ... * 1 or X=A! or A factorial. For example, 5! = 5 * 4 * 3 * 2 * 1 = 120. Since the initial value of A is the number input (i.e. N), the algebraic expression is X = N!.